Quirky Interactive Reductions of Knowledge

Joseph Johnston

Abstract

Interactive proofs and arguments of knowledge can be generalized to the concept of interactive reductions of knowledge, where proving knowledge of a witness for one NP language is reduced to proving knowledge of a witness for another NP language. We take this generalization and specialize it to a class of reductions we refer to as 'quirky interactive reductions of knowledge' (or QUIRKs). This name reflects our particular design choices within the broad and diverse world of interactive reduction methods. A central design choice is allowing the prover to rewind or regress to any previous reduction and repeat it as many times as desired. We prove completeness and extractability properties for QUIRKs. We also offer tools for constructing extraction algorithms along with several simple examples of usage.

Contents

1 Introduction

The theory and practice of interactive proof systems have a long history beginning with [GMR85] and [Bab85]; we provide an overview of those founding papers in Appendix A. Many variational frameworks for proof systems emerged in the years following, and today one of the most popular frameworks for proof systems is that of ‘interactive oracle proofs’ (IOPs) from [BSCS16]. We aim to develop another framework that can capture most if not all practical proof systems while shifting perspective on how we may develop and analyze proof systems. We shift focus from the monolithic nature of proof systems as entire protocols, to the modular nature of how most proof systems can be decomposed into prover-moves and verifier-moves. Proof systems for NP languages are of concern, and in particular proofs of knowledge for NP, where the prover must prove knowledge of a witness for a particular instance to be in a particular language. Beyond proofs we also allow for ‘arguments’ in which the prover is assumed computationally bounded. We study ‘reductions of knowledge’ between languages, that is where the prover may not fully prove knowledge of a witness for an instance in a language but instead reduce to proving knowledge of a witness for another instance in another language. Any interactive proof or argument of knowledge can be viewed as an interactive reduction of knowledge, and vice versa. While the reduction-based perspective then is not fundamentally different, we believe it to be valuable and are not the first to write on the perspective (e.g. see [KP22]).

We develop a class of reductions we call ‘quirky interactive reductions of knowledge’ to reflect our particular and potentially peculiar design decisions. Most of these design decisions relate to how the prover and verifier interact, and how we define associated notions such as completeness and knowledge soundness. In relation to IOPs, we mention two other design decisions.

- QUIRKs are void of oracles, taking place entirely in the plain model. The popularity of IOPs today is likely due to the convenience of relying on oracles to represent data that is not readily available to the verifier, such as the witness the prover claims to hold. A similar convenience can be achieved with QUIRKs by means of composition.
- While traditional interactive proofs and arguments assume the prover cannot ask the verifier for a new challenge when not fond of a challenge given, QUIRKs indeed allow the prover to navigate the entire sequence of reductions both backwards and forwards as many times as desired. Moving backwards we call 'regression' and moving forwards we call 'progression'. The notion of 'state-restoration' presented in [BSCS16] for IOPs is similar, where the prover is allowed to rewind the verifier to any previous verifier state. In the case that the verifier only chooses random challenges, however, the verifier has no state. In QUIRKs we restrict the verifier to only sampling random challenges, and therefore according to [BSCS16] the verifier is stateless. Using the term 'state-restoration' then makes little sense for QUIRKs since there is never any state.

Sections are organized as follows.

- In Section 2 we define QUIRKs, associated notions, and we present the core theorem of QUIRKs.
- In Sections 3 and 4 we prove the completeness and extractability properties fo QUIRKs, respectively.
- In Sections 5 and 6 we present two auxiliary algorithms one may choose to use in constructing extractors. Both of their results, concerning the probability of success and expected time of sampling algorithms, have two versions. While both versions may be used to achieve the same extractability errors, one offers superior probability of success and the other offers superior expected running time. For simplicity of notation, in the definitions of QUIRKs and their proven properties, as well as all examples of QUIRKs we explore, we use the version with superior running time for notational simplicity. The versions offering superior probability of success, however, are important for achieving what are likely near optimal results.
- In Section 7 we introduce the notions of 'instance reductions' and 'witness reductions' and present example generic constructions of each.
- In Section 8 we present several toy examples of QUIRK constructions.

Throughout we use the following notation and conventions without re-introduction.

- We use several algebraic data types defined as

$$\begin{aligned}\mathsf{Maybe}\langle T\rangle &= \mathsf{No} \mid \mathsf{Yes}(T)\\ \mathsf{Result}\langle T, E\rangle &= \mathsf{Ok}(T) \mid \mathsf{Err}(E)\\ \mathsf{Either}\langle L, R\rangle &= \mathsf{Left}(L) \mid \mathsf{Right}(R)\end{aligned}$$

  We use two enumerated types called Way and Error. A core type will be $\mathsf{Result}\langle T, \mathsf{Error}\rangle$ for some type T.

$$\mathsf{Way} = \mathsf{Regress} \mid \mathsf{Progress}, \quad \mathsf{Error} = \mathsf{Regress} \mid \mathsf{Fail}$$

  We treat types also as sets, a type T naturally defined as the set containing all values of type T.

- We model any prover $\mathcal{P}$, whether honest or adversarial, as an object in a set Pvr characterized by three public methods.
  - The verifier invokes $\mathcal{P}.\mathsf{message}()$ to generate a prover message if the prover is the moving party. We expect $\mathcal{P}.\mathsf{message}$ to return a message of the appropriate type.
  - When the verifier generates a message m as the moving party, the verifier sends the prover the message by invoking $\mathcal{P}.\mathsf{update}(m)$ receiving no return value, or rather the unit return value $()$.
  - The verifier allows the prover to regress and progress through the reductions by invoking $\mathcal{P}.\mathsf{way}()$ and receiving a value of type Way.

- Choosing notation to express reduction composition was not an obvious choosing. Intuitively upon executing one reduction, one simply transitions to the next. But when it comes to witness extraction the second reduction must return a witness to the first reduction. Communication both directions can be modelled by message passing, but formalizing with message passing became too complex. We instead chose to make the second reduction a subroutine of the first reduction. Reductions involve multiple parameters, including a prover, an instance, and also whatever reductions are to follow. Currying becomes natural in this setting and we adopt a functional notation involving higher-order functions. We curry functions passing each parameter with parentheses, e.g. to invoke f with arguments x and then y we write $f(x)(y)$.

  Despite our function notation, our functions are far from pure, and in fact quite the opposite as they are centered around the randomness of both the prover and verifier. Care must be taken, then, in interpretation of function invocation as follows: When a function is not pure, the function does not execute even partially (i.e. no β-reduction occurs) until one explicitly executes the function and assigns an identifier to the result. A function is explicitly executed with the left arrow pointing from the function to the output identifier.

- It only makes sense to discuss witnesses with respect to a particular instance. We will often, however, discuss witnesses leaving it implicit to which instance a witness belongs. For example, we may say that at the end of a reduction from language $L(I; W)$ to language $L'(I'; W')$ the prover holds a W' witness. We don't mean the prover may choose any value of type W'. Instead, if we reduce from instance $x \in I$ to instance $x' \in I'$ we mean the prover holds a witness for x'.

- When we write algorithms, the return value of a function is the value of the last expression in the function. The unit type and unit value are both denoted $()$.

- To compose a sequence of functions $\{f_i\}_{i=j}^k$ we write $\bigcirc_{i=j}^k(f_i)$. The functions to compose are located inside the parentheses immediately following $\bigcirc$. As such it is clear that $\bigcirc_{i=j}^k(f_i) \circ g$ for another function g means $\big(\bigcirc_{i=j}^k (f_i)\big) \circ g$ and not $\bigcirc_{i=j}^k((f_i) \circ g)$.

- We typeset types and variables in the Roman and Greek alphabets. Types are written capitalized in the Roman alphabet. These include enumeration types and their variants. All other symbols are written in the lower case Roman alphabet or the Greek alphabet. We use sans-serif font for the Roman alphabet if and only if the symbol is multi-letter. Sans-serif font is not to be interpreted as semantically different.

Here we mention two points for improvement on QUIRKs.

- The QUIRK model as developed has no regard for zero-knowledge. Conceptualizing zero-knowledge for QUIRKs seems a tricky task such that it is defined independently per QUIRK but is preserved on concatenation of QUIRKs into poly-QUIRKs.

- Intuitively we may allow the prover more freedom than regression, allowing the prover to choose not only between whether to regress or progress, but also what QUIRK to execute next in the case of progression. This would offer what is likely a new paradigm in interactive proofs. In contrast to typical interactive proofs where the sequence of prover and verifier moves are hardcoded, in this relaxed model the prover could choose during interaction which reduction to apply at what step, while still having complete freedom to regress and perhaps choose a different reduction after each regression. This framework would require adapting communication between the prover and verifier to agree not only on when to progress, but with which QUIRK to progress. The proof of extractability must also be reformulated, though the basic techniques underling and proof as well as the functional formalization seem still applicable. The task would largely be one of adapting definitions and notation.

2 QUIRKs

This section is organized as follows.

- In Section 2.1 we outline how reduction is performed for a QUIRK without allowing regression.

- In Section 2.2 we outline how reduction changes when we allow for regression.

- In Section 2.3 we define QUIRKs and related notions.
- In Section 2.4 list the properties of QUIRKs.

2.1 Reduction without regression

A QUIRK Q for reduction from a language $L(I;W)$ to a language $L'(I';W')$ consists of five functions

$$\big(\mathsf{message},\ \mathsf{instance},\ \mathsf{witness},\ \mathsf{extract},\ \mathsf{solve}\big)$$

We outline how the prover and verifier use the first three functions to reduce some instance $x \in I$ to some (potentially random) instance $x' \in I'$. An honest prover holds a witness $w \in W$ in hopes of reducing to a witness $w' \in W'$. The latter two functions solve and extract serve to justify security and are not invoked during reduction.

- The moving party invokes the *probabilistic* function Q.message and sends a message of some type M to the other party. We use binary valued utility functions Pmove and Vmove to indicate the moving party. If $\mathsf{Pmove}(Q) = 1$ then Q.message may depend on x and w. If $\mathsf{Vmove}(Q) = 1$ then Q.message accepts no arguments.
- Upon both parties holding a message $m \in M$, the *deterministic* function Q.instance is invoked by both parties to compute a new instance $x' \in I'$, which may depend on the current instance x as well as the message m.
- At this point an honest prover also invokes the *deterministic* function Q.witness accepting x, m, and w as arguments, in hopes the output is a new witness $w' \in W'$. Due to what is called *completeness error*, however, this may not be the case. To capture this circumstance we type the output of Q.witness as $\mathsf{Maybe}\langle W'\rangle$.
- Whether or not an honest prover has obtained a new witness, Q has finished execution, and the subsequent reduction is invoked. When we allow for regression the prover will be able to repeat the execution of Q as many times as necessary until obtaining a valid new witness.

We formalize this reduction in Algorithm 1, calling it δ to signify a change occurs in reducing from one language to the other. Despite δ not formally a component of Q, we identify it as $Q.\delta$ because it is fully determined by the components of Q, in particular by Q.message, Q.instance, and Q.witness. Shown in Algorithm 1, $Q.\delta$ is a higher-order function. We now examine the signature.

- Following the execution of δ there remain any number of subsequent reductions to perform. The composition of all subsequent reductions is passed to δ as first argument with signature $\mathsf{next}\colon I' \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$. The function next may be composed of not just reductions for other QUIRKS, but composed of a combination of extraction, solution, and reduction functions. Moreover, all of them will incorporate regression, which is why errors returned may have value $\mathsf{Err}(\mathsf{Regress})$.
- The instance $x \in I$ is passed as second argument.
- The prover $\mathcal{P} \in \mathsf{Pvr}$ is passed as third argument.
- The function next is tail called by δ with a new instance in I' and the prover $\mathcal{P}$. As next returns a value of type $\mathsf{Result}\langle T, \mathsf{Error}\rangle = \mathsf{Ok}(T) \mid \mathsf{Err}(\mathsf{Regress} \mid \mathsf{Fail})$, so does δ. If T is to be W', then next must be composed of extractors in order to extract W'. But as said, next may be composed of combinations of extraction, solution, and reduction functions, which may return results of other types T.

Algorithm 1 Reduction without regression

```
$\delta\langle T\rangle\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$
$\delta(\mathsf{next})(x)(\mathcal{P}) :=$
    **if** $\mathsf{Pmove}(Q)$
        $m \leftarrow \mathcal{P}.\mathsf{message}()$
    **if** $\mathsf{Vmove}(Q)$
        $m \leftarrow Q.\mathsf{message}()$
        $\mathcal{P}.\mathsf{update}(m)$
    $\mathsf{next}\big(Q.\mathsf{instance}(x)(m)\big)(\mathcal{P})$
```

Remark 1. *Regarding the design choice of* Q.message *for a verifier move accepting no arguments, we believe this to be natural for most cases. In exceptional cases this limitation can be circumvented by having* Q.message *sample a uniform string (e.g. a series of coin flips) and leaving* Q.instance *to compute the instance-dependent message.*

2.2 Reduction with regression

Regression is a matter of allowing the prover to regress and progress between reductions. Upon executing one QUIRK Q and prior to executing another QUIRK Q', the verifier queries the prover on a method called way to receive a value of type Way = Regress | Progress. If the prover responds with Regress then the verifier regresses to the point before Q and again queries the prover on way. If the prover responds with Progress then Q' is executed. Thus if a prover wishes to regress exactly $k \geq 0$ QUIRKs it will be queried on way $k + 1$ times returning Regress the first k times followed by Progress the last time.

Formalized in Algorithm 2 we denote regression with Γ. Similar to δ, the higher-function Γ accepts as first argument a function called next. Whereas with δ the function next has first argument with an instance type, with Γ the function next has first argument with a type variable H leaving Γ unassociated with any particular language. Though H is free it will resolve to a particular instance type when Γ is used in composition. The Γ process first queries the prover on way, and returns a regression error if the prover requests regression. If the prover requests progression, the function next is invoked, and if the return value is a regression error then by tail recursion we repeat the process. By this logic we capture the pattern of regression and progression between whatever QUIRK called Γ and whatever QUIRK is first encountered when executing next. The value returned by next is not a regression error if and only if next was fully executed until the end and the verifier made a final decision.

Note how Γ invoked on a function of type $H \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$ for any types H and T, yields a function of the same type. Since δ accepts a next function of such a type, composition $\delta \circ \Gamma$ with $H := I'$ is natural. Since $\delta \circ$ next returns a function of such a type, composition $\Gamma \circ \delta$ with $H := I$ is natural. In both cases, the signature remains that of δ.

Algorithm 2 Regression

```
$\Gamma\langle H, T\rangle : (H \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to H \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$
$\Gamma(\mathsf{next})(x)(\mathcal{P}) :=$
    **match** $w \leftarrow \mathcal{P}.\mathsf{way}()$
        $\mathsf{Regress} \Rightarrow \mathsf{Err}(\mathsf{Regress})$
        $\mathsf{Progress} \Rightarrow$
            **match** $r \leftarrow \mathsf{next}(x)(\mathcal{P})$
                $\mathsf{Err}(\mathsf{Regress}) \Rightarrow$
                    $r' \leftarrow \Gamma(\mathsf{next})(x)(\mathcal{P})$
                $\mathsf{Err}(\mathsf{Fail}) \Rightarrow \mathsf{Err}(\mathsf{Fail})$
                $\mathsf{Ok}(t) \Rightarrow \mathsf{Ok}(t)$
```

2.3 Definitions

Having introduced δ in Algorithm 1 and Γ in Algorithm 2, we are prepared to formalize QUIRKs, poly-QUIRKs, and associated definitions.

- QUIRKs are define in Definition 1
- Completeness and extractability errors, as well as expected extraction and solution times, are defined in Definition 2
- Poly-QUIRKs are defined in Definition 3
- Definition 2
- Definition 4

- Definition 5

Definition 1. *A* **QUIRK** *from language $L(I;W)$ to language $L'(I';W')$ consists of five functions*

$$\big(\mathsf{message},\ \mathsf{instance},\ \mathsf{witness},\ \mathsf{extract},\ \mathsf{solve}\big)$$

described as follows.

- ***Message.*** *The function* message *is probabilistic and outputs a message of some type M. Mono-QUIRKs can be categorized as* **prover-move** *or* **verifier-move** *and the distinction is captured by the signature of the* message *function. A prover-move QUIRK has* message *signature*

  $$\mathsf{message}\colon I \to W \to M$$

  A verifier-move QUIRK accepts no arguments and therefore has message *signature*

  $$\mathsf{message}\colon () \to M$$

- ***Instance.*** *The function* instance *is deterministic with signature*

  $$\mathsf{instance}\colon I \to M \to I'$$

- ***Witness.*** *The function* witness *is probabilistic with signature*

  $$\mathsf{witness}\colon I \to M \to W \to \mathsf{Maybe}\langle W'\rangle$$

  Implicitly, rather than enforced in signature, we assume that for any $x \in I$, $m \in M$, and $w \in W$ such that $(x;\ w) \in L$ and $\mathsf{witness}(x)(m)(w) = \mathsf{Yes}(w')$ for some $w' \in W'$, we have

  $$\big(\mathsf{instance}(x)(m);\ w'\big) \in L'(I';\ W') \tag{1}$$

  In other words, if witness *returns a new witness we assume it to be a witness for the new instance returned by* instance*. We may refer to this as the* consistency *assumption between the* instance *and* witness *functions.*

- ***Extract and solve.*** *The functions* extract *and* solve *are functions one can use to interact with a prover in effort to extract a witness of type W or a 'solution' of type S, respectively, as described next.*

  – *The type W is the set of all witnesses for all instances I in L. The function* extract *is passed an instance $x \in I$ as second argument, and returns an error (of type* Error*) or some witness value w (of type W). The task of extracting a witness, however, is only meaningful if w is constrained to be a witness for instance x. We therefore implicitly constrain* extract *to output a witness only for the instance given, though this constraint is not expressed in the signature.*

  – *The type S in the set-theoretic sense consists of elements we may call 'solutions' to the search problem of finding an $s \in S$. The function* solve *returns a result with an error (of type* Error*) or a solution (of type S).*

  $$\begin{aligned}\mathsf{extract}\colon \big(I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle\big) &\to I \to \mathsf{Pvr} \to \mathsf{Result}\langle W, \mathsf{Error}\rangle\\ \mathsf{solve}\colon \big(I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle\big) &\to I \to \mathsf{Pvr} \to \mathsf{Result}\langle S, \mathsf{Error}\rangle\end{aligned}$$

  An implicit assumption on how these functions operate is crucial. We describe how extract *operates and the description is naturally extended to* solve*. On arguments $\eta\colon I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle$, $x \in I$, and $\mathcal{P} \in \mathsf{Pvr}$, the function* extract *operates as follows. It immediately invokes $\delta(\eta)(x)(\mathcal{P})$ and examines the result, which is of type $\mathsf{Result}\langle W', \mathsf{Error}\rangle$. If the result holds the* Error *variant then* extract *returns that error. If the result holds the $\mathsf{Ok}\langle W'\rangle$ variant then* extract *may continue operating by and interacting with the prover as it chooses. Finally* extract *returns a result that holds the variant* Error *with value* Fail*, or holds the variant $\mathsf{Ok}\langle W\rangle$. Consequently,* extract *returns* Err(Regress) *if and only if δ does so on the initial invocation.*

QUIRK type notation. For a QUIRK Q we access the five functions with dot notation, for example writing $Q.\mathsf{message}$ to access the message function. When we wish to access a type T associated with a particular QUIRK Q, we use the type as a utility function and write $T(Q)$. For example, the signature of the witness function for a QUIRK Q may be written $I(Q) \to M(Q) \to W(Q) \to \mathsf{Maybe}\langle W'(Q)\rangle$. We may do the same for the languages, saying for example that a QUIRK Q reduces from language $L(Q)$ to language $L'(Q)$. Note that a language may be regarded as a type, consisting of all instances in the language. When a QUIRK is clear from context, we access all the associated types freely as done in Definition 1.

Definition 2. *For a QUIRK Q we define the following notions and associated notation. Let all η below have the signature $I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle$.*

- **Completeness error** *is the unique smallest constant denoted $\kappa(Q) \in \mathbb{R}$ such that the following inequality holds over all $x \in I$ and $w \in W$.*

$$\kappa(Q) \geq \Pr\left[v' = \mathsf{No} \;\middle|\; \begin{array}{l} \textbf{if } \mathsf{Pmove}(Q) \\ \quad m \leftarrow Q.\mathsf{message}(x)(w) \\ \textbf{if } \mathsf{Vmove}(Q) \\ \quad m \leftarrow Q.\mathsf{message}() \\ v' \leftarrow Q.\mathsf{witness}(x)(m)(w) \end{array} \right]$$

- **Extractability error** *is the unique smallest constant denoted $\epsilon(Q) \in \mathbb{R}$ such that the following inequality holds over all η, $x \in I$, and $\mathcal{P} \in \mathsf{Pvr}$.*

$$\begin{aligned} &\Pr\left[r \in \mathsf{Ok}(W) \;\middle|\; r \leftarrow Q.\mathsf{extract}(\eta)(x)(\mathcal{P})\right] + \Pr\left[r \in \mathsf{Ok}(S) \;\middle|\; r \leftarrow Q.\mathsf{solve}(\eta)(x)(\mathcal{P})\right] \\ &\quad \geq \Pr\left[r \in \mathsf{Ok}(W') \;\middle|\; r \leftarrow Q.\delta(\eta)(x)(\mathcal{P})\right] - \epsilon(Q) \end{aligned}$$

- **Expected extraction and solution times** *are the unique smallest constants denoted $\tau(Q.\mathsf{extract}) \in \mathbb{R}$ and $\tau(Q.\mathsf{solve}) \in \mathbb{R}$ such that the following hold over all η, $x \in I$, and $\mathcal{P} \in \mathsf{Pvr}$.*
  - *$Q.\mathsf{extract}(\eta)(x)(\mathcal{P})$ executes $Q.\delta(\eta)(x)(\mathcal{P})$ an expected number of times at most $\tau(Q.\mathsf{extract})$.*
  - *$Q.\mathsf{solve}(\eta)(x)(\mathcal{P})$ executes $Q.\delta(\eta)(x)(\mathcal{P})$ an expected number of times at most $\tau(Q.\mathsf{solve})$.*

  Note that by assumption $Q.\mathsf{extract}$ and $Q.\mathsf{solve}$ are constructed to each execute $Q.\delta$ at least once.

Definition 3. *A* **poly-QUIRK** *Q of length n is a sequence of $n \geq 1$ QUIRKs*

$$Q_0, Q_1, \ldots, Q_{n-1}$$

such that adjacent languages coincide, that is $L'(Q_{i-1}) = L(Q_i)$ for $1 \leq i \leq n-1$.

Poly-QUIRK type notation. We extend types as utility functions from QUIRKs to poly-QUIRKs by defining $T_i(Q) := T(Q_i)$ for a poly-QUIRK Q and an associated type T. When the poly-QUIRK is clear from context we simply write T_i.

Definition 4 ($Q.\mathsf{extract}$, $Q.\mathsf{solve}$, and $Q.\delta$)**.** *Let Q be a poly-QUIRK of length n. Define $Q.\mathsf{extract}$ as follows*

$$\begin{aligned} &Q.\mathsf{extract}\colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_0, \mathsf{Error}\rangle \\ &Q.\mathsf{extract} := \bigcirc_{j=0}^{n-1}\big(\Gamma \circ Q_j.\mathsf{extract}\big) \end{aligned}$$

For $i \in \{0, \ldots, n-1\}$ define $Q.\mathsf{solve}_i$ as follows

$$\begin{aligned} &Q.\mathsf{solve}_i\colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle S_i, \mathsf{Error}\rangle \\ &Q.\mathsf{solve}_i := \bigcirc_{j=0}^{i-1}\big(\Gamma \circ Q_j.\delta\big) \circ \big(\Gamma \circ Q_i.\mathsf{solve}\big) \circ \bigcirc_{j=i+1}^{n-1}\big(\Gamma \circ Q_j.\mathsf{extract}\big) \end{aligned}$$

Define $Q.\delta$ as follows

$$\begin{aligned} &Q.\delta\colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle \\ &Q.\delta := \bigcirc_{j=0}^{n-1}\big(\Gamma \circ Q_j.\delta\big) \end{aligned}$$

Definition 5 ($\mathcal{A}_j$)**.** *For a poly-QUIRK Q of length n we define the random variable $\mathcal{A}_j$ for $j \in \{0, \ldots, n-1\}$. The purpose of $\mathcal{A}_j$ is to capture the time complexity Q.*extract*, Q.*solve*$_i$, and $Q.\delta$ for any particular arguments. To do so $\mathcal{A}_j$ accepts one of these functions with arguments and counts the number of times $Q_j.\delta$ is executed. One may think of $\mathcal{A}_j$ as the number 'attempts' the prover makes at reduction $Q_j.\delta$, whether honestly or dishonestly, in effort to reach the end in a particular state such as holding a W_{j+1} witness.*

For example, for any $\eta\colon I_n \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle$, $x \in I_0$, and $\mathcal{P} \in \mathsf{Pvr}$ the number of times $Q_j.\delta$ is executed during execution of $Q.\mathsf{extract}(\eta)(x)(\mathcal{P})$ is denoted

$$\mathcal{A}_j\Big(Q.\mathsf{extract}(\eta)(x)(\mathcal{P})\Big)$$

More generally $\mathcal{A}_j$ is well defined on any function that involves $Q_j.\delta$. For example, since Γ composes with other reductions without affecting the signature, one can invoke $\mathcal{A}_j$ for $j \in \{0, \ldots, n-1\}$ on $Q.\mathsf{extract}$, $Q.\mathsf{solve}_i$, and $Q.\delta$ as defined in Definition 4 but stripped of one or more instances of Γ. In Theorem 1 we only invoke $\mathcal{A}_j$ on $Q.\mathsf{extract}$, $Q.\mathsf{solve}_i$, and $Q.\delta$. In the proof of Theorem 1 we additionally invoke $\mathcal{A}_j$ on these functions stripped of the leading Γ.

2.4 Properties

Theorem 1. *Suppose Q is a poly-QUIRK of length n. Suppose also η is a function of signature $I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle$, and $x \in I_0$.*

- ***Completeness.*** *Suppose η is poly-QUIRK, such that execution of η begins with execution of Γ. Consider a prover $\mathcal{P} \in \mathsf{Pvr}$ that may be given values of types I_0 and $\mathsf{Maybe}\langle W_0\rangle$ prior to execution of $Q.\delta(\eta)(x)(\mathcal{P})$. One may construct $\mathcal{P}$ to satisfy the following.*
  - ***Probability of completion.*** *Every time before η is invoked throughout the reduction of $Q.\delta(\eta)(x)(\mathcal{P})$ we say $\mathcal{P}$ has* **arrived** *at η. Arrival at η may occur multiple times and is always preceded by execution of $Q_{n-1}.\delta$. We consider $\mathcal{P}$ to be successful if it ever arrives at η holding a W_n witness.*
    - *Given x and $v = \mathsf{No}$, $\mathcal{P}$ arrives at η holding a W_n witness with probability* 0.
    - *Given x and $v = \mathsf{Yes}(w)$ for $w \in W_0$, $\mathcal{P}$ arrives at η holding a W_n witness with probability* 1.
  - ***Expected time of completion.***
    - *If $\mathcal{P}$ is given x and $v = \mathsf{No}$ then*

$$\forall j \in \{0, \ldots, n-1\},\ \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\delta(\eta)(x)(\mathcal{P})\Big)\Big] = 0$$

    - *If $\mathcal{P}$ is given x and $v = \mathsf{Yes}(w)$ for $w \in W_0$ then*

$$\forall j \in \{0, \ldots, n-1\},\ \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\delta(\eta)(x)(\mathcal{P})\Big)\Big] \leq \frac{1}{1-\kappa_j}$$

- ***Extractability.*** *For any $\mathcal{P} \in \mathsf{Pvr}$*
  - ***Probability of extraction and solution.***

$$\Pr\Big[r \in \mathsf{Ok}(W_0)\ \Big|\ r \leftarrow Q.\mathsf{extract}(\eta)(x)(\mathcal{P})\Big] + \sum_{i=0}^{n-1} \Pr\Big[r \in \mathsf{Ok}(S_i)\ \Big|\ r \leftarrow Q.\mathsf{solve}_i(\eta)(x)(\mathcal{P})\Big]$$
$$\geq \Pr\Big[r \in \mathsf{Ok}(W_n)\ \Big|\ r \leftarrow Q.\delta(\eta)(x)(\mathcal{P})\Big] - \sum_{i=0}^{n-1} \epsilon_i \cdot \mathbb{E}\Big[\mathcal{A}_i(\Delta(\eta)(x)(\mathcal{P}))\Big]$$

  - ***Expected time of extraction and solution.***

$$\forall j \in \{0, \ldots, n-1\},\ \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\mathsf{extract}(\eta)(x)(\mathcal{P})\Big)\Big] \leq \left(\prod_{k=0}^{j} \tau(Q_j.\mathsf{extract})\right) \cdot \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\delta(\eta)(x)(\mathcal{P})\Big)\Big]$$

For $i \in \{0, \dots, n-1\}$

$$\forall j \in \{0, \dots, i-1\}, \mathbb{E}\Big[A_j\Big(Q.\mathsf{solve}_i(\eta)(x)(\mathcal{P})\Big)\Big] \leq \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\delta(\eta)(x)(\mathcal{P})\Big)\Big]$$

$$\forall j \in \{i, \dots, n-1\}, \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\mathsf{solve}_i(\eta)(x)(\mathcal{P})\Big)\Big]$$

$$\leq \tau\big(Q_i.\mathsf{solve}\big)\left(\prod_{k=i+1}^{j} \tau\big(Q_k.\mathsf{extract}\big)\right) \cdot \mathbb{E}\Big[\mathcal{A}_j\Big(Q.\delta(\eta)(x)(\mathcal{P})\Big)\Big]$$

Proof. Completeness properties are proven in Section 3. Extractability properties are proven in Section 4. □

Corollary 1. *Extractability for a poly-QUIRK Q of length n in Theorem 1 is concerned only with extracting a W_0 witness. At least that is what the signature of $Q.\mathsf{extract}$ suggests. But in fact the event that $Q.\mathsf{extract}$ extracts a W_0 witness implies that $Q.\mathsf{extract}$ has extracted a W_i witness for $i \in \{1, \dots, n-1\}$ as well, and in particular the instances to which these witnesses belong are the instances of the final reduction trace. Therefore we believe extracting a witness for the input instance, thus implicitly extracting witnesses for all intermediate instances, is an appropriate definition of extractability for Q.*

3 Proof of completeness

We first lay out construction of a prover $\mathcal{P} \in \mathsf{Pvr}$ that satisfies the completeness properties of Theorem 1. To prove correctness of $\mathcal{P}$, we first prove the case when $\mathcal{P}$ is given values $x \in I_0$ and $v = \mathsf{No}$. Then we prove the case when $\mathcal{P}$ is given values $x \in I_0$ and $v = \mathsf{Yes}(w)$ for $w \in W_0$.

Honest prover reduction. The definition of Pvr as consisting of objects with three particular public methods naturally leads us to construct an honest prover $\mathcal{P} \in \mathsf{Pvr}$ by defining a class-like template for honest prover objects. In Algorithm 3 we lay out such a template $\mathcal{P}_Q$ which can be instantiated with two arguments of types I_0 and $\mathsf{Maybe}\langle W_0\rangle$. On initialization the private method init is immediately invoked to initialize the state.

After initialization the state of an object consists of four values with types

$$i\colon \mathbb{N},\ y\colon \mathsf{Maybe}\langle I_i\rangle,\ v\colon \mathsf{Maybe}\langle W_i\rangle,\ \mathsf{valid}\colon \{0,1\}$$

We are informally using dependent types, but in hopes of clarity on Algorithm 3 we are sure to keep consistency between them. Suppose we wish to reassign i along with new Yes values for y and v. One encounters the trouble that reassigning one before the others results in inconsistency, unless one manages to assign them all simultaneously. To circumvent this issue we first assign value No to both y and v, then reassign i, and only then reassign Yes values to y and v. This works in the sense that with type casting, value No may be interpreted as a value of type $\mathsf{Maybe}\langle I_i\rangle$ or $\mathsf{Maybe}\langle W_i\rangle$ for any i. Consistency of dependent types is in fact the sole reason we type y using Maybe, though v must be typed using Maybe since the witness may not be present on instantiation.

Algorithm 3 Honest prover template

```
$\mathcal{P}_Q :=$
    $i \colon \mathbb{N}$
    $y \colon \mathsf{Maybe}\langle I_i \rangle$
    $v \colon \mathsf{Maybe}\langle W_i \rangle$
    $\mathsf{valid} \colon \{0,1\}$
    $- \mathsf{init} \colon I_0 \to \mathsf{Maybe}\langle W_0 \rangle \to ()$
    $- \mathsf{init}(x_0)(v_0) :=$
        $i := 0$
        **match** $v_0$
            $\mathsf{No} \Rightarrow$
                $y := \mathsf{Yes}(x_0),\ v := \mathsf{No}$
                $\mathsf{valid} := 0$
            $\mathsf{Yes}(w_0) \Rightarrow$
                $y := \mathsf{Yes}(x_0),\ v := \mathsf{Yes}(w_0)$
                $\mathsf{valid} := 1$
    $- \mathsf{check} \colon I_{i+1} \to \mathsf{Maybe}\langle W_{i+1} \rangle \to ()$
    $- \mathsf{check}(x')(v') :=$
        **match** $v'$
            $\mathsf{No} \Rightarrow \mathsf{valid} := 0$
            $\mathsf{Yes}(w') \Rightarrow$
                $y := \mathsf{No},\ v := \mathsf{No}$
                $i := i + 1$
                $y := \mathsf{Yes}(x'),\ v := \mathsf{Yes}(w')$
                $\mathsf{valid} := 1$
    $\mathsf{message} \colon () \to M_j$
    $\mathsf{message}() :=$
        $x := \mathsf{unwrap}(y),\ w := \mathsf{unwrap}(v)$
        $m := Q_i.\mathsf{message}(x)(w)$
        $x' := Q_i.\mathsf{instance}(x)(m)$
        $v' := Q_i.\mathsf{witness}(x)(m)(w)$
        $\mathsf{check}(x')(v')$
        $m$
    $\mathsf{update} \colon M_i \to ()$
    $\mathsf{update}(m) :=$
        $x := \mathsf{unwrap}(y),\ w := \mathsf{unwrap}(v)$
        $x' := Q_i.\mathsf{instance}(x)(m)$
        $v' := Q_i.\mathsf{witness}(x)(m)(w)$
        $\mathsf{check}(x')(v')$
    $\mathsf{way} \colon () \to \mathsf{Way}$
    $\mathsf{way}() :=$
        **match** $\mathsf{valid}$
            $0 \Rightarrow$
                $\mathsf{valid} := 1$
                $\mathsf{Regress}$
            $1 \Rightarrow \mathsf{Progress}$
```

We turn to proving correctness of $\mathcal{P}_Q$. The reduction $Q.\delta(\eta) = \bigcirc_{j=0}^{n-1}\big(\Gamma \circ Q_j.\delta\big) \circ \eta$ executes alternating functions Γ and $Q_j.\delta$ followed by η. While all n instances of Γ are the same, let us rewrite $Q.\delta(\eta)$ as $\bigcirc_{j=0}^{n-1}\big(\Gamma_j \circ Q_j.\delta\big) \circ \eta$ with $\Gamma_j = \Gamma$ for clarity. By assumption execution of η starts with execution of Γ. Let us identify the Γ at the start of η as Γ_n.

Instantiating without a witness. Suppose we invoke $\mathcal{P}_Q(x)(\mathsf{No})$ to get $\mathcal{P} \in \mathsf{Pvr}$. Then by examination of init the state of $\mathcal{P}$ has $i = 0$ and $\mathsf{valid} = 0$. On execution of Γ_0 the call to $\mathcal{P}.\mathsf{way}()$ returns Regress, which Γ_0 receives to then return value $\mathsf{Err}(\mathsf{Regress})$. Therefore $Q.\delta$ returns $\mathsf{Err}(\mathsf{Regress})$ never reaching state with $i = n$ and $\mathsf{valid} = 1$. Moreover, $Q.\delta$ never executes $Q_j.\delta$ for any $j \geq 0$.

Instantiating with a witness. Suppose we invoke $\mathcal{P}_Q(x)(\mathsf{Yes}(w))$ for $w \in W_0$ to get $\mathcal{P} \in \mathsf{Pvr}$. Then by examination of init the state of $\mathcal{P}$ has $i = 0$, $y \in \mathsf{Yes}(I_0)$, $v \in \mathsf{Yes}(W_0)$, and $\mathsf{valid} = 1$. This state is a 'valid 0-state' as defined next.

Definition 6. *For $j \geq 0$ we define a* **valid j-state** *to be one of the form*

$$i = j,\ y \in \mathsf{Yes}(I_j),\ v \in \mathsf{Yes}(W_j),\ \mathsf{valid} = 1$$

and an **invalid j-state** *to be one of the form*

$$i = j,\ y \in \mathsf{Yes}(I_j),\ v \in \mathsf{Yes}(W_j),\ \mathsf{valid} = 0$$

Claim 1. *Suppose prior to executing $\Gamma_j \circ Q_j.\delta$ for $j \in \{0, \ldots, n-1\}$, $\mathcal{P}$ holds a valid j-state.*

- *With probability at least $1 - \kappa_j$ upon executing $\Gamma_j \circ Q_j.\delta$ we arrive at Γ_{j+1} in a valid $(j+1)$-state.*
- *With probability at most κ_j upon executing $\Gamma_j \circ Q_j.\delta$ we arrive at Γ_j in a valid j-state.*

Proof. Suppose at the start of Γ_j we have a valid j-state. Executing Γ_j we arrive at $Q_j.\delta$ because with $\mathsf{valid} = 1$ $\mathcal{P}.\mathsf{way}()$ returns $\mathsf{Progress}$. Then $Q_j.\delta$ invokes $\mathcal{P}.\mathsf{message}()$ for a prover message $m \in M_j$ if $\mathsf{Pmove}(Q) = 1$, or $Q_j.\delta$ invokes $\mathcal{P}.\mathsf{update}(m)$ on verifier message $m \in M_j$ if $\mathsf{Vmove}(Q) = 1$. Regardless which, due to a valid j-state we may unwrap y and v for values $x \in I_j$ and $w \in W_j$. A new instance $x' \in I_{j+1}$ and potential witness $v' \in \mathsf{Maybe}\langle W_{j+1}\rangle$ are computed, and then $\mathsf{check}(x')(v')$ is invoked and we continue in one of two ways.

- If $v' \in \mathsf{Yes}(W_{j+1})$ then we arrive at Γ_{j+1} in a valid $(j+1)$-state.
- If $v' = \mathsf{No}$ then we arrive at Γ_{j+1} in an invalid j-state. Executing Γ_{j+1} we arrive at Γ_j in a valid j-state because $\mathcal{P}.\mathsf{way}()$ flips valid from 0 to 1 and returns $\mathsf{Regress}$.

By the definition of completeness error, see Definition 2, the probability over m that $v' = \mathsf{No}$ is at most κ_j. The claim follows. $\square$

Recall that instantiation puts $\mathcal{P}$ in a valid 0-state prior to any execution, which will begin with Γ_0. By Claim 1 we infer that upon executing $\Gamma_0 \circ Q_0.\delta$ at most an expected $1/(1-\kappa_0)$ times we arrive at Γ_1 in a valid 1-state. Continuing like this for $j \in \{1, \dots, n-1\}$ we infer that upon executing $\Gamma_j \circ Q_j$ at most an expected $1/(1-\kappa_j)$ times we arrive at Γ_{j+1} in a valid $(j+1)$-state. Finally we arrive at Γ_n in a valid n-state. Therefore $\mathcal{P}$ arrives at η holding a W_n witness, and in the process executing $Q_j.\delta$ for $k \in \{0, \dots, n-1\}$ an expected number of times at most $1/(1-\kappa_j)$.

4 Proof of extractability

We will define new notation and use it to restate the extractability results of Theorem 1 with Theorem 2. We also state Lemma 1 to help in proving Theorem 2. In Section 4.1 we prove Lemma 1 and in Section 4.2 we prove Theorem 2.

Throughout the proof we invoke an additional public method on prover objects in Pvr not mentioned previously. We are able to clone the state of a prover $\mathcal{P} \in \mathsf{Pvr}$ with $\mathcal{P}.\mathsf{clone}()$, receiving back a new object in Pvr. While this ability to clone the state of a prover is often used inside extraction and solution functions, in the case of this proof it is instead used for analytical purposes. Since $\mathcal{P}$ is stateful we cannot assume that $\mathcal{P}$ is in the same state prior to a reduction as after the reduction. We may clone $\mathcal{P}$ before or after a reduction to capture the prover's current state.

Definition 7 defines new notation. For example, we rename $Q.\delta$ as Δ for ease of differentiating $Q.\delta$ and $Q_j.\delta$ for $j \in \{0, \dots, n-1\}$. We'd like to similarly rename $Q.\mathsf{extract}$ as $\mathsf{Extract}$ but doing so would violate the convention of capitalization reserved for types in Roman alphabet. We use the Greek alphabet instead. Note $Q.\delta$ becomes Δ, $Q.\mathsf{extract}$ becomes Φ, and $Q.\mathsf{solve}_i$ becomes Ψ_i. The reason we also define the prime notation versions not involving Q_0 is for use in induction.

Definition 7. *For $i \in \{0, \dots, n-1\}$ we define ϕ_i, ψ_i, δ_i as follows*

$$\phi_i \coloneqq Q_i.\mathsf{extract}, \quad \psi_i \coloneqq Q_i.\mathsf{solve}, \quad \delta_i \coloneqq Q_i.\delta$$

We define Φ', Φ as follows

$$\begin{aligned}
&\Phi' \colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_1, \mathsf{Error}\rangle \\
&\Phi' \coloneqq \bigcirc_{j=1}^{n-1}\big(\Gamma \circ \phi_j\big) \\
&\Phi \colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_0, \mathsf{Error}\rangle \\
&\Phi \coloneqq \Gamma \circ \phi_0 \circ \Phi' = \bigcirc_{j=0}^{n-1}\big(\Gamma \circ \phi_j\big)
\end{aligned}$$

For $i \in \{1, \dots, n-1\}$ we define Ψ'_i and for $i \in \{0, \dots, n-1\}$ we define Ψ_i as follows

$$\begin{aligned}
&\Psi'_i \colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle S_i, \mathsf{Error}\rangle \\
&\Psi'_i \coloneqq \bigcirc_{j=1}^{i-1}\big(\Gamma \circ \delta_j\big) \circ \big(\Gamma \circ \psi_i\big) \circ \bigcirc_{j=i+1}^{n-1}\big(\Gamma \circ \phi_j\big) \\
&\Psi_i \colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle S_i, \mathsf{Error}\rangle \\
&\Psi_0 \coloneqq \Gamma \circ \psi_0 \circ \Phi' = \big(\Gamma \circ \psi_0\big) \circ \bigcirc_{j=1}^{n-1}\big(\Gamma \circ \phi_j\big) \\
&\Psi_{i>0} \coloneqq \Gamma \circ \delta_0 \circ \Psi'_i = \bigcirc_{j=0}^{i-1}\big(\Gamma \circ \delta_j\big) \circ \big(\Gamma \circ \psi_i\big) \circ \bigcirc_{j=i+1}^{n-1}\big(\Gamma \circ \phi_j\big)
\end{aligned}$$

We define Δ', Δ as follows

$$\begin{aligned}
&\Delta' \colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle \\
&\Delta' \coloneqq \bigcirc_{j=1}^{n-1}\big(\Gamma \circ \delta_j\big) \\
&\Delta \colon (I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle \\
&\Delta \coloneqq \Gamma \circ \delta_0 \circ \Delta' = \bigcirc_{j=0}^{n-1}\big(\Gamma \circ \delta_j\big)
\end{aligned}$$

We define β, β' as follows

$$\begin{aligned}
&\beta'\langle T\rangle \colon (I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to I_1 \to \mathsf{Pvr} \to \mathbb{R} \\
&\beta'(f)(x')(\mathcal{P}) \coloneqq \Pr\Big[r \in \mathsf{Ok}\langle T\rangle \,\Big|\, r \leftarrow f(x')(\mathcal{P})\Big] \\
&\beta\langle T\rangle \colon (I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathbb{R} \\
&\beta(f)(x)(\mathcal{P}) \coloneqq \Pr\Big[r \in \mathsf{Ok}\langle T\rangle \,\Big|\, r \leftarrow f(x)(\mathcal{P})\Big]
\end{aligned}$$

For $j \in \{1, \ldots, n-1\}$ we define α'_j and for $j \in \{0, \ldots, n-1\}$ we define α_j as follows

$$\begin{aligned}
&\alpha'_j\langle T\rangle \colon (I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to I_1 \to \mathsf{Pvr} \to \mathbb{R} \\
&\alpha'_j(f)(x')(\mathcal{P}) \coloneqq \mathbb{E}\Big[\mathcal{A}_j(f(x')(\mathcal{P}))\Big] \\
&\alpha_j\langle T\rangle \colon (I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathbb{R} \\
&\alpha_j(f)(x)(\mathcal{P}) \coloneqq \mathbb{E}\Big[\mathcal{A}_j(f(x)(\mathcal{P}))\Big]
\end{aligned}$$

Using Definition 7 we rewrite the extractability results of Theorem 1. Henceforth we write η without re-introduction assuming it to be a function of signature $I_n \to \mathsf{Pvr} \to \mathsf{Result}\langle W_n, \mathsf{Error}\rangle$.

Theorem 2 (Theorem 1 using Definition 7)**.** *For Q a poly-QUIRK of length n the following hold.*

$$\beta(\Phi(\eta)) + \sum_{i=0}^{n-1} \beta(\Psi_i(\eta)) \geq \beta(\Delta(\eta)) - \sum_{i=0}^{n-1} \chi_i \cdot \alpha_i(\Delta(\eta)) \tag{2}$$

$$\forall j \in \{0, \ldots, n-1\},\ \alpha_j(\Phi) \leq \left(\prod_{k=0}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta(\eta)) \tag{3}$$

For $i \in \{0, \ldots, n-1\}$

$$\forall j \in \{0, \ldots, i-1\},\ \alpha_j(\Psi_i(\eta)) \leq \alpha_j(\Delta(\eta)) \tag{4}$$

$$\forall j \in \{i, \ldots, n-1\},\ \alpha_j(\Psi_i(\eta)) \leq \tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta(\eta)) \tag{5}$$

The following lemma will be proven in Section 4.1. In Section 4.2 we utilize Lemma 1 to prove Theorem 2.

Lemma 1 (Extractability without regression)**.** *For Q a poly-QUIRK of length n the following hold.*

$$\beta\big(\phi_0(\Phi'(\eta))\big) + \beta\big(\psi_0(\Phi'(\eta))\big) + \sum_{i=1}^{n-1} \beta\big(\delta_0(\Psi'_i(\eta))\big) \geq \beta\big(\delta_0(\Delta'(\eta))\big) - \sum_{i=0}^{n-1} \chi_i \cdot \alpha_i\big(\delta_0(\Delta'(\eta))\big) \tag{6}$$

$$\forall j \in \{0, \ldots, n-1\},\ \alpha_j\big(\phi_0(\Phi'(\eta))\big) \leq \left(\prod_{k=0}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\delta_0(\Delta'(\eta))) \tag{7}$$

$$\forall j \in \{0, \ldots, n-1\},\ \alpha_j\big(\psi_0(\Phi'(\eta))\big) \leq \tau(\psi_0)\Big(\prod_{k=1}^{j} \tau(\phi_k)\Big) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big) \tag{8}$$

For $i \in \{1, \ldots, n-1\}$

$$\forall j \in \{0, \ldots, i-1\},\ \alpha_j\big(\delta_0(\Psi'_i(\eta))\big) \le \alpha_j\big(\delta_0(\Delta'(\eta))\big) \tag{9}$$

$$\forall j \in \{i, \ldots, n-1\},\ \alpha_j\big(\delta_0(\Psi'_i(\eta))\big) \le \tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big) \tag{10}$$

4.1 Extraction and solution without regression

To prove Lemma 1 for $n \ge 2$ we proceed assuming Theorem 2 for $n-1 \ge 1$. We state this induction assumption in the following claim. For the case $n = 1$ we argue the inductive assumption can be extended to $n - 1 = 0$.

Claim 2. *For Q a poly-QUIRK of length $n \ge 1$ the following hold.*

$$\beta(\Phi'(\eta)) + \sum_{i=1}^{n-1} \beta(\Psi'_i(\eta)) \ge \beta(\Delta'(\eta)) - \sum_{i=1}^{n-1} \chi_i \cdot \alpha_i(\Delta'(\eta)) \tag{11}$$

$$\forall j \in \{1, \ldots, n-1\},\ \alpha_j(\Phi'(\eta)) \le \left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta'(\eta)) \tag{12}$$

For $i \in \{1, \ldots, n-1\}$

$$\forall j \in \{1, \ldots, i-1\},\ \alpha_j(\Psi'_i(\eta)) \le \alpha_j(\Delta'(\eta)) \tag{13}$$

$$\forall j \in \{i, \ldots, n-1\},\ \alpha_j(\Psi'_i(\eta)) \le \tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta'(\eta)) \tag{14}$$

Proof. For $n \ge 2$ these equations hold by Theorem 2 for $n-1 \ge 1$. For $n = 1$ we cannot invoke Theorem 2 for $n - 1 = 0$, but we observe these equations trivially hold by the fact $\Phi'(\eta) = \Psi'_i(\eta) = \Delta'(\eta) = \eta$, and they reduce to $\beta(\eta) = \beta(\eta)$. ☐

4.1.1 red and properties

We introduce a function red and claim two properties.

Definition 8 (red)**.** *The function* red *(short for 'reduce') accepts a function f, an instance $x \in I_0$, and a prover $\mathcal{P} \in$ Pvr. With f some function that accepts a new instance $x' \in I_1$ and the prover in a new state $\mathcal{P}'$, the value* $\mathsf{red}(f)(x)(\mathcal{P})$ *is the expected value of f, had by iterating over all possible new instances x' and new prover states $\mathcal{P}'$, and multiplying the probability of x' and $\mathcal{P}'$ by the value $f(x')(\mathcal{P}')$.*

$$\mathsf{red}\colon (I_1 \to \mathsf{Pvr} \to \mathbb{R}) \to I_0 \to \mathsf{Pvr} \to \mathbb{R}$$

$$\mathsf{red}(f)(x)(\mathcal{P}) := \sum_{x' \in I_1} \sum_{P' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c|l} & \textbf{if } \mathsf{Pmove}(Q_0) \\ & \quad m \leftarrow \mathcal{P}.\mathsf{message}() \\ Q_0.\mathsf{instance}(m)(x) = x' & \textbf{if } \mathsf{Vmove}(Q_0) \\ p = \mathcal{P}' & \quad m \leftarrow Q_0.\mathsf{message}() \\ & \quad \mathcal{P}.\mathsf{update}(m) \\ & p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \cdot f(x')(\mathcal{P}')$$

Claim 3 (red linearity)**.** *For $f, g\colon$ Pvr $\to \mathbb{R}$ and $\alpha \in \mathbb{R}$ we see* red *to be linear, that is*

$$\lambda_0(\mathcal{P}) := f(\mathcal{P}) + g(\mathcal{P}) \qquad\qquad \lambda_1(\mathcal{P}) := \alpha \cdot f(\mathcal{P})$$
$$\mathsf{red}(\lambda_0) = \mathsf{red}(f) + \mathsf{red}(g) \qquad\qquad \mathsf{red}(\lambda_1) = \alpha \cdot \mathsf{red}(f)$$

Proof. Linearity is evident by examination of red and properties of real arithmetic, including distributivity of multiplication over addition. □

Claim 4 (red inequality)**.** *For* $f, g\colon I_1 \to \mathsf{Pvr} \to \mathbb{R}$

$$f \geq g \implies \mathsf{red}(f) \geq \mathsf{reg}(g)$$

where inequality between two real-valued functions means inequality between their evaluations for all possible arguments.

Proof. Inequality is evident by examination of red and properties of real arithmetic. □

4.1.2 Extending with reduction

We use the function red to relate the execution of functions $\Phi'(\eta)$, $\Psi'_i(\eta)$, and $\Delta'(\eta)$ to the execution of reduction δ_0 followed by those functions, that is $\delta_0(\Phi'(\eta))$, $\delta_0(\Psi'_i(\eta))$, and $\delta_0(\Delta'(\eta))$.

Claim 5.

$$\begin{aligned} \mathsf{red}\big(\beta(\Phi'(\eta))\big) &= \beta\big(\delta_0(\Phi'(\eta))\big) \\ \forall i \in \{1, \ldots, n-1\},\ \mathsf{red}\big(\beta(\Psi'_i(\eta))\big) &= \beta\big(\delta_0(\Psi'_i(\eta))\big) \\ \mathsf{red}\big(\beta(\Delta'(\eta))\big) &= \beta\big(\delta_0(\Delta'(\eta))\big) \end{aligned}$$

For $j \in \{1, \ldots, n-1\}$

$$\begin{aligned} \mathsf{red}\big(\alpha_j(\Phi'(\eta))\big) &= \alpha_j\big(\delta_0(\Phi'(\eta))\big) \\ \forall i \in \{1 \ldots, n-1\},\ \mathsf{red}\big(\alpha_j(\Psi'_i(\eta))\big) &= \alpha_j\big(\delta_0(\Psi'_i(\eta))\big) \\ \mathsf{red}\big(\alpha_j(\Delta'(\eta))\big) &= \alpha_j\big(\delta_0(\Delta'(\eta))\big) \end{aligned}$$

Proof. These equalities hold by Definition 7 and Definition 8, as shown below for the case of $\mathsf{red}\big(\beta(\Phi'(\eta))\big)$ and $\beta\big(\delta_0(\Phi'(\eta))\big)$, and can be shown similarly for the other cases.

$$\begin{aligned} &\mathsf{red}\big(\beta(\Phi'(\eta))\big)(x)(\mathcal{P}) \\ &= \sum_{x' \in I_1} \sum_{P' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c|l} Q_0.\mathsf{instance}(m)(x) = x' & \textbf{if } \mathsf{Pmove}(Q_0) \\ p = \mathcal{P}' & \quad m \leftarrow \mathcal{P}.\mathsf{message}() \\ & \textbf{if } \mathsf{Vmove}(Q_0) \\ & \quad m \leftarrow Q_0.\mathsf{message}() \\ & \quad \mathcal{P}.\mathsf{update}(m) \\ & p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \cdot \Pr\Big[r \in \mathsf{Ok}\langle W_n \rangle \;\Big|\; r \leftarrow \Phi'(\eta)(x')(\mathcal{P}')\Big] \\ &= \Pr\left[\begin{array}{c|l} r \in \mathsf{Ok}\langle W_n \rangle & \textbf{if } \mathsf{Pmove}(Q_0) \\ & \quad m \leftarrow \mathcal{P}.\mathsf{message}() \\ & \textbf{if } \mathsf{Vmove}(Q_0) \\ & \quad m \leftarrow Q_0.\mathsf{message}() \\ & \quad \mathcal{P}.\mathsf{update}(m) \\ & r \leftarrow \Phi'(\eta)\big(Q_0.\mathsf{instance}(m)(x)\big)(\mathcal{P}) \end{array}\right] \\ &= \Pr\Big[r \in \mathsf{Ok}\langle W_n \rangle \;\Big|\; r \leftarrow \delta_0(\Phi'(\eta))(x)(\mathcal{P})\Big] = \beta\big(\delta_0(\Phi'(\eta))\big)(x)(\mathcal{P}) \end{aligned}$$

□

Claim 6.

$$\beta\big(\delta_0(\Phi'(\eta))\big) + \sum_{i=1}^{n-1} \beta\big(\delta_0(\Psi'_i(\eta))\big) \geq \beta\big(\delta_0(\Delta'(\eta))\big) - \sum_{i=1}^{n-1} \chi_i \cdot \alpha_i\big(\delta_0(\Delta'(\eta))\big) \tag{15}$$

$$\forall j \in \{1, \ldots, n-1\},\ \alpha_j\big(\delta_0(\Phi'(\eta))\big) \leq \left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big) \tag{16}$$

For $i \in \{1, \ldots, n-1\}$

$$\forall j \in \{1, \ldots, i-1\},\ \alpha_j\big(\delta_0(\Psi'_i(\eta))\big) \leq \alpha_j\big(\delta_0(\Delta'(\eta))\big) \tag{17}$$

$$\forall j \in \{i, \ldots, n-1\},\ \alpha_j\big(\delta_0(\Psi'_i(\eta))\big) \leq \tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big) \tag{18}$$

Proof. We prove each of the four equations, listing the justifying claims on the right.

Proving Equation (15).

$$\begin{aligned}
\beta\big(\delta_0(\Phi'(\eta))\big) + \sum_{i=1}^{n-1} \beta\big(\delta_0(\Psi'_i(\eta))\big) &= \mathsf{red}\big(\beta(\Phi'(\eta))\big) + \sum_{i=1}^{n-1} \mathsf{red}\big(\beta(\Psi'_i(\eta))\big) && \text{by Claim 5} \\
&= \mathsf{red}\left(\beta(\Phi'(\eta)) + \sum_{i=1}^{n-1} \beta(\Psi'_i(\eta))\right) && \text{by Claim 3} \\
&\geq \mathsf{red}\left(\beta(\Delta'(\eta)) - \sum_{i=1}^{n-1} \chi_i \cdot \alpha_i(\Delta'(\eta))\right) && \text{by Claim 4 with Equation (11)} \\
&= \mathsf{red}\big(\beta(\Delta'(\eta))\big) - \sum_{i=1}^{n-1} \chi_i \cdot \mathsf{red}\big(\alpha_i(\Delta'(\eta))\big) && \text{by Claim 3} \\
&= \beta\big(\delta_0(\Delta'(\eta))\big) - \sum_{i=1}^{n-1} \chi_i \cdot \alpha_i\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 5}
\end{aligned}$$

Proving Equation (16). For $j \in \{1, \ldots, n-1\}$

$$\begin{aligned}
\alpha_j\big(\delta_0(\Phi'(\eta))\big) &= \mathsf{red}\big(\alpha_j(\Phi'(\eta))\big) && \text{by Claim 5} \\
&\leq \mathsf{red}\left(\left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta'(\eta))\right) && \text{by Claim 4 with Equation (12)} \\
&= \left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \mathsf{red}\big(\alpha_j(\Delta'(\eta))\big) && \text{by Claim 3} \\
&= \left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 5}
\end{aligned}$$

Proving Equation (17). For $i \in \{1, \ldots, n-1\}$ and $j \in \{1, \ldots, i-1\}$

$$\begin{aligned}
\alpha_j\big(\delta_0(\Psi'(\eta))\big) &= \mathsf{red}\big(\alpha_j(\Psi'(\eta))\big) && \text{by Claim 5} \\
&\leq \mathsf{red}\big(\alpha_j(\Delta'(\eta))\big) && \text{by Claim 4 with Equation (13)} \\
&= \alpha_j(\Delta'(\eta)) && \text{by Claim 5}
\end{aligned}$$

Proving Equation (18). For $i \in \{1, \ldots, n-1\}$ and $j \in \{i, \ldots, n-1\}$

$$\begin{aligned}
\alpha_j\big(\delta_0(\Psi'(\eta))\big) &= \mathsf{red}\big(\alpha_j(\Psi'(\eta))\big) && \text{by Claim 5} \\
&= \mathsf{red}\left(\tau(\psi_i) \cdot \left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta'(\eta))\right) && \text{by Claim 3} \\
&\leq \tau(\psi_i) \cdot \left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \mathsf{red}\big(\alpha_j(\Delta'(\eta))\big) && \text{by Claim 4 with Equation (14)} \\
&= \tau(\psi_i) \cdot \left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 5}
\end{aligned}$$

□

4.1.3 Joining claims

Claim 7. *For $x \in I_0$, $\mathcal{P} \in \mathsf{Pvr}$, and $f \colon I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_1, \mathsf{Error}\rangle$*

$$\mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\mathsf{extract}(f)(x)(\mathcal{P})\Big)\Big] \leq \tau\big(Q_0.\mathsf{extract}\big) \cdot \mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\delta(f)(x)(\mathcal{P})\Big)\Big]$$
$$\mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\mathsf{solve}(f)(x)(\mathcal{P})\Big)\Big] \leq \tau\big(Q_0.\mathsf{solve}\big) \cdot \mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\delta(f)(x)(\mathcal{P})\Big)\Big]$$

Proof. We argue this inequality for $Q_0.\mathsf{extract}$ and the same reasoning may be applied for $Q_0.\mathsf{solve}$.

The left side represents the expected number of times $Q_0.\mathsf{extract}$ executes $Q_j.\delta$ on arguments f, x, and $\mathcal{P}$. Function $Q_0.\mathsf{extract}$ operates simply by executing $Q_0.\delta$ any number of times. Therefore the expected value of $\mathcal{A}_j$ for $Q_0.\mathsf{extract}$ as written on the left side is determined by the expected value of $\mathcal{A}_j$ for $Q_0.\delta$ and also the expected number of times $Q_0.\mathsf{extract}$ executes $Q_0.\delta$. In particular, we multiply these two metrics.

By the expected extraction and solution times in Definition 2, we know the expected number of times $Q_0.\mathsf{extract}$ executes $Q_0.\delta$ for any arguments is at most $\tau\big(Q_0.\mathsf{extract}\big)$. Therefore multiplying this value by the expected value of $\mathcal{A}_j$ for $Q_0.\delta$ as done on the right side yields an upper bound on the left side. □

Claim 8.

$$\beta\big(\phi_0(\Phi'(\eta))\big) + \beta\big(\psi_0(\Phi'(\eta))\big) \geq \beta\big(\delta_0(\Phi'(\eta))\big) - \chi_0 \tag{19}$$

For $j \in \{0, \ldots, n-1\}$

$$\alpha_j\big(\phi_0(\Phi'(\eta))\big) = \tau(\phi_0) \cdot \alpha_j\big(\delta_0(\Phi'(\eta))\big) \tag{20}$$
$$\alpha_j\big(\psi_0(\Phi'(\eta))\big) = \tau(\psi_0) \cdot \alpha_j\big(\delta_0(\Phi'(\eta))\big) \tag{21}$$

Proof. Equation (19) holds by the definition of $\chi_0 = \chi(Q_0)$ in Definition 2. Expanding the definition of β along with $\phi_0 = Q_0.\mathsf{extract}$, $\psi_0 = Q_0.\mathsf{solve}$, and Definition 2 (with η there replaced with $\Phi'(\eta)$ here) we see

$$\begin{aligned}
&\beta\big(\phi_0(\Phi'(\eta))\big) + \beta\big(\psi_0(\Phi'(\eta))\big) \\
&\quad = \Pr\left[r \in \mathsf{Ok}(W_0) \;\middle|\; r \leftarrow Q_0.\mathsf{extract}(\Phi'(\eta))(x)(\mathcal{P})\right] + \Pr\left[r \in \mathsf{Ok}(S_0) \;\middle|\; r \leftarrow Q_0.\mathsf{solve}(\Phi'(\eta))(x)(\mathcal{P})\right] \\
&\quad \geq \Pr\left[r \in \mathsf{Ok}(W_1) \;\middle|\; r \leftarrow Q_0.\delta(\Phi'(\eta))(x)(\mathcal{P})\right] - \chi(Q_0) \\
&\quad = \beta\big(\delta_0(\Phi'(\eta))\big) - \chi_0
\end{aligned}$$

Equations (20) and (21) are verified via Claim 7. Expanding the definition of α_j along with $\phi_0 = Q_0.\mathsf{extract}$ and $\psi_0 = Q_0.\mathsf{solve}$ we arrive at Claim 7 with $f \coloneqq \Phi'(\eta)$

$$\mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\mathsf{extract}(\Phi'(\eta))(x)(\mathcal{P})\Big)\Big] \leq \tau\big(Q_0.\mathsf{extract}\big) \cdot \mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\delta(\Phi'(\eta))(x)(\mathcal{P})\Big)\Big]$$
$$\mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\mathsf{solve}(\Phi'(\eta))(x)(\mathcal{P})\Big)\Big] \leq \tau\big(Q_0.\mathsf{solve}\big) \cdot \mathbb{E}\Big[\mathcal{A}_j\Big(Q_0.\delta(\Phi'(\eta))(x)(\mathcal{P})\Big)\Big]$$

□

Claim 9.

$$\alpha_0\big(\delta_0(\Phi'(\eta))\big) = \alpha_0\big(\delta_0(\Delta'(\eta))\big) = 1 \tag{22}$$

Proof. Let $f \in \{\Phi'(\eta),\ \Delta'(\eta)\}$. Expanding the definition of α_0 and δ_0 we have for any $x \in I$ and $\mathcal{P} \in \mathsf{Pvr}$

$$\alpha_0(\delta_0(f))(x)(\mathcal{P}) = \mathbb{E}\Big[\mathcal{A}_0\Big(Q_0.\delta(f)(x)(\mathcal{P})\Big)\Big]$$

We assert the following holds and the claim follows.

$$\mathcal{A}_0\Big(Q_0.\delta(f)(x)(\mathcal{P})\Big) = 1$$

By the definition of $\mathcal{A}_0$ and $Q_0.\delta$ this equation asserts that $Q_0.\delta$ is executed once during the execution of $Q_0.\delta(f)$. Examining $Q_0.\delta$ via Algorithm 1 we see it tail calls f, returning whatever f returns, therefore executing exactly once. $\square$

Proving Equation (6). Using Equation (15), Equation (19), and Equation (22) we can now prove Equation (6). Rewriting Equation (19) as

$$\beta\big(\phi_0(\Phi'(\eta))\big) + \beta\big(\psi_0(\Phi'(\eta))\big) + \chi_0 \geq \beta\big(\delta_0(\Phi'(\eta))\big)$$

and substituting the left side for $\beta\big(\delta_0(\Phi'(\eta))\big)$ in Equation (15) yields

$$\beta\big(\phi_0(\Phi'(\eta))\big) + \beta\big(\psi_0(\Phi'(\eta))\big) + \chi_0 + \sum_{i=1}^{n-1}\beta\big(\delta_0(\Psi_i'(\eta))\big) \geq \beta\big(\delta_0(\Delta'(\eta))\big) - \sum_{i=1}^{n-1}\chi_i \cdot \alpha_i\big(\delta_0(\Delta'(\eta))\big)$$

Adding $\chi_0 \cdot 1 = \chi_0 \cdot \alpha_0\big(\delta_0(\Delta'(\eta))\big)$ by Equation (22) to both sides yields the desired inequality of Equation (6).

Proving Equations (7) and (8). We write Equations (20) and (21) holding for $j \in \{0, \ldots, n-1\}$

$$\alpha_j\big(\phi_0(\Phi'(\eta))\big) = \tau(\phi_0) \cdot \alpha_j\big(\delta_0(\Phi'(\eta))\big)$$
$$\alpha_j(\psi_0(\Phi'(\eta))) = \tau(\psi_0) \cdot \alpha_j(\delta_0(\Phi'(\eta)))$$

For $j = 0$ we substitue for $\alpha_0\big(\delta_0(\Phi'(\eta))\big) = 1$ using Equation (22) to get

$$\alpha_0\big(\phi_0(\Phi'(\eta))\big) = \tau(\phi_0) \cdot 1$$
$$\alpha_0\big(\psi_0(\Phi'(\eta))\big) = \tau(\psi_0) \cdot 1$$

For $j \in \{1, \ldots, n-1\}$ we substitute for $\alpha_j\big(\delta_0(\Phi'(\eta))\big)$ using Equation (16) to get

$$\alpha_j\big(\phi_0(\Phi'(\eta))\big) = \tau(\phi_0) \cdot \alpha_j\big(\delta_0(\Phi'(\eta))\big) = \left(\prod_{k=0}^{j}\tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)$$
$$\alpha_j\big(\psi_0(\Phi'(\eta))\big) = \tau(\psi_0) \cdot \alpha_j\big(\delta_0(\Phi'(\eta))\big) = \tau(\psi_0) \cdot \left(\prod_{k=1}^{j}\tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)$$

The latter two equations at $j = 0$ are consistent with the former two equations for $j = 0$, and indeed the latter two equations considered for $j \in \{0, \ldots, n-1\}$ yield Equations (7) and (8) as desired.

Proving Equations (9) and (10). We write Equations (17) and (18) holding for $i \in \{1, \ldots, n-1\}$

$$\forall j \in \{1, \ldots, i-1\},\ \alpha_j\big(\delta_0(\Psi_i'(\eta))\big) = \alpha_j\big(\delta_0(\Delta'(\eta))\big)$$
$$\forall j \in \{i, \ldots, n-1\},\ \alpha_j\big(\delta_0(\Psi_i'(\eta))\big) = \tau(\psi_i)\left(\prod_{k=i+1}^{j}\tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)$$

By Equation (22) the former equation holds too for $j = 0$, yielding Equation (9). The latter equation already yields Equation (10).

4.2 Extraction and solution with regression

4.2.1 pro and reg and properties

We introduce functions pro and reg, their combination proreg* to be used later, and we show relevant properties.

Definition 9 (pro function)**.**

$$\begin{gathered}\mathsf{pro}\langle I\rangle\colon (I \to \mathsf{Pvr} \to \mathbb{R}) \to I \to \mathsf{Pvr} \to \mathbb{R}\\ \mathsf{pro}(x)(g)(\mathcal{P}) \coloneqq \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{matrix} r = \mathsf{Progress} \\ p = \mathcal{P}' \end{matrix}\ \middle|\ \begin{matrix} r \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{matrix}\right] \cdot g(x)(\mathcal{P}')\end{gathered}$$

Definition 10 (reg function)**.**

$$\begin{gathered}\mathsf{reg}\langle I, T\rangle\colon (I \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to (I \to \mathsf{Pvr} \to \mathbb{R}) \to I \to \mathsf{Pvr} \to \mathbb{R}\\ \mathsf{reg}(f)(g)(x)(\mathcal{P}) \coloneqq \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{matrix} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}' \end{matrix}\ \middle|\ \begin{matrix} r \leftarrow f(x)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{matrix}\right] \cdot g(x)(\mathcal{P}')\end{gathered}$$

Claim 10 (pro and reg linearity)**.** *For $f\langle I, T\rangle\colon I \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$ and $g, g'\langle I\rangle\colon I \to \mathsf{Pvr} \to \mathbb{R}$ we see* pro *and* reg *to be linear in the sense that*

$$\begin{aligned} \lambda_0(x)(\mathcal{P}) &\coloneqq g(x)(\mathcal{P}) + g'(x)(\mathcal{P}) & \lambda_1(x)(\mathcal{P}) &\coloneqq \alpha \cdot g(x)(\mathcal{P})\\ \mathsf{pro}(\lambda_0) &= \mathsf{pro}(g) + \mathsf{pro}(g') & \mathsf{pro}(\lambda_1) &= \alpha \cdot \mathsf{pro}(g)\\ \mathsf{reg}(f)(\lambda_0) &= \mathsf{reg}(f)(g) + \mathsf{reg}(f)(g') & \mathsf{reg}(f)(\lambda_1) &= \alpha \cdot \mathsf{reg}(f)(g) \end{aligned}$$

Proof. Linearity is evident by examination of pro and reg and properties of real arithmetic, including distributivity of multiplication over addition. □

Claim 11 (pro and reg inequality)**.** *For $f\langle I, T\rangle\colon I \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$ and $g, g'\langle I\rangle\colon I \to \mathsf{Pvr} \to \mathbb{R}$ we see* pro *and* reg *preserve inequalities, that is*

$$g \geq g' \implies \mathsf{pro}(g) \geq \mathsf{pro}(g'),\ \mathsf{reg}(f)(g) \geq \mathsf{reg}(f)(g')$$

where inequality between two real-valued functions means inequality between their evaluations for all possible arguments.

Proof. Inequality is evident by examination of pro and reg and properties of real arithmetic. □

reg invariance The following lemma presents two equations. First we will prove Equation (24). Then we will utilize Equation (24) to prove Equation (23). Equation (24) will be used subsequently in Section 4.2 while Equation (23) will not. We mention now that type variables are written on invocation whenever we feel it adds clarity.

Lemma 2 (reg invariance)**.** *The following two equations hold for $n \geq 1$ and $i \in \{1, \ldots, n-1\}$.*

$$\mathsf{reg}\langle I_0, W_1\rangle(\Phi(\eta)) = \mathsf{reg}\langle I_0, S_i\rangle(\Psi_i(\eta)) = \mathsf{reg}\langle I_0, W_n\rangle(\Delta(\eta)) \tag{23}$$

$$\begin{aligned} &\mathsf{reg}\langle I_0, W_0\rangle\big(\phi_0(\Phi'(\eta))\big) = \mathsf{reg}\langle I_0, S_0\rangle\big(\psi_0(\Phi'(\eta))\big)\\ &\quad = \mathsf{reg}\langle I_0, S_i\rangle\big(\delta_0(\Psi_i'(\eta))\big) = \mathsf{reg}\langle I_0, W_n\rangle\big(\delta_0(\Delta'(\eta))\big) \end{aligned} \tag{24}$$

Proving Equation (24) To prove Equation (24) for $n \geq 2$ we proceed assuming Equation (23) for $n-1 \geq 1$. We state this induction assumption in the following claim. For the case $n = 1$ we argue the inductive assumption can be extended to $n - 1 = 0$.

Claim 12. *The following two equations hold for $n \geq 1$ and $i \in \{1, \dots, n-1\}$.*

$$\mathsf{reg}\langle I_1, W_1\rangle(\Phi'(\eta)) = \mathsf{reg}\langle I_1, S_i\rangle(\Psi'_i(\eta)) = \mathsf{reg}\langle I_1, W_n\rangle(\Delta'(\eta)) \tag{25}$$

Proof. For $n \geq 2$ these equalities hold by Equation (23) for $n-1 \geq 1$. For $n = 1$ we cannot invoke Equation (23) for $n - 1 = 0$, but we observe these equalities trivially hold by the fact $\Phi'(\eta) = \Psi'_i(\eta) = \Delta'(\eta) = \eta$. ☐

Claim 13 (Extending with regression)**.** *For f and f' with signatures*

$$f\langle T\rangle \colon I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$$
$$f'\langle T'\rangle \colon I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle T', \mathsf{Error}\rangle$$

the following implication holds.

$$\mathsf{reg}\langle I_1, T\rangle(f) = \mathsf{reg}\langle I_1, T'\rangle(f') \implies \mathsf{reg}\langle I_0, T\rangle(\delta_0(f)) = \mathsf{reg}\langle I_0, T'\rangle\left(\delta_0(f')\right)$$

Proof. Plugging $\delta_0(f)$ into reg along with some $g\colon I_0 \to \mathsf{Pvr} \to \mathbb{R}$, $x_0 \in I_0$, and $\mathcal{P} \in \mathsf{Pvr}$, we obtain Equation (26). Equation (27) holds by the definition of δ_0, see Algorithm 1. Equation (28) expands the meaning of $r \leftarrow f(\mathsf{instance}(m)(x_0))(\mathcal{P})$.

$$\begin{aligned}
&\mathsf{reg}\langle I_0, T\rangle\big(\delta_0(f)\big)(g)(x_0)(\mathcal{P}) \\
&= \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}' \end{array} \middle| \begin{array}{c} r \leftarrow \delta_0(f)(x_0)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \cdot g(x_0)(\mathcal{P}') && (26) \\
&= \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}' \end{array} \middle| \begin{array}{l} \textbf{if } \mathsf{Pmove}(Q_0) \\ \quad m \leftarrow \mathcal{P}.\mathsf{message}() \\ \textbf{if } \mathsf{Vmove}(Q_0) \\ \quad m \leftarrow Q_0.\mathsf{message}() \\ \quad \mathcal{P}.\mathsf{update}(m) \\ r \leftarrow f\big(\mathsf{instance}(m)(x_0)\big)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \cdot g(x_0)(\mathcal{P}') && (27) \\
&= \sum_{\mathcal{P}' \in \mathsf{Pvr}} \sum_{x_1 \in I_1} \Pr\left[\begin{array}{c} \mathsf{instance}(m)(x_0) = x_1 \\ p = \mathcal{P}' \end{array} \middle| \begin{array}{l} \textbf{if } \mathsf{Pmove}(Q_0) \\ \quad m \leftarrow \mathcal{P}.\mathsf{message}() \\ \textbf{if } \mathsf{Vmove}(Q_0) \\ \quad m \leftarrow \mathcal{V}.\mathsf{message}() \\ \quad \mathcal{P}.\mathsf{update}(m) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \\
&\quad \times \sum_{\mathcal{P}'' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}'' \end{array} \middle| \begin{array}{c} r \leftarrow f(x_1)(\mathcal{P}') \\ p \leftarrow \mathcal{P}'.\mathsf{clone}() \end{array}\right] \cdot g(x_0)(\mathcal{P}'') && (28)
\end{aligned}$$

Define $\lambda\colon I_0 \to I_1 \to \mathsf{Pvr} \to \mathbb{R}$ by $\lambda(x_0)(x_1)(\mathcal{P}) \coloneqq g(x_0)(\mathcal{P})$. Consider the second factor in Equation (28), that is the bottom sum multiplying the top double sum. With $g(x_0)(\mathcal{P}'') = \lambda(x_0)(x_1)(\mathcal{P}'')$ we may make the replacement, and the bottom sum becomes

$$\sum_{\mathcal{P}'' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}'' \end{array} \middle| \begin{array}{c} r \leftarrow f(x_1)(\mathcal{P}') \\ p \leftarrow \mathcal{P}'.\mathsf{clone}() \end{array}\right] \cdot \lambda(x_0)(x_1)(\mathcal{P}'') = \mathsf{reg}\langle I_1, T\rangle(f)\big(\lambda(x_0)\big)(x_1)(\mathcal{P}'')$$

The top double sum is independent of f. With $\mathsf{reg}\langle I_1, T\rangle(f) = \mathsf{reg}\langle I_1, T\rangle(f')$ by assumption, and feeding both the same arguments $\lambda(x_0)$, x_1, and $\mathcal{P}''$, we conclude $\mathsf{reg}\langle I_0, T\rangle(\delta_0(f)) = \mathsf{reg}\langle I_0, T\rangle(\delta_0(f'))$ on any arguments g, x_0, and $\mathcal{P}$. □

Claim 14. *For $n \geq 1$ and $i \in \{1, \dots, n-1\}$*

$$\mathsf{reg}\langle I_0, W_1\rangle\big(\delta_0(\Phi'(\eta))\big) = \mathsf{reg}\langle I_0, S_i\rangle\big(\delta_0(\Psi'_i(\eta))\big) = \mathsf{reg}\langle I_0, W_n\rangle\big(\delta_0(\Delta'(\eta))\big) \tag{29}$$

Proof. For $n \geq 1$ and $i \in \{1, \dots, n-1\}$ by induction assumption Claim 12 we have

$$\mathsf{reg}\langle I_1, W_1\rangle(\Phi'(\eta)) = \mathsf{reg}\langle I_1, S_i\rangle\big(\Psi'_i(\eta)\big) = \mathsf{reg}\langle I_1, W_n\rangle(\Delta'(\eta))$$

Applying Claim 13 we conclude Equation (29). □

The two rightmost functions in Equation (29) yield two of the desired functions in Equation (24). It remains to obtain the two other desired functions in Equation (24) from the leftmost function in Equation (29).

Claim 15. *For $f\colon I_1 \to \mathsf{Pvr} \to \mathsf{Result}\langle W_1, \mathsf{Error}\rangle$*

$$\mathsf{reg}\langle I_0, W_0\rangle(\phi_0(f)) = \mathsf{reg}\langle I_0, S_0\rangle(\psi_0(f)) = \mathsf{reg}\langle I_0, W_1\rangle(\delta_0(f))$$

Proof. Expanding the definition of reg for the rightmost funcntion with $\delta_0(f)$ we see the value of $\mathsf{reg}\langle I_0, W_1\rangle(\delta_0(f))$ on inputs $g\colon I_0 \to \mathsf{Pvr} \to \mathbb{R}$, $x \in I_0$, and $\mathcal{P} \in \mathsf{Pvr}$ is determined from the probability $\delta_0(f)(x)(\mathcal{P})$ returns value $\mathsf{Err}(\mathsf{Regress})$ and returns the prover in a particular state $\mathcal{P}'$, multiplied by $g(x)(\mathcal{P}')$.

$$\mathsf{reg}\langle I_0, W_1\rangle(\delta_0(f))(g)(x)(\mathcal{P}) := \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{matrix} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}' \end{matrix} \;\middle|\; \begin{matrix} r \leftarrow \delta_0(f)(x)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{matrix}\right] \cdot g(x)(\mathcal{P}')$$

By assumption, $\phi_0 = Q_0.\mathsf{extract}$ and $\psi = Q_0.\mathsf{solve}$ invoke $\delta_0 = Q_0.\delta$ and return the error $\mathsf{Err}(\mathsf{Regress})$ if and only if $Q_0.\delta$ does. Therefore the probability $Q_0.\mathsf{extract}(f)(x)(\mathcal{P})$ and $Q_0.\mathsf{solve}(f)(x)(\mathcal{P})$ return $\mathsf{Err}(\mathsf{Regress})$ is the same as the probability $Q_0.\delta(f)(x)(\mathcal{P})$ does. In the case that $Q_0.\mathsf{extract}$ and $Q_0.\mathsf{solve}$ return error $\mathsf{Err}(\mathsf{Regress})$, they have executed precisely $Q_0.\delta$ and therefore return the prover in the same state as $Q_0.\delta$ does. The reg function exhibits the same behavior whether its first argument is $Q_0.\mathsf{extract}(f)$, $Q_0.\mathsf{solve}(f)$, or $Q_0.\delta(f)$. □

The two leftmost functions in the following claim yield the remaining two desired functions for Equation (24).

Claim 16.

$$\mathsf{reg}\langle I_0, W_0\rangle\big(\phi_0(\Phi'(\eta))\big) = \mathsf{reg}\langle I_0, S_0\rangle\big(\psi_0(\Phi'(\eta))\big) = \mathsf{reg}\langle I_0, W_1\rangle\big(\delta_0(\Phi'(\eta))\big)$$

Proof. This holds by invoking Claim 15 with $f := \Phi'(\eta)$. □

Proving Equation (23). The following is the first of two recursive expressions we convert to the additive form of an infinite series. The expression in Claim 17 relates to the probability the prover requests *regression*, while the recursive expression we will encounter in Claim 21 relates to the probability the prover requests *progression*. They also differ, however, by Claim 17 considering not only the probability of regression, but the probability that upon regression the prover is in a certain state, as will be needed in Claim 18. In Claim 21, by constrast, we only consider the probability of progression and need not consider the returning state of the prover.

Claim 17. *Consider the function $\lambda\colon \mathsf{Pvr} \to I_0 \to \mathsf{Pvr} \to \mathbb{R}$*

$$\lambda(\mathcal{P}')(\cdot)(\mathcal{P}) := \Pr\left[\begin{matrix} r = \mathsf{Regress} \\ p = \mathcal{P}' \end{matrix} \;\middle|\; \begin{matrix} r \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{matrix}\right]$$

For some $f\langle T\rangle \colon I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$, $x \in I_0$, and $\mathcal{P}, \mathcal{P}''' \in \mathsf{Pvr}$

$$\Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}''' \end{array} \middle| \begin{array}{c} r \leftarrow \Gamma(f)(x)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \tag{30}$$

$$= \Pr\left[\begin{array}{c} w = \mathsf{Regress} \\ p = \mathcal{P}''' \end{array} \middle| \begin{array}{c} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] + \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} w = \mathsf{Progress} \\ p = \mathcal{P}' \end{array} \middle| \begin{array}{c} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \Bigg(\tag{31}$$

$$\sum_{\mathcal{P}'' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}'' \end{array} \middle| \begin{array}{c} r \leftarrow f(x)(\mathcal{P}') \\ p \leftarrow \mathcal{P}'.\mathsf{clone}() \end{array}\right] \cdot \Pr\left[\begin{array}{c} r' = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}''' \end{array} \middle| \begin{array}{c} r' \leftarrow \Gamma(f)(x)(\mathcal{P}'') \\ p \leftarrow \mathcal{P}''.\mathsf{clone}() \end{array}\right] \Bigg) \tag{32}$$

$$= \sum_{i=0}^{\infty} \Big(\bigcirc_{j=0}^{i-1} \big(\mathsf{pro} \circ \mathsf{reg}\langle I_0, T\rangle(f)\big) \circ \lambda(\mathcal{P}''') \Big)(x)(\mathcal{P}) \tag{33}$$

Note that the series converges since it represents a probability.

Proof. The recursive form expresses the probability that execution of $\Gamma(f)(x)$ with prover $\mathcal{P}$ results in the error $\mathsf{Err}(\mathsf{Regress})$, and also whether the prover returns having shifted to a particular state $\mathcal{P}'''$. This event occurs as follows.

1. Prover $\mathcal{P}$ is queried on the method way and returns a variant of $\mathsf{Way} = \mathsf{Regress} \mid \mathsf{Progress}$. Either Step 2 or Steps 3, 4, and 5 follow.
2. If $\mathcal{P}$ returns Regress and shifts to state $\mathcal{P}'''$ then Γ, see Algorithm 2, returns the value $\mathsf{Err}(\mathsf{Regress})$ In this case the event has occured.
3. If $\mathcal{P}$ returns Progress and shifts to state $\mathcal{P}'$, then Γ executes $f(x)(\mathcal{P}')$ and examines the result r.
4. If r holds a result variant $\mathsf{Ok}\langle T\rangle$ or $\mathsf{Err}(\mathsf{Fail})$ then Γ returns those values and the event never occurred. If r holds the result variant $\mathsf{Err}(\mathsf{Regress})$ and the prover has shifted to state $\mathcal{P}''$ then Γ recurses by invoking $\Gamma(f)(x)(\mathcal{P}'')$.
5. On re-invocation of Γ we reconsider these five steps, but now starting with prover $\mathcal{P}''$. The event occurs at this point if and only if $\Gamma(f)(x)(\mathcal{P}'')$ returns $\mathsf{Err}(\mathsf{Regress})$ with the prover in state $\mathcal{P}'''$.

Expanding the recursive form and rewriting terms using pro, reg, and λ one obtains the series. For illustration we write out the first two expanded terms and the first three terms using pro, reg, and λ.

$$\Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}''' \end{array} \middle| \begin{array}{c} r \leftarrow \Gamma(f)(x)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] = \Pr\left[\begin{array}{c} w = \mathsf{Regress} \\ p = \mathcal{P}''' \end{array} \middle| \begin{array}{c} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right]$$
$$+ \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} w = \mathsf{Progress} \\ p = \mathcal{P}' \end{array} \middle| \begin{array}{c} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \cdot \sum_{\mathcal{P}'' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}'' \end{array} \middle| \begin{array}{c} r \leftarrow f(x)(\mathcal{P}') \\ p \leftarrow \mathcal{P}'.\mathsf{clone}() \end{array}\right]$$
$$\times \Pr\left[\begin{array}{c} w = \mathsf{Regress} \\ p = \mathcal{P}''' \end{array} \middle| \begin{array}{c} w \leftarrow \mathcal{P}''.\mathsf{way}() \\ p \leftarrow \mathcal{P}''.\mathsf{clone}() \end{array}\right] + \ \ldots$$
$$= \lambda(\mathcal{P}''')(x)(\mathcal{P}) + \Big(\mathsf{pro} \circ \mathsf{reg}(f) \circ \lambda(\mathcal{P}''')\Big)(x)(\mathcal{P}) + \Big(\mathsf{pro} \circ \mathsf{reg}(f) \circ \mathsf{pro} \circ \mathsf{reg}(f) \circ \lambda(\mathcal{P}''')\Big)(x)(\mathcal{P}) + \ \ldots$$

□

Claim 18. *To prove Equation* (23) *we use the definitions of* Φ, Ψ_i *for* $i \in \{1, \ldots, n{-}1\}$, *and* Δ *to rewrite Equation* (23) *as follows.*

$$\mathsf{reg}\langle W_0\rangle\Big(\Gamma \circ \phi_0 \circ \Phi' \circ \eta\Big) = \mathsf{reg}\langle S_0\rangle\Big(\Gamma \circ \psi_0 \circ \Phi' \circ \eta\Big)$$
$$= \mathsf{reg}\langle S_i\rangle\Big(\Gamma \circ \delta_0 \circ \Psi_i' \circ \eta\Big) = \mathsf{reg}\langle W_n\rangle\Big(\Gamma \circ \delta_0 \circ \Delta' \circ \eta\Big)$$

Proof. For $f \in \{\phi_0(\Phi'(\eta)),\ \psi_0(\Phi'(\eta)),\ \delta_0(\Psi'_i(\eta)),\ \delta_0(\Delta'(\eta))\}$, $g\colon I_0 \to \mathsf{Pvr} \to \mathbb{R}$, $x \in I_0$, and $\mathcal{P} \in \mathsf{Pvr}$ we write $\mathsf{reg}(\Gamma(f))$ invoked on g and $\mathcal{P}$ in Equation (34) and in Equation (35) we invoke Claim 17 using λ unchanged from Claim 17.

$$\mathsf{reg}(\Gamma(f))(g)(x)(\mathcal{P}) = \sum_{\mathcal{P}' \in \mathsf{Pvr}} \Pr\left[\begin{array}{c} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}' \end{array} \middle| \begin{array}{c} r \leftarrow \Gamma(f)(x)(\mathcal{P}) \\ p \leftarrow \mathcal{P}.\mathsf{clone}() \end{array}\right] \cdot g(x)(\mathcal{P}') \tag{34}$$

$$= \sum_{\mathcal{P}' \in \mathsf{Pvr}} \sum_{i=0}^{\infty} \Big(\bigcirc_{j=0}^{i-1} \big(\mathsf{pro} \circ \mathsf{reg}(f)\big) \circ \lambda(\mathcal{P}') \Big)(x)(\mathcal{P}) \cdot g(x)(\mathcal{P}') \tag{35}$$

By Equation (24), previously proven, we conclude $\mathsf{reg}(f)$ is the same for all four functions f. Therefore $\mathsf{reg}(\Gamma(f))$ is the same for all four functions f, and the claim follows. ☐

4.2.2 proreg and properties

We define the composition of pro and reg for depth $i \geq 0$, calling it $\mathsf{proreg}^{(i)}$. We then define $\mathsf{proreg}^\star$ involving the compositions of pro and reg of every depth, capturing the probability any number of progressions followed by a regression occur. But rather than leaving the first argument of reg free, we fix it to $\delta_0(\Delta'(\eta))$ which will suffice for our purposes having previously proven the invariance property of reg in Section 4.2.1, Equation (24). We also fix

Definition 11 ($\mathsf{proreg}^{(i)}$)**.**

$$\begin{aligned} &\forall i \geq 0,\ \mathsf{proreg}^{(i)}\langle T\rangle\colon (I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to (I_0 \to \mathsf{Pvr} \to \mathbb{R}) \to I_0 \to \mathsf{Pvr} \to \mathbb{R} \\ &\qquad \mathsf{proreg}^{(0)}(\cdot)(g)(x)(\mathcal{P}) \coloneqq g(x)(\mathcal{P}) \\ &\forall i \geq 1,\ \mathsf{proreg}^{(i)}(f)(g)(x)(\mathcal{P}) \coloneqq \Big(\bigcirc_{j=0}^{i-1} \big(\mathsf{pro} \circ \mathsf{reg}\langle I_0, T\rangle(f)\big) \Big)(g)(x)(\mathcal{P}) \end{aligned}$$

Definition 12 ($\mathsf{proreg}^\star$)**.**

$$\begin{aligned} &\mathsf{proreg}^\star\colon (I_0 \to \mathsf{Pvr} \to \mathbb{R}) \to I_0 \to \mathsf{Pvr} \to \mathbb{R} \\ &\mathsf{proreg}^\star(g)(x)(\mathcal{P}) \coloneqq \sum_{i=0}^{\infty} \mathsf{proreg}^{(i)} \Big(\delta_0\big(\Delta'(\eta)\big) \Big)(g)(x)(\mathcal{P}) \end{aligned}$$

Claim 19 ($\mathsf{proreg}^\star$ linearity)**.** *For $g, g'\colon I_0 \to \mathsf{Pvr} \to \mathbb{R}$ and $\alpha \in \mathbb{R}$ we see* proreg *to be linear in the sense that*

$$\begin{array}{cc} \lambda_0(x)(\mathcal{P}) \coloneqq g(x)(\mathcal{P}) + g'(x)(\mathcal{P}) & \lambda_1(x)(\mathcal{P}) \coloneqq \alpha \cdot g(x)(\mathcal{P}) \\ \mathsf{proreg}^\star(\lambda_0) = \mathsf{proreg}^\star(g) + \mathsf{proreg}^\star(g') & \mathsf{proreg}^\star(\lambda_1) = \alpha \cdot \mathsf{proreg}^\star(g) \end{array}$$

Proof. Linearity is evident by the linearity of pro and reg by Claim 10 and the definition of $\mathsf{proreg}^\star$ in terms of pro and reg. ☐

Claim 20 ($\mathsf{proreg}^\star$ inequality)**.** *For $g, g'\colon I_0 \to \mathsf{Pvr} \to \mathbb{R}$ we see* proreg *to preserve inequalities, that is*

$$g \geq g' \implies \mathsf{proreg}^\star(g) \geq \mathsf{proreg}^\star(g')$$

where inequality between two real-valued functions means inequality between their evaluations for all possible arguments.

Proof. Inequality preservation is evident by the inequality preservation of pro and reg by Claim 11 and the definition of $\mathsf{proreg}^\star$ in terms of pro and reg. ☐

4.2.3 Extending with regression

Claim 21. *Consider the function signature labelled with f below and note that $\phi_0(\Phi'(\eta))$, $\psi_0(\Phi'(\eta))$, $\delta_0(\Psi'_i(\eta))$, and $\delta_0(\Delta'(\eta))$ are all instances of such a function.*

$$f\langle T\rangle\colon I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle$$

For $f \in \{\phi_0(\Phi'(\eta)),\ \psi_0(\Phi'(\eta)),\ \delta_0(\Psi'_i(\eta)),\ \delta_0(\Delta'(\eta))\}$, consider the function signature labelled with λ below and note that β and α_j for $j \in \{0, \dots, n-1\}$ are all instances of such a function. We redefine β and α_j for clarity, and note $\mathcal{A}_j$ is well defined on f.

$$\lambda\langle T\rangle\colon (I_0 \to \mathsf{Pvr} \to \mathsf{Result}\langle T, \mathsf{Error}\rangle) \to I_0 \to \mathsf{Pvr} \to \mathbb{R}$$
$$\beta(f)(x)(\mathcal{P}) \coloneqq \Pr\Big[r \in \mathsf{Ok}\langle T\rangle \,\Big|\, r \leftarrow f(x)(\mathcal{P})\Big]$$
$$\forall j \in \{0, \dots, n-1\},\ \alpha_j(f)(x)(\mathcal{P}) \coloneqq \mathbb{E}\Big[\mathcal{A}_j\big(f(x)(\mathcal{P})\big)\Big]$$

For $\lambda \in \{\beta\} \cup \{\alpha_j\}_{j\in\{0,\dots,n-1\}}$ as well as $x \in I_0$ and $\mathcal{P} \in \mathsf{Pvr}$ we claim the following.

$$\begin{aligned}
\lambda(\Gamma(f))(x)(\mathcal{P}) &= \sum_{\mathcal{P}'\in\mathsf{Pvr}} \Pr\left[\begin{matrix} w = \mathsf{Progress} \\ p = \mathcal{P}'\end{matrix} \,\middle|\, \begin{matrix} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}()\end{matrix}\right] \Bigg(\lambda(f)(x)(\mathcal{P}') \\
&\quad + \sum_{\mathcal{P}''} \Pr\left[\begin{matrix} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}''\end{matrix} \,\middle|\, \begin{matrix} r \leftarrow f(x)(\mathcal{P}') \\ p \leftarrow \mathcal{P}'.\mathsf{clone}()\end{matrix}\right] \cdot \lambda(\Gamma(f))(x)(\mathcal{P}'')\Bigg) \\
&= \sum_{i=0}^{\infty} \Big(\bigcirc_{j=0}^{i-1}\big(\mathsf{pro} \circ \mathsf{reg}\langle I_0, T\rangle(f)\big) \circ \mathsf{pro}(\lambda(f))\Big)(x)(\mathcal{P})
\end{aligned}$$

Proof. Before arguing that $\lambda(\Gamma(f))(x)(\mathcal{P})$ on the left side equates to the recursive formula, we first argue the recursive formula equates to the series. This recursive expression can be translated to the additive form of a series in the same way as Claim 17. We write out the first two terms.

$$\begin{aligned}
&\sum_{\mathcal{P}'\in\mathsf{Pvr}} \Pr\left[\begin{matrix} w = \mathsf{Progress} \\ p = \mathcal{P}'\end{matrix} \,\middle|\, \begin{matrix} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}()\end{matrix}\right] \cdot \lambda(f)(x)(\mathcal{P}') \\
&\quad + \sum_{\mathcal{P}'\in\mathsf{Pvr}} \Pr\left[\begin{matrix} w = \mathsf{Progress} \\ p = \mathcal{P}'\end{matrix} \,\middle|\, \begin{matrix} w \leftarrow \mathcal{P}.\mathsf{way}() \\ p \leftarrow \mathcal{P}.\mathsf{clone}()\end{matrix}\right] \cdot \sum_{\mathcal{P}''\in\mathsf{Pvr}} \Pr\left[\begin{matrix} r = \mathsf{Err}(\mathsf{Regress}) \\ p = \mathcal{P}''\end{matrix} \,\middle|\, \begin{matrix} r \leftarrow f(x)(\mathcal{P}') \\ p \leftarrow \mathcal{P}'.\mathsf{clone}()\end{matrix}\right] \\
&\quad \times \sum_{\mathcal{P}'''\in\mathsf{Pvr}} \Pr\left[\begin{matrix} w = \mathsf{Progress} \\ p = \mathcal{P}'''\end{matrix} \,\middle|\, \begin{matrix} w \leftarrow \mathcal{P}''.\mathsf{way}() \\ p \leftarrow \mathcal{P}''.\mathsf{clone}()\end{matrix}\right] \cdot \lambda(f)(x)(\mathcal{P}''') + \dots \\
&= \big(\mathsf{pro} \circ \lambda(f)\big)(x)(\mathcal{P}) + \Big(\mathsf{pro} \circ \mathsf{reg}(f) \circ \mathsf{pro} \circ \lambda(f)\Big)(x)(\mathcal{P}) + \dots \\
&= \sum_{i=0}^{\infty} \Big(\bigcirc_{j=0}^{i-1}\big(\mathsf{pro} \circ \mathsf{reg}(f)\big) \circ \mathsf{pro}(\lambda(f))\Big)(x)(\mathcal{P})
\end{aligned}$$

The recursive formula describes a real value in relation to execution of $\Gamma(f)(x)(\mathcal{P})$. Consider the following event in the execution of $\Gamma(f)(x)(\mathcal{P})$, parameterized by integer $i \geq 0$: The two steps below occur i times, followed by one occurance of Step 1.

1. The prover is queried on way and returns Progress.
2. Execution of $f(x)(\mathcal{P})$ is performed and results in value Err(Regress).

Term $i \geq 0$ in the series can be examined to represent the probability of this event with respect i, multiplied by real value $\lambda(f)(x)(\hat{\mathcal{P}})$ where $\hat{\mathcal{P}}$ is the state of the prover after the event, before execution $i+1$ of $f(x)(\hat{P})$.

When $\lambda := \beta$, term i represents the probability that i progressions and regressions occur, followed by another progression and then execution of $f(x)(\hat{\mathcal{P}})$ returning an $\mathsf{Ok}\langle T\rangle$ value. Therefore the whole series represents the probability of execution of $\Gamma(f)(x)(\mathcal{P})$ returning an $\mathsf{Ok}\langle T\rangle$ value, which indeed equates to the left side $\beta(\Gamma(f))(x)(\mathcal{P})$.

When $\lambda := \alpha_j$, term i represents the multiplication of two real values: (a) the probability that i progressions and regressions occur, followed by another progression; (b) the expected value of $\mathcal{A}_j\big(f(x)(\hat{\mathcal{P}})\big)$. Therefore the whole series represents the expected value of $\mathcal{A}_j\big(\Gamma(f)(x)(\mathcal{P})\big)$, which indeed equates to the left side $\alpha_j(\Gamma(f))(x)(\mathcal{P})$. □

Claim 22.

$$\begin{aligned}
\beta(\Phi(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\phi_0(\Phi'(\eta))\big) \\
\beta(\Psi_0(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\psi_0(\Phi'(\eta))\big) \\
\forall i \in \{1,\ldots,n-1\},\ \beta(\Psi_i(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\delta_0(\Psi_i'(\eta))\big) \\
\beta(\Delta(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\delta_0(\Delta'(\eta))\big)
\end{aligned}$$

For $j \in \{0,\ldots,n-1\}$

$$\begin{aligned}
\alpha_j(\Phi(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\phi_0(\Phi'(\eta))\big) \\
\alpha_j(\Psi_0(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\psi_0(\Phi'(\eta))\big) \\
\forall i \in \{1,\ldots,n-1\},\ \alpha_j(\Psi_i(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Psi_i'(\eta))\big) \\
\alpha_j(\Delta(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Delta'(\eta))\big)
\end{aligned}$$

Proof. This claim re-interprets Claim 21 in a convenient form. With f and λ as in Claim 21, the left sides of the equations above correspond to $\lambda(\Gamma(f))$. Therefore we reduce to equating the right sides of the equations above to

$$\sum_{i=0}^{\infty} \bigcirc_{j=0}^{i-1}\big(\mathsf{pro} \circ \mathsf{reg}\langle I_0, T\rangle(f)\big) \circ \mathsf{pro}(\lambda(f))$$

Note we are now equating functions without the application of x and $\mathcal{P}$ as done in Claim 21, e.g. $\lambda(\Gamma(f))$ instead of $\lambda(\Gamma(f))(x)(\mathcal{P})$.

We invoke the invariance property of reg proven Section 4.2.1, specifically Equation (24), restated as

$$\begin{aligned}
&\mathsf{reg}\langle I_0, W_0\rangle\big(\phi_0\big(\Phi'(\eta)\big)\big) = \mathsf{reg}\langle I_0, S_0\rangle\big(\psi_0\big(\Phi'(\eta)\big)\big) \\
&\quad = \mathsf{reg}\langle I_0, S_i\rangle\big(\delta_0\big(\Psi_i'(\eta)\big)\big) = \mathsf{reg}\langle I_0, W_n\rangle\big(\delta_0\big(\Delta'(\eta)\big)\big)
\end{aligned}$$

With $f \in \{\phi_0(\Phi'(\eta)),\ \psi_0(\Phi'(\eta)),\ \delta_0(\Psi_i'(\eta)),\ \delta_0(\Delta'(\eta))\}$, this means all four functions of $\mathsf{reg}(f)$ are equal. We may thus replace $\mathsf{reg}(f)$ in the series with $\mathsf{reg}(\delta_0(\Delta'(\eta)))$ and arrive at the claimed right sides of the equations as

$$\sum_{i=0}^{\infty} \bigcirc_{j=0}^{i-1}\Big(\mathsf{pro} \circ \mathsf{reg}\big(\delta_0(\Delta'(\eta))\big)\Big) \circ \mathsf{pro}(\lambda(f)) = \mathsf{proreg}^\star\big(\mathsf{pro}(\lambda(f))\big)$$

□

4.2.4 Joining claims

We invoke Lemma 1, along with Claim 22, Claim 19, and Claim 20 to prove Theorem 2.

Proving Equation (2).

$$\begin{aligned}
&\beta(\Phi(\eta)) + \sum_{i=0}^{n-1} \beta(\Psi_i(\eta)) \\
&\quad = \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\phi_0(\Phi'(\eta))\big) + \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\psi_0(\Phi'(\eta))\big) \\
&\qquad + \sum_{i=1}^{n-1} \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\delta_0(\Psi_i'(\eta))\big) && \text{by Claim 22} \\
&\quad = \mathsf{proreg}^\star \circ \mathsf{pro}\left(\beta\big(\phi_0(\Phi'(\eta))\big) + \beta\big(\psi_0(\Phi'(\eta))\big) + \sum_{i=1}^{n-1} \beta\big(\delta_0(\Psi_i'(\eta))\big)\right) && \text{by Claim 19} \\
&\quad \geq \mathsf{proreg}^\star \circ \mathsf{pro}\left(\beta\big(\delta_0(\Delta'(\eta))\big) - \sum_{i=0}^{n-1} \chi_i \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)\right) && \text{by Claim 20 with Equation (6)} \\
&\quad = \mathsf{proreg}^\star \circ \mathsf{pro} \circ \beta\big(\delta_0(\Delta'(\eta))\big) - \sum_{i=0}^{n-1} \chi_i \cdot \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 19} \\
&\quad = \beta(\Delta(\eta)) - \sum_{i=0}^{n-1} \chi_i \cdot \alpha_j(\Delta(\eta)) && \text{by Claim 22}
\end{aligned}$$

Proving Equation (3). For $j \in \{0, \ldots, n-1\}$

$$\begin{aligned}
\alpha_j(\Phi) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\phi_0(\Phi'(\eta))\big) && \text{by Claim 22} \\
&= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \left(\left(\prod_{k=0}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)\right) && \text{by Equation (7)} \\
&= \left(\prod_{k=0}^{j} \tau(\phi_k)\right) \cdot \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 19} \\
&= \left(\prod_{k=0}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta(\eta)) && \text{by Claim 22}
\end{aligned}$$

Proving Equation (4). For $j \in \{0, \ldots, n-1\}$

$$\begin{aligned}
\alpha_j(\Psi_0(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\psi_0(\Phi'(\eta))\big) && \text{by Claim 22} \\
&= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \left(\tau(\psi_0)\left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)\right) && \text{by Equation (8)} \\
&= \tau(\psi_0)\left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 19} \\
&= \tau(\psi_0)\left(\prod_{k=1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta(\eta)) && \text{by Claim 22}
\end{aligned}$$

Proving Equation (5). For $i \in \{1, \ldots, n-1\}$

$$\begin{aligned}
\forall j \in \{0, \ldots, i-1\},\ \alpha_j(\Psi_i(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Psi_i'(\eta))\big) && \text{by Claim 22} \\
&= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Equation (9)} \\
&= \alpha_j(\Delta(\eta)) && \text{by Claim 22} \\
\forall j \in \{i, \ldots, n-1\},\ \alpha_j(\Psi_i(\eta)) &= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Psi_i'(\eta))\big) && \text{by Claim 22} \\
&= \mathsf{proreg}^\star \circ \mathsf{pro} \circ \left(\tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j\big(\delta_0(\Delta'(\eta))\big)\right) && \text{by Equation (10)} \\
&= \tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \mathsf{proreg}^\star \circ \mathsf{pro} \circ \alpha_j\big(\delta_0(\Delta'(\eta))\big) && \text{by Claim 19} \\
&= \tau(\psi_i)\left(\prod_{k=i+1}^{j} \tau(\phi_k)\right) \cdot \alpha_j(\Delta(\eta)) && \text{by Claim 22}
\end{aligned}$$

5 Message-independent sampling

We develop a sampling algorithm we call a *message-independent* sampling algorithm that may be used to interact with the prover in extraction and solution algorithms. The application is verifier-move QUIRKs in which extraction or solution requires obtaining many message-witness pairs such that the messages may be independent, not requiring any particular relationships. We model the scenario as having multiple boolean random variables, each modelling a sampling algorithm to obtain a single instance-witness pair, and our goal is to successfully sample from all of them.

Let P and Q denote boolean random variables with success probabilities p and q. For a boolean random variable V let $\tau(V)$ denote the random variable of the time it takes to sample V. One can successfully sample both P and Q in expected time $\mathbb{E}[\tau(P)]/p + \mathbb{E}[\tau(Q)]/q$ by sampling each until they both succeed. We instead seek to sample both in expected time linear in $\mathbb{E}[\tau(P)]$ and $\mathbb{E}[\tau(Q)]$, and we wish to succeed with probability nearly $\min\{p, q\}$. Moreover, in the case $p = q = 1$ we should succeed with probability 1.

We will construct two algorithms. The first will sample P and Q, and the second will utilize the first to sample n boolean random variables $\{P_i\}_{i\in[n]-1}$. We state the properties of the first algorithm in Theorem 3 and those of the second in Corollary 2.

Theorem 3. *For integer $k \geq 1$ an algorithm exists to obtain successful samples of P and Q with expected time bounded by $(k+1)(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)])/2$, and success probability 0 for $p = q = 0$ and otherwise at least*

$$\frac{\min\{p, q\} - 1/(k \cdot 2^k)}{1 - 1/(k \cdot 2^k)}$$

Whether the algorithm executes P or Q first, the algorithm succeeds only if that first execution succeeds.

Proof. We present the algorithm after Corollary 2. In Section 5.1 we prove the algorithm's probability of success, and in Section 5.2 we prove the algorithm's expected time. □

Corollary 2. *For integer $k \geq 1$ an algorithm exists to obtain successful samples of $n \geq 1$ boolean random variables $\{P_i\}_{i\in[n]-1}$ with expected time bounded by $\big((k+1)/2\big)^{\log(n)} \sum_{i=0}^{n-1} \mathbb{E}[\tau(P_i)]$, and success probability 0 when $p_i = 0$ for all $i \in [n]-1$ and otherwise at least*

$$\frac{\min\{p_i\}_{i\in[n]-1} - \log(n)/(k \cdot 2^k)}{1 - \log(n)/(k \cdot 2^k)}$$

with the logarithm base 2.

Proof. We organize the n random variables $\{P_i\}_{i\in[n]-1}$ into a binary tree padding for the case n is not a power of 2 with dummy random variables that succeed with probability 1. We argue by induction on the depth $d := \lceil \log(n) \rceil$. The case $d = 1$ holds by Theorem 3. For $d = 0$ we simply sample the single boolean random variable and succeed with probability p_0, satisfying the claim. For $d > 1$ let L and R denote the boolean random variables had by sampling the left and right subtrees. By induction, L and R succeed with probabilities lower bound by ℓ and r, defined as

$$\ell := \frac{\min\{p_i\}_{i\in[n/2]-1} - (d-1)/(k\cdot 2^k)}{1-(d-1)/(k\cdot 2^k)}\ , \quad r := \frac{\min\{p_i\}_{i\in[n/2]+(n/2-1)} - (d-1)/(k\cdot 2^k)}{1-(d-1)/(k\cdot 2^k)}$$

Therefore the probability we succeed on both L and R using Algorithm 4 is lower bound by

$$\begin{aligned} \frac{\min\{\ell,r\} - 1/(k\cdot 2^k)}{1-1/(k\cdot 2^k)} &= \left(\frac{\min\{p_i\}_{i\in[n]-1} - (d-1)/(k\cdot 2^k)}{1-(d-1)/(k\cdot 2^k)} - 1/(k\cdot 2^k)\right) \Big/ \big(1-1/(k\cdot 2^k)\big) \\ &= \frac{\min\{p_i\}_{i\in[n]-1} - \epsilon}{1-\epsilon} \\ \epsilon := 1 - \left(1-\frac{1}{k\cdot 2^k}\right)\left(1-\frac{d-1}{k\cdot 2^k}\right) &\le \frac{1}{k\cdot 2^k} + \frac{d-1}{k\cdot 2^k} = \frac{d}{k\cdot 2^k} \end{aligned}$$

The expected times for sampling L and R are bounded by Expressions 36 and 37, respectively.

$$\big((k+1)/2\big)^{d-1} \sum_{i\in[n/2]-1} \mathbb{E}[\tau(P_i)] \tag{36}$$

$$\big((k+1)/2\big)^{d-1} \sum_{i\in[n/2]+(n/2-1)} \mathbb{E}[\tau(P_i)] \tag{37}$$

Therefore the expected time of Algorithm 4 is bounded by

$$\begin{aligned} &\big((k+1)/2\big)\left(\big((k+1)/2\big)^{d-1} \sum_{i\in[n/2]-1} \mathbb{E}[\tau(P_i)] + \big((k+1)/2\big)^{d-1} \sum_{i\in[n/2]+(n/2-1)} \mathbb{E}[\tau(P_i)]\right) \\ &= \big((k+1)/2\big)\left(\big((k+1)/2\big)^{d-1} \sum_{i\in[n]-1} \mathbb{E}[\tau(P_i)]\right) = \big((k+1)/2\big)^{d} \sum_{i\in[n]-1} \mathbb{E}[\tau(P_i)] \end{aligned}$$

□

Remark 2. *Flipping coins randomizes the order of sampling. There is another version of this algorithm that succeeds with the same probability but has twice the expected running time for the case $n = 2$. For our purposes unordered sampling is sufficient so we choose the algorithm with half the expected running time.*

To execute Algorithm 4 for the binary case of P and Q, one first flips a coin and assigns P or Q as the *head random variable* H and the other as the *tail random variable* T depending on the coin flip. One then alterates between sampling H and T. One first samples H, and if it fails the algorithm aborts. Otherwise one proceeds to continue sampling, next sampling T, alternating until one of two events occur: T succeeds once, or H succeeds k times in addition to its first success. One executes the algorithm as follows with C the coin flip distribution and j the number of remaining times H can succeed before the algorithm fails.

Algorithm 4 Sampler algorithm for independents

```
1. match $c \leftarrow C$
     heads $\Rightarrow$
         $H := P, h := p, T := Q, t := q$
     tails $\Rightarrow$
         $H := Q, h := q, T := P, t := p$
2. match $b \leftarrow H$
     $0 \Rightarrow$ return $0$
     $1 \Rightarrow$ continue
3. $j := k$
4. if $j = 0$ return $0$
5. match $b \leftarrow T$
     $0 \Rightarrow$ continue
     $1 \Rightarrow$ return $1$
6. match $b \leftarrow H$
     $0 \Rightarrow$ goto Step 5
     $1 \Rightarrow$
         $j := j - 1$
         goto Step 4
```

5.1 Probability of success

We prove the lower bound on the probability of success in two parts. In Section 5.1.1 we model the probability of success as an expression written in h and t. We posit that the probability increases monotonically in both variables, and therefore to lower bound to expression we may lower bound the expression as a univariate with $h, t = \min\{h, t\}$. In Section 5.1.2 we lower bound the univariate expression for $k = 1$, and in Section 5.1.3 we lower bound the univariate expression for $k \geq 2$. Note that if $p = q = 0$ then the probability of success is 0 because with $h = 0$ the algorithm exits with failure in Step 3. Thus we are concerned below for the case $\max\{p, q\} > 0$ which translates to $\max\{h, t\} > 0$ by the way h and t are assigned.

5.1.1 Obtaining the univariate.

After an initial success it must be that $h > 0$, and we then repeat the following subroutine at most k times. The subroutine performs *rounds*, in each round sampling T and then H, until a round occurs in which either T or H succeeds. There may be $i \geq 0$ *failed rounds* in which both T and H fail. The probability a round fails is $(1-t)(1-h)$. We say the *subroutine succeeds* if after any number of failed rounds, in the final round T succeeds. We say the *subroutine fails* if after any number of failed rounds, in the final round T fails and then H succeeds. The probability the subroutine fails may be written

$$\sum_{i=0}^{\infty} (1-t)^i (1-h)^i \cdot (1-t)h = \frac{(1-t)h}{1-(1-t)(1-h)}$$

where we have written the geometric series in closed form taking care that $1-(1-t)(1-h) < 1$ due to $h > 0$.

The probability the algorithm succeeds is the probability H succeeds on initial execution (Step 2) and then the subroutine fails fewer than k times. We lower bound the probability the algorithm succeeds as

$$h\left(1 - \left(\frac{(1-t)h}{1-(1-t)(1-h)}\right)^k\right) \tag{38}$$

Claim 23 (Monotinicity of Equation (38))**.** *Probability 38 is monotonically non-decreasing in both variables.*

Proof. We prove monotinicity in each variable by showing non-negativity of the partial derivatives, though we omit the lengthy derivative calculations. Let

$$f(h,t) \coloneqq \frac{(1-t)h}{1-(1-t)(1-h)}$$

Monotinicity in h. The partial derivative of expresion 38 with respect to h is

$$1 - f(h,t)^k \cdot \left(1 + \frac{kt}{t+(1-t)h}\right)$$

Showing non-negativity of this derivative means showing the inequality

$$1 + \frac{kt}{t+(1-t)h} \le 1/f(h,t)^k = \left(1 + \frac{t}{(1-t)h}\right)^k$$

We rewrite the right side using the binomial formula as

$$\sum_{i=0}^{k} \binom{k}{i} \left(\frac{t}{(1-t)h}\right)^i = 1 + k\left(\frac{t}{(1-t)h}\right) + \sum_{i=2}^{k} \binom{k}{i} \left(\frac{t}{(1-t)h}\right)^i$$

With all terms non-negative and comparing denominators $(1-t)h \le t + (1-t)h$ we see the inequality holds.

Monotinicity in t. The partial derivative of expression 38 with respect to t is

$$f(h,t)^k \cdot \frac{kh}{(1-t)\big(h+(1-h)t\big)}$$

Both the numerator and denominator are non-negative. □

Due to probability 38 monotonically non-decreasing in both variables by Claim 23, we may lower bound the expression by plugging in $x \coloneqq \min\{h,t\}$ to get

$$h\left(1 - \left(\frac{(1-t)h}{1-(1-t)(1-h)}\right)^k\right) \ge x\left(1 - \left(\frac{(1-t)x}{1-(1-t)(1-x)}\right)^k\right) \ge x\left(1 - \left(\frac{(1-x)x}{1-(1-x)(1-x)}\right)^k\right)$$

where the first inequality holds by monotinicity in h, and the second by monotinicity in t.

Recall we are concerned with proving a lower bound assuming $\max\{h,t\} > 0$. So far this assumption has been enough to ensure the expressions considered are well defined. But with $x \coloneqq \min\{h,t\}$, the univariate expression in x is undefined when $h = 0$ or $t = 0$. To cover the case $\max\{h,t\} > 0$ but $h = 0$ or $t = 0$ one may refer back to the bivariate expression in h and t and confirm that it is well defined and equal to 0 for such cases. We therefore need only focus on the univariate expression for $x > 0$. With $x > 0$ we may simplify the expression by rearranging the fraction to cancel x/x and rewriting as

$$g(x) \coloneqq x\left(1 - \left(\frac{1-x}{2-x}\right)^k\right)$$

The function g represents a curve in the interval $[0,1]$ and passes through $(0,0)$ and $(1,1)$. We obtained g as a lower bound for the probability the algorithm succeeds assuming $x > 0$, and therefore g only applies as a lower bound for $x > 0$. But in the case $\min\{h,t\} = 0$ the probability of the algorithm succeeding is 0, which matches g since $g(0) = 0$. We may therefore treat g as a lower bound for the probability of success for all $\min\{h,t\} = x \in [0,1]$.

In Sections 5.1.2 and 5.1.3 we lower bound g in the unit interval with a line that passes through $(1,1)$. Relevant to both sections are the first and second derivatives of g written below, again omitting the calculations. One may calculate these derivatives using software like SageMath or SymPy.

$$g'(x) = \left(\frac{1-x}{2-x}\right)^k \left(\frac{kx}{(1-x)(2-x)} - 1\right) + 1$$

$$g''(x) = k \cdot \left(\frac{1-x}{2-x}\right)^k \left(\frac{4-(k+3)x}{(1-x)^2(2-x)^2}\right)$$

The tangent line touching the curve at $\alpha \in [0,1]$ may be written

$$g'(\alpha)(x-\alpha)+g(\alpha)$$

5.1.2 Lower bounding g for $k=1$.

With $k=1$ we rewrite the derivatives as

$$\begin{aligned} g'(x) &= \frac{(1-x)^{k-1}}{(2-x)^{k+1}}\big(kx-(1-x)(2-x)\big)+1 = \left(\frac{x-(1-x)(2-x)}{(2-x)^2}\right)+1 \qquad (39)\\ g''(x) &= \left(\frac{1-x}{2-x}\right)\left(\frac{4-4x}{(1-x)^2(2-x)^2}\right) = \frac{4}{(2-x)^3} \end{aligned}$$

Both derivatives are well defined and positive in $[0,1]$. Therefore the curve from $(0,0)$ to $(1,1)$ is concave up, and the tangent line at $\alpha \coloneqq 1$ lies at or below g for $[0,1]$. The tangent line then serves as our lower bound and we write it as

$$g'(1)(x-1)+g(1) = (1+1)(x-1)+1 = 2x-1 = \frac{x-1/2}{1-1/2}$$

5.1.3 Lower bounding g for $k \geq 2$.

With $k \geq 2$ we rewrite the first derivative as we did in the first equality of Equation (39). We also rewrite the second derivative.

$$\begin{aligned} g'(x) &= \frac{(1-x)^{k-1}}{(2-x)^{k+1}}\big(kx-(1-x)(2-x)\big)+1\\ g''(x) &= k\cdot\frac{(1-x)^{k-2}}{(2-x)^{k+2}}\big(4-(k+3)x\big) \end{aligned}$$

Both derivaties are well defined in $[0,1]$. The first derivative is positive in $[0,1]$. The second derivative crosses zero at $x=4/(k+3)$, is positive in $[0,4/(k+3)]$ and negative in $[4/(k+3),1)$. Without need to consider the second derivative at $x=1$ we conclude the curve is concave up in $[0,4/(k+3)]$ and concave down in $[4/(k+3),1]$.

Let α be a dynamic tangent point, and consider the tangent line ℓ at α. First setting $\alpha \coloneqq 4/(k+3)$ we infer by the concavities the following inequality relations between ℓ and g.

$$\begin{aligned} &\forall x \in [0,4/(k+3)],\, \ell(x) \leq g(x)\\ &\forall x \in [4/(k+3),1],\, g(x) \leq \ell(x) \end{aligned}$$

Suppose we gradually decrease α from $4/(k+3)$ towards 0. Then there appears a second intersection point $\beta \in [0,1]$ between ℓ and g such that we infer, again from the concavities, that

$$\begin{aligned} &\forall x \in [0,\alpha],\, \ell(x) \leq g(x)\\ &\forall x \in [\alpha,\beta],\, \ell(x) \leq g(x)\\ &\forall x \in [\beta,1],\, g(x) \leq \ell(x) \end{aligned}$$

Since we wish to make ℓ a lower bound for g we must decrease α to the point that $\beta=1$. Therefore we seek the unique line ℓ that is tangent to g a some point $\alpha \in [0,4/(k+3)]$ and passes through $(1,1)$.

The first part below is devoted to determining ℓ by determining the defining parameter α. Given the correct value α we obtain ℓ by plugging α into the definition of ℓ as a tangent line at point α. The second part below is devoted to solving ℓ such that we may express ℓ in terms of its root rather than as a tangent line.

Solving for α. As a line tangent to g at point α and as a line passing through $(1,1)$, ℓ may be defined as the unique line satisfying the two constraints

$$\begin{aligned} \exists\alpha \in [0,4/(k+3)],\ \ell(x) &= g'(\alpha)(x-\alpha)+g(\alpha)\\ 1 &= g'(\alpha)(1-\alpha)+g(\alpha) \qquad (40) \end{aligned}$$

To solve for α we solve Equation (40) reusing the original forms of g' and g''.

$$\begin{aligned}
1 &= g'(\alpha)(1-\alpha) + g(\alpha) \\
&= \left(\left(\frac{1-x}{2-x}\right)^k \left(\frac{k\alpha}{(1-\alpha)(2-\alpha)} - 1\right) + 1\right)(1-\alpha) + \alpha\left(1 - \left(\frac{1-\alpha}{2-\alpha}\right)^k\right) \\
&= \left(\frac{1-\alpha}{2-\alpha}\right)^k \frac{k\alpha(1-\alpha)}{(1-\alpha)(2-\alpha)} - \left(\frac{1-\alpha}{2-\alpha}\right)^k (1-\alpha) + (1-\alpha) + \alpha - \alpha\left(\frac{1-\alpha}{2-\alpha}\right)^k \\
&= \left(\frac{1-\alpha}{2-\alpha}\right)^k \frac{k\alpha}{2-\alpha} - \left(\frac{1-\alpha}{2-\alpha}\right)^k + 1 \\
\iff & \left(\frac{1-\alpha}{2-\alpha}\right)^k \left(\frac{k\alpha}{2-\alpha} - 1\right) = 0 \iff \frac{k\alpha}{2-\alpha} = 1 \iff \alpha = \frac{2}{k+1}
\end{aligned}$$

where $(1-\alpha)^k/(2-\alpha)^k > 0$ since $\alpha \in [0, 4/(k+3)]$ with $k > 1$.

Solving ℓ. Solving ℓ means solving the following equation for root x where $\alpha = 2/(k+1)$.

$$\ell(x) = g'(\alpha)(x-\alpha) + g(\alpha) = 0$$

After a long, omitted series of simplifications we arrive at the root $x = 1/(1+\gamma)$ for

$$\gamma := \frac{2^k(k-1)}{2\left(1-\frac{1}{k}\right)^k}$$

We handle $k = 2$, $k = 3$, and $k \geq 4$ independently.

- For $k = 2$ we have $\gamma = 8$, yielding root upper bound $1/9 > 1/(2 \cdot 2^2) = 1/(k \cdot 2^k)$.
- For $k = 3$ we have $\gamma = 27$, yielding root upper bound $1/28 > 1/(3 \cdot 2^3) = 1/(k \cdot 2^k)$.
- We wish to show the root is upper bounded by $k \cdot 2^k$ for $k \geq 4$. That translates to proving $k \cdot 2^k \leq 1 + \gamma$. By the bound $(1 - 1/k)^k < 1/e$ for $k \geq 1$ we have $2^k(k-1) \cdot e/2 < \gamma$ and may therefore alternatively prove $k \cdot 2^k \leq 1 + 2^k(k-1) \cdot e/2$.

$$\begin{aligned}
k \cdot 2^k \leq 1 + 2^k(k-1) \cdot e/2 &\Longleftarrow k \cdot 2^k \leq 2^k(k-1) \cdot e/2 \\
\iff 2k \leq (k-1) \cdot e &\iff e \leq k(e-2)
\end{aligned}$$

  With $k \geq 4$ we indeed have $k \geq e/(e-2) \approx 3.78$.

5.2 Expected time

Let S_j denote the expected remaining time of the algorithm upon reaching Step 4 with a particular $j \in \{0, \ldots, k\}$. Then the expected time of the algorithm is $\mathbb{E}[\tau(H)] + h \cdot S_k$.

Claim 24. *For $j \in [k]$,*

$$S_j = \frac{\mathbb{E}[\tau(T)] + (1-t)(\mathbb{E}[\tau(H)] + h \cdot S_{j-1})}{1 - (1-t)(1-h)} \tag{41}$$

Proof. Note $S_0 = 0$ because Step 4 immediately exits. When $j > 0$ we have S_j equal to the expected remaining time upon reaching Step 5 because Step 4 immediately proceeds to Step 5.

We may characterize S_j for $j > 0$ in terms of S_{j-1} and S_j. If T succeeds in Step 5 the expected remaining time is 0. If T fails and H succeeds the expected remaining time is S_{j-1} because we then enter Step 4 with $j := j - 1$. If T fails and H also fails the expected remaining time is S_j because we return to Step 5 with j unchanged. Accounting also for the expected times to sample T and H we may write

$$S_j = \mathbb{E}[\tau(T)] + t \cdot 0 + (1-t)\big(\mathbb{E}[\tau(H)] + h \cdot S_{j-1} + (1-h) \cdot S_j\big)$$

Rearranging we have

$$S_j - (1-t)(1-h) \cdot S_j = \mathbb{E}[\tau(T)] + (1-t)\big(\mathbb{E}[\tau(H)] + h \cdot S_{j-1}\big)$$

and the result follows upon dividing by $1-(1-t)(1-h)$. □

To proceed further in calculating expected time we must account for the random assignment of H and T between P and Q. Let SP_j denote S_j in the case $H \coloneqq P$ and let SQ_j denote S_j in the case $H \coloneqq Q$. Then by Equation (41) we have for $j > 0$

$$\begin{aligned}
\mathsf{SP}_j &= \frac{\mathbb{E}[\tau(Q)] + (1-q)\big(\mathbb{E}[\tau(P)] + p \cdot \mathsf{SP}_{j-1}\big)}{1-(1-q)(1-p)} \\
\mathsf{SQ}_j &= \frac{\mathbb{E}[\tau(P)] + (1-p)\big(\mathbb{E}[\tau(Q)] + q \cdot \mathsf{SQ}_{j-1}\big)}{1-(1-p)(1-q)}
\end{aligned}$$

Then the expected time of the algorithm may be written

$$\begin{aligned}
&\big(\mathbb{E}[\tau(P)] + p \cdot \mathsf{SP}_k\big)/2 + \big(\mathbb{E}[\tau(Q)] + q \cdot \mathsf{SQ}_k\big)/2 \\
&\quad = \big(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)]\big)/2 + \big(p \cdot \mathsf{SP}_k + q \cdot \mathsf{SQ}_k\big)/2
\end{aligned}$$

Therefore by proving $p \cdot \mathsf{SP}_k + q \cdot \mathsf{SQ}_k \le k\big(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)]\big)$ the desired expected time for the algorithm follows. We prove this inequality in the following claim.

Claim 25. *For* $j \in \{0, \dots, k\}$,

$$p \cdot \mathsf{SP}_j + q \cdot \mathsf{SQ}_j \le j\big(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)]\big)$$

Proof. We prove by induction. The base case for $j = 0$ holds by the fact $\mathsf{SQ}_0 = \mathsf{SP}_0 = 0$. Assuming the inequality holds for $j - 1 \ge 0$ we may write

$$\begin{aligned}
p \cdot \mathsf{SP}_j + q \cdot \mathsf{SQ}_j &\le p\left(\frac{\mathbb{E}[\tau(Q)] + (1-q)\big(\mathbb{E}[\tau(P)] + p \cdot \mathsf{SP}_{j-1}\big)}{1-(1-q)(1-p)}\right) \\
&\quad + q\left(\frac{\mathbb{E}[\tau(P)] + (1-p)\big(\mathbb{E}[\tau(Q)] + q \cdot \mathsf{SQ}_{j-1}\big)}{1-(1-p)(1-q)}\right) \\
&= \left(\frac{p \cdot \mathbb{E}[\tau(Q)] + p(1-q) \cdot \mathbb{E}[\tau(P)] + q \cdot \mathbb{E}[\tau(P)] + q(1-p) \cdot \mathbb{E}[\tau(Q)]}{1-(1-q)(1-p)}\right) \\
&\quad + \left(\frac{p(1-q)p \cdot \mathsf{SP}_{j-1} + q(1-p)q \cdot \mathsf{SQ}_{j-1}}{1-(1-p)(1-q)}\right) \\
&\le \left(\frac{\big(q + p(1-q)\big) \cdot \mathbb{E}[\tau(P)] + \big(p + q(1-p)\big) \cdot \mathbb{E}[\tau(Q)]}{1-(1-q)(1-p)}\right) \\
&\quad + \left(\frac{\big(q + p(1-q)\big)p \cdot \mathsf{SP}_{j-1} + \big(p + q(1-p)\big)q \cdot \mathsf{SQ}_{j-1}}{1-(1-p)(1-q)}\right) && (42) \\
&= \big(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)]\big) + \big(p \cdot \mathsf{SP}_{j-1} + q \cdot \mathsf{SQ}_{j-1}\big) \\
&\le \big(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)]\big) + (j-1)\big(\mathbb{E}[\tau(P)] + \mathbb{E}[\tau(Q)]\big) && (43)
\end{aligned}$$

where in Equation (42) we have added pq to the factors multiplying SP_{j-1} and SQ_{j-1}, and in Equation (43) we invoke induction. □

6 Message-dependent sampling

We develop a sampling algorithm we call a *message-dependent* sampling algorithm that may be used to interact with the prover in extraction and solution algorithms. The application is verifier-move QUIRKs in which extraction or solution requires obtaining many message-witness pairs such that the messages are dependent, requiring particular relationships we capture with a notion of monotonicity. The notion of monotonicity also appears in [AFR23] serving a similar purpose, but the two techniques are independent and we achieve a tighter result.

To model a relationship via monotonicity we consider a function χ that maps from sequences (of any finite length) of elements in a set Ω to subsets of Ω.

$$\chi\colon \Omega^\star \to 2^\Omega$$

Definition 13. *Let $\chi\colon \Omega^\star \to 2^\Omega$ denote a function from all sequences of elements of a set Ω to all subsets of Ω. We say χ is* monotonic *if for any $j \geq 0$, sequence $(\omega_1, \dots, \omega_j) \in \Omega^j$, and $\omega_{j+1} \in \Omega$ we have*

$$\chi(\omega_1, \dots, \omega_{j-1}) \subseteq \chi(\omega_1, \dots, \omega_{j-1}, \omega_j)$$

We say χ is strictly monotonic *if the subset is a strict subset.*

Theorem 4. *For $k \geq 0$ and monotonic function χ we construct algorithms $\mathsf{V1}_{k+1}$ (version 1) and $\mathsf{V2}_{k+1}$ (version 2) that attempt to sample $(e_1, \dots, e_{k+1}) \in E^{k+1} \subseteq \Omega^{k+1}$ such that*

$$\forall i \in [k+1],\ \chi\big(e_1, \dots, e_{i-1}\big) \subset \chi\big(e_1, \dots, e_i\big)$$

Algorithms $\mathsf{V1}_{k+1}$ and $\mathsf{V2}_{k+1}$ operate with probabilities of success

$$\Pr\Big[m \in \mathsf{Yes}\big(E^{k+1}\big)\ \Big|\ m \leftarrow \mathsf{V1}_{k+1}()\Big] \geq \Pr[E] - \Pr[\chi(e_1, \dots, e_k)]$$
$$\Pr\Big[m \in \mathsf{Yes}\big(E^{k+1}\big)\ \Big|\ m \leftarrow \mathsf{V2}_{k+1}()\Big] \geq \frac{\Pr[E] - \Pr[\chi(e_1, \dots, e_k)]}{1 - \Pr[\chi(e_1, \dots, e_k)]}$$

and expected running times

$$\mathbb{E}\big[\mathsf{V1}_{k+1}()\big] \leq k+1,\ \ \mathbb{E}\big[\mathsf{V2}_{k+1}()\big] \leq 2(k+1)$$

Proof. We prove here the base case for $k = 0$ and will subsequently prove by induction for $k \geq 1$. The algorithms for $\mathsf{V1}_1$ and $\mathsf{V2}_1$ are identical with common algorithm denoted V presented in Algorithm 5. Algorithm V succeeds with probability $\Pr[E]$ by simply sampling $\omega \leftarrow \Omega$ and succeeds if and only if $\omega \in E$. With $\chi(()) = \varnothing$ we have $\Pr[\chi(())] = 0$ and therefore both algorithms succeed with the probabilities claimed. The expected time of V is 1 so both algorithms have expected times bounded as claimed. $\square$

In Section 6.1 we present the algorithms and prove two preliminary claims. In Section 6.2 we lower bound the probabilities of success. In Section 6.3 we upper bound the probabilities of success for the purpose of upper bounding the expected times. We upper bound the expected times in Section 6.4.

6.1 The algorithm

Algorithm 5 Message-dependent sampling algorithms trySuccess and tryOutside

```
trySuccess: $2^\Omega \to 2^\Omega \to \mathsf{Maybe}\langle E\rangle$
trySuccess($S_{k-1}$)($S_k$) :=
    $\omega \leftarrow \Omega \setminus S_{k-1}$
    match $\omega \in E$
        $0 \Rightarrow$ trySuccess($S_{k-1}$)($S_k$)
        $1 \Rightarrow$
            match $\omega \in S_k$
                $0 \Rightarrow$ Yes($\omega$)
                $1 \Rightarrow$ No

tryOutside: $2^\Omega \to \mathsf{Maybe}\langle E\rangle$
tryOutside($S_k$) :=
    $\omega \leftarrow \Omega \setminus S_k$
    match $\omega \in E$
        $0 \Rightarrow$ No
        $1 \Rightarrow$ Yes($\omega$)
```

Algorithm 6 Message-dependent sampling algorithms V, $\mathsf{V1}_{k+1}$, and $\mathsf{V2}_{k+1}$

```
$\mathsf{V}\colon () \to \mathsf{Maybe}\langle E\rangle$
$\mathsf{V}() :=$
    $\omega \leftarrow \Omega$
    **match** $\omega \in E$
        $0 \Rightarrow \mathsf{No}$
        $1 \Rightarrow \mathsf{Yes}(\omega)$

$\forall k \geq 1\colon$
$\mathsf{V1}_{k+1}\colon () \to \mathsf{Maybe}\langle E^{k+1}\rangle$
$\mathsf{V1}_{k+1}() :=$
    **match** $m \leftarrow \mathsf{V1}_k()$
        $\mathsf{No} \Rightarrow \mathsf{No}$
        $\mathsf{Yes}((e_1, \ldots, e_k)) \Rightarrow$
            $S_{k-1} := \chi(e_1, \ldots, e_{k-1})$
            $S_k := \chi(e_1, \ldots, e_k)$
            **match** $\alpha \leftarrow \mathsf{trySuccess}(S_{k-1})(S_k)$
                $\mathsf{No} \Rightarrow \mathsf{No}$
                $\mathsf{Yes}(e_{k+1}) \Rightarrow (e_1, \ldots, e_k, e_{k+1})$

$\forall k \geq 1\colon$
$\mathsf{V2}_{k+1}\colon () \to \mathsf{Maybe}\langle E^{k+1}\rangle$
$\mathsf{V2}_{k+1}() :=$
    **match** $m \leftarrow \mathsf{V2}_k()$
        $\mathsf{No} \Rightarrow \mathsf{No}$
        $\mathsf{Yes}((e_1, \ldots, e_k)) \Rightarrow$
            $S_{k-1} := \chi(e_1, \ldots, e_{k-1})$
            $S_k := \chi(e_1, \ldots, e_k)$
            $\alpha \leftarrow \mathsf{trySuccess}(S_{k-1})(S_k)$
            $\beta \leftarrow \mathsf{tryOutside}(S_k)$
            **match** $(\alpha, \beta)$
                $(\mathsf{No}, \mathsf{No}) \Rightarrow \mathsf{No}$
                $(\mathsf{Yes}(e_{k+1}), \cdot\,) \Rightarrow (e_1, \ldots, e_k, e_{k+1})$
                $(\,\cdot\,, \mathsf{Yes}(e_{k+1})) \Rightarrow (e_1, \ldots, e_k, e_{k+1})$
```

Claim 26.

$$\Pr\left[\alpha \in \mathsf{Yes}\langle E\rangle \;\middle|\; \alpha \leftarrow \mathsf{trySuccess}(S_{k-1})(S_k)\right] = \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}$$

Proof. In $\mathsf{trySuccess}$ we sample from the space $\Omega \setminus S_{k-1}$ until obtaining a sample in E, that is a sample in $E \cap (\Omega \setminus S_{k-1})$. The algorithm returns a $\mathsf{Yes}\langle E\rangle$ value if and only if this sample is not in S_k, that is the sample is in $E \cap (\Omega \setminus S_k)$. Therefore the probability the algorithm returns a $\mathsf{Yes}\langle E\rangle$ value is the probability a sample is in $E \cap (\Omega \setminus S_k)$ given that it is in $E \cap (\Omega \setminus S_{k-1})$. We write this conditional probability as

$$\Pr\left[E \cap (\Omega \setminus S_k) \mid E \cap (\Omega \setminus S_{k-1})\right] = \frac{\Pr\left[E \cap (\Omega \setminus S_k) \cap (\Omega \setminus S_{k-1})\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]} = \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}$$

where the first equality is due to the definition of conditional probability, and the second equality is due to $S_{k-1} \subseteq S_k$. ☐

Claim 27.

$$\Pr\left[\beta \in \mathsf{Yes}\langle E\rangle \;\middle|\; \beta \leftarrow \mathsf{tryOutside}(S_k)\right] = \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]}$$

Proof. In $\mathsf{tryOutside}$ we sample once from the space $\Omega \setminus S_k$ and return a $\mathsf{Yes}\langle E\rangle$ value if and only if the sample is in E, that is the sample is in $E \cap (\Omega \setminus S_k)$. We write this conditional probability as

$$\Pr\left[E \cap (\Omega \setminus S_k) \mid \Omega \setminus S_k\right] = \frac{\Pr\left[E \cap (\Omega \setminus S_k) \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]} = \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]}$$

☐

6.2 Lower bounding probability of success

Lower bounding for V1

Claim 28.

$$\Pr\left[m \in \mathsf{Yes}(E^{k+1}) \;\middle|\; m \leftarrow \mathsf{V1}_{k+1}()\right] = \Pr\left[E \cap (\Omega \setminus S_k)\right]$$

Proof. We prove by induction on k. For the base case of $k = 0$ first note that by assumption of monotonicity we regard $S_k \coloneqq \chi(()) = \varnothing$ and therefore $E \cap (\Omega \setminus S_0) = E$. With $k = 0$ we have $\mathsf{V1}_{k+1} = \mathsf{V}$ which succeeds with probability $\Pr[E]$, and the result follows.

For $k \geq 1$, the probability that $\mathsf{V1}_{k+1}$ succeeds is the probability that $\mathsf{V1}_k$ succeeds and $\mathsf{trySuccess}$ on S_{k-1} and S_k also succeeds. By induction $\mathsf{V1}_k$ succeeds with probability $\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]$. Multiplying this probability by the probability that $\mathsf{trySuccess}$ succeeds as established in Claim 26 we write probability of success for $\mathsf{V1}_{k+1}$ as

$$\begin{aligned}
&\Pr\left[m \in \mathsf{Yes}(E^k) \,\middle|\, m \leftarrow \mathsf{V1}_k()\right] \cdot \Pr\left[\alpha \in \mathsf{Yes}(E) \,\middle|\, \alpha \leftarrow \mathsf{trySuccess}(S_{k-1})(S_k)\right] \\
&= \Pr\left[E \cap (\Omega \setminus S_{k-1})\right] \cdot \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]} = \Pr\left[E \cap (\Omega \setminus S_k)\right]
\end{aligned}$$

□

To obtain the desired lower bound for $\mathsf{V1}$ we use Claim 28 to write

$$\Pr\left[m \in \mathsf{Yes}(E^{k+1}) \,\middle|\, m \leftarrow \mathsf{V1}_{k+1}()\right] = \Pr\left[E \cap (\Omega \setminus S_k)\right] = \Pr[E] - \Pr[E \cap S_k] \geq \Pr[E] - \Pr[S_k]$$

Lower bounding for $\mathsf{V2}$

Lemma 3. *For $a, b, A, B \geq 0$ with $b \geq a$ and $B \geq A$ we have*

$$\forall x > a,\ B/A \geq (x - b)/(x - a)$$

Proof. We instead show $B(x - a) \geq A(x - b)$. First we establish $a(B - A) \geq Ba - Ab$.

$$b \geq a \iff Ab \geq Aa \iff Ba - Aa \geq Ba - Ab \iff a(B - A) \geq Ba - Ab$$

Second we establish $x(B - A) \geq Ba - Ab$ using the previous inequality for the last inequality below.

$$x \geq a \iff x(B - A) \geq a(B - A) \geq Ba - Ab$$

We conclude assuming $x(B - A) \geq Ba - Ab$.

$$x(B - A) \geq Ba - Ab \iff Bx - Ax \geq Ba - Ab \iff Bx - Ba \geq Ax - Ab \iff B(x - a) \geq A(x - b)$$

□

Assigning x, a, b, A, B as below we note that with $S_{k-1} \subseteq S_k$ we have $a \leq b$ and $A \leq B$. Also recall the assumption $\Pr[E] > \Pr[S_{k-1}]$ and thus $x > a$.

$$x \coloneqq \Pr[E],\ a \coloneqq \Pr[S_{k-1}],\ b \coloneqq \Pr[S_k],\ A \coloneqq \Pr[E] - \Pr[E \cap S_k],\ B \coloneqq \Pr[E] - \Pr[E \cap S_{k-1}]$$

Then we may invoke Lemma 3 and write

$$\frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]} = \frac{\Pr[E] - \Pr[E \cap S_k]}{\Pr[E] - \Pr[E \cap S_{k-1}]} \geq \frac{\Pr[E] - \Pr[S_k]}{\Pr[E] - \Pr[S_{k-1}]}$$

The probability $\mathsf{V2}_{k+1}$ succeeds is the probability $\mathsf{V2}_k$ succeeds and either $\mathsf{trySuccess}$ or $\mathsf{tryOutside}$ succeeds. The probability of the disjunction that not both $\mathsf{trySuccess}$ and $\mathsf{tryOutside}$ fail may be written

$$\begin{aligned}
&\left(1 - \left(1 - \Pr\left[\alpha \in \mathsf{Yes}(E) \,\middle|\, \alpha \leftarrow \mathsf{trySuccess}(S_{k-1})(S_k)\right]\right)\left(1 - \Pr\left[\beta \in \mathsf{Yes}(E) \,\middle|\, \beta \leftarrow \mathsf{tryOutside}(S_k)\right]\right)\right) \\
&= \left(1 - \left(1 - \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}\right)\left(1 - \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]}\right)\right) \qquad (44)
\end{aligned}$$

Combining results we have the inequality below, and the following equality is had be simplification.

$$\begin{aligned}
&\Pr\left[m \in \mathsf{Yes}(E^{k+1}) \,\middle|\, m \leftarrow \mathsf{V2}_{k+1}()\right] \\
&\geq \left(\frac{\Pr[E] - \Pr[S_{k-1}]}{1 - \Pr[S_{k-1}]}\right)\left(1 - \left(1 - \frac{\Pr[E] - \Pr[S_k]}{\Pr[E] - \Pr[S_{k-1}]}\right)\left(1 - \frac{\Pr[E] - \Pr[S_k]}{1 - \Pr[S_k]}\right)\right) \\
&= \frac{\Pr[E] - \Pr[S_k]}{1 - \Pr[S_k]}
\end{aligned}$$

6.3 Upper bounding probability of success

The upper bound for the probability that $\mathsf{V1}$ succeeds is had by Claim 28 since it is an equality. We turn to upper bounding the probability of success for $\mathsf{V2}$. These upper bounds serve in lower bounding the expected times.

Claim 29.

$$\Pr\left[m \in \mathsf{Yes}\big(E^{k+1}\big) \;\Big|\; m \leftarrow \mathsf{V2}_{k+1}()\right] \le \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]} \tag{45}$$

Proof. for Version 2 Equation (44)

$$\begin{aligned}
&\Pr\left[m \in \mathsf{Yes}\big(E^{k+1}\big) \;\Big|\; m \leftarrow \mathsf{V2}_{k+1}()\right] \\
&\le \left(\frac{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]}\right)\left(1 - \left(1 - \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}\right)\left(1 - \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]}\right)\right) \\
&= \left(\frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_k\right]}\right)\left(\frac{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]} + \frac{\Pr\left[\Omega \setminus S_k\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]} - \frac{\Pr\left[E \cap (\Omega \setminus S_k)\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]}\right)
\end{aligned}$$

$$\begin{aligned}
&\Pr\left[E \cap (\Omega \setminus S_{k-1})\right] + \Pr\left[\Omega \setminus S_k\right] - \Pr\left[E \cap (\Omega \setminus S_k)\right] \le \Pr\left[\Omega \setminus S_{k-1}\right] \\
&\iff \Pr\left[\Omega \setminus S_k\right] - \Pr\left[E \cap (\Omega \setminus S_k)\right] \le \Pr\left[\Omega \setminus S_{k-1}\right] - \Pr\left[E \cap (\Omega \setminus S_{k-1})\right] \\
&\iff 1 - \Pr\left[S_k\right] - \big(\Pr[E] - \Pr\left[E \cap S_k\right]\big) \le 1 - \Pr\left[S_{k-1}\right] - \big(\Pr[E] - \Pr\left[E \cap S_{k-1}\right]\big) \\
&\iff \Pr\left[S_{k-1}\right] - \Pr\left[E \cap S_{k-1}\right] \le \Pr\left[S_k\right] - \Pr\left[E \cap S_k\right] \\
&\quad = \Pr\left[S_{k-1} \cup (S_k \setminus S_{k-1})\right] - \Pr\left[E \cap \big(S_{k-1} \cup (S_k \setminus S_{k-1})\big)\right] \\
&\quad = \Pr\left[S_{k-1}\right] + \Pr\left[S_k \setminus S_{k-1}\right] - \Pr\left[E \cap S_{k-1}\right] - \Pr\left[E \cap (S_k \setminus S_{k-1})\right] \\
&\iff \Pr\left[E \cap (S_k \setminus S_{k-1})\right] \le \Pr\left[S_k \setminus S_{k-1}\right]
\end{aligned}$$

where we have utilized $S_{k-1} \subseteq S_k$.

□

6.4 Expected running time

Claim 30.

$$\begin{aligned}
\mathbb{E}\Big[\tau\big(\mathsf{trySuccess}(S_{k-1})(S_k)\big)\Big] &= \frac{\Pr\left[\Omega \setminus S_{k-1}\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]} \\
\mathbb{E}\big[\tau\big(\mathsf{tryOutside}(S_k)\big)\big] &= 1
\end{aligned}$$

Proof. The algorithm $\mathsf{trySuccess}$ samples from the space $\Omega \setminus S_{k-1}$ until obtaining a value in $E \cap (\Omega \setminus S_{k-1})$. The probability the algorithm succeeds on each sample is thus

$$\frac{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]}$$

so the expected number of samples needed is the inverse.

The algorithm $\mathsf{tryOutside}$ samples once from the space $\Omega \setminus S_k$ so takes time 1. □

Expected time for $\mathsf{V1}$. The expected time fo $\mathsf{V1}_{k+1}$ is the expected time for $\mathsf{V1}_k$ plus the time of $\mathsf{trySuccess}$ should $\mathsf{V1}_k$ succeed. Therefore we write

$$\mathbb{E}\big[\tau\big(\mathsf{V1}_{k+1}\big)\big] = \mathbb{E}\big[\tau\big(\mathsf{V1}_k\big)\big] + \Pr\left[m \in \mathsf{Yes}\big(E^k\big) \;\Big|\; m \leftarrow \mathsf{V1}_k()\right] \cdot \mathbb{E}\Big[\tau\big(\mathsf{trySuccess}(S_{k-1})(S_k)\big)\Big]$$

Plugging in the probability $\mathsf{V1}_k$ succeeds as established in Claim 28 and plugging in the expected time of $\mathsf{V1}_k$ had by induction we arrive at By induction we have The probability $\mathsf{V1}_k$ succeeds is $\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]$ which holds by Claim 28 for k rather than $k+1$.

$$\begin{aligned}\mathbb{E}\big[\tau\big(\mathsf{V1}_{k+1}\big)\big] &= \mathbb{E}\big[\tau(\mathsf{V1}_k)\big] + \Pr\left[E \cap (\Omega \setminus S_{k-1})\right] \cdot \frac{\Pr\left[\Omega \setminus S_{k-1}\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]} \\ &\leq k + \Pr\left[\Omega \setminus S_{k-1}\right] \leq k+1\end{aligned}$$

Expected time for $\mathsf{V2}$**.** The expected time for $\mathsf{V2}_{k+1}$ is the expected time for $\mathsf{V2}_k$ plus the time of $\mathsf{trySuccess}$ and $\mathsf{tryOutside}$ should $\mathsf{V2}_k$ succeed. Therefore we write

$$\begin{aligned}&\mathbb{E}\big[\tau\big(\mathsf{V2}_{k+1}\big)\big] = \mathbb{E}\big[\tau(\mathsf{V2}_k)\big] \\ &\quad + \Pr\left[m \in \mathsf{Yes}\big(E^k\big) \;\middle|\; m \leftarrow \mathsf{V2}_k()\right] \cdot \mathbb{E}\Big[\tau\big(\mathsf{trySuccess}(S_{k-1})(S_k)\big)\Big] \cdot \mathbb{E}\Big[\tau\big(\mathsf{tryOutside}(S_k)\big)\Big]\end{aligned}$$

Plugging in the upper bound on the probability $\mathsf{V2}_k$ succeeds as established in Equation (45) for k rather than $k+1$, and plugging in the expected time of $\mathsf{V2}_k$ had by induction we arrive at

$$\begin{aligned}\mathbb{E}\big[\tau\big(\mathsf{V2}_{k+1}\big)\big] &= \mathbb{E}\big[\tau(\mathsf{V2}_k)\big] + \left(\frac{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]}\right) \cdot \left(\frac{\Pr\left[\Omega \setminus S_{k-1}\right]}{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]} + 1\right) \\ &\leq (2k) + 1 + \frac{\Pr\left[E \cap (\Omega \setminus S_{k-1})\right]}{\Pr\left[\Omega \setminus S_{k-1}\right]} \leq 2(k+1)\end{aligned}$$

7 Instance and witness reductions

We classify QUIRKs into two types we call 'instance reductions' and 'witness reductions.' Intuitively, an instance reduction is one in which the language may change and the instance may undergo trivial or non-trivial reduction, but the witness only undergoes trivial reduction. The term can be misleading because it does not include QUIRKs in which both the instance and the witness undergo non-trivial reduction. We choose the name, however, to give emphasis to the instance, rather than the witness, as undergoing non-trivial reduction. A QUIRK is a witness reduction if it is not an instance reduction. To capture the special class of QUIRKs that are instance reductions, we define a QUIRK as an instance reduction if it executes δ only once during extraction. Intuitively, this requires that a witness for the input instance be immediately derived from a single output witness. The immediate transformation of an output witness into an input witness suggests the input witness did not undergo non-trivial reduction.

Definition 14 (Instance and witness reduction)**.** *A QUIRK is an* instance reduction *if the* $\mathsf{extract}$ *function executes δ only once. Otherwise, the QUIRK is a* witness reduction.

In the following four subsections we distinguish these two types of reductions for prover-move and verifier-move QUIRKs, and we present an example QUIRK pattern for the latter three subsections each.

7.1 Prover witness reductions

When a prover-move reduction involves non-trivial reduction to the witness, the verifier has no randomness to exercise in attempt at extraction. If the verifier rewinds the prover, there's no guarantee the prover will return a different witness useful for extraction. It seems the best one may do in this case is make use of assumptions, likely non-falsifiable assumptions. In this case one may leave the $\mathsf{extract}$ or solve functions of a QUIRK undefined and instead conjecture security.

7.2 Prover instance reductions

Consider functions $\mathsf{message}$, $\mathsf{instance}$, and $\mathsf{witness}$ for building a prover-move QUIRK with a message type M between language $L(I;W)$ and $L'(I';W')$. We show how another function $\mathsf{witOrSol}$ allows us to construct functions extract and solve to complete the prover-move QUIRK. Intuitively the resulting class of QUIRKs is intended to capture all prover-move reductions in which the instance and the language may change, but the witness hardly changes if at all. The function name witOrSol is short for ‘witness or solution’ and given an input instance, prover message, and output witness (in W'), we ask that it immediately (without further interaction with the prover) yields either an input witness (in W) or a solution for some solution type S associated with the QUIRK. Such a function implies that an output witness effectively contains all that’s needed to derive an input witness (or a solution), and thus the witness has not undergone any non-trivial reduction.

The function witOrSol has signature

$$\mathsf{witOrSol}\colon I \to M \to W' \to \mathsf{Either}\langle W, S\rangle$$

and it is expected to output an input witness or a solution assuming inputs $x \in I$, $m \in M$, and $w' \in W'$ satisfy $\big(\mathsf{instance}(x)(m)\ ;\ w'\big) \in L'$. We construct both extract and solve in terms of a subroutine we call core in which we utilize witOrExt. We construct core, extract, and solve in Algorithm 7

Algorithm 7 Functions for prover-move instance reduction

```
$\mathsf{core}\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle \mathsf{Either}\langle W, S\rangle, \mathsf{Error}\rangle$
$\mathsf{core}(\eta)(x)(\mathcal{P}) \coloneqq$
    $m \leftarrow \mathcal{P}.\mathsf{message}()$
    **match** $r \leftarrow \eta\big(\mathsf{instance}(x)(m)\big)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}(w') \Rightarrow \mathsf{witOrSol}(x)(m)(w')$
$\mathsf{extract}\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle W, \mathsf{Error}\rangle$
$\mathsf{extract}(\eta)(x)(\mathcal{P}) \coloneqq$
    **match** $r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}(\mathsf{Left}(w)) \Rightarrow \mathsf{Ok}(\mathsf{w})$
        $\mathsf{Ok}(\mathsf{Right}(s)) \Rightarrow \mathsf{Err}(\mathsf{Fail})$
$\mathsf{solve}\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle S, \mathsf{Error}\rangle$
$\mathsf{solve}(\eta)(x)(\mathcal{P}) \coloneqq$
    **match** $r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}(\mathsf{Left}(w)) \Rightarrow \mathsf{Err}(\mathsf{Fail})$
        $\mathsf{Ok}(\mathsf{Right}(s)) \Rightarrow \mathsf{Ok}(\mathsf{s})$
```

Lemma 4. *Given prover-move QUIRK functions* message*,* instance*, and* witness *as well as a* witOrSol *function, we may construct functions* extract *and* solve *to complete the QUIRK with extractability error* 0 *and expected extraction and solution times both* 1.

Proof. Regarding extractability error, note that by construction of extract and solve we have

$$\begin{aligned}
&\Pr\left[r \in \mathsf{Ok}(W) \,\middle|\, r \leftarrow \mathsf{extract}(\eta)(x)(\mathcal{P})\right] + \Pr\left[r \in \mathsf{Ok}(S) \,\middle|\, r \leftarrow \mathsf{solve}(\eta)(x)(\mathcal{P})\right]\\
&= \Pr\left[r \in \mathsf{Ok}(\mathsf{Either}\langle W, S\rangle) \,\middle|\, r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P})\right]\\
&= \Pr\left[r \in \mathsf{Ok}(W') \,\middle|\, r \leftarrow \delta(\eta)(x)(\mathcal{P})\right]
\end{aligned}$$

Therefore we have extractability error 0.

Regarding expected solution and extraction times, note how core executes the equivalent of δ exactly once, therefore so do extract and solve. □

7.3 Verifier-move witness reductions

Suppose we have three functions message, instance, and witness for a verifier-move QUIRK with message type M between two languages $L(I,W)$ and $L'(I',W')$. We will construct a class of QUIRKs by complementing the three functions with a class of extract and solve functions. In addition to the three functions, the message type, and the two languages, these QUIRKs are parameterized by the following.

- A solution type S.
- Integers $\ell > 0$ and $k_i > 0$ for $i \in [\ell]$.
- For each $i \in [\ell]$ a pair of types $U_i \times V_i \cong M$ that decompose the message type M, and such that sampling from M can be done by sampling $u \leftarrow U_i$ and $v \leftarrow V_i$ and joining them with a function $\mathsf{join}_i \colon U_i \to V_i \to M$ as $\mathsf{join}_i(u)(v)$. Note a trivial decomposition can always be had with U_i or V_i assigned the unit type and the other assigned the M type.
- For each $i \in [\ell]$ a monotonic function $\chi_i \colon V_i^\star \to 2^{V_i}$, as defined below in Definition 13.
- A 'witness or solution' function witOrSol with signature

  $$\mathsf{witOrSol} \colon I \to \textstyle\prod_{i\in[\ell]} U_i \times (V_i \times W')^{k_i+1} \to \mathsf{Either}\langle W, S\rangle$$

  that will only be invoked with inputs $x \in I$ and

  $$\big(u_i, (v_{i,j}, w'_{i,j})_{j\in[k_i+1]}\big)_{i\in[\ell]} \in \prod_{i\in[\ell]} U_i \times (V_i \times W')^{k_i+1}$$

  that satisfy

  $$\begin{aligned}&\forall i \in [\ell],\ \forall j \in [k_i+1] \colon\\ &\quad \big(\mathsf{instance}(x)\big(\mathsf{join}_i(u_i)(v_{i,j})\big)\ ;\ w'_{i,j}\big) \in L'(I';W')\\ &\quad \chi_i(v_{i,1},\ldots,v_{i,j-1}) \subset \chi_i(v_{i,1},\ldots,v_{i,j})\end{aligned}$$

  the former which we call *validity* and the latter *strict monotonicity*.

Given these parameters, we create a function core of signature

$$\mathsf{core} \colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle \mathsf{Either}\langle W, S\rangle, \mathsf{Error}\rangle$$

and create functions extract and solve by wrapping core as follows in Algorithm 8.

Algorithm 8 extract and solve wrapping core for verifier-move witness reduction

```
$\mathsf{extract} \colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle W, \mathsf{Error}\rangle$
$\mathsf{extract}(\eta)(x)(\mathcal{P}) :=$
    **match** $r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}(\mathsf{Left}(w)) \Rightarrow \mathsf{Ok}(w)$
        $\mathsf{Ok}(\mathsf{Right}(s)) \Rightarrow \mathsf{Err}(\mathsf{Fail})$
$\mathsf{solve} \colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle S, \mathsf{Error}\rangle$
$\mathsf{solve}(\eta)(x)(\mathcal{P}) :=$
    **match** $r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}(\mathsf{Left}(w)) \Rightarrow \mathsf{Err}(\mathsf{Fail})$
        $\mathsf{Ok}(\mathsf{Right}(s)) \Rightarrow \mathsf{Ok}(s)$
```

We must enforce that extract and solve meet the implicit assumption of returning a regression error if and only if the first execution of δ returns a regression error. To enforce this assumption we must enforce it on core. For simplicity we will not discuss further how we enforce this assumption on core, noting it may simply be done by observing the execution of core and adjusting returned errors as necessary.

We will construct core by utilizing the sampling algorithms presented in Section 5 and Section 6. In core we aim to extract the input to witOrSol of type $\prod_{i\in[\ell]} U_i \times (V_i \times W')^{k_i+1}$ such that we may feed it along with instance $x \in I$ to witOrSol and receive output of type $\mathsf{Either}\langle W, S\rangle$ for the Ok variant of the output for core. If we fail to extract the necessary input for witOrSol then we output an Error value for the Err variant of the output for core, but we must take care on whether to output Err(Regress) or Err(Fail) as mentioned in the previous paragraph.

Algorithm core is constructed in terms of algorithms core_i for $i \in [\ell]$ of signatures

$$\mathsf{core}_i \colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle U_i \times (V_i \times W')^{k_i+1}, \mathsf{Error}\rangle$$

We aim to extract with each core_i a value of type $U_i \times (V_i \times W')^{k_i+1}$, and then compose these values into one of type $\prod_{i\in[\ell]} U_i \times (V_i \times W')^{k_i+1}$ for the output of core.

In Section 7.3.1 we illustrate how we may construct core out of $(\mathsf{core}_i)_{i\in[\ell]}$ using the sampling algorithm of Section 5. In Section 7.3.2 we then construct $(\mathsf{core}_i)_{i\in[\ell]}$ using the sampling algorithm of Section 6.

7.3.1 Constructing core

Let us treat $(\mathsf{core}_i)_{i\in[\ell]}$ as ℓ boolean random variables, each yielding success or failure depending on whether the output is of variant Ok or Err. Define the algorithm core to be that which executes the message-independent sampling algorithm of Section 5 in attempt to sample all core_i binary random variables successfully, to then plug the results into witOrSol for output. Then by Corollary 2, for some $\lambda \in \mathbb{N}$, core succeeds with probability

$$\begin{aligned} &\Pr\left[r \in \mathsf{Ok}(\mathsf{Either}\langle W, S\rangle) \,\middle|\, r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P})\right] \\ &\quad \geq \min_{i\in[\ell]} \left\{ \Pr\left[r \in \mathsf{Ok}\big(U_i \times (V_i \times W')^{k_i+1}\big) \,\middle|\, r \leftarrow \mathsf{core}_i(\eta)(x)(\mathcal{P})\right]\right\} - \log(\ell)/\big(\lambda \cdot 2^\lambda\big) \end{aligned} \tag{46}$$

executing δ an expected number of times at most

$$\left(\frac{\lambda+1}{2}\right)^{\log(\ell)} \sum_{i\in[\ell]} \tau(\mathsf{core}_i) \tag{47}$$

where $\tau(\mathsf{core}_i)$ is the expected number of times core_i executes δ.

7.3.2 Constructing core_i

We construct core_i using the message-dependent sampler algorithm of Section 6. The message-dependent sampler is parameterized by a distribution Ω_i, meaning a set Ω_i and an algorithm to sample from it, as well as an event $E_i \subseteq \Omega_i$, and a 'clashing' function $\mathsf{clash} \colon E_i^\star \to 2^{\Omega_i}$. We define these as follows.

$$\begin{aligned} &\Omega_i \coloneqq V_i \times \mathsf{Result}\langle W', \mathsf{Error}\rangle \\ &E_i \coloneqq \{(\cdot, r) \in \Omega_i \mid r \in \mathsf{Ok}(W')\} \\ &\mathsf{clash}_i \colon E_i^\star \to 2^{\Omega_i} \\ &\mathsf{clash}_i((v_1,\cdot),\ldots,(v_\eta,\cdot)) \coloneqq \{(v,\cdot) \in \Omega_i \mid v \in \chi_i(v_1,\ldots,v_\eta)\} \end{aligned}$$

We let the core_i algorithm invoke the message-dependent sampler through a function invokeSampler parameterized by Ω_i, E_i, clash_i, as well as a function $\mathsf{sampleOmegaCond} \colon E_i^\star \to \Omega_i$. The signature of invokeSampler is

$$\mathsf{invokeSampler}_{\Omega_i,E_i} \colon (E_i^\star \to 2^{\Omega_i}) \to (E_i^\star \to \Omega_i) \to \mathsf{Result}\langle E_i^{k_i+1}, \mathsf{Error}\rangle$$

The function sampleOmegaCond allows the message-dependent sampler to sample from not only Ω_i but also from conditional distributions $\Omega_i \setminus \mathsf{clash}(e_1,\ldots,e_\eta)$ for $e_1,\ldots,e_\eta \in E_i$. To assist the message-dependent sampler we parameterize sampleOmegaCond by a sequence of zero or more E_i values to specify the conditional distribution.

The core_i algorithm operates by sampling u from U_i, defining $\mathsf{sampleOmegaCond}$, and then invoking the message-dependent sampler by calling $\mathsf{invokeSample}$. We write core_i in Algorithm 9.

Algorithm 9 core_i

```
$\mathsf{core}_i\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle U_i \times (V_i \times W')^{k_i+1}, \mathsf{Error}\rangle$
$\mathsf{core}_i(\eta)(x)(\mathcal{P}) :=$
    $\mathcal{P}' := \mathcal{P}.\mathsf{clone}()$
    $u \leftarrow U_i$
    $\mathsf{sampleOmegaCond}\colon E_i^\star \to \Omega_i$
    $\mathsf{sampleOmegaCond}((v_1, \cdot), \ldots (v_\eta, \cdot)) :=$
        $\mathcal{P} := \mathcal{P}'$
        $v \leftarrow V_i$
        **if** $v \in \chi_i(v_1, \ldots, v_\eta)$
            $\mathsf{sampleOmegaCond}((v_1, \cdot), \ldots, v_\eta)$
        $m := \mathsf{join}_i(u)(v)$
        $\mathcal{P}.\mathsf{update}(m)$
        $r \leftarrow \eta\big(\mathsf{instance}(x)(m)\big)(\mathcal{P})$
        $(v, r)$
    **match** $r \leftarrow \mathsf{invokeSampler}_{\Omega_i, E_i}(\mathsf{clash}_i)(\mathsf{sampleOmegaCond})$
        $\mathsf{Ok}((v_j, \mathsf{Ok}(w'_j))_{j\in[k_i+1]}) \Rightarrow \mathsf{Ok}((u, (v_j, w'_j)_{j\in[k_i+1]}))$
        $\mathsf{Err}(\mathsf{error}) \Rightarrow \mathsf{Err}(\mathsf{error})$
```

We capture the probability core_i succeeds in the following equations. We denote the random variable sampled from U_i by rvu. To indicate the input variables to core_i, on which several probabilities depend, we define the predicate ivs to accept the three input arguments in order. We justify the equations after stating them.

$$\begin{aligned}
&\Pr\Big[r \in \mathsf{Ok}\big(U_i \times (V_i \times W')^{k_i+1}\big) \;\Big|\; r \leftarrow \mathsf{core}_i(\eta)(x)(\mathcal{P})\Big] \\
&= \sum_{u \in U_i} \Pr\left[r \in \mathsf{Ok}\big(E_i^{k_i+1}\big) \;\middle|\; \begin{array}{l}\mathsf{rvu} = u,\ \mathsf{ivs}(\eta, x, \mathcal{P}) \\ r \leftarrow \mathsf{invokeSampler}_{\Omega_i,E_i}(\mathsf{clash}_i)(\mathsf{sampleOmegaCond})\end{array}\right] \\
&\quad \cdot \Pr\big[\mathsf{rvu} = u \;\big|\; u \leftarrow U_i\big] && (48) \\
&\geq \sum_{u \in U_i} \left(\Pr\left[\omega \in E_i \;\middle|\; \begin{array}{l}\mathsf{ivs}(\eta, x, \mathcal{P}),\ \mathsf{rvu} = u \\ \omega \in \mathsf{sampleOmegaCond}()\end{array}\right]\right. \\
&\quad \left. - \Pr\left[\omega \in \mathsf{clash}_i\big((v_1, \cdot), \ldots, (v_{k_i}, \cdot)\big) \;\middle|\; \begin{array}{l} v_1, \ldots, v_{k_i} \in V_i \\ \mathsf{ivs}(\eta, x, \mathcal{P}),\ \mathsf{rvu} = u \\ \omega \leftarrow \mathsf{sampleOmegaCond}()\end{array}\right]\right) \cdot \Pr\big[\mathsf{rvu} = u \;\big|\; u \leftarrow U_i\big] && (49) \\
&= \Pr\Big[r \in \mathsf{Ok}(W') \;\Big|\; r \leftarrow \delta(\eta)(x)(\mathcal{P})\Big] - \Pr\Big[v \in \chi_i(v_1, \ldots, v_{k_i}) \;\Big|\; v \leftarrow V_i,\ v_1, \ldots, v_{k_i} \in V_i\Big] && (50)
\end{aligned}$$

- Equation (48) holds by definition of core_i. We iterate over all $u \in U_i$ and multiply the probability of sampling u by the probability that $\mathsf{invokeSampler}$ succeeds and thus core_i succeeds given that particular u value and the input variables.

- We obtain Equation (49) by invoking Theorem 4 with distribution Ω_i, event E_i, and clashing function clash_i. By Theorem 4 the probability that $\mathsf{invokeSampler}$ succeeds is at least that probability of a sample from Ω_i occurring in E_i, minus the probability of a sample from Ω_i occurring in (clashing with) $\mathsf{clash}_i(e_1, \ldots, e_{k_i})$ for any $e_1, \ldots, e_{k_i} \in E_i$, which we may write as $\mathsf{clash}_i\big((v_1, \cdot), \ldots, (v_{k_i}, \cdot)\big)$ for any $v_1, \ldots, v_{k_i} \in V_i$. To sample from the non-conditional Ω_i distribution we sample from $\mathsf{sampleOmegaCond}$ with an empty sequence of E_i values.

- Equation (50) is had by distributing the probability of rvu taking a particular value, and analyzing the resulting two terms of the summand as follows.

– The first term in the summand becomes the probability a sample from Ω_i, given a fresh sample from U_i, occurs in E_i. That is the probability that sampling $u \leftarrow U_i$ and $v \leftarrow V_i$ to form $m := \mathsf{join}_i(u)(v)$, updating the prover with $\mathcal{P}.\mathsf{update}(m)$, and invoking η for result r, yields output $(v, r) \in E_i$ and thus $r \in \mathsf{Ok}(W')$. The probability that a sample from $\delta(\eta)(x)(\mathcal{P})$ occurs in $\mathsf{Ok}(W')$ is the same, because that is the probability that sampling m, updating the prover, and invoking η for result r, also yields output $r \in \mathsf{Ok}(W')$. Therefore the first term in the summand summed over U_i may be written as the probability of a sample from δ occurring in $\mathsf{Ok}(W')$.
– The second term in the summand becomes the probability a sample from Ω_i, given a fresh sample from U_i, occurs in $\mathsf{clash}_i\big((v_1, \cdot), \dots, (v_{k_i}, \cdot)\big)$ for any $v_1, \dots, v_{k_i} \in V_i$. By definition of clash_i, this is the probability a sample from V_i occurs in $\chi_i(v_1, \dots, v_{k_i})$. We similarly sum over U_i and write the new second term.

Regarding the expected number of times core_i executes δ, note that each execution of $\mathsf{sampleOmegaCond}$, that is each sample of Ω_i or a conditional distribution on Ω_i, corresponds to one execution of δ. By Theorem 4 the expected number of times Ω_i or a conditional distribution on Ω_i is sampled is at most $k_i + 1$, thus

$$\tau(\mathsf{core}_i) \le k_i + 1 \tag{51}$$

Lemma 5. *For some $\lambda \in \mathbb{N}$ we may complement verifier-move QUIRK functions* message, instance, *and* witness *with functions* extract *and* solve *with extractability error at most*

$$\max_{i \in [\ell]} \left\{ \Pr\left[v \in \chi_i(v_1, \dots, v_{k_i}) \,\middle|\, \begin{array}{l} v \leftarrow V_i \\ v_1, \dots, v_{k_i} \in V_i \end{array} \right] \right\} + \log(\ell) / \big(\lambda \cdot 2^\lambda\big)$$

and expected extraction and solution times at most

$$\left(\frac{\lambda + 1}{2}\right)^{\log(\ell)} \sum_{i \in [\ell]} (k_i + 1)$$

Proof. By construction of extract and solve we have

$$\begin{aligned} &\Pr\left[r \in \mathsf{Ok}(W) \,\middle|\, r \leftarrow \mathsf{extract}(\eta)(x)(\mathcal{P}) \right] + \Pr\left[r \in \mathsf{Ok}(S) \,\middle|\, r \leftarrow \mathsf{solve}(\eta)(x)(\mathcal{P}) \right] \\ &\quad = \Pr\left[r \in \mathsf{Ok}(\mathsf{Either}\langle W, S \rangle) \,\middle|\, r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P}) \right] \end{aligned}$$

By Equation (46) we have

$$\begin{aligned} &\Pr\left[r \in \mathsf{Ok}(\mathsf{Either}\langle W, S \rangle) \,\middle|\, r \leftarrow \mathsf{core}(\eta)(x)(\mathcal{P}) \right] \\ &\quad \ge \min_{i \in [\ell]} \left\{ \Pr\left[r \in \mathsf{Ok}\big(U_i \times (V_i \times W')^{k_i + 1}\big) \,\middle|\, r \leftarrow \mathsf{core}_i(\eta)(x)(\mathcal{P}) \right] \right\} - \log(\ell) / \big(\lambda \cdot 2^\lambda\big) \end{aligned}$$

By the series of equations ending with Equation (50) we have

$$\begin{aligned} &\Pr\left[r \in \mathsf{Ok}\big(U_i \times (V_i \times W')^{k_i + 1}\big) \,\middle|\, r \leftarrow \mathsf{core}_i(\eta)(x)(\mathcal{P}) \right] \\ &\quad \ge \Pr\left[r \in \mathsf{Ok}(W') \,\middle|\, r \leftarrow \delta(\eta)(x)(\mathcal{P}) \right] - \Pr\left[v \in \chi_i(v_1, \dots, v_{k_i}) \,\middle|\, \begin{array}{l} v \leftarrow V_i \\ v_1, \dots, v_{k_i} \in V_i \end{array} \right] \end{aligned}$$

Combining the three equations we get

$$\begin{aligned} &\Pr\left[r \in \mathsf{Ok}(W) \,\middle|\, r \leftarrow \mathsf{extract}(\eta)(x)(\mathcal{P}) \right] + \Pr\left[r \in \mathsf{Ok}(S) \,\middle|\, r \leftarrow \mathsf{solve}(\eta)(x)(\mathcal{P}) \right] \\ &\quad \ge \Pr\left[r \in \mathsf{Ok}(W') \,\middle|\, r \leftarrow \delta(\eta)(x)(\mathcal{P}) \right] \\ &\qquad - \max_{i \in [\ell]} \left\{ \Pr\left[v \in \chi_i(v_1, \dots, v_{k_i}) \,\middle|\, \begin{array}{l} v \leftarrow V_i \\ v_1, \dots, v_{k_i} \in V_i \end{array} \right] \right\} - \log(\ell) / \big(\lambda \cdot 2^\lambda\big) \end{aligned}$$

Regarding expected extraction and solution times, we must bound the expected numbers of times extract and solve invoke δ. By construction of extract and solve we see the answer reduces to the expected number of times core executes δ. By Equation (47) the function core executes δ an expected number of times at most

$$\left(\frac{\lambda+1}{2}\right)^{\log(\ell)} \sum_{i\in[\ell]} \tau(\mathsf{core}_i)$$

By Equation (51) the function core_i executes δ an expected number of times at most $k_i + 1$. Therefore we have total at most

$$\left(\frac{\lambda+1}{2}\right)^{\log(\ell)} \sum_{i\in[\ell]} (k_i + 1)$$

□

7.4 Verifier-move instance reductions

Given functions message, instance, and witness for a verifier-move QUIRK with message type M between two languages $L(I; W)$ and $L'(I'; W')$, and given the following parameters, we present how one may construct extract and solve functions to complete the QUIRK.

- A solution type S.
- Integer $k > 0$
- Monotonic function $\chi\colon M^\star \to 2^M$.
- Functions

  $$\begin{aligned}\mathsf{sol}\colon I \to (M \times W')^{k+1} \to \mathsf{Maybe}\langle S\rangle\\ \mathsf{wit}\colon I \to (M \times W') \to \mathsf{Maybe}\langle W\rangle\end{aligned}$$

  such that the following holds. Suppose $x \in I$ and $(m_i, w'_i)_{i\in[k+1]} \in (M \times W')^{k+1}$ such that it satisfies

  - Strict monotonicity:
    $$\forall i \in [k+1]\colon \chi(m_1, \ldots, m_{i-1}) \subset \chi(m_1, \ldots, m_i)$$
  - Validity:
    $$\forall i \in [k+1]\colon \big(\mathsf{instance}(x)(m_i)\ ;\ w'_i\big) \in L'$$

  Then it must hold that at least one of the two functions output a Yes variant, that is

  $$\mathsf{sol}(x)\big((m_i, w'_i)_{i\in[k+1]}\big) \in \mathsf{Yes}(S)\ \vee\ \mathsf{wit}(x)((m_1, w'_1)) \in \mathsf{Yes}(W)$$

To define extract and solve we borrow the function core_1 constructed in Section 7.3.2 which in our context has signature

$$\mathsf{core}_1\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle () \times (M \times W')^{k+1}, \mathsf{Error}\rangle$$

In Algorithm 10 we define solve and extract$'$ using core_1, while we construct extract without using core_1. We will explain the purposes of extract$'$ versus extract.

Algorithm 10 Functions for verifier-move instance reductions

```
$\mathsf{solve}\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle S, \mathsf{Error}\rangle$
$\mathsf{solve}(\eta)(x)(\mathcal{P}) :=$
    **match** $r \leftarrow \mathsf{core}_1(\eta)(x)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}\big(((), (m_i, w'_i)_{i\in[k+1]})\big) \Rightarrow$
            **match** $\mathsf{sol}(x)\big((m_i, w'_i)_{i\in[k+1]}\big)$
                $\mathsf{Yes}(s) \Rightarrow \mathsf{Ok}(s)$
                $\mathsf{No} \Rightarrow \mathsf{Err}(\mathsf{Fail})$

$\mathsf{extract}'\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle W, \mathsf{Error}\rangle$
$\mathsf{extract}'(\eta)(x)(\mathcal{P}) :=$
    **match** $r \leftarrow \mathsf{core}_1(\eta)(x)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}\big(((), (m_i, w'_i)_{i\in[k+1]})\big) \Rightarrow$
            **match** $\mathsf{wit}(x)\big((m_1, w'_1)\big)$
                $\mathsf{Yes}(w) \Rightarrow \mathsf{Ok}(w)$
                $\mathsf{No} \Rightarrow \mathsf{Err}(\mathsf{Fail})$
$\mathsf{extract}\colon (I' \to \mathsf{Pvr} \to \mathsf{Result}\langle W', \mathsf{Error}\rangle) \to I \to \mathsf{Pvr} \to \mathsf{Result}\langle W, \mathsf{Error}\rangle$
$\mathsf{extract}(\eta)(x)(\mathcal{P}) :=$
    $m \leftarrow M$
    $\mathcal{P}.\mathsf{update}(m)$
    **match** $r \leftarrow \eta\big(\mathsf{instance}(x)(m)\big)(\mathcal{P})$
        $\mathsf{Err}(\cdot) \Rightarrow r$
        $\mathsf{Ok}(w') \Rightarrow$
            **match** $\mathsf{wit}(x)((m, w'))$
                $\mathsf{Yes}(w) \Rightarrow \mathsf{Ok}(w)$
                $\mathsf{No} \Rightarrow \mathsf{Err}(\mathsf{Fail})$
```

Our algorithms for seeking extraction and solution are solve and extract. The purpose of extract′ is to aid in proving the extractability error of extract. We argue that

$$\Pr\left[r \in \mathsf{Ok}(W) \;\middle|\; r \leftarrow \mathsf{extract}(\eta)(x)(\mathcal{P})\right] \geq \Pr\left[r \in \mathsf{Ok}(W) \;\middle|\; r \leftarrow \mathsf{extract}'(\eta)(x)(\mathcal{P})\right]$$

Observe that extract executes the equivalent of δ and matches the return value. If δ returns an error, extract returns that. If δ returns a W' witness, extract attempts to return a W witness via wit. In contrast, extract′ executed core_1 but treats the return value similarly, returning it if it is an error, otherwise parsing it for (m_1, w'_1) and attempting to a return a W witness via wit. Therefore we are left to argue that execution of δ succeeds with probability no less than execution of core_1. By construction of core_1 and its message-dependent sampling subroutine, algorithm core_1 executes δ and if δ returns any error then core_1 returns that error. Therefore if core_1 succeeds so does δ. Moreover, the (m, w') pair that is fed to wit in both core_1 and δ is the same.

Lemma 6. *We may complement verifier-move QUIRK functions* message, instance, *and* witness *with functions* extract *and* solve *with extractability error at most*

$$\Pr\left[m \in \chi(m_1, \ldots, m_k) \;\middle|\; \begin{array}{l}(m, w') \leftarrow \delta \\ m_1, \ldots, m_k \in M\end{array}\right]$$

and expected extraction time 1 *and expected solution time at most* $k + 1$.

Proof. Regarding extractability error, it is clear by construction of $\mathsf{extract}'$ and solve that

$$\begin{aligned}&\Pr\left[r \in \mathsf{Ok}(W) \;\middle|\; r \leftarrow \mathsf{extract}'(\eta)(x)(\mathcal{P})\right] + \Pr\left[r \in \mathsf{Ok}(S) \;\middle|\; r \leftarrow \mathsf{solve}(\eta)(x)(\mathcal{P})\right]\\ &\quad = \Pr\left[r \in \mathsf{Ok}\big(() \times (M \times W')^{k+1}\big) \;\middle|\; r \leftarrow \mathsf{core}_1(\eta)(x)(\mathcal{P})\right]\end{aligned}$$

As established previously in the series of equations ending with Equation (50) we may write

$$\begin{aligned}&\Pr\left[r \in \mathsf{Ok}\big(() \times (M \times W')^{k+1}\big) \;\middle|\; r \leftarrow \mathsf{core}_1(\eta)(x)(\mathcal{P})\right]\\ &\quad \geq \Pr\left[r \in \mathsf{Ok}(W') \;\middle|\; r \leftarrow \delta(\eta)(x)(\mathcal{P})\right] - \Pr\left[m \in \chi(m_1, \ldots, m_k) \;\middle|\; \begin{matrix} m \leftarrow M \\ m_1, \ldots, m_k \in M \end{matrix}\right]\end{aligned}$$

Having argued that $\mathsf{extract}$ succeeds with probability no less than $\mathsf{extract}'$, we may combine the previous two equations replacing $\mathsf{extract}'$ with $\mathsf{extract}$ to establish the desired extractability error.

Regarding expected solution time, since core_1 is based on the message-dependent sampling algorithm of Section 6 with χ and k, we have expected solution time at most $k + 1$. Regarding expected extraction time, by construction $\mathsf{extract}$ executes the equivalent of δ exactly once. ☐

8 Examples

In Section 8.1 we construct a verifier-move instance reduction for checking identity between polynomials over a commutative ring. In Section 8.2 we utilize the identity checking QUIRK for a variation of the classic sumcheck protocol. In Section 8.3 we discuss three verifier-move witness reductions, the first two intended for discrete log based commitments, the last informal and for hash trees.

8.1 Univariate identity testing

Consider the following language $\mathsf{identity}$ where $\mathcal{R}$ is a commutative ring, $d \in \mathbb{N}$, and $p, q \in \mathcal{R}[X]$.

$$\begin{aligned}&\mathsf{identity}(\mathcal{R},\ d,\ p,\ q\ ;\ ()) \coloneqq\\ &\quad \deg(p) < d,\ \deg(q) < d\\ &\quad p = q\end{aligned}$$

We construct a verifier-move instance reduction Q from $\mathsf{identity}$ to the following language $\mathsf{randIdentity}$ where $c \in \mathcal{R}$.

$$\begin{aligned}&\mathsf{randIdentity}(\mathcal{R},\ d,\ p,\ q,\ c\ ;\ ()) \coloneqq\\ &\quad \deg(p) < d,\ \deg(q) < d\\ &\quad p(c) = q(c)\end{aligned}$$

Working in the commutative ring $\mathcal{R}$ we let $\mathsf{ZD}(\mathcal{R})$ denote the set of zero-divisors in $\mathcal{R}$. The verifier's message is a challenge $c \in \mathcal{R}$ thus we set the message type as $M \coloneqq \mathcal{R}$. We construct the message, instance, and witness functions for Q as follows.

Algorithm 11 Functions for verifier-move reduction to $\mathsf{randIdentity}$

```
$\mathsf{message}\colon () \to M$
$\mathsf{message}() \coloneqq c \leftarrow \mathcal{U}(\mathcal{R})$

$\mathsf{instance}\colon I \to M \to I'$
$\mathsf{instance}(x)(c) \coloneqq (\ldots x, b)$

$\mathsf{witness}\colon I \to M \to W \to \mathsf{Maybe}\langle W' \rangle$
$\mathsf{witness}(\cdot)(c)() \coloneqq \mathsf{Yes}(())$
```

For extractability we invoke the verifier-move instance reduction helper from Section 7.4 with the following parameters.

- Solution type $S \coloneqq \bot$.
- Integer $k \coloneqq d$.
- Monotonic function χ defined as

$$\begin{aligned} &\chi\colon \mathcal{R}^\star \to 2^{\mathcal{R}} \\ &\chi(r_1,\dots,r_\eta) \coloneqq \\ &\qquad \big\{ r \in \mathcal{R} \,\big|\, \exists i \in [\eta]\colon (r_i - r) \in \mathsf{ZD}(\mathcal{R}) \big\} \end{aligned}$$

- Functions wit and sol defined as

$$\begin{aligned} &\mathsf{sol}\colon I \to (M \times W')^{k+1} \to \mathsf{Maybe}\langle S \rangle \\ &\mathsf{sol}(\cdot)(\cdot) \coloneqq \mathsf{No} \\ \\ &\mathsf{wit}\colon I \to (M \times W') \to \mathsf{Maybe}\langle W \rangle \\ &\mathsf{wit}((\cdot,p,q))(\cdot) \coloneqq \\ &\qquad \textbf{match } p = q \\ &\qquad\qquad 0 \Rightarrow \mathsf{No} \\ &\qquad\qquad 1 \Rightarrow \mathsf{Yes}(()) \end{aligned}$$

We argue as required that given inputs $(r_i, w_i')_{i\in[k+1]}$ for sol with input (r_1, w_1') for wit, at least one of these two functions outputs a Yes variant. Clearly sol never outputs a Yes variant. To conclude that wit always outputs a Yes variant, recall that inputs $(m_i, w_i')_{i\in[k+1]}$ are promised to satisfy the validity and strict monotonicity assumptions. Validity means that for each r_i we have $p(r_i) = q(r_i)$. Strict monotonicity by definition of χ implies that for every pair r_i, r_j for $i,j \in [k+1]$ we have $(r_i - r_j) \notin \mathsf{ZD}(\mathcal{R})$. By standard algebra and induction on d one may argue if the polynomial $p - q \in \mathcal{R}[X]$ of degree less than d is non-zero, it has at most d roots in the set $\{r_i\}_{i\in[d+1]}$ given all differences in this set are regular elements. Since we have $d+1$ roots for $p - q$ it must be that $p = q$ and therefore wit may always return the Yes variant.

Lemma 7. *QUIRK Q from* identity *to* randIdentity *has completeness error* 0 *and extractability error at most*

$$\Pr\left[\exists i \in [d]\colon r - r_i \in \mathsf{ZD}(\mathcal{R}) \,\middle|\, \begin{matrix} r \leftarrow \mathcal{U}(\mathcal{R}) \\ r_1,\dots,r_d \in \mathcal{R} \end{matrix} \right]$$

Q has expected extraction time 1 *and expected solution time at most* $d+1$.

Proof. Completeness error follows by examination of the witness function. The claimed extractability error and expected extraction and solution times hold by Lemma 6. □

8.2 Evaluation check

We construct a poly-QUIRK to reduce between the two languages $\mathsf{evalCheck}_0$ and $\mathsf{evalCheck}_v$ for some $v \in \mathbb{N}$, written below. Witness types are the trivial unit type, and instance types involve the following. Let $\mathbb{F}$ be a finite field, $f \in \mathbb{F}[X_0,\dots,X_{v-1}]$, $d \in \mathbb{N}$, $t_0 \in \mathbb{F}$, $(B_i)_{i\in[v]-1} \in \mathbb{F}^v$, $(C_i)_{i\in[v]-1} \in \mathbb{F}^v$, and $(r_i, q_i)_{i\in[v]-1} \in \mathbb{F}[X]^2$. Each μ_i for $i \in [v]-1$ is the set of $|\mu_i|$ roots of unity in the multiplicative group. We define value $\Xi_{\mu_i}(B,b)$ for $b \in \mu_i$ and $B \in \mathbb{F}$ to be the (normalized) Lagrange basis monomial corresponding to b evaluated at B.

$$\begin{aligned}
&\mathsf{evalCheck}_0\big(\mathbb{F},\ f,\ d,\ t_0,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1}\ ;\ ()\big) \coloneqq \\
&\qquad t_0 = \sum_{b_0\in\mu_0} \Xi_{\mu_0}(B_0,b_0) \\
&\qquad\qquad \cdots \sum_{b_{v-1}\in\mu_{v-1}} \Xi_{\mu_{v-1}}(B_{v-1},b_{v-1})\cdot f(b_0,\dots,b_{v-1})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{evalCheck}_v\big(\mathbb{F},\ f,\ d,\ t_0,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[v]-1},\ (r_i,q_i)_{i\in[v]-1}\ ;\ ()\big) \coloneqq \\
&\qquad \forall i \in [v]-1: \\
&\qquad\qquad \deg(r_i) < |\mu_i|,\ \deg(q_i) < d - |\mu_i| \\
&\qquad\qquad t_i = r_i(B_i),\ t_{i+1} \coloneqq r_i(C_i) + (C_i^{|\mu_i|}-1)\cdot q_i(C_i) \\
&\qquad t_v = f(C_0,\dots,C_{v-1})
\end{aligned}$$

In order to reduce from $\mathsf{evalCheck}_0$ to $\mathsf{evalCheck}_v$ we use the following additional languages for $k \in [v+1]-1$ where $\mathsf{evalCheck}_k$ for $k=0$ and $k=v$ recovers languages $\mathsf{evalCheck}_0$ and $\mathsf{evalCheck}_v$.

$$\begin{aligned}
&\mathsf{eval}_k\big(\mathbb{F},\ f,\ t_k,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[k]-1}\ ;\ ()\big) \coloneqq \\
&\qquad t_k = \sum_{b_k\in\mu_k} \Xi_{\mu_k}(B_k,b_k) \\
&\qquad\qquad \cdots \sum_{b_{v-1}\in\mu_{v-1}} \Xi_{\mu_{v-1}}(B_{v-1},b_{v-1})\cdot f(C_0,\dots,C_{k-1},b_k,\dots,b_{v-1})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{evalCheck}_k\big(\mathbb{F},\ f,\ d,\ t_0,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[k]-1},\ (r_i,q_i)_{i\in[k]-1}\ ;\ ()\big) \coloneqq \\
&\qquad \forall i \in [k]-1: \\
&\qquad\qquad \deg(r_i) < |\mu_i|,\ \deg(q_i) < |\mu_i| - d \\
&\qquad\qquad t_i = r_i(B_i),\ t_{i+1} \coloneqq r_i(C_i) + (C_i^{|\mu_i|}-1)\cdot q_i(C_i) \\
&\qquad \mathsf{L}_k\big(f,\ t_k, (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[k]-1}\ ;\ ()\big)
\end{aligned}$$

In Section 8.2.1 we construct a reduction from $\mathsf{evalCheck}_k$ to another language $\mathsf{evalCheck}'_k$ for $k \in [v]-1$. In Section 8.2.2 we construct a reduction from $\mathsf{evalCheck}'_k$ to another language $\mathsf{evalCheck}''_k$ which is equivalent to $\mathsf{evalCheck}_{k+1}$ for $k \in [v]-1$. Composing these QUIRKs together into a poly-QUIRK we may reduce from $\mathsf{evalCheck}_0$ to $\mathsf{evalCheck}_v$.

8.2.1 A prover-move instance reduction

We construct a prover-move instance reduction from $\mathsf{evalCheck}_k$ to the following language $\mathsf{evalCheck}'_k$ for $k \in [v]-1$. Note the instance and witness types are the same as those of $\mathsf{evalCheck}_k$ with the instance additionally holding

$(r_k, q_k) \in \mathbb{F}[X] \times \mathbb{F}[X]$. This additional input is the prover's message and thus we assign $M \coloneqq \mathbb{F}[X] \times \mathbb{F}[X]$.

$$\begin{aligned}
&\mathsf{evalCheck}'_k\big(\mathbb{F},\ f,\ d,\ t_0,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[k]-1},\ (r_i,q_i)_{i\in[k]-1},\ (r_k,q_k)\ ;\ ()\big):\\
&\quad \forall i \in [k]-1:\\
&\qquad \deg(r_i) < |\mu_i|,\ \deg(q_i) < d - |\mu_i|\\
&\qquad t_i = r_i(B_i),\ t_{i+1} \coloneqq r_i(C_i) + (C_i^{|\mu_i|} - 1)\cdot q_i(C_i)\\
&\quad \deg(r_k) < |\mu_k|,\ \deg(q_k) < d - |\mu_k|\\
&\quad t_k = r_k(B_k) \qquad (52)\\
&\quad \mathsf{identity}\Big(\mathbb{F},\ d,\ r_k(X) + (X^{|\mu_k|}-1)\cdot q_k(X),\ \sum_{b_{k+1}\in\mu_{k+1}} \Xi_{\mu_{k+1}}(B_{k+1}, b_{k+1})\\
&\qquad \cdots \sum_{b_{v-1}\in\mu_{v-1}} \Xi_{\mu_{v-1}}(B_{v-1}, b_{v-1})\cdot f(C_0,\ldots,C_{k-1},X,b_{k+1},\ldots,b_{v-1})\ ;\ ()\Big)
\end{aligned}$$

We use Lemma 4 with the following message, instance, witness, and witOrSol functions. In message the prover computes r_k and q_k by dividing the polynomial in the last instance argument of identity by $(X^{|\mu_k|} - 1)$ to obtain quotient q_k and remainder r_k. Let I and W be the instance and witness types of evalCheck, and let I' and W' be the instance and witness types of evalCheck$'$. We'll use the never type for the solution type $S \coloneqq \bot$. The 'spreading' notation '$\ldots x$' in the instance function means unpacking the direct product element x into its components and inserting them into the new instance direct product element along with component (r_k, q_k).

Algorithm 12

```
message: $I \to W \to M$
message$(\cdot) \coloneqq (r_k, q_k)$

instance: $I \to M \to I'$
instance$(x)((r_k, q_k)) \coloneqq (\ldots x, (r_k, q_k))$

witness: $I \to M \to W' \to$ Maybe$\langle W\rangle$
witness$(\cdot)(\cdot)(\cdot) \coloneqq$ Yes$(())$

witOrSol: $I \to M \to W' \to$ Either$\langle W, S\rangle$
witOrSol$(\cdot)(\cdot)(\cdot) \coloneqq$ Yes$(())$
```

The fact witOrSol simply returns the unit means that whenever the output instance is valid then the input instance should be valid, as we now argue. With a valid output instance we may write the following to prove the input instance is valid.

$$\begin{aligned}
t_k &= r_k(B_k) &(53)\\
&= \sum_{b_k\in\mu_k} \Xi_{\mu_k}(B_k, b_k)\cdot r_k(b_k) &(54)\\
&= \sum_{b_k\in\mu_k} \Xi_{\mu_k}(B_k, b_k)\cdot \Big(r_k(b_k) + (b_k^{|\mu_k|} - 1)\cdot q_k(b_k)\Big)\\
&= \sum_{b_k\in\mu_k} \Xi_{\mu_k}(B_k, b_k) \sum_{b_{k+1}\in\mu_{k+1}} \Xi_{\mu_{k+1}}(B_{k+1}, b_{k+1})\\
&\quad \cdots \sum_{b_{v-1}\in\mu_{v-1}} \Xi_{\mu_{k+1}}(B_{v-1}, b_{v-1})\cdot f(C_0,\ldots,C_{k-1},b_k,b_{k+1},\ldots,b_{v-1})
\end{aligned}$$

where Equation (53) holds by Equation (52), and Equation (54) holds by the fact that with $\deg(r_k) < |\mu_k|$ evaluating r_k on the $|\mu_k|$ roots of unity and multiplying by the corresponding Lagrange monomials evaluated at B_k is equal to evaluating polynomial r_k at B_k.

Claim 31. *We may construct a QUIRK from from* $\mathsf{evalCheck}_k$ *to* $\mathsf{evalCheck}'_k$ *with functions* message, instance, *and* witness *having completeness error* 0, *extractability error* 0, *and expected extraction and solution times* 1.

Proof. Completeness holds by correctness of (r_k, q_k). Extractability error and expected extractability and solution times hold by Lemma 4 □

8.2.2 A verifier-move instance reduction

We construct a verifier-move QUIRK from $\mathsf{evalCheck}'_k$ to the following language $\mathsf{evalCheck}''_k$ for $k \in [v] - 1$.

$$\begin{aligned}
&\mathsf{evalCheck}''_k\big(\mathbb{F},\ f,\ d,\ t_0,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[k]-1},\ (r_i,q_i)_{i\in[k]-1},\ (r_k,q_k),\ C_k\ ;\ ()\big)\colon\\
&\quad \forall i \in [k]-1\colon\\
&\qquad \deg(r_i) < |\mu_i|,\ \deg(q_i) < |\mu_i| - d\\
&\qquad t_i = r_i(B_i),\ t_{i+1} \coloneqq r_i(C_i) + (C_i^{|\mu_i|} - 1)\cdot q_i(C_i)\\
&\quad \deg(r_k) < |\mu_k|,\ \deg(q_k) < |\mu_k| - d\\
&\quad t_k = r_k(B_k)\\
&\quad \mathsf{randIdentity}\Big(\mathbb{F},\ d,\ r_k(X) + (X^{|\mu_k|}-1)\cdot q_k(X),\ \sum_{b_{k+1}\in\mu_{k+1}} \Xi_{\mu_{k+1}}(B_{k+1}, b_{k+1})\\
&\qquad \cdots \sum_{b_{v-1}\in\mu_{v-1}} \Xi_{\mu_{v-1}}(B_{v-1}, b_{v-1})\cdot f(C_0,\ldots,C_{k-1},X,b_{k+1},\ldots,b_{v-1}),\ C_k\ ;\ ()\Big)
\end{aligned}$$

which we can rewrite as

$$\mathsf{evalCheck}_{k+1}\big(\mathbb{F},\ f,\ d,\ t_0,\ (\mu_i)_{i\in[v]-1},\ (B_i)_{i\in[v]-1},\ (C_i)_{i\in[k+1]-1},\ (r_i,q_i)_{i\in[k+1]-1}\ ;\ ()\big)$$

Our reduction is had by applying the univariate identity testing reduction of Section 8.1 from language identity to language randIdentity.

Claim 32. *We may reduce from* $\mathsf{evalCheck}'_k$ *to* $\mathsf{evalCheck}''_k$ *using the identity testing reduction of Section 8.1 with completeness* 0, *extractability error* $d/|\mathbb{F}|$, *expected extraction time* 1, *and expected solution time* $d+1$.

Proof. Suppose degree bound d is large enough to accommodate all polynomials tested. Expected extraction and solution times, as well as completeness error hold by Lemma 7. The extractability error given by Lemma 7 translates to

$$\Pr\left[\exists i \in [d]\colon r - r_i \in \mathsf{ZD}(\mathbb{F}) \;\middle|\; \begin{array}{l} r \leftarrow \mathcal{U}(\mathbb{F}) \\ r_1,\ldots,r_d \in \mathbb{F}\end{array}\right]$$

with $\mathbb{F}$ a field we have $\mathsf{ZD}(\mathbb{F}) = \{0\}$, thus the probability is that of r coinciding with any r_i for $i \in [d]$. With d possible collisions the probability is $d/|\mathbb{F}|$. □

8.3 Three examples of verifier-move witness reduction

We illustrate three example verifier-move witness reductions in the context of extracting from commitments. The first two are both intended for discrete log commitments, the third is informal and pertains to commitments by linear codes.

We set the context for the two discrete log based examples. Let $\mathbb{F}_p$ be a prime field, $\mathbb{G}$ a commutative group, and $\phi\colon \mathbb{F}_p^m \to \mathbb{G}$ a linear function for some commitment length $m \in \mathbb{N}$. We will invoke ϕ both by function calling and by interpreting it as a linear functional, multiplying by the row vector ϕ^T.

We illustrate an example of ϕ and $\mathbb{G}$ for univariate polynomial commitment by ϕ computing both a discrete log commitment opening on the input and a univariate evaluation of the input. The type $\mathbb{G}$ is a direct product of the claimed discrete log commitment and the claimed polynomial evaluation. Let $q \in \mathbb{Z}$ be a prime such that an elliptic curve $E(\mathbb{F}_q)$ over $\mathbb{F}_q$ has a subgroup of order p. Let $\mathbf{b} \in E(\mathbb{F}_q)^m$ be points in the subgroup to be used for Pedersen commitment. We write the group operations in additive form, but represent the output as x and y coordinates in $\mathbb{F}_q$.

The polynomial is over $\mathbb{F}_p$, and let $a \in \mathbb{F}_p$ be the evaluation point. Our group $\mathbb{G}$ consists of the two $\mathbb{F}_q$ coordinates for the commitment, along with the single $\mathbb{F}_p$ evaluation point. With $\mathbb{G} := \mathbb{F}_q^2 \times \mathbb{F}_p$ we define ϕ as

$$\begin{aligned}
&\phi \colon \mathbb{F}_p^m \to \mathbb{F}_q^2 \times \mathbb{F}_p \\
&\phi(\mathbf{f}) := \\
&\qquad (x_j, y_j) := \sum_{i \in [m]} \mathbf{b}_i \cdot \mathbf{f}_i, \;\; e_j := \sum_{i \in [m]} a^{i-1} \cdot \mathbf{f}_i \\
&\qquad (x_j, y_j, e_j)
\end{aligned}$$

Regardless the particular definition of ϕ, consider the following two languages $L(I; W)$ and $L'(I', W')$ where $I := \mathbb{G}^n$, $W := \mathbb{F}_p^{m \times n}$, $I' := \mathbb{G}^n \times \mathbb{F}_p^n$, and $W' := \mathbb{F}_p^m$.

$$\begin{aligned}
L(\mathbf{g}\;;\;\mathbf{A}) &:= \phi^T \cdot \mathbf{A} = \mathbf{g}^T \\
L'(\mathbf{g},\;\mathbf{c}\;;\;\mathbf{a}) &:= \phi^T \cdot \mathbf{a} = \mathbf{g}^T \cdot \mathbf{c}
\end{aligned}$$

We define the message, instance, and witness functions for a natural verifier-move QUIRK with message type $M := \mathbb{F}_p^n$ where the verifier sends a challenge $\mathbf{c} \in M$.

Algorithm 13 Functions for $\mathbb{G}$-based commitment reduction

```
message: $() \to M$
message$() := \mathbf{c} \leftarrow \mathcal{U}(\mathbb{F}_p^n)$

instance: $I \to M \to I'$
instance$((\mathbf{g}))(\mathbf{c}) := (\mathbf{g},\ \mathbf{c})$

witness: $I \to M \to W \to$ Maybe$\langle W' \rangle$
witness$(\cdot)(\mathbf{c})(\mathbf{A}) :=$
    Yes$(\mathbf{A} \cdot \mathbf{c})$
```

The three functions in Algorithm 13 feature completeness error 0 because the function witness always yields a new witness since $\phi^T \cdot \mathbf{A} \cdot \mathbf{c} = \mathbf{g}^T \cdot \mathbf{c}$ holds given $\phi^T \cdot \mathbf{A} = \mathbf{g}^T$. In Section 8.3.1 and Section 8.3.2 we complete the three functions by two contrasting applications of Lemma 5.

8.3.1 Extracting Pedersen commitments by linear independence

We will invoke Lemma 5 with the following parameters.

- Solution type $S := \bot$, that is the never type.
- Integer $\ell := 1$ and $k_1 := n - 1$.
- Types $U_1 := ()$ and $V_1 := M$ and $\mathsf{join}_1 \colon U_1 \to V_1 \to M$ defined as $\mathsf{join}_1()(v) := v$.
- Monotonic function χ_1 defined as mapping a seqence of vectors in $\mathbb{F}_p^n$ to their linear span, that is

$$\begin{aligned}
&\chi_1 \colon V_1^{\star} \to 2^{V_1} \\
&\chi_1\big(\mathbf{c}^{(1)}, \ldots, \mathbf{c}^{(\eta)}\big) := \left\{ \sum_{j=1}^{\eta} s_j \cdot \mathbf{c}^{(j)} \;\middle|\; \forall j \in [\eta] \colon s_j \in \mathbb{F}_p \right\}
\end{aligned}$$

The function χ_1 maps from all sequences of elements of $\mathbb{F}_p^n$ to all subsets of $\mathbb{F}_p$. Monotonicity of χ_1 as per Definition 13 is due to the linear span of some set of $j \geq 0$ vectors $\mathbf{c}^{(1)}, \ldots, \mathbf{c}^{(j)}$ containing the linear span of vectors $\mathbf{c}^{(1)}, \ldots, \mathbf{c}^{(j-1)}$.

- Function $\mathsf{witOrSol}$. It receives an input in I and an input in $U_1 \times (V_1 \times W')^{k_1+1}$ which is $() \times (M \times W')^n$.

$$\begin{aligned}
&\mathsf{witOrSol}\colon I \to () \times (M \times W')^n \to \mathsf{Either}\langle W, S\rangle \\
&\mathsf{witOrSol}(\cdot)\Big(\big((), (\mathbf{c}^{(j)}, \mathbf{a}^{(j)})_{j\in[n]}\big)\Big) \coloneqq \\
&\quad \mathbf{C} \coloneqq \left[\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(n)}\right],\ \mathbf{A}' \coloneqq \left[\mathbf{a}^{(1)}, \dots, \mathbf{a}^{(n)}\right] \\
&\quad \mathbf{A} \coloneqq \mathbf{A}' \cdot \mathbf{C}^{-1} \\
&\quad \mathsf{Left}(\mathbf{A})
\end{aligned}$$

We argue that $\mathsf{witOrSol}$ indeed fulfills its signature assuming the two promises of Lemma 5, those are

– Strict monotonicity:
$$\forall j \in [n]\colon\ \chi_1\big(\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(j-1)}\big) \subset \chi_1\big(\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(j)}\big)$$
By definition of χ_1 this means in the vector space $\mathbb{F}_p^n$ that $\mathbf{c}^{(1)}$ is outside the span of $\varnothing$ and is thus a non-zero vector, $\mathbf{c}^{(2)}$ is outside the span of $\mathbf{c}^{(1)}$, and $\mathbf{c}^{(3)}$ is outside the span of $\big(\mathbf{c}^{(1)}, \mathbf{c}^{(2)}\big)$, etc. Therefore $\big(\mathbf{c}^{(j)}\big)_{j\in[n]}$ is a list of n linearly independent vectors, enabling us to invert the matrix $\mathbf{C}$ formed with columns $\mathbf{c}^{(j)}$ for $j \in [n]$.

– Validity:
$$\begin{aligned}
&\forall j \in [n],\ \big(\mathsf{instance}((\mathbf{g}))\big(\mathsf{join}()\big(\mathbf{c}^{(j)}\big)\big)\ ;\ \mathbf{a}^{(j)}\big) \in L' \\
&\quad \Longrightarrow\ \big(\mathsf{instance}((\mathbf{g}))\big(\mathbf{c}^{(j)}\big)\ ;\ \mathbf{a}^{(j)}\big) = \big(\mathbf{g},\ \mathbf{c}^{(j)}\ ;\ \mathbf{a}^{(j)}\big) \in L' \\
&\quad \Longrightarrow\ \phi^T \cdot \mathbf{a}^{(j)} = \mathbf{g}^T \cdot \mathbf{c}^{(j)}
\end{aligned}$$

Thus we may write $\phi^T \cdot \mathbf{A}' = \mathbf{g}^T \cdot \mathbf{C}$, and with $\mathbf{C}$ invertible this becomes $\phi^T \cdot \mathbf{A} = \mathbf{g}^T$, thus $(\mathbf{g}\ ;\ \mathbf{A}) \in L$.

Claim 33. *Functions* $\mathsf{message}$, $\mathsf{instance}$, *and* $\mathsf{witness}$ *may be completed with* $\mathsf{extract}$ *and* solve *functions for a QUIRK with extractability error at most* $1/p$ *and expected extraction and solution times at most* n*.*

Proof. Given our parameters Lemma 5 constructs functions extract and solve that complete functions message, instance, and witness for a QUIRK with extractability error at most

$$\Pr\left[\mathbf{c} \in \chi_i\big(\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(k_1)}\big)\ \middle|\ \begin{array}{l}\mathbf{c} \leftarrow \mathcal{U}(\mathbb{F}_p^n) \\ \mathbf{c}^{(1)}, \dots, \mathbf{c}^{(k_1)} \in \mathbb{F}_p^n\end{array}\right] + \log(1)/(\lambda \cdot 2^\lambda)$$

for some $\lambda \in \mathbb{N}$, noting the second term disappears regardless of λ. With $k_1 \coloneqq n-1$, this is the probability that a uniformly sampled vector $\mathbf{c} \in \mathbb{F}_p^n$ occurs in the linear span of an $(n-1)$-dimensional subspace spanned by an arbitrary set of $n-1$ linearly independent vectors $\mathbf{c}^{(1)}, \dots, \mathbf{c}^{(n-1)} \in \mathbb{F}_p^n$. With p^{n-1} vectors in the subspace and p^n vectors in the n-dimensional vector space, the probability of a uniformly random vector occurring in the subspace is $p^{n-1}/p^n = 1/p$.

Lemma 5 yields expected solution and extraction times at most

$$\left(\frac{\lambda+1}{2}\right)^{\log(\ell)} \sum_{i\in[\ell]} (k_i + 1)$$

which reduces to n given $\ell = 1$, $k_i = n-1$, and again regardless of λ. □

8.3.2 Extracting Pedersen commitments independently

There are circumstances in extraction of homomorphic commitments by by linear independence as in Section 8.3.1 is an inconvenient option. In particular, when extracting for lattice based homomorphic commitments we wish to avoid inverting the challenge matrix $\mathbf{C}$ as done in Section 8.3.1 because an inverted matrix leads to large values and large norm. The smaller n is the smaller norm, so we wish to minimize n to 2. But we may still need to extract for many commitments. Our solution is to let n remain large and instead extract such that we are effectively inverting as if $n = 2$ by extracting for each commitment independently. A popular extraction technique in the literature for lattice based commitments is called the 'heavy row extractor' (e.g. see [BBC$^+$18]) and is inspired by the extraction technique of [Dam10] intended for the discrete log setting. The heavy row extractor also extracts for each commitment independently, but suffers worse extractability error and expected extraction time than we are able to achieve using Lemma 5. While the core lemma of the heavy row extractor can be tightened, it is not possible to reach the tight result we achieve using the basic idea underlying the heavy row extractor.

We will invoke Lemma 5 with the following parameters.

- Solution type $S := \bot$, that is the never type.
- Integer $\ell := n$ and $k_j := 1$ for $j \in [n]$.
- For $j \in [n]$ types $U_j := \mathbb{F}_p^{n-1}$ and $V_j := \mathbb{F}_p$ and $\mathsf{join}_j \colon U_j \to V_j \to M$ defined as
$$\mathsf{join}_j(\mathbf{u})(v) := (\mathbf{u}_1, \dots, \mathbf{u}_{j-1}, v, \mathbf{u}_{j+1}, \dots, \mathbf{u}_n)$$
- Monotonic function χ_j for $j \in [n]$ defined as mapping a sequence of V_j values, that is field elements, to their union.
$$\begin{aligned} &\chi_j \colon \mathbb{F}_p^\star \to 2^{\mathbb{F}_p} \\ &\chi_j\big(v_1, \dots, v_\eta\big) := \bigcup_{i \in [\eta]} v_i \end{aligned}$$
Monotonicity follows naturally by use of unionization.
- Function $\mathsf{witOrSol}$. It receives an input in I and an input in
$$\prod_{j \in [n]} U_j \times (V_j \times W')^{k_j+1} = \big(\mathbb{F}_p^{n-1} \times (\mathbb{F}_p \times W')^2\big)^n$$
We define $\mathsf{witOrSol}$ as
$$\begin{aligned} &\mathsf{witOrSol} \colon I \to \big(\mathbb{F}_p^{n-1} \times (\mathbb{F}_p \times W')^2\big)^n \to \mathsf{Either}\langle W, S \rangle \\ &\mathsf{witOrSol}(\cdot)\Big(\Big(\mathbf{u}^{(j)}, \big(v^{(1,j)}, \mathbf{b}^{(1,j)}\big), \big(v^{(2,j)}, \mathbf{b}^{(2,j)}\big)\Big)_{j \in [n]}\Big) := \\ &\quad \forall j \in [n] \colon \mathbf{a}^{(j)} := \big(\mathbf{b}^{(1,j)} - \mathbf{b}^{(2,j)}\big) / \big(v^{(1,j)} - v^{(2,j)}\big) \\ &\quad \mathbf{A} := \big[\mathbf{a}^{(1)}, \dots, \mathbf{a}^{(n)}\big] \\ &\quad \mathsf{Left}(\mathbf{A}) \end{aligned}$$
We argue that $\mathsf{witOrSol}$ indeed fulfills its signature assuming the two promises of Lemma 5, those are
  - Strict monotonicity:
$$\begin{aligned} &\forall j \in [n],\ \forall i \in [2] \colon\ \chi_j\big(v^{(1,j)}, \dots, v^{(j-1,j)}\big) \subset \chi_j\big(v^{(1,j)}, \dots, v^{(i,j)}\big) \\ &\implies \forall j \in [n] \colon \chi_j() \subset \chi_j\big(v^{(1,j)}\big) \subset \chi_j\big(v^{(1,j)}, v^{(2,j)}\big) \end{aligned}$$
By definition of χ_j this means $v^{(1,j)} \neq v^{(2,j)}$ allowing us to divide by the difference $v^{(1,j)} - v^{(2,j)}$.
  - Validity: For $j \in [n]$ and $i \in [2]$ define
$$\mathbf{c}^{(i,j)} := \mathsf{join}\big(\mathbf{u}^{(j)}\big)\big(v^{(i,j)}\big) = \Big(\mathbf{u}_1^{(j)}, \dots, \mathbf{u}_j^{(j)}, v^{(i,j)}, \mathbf{u}_{j+1}^{(j)}, \dots, \mathbf{u}_n^{(j)}\Big) \in \mathbb{F}_p^n$$
Then by validity
$$\begin{aligned} &\forall j \in [n],\ \forall i \in [2] \colon \big(\mathsf{instance}((\mathbf{g}))\big(\mathsf{join}\big(\mathbf{u}^{(j)}\big)\big(v^{(i,j)}\big)\big)\ ;\ \mathbf{b}^{(i,j)}\big) \in L' \\ &\quad \implies \big(\mathsf{instance}((\mathbf{g}))\big(\mathbf{c}^{(i,j)}\big)\ ;\ \mathbf{b}^{(i,j)}\big) \in L' \\ &\quad \implies \phi^T \cdot \mathbf{b}^{(i,j)} = \mathbf{g}^T \cdot \mathbf{c}^{(i,j)} \\ &\quad \implies \phi^T \cdot \frac{\mathbf{b}^{(1,j)} - \mathbf{b}^{(2,j)}}{v^{(1,j)} - v^{(2,j)}} = \mathbf{g}^T \cdot \frac{\mathbf{c}^{(1,j)} - \mathbf{c}^{(2,j)}}{v^{(1,j)} - v^{(2,j)}} \\ &\quad \implies \phi^T \cdot \mathbf{a}^{(j)} = \mathbf{g}_j \end{aligned}$$

where the last implication comes from multiplying $\mathbf{g}^T$ by the vector having zeros in all entries except having 1 in entry j. Thus we conclude $\phi^T \cdot \mathbf{A} = \mathbf{g}^T$ thus $(\mathbf{g}\ ;\ \mathbf{A}) \in L$.

Claim 34. *Functions* message, instance, *and* witness *may be completed with* extract *and* solve *functions for a QUIRK with extractability error at most*

$$1/p + \log(n)/\big(\lambda \cdot 2^\lambda\big)$$

for some parameter $\lambda \in \mathbb{N}$, and expected extraction and solution time at most

$$2n\left(\frac{\lambda+1}{2}\right)^{\log(n)}$$

In an asymptotic setting one would set λ as the security parameter and n as a constant.

Proof. Given our parameters Lemma 5 constructs functions extract and solve that complete functions message, instance, and witness for a QUIRK with extractability error and expected extraction and solution times as follows. Let $\lambda \in \mathbb{N}$ be some parameter that offers a tradeoff between extractability error and expected times. Lemma 5 promises extractability error at most

$$\max_{j\in[n]}\left\{ \Pr\left[v \in \chi_j(v_1) \,\middle|\, \begin{array}{c} v \leftarrow V_j \\ v_1 \in V_j \end{array} \right] \right\} + \log(n)/\big(\lambda \cdot 2^\lambda\big)$$

By definition of χ_j, the first term is the probability $v = v_1$, which is at most $1/p$ since v is sampled uniformly from $\mathbb{F}_p$. Lemma 5 promises expected extraction and solution times at most

$$\left(\frac{\lambda+1}{2}\right)^{\log(n)} \sum_{j\in[n]} (k_j + 1) = 2n\left(\frac{\lambda+1}{2}\right)^{\log(n)}$$

□

8.3.3 Hash trees

We informally present how Lemma 5 can be used to extract hash path openings spanning a particular fraction δ of a hash tree. This reduction is of particular importance to polynomial commitments based on linear codes, in which the prover commits to the encoding of the coefficient vector by hashing the codeword using a hash tree. The verifier must check that the prover has knowledge of openings for leaves that cover some fraction δ of the domain. Supposing the codeword length is n, we aim to extract openings for δn leaves, supposing for simplicity $\delta n \in \mathbb{N}$. We will argue for extractability error at most $(\delta n - 1)^s$ and expected time at most $2\delta n$.

Remark 3. *MDS codes can be used for commitments to unique polynomials by setting δ to the (fractional) unique decoding radius. If non-unique polynomial commitments are tolerable one can set δ to the largest distance for which a list-decoding algorithm is known or conjectured. One may even go further, beyond provable extractability, and in the extreme case set δ to the code rate.*

We start with the language in which an instance is a hash tree root r, and the witness is at least δn leaf openings. Full completeness is conditioned on instantiating the prover with not just δn leaf openings but all n leaf openings. The witness only asks for δn leaf openings, however, because that is all we will extract. The natural verifier-move reduction from this language is to uniformly sample $s \in \mathbb{N}$ leaves by sampling $\mathbf{s} \leftarrow \mathcal{U}([n]^s)$ (thus $M := [n]^s$), reducing to a language in which the instance consists of both the root r and the sampled indices vector $\mathbf{s}$, while the witness consists of the s opening paths to the leaves indicated by $\mathbf{s}$. For simplicity we sample with replacement. For completeness, clearly a prover instantiated with all n openings paths is able to trim the witness to the s random opening paths.

For extractability we use Lemma 5 with $S := \perp$, $\ell := 1$, $U_1 := ()$, $V_1 := M = [n]^s$, and $k_1 := \delta n - 1$. The idea is that χ_1 gather all distinct leaves located in the input sequence vectors into a set S, representing the set of all leaves already extracted. The output of χ_1 is then all vectors which don't have any leaves outside S. We write χ_1 in Algorithm 14, though we are not able to use it as written.

Algorithm 14 χ_1 for hash tree

```
$\chi_1 \colon ([n]^s)^\star \to 2^{([n]^s)}$
$\chi_1(\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(\eta)}) :=$
    $S := \bigcup_{i \in [\eta]} \bigcup_{j \in [n]} \mathbf{s}_j^{(i)}$
    $\{\mathbf{s} \in [n]^s \mid \forall i \in [n] \colon\ \mathbf{s}_i \in S\}$
```

The issue with χ_1 is that it could have an input sequence of length $\eta := k_1 + 1$ such that each input vector $\mathbf{s}^{(i)}$ only contributes one 'new' leaf to S ('new' meaning not present in a vector $\mathbf{s}^{(j)}$ for $j < i$). In this case, when $\eta := k_1 + 1$ we would have $|S| = k_1 + 1$ and needing $|S| = \delta n$ translates to $k_1 := \delta n - 1$. But it could also be the case, as is likely with an honest prover, that each input vector contributes many new leaves to S. As η gets larger (ultimately reaching $k_1 + 1$), $|S|$ gets larger and thus the output set gets larger, increasing the probability of 'clashing'. It may well be that with an $\eta < k_1 + 1$ we have $|S| = n$ in which case it would be impossible to extract $k_1 + 1$ vectors that satisfy the 'strict monotonicity' property promised by Lemma 5. Our solution is to set $k_1 := \delta n - 1$, but we observe the message-dependent sampling algorithm as it executes, and when we observe an invocation of χ_1 with $|S| \geq \delta n$ we exit the message-dependent sampling algorithm early, even if less than $k_1 + 1$ vectors have been extracted (despite $k_1 + 1$ of them promised).

Given the above strategy of exiting early once $|S| \geq \delta n$, we are able to extract opening paths for δn distinct leaves. Our witOrSol function simply outputs these leaves with their paths as the witness. Early exits don't increase the expected time of the message-dependent sampling algorithm. Regarding probability of success, we may calculate the extractability error of the message-dependent sampling algorithm using χ_1 and $k_1 := \delta n - 1$, but with the assumption that $|S| < \delta n$ due to having exited early if $|S| \geq \delta n$. In that case Lemma 5 yields extractability error at most

$$\Pr\left[\mathbf{c} \in \chi_1\big(\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(k_1)}\big) \;\middle|\; \begin{array}{l} \mathbf{s} \leftarrow \mathcal{U}([n]^s) \\ \mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(k_1)} \in [n]^s \end{array}\right] + \log(1)/(\lambda \cdot 2^\lambda)$$

for some $\lambda \in \mathbb{N}$, noting the second term disappears. By the definition of χ_1, this is the probability that a uniformly sampled $\mathbf{s} \in [n]^s$ have all entries occurring in S. Assuming $|S| \leq \delta n - 1$ this probability is at most $(\delta n - 1)^s$. Regarding expected time, Lemma 5 promises at most $k_1 + 1 = \delta n$.

A Founding papers of interactive proofs

Rather than dwelling on the rich history of interactive proofs in the introduction, we choose to develop here in the Appendix an exploration of the two founding papers of interactive proofs. Interactive proofs are both the foundation for all other proof system models following, and the proof system model most closely related to the QUIRK model.

Two works are usually regarded as the primary founding papers of (probabilistic) proof systems. The first paper is by Goldwasser, Micali, and Rackoff [GMR85] and doesn't have a precise publishing date because early versions of the paper circulated as early as 1982, but it was rejected from major journals three times before being accepted into *STOC'85* more than a decade later [Gol02]. The second paper is by Babai [Bab85] and appeared in 1985 and was in fact presented at the same conference (*STOC'85*) as [GMR85]. The two works were independently created, but given their simultaneous publishing and related topics each work cited the other.

We note that [GMR85] was also the founding paper of the concept of zero-knowledge, more generally knowledge complexity. After introducing interacting proofs, [GMR85] took a natural next step and considered how much knowledge a proving process reveals to the verifier beyond the validity of the statement. The concept of knowledge complexity, however, can be examined separately and we do not examine it here.

The two works independently define their own basic tools, proof systems employing these tools, and complexity classes describing languages in terms of these proof systems. The resulting proof systems and complexity classes carry both similarities and differences. A subsequent paper [GS86] proved the complexity classes are in fact equivalent.

A.1 The framework of GMR85

A.1.1 Basic Tools

The tools defined in [GMR85] are pairs of interactive Turing machines. Both machines read the same input tape. Each machine has two private tapes, a random tape for reading, and a work tape for reading and writing. Additionally there are two tapes for communication between the machines. Each communication tape facilitates communication in one direction by one machine writing to it and the other reading from it. They contrast this model of communicating Turing machines with the model for the traditional NP proving process in which the random tapes are nonexistent and there is only one communication tape from the prover to the verifier.

A.1.2 Proof systems

To convert a pair of interactive Turing machines into a proof system for a language L, [GMR85] treats one machine as the prover and the other as the verifier and enforces the following conditions.

- Unbounded prover The prover has unbounded computational resources.
- Bounded verifier The verifier is bounded by polynomial time.
- Completeness For every $x \in L$, the verifier halts and accepts with probability at least $1 - 1/p(|x|)$ for all polynomials p.
- Soundness For every $x \notin L$, the verifier halts and accepts with probability at most $1/p(|x|)$ for all polynomials p.

In this case we say L *has* a proof system.

A.1.3 Complexity classes

With a definition of proof systems in hand, [GMR85] defines induced complexity classes. They define the complexity class *Interactive Polynomial-time*, denoted IP, as containing all languages that have a proof system as previously defined. Any such proof system is referred to as an interactive proof system. The authors believe the number of messages exchanged between the two machines, capturing the amount of interaction required, is the primary metric for efficiency. In light of this they further partition IP into subclasses according to how the amount of interaction grows with the input size. The class $\mathcal{IP}[t(\cdot)]$ for $t \colon \mathbb{N} \to \mathbb{N}$ comprises languages having a proof system with the exchange of $t(|x|)$ messages on instance x. To make this precise, [GMR85] assumes the verifier sends the first message.

A.2 The framework of Bab85

A.2.1 Basic Tools

The tools defined in [Bab85] are *Arthur vs. Merlin games* or Arthur-Merlin games. Arthur-Merlin games are a special case of *Games against Nature* from [Pap84]. A Game against Nature, in turn, is a special case of a more general game involving two players on a common input that sequentially exchange an prespecified number of messages, and at the end a deterministic polynomial time Turing machine views all moves taken and declares a winner. A Game against Nature is when one of the players, called Nature, is indifferent to winning and on each move simply sends a random message. An Arthur-Merlin game is a Game against Nature in which Arthur plays the role of Nature and for any input the probability of Merlin winning is bounded away from one-half in one direction or the other (e.g. probability less than $1/3$ or greater than $2/3$). The purpose of the latter condition will become apparent when using Arthur-Merlin games as proof systems.

A.2.2 Proof systems

While Arthur-Merlin games may seem irrelevant to proof systems they may in fact serve as proof systems. The prover can be identified with the powerful wizard Merlin, appropriately so in that Arthur-Merlin games impose no computational restrictions on Merlin. The verifier can be identified as the machine that makes the random moves of Arthur and then invokes the deciding Turing machine at the end. Thus the verifier is restricted to only asking random questions and once all answers are received to making a decision in deterministic polynomial time. If we invoke this proof system on a language containing exactly those strings on which Merlin wins with probability more than half, then the final condition of Arthur-Merlin games promises notions of completeness and soundness analogous to those of [GMR85]. In order to guarantee completeness and soundness, Arthur-Merlin proof systems are appropriate for exactly such languages. It may seem backwards that we first define Arthur-Merlin proof systems and then seek appropriate languages, but this is an artifact of how [Bab85] organizes definitions. In practice we first define languages and then seek appropriate Arthur-Merlin proof systems. Either way, the same pairs of languages and proof systems exist.

A.2.3 Complexity classes

With Arthur-Merlin games in hand, [Bab85] defines induced complexity classes. Every Arthur-Merlin game induces a language consisting of all inputs for which Merlin succeeds with probability above one half. Thus we may define sets of languages, that is complexity classes, by defining sets of Arthur-Merlin games. Suppose we partition Arthur-Merlin games by which player moves first and also by how many moves take place. Let $t\colon \mathbb{N} \to \mathbb{N}$ be a function of game input lengths. For games where Arthur moves first and there are $t(\cdot)$ moves, denote the corresponding complexity class by $\mathcal{AM}[t(\cdot)]$. For games where Merlin moves first and there are $t(\cdot)$ moves, denote the corresponding complexity class by $\mathcal{MA}[t(\cdot)]$. In the case that there are a constant number of moves, we may omit the brackets and instead expand the capital string to indicate the sequence of moves. For example, the complexity classes induced by single-move games may be denoted A and M, two-move games AM and MA, three-move games AMA and MAM, etc.

A.3 Similarities

Both forms of (probabilistic) proof systems rely crucially on *interaction*, in particular non-trivial interaction beyond the prover simply sending the verifier a proof. The interaction of [GMR85] takes place via interacting Turing machines, while the interaction of [Bab85] takes place via a game and a sequence of moves between two parties. All proof systems that have followed these works (including MIPs, PCPs, NIZKs) still employ some form of non-trivial interaction, even if the verifier only interacts once with the prover.

Both forms of proof systems employ a randomized verifier. In contrast, verifiers for the NP proof system are deterministic. The need for verifier randomization is equally important to the need for verifier interaction. In fact, in most proof systems including all those explored in this project, the randomization and interaction of the verifier can be thought of as the same. They are the same in that all randomness is only used in interaction, and all interaction only uses randomness. This is how verifiers function in the proof systems of [Bab85], whereas verifiers function more freely in the proof systems of [GMR85].

In both forms of proof systems the prover has unbounded computational resources and the verifier is restricted to probabilistic polynomial time (and [Bab85] further restricts exactly how the randomness is used). It seems both papers arrived at this asymmetric formulation for the same reasons. If the prover were restricted, the prover could not prove many statements otherwise provable. If the verifier were unrestricted, the verifier could prove many statements to itself with no need for a prover. Both works intend to capture the theoretical power of interactive proof systems with no intention for practicality. Works that follow, such as this project, narrow focus to proof systems with probabilistic polynomial time provers.

A.4 Differences

The primary difference between the proof systems of [Bab85] and [GMR85] arises due to the already mentioned difference of how [Bab85] restricts the verifier's use of randomness. The verifier of [GMR85] has a private random tape and may thus hide randomness from the prover. The verifier of [Bab85], on the other hand, only may use the randomness of Arthur all of which is sent to the prover. A useful analogy is drawn by [Bab85] wherein interactive proofs from [GMR85] are card games (in which cards may be private) and interactive proofs from [Bab85] are chess games (in which all moves are public). Interactive proofs with private randomness are qualified as 'private coin' whereas those with public randomness are qualified as 'public coin'.

The notation of complexity classes also differs between the two works. The two notations $\mathcal{IP}[t(\cdot)]$ and $\mathcal{AM}[t(\cdot)]$ (and $\mathcal{MA}[t(\cdot)]$) are similar in that they both involve brackets indicating the number of messages exchanged. But when we omit the brackets they suddenly carry contrasting meanings. IP captures languages with interactive proofs of any polynomial number of messages, while AM (and MA) captures languages with interactive proofs of only two messages. Thus with brackets omitted IP allows a maximum number of bidirectional messages, while AM (and MA) allows a minimum number of bidirectional messages.

We also mention that the completeness and soundness probabilities are presented differently but they are effectively equivalent. The error probabilities in [GMR85] are required to be negligible, whereas the error probabilities in [GMR85] may not be negligible but they are formulated such that they will become negligible upon repeating the protocol any non-constant polynomial number of times.

A.5 Analysis and equivalence

The complexity classes formulated by [Bab85] are more amenable to analysis than those of [GMR85] due to the public coin restriction [Bab85] imposes on the verifier. While [GMR85] provides no analysis of the complexity classes $\mathcal{IP}[\cdot]$, [Bab85] makes several trivial and nontrivial observations about $\mathcal{AM}[\cdot]$ and $\mathcal{MA}[\cdot]$. Many papers followed making further observations about these classes and their relationships to each other and to other classes.

Trivially, [Bab85] notes A = BPP and M = NP. To see why A = BPP, see that Arthur tosses coins and Merlin sends nothing, so a deterministic polynomial time machine is left to decide membership only with the help of Arthur's coins. To see why M = NP, see that Merlin sends the equivalent of an NP witness and Arthur sends nothing, so a deterministic polynomial time machine is left to decide membership only with the help of Merlin's witness. It is also trivial to note the inclusion relation relative to the number of messages exchanged which holds for the IP classes as well. In particular, if a language has a proof system with k messages exchanged then it also has a proof system with $k' > k$messages exchanged where the additional messages are empty. Extending the inclusion relation by concatenating moves also at the beginning of the game we have

$$\mathcal{AM}[k] \cup \mathcal{MA}[k] \subseteq \mathcal{AM}[k+1] \cap \mathcal{MA}[k+1] \tag{55}$$

Non-trivially, [Bab85] proves that an MA game can be simulated by an AM game with only a polynomial increase in the total size of messages sent, and thus any constant length game can be simulated by a single AM game. Reversing the moves of an MA game to obtain an AM game lets Merlin adaptively choose his message after learning the randomness of Arthur which is problematic for soundness. To overcome this bias in Merlin's message, [Bab85] plays the AM game many times in parallel requiring Merlin to reply the same to each. The result is a single AM game with polynomially larger messages. One may then take any Arthur-Merlin game and make a constant number of MA to AM swaps and then merge adjacent A's and M's all while maintaining polynomial message sizes. Consequently all finite levels of the hierarchy above AM collapse down to AM, that is for any constant k

$$\mathcal{AM}[k] = \mathcal{AM}[2] \tag{56}$$

Unbounded levels of the hierarchy specified by any non-constant polynomial, however, are not known to collapse.

Following [GMR85] and [Bab85], [GS86] proves that private coin interactive proofs can be simulated by public coin interactive proofs with the same number of messages but with one additional message sent from Merlin to Arthur at the beginning. In particular for any polynomial t they prove

$$\mathcal{IP}[t(\cdot)] \subseteq \mathcal{MA}[t(\cdot)+1] \subseteq \mathcal{AM}[t(\cdot)+2] \tag{57}$$

Combined with the collapsing theorem of [Bab85] one may collapse the additional two messages of the AM game into the polynomial t. Combining further with the trivial fact that proof systems from [Bab85] are special cases of proof systems from [GMR85] we obtain the equality

$$\mathcal{IP}[t(\cdot)] = \mathcal{AM}[t(\cdot)] \tag{58}$$

Thus the two forms of proof systems have equal power for proving statements, and private verifier randomness provides no additional power over public verifier randomness. We warn, however, that this does not immediately imply equivalence with respect to any feature of proof systems (e.g. zero-knowledge).

The power of interactive proofs was well underestimated in these early days. An early extended abstract of [GMR85] conjectured that AM is a strict subset of $\mathcal{IP}[2]$, and that the IP hierarchy doesn't collapse, that is $\mathcal{IP}[k]$ is a strict subset of $\mathcal{IP}[k+1]$. Given the above equivalence and collapsing theorem, both of these conjectures turned out to be false as $\mathcal{AM} = \mathcal{IP}[2]$ and the finite IP hierarchy collapses. On the other hand, [Bab85] describes the family of languages having Arthur-Merlin proof systems as ”just above NP” believing it is not much larger than NP, and such belief was not unique to [Bab85]. The Arthur-Merlin games were inspired by the Games against Nature of [Pap84] with the added constraint that winning and losing probabilities against Nature are bounded away from one-half. Though [Pap84] showed Games against Nature characterize PSPACE, [Bab85] believed Arthur-Merlin games characterize a much smaller complexity class. In particular, [Bab85] believed that even coNP could not exist within $\mathcal{AM}[\mathsf{poly}(\cdot)]$, but later [LFKN92] proved to the contrary that $co\mathcal{NP} \subseteq \mathcal{IP} = \mathcal{AM}[\mathsf{poly}(\cdot)]$. Ultimately it was proven by [Sha92] that interactive proof systems characterize PSPACE. In other words, languages with interactive proof systems are equivalent to languages computable in polynomial space, that is

$$\mathcal{PSPACE} = \mathcal{IP} = \mathcal{AM}[\mathsf{poly}(\cdot)] \tag{59}$$

This result highlights how interactive proof systems are believed to be much more powerful than NP proof systems.

A.6 Motivations and results

The founding works [GMR85] and [Bab85] each arrive at interactive proof systems by their own motivations and each arrive at their own corresponding results. In [GMR85] attention is drawn to interactive proof systems because they seem efficient by intuition and because they offer the ability to prove statements in zero-knowledge in contrast to previous nontrivial proving systems. They construct zero-knowledge interactive proofs for quadratic residuosity and quadratic nonresiduosity. If it were not for these zero-knowledge proofs the paper would not have presented anything about interactive proof systems provably superior to previous proof systems. We also note that the zero-knowledge proofs they present rely on a non-deterministic prover and thus do not suffice for efficient protocols. In [Bab85] attention is drawn to interactive proof systems because they allow proving certain statements from group theory either not known to be provable in the NP proof system or rather complex to prove in the NP proof system. In particular, the problems are deciding membership and non-membership in a group and deciding order and non-order of a group when groups are matrix groups represented by generating sets. A particular version of the membership problem (4 by 4 integral matrices) is even known to be undecidable, and yet [Bab85] shows it may be proven probabilistically via interactive proofs.

Each work also comments philosophically on the significance of interactive proofs. The authors of [GMR85] emphasize the interactivity, contrasting how NP proofs are analogous to those that can be ”written down in a book,” whereas interactive proofs are analogous to those that can be ”explained in class.” A proof written down in a book must answer all possible questions, whereas a proof explained in class must only answer questions as they arise. The author of [Bab85] emphasizes randomness, saying ”a random string can sometimes replace the most formidable mathematical hypothesis” and recalling how randomness often radically simplifies solutions, the classic example being primality testing. The philosophy of [Bab85] is contained in its title, ”Trading group theory for randomness,” referring to a trade off that statements not easily proven via group theory can be easily proven via randomness if one is willing to trade complete soundness for statistical soundness. Lastly, we note that the class AM has evidence for being a natural class. As [Bab85] points out, one may prove AM relates to NP in the same naturally relativized way that BPP relates to P. In particular, for a random oracle O (meaning for almost every oracle O in a measure-theoretic sense)

$$\mathcal{BPP} = \mathcal{P}^O \text{ and } \mathcal{AM} = \mathcal{NP}^O \tag{60}$$

Thus AM might be thought of as a probabilistic version of NP in the same way BPP is a probabilistic version of P. At the same time, MA might also be thought of as a probabilistic version of NP in which the NP verifier may use randomness. Perhaps both AM and MA are indeed deserving of the name ”just above NP.”

References

[AFR23] Thomas Attema, Serge Fehr, and Nicolas Resch. Generalized special-sound interactive proofs and their knowledge soundness. Cryptology ePrint Archive, Paper 2023/818, 2023. https://eprint.iacr.org/2023/818.

[Bab85] L. Babai. Trading group theory for randomness. In *STOC '85*, 1985.

[BBC$^+$18] Carsten Baum, Jonathan Bootle, Andrea Cerulli, Rafael del Pino, Jens Groth, and Vadim Lyubashevsky. Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits. Cryptology ePrint Archive, Paper 2018/560, 2018. https://eprint.iacr.org/2018/560.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Paper 2016/116, 2016. https://eprint.iacr.org/2016/116.

[Dam10] I. Damgård. On sigma-protocols. 2010.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *STOC '85*, 1985.

[Gol02] Oded Goldreich. Zero-knowledge twenty years after its invention. *IACR Cryptol. ePrint Arch.*, 2002:186, 2002.

[GS86] S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. *Adv. Comput. Res.*, 5:73–90, 1986.

[KP22] Abhiram Kothapalli and Bryan Parno. Algebraic reductions of knowledge. Cryptology ePrint Archive, Paper 2022/009, 2022. https://eprint.iacr.org/2022/009.

[LFKN92] Carsten Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39:859–868, 1992.

[Pap84] C. Papadimitriou. Games against nature. *Journal of Computer and System Sciences*, 31:288–301, 1984.

[Sha92] A. Shamir. Ip = pspace. *J. ACM*, 39:869–877, 1992.