1 GKR Protocol and Its Efficient Implementation

1.1 Intuition [OpenCourseware, 2024]

Given a circuit C of depth D and size S an input $x \in \{0,1\}^n$ and an output y the prover needs to convince the verifier that $C(x) = y$ In other words, the verifier wants to catch the prover if y is incorrect. Here is a simple idea:

The verifier will ask the prover for the value of the two children corresponding to the output gate, and will check consistency with y Note that if the values are consistent and y is false, then the value of at least one of its children must be false. The verifier will guess which one is false randomly, and will continue this process until a leaf x_i is reached. Note that if y is false, and every time the verifier guesses correctly who the false child is, then at the end of this protocol the verifier will receive a false value of x_i and thus the prover will be rejected!

An illustrative example:

```python
from random import choice
import logging

logging.basicConfig(level=logging.DEBUG)

class Gate:
    def __init__(self, op, value=None, left=None, right=None):
        self.op = op  # 'AND', 'OR', 'NOT', or 'INPUT'
        self.value = value
        self.left = left
        self.right = right

def evaluate_gate(gate):
    if gate.op == 'INPUT':
        return gate.value
    elif gate.op == 'NOT':
        return not evaluate_gate(gate.left)
    elif gate.op == 'AND':
        return evaluate_gate(gate.left) and evaluate_gate(gate.right)
    elif gate.op == 'OR':
        return evaluate_gate(gate.left) or evaluate_gate(gate.right)

def verify_circuit(gate, claimed_value):
    current = gate
    logging.debug(f"Starting verification at gate type {current.op}")
```

```
    while current.op != 'INPUT':
        actual = evaluate_gate(current)
        logging.debug(f"Gate {current.op}: claimed={current.value},
        ↪  actual={actual}")

        if current.value != actual:
            logging.debug("Detected inconsistency!")
            return False

        # Random path checking
        if current.op in ['AND', 'OR']:
            current = choice([current.left, current.right])
            logging.debug(f"Randomly choosing child gate: {current.op}")
        else:  # NOT gate
            current = current.left

    logging.debug(f"Reached input gate with value: {current.value}")
    return current.value == claimed_value

# Example circuit: (NOT x1 AND x2) OR x3
x1 = Gate('INPUT', True)
x2 = Gate('INPUT', False)
x3 = Gate('INPUT', True)

not_x1 = Gate('NOT', False, x1)
and_gate = Gate('AND', False, not_x1, x2)
or_gate = Gate('OR', True, and_gate, x3)

print("Testing incorrect claim:")
or_gate.value = False  # False claim
result = verify_circuit(or_gate, False)
print(f"Verification result (false claim): {result}\n")

print("Testing correct claim:")
or_gate.value = True  # Correct claim
result = verify_circuit(or_gate, True)
print(f"Verification result (correct claim): {result}")
```

This code demonstrates a key intuition behind the GKR protocol:

1. It shows verifier-prover interaction where:
    - Prover makes a claim about circuit output (here, falsely claiming 1)
    - Verifier performs random checks (random path selection)

- False claims are caught with probability >= 0.5

2. Key GKR concepts illustrated:
   - Random path checking (rather than checking entire circuit)
   - Probability-based verification
   - Soundness (false proofs get caught)

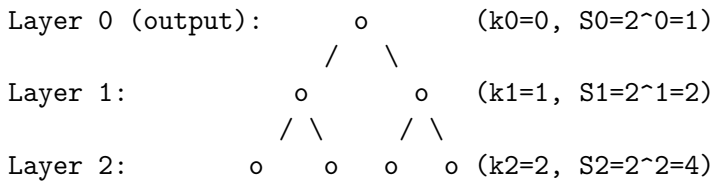
However, this is highly simplified. Real GKR protocol:

- Uses sum-check protocol
- Handles multivariate polynomials
- Achieves much higher soundness guarantees
- Works with arithmetic circuits

1.2 Detailed description of GKR protocol [OpenCourseware, 2024]

1.2.1 Notation

Suppose we are given a layered arithmetic circuit C (over $GF[2]$ of size S depth d and fan-in two (C may have more than one output gate). Number the layers from 0 to d, with 0 being the output layer and d being the input layer. Let S_i denote the number of gates at layer i of the circuit C. Assume S_i is a power of 2 and let $S_i = 2^{k_i}$ The GKR protocol makes use of several functions, each of which encodes certain information about the circuit. [Thaler, 2013]



- Output: Top of the circuit (single `o`)
- Input: Bottom of the circuit (multiple `o`)
- Fan-in 2: Each gate has 2 input wires
- Layer 0: Output layer
- Layer d: Input layer

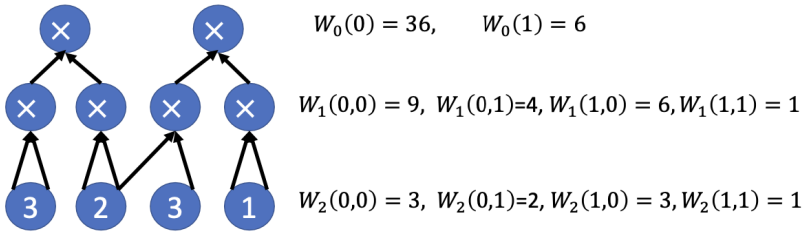


Figure 4.12: Example circuit C and input x, and resulting functions W_i for each layer i of C. Note that C has two output gates.

Number the gates at layer i from 0 to $S_i - 1$, and let $W_i : \{0,1\}^{k_i} \to \mathbb{F}$ denote the function that takes as input a *binary gate label*, and outputs the corresponding gate's value at layer i. As usual, let $\tilde{W}_i$ denote the multilinear extension of W_i.

The GKR protocol also makes use of the notion of a "*wiring predicate*" that encodes which pairs of wires from layer $i+1$ are connected to a given gate at layer i in C. Let $in_{1,i}, in_{2,i} : \{0,1\}^{k_i} \to \{0,1\}^{k_{i+1}}$ denote the functions that take as input the label a of a gate at layer i of C, and respectively output the label of the first and second in-neighbor of gate a. So, for example, if gate a at layer i computes the sum of gates b and c at layer $i+1$, then $in_{1,i}(a) = b$ and $in_{2,i}(a) = c$.

Define two functions, add_i and $mult_i$, mapping $\{0,1\}^{k_i+2k_{i+1}}$ to $\{0,1\}$, which together constitute the wiring predicate of layer i of C. Specifically, these functions take as input three gate labels (a,b,c), and return 1 $\iff$ $(b,c) = (in_{1,i}(a), in_{2,i}(a))$ and gate a is an addition (respectively, multiplication) gate. As usual, let $\tilde{add}_i$ and $\tilde{mult}_i$ denote the multilinear extensions of add_i and $mult_i$.

For an example, consider the circuit depicted in Figure 4.12. Since the circuit contains no addition gates, add_0 and add_1 are the constant 0 function. Meanwhile, $mult_0$ is the function defined over domain $\{0,1\} \times \{0,1\}^2 \times \{0,1\}^2$ as follows. $mult_0$ evaluates to 1 on the following two inputs: $(0,(0,0),(0,1))$ and $(1,(1,0),(1,1))$. On all other inputs, $mult_0$ evaluates to zero. This is because the first and second in-neighbors of gate 0 at layer 0 are respectively gates $(0,0)$ and $(0,1)$ at layer 1, and similarly the first and second in-neighbors of gate 1 at layer 0 are respectively gates $(1,0)$ and $(1,1)$ at layer 1.

The statement describes the connectivity in a layered arithmetic circuit.
Let G_i^j denote the j-th gate at layer i. For layer 0:

1. G_0^0 (gate 0) connects to $G_1^{(0,0)}$ and $G_1^{(0,1)}$ in layer 1
2. G_0^1 (gate 1) connects to $G_1^{(1,0)}$ and $G_1^{(1,1)}$ in layer 1

This connectivity is encoded in the $mult_0$ function, which is essentially a *characteristic function for the graph edges between layers 0 and 1.*

The binary representations (0,0), (0,1), (1,0), (1,1) correspond to the 2-bit addressing scheme for the 4 gates in layer 1, reflecting the circuit's binary tree structure.

```
def mult_0(a, b, c):
    # a in {0,1}, b and c in {0,1}^2
    if (a == 0 and b == (0,0) and c == (0,1)) or \
       (a == 1 and b == (1,0) and c == (1,1)):
        return 1
    return 0

def add_0(a, b, c):
    return 0  # constant 0 function

def add_1(a, b, c):
    return 0  # constant 0 function

# Example usage
print(mult_0(0, (0,0), (0,1)))  # Should return 1
print(mult_0(1, (1,0), (1,1)))  # Should return 1
print(mult_0(0, (0,0), (0,0)))  # Should return 0
```

Imagine a simple circuit with two layers:

1. Top layer (Layer 0): Has 2 gates
2. Bottom layer (Layer 1): Has 4 gates

```
Layer 0:   0       1
          / \     / \
Layer 1: 0   1   2   3
        (00)(01)(10)(11)
```

The circuit only uses multiplication, no addition.

The function mult_0 describes how gates connect:

- It takes 3 inputs: (a, b, c)
- 'a' is the gate number in Layer 0
- 'b' and 'c' are gate numbers in Layer 1

$mult_0$ equals 1 only when: 1. Gate 0 in Layer 0 connects to gates (0,0) and (0,1) in Layer 1 2. Gate 1 in Layer 0 connects to gates (1,0) and (1,1) in Layer 1. For all other connections, mult_0 equals 0. This function helps describe how the circuit is wired.

In this diagram:

- 0 and 1 in Layer 0 are the gate numbers
- 0, 1, 2, 3 in Layer 1 are the gate numbers
- (00), (01), (10), (11) are the binary representations of these numbers

Similarly, $mult_1$ is a function on domain $\{0,1\}^2 \times \{0,1\}^2 \times \{0,1\}^2$. It evaluates to 0 on all inputs except for the following four, on which it evaluates to 1:

- $((0,0),(0,0),(0,0))$.
- $((0,1),(0,1),(0,1))$.
- $((1,0),(0,1),(1,0))$.
- $((1,1),(1,1),(1,1))$.

Note that for each layer i, add_i and $mult_i$ depend only on the circuit C and not on the input x to C. In contrast, the function W_i does depend on x. This is because W_i maps each gate label at layer i to the value of the gate when C is evaluated on input x.

```python
def mult_1(a, b, c):
    valid_inputs = [
        ((0,0),(0,0),(0,0)),
        ((0,1),(0,1),(0,1)),
        ((1,0),(0,1),(1,0)),
        ((1,1),(1,1),(1,1))
    ]
    return 1 if (a, b, c) in valid_inputs else 0

# Example usage
print(mult_1((0,0),(0,0),(0,0)))  # 1
print(mult_1((0,1),(0,1),(0,1)))  # 1
print(mult_1((1,0),(0,1),(1,0)))  # 1
print(mult_1((1,1),(1,1),(1,1)))  # 1
print(mult_1((0,0),(0,1),(0,0)))  # 0
```

```
Layer 1:   00      01      10      11
          /  \    /  \    /  \    /  \
Layer 2: 00  00  01  01  01  10  11  11
        (000)(000)(010)(010)(010)(100)(110)(110)
```

This diagram shows the connections where $mult_1 = 1$. Each node is labeled with its binary representation.

1.2.2 Step 1: Arithmetize C

Arithmetic circuit (over $GF[2]$ means that it consists only of gates of the form ADD and MULT (where addition and multiplication are done modulo 2). We can convert any Boolean circuit, with gates $\wedge$ and $\neg$ into an (turning-complete) arithmetic circuit, by converting a gate $\wedge$ into a gate MULT, and converting a gate $\neg$ into a gate ADD where we add a constant 1 as an input to the gate.

```
def arithmetize_circuit(boolean_circuit):
    arithmetic_circuit = {}

    for gate_id, (gate_type, inputs) in boolean_circuit.items():
        if gate_type == 'AND':
            arithmetic_circuit[gate_id] = ('MULT', inputs)
        elif gate_type == 'NOT':
            arithmetic_circuit[gate_id] = ('ADD', [inputs[0], 'const_1'])

    return arithmetic_circuit

# Example usage
boolean_circuit = {
    'g1': ('AND', ['x1', 'x2']),
    'g2': ('NOT', ['x3']),
    'g3': ('AND', ['g1', 'g2'])
}

arithmetic_circuit = arithmetize_circuit(boolean_circuit)
print(arithmetic_circuit)
```

Results:

```
{'g1': ('MULT', ['x1', 'x2']), 'g2': ('ADD', ['x3', 'const_1']), 'g3': ('MULT', ['g1', 'g2
```

1.2.3 Step 2: Pick H and m s.t.$S = |H|^m$

Pick a subset $H \subseteq F$ and an integer m such that $S = |H|^m$

This way, we can give each of the S gates in a given layer a unique label encoded in H^m A natural choice is:

- $H = \{0, 1\}$ and $m = \log S$

A less natural choice but a common one is:

- $H = \{0, 1, ..., \log S - 1\}$ and $m = \frac{\log S}{\log \log S}$

Jumping ahead, the reason the latter choice is that one can take a field F that contains H of size poly(|H|) so that:

- $|F| >> m \cdot |H|$ and $|F^m| = poly(S)$

This cannot be done with the natural choice of $H = \{0, 1\}$ since then we need to take $|F| \geq \log S$ which results in $|F^m| \geq S^{\log \log S}$ which is super-polynomial.

An illustrative example:

```
import math
import itertools

def next_prime(n):
      def is_prime(n):
          return n > 1 and all(n % i != 0 for i in range(2,
          ↪  int(math.sqrt(n)) + 1))

      while not is_prime(n):
          n += 1
      return n

  def setup_field_encoding(S):
      H = list(range(int(math.log2(S))))
      m = int(math.log2(S) / math.log2(math.log2(S)))
      F_size = next_prime(m * len(H))
      F = range(F_size)
      gate_labels = []
      for indices in itertools.product(H, repeat=m):
          if len(gate_labels) < S:
              gate_labels.append(indices)
      return {"H": H, "m": m, "F": F, "gate_labels": gate_labels}

# Example 1: Small circuit with 8 gates
S1 = 8
encoding1 = setup_field_encoding(S1)
print(f"For S = {S1}:")
print(f"H = {encoding1['H']}")  # H = [0,1,2]
print(f"m = {encoding1['m']}")  # m = 2
print(f"First few gate labels: {encoding1['gate_labels'][:4]}\n")

# Example 2: Medium circuit with 32 gates
S2 = 32
```

```
encoding2 = setup_field_encoding(S2)
print(f"For S = {S2}:")
print(f"H = {encoding2['H']}")  # H = [0,1,2,3,4]
print(f"m = {encoding2['m']}")  # m = 2
print(f"First few gate labels: {encoding2['gate_labels'][:4]}")
```

1.2.4 Step 3 Compute ME/LDE

The prover computes the values of all gates in every layer of the circuit. For layer i define the function $V_i : H^m \to \{0, 1\}$ as the mapping from an encoding of a gate label to the value of the gate, and $\tilde{V}_i : F^m \to F$ be its low-degree extension (LDE), which is the unique function of degree $\leq |H| - 1$ in each variable that agrees with V_i on inputs in H^m

```
import numpy as np
from functools import reduce
import itertools

def compute_gate_values(circuit, layer_i, H, m):
    def V_i(gate_label):
        # Map gate label to its value (0 or 1)
        if tuple(gate_label) in circuit[layer_i]:
            return circuit[layer_i][tuple(gate_label)]
        return 0

    def LDE(x):
        # Compute low-degree extension
        result = 0
        for gate_label in itertools.product(H, repeat=m):
            value = V_i(gate_label)
            term = value
            for j, xj in enumerate(x):
                # Lagrange interpolation
                term *= reduce(lambda a, b: a * b,
                    [(xj - h) / (gate_label[j] - h)
                    for h in H if h != gate_label[j]], 1)
            result += term
        return result

    return V_i, LDE

# Example usage
circuit = {
```

```
    0: {  # layer 0
        (0,0): 1,
        (0,1): 0,
        (1,0): 1,
        (1,1): 0
    }
}
H = [0,1]
m = 2
V_i, LDE = compute_gate_values(circuit, 0, H, m)
# Example usage
circuit = {
    0: {  # layer 0
        (0,0): 1,
        (0,1): 0,
        (1,0): 1,
        (1,1): 0
    }
}
H = [0,1]
m = 2
V_i, LDE = compute_gate_values(circuit, 0, H, m)

print(f"V_i(0,0) = {V_i((0,0))}")  # Output: 1
print(f"V_i(1,1) = {V_i((1,1))}")  # Output: 0
print(f"LDE([0, 0]) = {LDE([0,0])}")  # Output: 1
print(f"LDE([1, 0]) = {LDE([1,0])}")  # Output: 1
print(f"LDE([0, 1]) = {LDE([0,1])}")  # Output: 0
print(f"LDE([1, 1]) = {LDE([1,1])}")  # Output: 0
```

1.2.5 Step 4+: Reductions

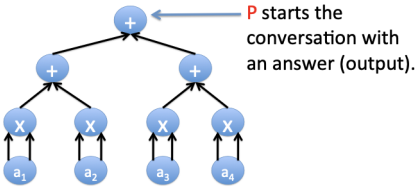


Figure 4.8: Start of GKR Protocol.

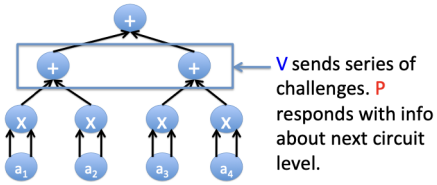


Figure 4.9: Iteration 1 reduces a claim about the output of $\mathcal{C}$ to one about the MLE of the gate values in the previous layer.

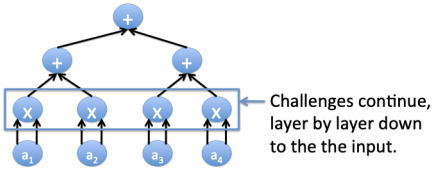


Figure 4.10: In general, iteration i reduces a claim about the MLE of gate values at layer i, to a claim about the MLE of gate values at layer $i+1$.

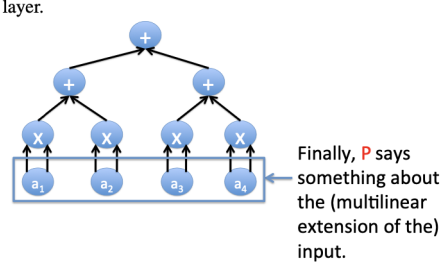


Figure 4.11: In the final iteration, $\mathcal{P}$ makes a claim about the MLE of the input (here, the input of length n with entries in $\mathbb{F}$ is interpreted as a function mapping $\{0,1\}^{\log_2 n} \to \mathbb{F}$. Any such function has a unique MLE by Fact 3.5). $\mathcal{V}$ can check this claim without help, since $\mathcal{V}$ sees the input explicitly.

The GKR protocol consists of d iterations (reductions), one for each layer of the circuit. Each iteration i starts with P claiming a value for $\tilde{W}_i(r_i)$ for some point $r_i \in F^{k_i}$.

At the start of the first iteration, this claim is derived from the claimed outputs of the circuit. Specifically, if there are $S_0 = 2^{k_0}$ outputs of C, let $D : \{0,1\}^{k_0} \to F$ denote the function that maps the label of an output gate to the claimed value of that output. Then the verifier can pick a random point $r_0 \in F^{k_0}$, and evaluate $\tilde{D}(r_0)$ in time $O(S_0)$ using Lemma 3.8 [1]. By the Schwartz-Zippel lemma, if $\tilde{D}(r_0) = \tilde{W}_0(r_0)$ (i.e., if the multilinear extension of the claimed outputs equals the multilinear extension of the correct outputs when evaluated at a randomly chosen point), then it is safe for the verifier to believe that $\tilde{D}$ and $\tilde{W}_0$ are the same polynomial, and hence that all of the claimed outputs are correct. Unfortunately, the verifier cannot evaluate $\tilde{W}_0(r_0)$ without help from the prover.

The purpose of iteration i is to reduce the claim about the value of $\tilde{W}_i(r_i)$ to a claim about $\tilde{W}_{i+1}(r_{i+1})$ for some $r_{i+1} \in F^{k_{i+1}}$, in the sense that it is safe for V to assume that the first claim is true as long as the second claim is true. To accomplish this, the iteration applies the sum-check protocol to a specific polynomial derived (by **Lemma 4.7**) from $\tilde{W}_{i+1}$, $\tilde{add}_i$, and $\tilde{mult}_i$. Our description of the protocol actually makes use of a simplification. The following is based on a simplification by [Thaler, 2015].

[1] **Lemma 3.8** ([Vu et al., 2013]). Fix a positive integer v, and let $n = 2^v$. Given as input $f(w)$ for all $w \in \{0,1\}^v$ and a vector $r = (r_1, ..., r_v) \in F^{\log n}$, V can compute $f(r)$ in $O(n)$ time and $O(n)$ space.

$$\tilde{W}_i(z) = \sum_{b,c \in \{0,1\}^{k_{i+1}}} \widetilde{\text{add}}_i(z,b,c)\left(\tilde{W}_{i+1}(b) + \tilde{W}_{i+1}(c)\right) + \widetilde{\text{mult}}_i(z,b,c)\left(\tilde{W}_{i+1}(b) \cdot \tilde{W}_{i+1}(c)\right)$$

Therefore, in order to check the prover's claim about $\widetilde{W}_i(r_i)$, the verifier applies the sum-check protocol to the polynomial:

$$f_{r_i}^{(i)}(b,c) = \widetilde{\text{add}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b) + \bar{W}_{i+1}(c)\right) + \widetilde{\text{mult}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b) \cdot \bar{W}_{i+1}(c)\right).$$

1.2.6 Reducing to Verification of a Single Point:

Let $\ell : \mathbb{F} \to \mathbb{F}^{i+1}$ be the unique line such that $\ell(0) = b^*$ and $\ell(1) = c^*$. P sends a univariate polynomial q of degree at most k_{i+1} that is claimed to be $\tilde{W}_{i+1} \circ \ell$, the restriction of $\tilde{W}_{i+1}$ to the line l. V checks that $q(0) = z_1$ and $q(1) = z_2$ (rejecting if this is not the case), picks a random point $r^* \in F$, and asks P to prove that $\tilde{W}_{i+1}(\ell(r^*)) = q(r^*)$. By Claim 4.6, as long as V is convinced that $\tilde{W}_{i+1}(\ell(r^*)) = q(r^*)$, it is safe for V to believe that q does in fact equal $\tilde{W}_{i+1} \circ \ell$, and hence that $\tilde{W}_{i+1}(b^*) = z_1$ and $\tilde{W}_{i+1}(c^*) = z_2$ as claimed by P. See Section 4.5.2 (recapped below in A Useful Subroutine: Reducing Multiple Polynomial Evaluations to One) for a picture and example of this sub-protocol.

This completes iteration i; P and V then move on to the iteration for layer $i+1$ of the circuit, whose purpose is to verify that $\tilde{W}_{i+1}(r_{i+1})$ has the claimed value, where $r_{i+1} := \ell(r^*)$.

1. A Useful Subroutine: Reducing Multiple Polynomial Evaluations to One

1.2.7 Final Iteration

At the final iteration d, V must evaluate $\tilde{W}_d(r_d)$ on her own. But the vector of gate values at layer d of C is simply the input x to C. By Lemma 3.8 [1], V can compute $\tilde{W}_d(r_d)$ on her own in $O(n)$ time, where recall that n is the size of the input x to C.

1.2.8 GKR Protocol

Description of the GKR protocol, when applied to a layered arithmetic circuit $\mathcal{C}$ of depth d and fan-in two on input $x \in \mathbb{F}^n$. Throughout, k_i denotes $\log_2(S_i)$ where S_i is the number of gates at layer i of $\mathcal{C}$.

- At the start of the protocol, $\mathcal{P}$ sends a function $D : \{0,1\}^{k_0} \to \mathbb{F}$ claimed to equal W_0 (the function mapping output gate labels to output values).
- $\mathcal{V}$ picks a random $r_0 \in \mathbb{F}^{k_0}$ and lets $m_0 \leftarrow \tilde{D}(r_0)$. The remainder of the protocol is devoted to confirming that $m_0 = \tilde{W}_0(r_0)$.
- For $i = 0, 1, \ldots, d-1$:

– Define the $(2k_{i+1})$-variate polynomial:

$$f_{r_i}^{(i)}(b,c) := \widetilde{\text{add}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b)+\tilde{W}_{i+1}(c)\right)+\widetilde{\text{mult}}_i(r_i,b,c)\left(\tilde{W}_{i+1}(b)\cdot\tilde{W}_{i+1}(c)\right). \tag{1}$$

– $\mathcal{P}$ claims that $\sum_{b,c\in\{0,1\}^{k_{i+1}}} f_{r_i}^{(i)}(b,c) = m_i$.

– So that $\mathcal{V}$ may check this claim, $\mathcal{P}$ and $\mathcal{V}$ apply the sum-check protocol to $f_{r_i}^{(i)}$, up until $\mathcal{V}$'s final check in that protocol, when $\mathcal{V}$ must evaluate $f_{r_i}^{(i)}$ at a randomly chosen point $(b^*,c^*)\in\mathbb{F}^{k_{i+1}}\times\mathbb{F}^{k_{i+1}}$. See Remark (a) at the end of this codebox.

– Let ℓ be the unique line satisfying $\ell(0)=b^*$ and $\ell(1)=c^*$. $\mathcal{P}$ sends a univariate polynomial q of degree at most k_{i+1} to $\mathcal{V}$, claimed to equal $\tilde{W}_{i+1}$ restricted to ℓ.

– $\mathcal{V}$ now performs the final check in the sum-check protocol, using $q(0)$ and $q(1)$ in place of $\tilde{W}_{i+1}(b^*)$ and $\tilde{W}_{i+1}(c^*)$. See Remark (b) at the end of this codebox.

– $\mathcal{V}$ chooses $r^*\in\mathbb{F}$ at random and sets $r_{i+1}=\ell(r^*)$ and $m_{i+1}\leftarrow q(r_{i+1})$.

- $\mathcal{V}$ checks directly that $m_d=\widetilde{W}_d(r_d)$ using Lemma 3.8 [Vu et al., 2013].

  Note that $\widetilde{W}_d$ is simply $\widetilde{x}$, the multilinear extension of the input x when x is interpreted as the evaluation table of a function mapping $\{0,1\}^{\log n}\to\mathbb{F}$.

Remark a. Note that $\mathcal{V}$ does not actually know the polynomial $f_{r_i}^{(i)}$, because $\mathcal{V}$ does not know the polynomial $\tilde{W}_{i+1}$ that appears in the definition of $f_{r_i}^{(i)}$. However, the sum-check protocol does not require $\mathcal{V}$ to know anything about the polynomial to which it is being applied, until the very final check in the protocol (see Remark (4.2) [2].

Remark b. We assume here that for each layer i of $\mathcal{C}$, $\mathcal{V}$ can evaluate the multilinear extensions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at the point (r_i,b^*,c^*) in polylogarithmic time. Hence, given $\tilde{W}_{i+1}(b^*)$ and $\tilde{W}_{i+1}(c^*)$, $\mathcal{V}$ can quickly evaluate $f_{r_i}^{(i)}(b^*,c^*)$ and thereby perform its final check in the sum-check protocol applied to $f_{r_i}^{(i)}$.

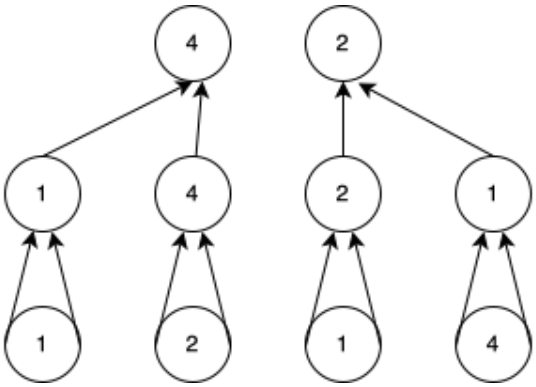
Remark c. The sum-check process is similar as **??**:

- For v-variate polynomial g, $H=\sum_{(b_1,\cdots,b_v)\in\{0,1\}^v} g(b_1,\cdots,b_v)$

[2] **Remark 4.2** An important feature of the sum-check protocol is that the verifier's messages to the prover are simply random field elements, and hence entirely independent of the input polynomial g. In fact, the only information V needs about the polynomial g to execute its part of the protocol is an upper bound on the degree of g in each of its v variables, and the ability to evaluate g at a random point $r\in\mathbb{F}^v$. This means that V can apply the sum-check protocol *even without knowing the polynomial g to which the protocol is being applied*, so long as V knows an upper bound on the degree of the polynomial in each variable, and later obtains the ability to evaluate g at a random point $r\in\mathbb{F}^v$. In contrast, the prover does need to know the precise polynomial g in order to compute each of its messages over the course of the sum-check protocol.

- Prover $\mathscr{P}$ sends a value H and the univariate polynomial $g_1(X_1)$ $g_1(X_1) = \sum_{(x_2,\cdots,x_v)\in\{0,1\}^v} g(X_1, x_2,$
  Verifier $\mathscr{V}$ checks that $H = g_1(0) + g_1(1)$
- For $1 < j < v$, $\mathscr{P}$ sends to $\mathscr{V}$ a univariate polynomial $g_j(X_j)$ claimed to equal:
  $g_i(X_i) = \sum_{(x_{i+1},\cdots,x_v)\in\{0,1\}^{v-i}} g(r_1, \cdots, r_{i-1}, X_i, x_{i+1}, \cdots, x_v)$ $\mathscr{V}$ checks that $g_{i-1}(r_{i-1}) =$
  $g_i(0) + g_i(1)$
- $\mathscr{V}$ chooses a random value $r_j \in \mathbb{F}$ and sends it to $\mathscr{P}$

1.3 Example



Depiction of a circuit over $\mathbb{F}_5$ consisting entirely of multiplication gates, and the multilinear extension encodings $\tilde{W}_i$ of each layer i when the circuit is evaluated on the length-4 input $(1, 2, 1, 4)$. Due to there being two outputs, $\tilde{W}_0$ is a univariate polynomial, and hence its evaluation table consists of $|\mathbb{F}_5| = 5$ values. The other two layers have four gates each, and hence $\tilde{W}_1$ and $\tilde{W}_2$ are bivariate polynomials, the evaluations tables of which each contain $5^2 = 25$ values, indexed from $(0, 0)$ to $(4, 4)$. Entries of the multilinear extension encodings indexed by Boolean vectors are highlighted in blue. In the GKR protocol applied to this circuit on this input, the prover begins by sending the claimed values of the two output gates, thereby specifying W_0, and the verifier evaluates $\tilde{W}_0$ at a random point. Then at the end of each iteration i of the for loop in Figure 4.13, the prover is forced to make a claim about a single (randomly chosen) evaluation of $\tilde{W}_i$.

As illustrated below, we have $\tilde{W}_0(r) = 4 + 3r$, $\tilde{W}_1(r_1, r_2) = (1 - r_1)(1 - r_2) + 4(1 - r_1)r_2 + 2r_1(1 - r_2) + r_1r_2)$ and $\tilde{W}_2(r_1, r_2) = (1-r_1)(1-r_2)+2(1-r_1)r_2+r_1(1-r_2)+4r_1r_2$

Here's a Python implementation using sympy to derive and verify the polynomials:

```
from sympy import symbols, expand

def multilinear_extension_w0():
    r = symbols('r')
```

```
    # W_0 values at points 0 and 1: [4, 7]
    return 4 + 3*r  # Linear interpolation

def multilinear_extension_w1():
    r1, r2 = symbols('r1 r2')
    # Boolean points mapping: (0,0)->1, (0,1)->4, (1,0)->2, (1,1)->1
    return ((1-r1)*(1-r2)*1 +
            (1-r1)*r2*4 +
            r1*(1-r2)*2 +
            r1*r2*1)

def multilinear_extension_w2():
    r1, r2 = symbols('r1 r2')
    # Boolean points mapping: (0,0)->1, (0,1)->2, (1,0)->1, (1,1)->4
    return ((1-r1)*(1-r2)*1 +
            (1-r1)*r2*2 +
            r1*(1-r2)*1 +
            r1*r2*4)
```

This implements the multilinear extensions by:

1. $\tilde{W}_0$: Linear interpolation between output points
2. $\tilde{W}_1, \tilde{W}_2$: Using boolean hypercube interpolation formula with values at $(0,0), (0,1), (1,0), (1,1)$

The boolean hypercube interpolation formula for a multivariate function creates a polynomial that matches given values at boolean inputs (0s and 1s). For two variables, it's:
$f(r_1, r_2) = \sum_{b_1,b_2 \in \{0,1\}} f(b_1, b_2) \cdot [\prod_{i=1}^{2}(b_i r_i + (1 - b_i)(1 - r_i))]$
where:

- $f(b_1, b_2)$ is the known value at boolean point (b_1, b_2)
- $(b_i r_i + (1 - b_i)(1 - r_i))$ is the selector term

Example for point $(0, 0)$:

- $b_1 = 0, b_2 = 0$
- Selector: $(1 - r_1)(1 - r_2)$
- If $r_1 = r_2 = 0$, selector $= 1$
- If any $r_i = 1$, selector $= 0$

A simplified GKR main protocol implementation below:

```
import numpy as np

# Define the field
F = np.array(range(5))

# Define the input
x = np.array([1, 2, 1, 4])

# Define the circuit
W0 = np.array([1, 4, 4, 2, 3])  # Output layer
W1 = np.array([[1, 2, 1, 4],
               [2, 4, 2, 3],
               [1, 2, 1, 4],
               [4, 3, 4, 1]])
W2 = np.array([[1, 2, 1, 4],
               [2, 4, 2, 3],
               [1, 2, 1, 4],
               [4, 3, 4, 1]])

# Define the multilinear extensions
def multilinear_extension(W):
    return np.polynomial.polynomial.polyval2d(F, F, W)

W0_tilde = multilinear_extension(W0)
W1_tilde = multilinear_extension(W1)
W2_tilde = multilinear_extension(W2)

# Define the add and mult gates
def add_gate(r, b, c):
    return r + b + c

def mult_gate(r, b, c):
    return r * b * c

# GKR protocol
def gkr_protocol(W0, W1, W2, x):
    k0 = int(np.log2(len(W0)))
    k1 = int(np.log2(len(W1)))
    k2 = int(np.log2(len(W2)))

    # Step 1: Prover sends W0
    D = W0
```

```
    # Step 2: Verifier chooses random r0
    r0 = np.random.randint(5)

    # Step 3: Verifier checks D(r0) == W0_tilde(r0)
    m0 = W0_tilde[r0]

    # Step 4: Iterate through layers
    for i in range(2):
        # Define f_r_i
        if i == 0:
            f_r_i = lambda b, c: add_gate(r0, b, c) * (W1_tilde[b] +
            ↪  W1_tilde[c]) + mult_gate(r0, b, c) * (W1_tilde[b] *
            ↪  W1_tilde[c])
        else:
            f_r_i = lambda b, c: add_gate(r1, b, c) * (W2_tilde[b] +
            ↪  W2_tilde[c]) + mult_gate(r1, b, c) * (W2_tilde[b] *
            ↪  W2_tilde[c])

        # Step 5: Prover claims sum of f_r_i
        sum_f_r_i = np.sum(f_r_i(F, F))

        # Step 6: Apply sum-check protocol (not implemented here)

        # Step 7: Prover sends univariate polynomial q
        if i == 0:
            b_star = np.random.randint(5)
            c_star = np.random.randint(5)
            q = lambda r: W1_tilde[int(np.round(r))]
        else:
            b_star = np.random.randint(5)
            c_star = np.random.randint(5)
            q = lambda r: W2_tilde[int(np.round(r))]

        # Step 8: Verifier checks sum-check protocol
        m_i = q(0) + q(1)

        # Step 9: Verifier chooses random r*
        r_star = np.random.randint(5)

        # Step 10: Verifier sets r_(i+1) and m_(i+1)
        if i == 0:
            r1 = np.round(r_star)
            m1 = q(r1)
        else:
```

```
            r2 = np.round(r_star)
            m2 = q(r2)

    # Step 11: Verifier checks m_d == W_d(r_d)
    if m2 == W2_tilde[r2]:
        print("Proof accepted.")
    else:
        print("Proof rejected.")

gkr_protocol(W0, W1, W2, x)
```

Or using sympy:

```
import numpy as np
from sympy import symbols, FiniteField as FF
import random


class GKRProtocol:
    def __init__(self, p=5):
        self.F = FF(p)
        self.b, self.c = symbols("b c")

    def random_field_element(self):
        return self.F(random.randint(0, self.F.mod - 1))

    def create_multilinear_extension(self, values):
        def evaluate(*vars):
            result = self.F(0)
            for i, val in enumerate(values):
                term = self.F(val)
                for j, var in enumerate(vars):
                    if i & (1 << j):  # Check if the j-th bit of i is 1
                        term *= self.F(var)
                result += term
            return result

        return evaluate

    def evaluate_gate(self, gate_type, r, b, c):
        if gate_type == "add":
            return self.F(r + b + c)
        return self.F(r * b * c)
```

```
def run_protocol(self, circuit_input, W0, W1, W2):
    # Create multilinear extensions
    V0_tilde = self.create_multilinear_extension(W0)
    V1_tilde =
    ↪  self.create_multilinear_extension(np.hstack(W1.flatten()))
    V2_tilde =
    ↪  self.create_multilinear_extension(np.hstack(W2.flatten()))

    # Step 2: Verifier picks random point
    r0 = self.random_field_element()
    m0 = V0_tilde(r0)

    # Iterate through layers
    for i in range(2):
        current_V = V1_tilde if i == 0 else V2_tilde
        current_r = r0 if i == 0 else r1

        def f_r_i(b, c):
            add_term = self.evaluate_gate("add", current_r, b, c)
            mult_term = self.evaluate_gate("mult", current_r, b, c)
            if i == 0:  # W1 to W0 transition
                return add_term
            else:  # W2 to W1 transition (assuming W1 outputs are XOR
            ↪  for simplicity)
                return self.F(
                    add_term + mult_term
                )  # Note: This logic might need adjustment based on
                ↪  your circuit

        # Sum-check protocol
        b_star = self.random_field_element()
        c_star = self.random_field_element()

        # Prover sends univariate polynomial q
        q = lambda c: f_r_i(b_star, c)

        # Verifier checks
        r_star = self.random_field_element()
        if i == 0:
            r1 = r_star
            m1 = q(r_star)
        else:
            r2 = r_star
```

```
                m2 = q(r_star)

        # Final check
        final_check = V2_tilde(r2, r2)
        return final_check == m2


def main():
    F = np.array(range(5))
    x = np.array([1, 2, 1, 4])
    W0 = np.array([1, 4, 4, 2, 3])
    W1 = np.array([[1, 2, 1, 4], [2, 4, 2, 3], [1, 2, 1, 4], [4, 3, 4,
    ↪  1]])
    W2 = np.array([[1, 2, 1, 4], [2, 4, 2, 3], [1, 2, 1, 4], [4, 3, 4,
    ↪  1]])

    gkr = GKRProtocol(5)
    result = gkr.run_protocol(x, W0, W1, W2)
    print(f"Protocol verification result: {result}")


if __name__ == "__main__":
    main()
```

References

MIT OpenCourseware. 6.5610: Applied cryptography. https://65610.csail.mit.edu/2024/overview.html, 2024.

Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33rd Annual Conference on Advances in Cryptology*, CRYPTO'13. Springer-Verlag, 2013.

Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013*, pages 223–237. IEEE Computer Society, May 2013. 19-22.

Justin Thaler. A note on the gkr protocol, 2015. URL http://people.cs.georgetown.edu/jthaler/GKRNote.pdf.