理解 Plonk（五）：多项式承诺

什么是多项式承诺

所谓承诺，是对消息「锁定」，得到一个锁定值。这个值被称为对象的「承诺」。

$$c = commit(x)$$

这个值和原对象存在两个关系，即 Hiding 与 Binding。

Hiding： c 不暴露任何关于 x 的信息；

Binding：难以找到一个 $x', x' \neq x$，使得 $c = commit(x')$。

最简单的承诺操作就是 Hash 运算。请注意这里的 Hash 运算需要具备密码学安全强度，比如 SHA256, Keccak 等。除了 Hash 算法之外，还有 Pedersen 承诺等。

顾名思义，多项式承诺可以理解为「多项式」的「承诺」。如果我们把一个多项式表达成如下的公式，

$$f(X) = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$$

那么我们可以用所有系数构成的向量来唯一标识多项式 $f(X)$。

$$(a_0, a_1, a_2, \ldots, a_n)$$

如何对一个多项式进行承诺？很容易能想到，我们可以把「系数向量」进行 Hash 运算，得到一个数值，就能建立与这个多项式之间唯一的绑定关系。

$$C_1 = \mathrm{SHA256}(a_0 \parallel a_1 \parallel a_2 \parallel \cdots \parallel a_n)$$

或者，我们也可以使用 Petersen 承诺，通过一组随机选择的基，来计算一个 ECC 点：

$$C_2 = a_0G_0 + a_1G_1 + \cdots + a_nG_n$$

如果在 Prover 承诺多项式之后，Verifier 可以根据这个承诺，对被锁定的多项式进行求值，并希望 Prover 可以证明求值的正确性。假设 $C = Commit(f(X))$，Verifier 可以向提供承诺的 Prover 询问多项式在 $X = \zeta$ 处的取值。Prover 除了回复一个计算

结果之外（如 $f(\zeta)=y$），还能提供一个证明 π，证明 C 所对应的多项式 $f(X)$ 在 $X=\zeta$ 处的取值 y 的正确性。

多项式承诺的这个「携带证明的求值」特性非常有用，它可以被看成是一种轻量级的「可验证计算」。即 Verifier 需要把多项式 $f(X)$ 的运算代理给一个远程的机器（Prover），然后验证计算（计算量要小于直接计算 $f(X)$）结果 y 的正确性；多项式承诺还能用来证明秘密数据（来自Prover）的性质，比如满足某个多项式，Prover 可以在不泄漏隐私的情况下向 Verifier 证明这个性质。

虽然这种可验证计算只是局限在多项式运算上，而非通用计算。但通用计算可以通过各种方式转换成多项式计算，从而依托多项式承诺来最终实现通用的可验证计算。

按上面 C_2 的方式对多项式的系数进行 Pedersen 承诺，我们仍然可以利用 Bulletproof-IPA 协议来实现求值证明，进而实现另一种多项式承诺方案。此外，还有 KZG10 方案，FRI，Dark，Dory 等等其它方案。

KZG10 构造

与 Pedersen 承诺中用的随机基向量相比，KZG10 多项式承诺需要用一组具有内部代数结构的基向量来代替。

$$(G_0,G_1,G_2,\ldots,G_{d-1},H_0,H_1)=(G,\chi G,\chi^2G,\ldots,\chi^{d-1}G,H,\chi H)$$

请注意，这里的 χ 是一个可信第三方提供的随机数，也被称为 Trapdoor，需要在第三方完成 Setup 后被彻底删除。它既不能让 Verifier 知道，也不能让 Prover 知道。当 $\vec{G}$ 设置好之后，χ 被埋入了基向量中。这样一来，从外部看，这组基向量与随机基向量难以被区分。其中 $G\in\mathbb{G}_1$，而 $H\in\mathbb{G}_2$，并且存在双线性映射 $e\in\mathbb{G}_1\times\mathbb{G}_2\to\mathbb{G}_T$。

对于一个多项式 $f(X)$ 进行 KZG10 承诺，也是对其系数向量进行承诺:

$$\begin{aligned}C_{f(X)}&=a_0G_0+a_1G_1+\cdots+a_{n-1}G_{n-1}\\&=a_0G+a_1\chi G+\cdots+a_{n-1}\chi^{n-1}G\\&=f(\chi)G\end{aligned}$$

这样承诺 $C_{f(X)}$ 巧好等于 $f(\chi)G$。

对于双线性群，我们下面使用 Groth 发明的符号 $[1]_1\triangleq G$，$[1]_2\triangleq H$ 表示两个群上的生成元，这样 KZG10 的系统参数（也被称为 SRS, Structured Reference String）

可以表示如下：

$$\mathsf{srs} = ([1]_1, [\chi]_1, [\chi^2]_1, [\chi^3]_1, \ldots, [\chi^{n-1}]_1, [1]_2, [\chi]_2)$$

而 $C_{f(X)} = [f(\chi)]_1$。

下面构造一个 $f(\zeta) = y$ 的 Open 证明。根据多项式余数定理，我们可以得到下面的等式：

$$f(X) = q(X) \cdot (X - \zeta) + y$$

这个等式可以解释为，任何一个多项式都可以除以另一个多项式，得到一个商多项式加上一个余数多项式。由于多项式在 $X = \zeta$ 处的取值为 y，那么我们可以确定：余数多项式一定为 y ，因为等式右边的第一项在 $X = \zeta$ 处取值为零。所以，如果 $f(\zeta) = y$ ，我们可以断定： $g(X) = f(X) - y$ 在 $X = \zeta$ 处等零，所以 ζ 为 $g(X)$ 的根，于是 $g(X)$ 一定可以被 $(X - \zeta)$ 这个不可约多项式整除，即一定**存在**一个商多项式 $q(X)$，满足上述等式。

而 Prover 则可以提供 $q(X)$ 多项式的承诺，记为 C_q，作为 $f(\zeta) = y$ 的证明，Verifier 可以检查 $[q(\chi)]$ 是否满足整除性来验证证明。因为如果 $f(\zeta) \neq y$，那么 $g(X)$ 则无法被 $(X - \zeta)$ 整除，即使 Prover 提供的承诺将无法通过整除性检查：

$$(f(X) - y) \cdot 1 \stackrel{?}{=} q(X) \cdot (X - \zeta)$$

承诺 $C_{f(X)}$ 是群 $\mathbb{G}_1$ 上的一个元素，通过承诺的加法同态映射关系，以及双线性映射关系 $e \in \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$，Verifier 可以在 $\mathbb{G}_T$ 上验证整除性关系：

$$e(C_{f(X)} - y[1]_1, [1]_2) \stackrel{?}{=} e(C_{q(X)}, [\chi]_2 - \zeta[1]_2)$$

有时为了减少 Verifier 在 $\mathbb{G}_2$ 上的昂贵操作，上面的验证等式可以变形为：

$$f(X) + \zeta \cdot q(X) - y = q(X) \cdot X$$

$$e(C_{f(X)} + \zeta \cdot C_{q(X)} - y \cdot [1]_1,\ [1]_2) \stackrel{?}{=} e(C_{q(X)},\ [\chi]_2)$$

同点 Open 的证明聚合

在一个更大的安全协议中，假如同时使用多个多项式承诺，那么他们的 Open 操作可以合并在一起完成。即把多个多项式先合并成一个更大的多项式，然后仅通过 Open 一点，来完成对原始多项式的批量验证。

假设我们有多个多项式， $f_1(X)$， $f_2(X)$，Prover 要同时向 Verifier 证明 $f_1(\zeta) = y_1$ 和 $f_2(\zeta) = y_2$，那么有

$$\begin{aligned} f_1(X) &= q_1(X) \cdot (X - \zeta) + y_1 \\ f_2(X) &= q_2(X) \cdot (X - \zeta) + y_2 \end{aligned}$$

通过一个随机数 ν，Prover 可以把两个多项式 $f_1(X)$ 与 $f_2(X)$ 折叠在一起，得到一个临时的多项式 $g(X)$：

$$g(X) = f_1(X) + \nu \cdot f_2(X)$$

进而我们可以根据多项式余数定理，推导验证下面的等式：

$$g(X) - (y_1 + \nu \cdot y_2) = (X - \zeta) \cdot (q_1(X) + \nu \cdot q_2(X))$$

我们把等号右边的第二项看作为「商多项式」，记为 $q(X)$：

$$q(X) = q_1(X) + \nu \cdot q_2(X)$$

假如 $f_1(X)$ 在 $X = \zeta$ 处的求值证明为 π_1，而 $f_2(X)$ 在 $X = \zeta$ 处的求值证明为 π_2，那么根据群加法的同态性，Prover 可以得到商多项式 $q(X)$ 的承诺：

$$[q(\chi)]_1 = \pi = \pi_1 + \nu \cdot \pi_2$$

因此，只要 Verifier 发给 Prover 一个额外的随机数 ν，双方就可以把两个（甚至多个）多项式承诺折叠成一个多项式承诺 C_g：

$$C_g = C_1 + \nu * C_2$$

并用这个折叠后的 C_g 来验证多个多项式在一个点处的运算取值：

$$y_g = y_1 + \nu \cdot y_2$$

从而把多个求值证明相应地折叠成一个，Verifier 可以一次验证完毕：

$$e(C - y * G_0, H_0) \stackrel{?}{=} e(\pi, H_1 - x * H_0)$$

由于引入了随机数 ν，因此多项式的合并不会影响承诺的绑定关系（Schwartz-Zippel 定理）。

协议：

公共输入：$C_{f_1} = [f_1(\chi)]_1$， $C_{f_2} = [f_2(\chi)]_1$， ζ， y_1， y_2

私有输入：$f_1(X)$， $f_2(X)$

证明目标：$f_1(\zeta) = y_1$， $f_2(\zeta) = y_2$

第一轮：Verifier 提出挑战数 ν

第二轮：Prover 计算 $q(X) = f_1(X) + \nu \cdot f_2(X)$，并发送 $\pi = [q(\chi)]_1$

第三轮：Verifier 计算 $C_g = C_{f_1} + \nu \cdot C_{f_2}$， $y_g = y_1 + \nu \cdot y_2$

$$e(C_g - [y_g]_1, [1]_2) \overset{?}{=} e(\pi, [\chi - \zeta]_2)$$

多项式约束与线性化

假设 $[f(\chi)]_1, [g(\chi)]_1, [h(\chi)]_1$ 分别是 $f(X), g(X), h(X)$ 的 KZG10 承诺，如果 Verifier 要验证下面的多项式约束：

$$f(X) + g(X) \overset{?}{=} h(X)$$

那么 Verifier 只需要把前两者的承诺相加，然后判断是否等于 $[h(\chi)]_1$ 即可

$$[f(\chi)]_1 + [g(\chi)]_1 \overset{?}{=} [h(\chi)]_1$$

如果 Verifier 需要验证的多项式关系涉及到乘法，比如：

$$f(X) \cdot g(X) \overset{?}{=} h(X)$$

最直接的方法是利用双线性群的特性，在 $\mathbb{G}_T$ 上检查乘法关系，即验证下面的等式：

$$e([f(\chi)]_1, [g(\chi)]_2) \overset{?}{=} e([h(\chi)]_1, [1]_2)$$

但是如果 Verifier 只有 $g(X)$ 在 $\mathbb{G}_1$ 上的承诺 $[g(\chi)]_1$，而非是在 $\mathbb{G}_2$ 上的承诺 $[g(\chi)]_2$，那么Verifer 就无法利用双线性配对操作来完成乘法检验。

另一个直接的方案是把三个多项式在同一个挑战点 $X = \zeta$ 上打开，然后验证打开值之间的关系是否满足乘法约束：

$$f(\zeta)\cdot g(\zeta)\stackrel{?}{=}h(\zeta)$$

同时 Prover 还要提供三个多项式求值的证明 $(\pi_{f(\zeta)}, \pi_{g(\zeta)}, \pi_{h(\zeta)})$ 供 Verifier 验证。

这个方案的优势在于多项式的约束关系可以更加复杂和灵活，比如验证下面的稍微复杂些的多项式约束：

$$f_1(X)f_2(X)+h_1(X)h_2(X)h_3(X)+g(X)=0$$

假设 Verifier 已拥有这些多项式的 KZG10 承诺，$[f_1(\chi)]_1$，$[f_2(\chi)]_1$，$[h_1(\chi)]_1$，$[h_2(\chi)]_1$，$[h_3(\chi)]_1$，$[g(\chi)]_1$。最直接粗暴的方案是让 Prover 在挑战点 $X=\zeta$ 处打开这 6 个承诺，发送 6 个 Open 值和对应的求值证明：

$$(f_1(\zeta),\pi_{f_1}),(f_2(\zeta),\pi_{f_2}),(h_1(\zeta),\pi_{h_1}),(h_2(\zeta),\pi_{h_2}),(h_3(\zeta),\pi_{h_3}),(g(\zeta),\pi_g)$$

Verifier 验证 6 个求值证明，并且验证多项式约束：

$$f_1(\zeta)f_2(\zeta)+h_1(\zeta)h_2(\zeta)h_3(\zeta)+g(\zeta)\stackrel{?}{=}0$$

我们可以进一步优化，比如考虑对于 $f(X)\cdot g(X)=h(X)$ 这样一个简单的多项式约束，Prover 可以减少 Open 的数量。比如 Prover 先 Open $\bar{f}=f(\zeta)$，发送求值证明 $\pi_{f(\zeta)}$ 然后引入一个辅助多项式 $L(X)=\bar{f}\cdot g(X)-h(X)$，再 Open $L(X)$ 在 $X=\zeta$ 处的取值。

显然对于一个诚实的 Prover，$L(\zeta)$ 求值应该等于零。对于 Verifier，它在收到 $\bar{f}$ 之后，就可以利用承诺的加法同态性，直接构造 $L(X)$ 的承诺：

$$[L(\chi)]_1=\bar{f}\cdot[g(\chi)]_1-[h(\chi)]_1$$

这样一来，Verifier 就不需要单独让 Prover 发送 $L(X)$ 的 Opening，也不需要发送新多项式 $L(X)$ 的承诺。Verifier 然后就可以验证 $f(X)\cdot g(X)=h(X)$ 这个多项式约束关系：

$$e([L(\chi)]_1,[1]_2)\stackrel{?}{=}e(\pi_{L(\zeta)},[\chi-\zeta]_2)$$

这个优化过后的方案，Prover 只需要 Open 两次。第一个 Opening 为 $\bar{f}$，第二个 Opening 为 0。而后者是个常数，不需要发送给 Verifier。Prover 只需要发送两个求值证明，不过我们仍然可以用上一节提供的聚合证明的方法，通过一个挑战数 ν，Prover 可以聚合两个多项式承诺，然后仅需要发送一个求值证明。

我们下面尝试优化下 6 个多项式的约束关系的协议：$f_1(X)f_2(X)+h_1(X)h_2(X)h_3(X)+g(X)=0$。

协议：

公共输入： $C_{f_1}=[f_1(\chi)]_1$， $C_{f_2}=[f_2(\chi)]_1$， $C_{h_1}=[h_1(\chi)]_1$， $C_{h_2}=[h_2(\chi)]_1$， $C_{h_3}=[h_3(\chi)]_1$， $C_g=[g(\chi)]_1$，

私有输入： $f_1(X)$， $f_2(X)$， $h_1(X)$， $h_2(X)$， $h_3(X)$， $g(X)$

证明目标： $f_1(X)f_2(X)+h_1(X)h_2(X)h_3(X)+g(X)=0$

第一轮：Verifier 发送 $X=\zeta$

第二轮：Prover 计算并发送三个Opening， $\bar{f}_1=f_1(\zeta)$， $\bar{h}_1=h_1(\zeta)$， $\bar{h}_2=h_2(\zeta)$，

第三轮：Verifier 发送 ν 随机数

第四轮：Prover 计算 $L(X)$ ，利用 ν 折叠 $(L(X),f_1(X),h_1(X),h_2(X))$ 这四个承诺，并计算商多项式 $q(X)$，发送其承诺 $[q(\chi)]_1$ 作为折叠后的多项式在 $X=\zeta$ 处的求值证明

$$L(X)=\bar{f}_1\cdot f_2(X)+\bar{h}_1\bar{h}_2\cdot h_3(X)+g(X)$$

$$q(X)=\frac{1}{X-\zeta}\Big(L(X)+\nu\cdot(f_1(X)-\bar{f}_1)+\nu^2\cdot(h_1(X)-\bar{h}_1)+\nu^3\cdot(h_2(X)$$

第五轮：Verifier 计算辅助多项式 $L(X)$ 的承诺 $[L]_1$：

$$[L]_1=\bar{f}_1\cdot[f_2(\chi)]_1+\bar{h}_1\bar{h}_2\cdot[h_3(\chi)]_1+[g(\chi)]_1$$

计算折叠后的多项式的承诺：

$$[F]_1=[L]_1+\nu\cdot[f_1(\chi)]_1+\nu^2[h_1(\chi)]_1+\nu^3[h_2(\chi)]_1$$

计算折叠后的多项式在 $X=\zeta$ 处的求值：

$$E=\nu\cdot\bar{f}_1+\nu^2\cdot\bar{h}_1+\nu^3\cdot\bar{h}_2$$

检查下面的验证等式：

$$e([F]_1-[E]_1+\zeta[q(\chi)]_1,[1]_2)\overset{?}{=}e([q(\chi)]_1,[\chi]_2)$$

这个优化后的协议，Prover 仅需要发送三个 Opening，一个求值证明；相比原始方案的 6 个 Opening和 6 个求值证明，大大减小了通信量（即证明大小）。

Reference

Found a bug?! Edit this page on GitHub.