Arithmetisation of computation via polynomial semantics for first-order logic

Murdoch J. Gabbay

Abstract. We propose a compositional shallow translation from a first-order logic with equality, into polynomials; that is, we arithmetise the semantics of first-order logic. Using this, we can translate specifications of mathematically structured programming into polynomials, in a form amenable to succinct cryptographic verification. We give worked example applications, and we propose a proof-of-concept succinct verification scheme based on inner product arguments.
First-order logic is widely used, because it is simple, highly expressive, and has excellent mathematical properties. Thus a compositional shallow embedding into polynomials suggests a simple, high-level, yet potentially very practical new method for expressing verifiable computation.

Keywords: First-order logic · cryptology · verifiable computation · verifiable logic · zero-knowledge proofs · polynomial semantics

Contents

1 Introduction

We take a step towards unifying mathematically structured programming (by which we mean '*computation that is specified using logic*') with cryptographically verifiable computation (meaning '*executions that can be cryptographically verified*'). We do this by giving

1. a *compositional, shallow translation* of first-order logic to polynomials (Figure 2),
2. a sound and complete translation of logical validity (i.e. predicates being true) to corresponding polynomial equalities (Theorems 2.4.4 and 4.2.7), and
3. a schema for *succinct proofs* (in the cryptographic sense) of the resulting polynomial equalities.

Schemes with this effect exist for von Neumann style abstract machines (a brief but clear survey is included here), logic circuits [DFGK14, LCL+24], and even LISP [LUR] (reference list not exhaustive).[1] Yet first-order logic has as good a claim as any to be a universal language of mathematics — and in particular it is a commonly and successfully used as a high-level language for specifying computation. To our knowledge, a compositional semantics for FOL has not yet been given in polynomials.

We summarise what this means and why it matters. This summary is aimed not only at cryptographers but also at readers from a logic and computation background, to provide some context:

Boolean circuits contrasted with arithmetic circuits

Logic uses Boolean circuits: combinations of *and*, *or*, and *not* operating on variables that each hold one bit of information: *true* or *false*. Because *and* and *or* distribute over one another, and because *not* satisfies de Morgan dualities, Boolean circuits have a normal form called a *disjunctive normal form*.

Cryptography uses arithmetic circuits: combinations of $+$, $*$, and $x \in \mathbb{Z}$ operating on variables that hold numbers (either from a finite field or possibly just integers). Because $+$ distributes over $*$, arithmetic circuits also have a normal form: *polynomials*, as familiar e.g. from maths in school.

Polynomials are very expressive: they can represent functions (through interpolation [SW22, Chapter 9][2]), vectors of integers (by evaluating at $\{1, \ldots, n\}$ where n is the length of the vector), multisets (by identification with their multiset of roots), and they can be composed. Polynomials over integers can be evaluated to numbers, and this has unexpected implications including the Schwartz-Zippel lemma (permalink) [Tha22, Lemma 3.3, Subsection 3.4].

[1] . . . and it would be impossible to even try, because progress in verifiable computation is currently so rapid.
[2] See also this compact but clear overview.

Propositions in disjunctive normal form are not quite as expressive as polynomials, but they have excellent properties: distributivity of *and* over *or* and of *or* over *and*; de Morgan duality between *true* and *false*; and bitwise number representation which are the basis of computer chips;[3] and an extremely well-developed theory which includes logic (including first-, higher-order, and modal logics), declarative programming languages, type-checkers, model-checkers, SAT solvers, formal verification tools, and moving on from there to dependent type theories, modal logics (like TLA+), model theory, and more.

Mathematically structured programming contrasted with cryptography

Mathematically structured programming (**MSP**) is an approach to programming, based on logic, that tries to minimise the distance between a program and its specification, such that (ideally) the specification *is* the program and is directly executable, without requiring the programmer to code up a realisation in a lower-level programming language.[4]

Benefits include (arguably) easier verification, and greater productivity, because the code is kept close to its logical intention. Disadvantages include lower efficiency: it can be harder to write high-level code that runs quickly (on silicon; in polynomials it may be that a higher-level MSP-based approach is actually more efficient).[5]

Cryptography has historically been about secure and verifiable communication (think: encryption and error-correcting codes); but with the rise of distributed systems, cloud computing, and layer 2 blockchains, the focus of interest in cryptographic techniques has shifted and broadened to include *secure and verifiable computation*. Thus we see modern interest in applications such as succinct proofs of executions, fully homomorphic encryption, multiparty computation, zero knowledge computation, garbled circuits, and more.

The Unique Selling Point

First-order logic is a simple yet powerful logic which has as good a claim as any logic to be a universal (or at least a canonical) language of mathematics. Thanks to this simplicity and power, it is heavily used in mathematically structured programming. Thus, giving a compositional shallow mapping of first-order logic predicates to polynomials has as good a claim as any to being a canonical simple and high-level, and yet potentially very practical, treatment of verifiable computation.

Note that our mapping encodes *semantics and validity*, not *syntax and derivability*. In particular, it would be mistaken to assume that this paper works by encoding the syntax of FOL and the structure of well-formed derivation trees into arithmetic circuits. This paper takes a semantic approach that is arguably simpler; see Remark 6.3.4 for more discussion, with an example.

We will see how we can write specifications in first-order logic, compile them to polynomials, and evaluate them there. The translation has low overhead, and it bridges the gap between the abstractness of logic and having direct access to the underlying polynomials. Because our translation is very compositional, we will be able to compose simple specifications to make more complex ones. We hope this will inform both concrete applications as well as new research in mathematical logic.

[3] There are two distinct notions of 'number' in play here: *integers*, which relate to arithmetic circuits; and *bitwise integers*, which are a distinct thing and relate to Boolean circuits.

[4] Of course, all programming languages do this to some extent (an XKCD comic strip makes a pithy, and entertaining, comment on this), but MSP takes this abstraction to its logical (pun intended) extreme. Aside from the exceptionally high levels of abstraction, MSP is also characterised by something else: using specification languages based on mathematical logic and its semantics.

[5] It is debatable whether the standard instruction-set based architectures that have evolved for efficiency in silicon are also an optimal compilation target in a world made out of polynomials. A remarkable article in The Register (permalink) provides further context and is well worth reading (see the discussion of Dylan).

Who should read this paper

I hope that practitioners will be interested in the ideas in this paper and think about basing practical systems on them. For such readers, the use of logic in MSP style may be new, but we will explain it clearly and with examples. I would argue that this application of logic is not particularly difficult — this is not rocket-science grade logic, it is just propositions and some easy quantifiers — and this is worth looking into because it will yield dividends in simplicity, and offers a promise of reductions in engineering effort and risk. Shorter, simpler, and safer: *pick three!*

The reader with a logic background should also find plenty of food for thought and future research. There is a new polynomial-based denotation, potential for the reader to develop their own tailored logics based on the same ideas (classical, non-classical, and resource-sensitive), and there are some interesting things going on in our applications having to do with use of reflection, truth-modalities, and more (see Remark 6.1.1). The use of cryptography, if this is unfamiliar to the reader, is clearly signposted and clearly explained, and even if the reader prefers to just focus on the logic, this paper should still be entirely clear.

2 Polynomial semantics for FOL

2.1 Some background on polynomials

We recall some standard definitions and notations. We start off very simply:

NOTATION 2.1.1. We will be interested in $\mathbb{Z}$ (integers), $\mathbb{N}$ (nonnegative whole numbers), and $\mathbb{Q}$ (rationals).

We may use subscripts to indicate ranges, writing e.g. $\mathbb{N}_{\geq 0}$ (which is just $\mathbb{N}$), $\mathbb{N}_{\geq 1}$ (strictly positive whole numbers), $\mathbb{Q}_{\geq 0}$, and so on.

Note in particular that $\mathbb{N} = \{0, 1, 2, \dots\}$; as is standard in computer science, we start counting at 0, not 1. However, in keeping with standard usage in mathematics, we will start indexing matrix rows and columns from 1, not 0.

DEFINITION 2.1.2. For the purposes of this paper, a **rational univariate polynomial** (or just **polynomial** for short) is an element $P \in \mathbb{Q}[X]$, where X is a **variable**.

Thus, $P \in \mathbb{Q}[X]$ is a formal sum of powers of X. For example 0, $1 + X$, and X^2 are all polynomials.

Polynomials can be added and multiplied in the usual way. For example, $(1 + X) * (1 - X) = 1 - X^2$.

REMARK 2.1.3. Definition 2.1.2 might seem simple, but for clarity we note a distinction between *expressions denoting polynomials*, and polynomials.

For example: $(1 + X) * (1 - X)$ a polynomial expression. It denotes the polynomial $1 - X^2$, but is not identical to it.

So given $P, Q \in \mathbb{Q}[X]$, when we write $P * Q$ the reader should understand this polynomial expression to mean 'that polynomial in $\mathbb{Q}[X]$ obtained by performing a polynomial multiplication of P and Q', unless stated otherwise.

Polynomial interpolation in Definition 2.1.4 describes a standard method for using a polynomial to express a vector [SW22, Chapter 9].[6]

DEFINITION 2.1.4 (Polynomial interpolation). Suppose $n \geq 0$ and suppose that $M = (m_1, \dots, m_n) \in \mathbb{Z}^n$ is an n-tuple of integers.

1. Say that a polynomial $P \in \mathbb{Q}[X]$ **interpolates** M at $(1, 2, \dots, n)$ when $P(i) = m_i$ for each $1 \leq (i \in \mathbb{N}_{\geq 1}) \leq n$.

[6] The Wikipedia page and this overview are excellent and very accessible introductions.

$$\begin{array}{rl} t ::= & X \mid q \in \mathbb{Q} \mid t + t \mid t - t \mid t * t \\ & \mid \mathsf{C}_i(t) \mid \mathsf{len}(\mathsf{C}) \quad (1 \leq i{\in}\mathbb{N}_{\geq 1} \leq \mathit{arity}(\mathsf{C})) \\ & \mid \mathsf{term}(\phi) \\ \\ \phi, \psi ::= & \mathsf{T} \mid \bot \mid t = t \mid \phi \wedge \phi \mid \phi \vee \phi \\ & \mid \forall X_{\mathsf{C}}.\phi \mid \exists X_{\mathsf{C}}.\phi \\ & \mid \mathsf{C}_i < \mathit{vft} \mid \mathit{vft} < \mathsf{C}_i \\ & \qquad (\mathit{vft} \text{ variable-free}, \ 1 \leq i{\in}\mathbb{N}_{\geq 1} \leq \mathit{arity}(\mathsf{C})) \end{array}$$

The grammars above are in BNF style, such that ϕ to the right of $::=$ represents any predicate. In particular, $\phi \wedge \phi$ is not a typo; it denotes a conjunction of two (possibly non-equal) predicates. Similarly the two t in $t = t$ denote two (possibly non-equal) terms.

Figure 1: Syntax of terms and predicates (Definition 2.2.3(3))

2. Recall that every n-tuple of integers M can be interpolated by a unique **polynomial interpolant**
$$\mathit{intrplnt}(m_1, \ldots, m_n) \in \mathbb{Q}[X]$$
of degree n-1 (at most). Thus, $\mathit{intrplnt}(m_1, \ldots, m_n)(i) = m_i$ for $1 \leq (i \in \mathbb{N}_{\geq 1}) \leq n$.

See also Definition 5.1.3, where we extend interpolation to matrices.

NOTATION 2.1.5 (Instantiation). Suppose $P, Q \in \mathbb{Q}[X]$ are polynomials. Write $P(Q)$ for the polynomial obtained by **instantiating** X to Q in P. For example if $P = X + 1$ and $Q = 2 * X$ then
$$P(Q) = 2 * X + 1.$$
In the case that Q is just a number $x \in \mathbb{Q}$, then we may identify this with the number obtained by instantiating X to x and evaluating the result using arithmetic. For example:
$$(X * (X + 1))(6) \ = \ 6 * (6 + 1) \ = \ 42.$$
It will alway be clear what is intended.

2.2 The syntax of our logic (with examples)

REMARK 2.2.1. Our goal in this paper is to define a logic, give it a semantics in $\mathbb{Q}[X]$, and explore the properties of this semantics. We will:

1. define the formal syntax of terms and predicates in Definition 2.2.3;
2. discuss some intuitions in Remark 2.2.4;
3. define the polynomial semantics for our formal syntax in Definition 2.3.2;
4. discuss the semantics in Remark 2.3.4; and
5. prove a soundness and completeness result in Theorem 2.4.4.

REMARK 2.2.2. Note that in the interpretation we are about to build in Figure 2, 0 represents truth and any non-zero value represents false; furthermore $\wedge$ and $\forall$ (conjunction and universal quantification) map to $+$ (addition), and $\vee$ and $\exists$ (disjunction and existential quantification) map to $*$.

If the reader is used to thinking of truth as 1 and false as 0, as is traditional in logic circuits, then be careful: everything below will be flipped with respect to what you initially expect. Our motivation for representing truth by 0 is that it gives us Theorem 2.4.4.[7] More on this in Remarks 2.3.4 and 2.4.5).

[7] Representing truth with 0 and non-truth with nonzero does have some pedigree. In a paper from 1943 [Kle43] on page 51 just after equation 17, Kleene writes *"a representing function π of P, the value of which is to be* 0, 1, *or undefined according as the value of P is true, false, or undefined"*. More recently, in the `sysexists.h` file (credited to Eric Allman in 1980) which describes status codes for system programs, 0 represents successful termination and nonzero values represent various kinds of failure.

DEFINITION 2.2.3 (Syntax of terms and predicates).

1. Fix one **index variable symbol** X.
2. Fix a set of **matrix variable symbols** $\mathsf{C}, \mathsf{D} \in \mathsf{MatrixVar}$.
   To each matrix variable symbol we associate a fixed but arbitrary **arity** $\mathit{arity}(\mathsf{C}) \in \mathbb{N}_{\geq 1}$.
3. Define **terms** and **predicates** inductively as in Figure 1. In that Figure:
   (a) i ranges over whole numbers between 1 and the arity of C inclusive, or in symbols: $1 \leq (i \in \mathbb{N}_{\geq 1}) \leq \mathit{arity}(\mathsf{C})$.
   (b) This is just formal syntax; we give it meaning later in Figure 2. In particular, C_i, $\mathsf{C}_i(t)$, $\mathsf{len}(\mathsf{C})$, and $\mathsf{term}(\phi)$ are just formal syntax.
   (c) We call a predicate or term **variable-free** when it does not mention X.
   vft in the clauses $\mathsf{C}_i < \mathit{vft}$ and $\mathit{vft} < \mathsf{C}_i$ in Figure 2 ranges over variable-free terms. For instance, *vft* could be 0, or $\mathsf{len}(\mathsf{C})$. Put simply, *vft* ranges over terms that will denote a polynomial that is just a number.

REMARK 2.2.4. We give some intuitive discussion of the syntax in Definition 2.2.3:

1. The index variable symbol X is the (unique) variable of the polynomial semantics. Terms and predicates will map to $\mathbb{Q}[X]$ (polynomials over X), as per Figure 2.
2. Intuitively a matrix variable symbol $\mathsf{C} \in \mathsf{MatrixVar}$ ranges over all integer matrices of size $\mathit{arity}(\mathsf{C}) \times n$, for all $n \in \mathbb{N}_{\geq 1}$.
   Using interpolation (Definition 2.1.4) we can also think of C as ranging over elements of $\mathbb{Q}[X]^{\mathit{arity}(\mathsf{C})} \times \mathbb{N}_{\geq 1}$, where the first component is an $\mathit{arity}(\mathsf{C})$-tuple of polynomials over $\mathbb{Q}[X]$, and the second component is the $n \geq 1$ that tells us how to reconstruct the matrix by considering the values of those polynomials on $\{1, \ldots, n\}$.
3. Terms let us build elements of $\mathbb{Q}[X]$.
   The intuition of $\mathsf{C}_i(t)$ is an interpolating polynomial for the ith row of C, applied to t (more on this below). The intuition of $\mathsf{len}(\mathsf{C})$ is the number of columns in C (its *length*). The intuition of $\mathsf{term}(\phi)$ is to treat ϕ as a number (this makes the logic *impredicative*; we fold predicates back down into terms).
4. Predicates are based on first-order logic. There is equality (of terms) but no negation (because negation is expressible; see Subsection 4.2.2).
   Intuitively, quantification $\forall X_{\mathsf{C}}$ quantifies over X ranging over whole numbers from 1 to the number of columns in the matrix denoted by C; so if C denotes an $\mathit{arity}(\mathsf{C}) \times n$ matrix, then $\forall X_{\mathsf{C}}.\phi$ checks $\phi(1), \ldots, \phi(n)$, where $\phi(i)$ is the value of (some polynomial denotation which we have not yet constructed of) ϕ for X instantiated to $i \in \{1, \ldots, n\}$.
5. The intuition of the range check $\mathit{vft} < \mathsf{C}_i$ and $\mathsf{C}_i < \mathit{vft}$ is to statically check that elements of the ith row of C takes values greater than or less than *vft* respectively. We use this later to statically check for out-of-bounds pointer errors (see the discussion in Examples 3.2.8 and 3.2.9), and we make further use of this in Section 4.

EXAMPLE 2.2.5. We give some examples of the syntax in action (even if we have not yet assigned it any formal meaning, we can illustrate how it is supposed to be used). Assume three matrix variable symbols:

1. neg with $\mathit{arity}(\mathsf{neg}) = 2$.
2. add with $\mathit{arity}(\mathsf{add}) = 3$.
3. pos with arity $\mathit{arity}(\mathsf{pos}) = 1$.

Then we can write some easy predicates:

1. $\forall X_{\mathsf{neg}}.\mathsf{neg}_2(X) = (\text{-}1) * \mathsf{neg}_1(X)$.
2. $\forall X_{\mathsf{add}}.\mathsf{add}_3(X) = \mathsf{add}_1(X) + \mathsf{add}_2(X)$.
3. $0 < \mathsf{pos}_1$.

Intuitively, we are supposed to read these predicates as follows:

1. neg is ($=$ denotes) a matrix with two rows, and the Xth entry in row 2 is (as per the predicate) supposed to be the negation of the Xth entry in row 1, like this:
$$\begin{pmatrix} 0 & 1 & 2 \\ 0 & -1 & -2 \end{pmatrix}$$
2. add is a matrix with three rows. Entries in row 3 are (as per the predicate) supposed to be the sum of the entries in rows 1 and 2, like this:
$$\begin{pmatrix} 0 & 1 & 2 \\ 2 & 1 & 0 \\ 2 & 2 & 2 \end{pmatrix}$$
3. pos is a matrix with just one row, which should consist entirely of strictly positive numbers, like this:
$$\begin{pmatrix} 3 & 2 & 1 \end{pmatrix}$$

In terms of intuition, this is all straightforward.[8]

We do not have a notion of validity yet, so all of the above is just intuition. Once we have defined validity, we will come back to this; see Example 2.3.3.

REMARK 2.2.6. We make a few comments on some simple (non)redundancies in the syntaxes of terms and predicates:

1. Subtraction in terms is redundant: we can just express $t - t'$ as $t + (-1) * t'$.
2. T and $\bot$ are redundant: we can just express T as $0 = 0$, and $\bot$ as $1 = 0$.
3. If we admit a unary predicate-former $\mathsf{isZero}(t)$ with semantics $[\![\mathsf{isZero}(t)]\!]_\varsigma = [\![t]\!]^2_\varsigma$, then T, $\bot$, and $t - t'$ become expressible as
$$\mathsf{T} = \mathsf{isZero}(0), \qquad \bot = \mathsf{isZero}(1), \quad \text{and} \quad (t = t') = \mathsf{isZero}(t - t').$$
4. Negation does not appear explicitly in the syntax or semantics of Figures 1 and 2. Surprisingly, negation is expressible in the rest of the logic: see Subsection 4.2.2.

See also Remark 2.4.6.

NOTATION 2.2.7. We set up some convenient abbreviations, writing:

$vft \leq \mathsf{C}_i$	for	$vft - 1 < \mathsf{C}_i$
$\mathsf{C}_i \leq vft$	for	$\mathsf{C}_i < vft + 1$
$vft < \mathsf{C}_i < vft'$	for	$(vft < \mathsf{C}_i) \wedge (\mathsf{C}_i < vft')$
$vft \leq \mathsf{C}_i \leq vft'$	for	$(vft - 1 < \mathsf{C}_i) \wedge (\mathsf{C}_i < vft + 1)$.

Note that this is meaningful because when we build our semantics, our notion of valuation in Definition 2.3.2(1) will evaluate matrix variable symbols C to *integer* matrices.

There is not much to this otherwise: we could also just as well take the notation above as primitive in our syntax from Figure 1, and interpret $\leq$ (a binary predicate symbol in our formal syntax) using $\leq$ (the binary predicate 'less than or equals' on numbers, which returns a truth-value) in Figure 2, alongside interpreting $<$ (the predicate symbol) using $<$ (the predicate 'strictly less than').

2.3 The semantics (with examples)

We are now ready to define a semantics in polynomials for the syntax from Definition 2.2.3.

Some notation will be useful for Figure 2 and Definition 2.3.2:

[8] In fact, it will turn out that pos is more interesting than one might expect. We will see more of it later in Subsection 4.1.1.

$$
\begin{aligned}
[\![X]\!]_\varsigma &= X & [\![q]\!]_\varsigma &= q \qquad (q \in \mathbb{Q}) \\
[\![t + t']\!]_\varsigma &= [\![t]\!]_\varsigma + [\![t']\!]_\varsigma & [\![t - t']\!]_\varsigma &= [\![t]\!]_\varsigma - [\![t']\!]_\varsigma \\
[\![t * t']\!]_\varsigma &= [\![t]\!]_\varsigma * [\![t']\!]_\varsigma & [\![\mathsf{C}_i(t)]\!]_\varsigma &= intrplnt(\varsigma(\mathsf{C})_i)([\![t]\!]_\varsigma) \\
[\![\mathsf{len}(\mathsf{C})]\!]_\varsigma &= len(\varsigma(\mathsf{C})) & [\![\mathsf{term}(\phi)]\!]_\varsigma &= [\![\phi]\!]_\varsigma
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{T}]\!]_\varsigma &= 0 & [\![\bot]\!]_\varsigma &= 1 \\
[\![t = t']\!]_\varsigma &= ([\![t]\!]_\varsigma - [\![t']\!]_\varsigma)^2 & & \\
[\![\phi \wedge \phi']\!]_\varsigma &= [\![\phi]\!]_\varsigma + [\![\phi']\!]_\varsigma & [\![\phi \vee \phi']\!]_\varsigma &= [\![\phi]\!]_\varsigma * [\![\phi']\!]_\varsigma \\
[\![\forall X_{\mathsf{C}}.\phi]\!]_\varsigma &= \textstyle\sum_{1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))} [\![\phi]\!]_\varsigma & &
\end{aligned}
$$

$$
[\![\exists X_{\mathsf{C}}.\phi]\!]_\varsigma = \textstyle\prod_{1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))} [\![\phi]\!]_\varsigma
$$

$$
[\![\mathsf{C}_i < vft]\!]_\varsigma = \begin{cases} 0 & \text{if } \forall 1 \leq j \leq len(\varsigma(\mathsf{C})).(\varsigma(\mathsf{C})_{i,j} < [\![vft]\!]_\varsigma) \\ 1 & \text{otherwise} \end{cases}
$$

$$
[\![vft < \mathsf{C}_i]\!]_\varsigma = \begin{cases} 0 & \text{if } \forall 1 \leq j \leq len(\varsigma(\mathsf{C})).([\![vft]\!]_\varsigma < \varsigma(\mathsf{C})_{i,j}) \\ 1 & \text{otherwise} \end{cases}
$$

$$
x \vDash_\varsigma \phi \iff [\![\phi]\!]_\varsigma(x) = 0
$$

Negation is expressible using the rest of the logic: see Subsection 4.2.2.

Figure 2: Polynomial semantics for terms and predicates (Definition 2.3.2)

NOTATION 2.3.1 (Some basic matrix syntax). Suppose M is a matrix. Then:

1. We may write $len(M)$ for the number of columns in M, and we may call this the **length** of M.
2. We may write M_i for the ith row in M, which is a vector of length $len(M)$.
3. As usual, we may write $M_{i,j}$ for the i,j-th element in M; i is the row number and j is the column number.

DEFINITION 2.3.2 (Polynomial semantics).

1. A **valuation** ς assigns to each matrix variable symbol $\mathsf{C} \in \mathsf{MatrixVar}$ with arity $arity(\mathsf{C}) \in \mathbb{N}_{\geq 1}$ an $arity(\mathsf{C}) \times n$ integer matrix (= having elements from $\mathbb{Z}$).
   Using interpolation (Definition 2.1.4) the reader can also think of $\varsigma(\mathsf{C})$ as a vector of $arity(\mathsf{C})$ many polynomials in X, along with a number n encoding the information that we care about the values of these polynomials on $\{1, \ldots, n\}$.
2. Given a valuation ς, define **polynomial semantics**
   $$[\![t]\!]_\varsigma \in \mathbb{Q}[X] \quad \text{and} \quad [\![\phi]\!]_\varsigma \in \mathbb{Q}[X]$$
   for terms and predicates as in Figure 2. See Remark 2.3.4 below for exposition on the notation and design.
   If ς is irrelevant (e.g. because no matrix variable symbols are mentioned), we may drop the subscript ς and write $[\![t]\!]_\varsigma$ just as $[\![t]\!]$, or $[\![\phi]\!]_\varsigma$ just as $[\![\phi]\!]$. For example, we may write $[\![2 = 0]\!] = 4$.
3. Given $x \in \mathbb{Q}$, define a **judgement** by
   $$x \vDash_\varsigma \phi \quad \text{means} \quad [\![\phi]\!]_\varsigma(x) = 0,$$
   as per Figure 2. Judgements return truth-values ('true' or 'false').[9]

[9] So if ϕ is a predicate and $x \in \mathbb{Q}$ and ς is a valuation then: $[\![\phi]\!]_\varsigma$ is a polynomial; $[\![\phi]\!]_\varsigma(x)$ is a rational number; and $x \vDash_\varsigma \phi$ is a truth-value.

4. If x or ς is irrelevant in $x \vDash_\varsigma \phi$ (e.g. because ϕ is fully-quantified, or mentions no matrix variable symbols) then we may omit it, writing

$$\vDash_\varsigma \phi \quad \text{or} \quad x \vDash \phi \quad \text{or even just} \quad \vDash \phi.$$

When $x \vDash_\varsigma \phi$ we will call ϕ **valid** (in the context of x and ς). If x and/or ς is irrelevant, we may just call ϕ **valid**.

EXAMPLE 2.3.3. We now use Figure 2 to translate the predicates from Example 2.3.3 into polynomials. Recall that we assumed three matrix variable symbols with associated predicates — since we now have a notion of validity, we will call these *validity predicates* — as follows:

1. neg with $\mathit{arity}(\mathsf{neg}) = 2$ and validity predicate $\chi_{\mathsf{neg}} = \forall X_{\mathsf{neg}}.\mathsf{neg}_2(X) = (\text{-}1) * \mathsf{neg}_1(X)$.
2. add with $\mathit{arity}(\mathsf{add}) = 3$ and validity predicate $\chi_{\mathsf{add}} = \forall X_{\mathsf{add}}.\mathsf{add}_3(X) = \mathsf{add}_1(X) + \mathsf{add}_2(X)$.
3. pos with arity $\mathit{arity}(\mathsf{pos}) = 1$ and validity predicate $0 < \mathsf{pos}_1$.

Given a valuation ς, the translations of these validity predicates are polynomials

$$[\![\mathsf{neg}]\!]_\varsigma,\ [\![\mathsf{add}]\!]_\varsigma,\ [\![\mathsf{pos}]\!]_\varsigma\ \in \mathbb{Q}[X]$$

as per Figure 2 as follows:

$$[\![\chi_{\mathsf{neg}}]\!]_\varsigma = \textstyle\sum_{1\leq(x\in\mathbb{N}_{\geq 1})\leq \mathit{len}(\varsigma(\mathsf{neg}))}(\mathit{intrplnt}(\mathsf{neg}_2)(x) - (\text{-}1) * \mathit{intrplnt}(\mathsf{neg}_1)(x))^2$$
$$[\![\chi_{\mathsf{add}}]\!]_\varsigma = \textstyle\sum_{1\leq(x\in\mathbb{N}_{\geq 1})\leq \mathit{len}(\varsigma(\mathsf{add}))}(\mathit{intrplnt}(\mathsf{add}_3)(x) - (\mathit{intrplnt}(\mathsf{add}_1)(x)+\mathit{intrplnt}(\mathsf{add}_2)(x)))^2$$
$$[\![\chi_{\mathsf{pos}}]\!]_\varsigma = \begin{cases} 0 & \text{every entry in } \varsigma(\mathsf{pos}) \text{ is } > 0 \\ 1 & \text{some entry in } \varsigma(\mathsf{pos}) \text{ is } \leq 0 \end{cases}$$

Note that none of these polynomials have X free in them, so in fact they are just numbers. Thus, following Definition 2.3.2(4) the predicates are *valid* in the context of ς when that number is 0, and they are *invalid* when that number is strictly greater than 0.

The reader can check that the predicates are valid of ς, precisely when ς satisfies the conditions that we would intuitively expect (as spelled out in Example 2.3.3) based on reading the predicates. This observation will be formalised in our soundness and completeness result in Theorem 2.4.4.

REMARK 2.3.4. Some comments on Figure 2:

1. T maps to $[\![\mathsf{T}]\!]_\varsigma = 0$ and $\bot$ maps to $[\![\bot]\!]_\varsigma = 1$. Conjunction and universal quantification map to addition, and disjunction and existential quantification map to multiplication (cf. also Remark 2.4.5). Consequently we can say:

   In our denotation, 'zero' means 'true' and 'nonzero' means 'false'.

   This might be confusing because usually (e.g. in logic gates) 1 means 'true' and (just) 0 means 'false', but setting things up as we do gives us Theorem 2.4.4.

2. The semantics maps terms and predicates to polynomials over X. So for example, the clause $[\![X]\!]_\varsigma = X$ says that the X denotes itself.

3. Similary the clause $[\![q]\!]_\varsigma = q$ says that $q \in \mathbb{Q}$ denotes itself.

4. The clause $[\![\mathsf{C}_i(t)]\!]_\varsigma = \mathit{intrplnt}(\varsigma(\mathsf{C})_i)([\![t]\!]_\varsigma)$ carries an implicit condition that $1 \leq (i \in \mathbb{N}) \leq \mathit{arity}(\mathsf{C})$, since as per Definition 2.2.3(3a) and Notation 2.3.1(2) it would not make sense to talk about C_i or $\varsigma(\mathsf{C})_i$ otherwise.

5. We break down the clause $[\![\mathsf{C}_i(t)]\!]_\varsigma = \mathit{intrplnt}(\varsigma(\mathsf{C})_i)([\![t]\!]_\varsigma)$, as follows:

- $\varsigma(\mathsf{C})$ is an integer matrix with $arity(\mathsf{C})$ rows and $len(\varsigma(\mathsf{C}))$ columns. For brevity we may write
$$C = \varsigma(\mathsf{C}).$$
Note that the number of rows in C *must* be $arity(\mathsf{C})$; but the number of columns in C is determined by the choice of ς.
- As per Notation 2.3.1(2), C_i denotes the ith row of C, which is a vector of $len(C)$ integers.
- $intrplnt(C_i) = P$ is a polynomial in X that interpolates the vector C_i at values $1, \ldots, n$, as per Definition 2.1.4.
- $P(\llbracket t \rrbracket_\varsigma)$ is P instantiated at $\llbracket t \rrbracket_\varsigma$, as per Notation 2.1.5.

6. Reading the previous point, the reader might ask why we bother with matrices at all. Why not just let $\varsigma(\mathsf{C})_i$ *be* a polynomial?

   Because a matrix and its interpolant are not the same thing, if we care about bounds (which we do); a vector has a *length*, and a polynomial *a priori* does not. For example, the same polynomial X interpolates many distinct vectors, including (1), $(1,2)$ and $(1,2,3)$.[10]

Remark 2.3.5. Some comments on Definition 2.3.2:

1. $\varsigma(\mathsf{C})$ in Definition 2.3.2(1) is an integer matrix C with $arity(\mathsf{C})$ rows and n columns. The number of rows in C is determined by $arity(\mathsf{C})$, but its number of columns depends on ς.

2. Valuations ς map matrix variable symbols to integer matrices. They do not instantiate the (single) variable symbol X!

3. In Definition 2.3.2(2), $\llbracket \phi \rrbracket_\varsigma$ is a polynomial in X — it is not a truth-value. For example,
$$\llbracket X = 2 * X \rrbracket_\varsigma = X^2$$
(there are no matrix variable symbols here so the choice of ς does not matter).

4. In Definition 2.3.2(3), $x \vDash_\varsigma \phi$ is a truth-value — it is not a polynomial. For example,
$$0 \vDash_\varsigma X = 2 * X \text{ is true, and } 1 \vDash_\varsigma X = 2 * X \text{ is false.}$$

5. The denotation of universal quantification $\forall$ is a summation, like the denotation of $\bigwedge$. However, $\forall$ cannot be emulated in the logic by repeated $\bigwedge$, because the length of the summation depends on the number of columns in the matrix $\varsigma(\mathsf{C})$, which varies depending on the choice of ς. So universal quantification has its own distinct identity.

6. A design path which we do not explore in this paper is that our denotation is compatible with notions of 'logical implication' $P \Longrightarrow Q$ and 'logical equivalence' $P \Longleftrightarrow Q$ on polynomials $P, Q \in \mathbb{Q}[X]$, such that
$$\begin{array}{lll} P \Longrightarrow Q & \text{when } \forall x{\in}\mathbb{Q}.P(x) = 0 \Longrightarrow Q(x) = 0 & \text{and} \\ P \Longleftrightarrow Q & \text{when } \forall x{\in}\mathbb{Q}.P(x) = 0 \Longleftrightarrow Q(x) = 0. & \end{array}$$
Note that $P \Longrightarrow Q$ does not quite mean the same thing as $P \mid Q$ (P divides Q), because the multiplicities of the roots in P may differ from those in Q. Similarly, $P \Longleftrightarrow Q$ does not quite mean the same thing as $P = Q$. For example:
$$(X-1) \Longleftrightarrow (X-1) * (X^2+1) \quad \text{and} \quad (X-1) \Longleftrightarrow (X-1)^2.$$
We do not explore these notions of implication and equivalence explicitly in what follows, but they are implicit in much of the maths; exploring this explicitly is future work.

   For now, it seems to be more useful to explore a technically different design path and consider a *complementation operation* $\sim\!\phi$; see Notation 4.2.6 and Theorem 4.2.7.

[10] Obviously, in an implementation we care about efficiency so we might (or might not) prefer the rows of our matrix to be delivered as a vector of interpolating polynomials, along with an intended length. We can choose how best to represent our data. But, what that representation *represents*, is a matrix.

2.4 Soundness and completeness

Definition 2.4.1. Call a polynomial $P \in \mathbb{Q}[X]$ **nonnegative** when $P(x) \geq 0$ for every $x \in \mathbb{Q}$.

Lemma 2.4.2. *Suppose ϕ is a predicate and ς is a valuation and $x \in \mathbb{Q}$.*
Then $[\![\phi]\!]_\varsigma$ is a nonnegative polynomial (Definition 2.4.1).

Proof. By a routine induction, following Figure 2:

- $[\![\mathsf{T}]\!]_\varsigma = 0$ and $[\![\bot]\!]_\varsigma = 1$. We note that 0 and 1 are nonnegative.
- $[\![t = t']\!] = ([\![t]\!]_\varsigma - [\![t']\!]_\varsigma)^2$. It is a fact of arithmetic that this is a nonnegative polynomial, because it is a square.[11]
- $[\![\phi \wedge \phi']\!]_\varsigma = [\![\phi]\!]_\varsigma + [\![\phi']\!]_\varsigma$ and $[\![\phi \vee \phi']\!]_\varsigma = [\![\phi]\!]_\varsigma * [\![\phi']\!]_\varsigma$. We observe that the sum and product of two (by inductive hypothesis) nonnegative polynomials, is nonnegative.
- As per figure 2, $[\![\forall X_{\mathsf{C}}.\phi]\!] = \sum_{1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))} [\![\phi]\!]_\varsigma$. We observe that this is a sum of (by inductive hypothesis) nonnegative polynomials, and so it is also nonnegative. Similarly for $\exists_{\mathsf{C}} X.\phi$.
- $[\![vft < \mathsf{C}_i]\!]_\varsigma$ and $[\![\mathsf{C}_i < vft]\!]_\varsigma$ are equal to 0 or 1, and both are nonnegative. □

Remark 2.4.3. Lemma 2.4.2 is interesting because it suggests that we can identify the nonnegative polynomials in $\mathbb{Q}[X]$ as having *logical content*, in the sense that these can be the denotations of predicates; in contrast to the class of all polynomials in $\mathbb{Q}[X]$, which can be the denotation of terms but not necessarily of predicates.

Theorem 2.4.4 (Soundness & completeness). *Suppose ϕ and ϕ' are predicates, and t and t' are terms, and ς is a valuation and $x \in \mathbb{Q}$. Then:*

1. $x \vDash_\varsigma \mathsf{T}$ *and* $x \nvDash_\varsigma \bot$
2. $x \vDash_\varsigma t = t'$ *if and only if* $[\![t]\!]_\varsigma(x) = [\![t']\!]_\varsigma(x)$.
3. $x \vDash_\varsigma \phi \wedge \phi'$ *if and only if* $x \vDash_\varsigma \phi \ \wedge \ x \vDash_\varsigma \phi'$.
4. $x \vDash_\varsigma \phi \vee \phi'$ *if and only if* $x \vDash_\varsigma \phi \ \vee \ x \vDash_\varsigma \phi'$.
5. $x \vDash_\varsigma \forall X_{\mathsf{C}}.\phi$ *if and only if* $x' \vDash_\varsigma \phi$ *for every* $1 \leq (x' \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))$.
6. $x \vDash_\varsigma \exists X_{\mathsf{C}}.\phi$ *if and only if* $x' \vDash_\varsigma \phi$ *for some* $1 \leq (x' \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))$.
7. $x \vDash_\varsigma vft < \mathsf{C}_i$ *if and only if every entry in the ith row of $\varsigma(\mathsf{C})$ is strictly greater than* $[\![vft]\!]_\varsigma$.[12]
8. $x \vDash_\varsigma \mathsf{C}_i < vft$ *if and only if every entry in the ith row of $\varsigma(\mathsf{C})$ is strictly less than* $[\![vft]\!]_\varsigma$.

Proof. We consider each part in turn, unpacking Figure 2. The argument is just by routine arithmetic:

1. $x \vDash \mathsf{T}$ when $0 = 0$, which is true. $x \vDash \bot$ when $1 = 0$, which is false.
2. $x \vDash t = t'$ when $(t - t')^2(x) = 0$. By elementary facts of arithmetic, this is if and only if $t(x) = t'(x)$.
3. $x \vDash \phi \wedge \phi'$ when $[\![\phi]\!](x) + [\![\phi']\!](x) = 0$. Using Lemma 2.4.2, this holds if and only if $[\![\phi]\!] = 0$ and $[\![\phi']\!] = 0$, and thus if and only if $x \vDash \phi \wedge x \vDash \phi'$.
4. $x \vDash \phi \vee \phi'$ when $[\![\phi]\!] * [\![\phi']\!] = 0$. Using the fact that $\mathbb{Q}$ is an *integral domain* [Hun12, page 48], this holds and only if $[\![\phi]\!] = 0$ or $[\![\phi']\!] = 0$.
5. As for the case of $\wedge$, using Lemma 2.4.2 and arithmetic, the sum $\sum_{1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))} [\![\phi]\!]_\varsigma$ is zero if and only if all of its component summands are zero.
6. The product $\prod_{1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))} [\![\phi]\!]_\varsigma$ is zero if and only if one of its component summands is zero.

[11] Indeed, the square is there precisely to guarantee nonnegativity.
[12] Note that $[\![vft]\!]_\varsigma$ is just a number, because vft is a term that is assumed to not mention X.

7. By Figure 2, $[\![vft < \mathsf{C}_i]\!]_\varsigma$ means precisely that vft is strictly less than every entry in the ith row of $\varsigma(\mathsf{C})$. Similarly for $\mathsf{C}_i < vft$.

   We will also use $vft \leq \mathsf{C}_i$ and $\mathsf{C}_i \leq vft$ as per Notation 2.2.7. Because $\varsigma(\mathsf{C})$ is an integer matrix, $[\![vft \leq \mathsf{C}_i]\!]_\varsigma$ means precisely that vft is less than or equal to every entry in the ith row of $\varsigma(\mathsf{C})$, and similarly for $\mathsf{C}_i \leq vft$. □

Remark 2.4.5. Theorem 2.4.4 is important — a logical judgements is valid precisely when its polynomial translation is — but mathematically it is elementary. It just dresses up the fact that in $\mathbb{Q}$,

- a sum of two nonnegative numbers is zero if and only if both numbers are zero, and
- a product of two nonnegative numbers is zero if and only if at least one of the numbers is zero.

But of course, this is just what we need to precisely interpret logical conjunction / universal quantification, and logical disjunction / existential quantification, respectively.

What is absent from Theorem 2.4.4, because it is absent from our syntax in Figure 1, is *negation* — there is no explicit predicate-former $\neg\phi$. This is because it can be expressed using the rest of the logic; we will see later in Notation 4.2.6 and Theorem 4.2.7 that our logic is *complemented*. That is: given ϕ, we can write a complement predicate $\sim\phi$ that is valid if and only if ϕ is not valid.

But note also that our examples will show that even the negation-free fragment of FOL is very expressive (expressive enough to e.g. encode a Turing-complete model of computation; see Subsection 3.9).

For now, we just note that Theorem 2.4.4 is more powerful than it looks, and it will turn out that it can indeed lay claim to expressing full first-order logic with negation.

Remark 2.4.6 is in a sense a continuation of Remark 2.2.6:

Remark 2.4.6 (Dualities).

1. $\exists$ is a natural de Morgan dual to $\forall$ (as usual). We will use $\forall$ all the time in our practical examples below, but not make much use of $\exists$. It costs nothing to include it anyway.
2. term maps from nonnegative polynomials to all polynomials just by being the identity. This is in a sense dual to the 'comparison with zero' $(t = 0)$, which maps from all polynomials to nonnegative polynomials by squaring. We *will* use this, most notably in our complementation operation from Notation 4.2.6.

Remark 2.4.7. Later on in Subsection 5.1.3 we show how to obtain succinct proofs of our polynomial semantics using an inner product argument. Soundness and completeness of the overall translation follow by combining Theorem 2.4.4 with soundness and completeness of the inner product argument. This is not complicated, but readers from cryptography may not be expecting it (they may be used to seeing monolithic arguments for soundness and completeness). We see this as a feature, not a bug, because it makes the reasoning simple and modular.

3 Expressing functions

3.1 Simpler is better

Remark 3.1.1. We have our logic and its denotation. We are now ready to use it to specify functions.

The format of the results in this section is generally the same:

1. We specify some (well-known) mathematical function, such as exponentiation.
2. We write a predicate in our formal syntax from Definition 2.2.3(3). This will correspond in some fairly natural way to the specification.
3. We state a soundness lemma, whose proof is usually elementary from Theorem 2.4.4, that the semantics of the predicate as per Figure 2 accurately captures the specification.

This can be a bit fiddly, because concretely manipulating large-ish expressions by hand can be fiddly, but mathematically it is straightforward. There is *supposed* to be a close match between the mathematical intuition, the formal syntax, and the semantics, so that the soundness proofs become straightforward. Our framework is designed to impose a minimal overhead, above and beyond any complexities that may be inherent in what is being expressed.

So for example, if the reader

- looks at Definition 3.2.1 (mathematical definition of exponentiation), and then
- looks at Figure 3 (formal predicate in our logic), and finally perhaps
- looks at Example 3.2.11 (unpacking the polynomial to which the formal predicate compiles)

and finds that it all looks rather straightforward, then that simplicitly is a feature, not a bug.

3.2 Expressing a power function (with worked examples)

We start with a simple example: how to express a power function (exponentiation) a^b for $a, b \in \mathbb{N}_{\geq 0}$.

DEFINITION 3.2.1. Recall the standard inductive definition of a^b for $a, b \in \mathbb{N}_{\geq 0}$:

$$\begin{array}{ll} \text{base case} & pow(a, 0) = 1 \\ \text{inductive step} & pow(a, b+1) = a * pow(a, b) \end{array}$$

REMARK 3.2.2. The reader may know that this definition is not particularly efficient. In Remark 3.2.14 we mention the efficient definition; but for the purposes of an initial example, Definition 3.2.1 will serve us very well.

Note an elementary point that definitions like Definition 3.2.1 implicity equate *computation* with *derivation*, because when we try to compute a value — $pow(2, 2)$, say — we end up with a simple derivation-tree as follows:

$$\dfrac{\dfrac{\dfrac{}{pow(2,0)=1}\text{ base case}}{pow(2,1)=2}\text{ inductive step}}{pow(2,2)=4}\text{ inductive step}$$

This derivation-tree, simple as it is, illustrates the close link between logic and computation.

DEFINITION 3.2.3. Assume a matrix variable symbol pow with arity $arity(\mathsf{pow}) = 4$. Thus, a denotation $\varsigma(\mathsf{pow})$ is a $4 \times n$ integer matrix, for $n \geq 1$. We will encode an assertion that pow encodes a (partial) power function, such that:

1. Row 1 (the values of $\mathsf{pow}_1(X)$) stores input values a.
2. Row 2 (the values of $\mathsf{pow}_2(X)$) stores input values b.
3. Row 3 (the values of $\mathsf{pow}_3(X)$) stores result values a^b.
4. Row 4 (the values of $\mathsf{pow}_4(X)$) stores a pointer (specifically: a column index) to a column in the pow matrix that represents the recursive call to pow in clause 2 of Definition 3.2.1 above.

The schema in Definition 3.2.5 is typical of how we will work, so we spell it out now:

NOTATION 3.2.4.

1. For each matrix variable symbol C of interest, we write out a corresponding **validity predicate** χ_{C}, in which we assume X is not free (e.g. it is bound by a $\forall X$ quantifier).
2. We say that ς **satisfies** this validity predicate when $\models_\varsigma \chi_{\mathsf{C}}$.
   Recall that the $\models_\varsigma \chi_{\mathsf{C}}$ notation is from Definition 2.3.2(4); it means $x \models_\varsigma \chi_{\mathsf{C}}$ and so $[\![\chi_{\mathsf{C}}]\!](x) = 0$ for some x, the choice of which does not matter, because X is not free in χ_{C}.

DEFINITION 3.2.5. In Figure 3 we

$$\begin{array}{ll} \mathsf{pow}_1(t) \text{ sugars to } \mathsf{a}(t) & \mathsf{pow}_2(t) \text{ sugars to } \mathsf{b}(t) \\ \mathsf{pow}_3(t) \text{ sugars to } \mathsf{out}(t) & \mathsf{pow}_4(t) \text{ sugars to } \mathsf{rec}(t) \end{array}$$

$$\begin{aligned} \mathsf{clause1}(X) = \quad & (\mathsf{b}(X) = 0) \wedge (\mathsf{out}(X) = 1) \\ \mathsf{clause2}(X) = \quad & \mathsf{a}(X) = \mathsf{a}(\mathsf{rec}(X)) \wedge \\ & \mathsf{b}(X) = \mathsf{b}(\mathsf{rec}(X)) + 1 \wedge \\ & \mathsf{out}(X) = \mathsf{a}(X) * \mathsf{out}(\mathsf{rec}(X)) \\ \chi_{\mathsf{pow}} = \quad & 1 \leq \mathsf{rec} \leq \mathsf{len}(\mathsf{pow}) \ \wedge \ \forall X_{\mathsf{pow}}.(\mathsf{clause1}(X) \ \vee \ \mathsf{clause2}(X)) \end{aligned}$$

Figure 3: Expressing a power function (exponentiation) a^b (Definitions 3.2.1 and 3.2.5)

1. define syntactic sugar,
2. use it to express the clauses from Definition 3.2.1, and
3. write a *validity predicate* χ_{pow}, to express that pow encodes a partial power function.

Examples of matrices that do and do not satisfy the validity predicate χ_{pow} are in Examples 3.2.8 and 3.2.9.

Definition 3.2.6. Suppose $n \in \mathbb{N}_{\geq 1}$. Say that a $4 \times n$ matrix M **encodes a partial power function** when

$$\forall 1 \leq x \leq n. M(3,x) = M(1,x)^{M(2,x)}.$$

In words, M encodes a partial power function when for each column, the third entry of that column is equal to the first entry raised to the power of the second.

Lemma 3.2.7. *Suppose ς is a valuation (Definition 2.3.2(1)). Then*

- *if* $\vDash_\varsigma \chi_{\mathsf{pow}}$
- *then* $\varsigma(\mathsf{pow})$ *encodes a partial partial power function (Definition 3.2.6).*

Proof. We examine the clauses in Figure 3 and using Theorem 2.4.4 we check that they correctly encode the definition in Definition 3.2.1. □

Example 3.2.8. We illustrate some matrices that pass the validity check $\vDash_\varsigma \chi_{\mathsf{pow}}$ and so by Lemma 3.2.7 encode a partial power function:

$$\begin{pmatrix} 2 & 2 & 2 \\ 0 & 1 & 2 \\ 1 & 2 & 4 \\ 1 & 1 & 2 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 2 & 2 & 2 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \\ 2 & 3 & 3 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 2 & 2 & 2 & 2 \\ 0 & 2 & 1 & 0 \\ 1 & 4 & 2 & 1 \\ 2 & 3 & 1 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 3 & 2 & 3 & 2 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 3 & 2 \\ 1 & 1 & 1 & 2 \end{pmatrix}$$

Note how all of these matrices can be viewed as concrete implementations of the derivation-tree illustrated in Remark 3.2.2:

1. The leftmost matrix encodes the base case of a derivation that $2^2 = 4$ in column 1, the second step in column 2, and the third and final step of the derivation in column 3. The fourth entry in columns 2 and 3 contain a number which is a pointer to the previous step/column in the derivation; the fourth entry in column 1 also contains a number, but it is not used.
2. The columns do not need to appear in any particular order, so long as the pointers match up; the next matrix illustrates this (by putting the columns in the reverse order).
3. Columns can also be duplicated; the third matrix illustrates this.
4. The fourth (rightmost) matrix includes information from two computations; that $2^1 = 2$ and $3^1 = 3$.

EXAMPLE 3.2.9. The converse implication for Lemma 3.2.7 does not hold: it is possible for $\varsigma(\mathsf{pow})$ to encode a partial power function yet $\neg(\models_\varsigma \chi_{\mathsf{pow}})$. Take for example

$$\varsigma(\mathsf{pow}) = \begin{pmatrix} 2 \\ 2 \\ 4 \\ 2 \end{pmatrix}$$

This one-column matrix encodes a partial power function because $2^2 = 4$, but it fails the validity predicate because the entry in the fourth row does not point to a column showing that $2^1 = 2$, so this fails the last line of clause2 in Figure 3. We can think of it as corresponding to an *incomplete* derivation-tree:

$$\frac{????}{pow(2,2) = 4}\ \text{inductive step}$$

We consider another example:

$$\varsigma(\mathsf{pow}) = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 1 & 0 \\ 4 & 2 & 1 \\ 2 & 3 & 0 \end{pmatrix}$$

This matrix encodes a partial power function because $2^2 = 4$ and $2^1 = 2$ and $2^0 = 1$. Furthermore, the entry in the fourth row of the first column *does* point to a column showing that $2^1 = 2$, that the fourth row of the second column does point to a column showing that $2^0 = 1$. However, this fails the range check just for a technical reason: the entry in the fourth row of the third column is 0, which is not in the range 1 to 3.

REMARK 3.2.10. In *constructive logic*, something is only deemed 'true' when we have a concrete witness for its truth.[13]

The way our encoding of logic works, as illustrated in Examples 3.2.8 and 3.2.9, feels constructive in this sense. It might help such a reader to think of matrices as follows: the matrices are used to encode deduplicated derivation-trees. Columns correspond to nodes, and we include index pointers to link these nodes together.

EXAMPLE 3.2.11. We take a moment to unpack the concrete polynomial that Figure 3 expresses.

Assume we have some valuation ς and write $pow = \varsigma(\mathsf{pow})$. Recall from Definition 2.3.2(1) that this is a $4 \times n$ matrix for some $n \geq 1$. Mirroring the sugar in Figure 3, we write:

$$\begin{aligned} a &= intrplnt(pow_1) \\ b &= intrplnt(pow_2) \\ out &= intrplnt(pow_3) \\ rec &= intrplnt(pow_4) \end{aligned}$$

Assume that every entry in pow_4 is a valid index of a column in pow, so that pow passes the range check and

$$[\![0 < \mathsf{pow}_4 < len(\mathsf{pow}) + 1]\!]_\varsigma = 0$$

— meaning that this is 'true'.

Then $[\![\chi_{\mathsf{pow}}]\!]_\varsigma$ from Figure 3 expands as follows — where, as per Figure 2, $\bigwedge$ maps to $+$, $\bigvee$ maps

[13] For example, in classical logic $P \Longrightarrow Q$ is equivalent to $\neg P \vee Q$. Constructively, $P \Longrightarrow Q$ is a concrete method for turning a witness to the truth of P, into a witness to the truth of Q.

to $*$, $\forall$ maps to $\sum$, and $t = t'$ maps to $(t - t')^2$:

$$\begin{aligned}\chi_{\mathsf{pow}} &= 1 \leq \mathsf{pow}_4 \leq \mathsf{len}(\mathsf{pow}) \wedge \\ &\quad \forall X_{\mathsf{pow}}.(\mathsf{clause1}(X) \vee \mathsf{clause2}(X))\end{aligned}$$

$$\begin{aligned}[\![\chi_{\mathsf{pow}}]\!]_\varsigma = 0 + \\ \textstyle\sum_{1\leq x\leq n} \Big(&\big(a(x)^2 + (out(x){-}1)^2\big) * \\ &\big((a(x){-}a(rec(x)))^2 + \\ &(b(x){-}(b(rec(x)){+}1))^2 + \\ &(out(x) - a(x){*}out(rec(x)))^2\big)\Big)\end{aligned}$$

If this evaluates to zero, then pow really does encode a partial power function.[14]

Remark 3.2.12 (Analysis of polynomial degree). Examining the polynomial in Example 3.2.11, we can make some comments on its degree:

1. Range checks correspond to constant numbers (0 if true, 1 if false). This makes range checks 'free', in the sense that they contribute nothing to the degree of the polynomial.
2. $\wedge$ and $\forall$ corresponds to $+$. This means that conjunction and universal quantification are 'free', in the sense that they do not increase the degree of a polynomial beyond the degrees of the components.
3. $\vee$ corresponds to $*$. This is linearly expensive, in the sense that the degree is the sum of the degrees of the components.
4. Polynomial instantiation, as triggered e.g. by the recursive call $out(rec(x))$ above, is multiplicatively expensive: it multiplies degrees of the polynomials involved. For example, if we set $P(X) = X^2$ and $Q(X) = X^3$ then $P(Q)(X) = (X^3)^2 = X^6$.
5. The length of the matrix is linearly expensive in the degree of the interpolating polynomial: as per Definition 2.1.4(2), to interpolate a vector of length n requires a polynomial of degree at most n-1.

For brevity write $n = len(pow)$; so this is the number of columns in pow and is (one plus) the degree of the interpolants a, b, out, and rec. Checking the polynomial, we see that its degree is $O(n^2)$, coming from the component $(a * out(rec))^2$ on the far right-hand side, and in particular from the polynomial instantiation $out(rec)$. Now for pow as specified in Definition 3.2.1, n (the number of columns) is equal to b, so the *degree order* of this specification, by which we mean the degree of the polynomials it produces, is $O(b^2)$.

Remark 3.2.13. Just because a polynomial has high degree does not necessarily make it complicated nor expensive to compute. For instance, X^{100} had degree 100, but 10^{100} is not particularly expensive to compute, and the polynomial has only one root up to multiplicities, so in the logical sense mentioned in Remark 2.3.5(6) it is equivalent to X^2.[15]

Degree order as a complexity metric is its own thing.

Remark 3.2.14. Continuing Remark 3.2.12, we can also try to keep n as small as possible, which amounts to minimising the number of calls to pow. As is well-known, a more efficient specification of a^b is this:

$$\begin{aligned}pow(a, 0) &= 1 \\ pow(a, 2 * b) &= pow(a, b) * pow(a, b) \\ pow(a, 2 * b + 1) &= a * pow(a, b) * pow(a, b)\end{aligned}$$

Logically, this leads to smaller derivations, and algorithmically it yields a runtime that is logarithmic in b rather than linear (parametrically over the complexity of our multiplication algorithm).

[14] We consider a succinct check of this using the Schwartz-Zippel lemma later, in Lemma 5.1.6.

[15] It might be possible to efficiently 'renormalise' our polynomials to eliminate repeated roots, using formal derivatives and polynomial long division. Indeed, what really matters to us is not the degree of the polynomial itself, but how many distinct rational roots it has. But for now, just looking a the degree of the polynomial will suffice.

$$\chi_{\mathsf{sqrt}} = \quad 0 \leq \mathsf{sqrt}_2 \wedge \\ \forall X_{\mathsf{sqrt}}.\mathsf{sqrt}_2(X) * \mathsf{sqrt}_2(X) = \mathsf{sqrt}_1(X)$$

Figure 4: Expressing a positive square root function $\sqrt{a}$ (Definitions 3.3.1 and 3.3.2)

We could translate this into our logic as we did in Definition 3.2.5, and this would yield an implementation of degree order $O(log(b)^2)$. More on efficiency in Remark 3.4.6(2) and Section 5.

3.3 Expressing a square root function

The square root $\sqrt{a}$ is a simple example which nicely illustrates that validity predicates are *assertions*, not computations. In particular, we can verify that a matrix represents a partial square root function; just square the claimed root (see Definition 3.3.3). We do not have to design a square root algorithm ourselves. This is an elementary point, which may be familiar to readers familiar with succinct proofs, but we spell out the details:

Definition 3.3.1. Recall that for $a \in \mathbb{Q}$, the **(positive) square root function** $\sqrt{a}$ is specified by

$$(\sqrt{a})^2 = a \wedge \sqrt{a} \geq 0.$$

Definition 3.3.2. Assume a matrix variable symbol sqrt with arity $arity(\mathsf{sqrt}) = 2$. In Figure 4 we use our logic to express a validity predicate χ_{sqrt} that sqrt encodes a partial positive square root function, such that:

1. Row 1 (values of $\mathsf{sqrt}_1(X)$) stores input values a.
2. Row 2 (values of $\mathsf{sqrt}_2(X)$) stores output values $\sqrt{a} \geq 0$.

Definition 3.3.3. Suppose $n \in \mathbb{N}_{\geq 1}$. Say that a $2 \times n$ matrix M **encodes a partial positive square root function** when

$$\forall 1 \leq x \leq n. 0 \leq M(2,x) \wedge M(2,x)^2 = M(1,x).$$

Lemma 3.3.4. *Suppose ς is a valuation (Definition 2.3.2(1)). Then the following are equivalent:*

1. $\vDash_\varsigma \chi_{\mathsf{sqrt}}$.
2. $\varsigma(\mathsf{sqrt})$ *encodes a partial positive square root function, as per Definition 3.3.3.*

Proof. We examine χ_{sqrt} in Figure 4 and see that it correctly encodes the definition in Definition 3.5.1. □

Remark 3.3.5. There are no pointers involved in Lemma 3.3.4, so we get a logical equivalence. Contrast with Lemma 3.2.7 and Examples 3.2.8 and 3.2.9 for pow.

Remark 3.3.6. The point made by Definition 3.3.3, Figure 4, and Lemma 3.3.4 is not that positive square roots are particularly interesting.[16] What matters is that actually *computing* $\sqrt{a}$ is done by whoever provides ς (i.e. the 'prover'). Intuitively, our logic is about *verification*. Thus, we assume a ς is provided, and we verify that it satisfies whatever validity predicates have been claimed of it.

3.4 Expressing Fibonacci

The Fibonacci function is a classic example in a certain kind of undergraduate course. We consider it in our system. It is a simple example, though nontrivial, and as we discuss in Remark 3.4.6 it raises some useful points:

[16] They are, but not in ways that matter to us here.

$$\begin{array}{lll}
\mathsf{fib}_1(t) \text{ sugars to } \mathsf{a}(t) & & \mathsf{fib}_2(t) \text{ sugars to } \mathsf{out}(t) \\
\mathsf{fib}_3(t) \text{ sugars to } \mathsf{rec1}(t) & & \mathsf{fib}_4(t) \text{ sugars to } \mathsf{rec2}(t)
\end{array}$$

$$\begin{array}{rl}
\mathsf{clause1}(X) = & (\mathsf{a}(X) = 0) \wedge (\mathsf{out}(X) = 1) \\
\mathsf{clause2}(X) = & (\mathsf{a}(X) = 1) \wedge (\mathsf{out}(X) = 1) \\
\mathsf{clause3}(X) = & \mathsf{a}(X) = \mathsf{a}(\mathsf{rec1}(X)) + 2 \quad \wedge \\
& \mathsf{a}(\mathsf{rec2}(X)) = \mathsf{a}(\mathsf{rec1}(X)) + 1 \quad \wedge \\
& \mathsf{out}(X) = \mathsf{out}(\mathsf{rec1}(X)) + \mathsf{out}(\mathsf{rec2}(X)) \\
\chi_{\mathsf{fib}} = & 1 \leq \mathsf{rec1} \leq \mathsf{len}(\mathsf{fib}) \ \wedge \ 1 \leq \mathsf{rec2} \leq \mathsf{len}(\mathsf{fib}) \wedge \\
& \forall X_{\mathsf{fib}}.(\mathsf{clause1}(X) \ \vee \ \mathsf{clause2}(X) \ \vee \ \mathsf{clause3}(X))
\end{array}$$

Figure 5: Expressing the Fibonacci function $fib(a)$ (Definitions 3.4.1 and 3.4.4)

DEFINITION 3.4.1. Recall that the Fibonacci sequence $fib(a)$ for $a \in \mathbb{N}_{\geq 0}$ is defined by:

$$\begin{array}{c}
fib(0) = 0 \\
fib(1) = 1 \\
fib(a+2) = fib(a) + fib(a+1)
\end{array}$$

DEFINITION 3.4.2. Assume a matrix variable symbol fib with arity $arity(\mathsf{fib}) = 3$. We can think of this as a three-row matrix.

We will use our logic to express a validity predicate χ_{fib} that fib encodes the Fibonacci function, such that:

1. Row 1 (values of $\mathsf{fib}_1(X)$) stores input values a.
2. Row 2 (values of $\mathsf{fib}_2(X)$) stores result values $fib(a)$.
3. Row 3 (values of $\mathsf{fib}_3(X)$) stores an index of a column in fib, corresponding to the first recursive call to fib in clause 3 of Definition 3.4.1.
4. Row 4 (values of $\mathsf{fib}_3(X)$) stores an index of a column in fib, corresponding to the second recursive call to fib in clause 3 of Definition 3.4.1.

DEFINITION 3.4.3. Suppose $n \in \mathbb{N}_{\geq 1}$. Say that a $4 \times n$ matrix M **encodes a partial Fibonacci function** when

$$\forall 1 \leq x \leq n. M(2,x) = fib(M(1,x)).$$

DEFINITION 3.4.4. In Figure 5 we

1. define syntactic sugar,
2. use it to express the clauses from Definition 3.4.1, and
3. write a validity predicate χ_{fib} to express that fib encodes a partial Fibonacci function.

LEMMA 3.4.5. *Suppose ς is a valuation (Definition 2.3.2(1)). Then*

1. *if $\vDash_\varsigma \chi_{\mathsf{fib}}$,*
2. *then $\varsigma(\mathsf{fib})$ encodes a partial Fibonacci function, as per Definition 3.4.3.*

Proof. We examine the clauses in Figure 5 and see that they correctly encode the definition in Definition 3.4.1. The range checks $1 \leq \mathsf{fib}_3 \leq \mathsf{len}(\mathsf{fib})$ and $1 \leq \mathsf{fib}_4 \leq \mathsf{len}(\mathsf{fib})$ are there to exclude out-of-bounds pointer errors. □

REMARK 3.4.6. There are a few interesting observations to make about Definition 3.4.1 and Figure 5.

1. It is not the case that $\varsigma(\mathsf{fib})$ has to contain the entries of the Fibonacci sequence *in order*. Each column has to contain a value a, the number $fib(a)$, and (if $a \notin \{0,1\}$) it contains pointers to two columns representing 'recursive calls'. Where those recursive calls are located inside $\varsigma(\mathsf{fib})$ does not really matter.

2. We put 'recursive calls' in scare quotes because these are not recursive calls to a computation.[17]
   Definition 3.4.1 is famous in undergraduate courses because, if translated directly to code, it is very inefficient; each call to *fib* creates two sub-calls, thus giving the program exponential complexity.
   A standard solution is to memoise the computation; store the results of previous calls and replace them with lookups if the function is called again on the same input.
   Figure 5 is already *naturally memoised*. Indeed, it is nothing more than a lookup-table. As we noted in Remark 3.3.6, our logic is *verifying* (not computing!) a ς.[18]
   So the complexity of the program implicit in Definition 3.4.1 is actually linear (not exponential), because our semantics is in some sense inherently memoised.

REMARK 3.4.7 (Fast Fibonacci and algebraic extensions of $\mathbb{Q}$). Optimised methods for computing Fibonacci numbers exist. See for instance this webpage (permalink) and especially this webpage (permalink), which uses the Binet formula — a paper on this topic exists [Das18], however, the webpage incudes significant additional optimisations.

This is interesting because The Binet formula uses $\sqrt{5}$, which is irrational and so is not in $\mathbb{Q}$, which is the field we use in this paper. So what this example illustrates is that $\mathbb{Q}$ may not be enough: we may wish to talk about roots of polynomials that are not rationals, such as $x^2 = 5$, and if so, we might need to consider algebraic extensions, such as $\mathbb{Q}(\sqrt{5})$. Technically, this should not be a problem.

3.5 Expressing Ackermann's function

Ackermann's function is another classic example. We check how it comes out in our framework:

DEFINITION 3.5.1. Recall Ackermann's function $Ack(a,b)$ for $a,b \in \mathbb{N}_{\geq 0}$, defined by:

$$\begin{array}{rl} Ack(0,b) &= b+1 \\ Ack(a+1,0) &= Ack(a,1) \\ Ack(a+1,b+1) &= Ack(a,Ack(a+1,b)) \end{array}$$

DEFINITION 3.5.2. Assume a matrix variable symbol Ack with arity $arity(\mathsf{Ack}) = 5$. We can think of this as a five-row matrix.

We will use our logic to express a validity predicate χ_{Ack} that Ack encodes a (partial) Ackermann function, such that:

1. Row 1 (values of $\mathsf{Ack}_1(X)$) stores input values a.
2. Row 2 (values of $\mathsf{Ack}_2(X)$) stores input values b.
3. Row 3 (values of $\mathsf{Ack}_3(X)$) stores result values $Ack(a,b)$.
4. Row 4 (values of $\mathsf{Ack}_4(X)$) stores an index of a column in the Ack matrix, corresponding to the first recursive call to Ack in clauses 2 and 3 of Definition 3.5.1.
5. Row 5 (the values of $\mathsf{Ack}_5(X)$) stores an index of another column in the Ack matrix, corresponding to the second recursive call to Ack in clause 3 of Definition 3.5.1.

DEFINITION 3.5.3. Suppose $n \in \mathbb{N}_{\geq 1}$. Say that a $5 \times n$ matrix M **encodes a partial Ackermann's function** when

$$\forall 1 \leq x \leq n. M(3,x) = Ack(M(1,x), M(2,x)).$$

DEFINITION 3.5.4. In Figure 6 we

1. define syntactic sugar,

[17] Well ... it depends on your point of view. They are recursive calls, but in a computation that is a verification of another computation that we assume has already been made.

[18] A valuation ς could contain repeated columns, corresponding to repeated calls to the same function on the same inputs, but this would be an inefficient prover; the verifier is still efficient on that (needlessly complex) input. Example 3.2.8 includes some basic examples with duplicated columns; they could be removed, but beyond padding out the matrix they do no harm.

$$\mathsf{Ack}_1(t) \text{ sugars to } \mathsf{a}(t) \qquad \mathsf{Ack}_2(t) \text{ sugars to } \mathsf{b}(t)$$
$$\mathsf{Ack}_3(t) \text{ sugars to } \mathsf{out}(t) \qquad \mathsf{Ack}_4(t) \text{ sugars to } \mathsf{rec1}(t) \qquad \mathsf{Ack}_5(t) \text{ sugars to } \mathsf{rec2}(t)$$

$$\begin{aligned}
\mathsf{clause1}(X) = \quad & (\mathsf{a}(X) = 0) \wedge (\mathsf{out}(X) = \mathsf{b}(X) + 1) \\
\mathsf{clause2}(X) = \quad & \mathsf{a}(X) = \mathsf{a}(\mathsf{rec1}(X)) + 1 \wedge \\
& \mathsf{b}(X) = 0 \wedge \\
& \mathsf{out}(X) = \mathsf{out}(\mathsf{rec1}(X)) \\
\mathsf{clause3}(X) = \quad & \mathsf{a}(X) = \mathsf{a}(\mathsf{rec2}(X)) + 1 \wedge \\
& \mathsf{b}(X) = \mathsf{b}(\mathsf{rec1}(X)) + 1 \wedge \\
& \mathsf{out}(X) = \mathsf{out}(\mathsf{rec2}(X)) \wedge \\
& \mathsf{a}(\mathsf{rec1}(X)) = \mathsf{a}(\mathsf{rec2}(X)) + 1 \wedge \\
& \mathsf{b}(\mathsf{rec2}(X)) = \mathsf{out}(\mathsf{rec1}(X)) \\
\chi_{\mathsf{Ack}} = \quad & 1 \leq \mathsf{rec1} \leq \mathsf{len}(\mathsf{Ack}) \wedge 1 \leq \mathsf{rec2} \leq \mathsf{len}(\mathsf{Ack}) \wedge \\
& \forall X_{\mathsf{Ack}}.(\mathsf{clause1}(X) \vee \mathsf{clause2}(X) \vee \mathsf{clause3}(X))
\end{aligned}$$

Figure 6: Expressing Ackermann's function $Ack(a, b)$ (Definitions 3.5.1 and 3.5.4)

2. use it to express the clauses from Definition 3.5.1, and
3. write a validity predicate χ_{Ack} to express that Ack encodes a partial Ackermann function.

LEMMA 3.5.5. *Suppose ς is a valuation (Definition 2.3.2(1)). Then*

1. *if $\models_\varsigma \chi_{\mathsf{Ack}}$*
2. *then $\varsigma(\mathsf{Ack})$ encodes a partial Ackermann function, as per Definition 3.5.3.*

Proof. We examine the three clauses in Figure 6 and see that they correctly encode the definition in Definition 3.5.1. The range checks $1 \leq \mathsf{Ack}_4 \leq \mathsf{len}(\mathsf{Ack})$ and $1 \leq \mathsf{Ack}_5 \leq \mathsf{len}(\mathsf{Ack})$ are there to exclude out-of-bounds pointer errors. □

3.6 Bitwise representation

Unsigned 64-bit integers — the familiar `uint64` datatype — are a standard datatype of practical interest. It is straightforward to represent them:

DEFINITION 3.6.1. We express that t is equal to zero or one (i.e. that t represents a single bit of binary data) using a little syntactic sugar:

$$\mathsf{isBinary}(t) \quad \text{is sugar for} \quad t = 0 \vee t = 1.$$

Unpacking Figure 2, $\mathsf{isBinary}(t)$ corresponds to a polynomial $t^2 * (t-1)^2$.

DEFINITION 3.6.2. Assume a matrix variable symbol uint64 with $arity(\mathsf{uint64}) = 65$, which we can think of as representing a 65-row matrix.

1. Row 1 should store a whole number $0 \leq n < 2^{64}$, and
2. rows 2 to 65 should store its binary expansion (with smallest bit first).

Validity of uint64 is simply

$$\begin{aligned}
\chi_{\mathsf{uint64}} = \forall X_{\mathsf{uint64}}.\big(&\mathsf{uint64}_1(X) = \textstyle\sum_{0 \leq (i \in \mathbb{N}_{\geq 0}) < 64} 2^i * \mathsf{uint64}_{i+2}(X) \wedge \\
&\mathsf{isBinary}(\mathsf{uint64}_2(X)) \wedge \ldots \wedge \mathsf{isBinary}(\mathsf{uint64}_{65}(X))\big)
\end{aligned}$$

This just asserts that for each column in uint64, elements 2 to 65 of that vector do indeed store bits representing the binary representation of element 1.

LEMMA 3.6.3. *Suppose ς is a valuation (Definition 2.3.2(1)). Then the following are equivalent:*

1. $\models_\varsigma \chi_{\mathsf{uint64}}$ *for* χ_{uint64} *as defined in Definition 3.6.2.*
2. $\varsigma(\mathsf{uint64})$ *encodes a list of bitwise expansions, by which we mean precisely that*

$$\forall 1 \leq x \leq len(\varsigma(\mathsf{uint64})).\ \varsigma(\mathsf{uint64})_1(x) = \sum_{0 \leq (i \in \mathbb{N}_{\geq 0}) < 64} 2^i * \varsigma(\mathsf{uint64})_{i+2}(x).$$

Proof. We examine χ_{uint64} in Definition 3.6.2 and using Theorem 2.4.4 and basic arithmetic see that this is precisely what it encodes. Note that there are no range checks, pointers, or recursive calls involved; this is just straightforward arithmetic. Note also that 2^i is in the predicate above is just a number (not a power of X), so has degree zero in X. □

REMARK 3.6.4. We get a decimal representation just by generalising isBinary to isDecimal:

$$\mathsf{isDecimal}(t) \quad \text{is sugar for} \quad t = 0 \mathbin{\mathsf{V}} t = 1 \mathbin{\mathsf{V}} \ldots \mathbin{\mathsf{V}} t = 9.$$

and use a matrix variable symbol $\mathsf{uint64}^{\text{base }10}$ with validity predicate

$$\begin{array}{rcl}\mathsf{clause}(X) & = & \big(\mathsf{uint64}_1^{\text{base }10}(X) = \sum_{0 \leq (i \in \mathbb{N}_{\geq 0}) < 64} 10^i * \mathsf{uint64}_{i+2}^{\text{base }10}(X) \mathbin{\mathsf{\wedge}} \\ & & \qquad \mathsf{isDecimal}(\mathsf{uint64}_2^{\text{base }10}(X)) \mathbin{\mathsf{\wedge}} \ldots \mathbin{\mathsf{\wedge}} \mathsf{isDecimal}(\mathsf{uint64}_{65}^{\text{base }10}(X))\big).\end{array}$$

We can also nest representations and for example represent a number in the range 0 to $(2^{64})^{64}$ by taking its 64-digit expansion base 2^{64}.

3.7 Range check

REMARK 3.7.1. The clause for the range check $<$ in Figure 2 is very simple — just check that the relevant row in the matrix satisfies the required bound, and return 0 if it is satisfied and 1 if it is not. However, later on in Subsection 5.1 it will be convenient to express $<$ inside the language *without* $<$.

In particular, we see from our examples — such as those in Figures 3 and 6 — that we typically want to prove range checks that have an *upper and lower* bound, having this form:

$$1 \leq \mathsf{C}_i \leq \mathsf{len}(\mathsf{C}).$$

We can translate predicates of this form into ones that does not mention $<$, as follows:

DEFINITION 3.7.2. Suppose $l, r \in \mathbb{Z}$ and $l \leq r$. Then define a polynomial $\mathsf{range}_{l,r}(X)$ by

$$\mathsf{range}_{l,r}(X) = (X = l) \mathbin{\mathsf{V}} (X = l{+}1) \mathbin{\mathsf{V}} \ldots \mathbin{\mathsf{V}} (X = r{-}1) \mathbin{\mathsf{V}} (X = r).$$

($\mathsf{range}_{l,r}$ is closely related to the *zeroes* polynomial which we define later in Definition 5.1.2. The difference is just that range is a predicate in our logic.)

REMARK 3.7.3. Range checks are special cases of *set membership* assertions. Succinct (and zero knowledge) set membership schemes are a field of research in their own right ([BCF+23] contains a recent survey). So: the schema in Definition 3.7.2 may not be optimal and in a practical implementation of these ideas we might wish to improve it further, but for proof-of-concept Definition 3.7.2 will serve our purposes perfectly well, and it has the benefit of being very simple. Lemma 3.7.4 says that it does what we need it to do:

LEMMA 3.7.4. *Suppose ς is a valuation. Then the following are equivalent:*

1. $\models_\varsigma 1 \leq \mathsf{C}_i \leq \mathsf{len}(\mathsf{C})$.
2. $\models_\varsigma \mathsf{\forall} X_{\mathsf{C}}.\mathsf{range}_{1,\mathsf{len}(\mathsf{C})}(C_i(X))$.

Proof. $\varsigma(\mathsf{C})$ is by Definition 2.3.2(1) an integer matrix, and the polynomial $[\![\mathsf{range}_{l,r}]\!]$ by construction in Definition 3.7.2 and Figure 2 has zeroes precisely at the integers between l and r inclusive, which by Notation 2.2.7 and Figure 2 is precisely what $1 \leq \mathsf{C}_i \leq \mathsf{len}(\mathsf{C})$ checks. □

REMARK 3.7.5. If $\varsigma(\mathsf{C})$ is a very long matrix then $[\![\mathsf{range}_{1,\mathsf{len}(\mathsf{C})}(C_i(X))]\!]$ might be a large (high-degree) polynomial. But, in the context of the *succinct proofs* that we will consider later in Subsection 5.1, this may be is a worthwhile tradeoff.

REMARK 3.7.6. We can also exploit uint64 from Subsection 3.6 to do a simple range check. Assume a matrix variable symbol range64 with $arity(\mathsf{range64}) = 2$, and with validity condition

$$\mathsf{clause}(X) \;=\; 1 \leq \mathsf{range64}_2 \leq \mathsf{len}(\mathsf{uint}) \;\wedge\; \big(\mathsf{range64}_1(X) = \mathsf{uint}_2(\mathsf{range64}_2(X))\big).$$

Intuitively, we can think of the above as follows:

1. $\mathsf{range64}_1(X)$ holds some number n.
2. $\mathsf{range64}_2(X)$ holds a pointer to a proof in uint64 that n has a 64-bit representation.
3. The pointers in $\mathsf{range64}_2$ must be in-bound for the addressable range in uint; i.e. they must be numbers between 1 and $\mathsf{len}(\mathsf{uint64})$.

The above can only happen if $0 \leq n < 2^{64}$, so this all amounts to doing range checks.

3.8 Iteration

DEFINITION 3.8.1. Recall that if $f : \mathbb{Z} \to \mathbb{Z}$ is a function and $a \in \mathbb{N}_{\geq 0}$, then the **$a$-fold iteration** $iter_f(a) : \mathbb{Z} \to \mathbb{Z}$ of f is the function that inputs $b \in \mathbb{Z}$ and returns $(\overbrace{f \circ \cdots \circ f}^{a \text{ times}})(b)$.

This can be defined by:

$$iter_f(0)(b) = b$$
$$iter_f(a+1)(b) = f(iter_f(a)(b))$$

DEFINITION 3.8.2. Assume a binary propositional constant symbol f, with arity $arity(\mathsf{f}) = 2$, along with a validity predicate χ_{f} which expresses that $\mathsf{f}_3(X) = f(\mathsf{f}_1(X), \mathsf{f}_2(X))$. We can think of this intuitively as a matrix of columns all having the form $(a, b, f(a, b))$, for various values of a and b. There is no obstacle to having more rows, as we did for instance in Figure 6; we take three just for simplicity.

We now assume a binary propositional constant iter_f with arity $arity(\mathsf{iter}_f) = 5$. In Figure 7 we use our logic to express that iter_f encodes an a-fold iteration of f on an input value b, such that:

1. Row 1 (values of $\mathsf{iter}_{f,1}(X)$) stores the number of iterations a.
2. Row 2 (values of $\mathsf{iter}_{f,2}(X)$) stores input values b.
3. Row 3 (values of $\mathsf{iter}_{f,3}(X)$) stores result values $iter_f(a)(b)$.
4. Row 4 (values of $\mathsf{iter}_{f,4}(X)$) stores pointers to recursive calls to iter_f, as required in clause 2 of Definition 3.8.1 above.
5. Row 5 (values of $\mathsf{iter}_f(5, X)$) stores pointers to calls to f, corresponding to the call to f in clause 2 of Definition 3.8.1 above.

DEFINITION 3.8.3. Suppose $f : \mathbb{Z} \to \mathbb{Z}$ is a unary function and suppose $n \in \mathbb{N}_{\geq 1}$. Say that a $5 \times n$ matrix M **encodes a partial iteration of** f when

$$\forall 1 \leq x \leq n. M(3, x) = f^{M(1,x)}(M(2, x)).$$

LEMMA 3.8.4. *Suppose ς is a valuation (Definition 2.3.2(1)). Then*

1. *if $\vDash_\varsigma iterfValid$*
2. *then $\varsigma(\mathsf{iterf})$ encodes a partial iteration of f, as per Definition 3.8.3.*

$\mathsf{iter}_{f,1}(t)$ sugars to $\mathsf{a}(t)$ $\quad$ $\mathsf{iter}_{f,2}(t)$ sugars to $\mathsf{b}(t)$
$\mathsf{iter}_{f,3}(t)$ sugars to $\mathsf{out}(t)$ $\quad$ $\mathsf{iter}_{f,4}(t)$ sugars to $\mathsf{rec}(t)$ $\quad$ $\mathsf{iter}_{f,5}(t)$ sugars to $\mathsf{callf}(t)$

$$\begin{aligned}
\mathsf{clause1}(X) = \quad & (\mathsf{a}(X) = 0) \wedge (\mathsf{out}(X) = \mathsf{b}(X)) \\
\mathsf{clause2}(X) = \quad & \mathsf{a}(X) = \mathsf{a}(\mathsf{rec}(X)) + 1 \wedge \\
& \mathsf{b}(X) = \mathsf{b}(\mathsf{rec}(X)) \wedge \\
& \mathsf{out}(X) = \mathsf{f}_2(\mathsf{callf}(X)) \wedge \\
& \mathsf{f}_1(\mathsf{callf}(X)) = \mathsf{out}(\mathsf{rec}(X)) \\
\chi_{\mathsf{iter}_f} = \quad & 1 \leq \mathsf{rec} \leq \mathsf{len}(\mathsf{iter}) \ \wedge \ 1 \leq \mathsf{callf} \leq \mathsf{len}(\mathsf{f}) \wedge \\
& \forall X_{\mathsf{iter}_f}.(\mathsf{clause1}(X) \ \vee \ \mathsf{clause2}(X))
\end{aligned}$$

Figure 7: Expressing $iter_f(a)(b)$; a-fold application of f to b (Definitions 3.8.1 and 3.8.2)

Proof. We examine the clauses in Figure 7 and using Theorem 2.4.4 see that they do indeed encode the definition in Definition 3.8.1. The bound checks $1 \leq \mathsf{iter}_4 \leq \mathsf{len}(\mathsf{iter})$ and $1 \leq \mathsf{iter}_5 \leq \mathsf{len}(\mathsf{f})$ exclude out-of-bounds pointer errors. □

REMARK 3.8.5. We see no inherent barrier to mutual inductive definitions. We just declare several matrix variable symbols and engineer pointers between them for the (mutual) inductive calls.

3.9 SK combinators

DEFINITION 3.9.1.

1. **Combinator expressions** are a formal syntax defined by:

$$t ::= S \mid K \mid tt.$$

   So a combinator is either an S, a K, or an **application** of a combinator term to a combinator term.

2. A big-step evaluation relation for combinators is as follows:[19]

$$\frac{}{K \Downarrow K}\,(\mathbf{Ki}) \qquad \frac{}{S \Downarrow S}\,(\mathbf{Si}) \qquad \frac{s \Downarrow s'}{(Ks)t \Downarrow s'}\,(\mathbf{Ke}) \qquad \frac{su \Downarrow s' \ \ tu \Downarrow t' \ \ s't' \Downarrow v}{((Ss)t)u \Downarrow v}\,(\mathbf{Se})$$

REMARK 3.9.2. Combinator expressions and their evaluation form a Turing-complete model of computation (see [Bim20] or [Bar84, Chapter 7].[20]

So, to specify combinators inside our logic is to build a Turing-complete model of computation inside it, which demonstrates that (at least in principle) it has a certain computational content.

We start with a trick that will make our encoding a little more compact:

DEFINITION 3.9.3.

1. The **Cantor pairing function** is a well-known way of bijecting $\mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$ with $\mathbb{N}_{\geq 0}$. It has the characteristic that it can be expressed as a polynomial over $\mathbb{Q}$ (indeed, this is the only quadratic polynomial pairing function) [Nat15]:[21]

$$\langle x, y\rangle' = \frac{(x+y)*(x+y+1)}{2} + x = \frac{x^2 + 3*x + 2*x*y + y + y^2}{2}. \qquad (1)$$

[19] The 'i' in the name ($\mathbf{Ki}$) and ($\mathbf{Si}$) is short for 'introduction', and 'e' in the name ($\mathbf{Ke}$) and ($\mathbf{Se}$) is short for 'elimination'. These are just names we gave the rules.

[20] A quite clear treatment is available online here (permalink).

[21] Clear and accessible accounts of the Cantor pairing function are also on Wikipedia here and here.

$$
\begin{array}{lll}
0 \text{ sugars to } \mathsf{S} & 1 \text{ sugars to } \mathsf{K} & \\
\mathsf{eval}_1(t) \text{ sugars to } \mathsf{lhs}(t) & \mathsf{eval}_2(t) \text{ sugars to } \mathsf{rhs}(t) & \mathsf{eval}_3(t) \text{ sugars to } \mathsf{s}(t) \\
\mathsf{eval}_4(t) \text{ sugars to } \mathsf{t}(t) & \mathsf{eval}_5(t) \text{ sugars to } \mathsf{u}(t) & \\
\mathsf{eval}_6(t) \text{ sugars to } \mathsf{rec1}(t) & \mathsf{eval}_7(t) \text{ sugars to } \mathsf{rec2}(t) & \mathsf{eval}_8(t) \text{ sugars to } \mathsf{rec3}(t)
\end{array}
$$

$$
\begin{aligned}
\mathsf{clause1}(X) = \quad & \mathsf{lhs}(X) = \mathsf{S} \ \wedge \ \mathsf{rhs}(X) = \mathsf{S} \\
\mathsf{clause2}(X) = \quad & \mathsf{lhs}(X) = \mathsf{K} \ \wedge \ \mathsf{rhs}(X) = \mathsf{K} \\
\mathsf{clause3}(X) = \quad & \mathsf{lhs}(X) = \langle\langle \mathsf{K}, \mathsf{lhs}(\mathsf{s}(X))\rangle, \mathsf{t}(X)\rangle \ \wedge \\
& \mathsf{rhs}(X) = \mathsf{rhs}(\mathsf{rec1}(X)) \ \wedge \\
& \mathsf{lhs}(\mathsf{rec1}(X)) = \mathsf{s}(X) \\
\mathsf{clause4}(X) = \quad & \mathsf{lhs}(X) = \langle\langle\langle \mathsf{S}, \mathsf{s}(X)\rangle, \mathsf{t}(X)\rangle, \mathsf{u}(X)\rangle \ \wedge \\
& \mathsf{rhs}(X) = \mathsf{rhs}(\mathsf{rec3}(X)) \ \wedge \\
& \mathsf{lhs}(\mathsf{rec1}(X)) = \langle \mathsf{s}(X), \mathsf{t}(X)\rangle \ \wedge \\
& \mathsf{lhs}(\mathsf{rec2}(X)) = \langle \mathsf{s}(X), \mathsf{u}(X)\rangle \ \wedge \\
& \mathsf{lhs}(\mathsf{rec3}(X)) = \langle \mathsf{rhs}(\mathsf{rec1}(X)), \mathsf{rhs}(\mathsf{rec2}(X))\rangle \\
\chi_{SK} = \quad & 1 \leq \mathsf{rec1} \leq \mathsf{len}(\mathsf{evl}) \ \wedge \ 1 \leq \mathsf{rec2} \leq \mathsf{len}(\mathsf{evl}) \ \wedge \ 1 \leq \mathsf{rec3} \leq \mathsf{len}(\mathsf{evl}) \ \wedge \\
& \forall X_{\mathsf{evl}}.(\mathsf{clause1}(X) \ \vee \ \mathsf{clause2}(X) \ \vee \ \mathsf{clause3}(X) \ \vee \ \mathsf{clause4}(X))
\end{aligned}
$$

Figure 8: Expressing big-step evaluation of combinator expressions (Definitions 3.9.1 and 3.9.5)

2. The pairing function $\langle x, y\rangle'$ above is not quite what we need, because we want to reserve 0 to represent the combinator S and 1 to represent the combinator K. Accordingly, we adjust it with a two-element offset:
$$\langle x, y\rangle = \langle x, y\rangle' + 2.$$

3. We then define a *Gödel encoding*[22] from the syntax of combinator expressions (Definition 3.9.1(1)) to $\mathbb{N}$, which is an injection defined by:
$$gdl(S) = 0 \quad gdl(K) = 1 \quad gdl(tt') = \langle gdl(t), gdl(t')\rangle.$$
It is routine to prove that this injection is actually a bijection between combinator expressions and the nonnegative natural numbers $\mathbb{N}_{\geq 0}$.

Remark 3.9.4. In Equation (1) in Definition 3.9.3, we divide by 2. If we want a polynomial with coefficients over $\mathbb{Z}$ we could just as well use $\langle x, y\rangle' * 2$, and retain integer coefficients throughout. We would lose bijectivity of the pairing function, but in the context of this prototype that would be fine too.

Definition 3.9.5. Assume a matrix variable symbol evl with arity $arity(\mathsf{evl}) = 8$, which we can think of as storing an 8-row matrix. In Figure 8, we

1. define syntactic sugar,
2. use it to express the clauses from Definition 3.9.1, and
3. write a validity predicate χ_{SK} to express that evl encodes a partial combinator big-step reduction relation.

Remark 3.9.6. We read through the rows in evl one by one:

1. Row 1 represents the left-hand (unreduced) term.
2. Row 2 represents the right-hand (reduced) term.

[22] By which we mean an injective map from the syntax of a language to natural numbers. Such maps are so called because Gödel used one such in his 1931 paper on incompleteness results.

3. Row 3 represents a subterm s of the term on the left-hand side (as required in rules $(\mathbf{Ke})$ and $(\mathbf{Se})$).
4. Row 4 represents a subterm t (as required in rules $(\mathbf{Ke})$ and $(\mathbf{Se})$).
5. Row 5 represents a subterm u (as required in rule $(\mathbf{Se})$).
6. Rows 6, 7, and 8 represent pointers to proof-obligations above the line (= inductive function-calls) as may be required: none are required for $(\mathbf{Ki})$ and $(\mathbf{Si})$; one is required for $(\mathbf{Ke})$; and three are required for $(\mathbf{Se})$.

DEFINITION 3.9.7. Suppose $n \in \mathbb{N}_{\geq 1}$. Say that an $8 \times n$ matrix M **encodes a partial big-step combinator reduction relation** when

$$\forall 1 \leq x \leq n. gdl^{-1}(M(1,x)) \Downarrow gdl^{-1}(M(2,x)).$$

Above, the $\Downarrow$ relation is defined in Definition 3.9.1. The gdl function bijects combinator expressions with nonnegative natural numbers (Definition 3.9.3(3)).

LEMMA 3.9.8. *Suppose ς is a valuation (Definition 2.3.2(1)). Then*

1. *if $\models_\varsigma \chi_{SK}$,*
2. *then $\varsigma(\mathsf{evl})$ encodes a partial combinator big-step reduction relation, as per Definition 3.9.7.*

Proof. We examine the clauses in Figure 8 and using Theorem 2.4.4 see that they do indeed encode the definition in Definition 3.9.1. This is a little bit fiddly because of the Gödel encoding of the combinator syntax, and the fact that there are four derivation rules, and one of them has three subderivations, so it is a challenge to not make a typo or miss out a bit of a rule — but conceptually this is a straightforward translation from one logical syntax into another. □

REMARK 3.9.9. We use the Cantor pairing function to build a simple Gödel encoding that packs a combinator term into a single number. We might prefer instead to store a combinator term as a list or as a tree of symbols. This is possible: it would add complexity to the representation above and thus to its validity predicate, but would not change the fundamental nature of what we are doing.

4 From range check to general recursion

4.1 Simple functions obtainable directly from the range check

REMARK 4.1.1. A curious feature of our syntax in Figure 1 is the range check predicate $<$.

Our logic does not have a 'runtime' and a 'compile time' as such, but $<$ has the flavour of a static 'compile time' check, in the sense that (for example) given ς, $[\![vft < \mathsf{C}_i]\!]_\varsigma$ computes a number rather than a polynomial in X.[23]

In the examples above, we use range check to check for out-of-bounds pointer errors (see Examples 3.2.8 and 3.2.9 for concrete examples). But what is the range check really doing, and what is its full power? In this section we will examine this operator in more detail. We start by using the range check to specify some simple predicates and functions:

4.1.1 Strictly positive numbers pos

DEFINITION 4.1.2 (Strictly positive numbers). Assume a matrix variable symbol pos with $arity(\mathsf{pos}) = 1$ and validity predicate

$$\chi_{\mathsf{pos}}(X) = (0 < \mathsf{pos}_1).$$

LEMMA 4.1.3. *Suppose ς is a valuation. Then the following are equivalent:*

1. $\models_\varsigma \chi_{\mathsf{pos}}$.

[23] So we can still say that our semantics is polynomial, even though $vft < \mathsf{C}_i$ does not *look* like a polynomial expression: it evaluates to a number $[\![vft < \mathsf{C}_i]\!]_\varsigma$, and a number is trivially a polynomial.

2. $\varsigma(\mathsf{pos})$ *is a 1-row matrix (i.e. a vector) of strictly positive integers in* $\mathbb{N}_{>0}$.

Proof. This is exactly what the clause for range check in Figure 2 expresses, for $0 < \mathsf{pos}_1$. □

REMARK 4.1.4. We introduce a strict inequality $<$ rather than an inequality $\leq$ in Figure 1 so that we can express being *strictly positive* pos. If we took $\leq$ as primitive, it would be harder to express pos. This is because our logic does not include negation as primitive, so we cannot just express pos from a combination of $\leq$ and $\neq$.

4.1.2 The sign function $sign$ and its corollary functions

DEFINITION 4.1.5. We define functions

$$\begin{aligned} sign &: \mathbb{Q} \to \{\text{-}1, 0, 1\} \subseteq \mathbb{Q} \\ isZero &: \mathbb{Q} \to \{0, 1\} \\ neg &: \mathbb{Q} \to \{0, 1\} \\ neq &: (\mathbb{Q} \times \mathbb{Q}) \to \{0, 1\} \\ leq &: (\mathbb{Q} \times \mathbb{Q}) \to \{0, 1\} \end{aligned}$$

such that for $a, b \in \mathbb{Q}$:

$$sign(a) = \begin{cases} 0 & \text{if } a = 0 \\ 1 & \text{if } a > 0 \\ \text{-}1 & \text{if } a < 0 \end{cases}$$

$$isZero(a) = \begin{cases} 0 & \text{if } a = 0 \\ 1 & \text{if } a \neq 0 \end{cases} \qquad neg(a) = \begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{if } a \neq 0 \end{cases}$$

$$neq(a, b) = \begin{cases} 0 & \text{if } a \neq b \\ 1 & \text{if } a = b \end{cases} \qquad leq(a, b) = \begin{cases} 0 & \text{if } a \leq b \\ 1 & \text{if } a \not\leq b \end{cases}$$

REMARK 4.1.6. $sign$ is the fundamental entity in Definition 4.1.5, and $isZero$, neg, neq, and leq follow from it, in the sense that we can derive everything using polynomial expressions using $sign$:

$$\begin{aligned} isZero(a) &= sign(a)^2 \\ neg(a) &= 1 - isZero(a) = 1 - sign(a)^2 \\ neq(a, b) &= neg(a - b) = 1 - (sign(a - b))^2 \\ leq(a, b) &= sign(a - b) * (sign(a - b) - 1)/2 \end{aligned}$$

We can use pos from Definition 4.1.2 to express $sign$, $isZero$, neg, neq, and leq from Definition 4.1.5:

DEFINITION 4.1.7. Assume matrix variable symbols sign, isZero, and neg all with arity 3, so that $arity(\mathsf{sign}) = arity(\mathsf{isZero}) = arity(\mathsf{neg}) = 3$.

1. Sugar $\mathsf{sign}_1(X)$ to $\mathsf{sign.in}(X)$. This stores an input value a.
2. Sugar $\mathsf{sign}_2(X)$ to $\mathsf{sign.out}(X)$. This stores the corresponding output value $sign(a)$.
3. Sugar $\mathsf{sign}_3(X)$ to $\mathsf{sign.ptr}(X)$. This stores a pointer to a column in pos_1 (Definition 4.1.2) holding the absolute value of a.

We use similar sugar for isZero and neg. The validity predicates for sign, isZero and neg (using

pos) are:

$$\begin{aligned}\chi_{\mathsf{sign}}(X) = 1 \leq \mathsf{sign.ptr} \leq \mathsf{len}(\mathsf{pos}) \wedge \\ \forall X.\big(\ (\mathsf{sign.in}(X) = 0 \wedge \mathsf{sign.out}(X) = 0) \vee \\ (\mathsf{sign.in}(X) = \mathsf{pos}_1(\mathsf{sign.ptr}(X)) \wedge \mathsf{sign.out}(X) = 1) \vee \\ (((\text{-}1) * \mathsf{sign.in}(X)) = \mathsf{pos}_1(\mathsf{sign.ptr}(X)) \wedge \mathsf{sign.out}(X) = \text{-}1)\big)\end{aligned}$$

$$\begin{aligned}\chi_{\mathsf{isZero}}(X) = 1 \leq \mathsf{isZero.ptr} \leq \mathsf{len}(\mathsf{pos}) \wedge \\ \forall X.\big(\ (\mathsf{isZero.in}(X) = 0 \wedge \mathsf{isZero.out}(X) = 0) \vee \\ (\mathsf{isZero.in}(X)^2 = \mathsf{pos}_1(\mathsf{isZero.ptr}(X)) \wedge \mathsf{isZero.out}(X) = 1)\big)\end{aligned}$$

$$\begin{aligned}\chi_{\mathsf{neg}}(X) = 1 \leq \mathsf{neg.ptr} \leq \mathsf{len}(\mathsf{pos}) \wedge \\ \forall X.\big(\ (\mathsf{neg.in}(X) = 0 \wedge \mathsf{neg.out}(X) = 1) \vee \\ (\mathsf{neg.in}(X)^2 = \mathsf{pos}_1(\mathsf{neg.ptr}(X)) \wedge \mathsf{neg.out}(X) = 0)\big)\end{aligned}$$

Assume matrix variable symbols neq, and leq with arity 4, so that $arity(\mathsf{neq}) = arity(\mathsf{leq}) = 4$.

1. Sugar $\mathsf{neq}_1(X)$ to $\mathsf{neq.a}(X)$. This stores an input value a.
2. Sugar $\mathsf{neq}_2(X)$ to $\mathsf{neq.b}(X)$. This stores an input value b.
3. Sugar $\mathsf{neq}_3(X)$ to $\mathsf{neq.out}(X)$. This stores the corresponding output value $neq(a, b)$.
4. Sugar $\mathsf{neq}_4(X)$ to $\mathsf{neq.ptr}(X)$. This stores a pointer to a column in pos_1 (Definition 4.1.2).

We use similar sugar for leq. The validity predicates for neq and leq (using pos) are:

$$\begin{aligned}\chi_{\mathsf{neq}}(X) = 1 \leq \mathsf{neq.ptr} \leq \mathsf{len}(\mathsf{pos}) \wedge \\ \forall X.\big(\ ((\mathsf{neq.a}(X) - \mathsf{neq.b}(X)) = 0 \wedge \mathsf{neq.out}(X) = 1) \vee \\ ((\mathsf{neq.a}(X) - \mathsf{neq.b}(X))^2 = \mathsf{pos}_1(\mathsf{neq.ptr}(X)) \wedge \mathsf{neq.out}(X) = 0)\big)\end{aligned}$$

$$\begin{aligned}\chi_{\mathsf{leq}}(X) = 1 \leq \mathsf{leq.ptr} \leq \mathsf{len}(\mathsf{pos}) \wedge \\ \forall X.\big(\ ((\mathsf{leq.a}(X) - \mathsf{leq.b}(X)) = 0 \wedge \mathsf{leq.out}(X) = 0) \vee \\ ((\mathsf{leq.b}(X) - \mathsf{leq.a}(X)) = \mathsf{pos}_1(\mathsf{leq.ptr}(X)) \wedge \mathsf{leq.out}(X) = 0) \vee \\ ((\mathsf{leq.a}(X) - \mathsf{leq.b}(X)) = \mathsf{pos}_1(\mathsf{leq.ptr}(X)) \wedge \mathsf{leq.out}(X) = 1)\big)\end{aligned}$$

LEMMA 4.1.8. *Suppose $f \in \{sign, isZero, neg, neq, leq\}$ is one of the functions from Definition 4.1.5. Then the corresponding validity predicate χ_{f} from Definition 4.1.7 correctly encodes f, in the formal sense that — taking the corresponding $\mathsf{f} \in \{\mathsf{sign}, \mathsf{isZero}, \mathsf{neg}, \mathsf{neq}, \mathsf{leq}\}$ — for every valuation ς, if $\vDash_\varsigma \chi_{\mathsf{pos}}$ (Definition 4.1.2) and $\vDash_\varsigma \chi_{\mathsf{f}}$ then for every $1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{f}))$ we have*

$$\begin{aligned}\varsigma(\mathsf{sign})(2, x) &= sign(\varsigma(\mathsf{sign})(1, x)) \\ \varsigma(\mathsf{isZero})(2, x) &= isZero(\varsigma(\mathsf{isZero}(1, x))) \\ \varsigma(\mathsf{neg})(2, x) &= neg(\varsigma(\mathsf{neg})(1, x)) \\ \varsigma(\mathsf{neq})(3, x) &= neq(\varsigma(\mathsf{neq})(1, x), \varsigma(\mathsf{neq})(2, x)) \\ \varsigma(\mathsf{leq})(3, x) &= leq(\varsigma(\mathsf{leq})(1, x), \varsigma(\mathsf{leq})(2, x))\end{aligned}$$

Proof. The functions in Definition 4.1.5 are simple enough, and so are the validity predicates in Definition 4.1.7. We check the correspondence and use Theorem 2.4.4. □

REMARK 4.1.9.

1. There is some apparent overlap between $isZero$ from Definition 4.1.5 and the 'is equal to zero' function $\lambda x.[\![X = 0]\!](x) = \lambda x.x^2$ which we can build from the syntax and denotation in Figures 1 and 2.
   But these are different functions: with $isZero$, the output is either 0 or 1, whereas with $\lambda x.x^2$ the output may be any nonnegative value. For example:

$$isZero(2) = 1 \quad \text{whereas} \quad [\![2 = 0]\!] = 4,$$

as per Figure 2.
Knowing that the output is precisely equal to 0 or 1 is a stronger property, and it matters because (for example) we can then easily get the 'negation' predicate (mapping 0 to 1 and any nonzero value to 0) as $1 - isZero(a)$. We cannot do this with $\lambda x.[\![X = 0]\!](x) = \lambda x.x^2$.

2. neg is short for *negation*. It maps 0 (corresponding to 'truth') to 1, and any nonzero value (corresponding to 'false') to 0.
   This is a negation, and we have:
   (a) We have *excluded middle*, since $a * neg(a) = 0$ for every $a \in \mathbb{Q}$.[24]
   (b) $a + neg(a) \neq 0$ for every $a \in \mathbb{Q}$. This equals a if $a \neq 0$, and equals 1 if $a = 0$.[25]
   (c) Double negation $neg(neg(a))$ maps 0 to 0 and any nonzero element to 1.
   If we think of a truth-value $a \in \mathbb{Q}_{\geq 0}$ as being either 0 or a nonzero 'error value', then $neg(neg(a))$ throws away the precise error value and just retains the information whether a was true or false.

   neg is not unique with the properties above; e.g. neg' mapping a to $2 * neg(a)$ has similar properties. This is as expected and it just reflects that there are many false (= nonzero) values.

4.1.3 Back from pos to the range check

REMARK 4.1.10 (From pos back to $<$). We saw above how, in the presence of the rest of the logic, the range checks $vft < t$ and $t < vft$ can express predicates for strictly positive numbers (Definition 4.1.2), and a $sign$ function (Definition 4.1.5).

We take a moment to note that we can also go in the other direction, from pos to range checks. Let us for this Remark take a pos predicate as primitive in our syntax.

We can express the range check

$$vft < \mathsf{C}_i \quad \text{as} \quad \forall X_{\mathsf{C}}.pos(\mathsf{C}_i(X) - vft),$$

and

$$\mathsf{C}_i < vft \quad \text{as} \quad \forall X_{\mathsf{C}}.pos(vft - \mathsf{C}_i(X)).$$

Thus in some sense, $<$, pos, and also $sign$ are all doing the same thing.

REMARK 4.1.11. Continuing Remark 4.1.10, $<$ is also clearly *different* from pos and $sign$, because $vft < \mathsf{C}_i$ is variable-free and so has the flavour of a static check on the valuation ς, whereas $pos(x)$ and $sign(x)$ have the flavour of being predicates.

Yet, in the context of the rest of the system they have the same expressivity. What deeper structure is being manifested here?

We propose that at a deeper level, what these constructs do is introduce the power of a *knowledge modality*. This is a modal operator that internalises some notion of necessitation, truth, knowledge, or provability inside the logic itself [Ver24] — that something is not just true or false, but also known to be such, proved to be such, or necessarily such, etc.

A feature of the notion of 'false' in our logic is that it is modelled by being nonzero, and there are infinitely many nonzero numbers so we cannot explicitly enumerate over all the possible ways of being false within our logic.[26] What is significant about $<$, pos, and $sign$ is not so much what they detect (though this is of course relevant); what really matters is that they map the truth or falsity of what they measure, to a finite set of values ($\{0, 1\}$ or $\{0, 1, -1\}$), and this is where the effect of a knowledge modality comes from.

[24] Excluded middle is the property 'ϕ-or-not-ϕ', which when filtered through our polynomial semantics from Figure 2 renders as $a * neg(a) = 0$.

[25] This corresponds to 'ϕ-and-not-ϕ', which we expect to be false, i.e. nonzero.

[26] This would not be solved by switching to a finite field. In practical terms, any finite field large enough to be cryptographically secure is also large enough to be practically unenumerable. This is why cryptographic assurances work! Thus, modelling 'true' by a single value and 'false' by many values is not a bug of our semantics. Far from it; it reflects an essential feature of the cryptographic notion of proof.

What makes $<$ special, and better to use as a primitive than pos and $sign$, is that it still has a polynomial semantics — because (as we mentioned in Remark 4.1.1) $[\![vft < \mathsf{C}_i]\!]_\varsigma$ returns a number, which is a polynomial. More on this in future work.

4.2 Representing arbitrary functions in terms

REMARK 4.2.1. The term language in Figure 1 does not accommodate arbitrary functions $f : \mathbb{Q}^n \to \mathbb{Q}$. This is by design, since Figure 2 presents a *polynomial* semantics. But it would be nice if we could have our cake and eat it too: use arbitrary functions in our terms, and *still* get the goodness of polynomial expressions.

This is impossible in the general case, but we do not need the general case: a simple polynomial approximation is sufficient. So, we will use a matrix variable symbol f, with a validity predicate that expresses that f approximates the desired function f.

4.2.1 Enriched syntax and its compilation back down to unenriched syntax

Suppose we have the following scenario:

1. Suppose we have some function $f : \mathbb{Q} \to \mathbb{Q}$. (For simplicity we assume f is unary, but functions with more arguments can also be accommodated.)
2. Suppose we have represented f in our logic with
   - a matrix variable symbol f of arity $arity(\mathsf{f}) \geq 2$, and
   - a validity predicate χ_{f}, such that f_1 represents input values, and f_2 represents output values. In symbols, we write:

     $$\vDash_\varsigma \chi_{\mathsf{f}} \quad \Longrightarrow \quad \forall 1 \leq (x \in \mathbb{N}_{\geq 0}) \leq len(\varsigma(\mathsf{f})).\mathsf{f}_2(x) = f(\mathsf{f}_1(x)).$$

   Other rows may be present in f (see for example our implementation of $sign$ in Definition 4.1.7, which requires a third row of data). This is fine and will not affect our reasoning.
3. Suppose we have another matrix variable symbol P with $arity(\mathsf{P})$, and suppose P has a validity predicate χ_{P} that uses an **f-enriched term syntax**, which enriches the syntax from Figure 1 such that if t is an f-enriched term then $f(t)$ is an f-enriched term.
   There are no other restrictions on χ_{P}; it is just some predicate, but in the f-enriched syntax.
   The denotation of the f-enriched term $f(t)$, is f applied to the denotation of t. That is, we enrich Figure 2 with a clause

   $$[\![f(t)]\!]_\varsigma(x) = f([\![t]\!]_\varsigma(x)). \tag{2}$$

We want to transform χ_{P} into another validity predicate χ'_{P} such that

- χ'_{P} means the same thing as χ_{P} (in a sense we will make formal in Definition 4.2.3 and Proposition 4.2.4) but
- χ'_{P} is in the unenriched syntax, and it will replace calls to f with uses of the matrix variable symbol f.

Specifically, in Definition 4.2.3 we will show how to transform

- a matrix variable symbol P with arity $arity(\mathsf{P})$ and validity predicate χ_{P} that contains some innermost instance of $f(t)$,[27] into
- a matrix variable symbol P' with arity $arity(\mathsf{P}') = arity(\mathsf{P})+1$ and a validity predicate $\chi_{\mathsf{P}'}$ that does not contain this instance of $f(t)$.

Since syntax is finite, we can eliminate all instances of f from our validity predicate by repeating this transformation.

[27] 'Innermost' means that t does not itself contain some subterm of the form $f(t')$.

REMARK 4.2.2. The idea that motivates the transformation outlined in Definition 4.2.3 is straightforward: we replace calls to f with pointers to a matrix that stores a polynomial approximation with the same values at those calls. A polynomial can approximate a function at any finite number of points, so we know this can work; the rest is just a matter of getting the technicalities right.

DEFINITION 4.2.3 (The transformation). For clarity, we sugar $\mathsf{f}_1(t)$ to $\mathsf{f.in}(t)$ and $\mathsf{f}_2(t)$ to $\mathsf{f.out}(t)$.

1. Looking at the syntax in Figure 1, we see that the instance of $f(t)$ must occur inside some subpredicate $\phi = (t' \mathbin{=} t'')$ where $f(t)$ appears in t' or t''. We obtain $\chi_{\mathsf{P}'}$ from χ_{P} by replacing ϕ in χ_{P} with
$$(\mathsf{f.in}(\mathsf{P}'_{n+1}(X)) \mathbin{=} t) \ \wedge \ \phi[f(t) \mapsto \mathsf{f.out}(\mathsf{P}'_{n+1}(X))],$$
where $\phi[f(t) \mapsto \mathsf{f.out}(\mathsf{P}'_{n+1}(X))]$ denotes the predicate obtained by replacing any instances of $f(t)$ in ϕ (of which we assumed there is at least one) with $\mathsf{f.out}(\mathsf{P}'_{n+1}(X))$.
Using Theorem 2.4.4 we see that $\chi_{\mathsf{P}'}$ expresses (subject to range checks which we will add below) that the value in $\mathsf{P}'_{n+1}(X)$ points to a column in f that contains $(t, f(t))$.
2. We repeat the transformation above until there are no mentions of f left. The result is a matrix variable symbol $\mathsf{P}^{(n)}$ of arity $\mathit{arity}(\mathsf{P}^{(n)}) = \mathit{arity}(\mathsf{P}) + n$ (where n is the number of times we applied our transformation), and with validity predicate $\chi_{\mathsf{P}^{(n)}}$.
3. Now we just need to wrap this in range checks to prevent out-of-bounds pointer errors in the extra rows, so we set:
$$\chi'_{\mathsf{P}} = \chi_{\mathsf{P}^{(n)}} \wedge (1 \leq \mathsf{P}^{(n)}_{i+1} \leq \mathsf{len}(\mathsf{f})) \wedge \ldots \wedge (1 \leq \mathsf{P}^{(n)}_{i+n} \leq \mathsf{len}(\mathsf{f})).$$

PROPOSITION 4.2.4. *Suppose ς is a valuation such that $\vDash_\varsigma \chi_{\mathsf{f}}$. Then the following are equivalent:*

1. $\vDash_\varsigma \chi_{\mathsf{P}}$ *in the enriched syntax and denotation described above — in which $f(t)$ is a term if t is a term, and $[\![f(t)]\!]_\varsigma(x) = f([\![t]\!]_\varsigma(x))$ as per equation 2.*
2. $\vDash_\varsigma \chi'_{\mathsf{P}}$ *in the unenriched semantics.*

Proof. The argument is already in Definition 4.2.3: calls to f are replaced by range-checked pointers to columns in f. We have assumed that $\vDash_\varsigma \chi_{\mathsf{f}}$, and we have assumed that this implies that $\varsigma(\mathsf{f})_2(x) = f(\varsigma(\mathsf{f})_1(x))$ for every column in $\varsigma(\mathsf{f})$, so that a call to f in the enriched syntax is precisely equivalent to a lookup in $\varsigma(\mathsf{f})$ of a column starting with the denotation of that term. □

REMARK 4.2.5. Each step of the transformation in Definition 4.2.3(1) removes one innermost instance of $f(t)$ and increases the arity of the matrix variable symbol by 1 (modulo optimisations, e.g if $f(t)$ appears more than once). So we can say that, roughly speaking,

> each function-call to f costs an extra row in the validity matrix.

This is fine because it does not affect the width of the matrices, and so does not increase the degree of the interpolating polynomials.[28]

Overall, the process described above is not much different from what we do in ordinary mathematics, when we define e.g. Kuratowski pairs as $(x, y) = \{\{x\}, \{x, y\}\}$ and then proceed to use (x, y) directly; or when we define and use functions in any modern programming language and expect that these may be inlined, optimised, or compiled to a lower-level representation. In principle everything will get unwound to some more primitive syntax, so we know we are safe, but the higher-level syntax is more usable.

In a practical implementation of this logic, we might expect to provide functions as primitive, and let the compiler translate them to a lower-level representation as described above.

[28] Tall but narrow matrices are cheap; see Remark 6.2.1(2). Thus, a transformation that increases arity (and thus the height of matrices) but does not affect their width, is free, to a first approximation.

4.2.2 A simple application: predicate complementation

We conclude by bringing together what we have learned, to express negation as a macro. In more technical terminology: we will show that our logic is *complemented*.

Recall from Figures 1 and 2 that our syntax and semantics do not include predicate negation $\neg\phi$. This is because we have one 'true' truth-value 0, and many 'false' truth-values $\mathbb{Q} \setminus \{0\}$.

However, in the presence of the rest of the logic, the *range-check* $<$ allows us to implement negation as a macro operation. We saw a strong indication of this in neg and neg from Subsection 4.1.2. This negation acts at the level of terms, but using the ideas in this Subsection we can fold it back into our logic.

Notation 4.2.6 and Theorem 4.2.7 are written in a syntax enriched with neg, as per Subsection 4.2.1:

NOTATION 4.2.6. Suppose ϕ is a predicate. Then we define $\sim\phi$ the **complement** of ϕ by[29]

$$\sim\phi \quad \text{as shorthand for} \quad neg(\mathsf{term}(\phi)) = 0.$$

As described in Proposition 4.2.4, $\sim\phi$ can be compiled down to an unenriched syntax that assumes a matrix variable symbol neg of arity 3 and a validity predicate χ_{neg} as per Definition 4.1.7. This licenses us to work directly with neg-enriched syntax, and we obtain Theorem 4.2.7:

THEOREM 4.2.7. *Suppose ς is a valuation and $x \in \mathbb{Q}$ and ϕ is a predicate. Then*

$$x \vDash_\varsigma \sim\phi \quad \textit{if and only if} \quad x \nvDash_\varsigma \phi.$$

Proof. By chasing definitions. The fact that this works is in itself a mathematical result that needs to be checked, so we give full details:

$$\begin{array}{lll}
x \vDash_\varsigma \sim\phi \iff [\![\sim\phi]\!]_\varsigma(x) = 0 & & \text{Figure 2} \\
\iff [\![neg(\mathsf{term}(\phi)) = 0]\!]_\varsigma(x) = 0 & & \text{Notation 4.2.6} \\
\iff ([\![neg(\mathsf{term}(\phi))]\!]_\varsigma - 0)^2(x) = 0 & & \text{Figure 2} \\
\iff [\![neg(\mathsf{term}(\phi))]\!]_\varsigma(x) = 0 & & \text{Fact} \\
\iff neg([\![\mathsf{term}(\phi)]\!]_\varsigma(x)) = 0 & & \text{Equation 2 from Subsection 4.2.1} \\
\iff [\![\mathsf{term}(\phi)]\!]_\varsigma(x) \neq 0 & & \text{Definition 4.1.5} \\
\iff [\![\phi]\!]_\varsigma(x) \neq 0 & & \text{Figure 2} \\
\iff x \nvDash_\varsigma \phi & & \text{Figure 2}
\end{array}$$

The use of equation 2 from Subsection 4.2.1 above (i.e. treating neg as a primitive in our syntax) is justified by Proposition 4.2.4. □

Theorems 4.2.7 and 2.4.4 taken together prove that our logic can translate the expressive power of full first-order logic — at least! — into polynomials. Each step in the path to this is arguably quite elementary, but the overall result seems surprising and powerful.

4.3 General recursion

We show how to implement a minimisation function, in the sense of general recursive functions [DN24], in our logic.[30] The construction below is rather fiddly, but this is worth doing because it demonstrates that our logic is expressive enough to verify general recursion.

We already had a hint of this, because we specified Ackermann's function in Subsection 3.5 and this is known as a canonical function that is recursive but not primitive recursive. But here, we isolate and implement a minimisation function in and of itself.

Recall the (standard) definition:

[29] This implementation is not necessarily optimal, but it is simple.

[30] The Wikipedia page gives a clear and accessible presentation.

DEFINITION 4.3.1. Suppose $f : (\mathbb{N}_{\geq 1} \times \mathbb{N}^k) \to \mathbb{N}$.[31] Then define the **minimisation** $\mu(f) : \mathbb{N}^k \to \mathbb{N}$ by

$$\mu(f)(x_1, \ldots, x_k) = min\{n \in \mathbb{N}_{\geq 1} \mid f(n, x_1, \ldots, x_k) = 0\}.$$

In words: $\mu(f)$ maps $(x_1, \ldots, x_k) \in \mathbb{N}^k$ to the least zero value of $\lambda n.f(n, x_1, \ldots, x_k)$.

If no such zero value exists, so that $\{n \in \mathbb{N}_{\geq 1} \mid f(n, x_1, \ldots, x_k) = 0\} = \varnothing$, then the value of $\mu(f)(x_1, \ldots, x_k)$ is not specified.[32]

REMARK 4.3.2. We can think of μ as generalising a *while* loop: given some parameters $(x_1, \ldots, x_k) \in \mathbb{N}^k$, μ searches $n \in \mathbb{N}_{\geq 1}$ and continues while the value of $f(n, x_1, \ldots, x_k) \neq 0$. If $f(n, x_1, \ldots, x_k) = 0$, then μ stops and returns that (first) value of n.

Now we give a slightly lower-level version of Definition 4.3.1. It is still abstract, but it is closer to what we would implement in our logic.

NOTATION 4.3.3. Suppose $U \subseteq \mathbb{N}_{\geq 1} \times \mathbb{N}$ and $b \in \mathbb{N}$. Then define

$$U_b = \{a \in \mathbb{N}_{\geq 1} \mid (a, b) \in U\}.$$

If U_b is an initial segment of $\mathbb{N}_{\geq 1}$ for every $b \in \mathbb{N}$, then call U **initial in its first component**.[33]

Recall that $sign$ is defined in Definition 4.1.5, and note that if we restrict $sign$ to $\mathbb{N}$ then it maps to $\{0, 1\}$.

DEFINITION 4.3.4. Assume we are given the following data:

1. A function $f : (\mathbb{N}_{\geq 1} \times \mathbb{N}) \to \mathbb{N}$ (i.e. for simplicity we have set $k = 1$ in Definition 4.3.1).
2. A $U \subseteq \mathbb{N}_{\geq 1} \times \mathbb{N}$ that is initial in its first component (Notation 4.3.3).

We specify a pair of functions

$$min_{f,U}, acc_{f,U} : U \to \mathbb{N}$$

($min_{f,U}$ is short for 'minimal point' and $acc_{f,U}$ is short for 'accumulator') as follows:

$$\begin{aligned}
&1 \leq min_{f,U}(a, b) \\
a \neq 1 &\Longrightarrow min_{f,U}(a, b) = min_{f,U}(a - 1, b) \\
&f(min_{f,U}(a, b), b) \leq f(a, b) \\
a = 1 &\Longrightarrow acc_{f,U}(a, b) = sign(f(a, b) - f(min_{f,U}(a, b), b)) \\
a \neq 1 &\Longrightarrow acc_{f,U}(a, b) = acc_{f,U}(a\text{-}1, b) * sign(f(a, b) - f(min_{f,U}(a, b), b)) \\
min_{f,U}(a, b) \neq 1 &\Longrightarrow acc_{f,U}(min_{f,U}(a, b) - 1, b) = 1
\end{aligned}$$

We make sense of Definition 4.3.4 using a lemma:

LEMMA 4.3.5. *Continuing the notation of Definition 4.3.4, we have the following properties, for each $b \in \mathbb{N}$:*

1. *$\lambda a{\in}U_b.min_{f,U}(a, b)$ is a constant function into $\mathbb{N}_{\geq 1}$.*
   Write $\mu_{f,U}(b) \in \mathbb{N}_{\geq 1}$ for its unique value.
2. *$f(\mu_{f,U}(b), b) \leq f(a, b)$ for every $a \in U_b$.*
   Thus, $f(\mu_{f,U}(b), b)$ is a lower bound for $\{f(a, b) \mid a \in U_b\}$.
3. *Furthermore, since $f(\mu_{f,U}(b), b) = f(\mu_{f,U}(b), b)$, also $f(\mu_{f,U}(b))$ is a* greatest *lower bound for this set. Thus,*

   $$\mu_{f,U}(b) \in \mathbb{N}_{\geq 1} \text{ is the index of a minimal value of } \lambda a{\in}U_b.f(n, b).$$

4. *$acc_{f,U}(a, b) = \prod_{1 \leq i \leq a} sign(f(a, b) - \mu_{f,U}(b))$.*
   We observe by arithmetic that this is equal to 1 or 0, and furthermore $acc_{f,U}(a, b)$ is equal to 0 precisely when $f(i, b) = f(\mu_{f,U}(b), b)$ for some $1 \leq i \leq a$.

[31] We starting counting at 1 in the first argument for the technical reason that we will be most interested in applying this to identify columns in matrices, which we count from 1, not 0. There is no deeper mathematical significance here.

[32] ...or undefined.

[33] An initial segment $I \subseteq \mathbb{N}_{\geq 1}$ is such that if $a \in I$ and $a > 1$ then $a - 1 \in I$. Note that $\varnothing \subseteq \mathbb{N}_{\geq 1}$ is initial.

$$\begin{array}{rlrl}
\mathsf{mu}_{f,1}(X) & \text{sugars to } \mathsf{ptr.min}(X) & \mathsf{mu}_{f,2}(X) & \text{sugars to } \mathsf{ptr.f}(X) \\
\mathsf{f}_1(\mathsf{ptr.f}(X)) & \text{sugars to } \mathsf{a}(X) & & \\
\mathsf{f}_2(\mathsf{ptr.f}(X)) & \text{sugars to } \mathsf{b}(X) & \mathsf{f}_3(\mathsf{ptr.f}(X)) & \text{sugars to } \mathsf{f.ab}(X) \\
\mathsf{mu}_{f,3}(X) & \text{sugars to } \mathsf{acc}(X) & \mathsf{mu}_{f,4}(X) & \text{sugars to } \mathsf{ptr.prev}(X)
\end{array}$$

Write $t \neq t'$ as shorthand for $sign(t-t')^2 = 1$. Write $t \leq t'$ as shorthand for $sign(t-t'+1) = 1$.

$$\begin{array}{rl}
\chi_{\mu_f} = & 1 \leq \mathsf{ptr.min} \leq \mathsf{len(mu)} \ \wedge\ 1 \leq \mathsf{ptr.prev} \leq \mathsf{len(mu)} \ \wedge\ 1 \leq \mathsf{ptr.f} \leq \mathsf{len(f)} \ \wedge \\
& \forall X_{\mathsf{mu}}.\Big(\ (\mathsf{a}(X) = 1 \ \vee\ \mathsf{ptr.min}(\mathsf{ptr.prev}(X)) = \mathsf{ptr.min}(X)) \ \wedge \\
& \quad (\mathsf{f.ab}(\mathsf{ptr.min}(X)) \leq \mathsf{f.ab}(X)) \ \wedge \\
& \quad \left(\begin{array}{l} \big(\mathsf{a}(X) = 1 \ \wedge\ \mathsf{acc}(X) = sign(\mathsf{f.ab}(X) - \mathsf{f.ab}(\mathsf{ptr.min}(X)))\big) \ \vee \\ \big(\mathsf{a}(X) \neq 1 \ \wedge\ \mathsf{acc}(X) = sign(\mathsf{f.ab}(X) - \mathsf{f.ab}(\mathsf{ptr.min}(X))) * \mathsf{acc}(\mathsf{ptr.prev}(X))\big) \end{array} \right) \ \wedge \\
& \quad (\mathsf{a}(\mathsf{ptr.min}(X)) = 1 \ \vee\ \mathsf{acc}(\mathsf{ptr.prev}(X)) = 1) \ \wedge \\
& \quad (\mathsf{a}(X) = 1 \ \vee\ \mathsf{a}(\mathsf{ptr.prev}(X)) = \mathsf{a}(X) - 1) \ \wedge \\
& \quad (\mathsf{a}(X) = 1 \ \vee\ \mathsf{b}(\mathsf{ptr.prev}(X)) = \mathsf{b}(X))\Big)
\end{array}$$

Figure 9: Expressing a minimisation operator μ (Definition 4.3.6)

5. *If* $\mu_{f,U}(b) > 1$ *then* $acc_{f,U}(\mu_{f,U}(b)-1, b) = 1$, *which means that* $\mu_{f,U}(b) \lneq f(a,b)$ *for every* $1 \leq a < \mu_{f,U}(b)$. *Thus,*

   $\mu_{f,U}(b)$ *is also* least *such that* $\lambda a{\in}U_b.f(a,b)$ *attains its minimal value.*

Proof. Direct from the construction, as described in the statement of this Lemma. □

We can now implement minimisation in our logic, just by translating Definition 4.3.4 into it:

Definition 4.3.6. Suppose we are given a matrix variable symbol f whose first three rows f_1, f_2, and f_3 are intended to represent the two input values a and b and one output value $f(a,b)$ of a binary function $f : (\mathbb{N}_{\geq 1} \times \mathbb{N}) \to \mathbb{N}$.

Assume a matrix variable symbol mu_f with arity $arity(\mathsf{mu}) = 4$. In Figure 9, we define syntactic sugar and use it to express a validity predicate χ_{μ_f}, based on Definition 4.3.4.

Remark 4.3.7. Note that in the predicate χ_{μ_f} in Figure 9 we use the function $sign$ from Definition 4.1.5 directly in terms (its implementation in matrix semantics is in Definition 4.1.7). We can do this because as per Subsection 4.2, the function call can be compiled down to the purely polynomial term language. This is not a problem and our presentation in Figure 9 is more readable as a result.

A simple and natural bit of terminology will be useful in a moment:

Definition 4.3.8. Suppose that:

1. $f : (\mathbb{N}_{\geq 1} \times \mathbb{N}) \to \mathbb{N}$.
2. f with arity $arity(\mathsf{f}) \geq 3$ is a matrix variable symbol.
3. ς is a valuation.

Then say that ς **interprets** f when

$$\varsigma(\mathsf{f})_3(x) = f(\varsigma(\mathsf{f})_1(x), \varsigma(\mathsf{f})_2(x))$$

for every $1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{f}))$.

Proposition 4.3.9. *Suppose* $f : (\mathbb{N}_{\geq 1} \times \mathbb{N}) \to \mathbb{N}$, *and* f *with arity* $arity(\mathsf{f}) \geq 3$ *is a matrix variable symbol, and suppose* ς *is a valuation that interprets* f *(Definition 4.3.8). Then*

- *if* $\vDash_\varsigma \chi_{\mu_f}$ *(Figure 9),*
- *then* $[\![\mathsf{ptr.min}(X)]\!]_\varsigma(x) = \mu_{f,U}([\![\mathsf{b}(X)]\!]_\varsigma(x))$,

$$\begin{array}{ll}
\mathsf{nand}_1(X) \text{ sugars to } \mathsf{nand.a}(X) & \mathsf{nand}_2(X) \text{ sugars to } \mathsf{nand.b}(X) \\
\mathsf{nand}_3(X) \text{ sugars to } \mathsf{nand.out}(X) & \\
\mathsf{nand}_4(X) \text{ sugars to } \mathsf{a.ptr}(X) & \mathsf{nand}_5(X) \text{ sugars to } \mathsf{b.ptr}(X)
\end{array}$$

$$\begin{aligned}
\chi_{\mathsf{nand}} = \; & 0 \leq \mathsf{a.ptr} \leq \mathsf{len}(\mathsf{nand}) \;\wedge\; 0 \leq \mathsf{b.ptr} \leq \mathsf{len}(\mathsf{nand}) \;\wedge \\
& \forall X_{\mathsf{nand}}.\big(\; (\mathsf{nand.out}(X) = (1 - \mathsf{nand.a}(X)) * (1 - \mathsf{nand.b}(X))) \;\wedge \\
& \quad (\mathsf{a.ptr}(X) = 0 \;\vee\; \mathsf{nand.out}(\mathsf{a.ptr}(X)) = \mathsf{nand.a}(X)) \;\wedge \\
& \quad (\mathsf{b.ptr}(X) = 0 \;\vee\; \mathsf{nand.out}(\mathsf{b.ptr}(X)) = \mathsf{nand.b}(X)) \;\wedge \\
& \quad (\mathsf{nand.a}(X) = 0 \vee \mathsf{nand.a}(X) = 1) \;\wedge \\
& \quad (\mathsf{nand.b}(X) = 0 \vee \mathsf{nand.b}(X) = 1)\big)
\end{aligned}$$

Figure 10: Expressing a circuit of NAND gates (Definition 4.4.2)

where we define $U \subseteq \mathbb{N}_{\geq 1} \times \mathbb{N}$ *by*

$$U = \left\{ [\![\mathsf{a}(X)]\!]_\varsigma(x), [\![\mathsf{b}(X)]\!]_\varsigma(x) \mid 1 \leq (x{\in}\mathbb{N}_{\geq 1}) \leq \mathit{len}(\varsigma(\mathsf{mu})) \right\}$$

(this is just a fancy way of setting U *to be the set of* (a, b) *pairs over which* $\varsigma(\mathsf{mu})$ *minimises).*

Proof. We examine Figure 9 and see through careful inspection that it translates Definition 4.3.4 into our logic: $\varsigma(\mathsf{mu}_f)_1$ corresponds to $min_{f,U}$ and $\varsigma(\mathsf{mu}_f)_2$ corresponds to $acc_{f,U}$. □

Remark 4.3.10. Proposition 4.3.9 does not describe the full minimisation outlined in Definition 4.3.1, but this is only because minimisation implicitly contains an unbounded search, whereas a valuation ς delivers *finite* matrices.

This is reasonable, because it is the prover's job to run a program and — after a finite time — it must stop and present the ς it has generated. If the prover does not terminate then no ς is generated and there is nothing for the verifier to do. So in our context, minimisation means finding minimal values within a given run.

Thus Proposition 4.3.9 does the reasonable thing, which is to tell us that if we are given some $b \in \mathbb{N}$ and $a \in \mathbb{N}_{\geq 1}$ and a ς that satisfies the validation predicates and for which $\varsigma(\mathsf{mu})$ contains a column which references $f(a, b)$ — if all the above holds, then $\varsigma(\mathsf{mu})$ will correctly identify the first index $1 \leq i \leq a$ to yield the lowest value of $f(i, b)$ in the range $1 \leq i \leq a$.

4.4 Logic gates

We conclude with a small detour: how to encode a circuit of logic gates in our framework.

In one sense, considering logic gates is a waste: we already have predicates and terms, which are higher-level and (as we have seen) compile to polynomials. Yet it is still an interesting exercise to spell out how this might be done.

We will only implement a *nand* gate; this is known to be a *universal logic gate* from which all others may be derived (see [Kup24] for a clear presentation), and it will also be completely clear from the example of $nand$ below how other logic gates could be directly implemented.

Recall from Remark 2.3.4(1) that we model 0 as 'true' and 1 as 'false', so the bits are flipped. $nand$ should return 1 (false) if both of its inputs are 0 (true), and 0 (true) otherwise.

Definition 4.4.1. It is straightforward to encode NAND as a simple polynomial:

$$nand(a, b) = (1 - a) * (1 - b).$$

Now we want to connect our individual logic gates into circuits.

We could try to nest our gates as functions, but we see from Remark 3.2.12(4) that this would result in polynomials of high degree; a more efficient method is available.

DEFINITION 4.4.2. Assume a matrix variable symbol nand with arity $arity(\mathsf{nand}) = 5$. In Figure 10, we define syntactic sugar and use it to express a validity predicate χ_{nand}, based on Definition 4.4.1.

REMARK 4.4.3. We read through this carefully. Suppose we are given a valuation ς and consider column number x in $\varsigma(\mathsf{nand})$, which is a 5-tuple of elements $(a, b, r, p_1, p_2) \in \mathbb{Z}^5$:

1. a represents the first argument to the NAND gate. This must be equal to either 0 or 1.[34]
2. b represents the second argument to the NAND gate.
3. r represents the result. Note that χ_{nand} insists that $r = nand(a, b)$.
4. p_1 represents *either* a null pointer 0, or it is a pointer to the output of some other gate. Note that χ_{nand} insists that the value of that output is equal to a; thus we 'wire' the output to the first input of this gate.
5. Similarly for p_2.

Visibly this implements a circuit of NAND gates, where input wires are ones such that p_1 (or p_2) is zero, and output wires are outputs that are not pointed to.[35] A more explicit encoding would also be possible, but this will do.

REMARK 4.4.4.

1. Circuits with multiple types of gate (AND, OR, NOT) could be encoded, either as distinct matrix variable symbols with their own validity predicates, or by generalising Figure 10 directly with an additional row of data to describe what type of gate is stored.
2. The degree of the polynomial $[\![\chi_{\mathsf{nand}}]\!]_\varsigma$ scales as $O(n^2)$, where n is $len(\varsigma(\mathsf{nand}))$.
   Our framework is not optimised for the special case of representing a circuit of simple logic gates, but if we specialise anyway then this does not obviously incur any serious overhead.

5 Efficiency: succinctness and builtin functions

5.1 Succinctness

5.1.1 Some terminology and notation

REMARK 5.1.1. A **succinct proof** in the cryptographic sense is when a *prover* does a computation and provides a certificate that can be checked by a *verifier* to verify (to a high degree of certainty) that the computation has been correctly carried out.[36]

We will now illustrate how the maths in this paper is amenable to succinct proofs.

It will be useful to give a name to the polynomial that is 0 on $\{1, \ldots, n\}$:

DEFINITION 5.1.2. Suppose $n \in \mathbb{N}_{\geq 0}$. Define a polynomial $zeroes_n$ over X inductively on n as follows:

$$zeroes_0 = 1 \qquad zeroes_{n+1} = (X - (n+1)) * zeroes_n.$$

Thus for example:

$$zeroes_0 = 1 \qquad zeroes_1 = 1 * (X - 1) \qquad zeroes_3 = 1 * (X - 1) * (X - 2) * (X - 3).$$

DEFINITION 5.1.3. Suppose M is an integer matrix of size $a \times b$ and recall from Notation 2.3.1(2) that M_i denotes the ith row of M (which is a vector of length b), and $intrplnt(M_i)$ (Definition 2.1.4) is a polynomial that interpolates M_i on $\{1, \ldots, b\}$ (which means that $intrplnt(M_i)(j) = M_{i,j}$ for $1 \leq (j{\in}\mathbb{N}_{\geq 1}) \leq b$).

[34] We impose this condition explicitly in Figure 10 as $\mathsf{nand.a}(X){=}0 \vee \mathsf{nand.a}(X){=}1$ and similarly for $\mathsf{nand.b}$; however, we could also insist on $0 \leq \mathsf{nand.a} \leq 1$. Because a valuation ς returns an *integer* matrix, this would suffice. These two methods are however subtly different: if we care about zero knowledge then it may be that range checked values get exposed to the validator. See point 2 of the discussion in Subsection 5.2.

[35] Note that in this system wires can be duplicated; the output of a gate can be wired to zero, one, or many outputs.

[36] Note: a succinct proof in the cryptographic sense is not a formal proof in the logical sense.

Then define the **polynomial interpolant of** M

$$intrplnt(M) = (intrplnt(M_1), \ldots, intrplnt(M_a)).$$

Thus $intrplnt(M)$ is an a-tuple of polynomials such that that ith polynomial interpolates the ith row of M.

DEFINITION 5.1.4 (Tasks and solutions).

1. A **task** $\mathcal{T}$ is a finite set of pairs (C, ϕ) such that for each $(\mathsf{C}, \phi) \in \mathcal{T}$:

   (a) C is a matrix variable symbol.
   (b) ϕ is a quantifier-free predicate.

   We may abuse notation and write

   - $\mathsf{C} \in \mathcal{T}$ for "there exists a ϕ such that $(\mathsf{C}, \phi) \in \mathcal{T}$", and
   - $\phi \in \mathcal{T}$ for "there exists a C such that $(\mathsf{C}, \phi) \in \mathcal{T}$", respectively.

2. A **solution** to a task consists of a pair (ς, h), where:

   (a) ς is a valuation such that
   $$\forall (\mathsf{C}, \phi) \in \mathcal{T}.\ \vDash_\varsigma \forall X_{\mathsf{C}}.\phi. \tag{3}$$
   Recall that the $\vDash_\varsigma \forall X_{\mathsf{C}}.\phi$ notation is from Definition 2.3.2(4), and it makes sense because X is not free in $\forall X_{\mathsf{C}}.\phi$.

   (b) h is a map that assigns to each $(\mathsf{C}, \phi) \in \mathcal{T}$ a **quotient polynomial** $h_{\mathsf{C},\phi} \in \mathbb{Q}[X]$ such that
   $$[\![\phi]\!]_\varsigma = h_{\mathsf{C},\phi} * zeroes_{len(\varsigma(\mathsf{C}))}.$$
   Note that this is a polynomial equality in $\mathbb{Q}[X]$, not a numerical equality in $\mathbb{Q}$. Recall that $len(\varsigma(\mathsf{C}))$ is the number of columns in the matrix $\varsigma(\mathsf{C})$; the number of rows is always $arity(\mathsf{C})$, but the number of columns depends on the choice of ς.

5.1.2 Applying Schwartz-Zippel

This Subsection does not solve the problem of succinct proofs for tasks, but it will set the scene for the next Subsection.[37]

REMARK 5.1.5. Suppose that a *prover* wants to convince a *verifier* that it knows a solution (ς, h) to a task $\mathcal{T}$ (Definition 5.1.4). The prover could send (ς, h) to the verifier and invite it to check $\vDash_\varsigma \forall X_{\mathsf{C}}.\phi$ for every $(\mathsf{C}, \phi) \in \mathcal{T}$ but this would defeat the purpose of having a prover and a verifier, since the verifier is just duplicating the prover's work *and* the network has to communicate all of ς, which is the prover's data.

Now note that it would suffice for the verifier to check for each $(\mathsf{C}, \phi) \in \mathcal{T}$ the polynomial equality

$$[\![\phi]\!]_\varsigma \ = \ h_{\mathsf{C},\phi} * zeroes_{len(\varsigma(\mathsf{C}))}$$

because this implies that $[\![\phi]\!]_\varsigma(i) = 0$ for $1 \leq (i{\in}\mathbb{N}) \leq len(\varsigma(\mathsf{C}))$ and $\vDash_\varsigma \forall X_{\mathsf{C}}.\phi$ follows.

So, we have replaced checking a quantification over $1, \ldots, len(\varsigma(\mathsf{C}))$ with checking a polynomial equality involving $zeroes_{len(\varsigma(\mathsf{C}))}$. How is this an improvement?

We start with something very simple, which is just to apply the Schwartz-Zippel lemma:

LEMMA 5.1.6. *Suppose $\mathcal{T}$ is a task and suppose (ς, h) is a claimed solution to $\mathcal{T}$. We proceed as follows:*

1. *For each $(\mathsf{C}, \phi) \in \mathcal{T}$, choose a random $q \in \mathbb{Q}$, and*

[37] If you are familiar with this material then remember that other readers might not be, and this paper is for them too. Please be patient, or just skip to the next Subsection.

2. *check that*

$$[\![\phi]\!]_\varsigma(q) \;=\; h_{\mathsf{C},\phi}(q) * zeroes_{len(\varsigma(\mathsf{C}))}(q).$$

If are these equalities hold, then with a high degree of probability (ς, h) *is indeed a solution to* $\mathcal{T}$.

Proof. By elementary properties of polynomials, if $[\![\phi]\!]_\varsigma$ and $h_{\mathsf{C},\phi} * zeroes_{len(\varsigma(\mathsf{C}))}$ are not equal polynomials then they agree on at most $d \in \mathbb{N}$ points, where d is the highest power of X in either polynomial.

Thus if we choose $q \in \mathbb{Q}$ at random, it is almost certain to *not* be one of those points. This is the basis of the Schwartz-Zippel lemma; we can check polynomial equality to a high degree of probability, by checking equality at a random point [Tha22, Lemma 3.3, Subsection 3.4]. So, if we evaluate our two polynomials at a random point and get the same answer, then they are almost certainly equal. By the discussion in Remark 5.1.5 above, $\vDash_\varsigma \forall X_{\mathsf{C}}.\phi$ is proven (to a high degree of probability). □

Remark 5.1.7. The scheme above is not necessarily what a cryptographer would call a 'succinct proof system'. Lemma 5.1.6 does save the verifier *some* computation, since for each $(\mathsf{C}, \phi) \in \mathcal{T}$ the verifier needs to evaluate $[\![\phi]\!]_\varsigma$, $h_{\mathsf{C},\phi}$, and $zeroes_{len(\varsigma(\mathsf{C}))}$ at one point q instead of at every $1 \leq (x \in \mathbb{N}_{\geq 1}) \leq len(\varsigma(\mathsf{C}))$.

However, the prover still needs to communicate the full solution (ς, h) to the verifier, and the verifier still needs to directly evaluate large polynomials (albeit at a single point). There are ways to make this more efficient, as we discuss next.

5.1.3 Succinct proof with inner product arguments

For concreteness we consider our scheme for the case of the validity predicate for the power function from Subsection 3.2. This is a simple example, but it illustrates the issues involved and it will be clear how to generalise to the arbitrary case.

The reader does need to know much cryptography to understand this Subsection. We just appeal to the fact that it is possible for a prover to prove (to a high degree of confidence) to a verifier that the prover knows a polynomial P and values x and y such that $P(x) = y$ — and, *P does not need to be communicated to the verifier, and the verifier does not need to carry out the computation 'evaluate P at x'*. Details follow below.

Recall the validity predicate χ_{pow} for pow from Figure 3. Using Lemma 3.7.4 and Definition 3.7.2, we can express this as the following simple task:

$$\mathcal{T} = \{(\mathsf{pow}, \phi_1),\ (\mathsf{pow}, \phi_2)\}$$

where

$$\phi_1 = \mathsf{range}_{1,len(\varsigma(\mathsf{pow}))}(\mathsf{rec}(X)) \;\text{ and }\; \phi_2 = \mathsf{clause1}(X) \,\mathsf{V}\, \mathsf{clause2}(X).$$

Above, $\mathsf{range}_{1,len(\varsigma(\mathsf{pow}))}$ is from Definition 3.7.2, and clause1 and clause2 and rec are from Figure 3.

The prover computes a solution (ς, h) for this task. If the prover is honest and correct then by Lemma 3.2.7 $\varsigma(\mathsf{pow})$ encodes a partial power function (Definition 3.2.6).

The verifier should now choose some very large prime p — one that that is larger than any coefficient or power mentioned in $\mathcal{T}$ or its solution (ς, h). Bearing in mind that ς and h are physically constrained by the prover's memory, a prime close to 2^{256} should suffice. The verifier communicates this p to the prover.

The prover then homomorphically maps the equalities

$$[\![\phi_i]\!]_\varsigma = h_{\mathsf{pow},\phi_i} * zeroes_{len(\varsigma(\mathsf{pow}))} \quad \text{for } i \in \{1, 2\}$$

into the finite field $\mathbb{F}(p)$ (which is isomorphic to $\mathbb{Z}/\mathbb{Z}p$). The prover and verifier will perform the succinct check in $\mathbb{F}(p)$.[38]

[38] Note that we can do this *even if* our original logical assertion used conjunction, and *even if* we have done other things, such as the Gödel encoding of combinator syntax mentioned in Definition 3.9.3(3). We just have a polynomial in $\mathbb{F}(p)$ and we want to check something about it.

So now we have a polynomial equality over $\mathbb{F}(p)$. Note that ϕ_1 and ϕ_2 are fixed, because they define the problem to be solved (namely: computing an exponential). The parts that vary are $\varsigma(\mathsf{pow})$, h_{pow,ϕ_i} for $i = 1, 2$, and evaluating the interpolants; thus we have

- a (the interpolant to row 1 of $\varsigma(\mathsf{pow})$, mod p),
- b (interpolant to row 2),
- out (row 3),
- rec (row 4), and
- h_{pow,ϕ_1} and h_{pow,ϕ_2}.

These are the polynomials whose evaluations (at values x and r in $\mathbb{F}(p)$ described below) the prover will need to prove, and which the verifier will need to check. This is made mildly more complicated because we evaluate not only at some $x \in \mathbb{F}(p)$, but also at values derived from x. To see how, we unpack our example of Figure 3 and note that to check

$$[\![\phi_i]\!]_\varsigma(x) = h_{\mathsf{pow},\phi_i}(x) * zeroes_{len(\varsigma(\mathsf{pow}))}(x) \qquad (\text{mod } p)$$

it suffices to evaluate the following polynomial evaluations — which the reader can check are the component polynomials used in Figure 3 — mod p:

$$a(x),\ out(x),\ rec(x) = r,\ a(r),\ b(r),\ out(x),\ out(r),\ h_{\mathsf{pow},\phi_i}(x), \\ range_{1,\mathsf{len}(\varsigma(\mathsf{pow}))}(r) \text{ and } zeroes_{len(\varsigma(\mathsf{pow}))}(x). \quad (4)$$

Above, the value of $\mathsf{rec}(x)$ is re-used, so for brevity we give it the name r.

Now we need some cryptography: succinctly proving the polynomial evaluations above should be straightforward by (for example) an *inner product argument* [BCC+16][39] — except for the values of $range_{1,\mathsf{len}(\varsigma(\mathsf{pow}))}(r)$ and $zeroes_{len(\varsigma(\mathsf{pow}))}(x)$, which the verifier would need to compute itself.[40]

Once the prover has proven to the verifier that it knows polynomials a, out, rec, b, and h_{pow,ϕ_i} with suitable evaluations, the verifier can just perform a few $\mathbb{F}_p$-additions and -multiplications as per Example 3.2.11, as follows:

$$\begin{aligned}[\![\phi_1]\!]_\varsigma(x) &= range_{1,\mathsf{len}(\varsigma(\mathsf{pow}))}(r) \\ [\![\phi_2]\!]_\varsigma(x) &= \big(a(x)^2 + (out(x){-}1)^2\big) * \\ &\quad \big((a(x){-}a(r_x))^2 + (b(x){-}(b(r_x){+}1))^2 + (out(x) - a(x){*}out(r_x))^2\big)\end{aligned}$$

We count fifteen additions or multiplications above, so this is very easy. Then, with two more multiplications in $\mathbb{F}_p$, the verifier checks that $[\![\phi_i]\!]_\varsigma(x) = h_{\mathsf{pow},\phi_i}(x) * zeroes_{len(\varsigma(\mathsf{pow}))}(x)$ for $i \in \{1, 2\}$.

Example 5.1.8. Suppose a prover wants to prove to a verifier that $10^2 = 100$. It can proceed as follows:

1. The prover publishes the predicate

$$\phi_1 \ \wedge\ \phi_2 \ \wedge\ (\mathsf{a}(1) = 10 \wedge \mathsf{b}(1) = 2 \wedge \mathsf{out}(1) = 100)$$

   and computes a ς that makes this predicate valid.
2. The prover and verifier perform the verification ceremony as described above. The verifier is assured that the prover has a $\varsigma(\mathsf{pow})$ such that that χ_{pow}, and furthermore the first column of $\varsigma(\mathsf{pow})$ asserts that $10^2 = 100$.

[39] An precise yet accessible survey, with references, is here.

[40] The difference is that *range* and *zeroes* are known polynomials; they are not arbitrary ones provided by the prover. If the verifier lacks the computational resources to perform these computations then the task could be passed to a trusted service. Note that this is a one-off cost, and so can be amortised over multiple clauses.

Remark 5.1.9. Our scheme is sound and complete by construction, in the sense that by Theorem 2.4.4 we generate polynomial equalities that are valid if and only if the logical assertion is true, and to prove these equalities valid, it suffices to provide succinct proofs of their component parts as outlined above.

Still, the above is proof-of-concept. In a real implementation we would want to add more advanced techniques, to increase efficiency or to attain a zero-knowledge proof (in which the verifier has no information at all about the solution ς).

Fortunately such techniques are available off-the-shelf. A common practice to achieve zero-knowledge is to apply recursive proof composition [BCCT13] on an existing SNARK. Similarly, since the complexity of the verifier in an inner product argument scheme is linear in time, we can add another layer of recursion using a SNARK of sublinear verification time. Halo2 [BGH19] leverages recursive proof composition techniques to amortise verification costs by constructing an accumulation scheme on a proving system based on the inner product argument. A KZG-based method should also be possible [KZG10] but describing it is more complex. Investigating this is future work.

Remark 5.1.10. Our translation from FOL to polynomials works for all FOL predicates, but the schema for succinct proof above works only on (the translation of) predicates with a single universal quantifier at top level (so a fragment of Π^0_1 in the arithmetical hierarchy; but note Remark 6.1.1(6)). This is all we need for our examples.

We suspect it is all we will ever need, in the sense that we suspect (but have not proved) that the effects of nested quantifiers could be emulated using techniques similar to the encoding of arbitrary functions described in Subsection 4.2. Checking this, or generalising the schema, is future work.

5.2 Built-in functions

Having considered cryptographic notions of efficiency in the previous Subsection, we now make a few observations about making things run quickly on native silicon. In this case, we may prefer to check some validity properties directly:

1. We gave a polynomial circuit to check that a matrix expresses bitwise expansion of 64-bit unsigned integers in Subsection 3.6. But bitwise operations may be rapid on underlying hardware — not least because this is how numbers are represented anyway. In practice, rather than check a validity predicate, a verifier might recognise that this is a bitwise expansion operation and prefer to just traverse the matrix and check its correctness directly.
   In a monolithic system, where everything has to run in an abstract machine built in arithmetic circuits, this might be a bit problematic because it would involve stepping outside the abstraction provided. The system we describe in this paper is quite modular, so this may be less of a problem.
2. The range check primitive $<$ in Figure 1 can be checked rapidly in silicon.
   If we are just using $<$ to range check that pointers are in bound for the matrices they point to, then information leakage would be minimal: all a validator would have is a set of pointers, but without knowing how the prover has laid out its data in its matrices, this does not impart much useful information. For extra security, the numbers could always be obfuscated.
   Concretely, we would do this by providing pos from Definition 4.1.2 as a built-in matrix variable symbol of infinite length — calls to the validity check of pos would just get passed directly to the underlying silicon. As per Remark 4.1.10, just providing this would provide all the power of $<$.

So we see that if we just care about speed, then efficient checking in silicon for a small set of special built-in validity predicates, like range checks and bitwise operations, should be fine. If we want a full zero-knowledge treatment, such that the verifier never learns the precise witnesses used, this would not be possible.

But even here, we might (depending on circumstances) be happy to accept a small amount of information leakage, if this buys us enough efficiency. In particular, range checks for pointers could

be done outside of a zero knowledge wrapper and information leakage may be acceptable, if all the verifier has is a bunch of pointers but no information about what they point to or how they connect. This seems plausible, but working through how to do it in practice is future work.

6 Conclusions and further work

6.1 Future work and other observations

REMARK 6.1.1. There are plenty of directions to go from here. For example:

1. We have set up a shallow translation of logic to polynomials, and logical judgements to polynomial equalities. An obvious next step is to engineer a practical proving system based on these ideas.
2. We include an existential quantifier $\exists$ in our syntax and semantics, but in practice a more efficient technique might be to use *Skolem functions* [AZ20]. A Skolem function is just a way to replace an existential quantifier with a function symbol; e.g. $\forall x.\exists y.x = y$ becomes $\exists f.\forall x.x = f(x)$. The f can then be compiled down to a matrix variable symbol in our base syntax, using the techniques in Subsection 4.2.
   Thus mathematically, one way to view our matrix variable symbols is as Skolem symbols.
3. Our polynomials are univariate, and correspondingly our first-order logic has just one variable symbol X (and arbitrarily many matrix variable symbols).
   It would be natural to generalise to the multivariate case — thus permitting X, but also X' and Y, and so on, in Figure 1. This would naturally give a translation to multivariate first-order logic, just by setting $[\![X']\!] = X'$ for each variable symbols in Figure 2. Nothing in the maths in Figure 2 would be affected.
   This might be convenient and useful for applying multivariate arithmetisation techniques, but we do not think it would increase expressivity. It seems likely that we could emulate the multivariate case using Skolem functions (as discussed above) and (functions representing) ϵ-terms [AZ20]. Looking into this generalisation is future work.
4. We have mapped one logic to polynomials, but this only opens up a very interesting design space for other logics.
   For a start, we can consider extending the logic with constructs, including ones based on polynomial division. For instance: define $P \setminus Q$ to be P with any roots it shares with Q divided out. This would model non-implication $\phi \not\Rightarrow \psi$ (which can also be written as $\phi \wedge \neg\psi$).
   We can also ask:

   > Now that we see that polynomials have logical content in the sense observed in this paper, what is the minimal easily-representable logic that naturally reflects polynomial structure?

   Put another way, we encoded FOL semantics in polynomials, but what is the natural full 'preimage' of polynomials ... in logic?
5. It is natural to finesse the logic to make it resource-sensitive. The validity judgement of our logic is clean and simple
   $$x \vDash_\varsigma \phi \iff [\![\phi]\!]_\varsigma(x) = 0,$$
   as per Figure 2. This just says that x is a root of $[\![\phi]\!]_\varsigma$. But a polynomial is a *multiset* of roots, which suggests a validity judgement of the form
   $$\vDash_\varsigma (\Phi \vdash \psi) \iff [\![\psi]\!]_\varsigma \mid \prod_{\phi\in\Phi} [\![\phi]\!]_\varsigma.$$
   This polynomial divisibility assertion is also extremely natural.
   Note that this would change the logic; it would become resource-sensitive, in the sense of linear logic. Whether this is a feature or a bug depends on what we want to achieve; see e.g. Remark 3.2.12 which comments on the efficiencies of being able to remove repeated roots.

6. How powerful is our logic, really? In fact, extremely powerful.
   The syntax in Figure 1 has a term-former term that maps predicates back down to terms. This reflects the fact that in our semantics, terms map to polynomials, and so do predicates (predicates map to *non-negative* polynomials, but a non-negative polynomial is still just a polynomial).
   This is unusual and it makes the logic impredicative. We have called our logic 'first-order logic', and this was not untrue since it comes with a specific intended model in which the variable X ranges over numbers. But what that left unsaid at the start, for simplicity, was that our logic *also* has an in-built mapping between predicates and terms; with term going from predicates to terms, and $t = 0$ going from terms to predicates (cf. Remark 2.4.6).[41]
   So this is FOL, but not *just* FOL, and the full expressivity of the syntax in Figure 1, with its intended semantics in Figure 2 is a topic for future research.
7. Tree-structured data is a used pervasively in declarative programming. In light of this paper, it would be interesting to give labelled trees additive and multiplicative structure sufficient to do polynomial arithmetic directly over trees (something along these lines has been done [Tar16]) so that, instead of working as we do in this paper with a ring of polynomials over rationals ($\mathbb{Q}[X]$), we could work with a ring of polynomials over trees: $\mathsf{Tree}[X]$. Integers can already emulate trees via a Gödel encoding, but if we had polynomials with coefficients that already *are* trees, then we might be able to avoid some encoding and decoding. This is speculative but would be interesting to consider for future work.

6.2 More on efficiency

We discussed efficiency and succinctness in Remarks 3.2.12 and 3.4.6(2) and Section 5. We make a few more general observations, based on the following question:

How big do our polynomials get?

Remark 6.2.1.

1. When we succinctly check a polynomial equality

$$[\![\phi]\!]_{\varsigma}(q) = h_{\mathsf{C},\phi}(q) * zeroes_{len(\varsigma(\mathsf{C}))}(q),$$

   lower-degree polynomials are better because *a)* they are easier to evaluate and *b)* they have fewer roots (so that a succinct check is less likely to give a false positive).

   So we can ask what contributes to the degree of the polynomial that is $[\![\phi]\!]_{\varsigma}$. We disussed this in the context of a simple but paradigmatic example in Remark 3.2.12.

   One important point is that $\wedge$ maps to addition, which does not increase degree — we can say: *conjunction is free*. Thus, succinctly checking a conjunction of validity predicates is no more expensive than checking the most expensive (= highest degree) clause.

   Thus, a program consisting of a large number of matrix variable symbols with relatively simple, low-degree validity predicates, is better than a program with a smaller number of matrix variable symbols with more complex validity predicates. This fits in nicely with a modular programming philosophy, where we try to factor our system into a many relatively simple parts. The reader will know that factoring a system into smaller pieces, where possible, is generally accepted as good practice *anyway* — but here it has concrete mathematical implications: we get lower degree polynomials.

2. Having matrices with lots of rows is cheap, because rows do not directly increase the degree of polynomials. Long interpolation vectors cost, because (as we noted in Definition 2.1.4(2)

[41] This is *not* a Gödel encoding! A Gödel encoding would inject raw predicate syntax into numbers. term behaves more like a type casing function, mapping predicate semantics into term semantics — as a noop, since both are just polynomials.

and Remark 3.2.12(5)) a vector of length n requires in general a degree n-1 interpolating polynomial.

So we want to design our validation predicates to create tall but narrow matrices.

3. So our question becomes:

   What leads to wide matrices?

   Broadly speaking, this is bounded by how many *distinct* function calls are made overall ('distinct', because the system is naturally memoised; see Remark 3.4.6(2)).

   Counting such calls is a familiar task; for example, exponentiation a^b can be computed in $O(ln(b))$ distinct calls (as discussed in Remark 3.2.14). So after all this discussion, our conclusion is fairly straightforward:

   (a) computation and control flow are free, and
   (b) the cost of verification scales with the size of the *longest deduplicated derivation chain*.

   Broadly speaking, this corresponds to the inherent complexity of proving something, so we can say that our translation is efficient in the sense that it converts a logical problem into an arithmetic one of roughly equal (and thus not much larger) complexity.

6.3 Other comments

Remark 6.3.1 (Logic programming). Logic programming can be understood as the engineering of predicates and derivation rules to make the search for derivation-trees computationally efficient (an old but to-the-point research survey is in [MNPS89]).

Our examples in Section 3 show how we can encode computation using derivation-trees, as is standard in mathematically structured programming. A so-called *validity predicate* χ_{C} expresses when a matrix $\varsigma(\mathsf{C})$ expresses a valid derivation-tree, and thus a valid computation. The reader can think of $\varsigma(\mathsf{C})$ as being a kind of 'generalised computation trace' (but remember this is very general, and in particular derivation trees need not be linear).

Given that this paper uses derivation-trees, and so does logic programming, is there a connection? Yes, and no.

For us in this paper, *finding* a valid derivation-tree / providing a suitable ς, is for the prover. Many methods for generating derivation-trees exist — including logic-programming systems, but also rewrite systems, or just by some suitable evaluation strategy. What is true is that logic programming is one way to efficiently construct derivation-trees, and our semantics consumes them, so there is a natural match here.

Remark 6.3.2 (Why matrices?). Why do we need matrices variable symbols in Figure 2? Why not just encode all our information into natural numbers, as we do e.g. when we use a Cantor pairing function to Gödel encode combinator terms in Definition 3.9.3? Why not just do this or something like it throughout?

We could try, but a key issue is our use of *pointers*. The reader can see in the examples in Section 3 how we use entries in our matrices to point to columns in matrices (we make an effort to really spell this out in Examples 3.2.8 and 3.2.9). If we Gödel encode all structure in natural numbers, then this would be harder to do because we would have to unpack those pointers from the encoding before using them. This is not necessarily impossible, but it would add layers of complexity which for now we prefer to do without.

To put this another way: we use derivation-trees a lot in our semantics, and trees are graphs, and graphs are labelled nodes with edges. There is a natural structural map from here to matrices, and thence to interpolating polynomials and succinct proofs in the cryptographic sense. Other design decisions may be possible, but this paper follows a(n in retrospect) natural structural path through the mathematics.

Remark 6.3.3 (Why $\mathbb{Q}$?). We give predicates semantics as nonnegative polynomials in $\mathbb{Q}[X]$, interpret truth as the number 0, and map $\mathsf{\wedge}$ to $+$. This may seen counterintuitive: yes if $x, y \geq 0$ then $x+y=0$ if and on only if $x=0 \wedge y=0$ — but we lose this property if we work in a p-element finite field $\mathbb{F}(p)$, as is common in cryptography. How is this consistent with our semantics for $\mathsf{\wedge}$ as addition?

The point is that once we have transformed our logical judgements into a polynomial equalities over $\mathbb{Q}[X]$ as per Figure 2 or Lemma 5.1.6, all we have is a polynomial equality. We are free to verify this by the method of our choice, including mapping to a prime field (so long as the prime p is chosen sensibly and large enough that Schwartz-Zippel works), and use whatever cryptographic apparatus we wish. There is no contradiction between that and our use of $\mathbb{Q}$.

Remark 6.3.4 (FOL truth vs. FOL derivability). This paper maps FOL judgements directly to polynomial equalities (Figure 2), such that valid (true) judgements map to valid (true) equalities.

Another approach to encoding FOL is to map its syntax and derivation-trees to polynomials with a circuit that identifies the (encodings of) well-formed derivation trees. That is also a good approach, but it encodes *derivability*, not *validity*.

To see how these differ, consider the FOL predicate $0=0$. This is valid, since 0 *is* equal to 0; and it is derivable by the equality right-derivation rule. As per Figure 2 $[\![0=0]\!]_\varsigma = 0$, which represents 'true'. Thus, $[\![0=0]\!]_\varsigma$ is true because it literally *is* true; indeed no reasoning or circuitry is required in this case; it is just a fact! An arithmetisation of FOL that uses derivability would encode a (simple) derivation tree like this

$$\frac{}{\vdash 0=0}\ (\mathbf{=R})$$

along with a circuit expressing that this is well-formed.[42] It makes sense, but it is a different approach.

It is the difference between asserting '$\forall x.\exists y.x=y$ is true in some finite model' and 'I have a valid derivation-tree of $\forall x.\exists y.x=y$'. The former is a (true) assertion about concrete behaviour in finite models (finite, because ς is unbounded, but finite), whereas the latter is an assertion about derivability.

For computation, finite models are what matters most, so our design choice is natural. We labour this point because derivation-trees still feature in this paper (they are a common structure), but they arise in the spirit of being generalised computation traces; the embedding of the FOL logic remains semantic and direct as discussed above. An 'infinite models' version of the ideas in this paper, possibly working through explicit consideration of FOL derivation-trees, is possible future work.

Remark 6.3.5 (Rapid coprocessors). We could use the ideas in this paper for fast and efficient prototyping of a correct-by-construction *coprocessor* for an existing system: to create some function, specified and arithmetised using the techniques in this paper, for the specific purpose of being called from some other zero-knowledge abstract machine.

We have seen how the compositional nature of our semantics lends itself — by design — to specifying functional programs, which require no external state and which are correct by construction.[43] Functional components are modular and compositional, and therefore portable and well-suited to being plugged together.

This compositionality could be useful and may mean that the barrier to extracting practical value from these ideas may be relatively low.

[42] Note that this would require us to think about how to represent the syntax of FOL propositions and the syntax of FOL derivation-trees, in polynomials. This is possible but it's just an extra layer of complexity.

We can turn this around: the relative straightforwardness of the representation in this paper is a feature, not a bug, and it is no accident but the result of specific design decisions. Of course we hope there will be scope to make things *even simpler* and more direct, and if so this is future work.

[43] Correct by construction with respect to the specification. If the specification is buggy then the program constructed from it will be too. But it will faithfully satisfy the (incorrect) specification.

Furthermore, pure speed is not the only metric of success: shorter development time, greater modularity, readability, maintainability, portability, and amenability to formal verification of correctness, are also valid criteria, amongst others. High-level specifications tend to perform well by these metrics, so a compositional shallow translation directly to polynomials is interesting.

6.4 Final remarks

We have mapped logic to polynomials, and logical judgements to polynomial equalities. At a high level, the mapping is simple:

1. Truth maps to zero, and falsity maps to one.
2. $\bigwedge$ and $\forall$ map to $+$, and $\bigvee$ and $\exists$ map to $*$ (see Figure 2 and the discussion in Remark 3.2.12).

Simple facts of arithmetic — notably that a sum of two nonnegative numbers is zero if and only if both numbers are zero (Lemma 2.4.2) — give us a soundness and completeness result (Theorem 2.4.4).

It is perhaps surprising that from this elementary arithmetic beginning, we manage to pull out so much logical and computational content.

Using matrix variable symbols and some simple logic, we specify a library of exemplar functions (Subsection 3), including the Turing-complete SK-combinator system.[44] Note that the valuation ς may have arbitrary size; its size and the size of the validity predicate are two orthogonal parameters, so that even a fairly simple and compact validity predicate (like the one in Figure 8) can express the correctness of an arbitrarily complex computation trace as encoded in ς. Furthermore, computations corresponds to derivations (in the style of mathematically structured programming), which are trees: they are not restricted to being a linear trace. Finally, we note that the range check primitive acts in the context of our logic as a truth modality (see Remark 4.1.11) and using this we can encode even more interesting structure (see Section 4).

Our semantics maps logical assertions to polynomial equalities (Definition 2.3.2(3)), which can be checked succinctly and/or in zero knowledge (Subsection 5.1.3). The semantics is clean, compositional, and modular, and overheads seem to be pleasingly low — the cost of verifying a derivation scales (roughly speaking) with its deduplicated size. We get a nice mapping from derivation-trees to matrices, and modularity in the specification corresponds to relatively lower degree polynomials. Practical application is plausible both in and of itself, and as a coprocessor for other systems.

Because this system is based on logic, implementations based on it are likely to be amenable to tools from computer science such as formal verification, correctness-by-construction, and type systems. Although not everything in this paper is elementary, it is noteworthy that the complexities that arise come from *the applications*, and not from the translation of the logic, which is simple and direct. In other words: our logic and its polynomial translation do not seem to make a thing any harder than it inherently *already is*. For instance: a $\bigwedge$ turns into a $+$ and a $\bigvee$ turns into $*$, thus one connective = one arithmetic operation — it is hard to imagine a translation being more transparent than that!

Finally, logics other than first-order logic exist, so implicit in the act of mapping of *this* logic to polynomials is an implication that we could do the same for *others* — and conversely, there is a possibility for logicians to study interesting new logics based on polynomial semantics.

References

[AZ20] Jeremy Avigad and Richard Zach. The Epsilon Calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2020 edition, 2020.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. URL: http://www.andrew.cmu.edu/user/cebrown/notes/barendregt.html.

[44] SK combinators are not user-friendly, but they are one paradigmatic example of a Turing-complete system.

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 327–357. Springer, 2016.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 111–120, New York, NY, USA, 2013. Association for Computing Machinery. Also available at https://eprint.iacr.org/2012/095.pdf. doi:10.1145/2488608.2488623.

[BCF+23] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: efficient, succinct, modular. *Designs, Codes and Cryptography*, 91(11):3457–3525, Nov 2023. doi:10.1007/s10623-023-01245-1.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019. https://eprint.iacr.org/2019/1021. URL: https://eprint.iacr.org/2019/1021.

[Bim20] Katalin Bimbó. Combinatory Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2020 edition, 2020.

[Das18] Ali Dasdan. Twelve simple algorithms to compute fibonacci numbers, 2018. Available at https://arxiv.org/abs/1803.07199. arXiv:1803.07199.

[DFGK14] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 532–550, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[DN24] Walter Dean and Alberto Naibo. Recursive Functions. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2024 edition, 2024. https://plato.stanford.edu/entries/recursive-functions/ (permalink).

[Hun12] T.W. Hungerford. *Abstract Algebra: An Introduction*. Cengage Learning, 3 edition, 2012. URL: https://books.google.ch/books?id=zRkLAAAAQBAJ.

[Kle43] Stephen C. Kleene. *Transactions of the American Mathematical Society*, 53:41–73, 1943. Available online here (permalink).

[Kup24] Tony R. Kuphaldt. Gate universality. In *Lessons in Electric Circuits*. Accessed 2024. Free online textbook, available online (permalink).

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LCL+24] Xi Lin, Heyang Cao, Feng-Hao Liu, Zhedong Wang, and Mingsheng Wang. Shorter ZK-SNARKs from square span programs over ideal lattices. *Cybersecurity*, 7(1):33, Mar 2024. doi:10.1186/s42400-024-00215-x.

[LUR] LURK lab. The Lurk Circuit Specification. https://blog.lurk-lang.org/posts/circuit-spec/.

[MNPS89] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. Technical report, Durham, NC, USA, 1989.

[Nat15] Melvyn Nathanson. Cantor Polynomials and the Fueter-Pólya Theorem. *The American Mathematical Monthly*, 123, 12 2015. doi:10.4169/amer.math.monthly.123.10.1001.

[SW22] Abner J. Salgado and Steven M. Wise. *Polynomial Interpolation*, page 231–265. Cambridge University Press, 2022.

[Tar16] Paul Tarau. Computing with catalan families, generically. In Marco Gavanelli and John Reppy, editors, *Practical Aspects of Declarative Languages*, pages 117–134, Cham, 2016. Springer International Publishing.

[Tha22] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. Number 4:2–4 in Foundations and Trends in Privacy and Security. Now publishers, December 2022.

[Ver24] Rineke (L.C.) Verbrugge. Provability Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2024 edition, 2024.