CHIALISP

Andrés Riofrío - Valdivieso

12 de diciembre de 2021

Índice

1. Instalación

1.1. MAC

- Instalar Python3
- Instalar pip:

```
python3 get-pip.py
```

- Ir a la carpeta en donde guardaremos el entorno virtual (Documentos en este ejemplo):

```
cd /Users/[userName]/Documents
```

- Crear entorno virtual de python3:

```
python3 -m venv venv
```

- Activar el entorno virtual:

```
ln -s venv/bin/activate
```

- Ejecutamos el entorno virtual:

```
source activate
```

- Instalar clvm_tools:

```
pip install clvm_tools
```

- Probar que funciona (debe retornar 100):

```
brun '(q . 100)'
```

Nota: De aquí en adelante ir a la carpeta en donde guardamos el entorno virtual y ejecutar "source activate"para empezar a trabajar.

2. Conceptos Básicos de CLVM

CLVM es la versión compilada de ChiaLisp, la cuál es usada por la red de Chia.

2.1. Atoms

Un atom es un string de bytes. Puede ser interpretado como “signed big-endian integer” y como “byte string” dependiendo del operador que se use. Todos los átomos son inmutables. El resultado de todas las operaciones es un nuevo átomo. Estos átomo pueden ser impresos cómo: décimales, valor hexadecimal (prefijo 0x) o string (se encuetra en comillas). Cuando se usan átomo como enteros hay que recordar que tienen un signo (+ 0 -).

2.2. Cons Boxes

Conjunto de dos elementos separados por un punto. Ejemplo:

```
("hello" . ("world" . "!!!"))
```

2.3. Lists

Conjunto de 1 o más elementos (pueden estar mezclados string, decimales o hexadecimales). Por tanto una list puede ser representada por cons boxes. Por ejemplo:

```
(200 300 400) = (200 . (300 . (400 . ())))
```

2.4. Quoting (q)

Es una forma de interpretar un átomo como un valor y no como un programa. Por ejemplo:

```
(q . 100)
```

2.5. Programas

Los programas son un subconjunto de las lists y pueden ser evaluados usando el CLVM. Para ello se usa la notación polaca (se escribe primeramente el operador).

2.5.1. Regla 1 para que un programa sea valido:

- El primer item en la lista debe ser un operador válido
- Cada item después del primero, deber ser un programa válido

2.5.2. Regla 2 para que un programa sea valido:

Los valores (no programas) deben usar quots (q .). Ejemplo (resulado retorna 70):

```
$ brun '(i (= (q . 50) (q . 50)) (+ (q . 40) (q . 30)) (q . 20))' '()'
```

2.6. Operadores

- f: retorna el primer elemento de una lista

```
$ brun '(f (q . (80 90 100)))'
80
```

- r: retorna los elementos de la lista menos el primero

```
$ brun '(r (q . (80 90 100)))'
(90 100)
```

- c: antepone un nuevo elemento a una lista

```
$ brun '(c (q . 70) (q . (80 90 100)))'
(70 80 90 100)
```

- /,*,+,-: Operadores matemáticos. Se va a tener resultados no esperados cuando se divide para números negativos:

```
$ brun '(/ (q . 3) (q . 2))'
1
```

```
$ brun '(/ (q . 3) (q . -2))'
-2
```

```
$ brun '(/ (q . -3) (q . 2))'
-2
```

```
$ brun '(/ (q . -3) (q . -2))'
1
```

- =: operador boleano. Retorna () para falso o 1 si es verdadero

```
$ brun '(= (q . 100) (q . 90))'
()
```

```
$ brun '(= (q . 100) (q . 100))'
1
```

```
$ brun '(= (q . 0) ())'
1
```

- i: Actúa como el operador. Consta de 4 elementos (i,A,B,C), si A es verdadero entonces se evalúa el elemento B caso contrario se evalúa el elemento C.

```
$ brun '(i () (q . 70) (q . 80))'
80
```

- a: Permite que evaluemos el árbol binario por ramas cuando usamos el operador "i" (no todo al mismo tiempo como es por defecto) con la finalidad de no hacerlo todo al mismo tiempo. Cuando usamos el operador "a" e "i" debemos especificar la rama que se evaluará primero (B ó C) es decir: $(a(i \quad A \quad B \quad C)())$ ó $(a(i \quad A \quad B \quad C)1)$.

```
$ brun '(a (q . (+ 2 (q . 5))) (q . (70 80 90)))' '(20 30 40)'
75
```

```
$ brun '(a (q . (i (q . 0) (q . (x (q . 1337) ))(q . (q . 1)))) ())'
(q . 1)
```

```
$ brun '(a (i (q . 1) (q . (q . 100)) (q . (x (q . "still being evaluated")))) 1)'
100
```

2.7. Soluciones y Variables del entorno

Al momento de ejecutar en el terminal debemos usar 3 elementos:

"1) brun, 2) programa a ejecutar, puzzle, operadores, etc., 3) variables de entorno sobre el cuál actuarán operadores"

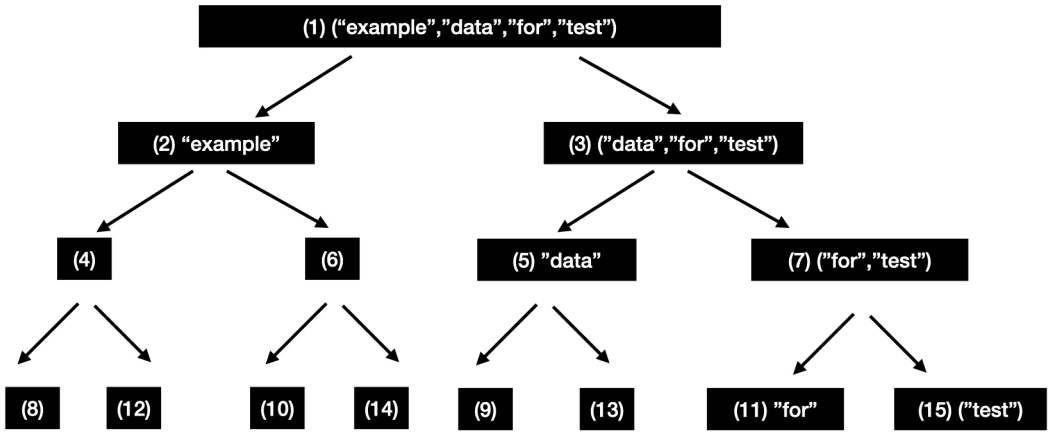
Como ejemplo tenemos:

```
$ brun '1' '("this" "is the" "solution")'
("this" "is the" "solution")
```

```
$ brun '(f (f (r 1)))' '((70 80) (90 100) (110 120))'
90
```

Se puede acceder a las variables de entorno mediante números enteros, tomando en cuenta un árbol binario de operadores "f" y "r". Ejemplo:

```
$ brun '5' '("example" "data" "for" "test")'
"data"
```



3. Monedas, Gastos y Billeteras

3.1. Monedas

El ID de la moneda esta contruida por:

- ID del padre
- El hash del padre (puzzlehash)
- La cantidad que vale

Entonces:

```
coinID == sha256(parent_ID + puzzlehash + amount)
```

No se puede cambiar el puzzle, o la cantidad de una moneda sólo su gasto. El cuerpo de una moneda consta de tres piezas de información. El siguiente código muestra cómo se define una moneda.

```
class Coin:
    parent_coin_info: bytes32
    puzzle_hash: bytes32
    amount: uint64
```

3.2. Gastos

Cuando se gasta una moneda, se drestruye la misma y su valor. A no ser que un puzzle designe que hacer cuando se gaste la misma.

Para gastar una moneda se necesitan 3 piezas de información y una cuarta que es opcional.

- El ID de la moneda
- La información completa del puzzle
- Solución del puzzle
- (Opcional) conjunto de firmas agrupadas (aggregated signature)

La moneda no tiene dueño, cualquier persona puede gastar cualquier moneda de la red. Por tal motivo, el puzzle previene que las monedas puedan ser robadas o gastadas de manera no deseadas.

3.3. Puzzles y soluciones en la práctica

Hasta ahora hemos cubierto los programas de ChiaLisp que se evaluarán hasta obtener algún resultado. La primera parte representa un puzzle que bloquea una moneda y la segunda parte es una solución que cualquiera puede enviar.

```
$ brun '(+ 2 5)' '(40 50)'
90
```

```
$ brun '(c (q . 800) 1)' '("some data" 0xdeadbeef)'
(800 "some data" 0xdeadbeef)
```

Hay que tomar en cuenta que el formato dado no puede ser utilizado para comunicar instrucciones en la blockchain o como va a ser el comportamiento cuando la moneda es gastada. Para esto se deben tomar en cuentas ciertas condiciones.

3.4. Condiciones

Las categorias son divididas en dos categorías: "Este gasto es sólo válido si X"y "Si el gasto es válido entonces X". Lista de condiciones junto a su formato y comportamiento.

- AGG_SIG_UNSAFE - [49] - (49 pubkey message): Este gasto es sólo válido si la firma adjunta contiene una firma de la llave pública del mensaje dado
- AGG_SIG_ME - [50] - (50 pubkey message): Este gasto es sólo válido si la firma agregada adjunta contiene una firma
- CREATE_COIN - [51] - (51 puzzlehash amount): Si el gasto es válido entonces crear una nueva moneda de acuerdo al puzzlehash y la cantidad dada
- ASSERT_FEE - [52] - (52 amount): Este gasto es sólo válido si hay un valor no utilizado en esta transacción igual a la cantidad que se utilizará explícitamente como fee.
- CREATE_COIN_ANNOUNCEMENT - [60] - (60 message): Si el gasto es válido entonces esto crea un anuncio efímero con una identificación dependiente de la moneda que lo crea. Otras monedas pueden afirmar que existe un anuncio para la comunicación entre monedas dentro de un bloque.
- ASSERT_COIN_ANNOUNCEMENT - [61] - (61 announcementID): Este gasto es sólo válido si hubo un anuncio en este bloque que coincida con el announcementID. El ID de anuncio es el hash del mensaje que se anunció concatenado con el ID de moneda, de la moneda que lo anunció:

$$announcementID == sha256(coinID + message)$$

- CREATE_PUZZLE_ANNOUNCEMENT - [62] - (62 message): Si el gasto es válido entonces esto crea un anuncio efímero con una identificación dependiente del puzzle que lo crea. Otras monedas pueden afirmar que existe un anuncio para la comunicación entre monedas dentro de un bloque.
- ASSERT_PUZZLE_ANNOUNCEMENT - [63] - (63 announcementID): Este gasto es sólo válido si hubo un anuncio en este bloque que coincida con el announcementID. El announcementID es el mensaje que se anunció concatenado con el hash del puzzle de la moneda que lo anunció

$$announcementID == sha256(puzzle_hash + message)$$

- ASSERT_MY_COIN_ID - [70] - (70 coinID): Este gasto es sólo válido si el ID de moneda presentado es exactamente el mismo que el ID de la moneda que contiene este puzzle
- ASSERT_MY_PARENT_ID - [71] - (71 parentID): Este gasto es sólo válido si la información de la moneda principal presentada es exactamente la misma que la información de la moneda principal de la moneda que contiene este puzzle

- ASSERT_MY_PUZZLE_HASH - [72] - (72 puzzlehash): Este gasto es sólo válido si el hash de puzzle presentado es exactamente el mismo que el hash del puzzle de la moneda que contiene este puzzle
- ASSERT_MY_AMOUNT - [73] - (73 amount): Este gasto es sólo válido si la cantidad presentada es exactamente la misma que la cantidad de la moneda que contiene este puzzle
- ASSERT_SECONDS_RELATIVE - [80] - (80 seconds): Este gasto es sólo válido si ha pasado el tiempo dado desde que se creó esta moneda
- ASSERT_SECONDS_ABSOLUTE - [81] - (81 time): Este gasto es sólo válido si el lapso de tiempo en este bloque es mayor que el lapso de tiempo especificado
- ASSERT_HEIGHT_RELATIVE - [82] - (82 block_age): Este gasto es sólo válido si el número especificado de bloques ha pasado desde que se creó esta moneda
- ASSERT_HEIGHT_ABSOLUTE - [83] - (83 block_height): Este gasto es sólo válido si se ha alcanzado la altura de bloque dada

Las condiciones dadas son retornadas como una lista de listas:

```
((51 0xabcd1234 200) (50 0x1234abcd) (53 0xdeadbeef))
```

Recuerde: esto es lo que un puzzle debe evaluar cuando se le presenta una solución para que un nodo completo pueda entenderla. Acontinuación vamos a ver ejemplos de puzzle y sus soluciones.

3.4.1. Moneda bloqueada con contraseña

Esta moneda sólo puede ser gastada por personas que conocen la contraseña. Para implementar debemos tener el hash de la contraseña usado en el puzzle. Para el sieguiente ejemplo la contraseña es “hello” que tiene el hash:

0x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

```
$ brun '(i (= (sha256 2) (q . 0
    ↪ x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824)) (c (q . 51) (
    ↪ c 5 (c (q . 100) ()))) (q . "wrong password"))'
'("hello" 0xdeadbeef)'
```

- El programa obtiene el hash de la contraseña dada por el usuario $(sha256 \quad 2)$

```
$ brun '(sha256 2)' '("hello" 0xdeadbeef)'
0x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

- Después se compara el hash que tiene el programa con el hash obtenido de la contraseña del usuario

```
$ brun '(= (sha256 2) (q . 0
    ↪ x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824))''("hello
    ↪ " 0xdeadbeef)'
1
```

- Si, los hash son iguales $(i(= (sha256 \quad 2)$... es decir “1 (verdadero)”. Entonces, el programa regresa $(c(q,51)(c \quad 5(c(q,100)())))$ caso contrario $(q.”wrongpassword”)$
- $(c(q,51)(c \quad 5(c(q,100)())))$ retorna la concatenación del número 100 (saldo a gastar), billetera dada por el usuario (quinto valor en el árbol binario) y el número 51 (debido a la condición CREATE_COIN).

```
$ brun '(c (q . 51) (c 5 (c (q . 100) ())))' '("hello" 0xdeadbeef)'
(51 0xdeadbeef 100)
```

- Con la finalidad de invalidar un gasto se debe generar una excepción. Por tanto se añade el valor de "x" la cuál evita que la moneda se gaste cuando la contraseña es incorrecta. Por tanto, cambiamos $(q.\text{“}wrongpassword\text{”})$ por $(q.(x(q.\text{“}wrongpassword\text{”})))$
- Con la finalidad de evaluar rama por rama añadimos el operador "a". Obteniendo como resultado:

```
$ brun '(a (i (= (sha256 2) (q . 0
    ↪ x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824)) (q . (c
    ↪ (c (q . 51) (c 5 (c (q . 100) ())))) ())) (q . (x (q . "wrong password"))))
    ↪ 1)' '("let me in" 0xdeadbeef)'
FAIL: clvm raise ("wrong password")
```

```
$ brun '(a (i (= (sha256 2)(q.0
    ↪ x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824))(q . (c (
    ↪ c (q . 51) (c 5 (c (q . 100) ())))) ())) (q . (x (q ."wrong password")))) 1)'
    ↪  '("hello" 0xdeadbeef)'
((51 0xdeadbeef 100))
```

3.4.2. Condiciones del lado del Puzzle vs del lado de la Solución

Se puede crear un código que sólo dependa del puzzle y no de la solución del mismo (datos pasados por el usuario). Como se muestra en el siguiente ejemplo:

```
$ brun '(q . ((51 0x365bdd80582fcc2e4868076ab9f24b482a1f83f6d88fd795c362c43544380e7a
    ↪ 100)))' '("it doesn't matter what we put here")'
((51 0x365bdd80582fcc2e4868076ab9f24b482a1f83f6d88fd795c362c43544380e7a 100))
```

De la misma forma se puede hacer que sólo dependa de la solución, dándole el valor de 1 al puzzle.

```
$ brun '1' '((51 0xf00dbabe 75) (51 0xfadeddab 15) (51 0x1234abcd 10))'
((51 0xf00dbabe 75) (51 0xfadeddab 15) (51 0x1234abcd 10))
```

Estos ejemplos se han mostrado con la finalidad de demostrar que las condiciones pueden venir desde el puzzle o la solución.

3.4.3. Firmas BLS

Se puede tomar una firma (en la que se puede confiar o no) y combinarla con otra, para producir una única firma que verifiquen la combinación de todos los mensajes. Por ejemplo: un puzzle puede devolver un cojunto de condiciones múltiples AGG_SIG:

```
((AGG_SIG_UNSAFE <pubkey A> <msg A>) (AGG_SIG_UNSAFE <pubkey B> <msg B>))
```

El nodo que procesa este gasto buscará una firma adjunta que sea una combinación de la clave pública de la firma A en el mensaje A con la clave pública de la firma B en el mensaje B. El gasto no se hará a menos que exista exactamente la combinación de las dos firmas.

3.4.4. Moneda Bloqueada con Firma

Para enviar una moneda a una persona es necesario crear un puzzle que requiera la firma del destinatario y luego les permita devolver cualquier condición. Esto significa que nadie más puede gastar las monedas. Entonces, podemos construir la siguiente transacción en donde AGG_SIG_ME es 50 y la clave pública del destinatario es 0xfadedcab.

```
$ brun '(c (c (q . 50) (c (q . 0xfadedcab) (c (sha256 2) ()))) 3)' '("hello" (51 0
    ↪ xcafef00d 200))'
((50 0xfadedcab 0x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824) (51
    ↪ 0xcafef00d 200))
```

3.5. Billeteras

Una billetera es un software que tiene varias características las mismas que facilitan al usuario la interacción con las monedas.

- Dan seguimiento a las claves públicas y privadas
- Pueden generar puzzles y soluciones
- Pueden realizar firmas con sus llaves
- Pueden identificar y recordar las monedas que posee un usuario (ya que reconocen sus pubkeys)
- Permiten realizar gastos de monedas por medio de un standard

Para enviar una moneda primeramente se solicita una dirección (hash codificado en bech32m). Luego se destruye la misma y crea otra con el puzzle del destinatario para que el mismo pueda identificarla y gastarla.

3.6. Hacer cambios

Si una billetera gasta menos monedas que las que tiene, se puede crear una nueva moneda (ó varias) con el valor que no se gasto y bloquear la misma con el puzzle standard. No se puede crear dos monedas con el mismo valor y puzzle del mismo padre debido a que existirá una colisión del ID y el gasto será rechazado.

3.7. SpendBundles

Permite hacer una transacción con varias monedas a la vez.

3.8. Pagar y Delegar el Puzzle

El destinatario puede ejecutar su propio programa como parte de la generación de la solución, también puede firmar el puzzle y dejar que otra persona proporcione la solución.

```
1 $ brun '(c (c (q . 50) (c (q . 0xfadedcab) (c (sha256 2) ())))) (a 5 11))' '("hello" (q .
      ↪  ((51 0xdeadbeef 50) (51 0xf00dbabe 50))) ())'
2
3 ((50 0xfadedcab 0x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824) (51
      ↪ 0xdeadbeef 50) (51 0xf00dbabe 50))
```

3.9. Conclusiones

Una moneda sólo puede ser gastada cuando es firmada por su propietario. Las billeteras generan e interpretan además de administrar puzzles y monedas.

4. Lenguaje de Alto Nivel, Compilador y Funciones

4.1. CLVM vs Chialisp

CLVM es lo que se serializa y almacena directamente en la cadena de bloques y es una cuestión de consenso. Nunca se puede cambiar.

Normalmente, escribiremos un lenguaje de nivel superior llamado Chialisp en lugar de CLVM. Chialisp se compila en CLVM. Chialisp comparte muchos de los fundamentos de cómo funcionan los programas con CLVM, pero también incluye algunas funciones útiles para facilitar la escritura de programas grandes.

4.2. Run

Chialisp se corre usando el comando "run" mientras que CLVM usa "brun"

```
$ brun '(+ 200 200)'
FAIL: first of non-cons ()
$ run '(+ 200 200)'
400
```

Run también nos da acceso a una serie de operadores convenientes de alto nivel, que cubriremos ahora:

- list: toma cualquier número de parámetros y los devuelve dentro de una lista. Esto nos ahorra tener que crear manualmente llamadas anidadas $(c(A)(c(B)(q())))$, que pueden complicarse rápidamente.

```
$ run '(list 100 "test" 0xdeadbeef)'
(100 "test" 0xdeadbeef)
```

- if: pone automáticamente nuestra declaración i en el formulario de evaluación diferida para que no tengamos que preocuparnos por la ruta del código no utilizado que se está evaluando. Es decir "if" combina el uso de "i" e "a" en CLVM.

```
$ run '(if 1 (q . "success") (x))' '(100)'
"success"
$ run '(if 0 (q . "success") (x))' '(100)'
FAIL: clvm raise ()
```

- qq,unquote,@: "qq" indica que vamos a tomar argumentos dentro de código a usar, "unquote" indica donde vamos a usar los argumentos y "@" representa los argumentos.

```
$ run '(qq (unquote @))' '(0xdeadbeef 0xdeadbeef2)'
(0xdeadbeef 0x0deadbeef2)
$ run '(qq (c (c (q . 50) (c (q (unquote (f @))) (c (sha256 2) ()))) (a 5 11)))'
    ↪ '(0xdeadbeef)'
(c (c (q . 50) (c (q . 0xdeadbeef) (c (sha256 2) ()))) (a 5 11))
```

- (mod A B): Permite compilar en CLVM código de Chialip esto es usado debido a que las monedas inteligentes corren en lenguaje de bajo nivel. Entonces, primero obtemos el código que puede ejecutar CLVM y después al código obtenido le pasamos los argumentos deseados.

```
$ run '(mod (arg_one arg_two) (list arg_one))'
(c 2 ())
$ brun '(c 2 ())' '(100 200 300)'
(100)
```

4.3. Funciones, Macros y Constantes

En el lenguaje de alto nivel se pueden definir funciones, macros y constantes. Usando la siguiente estructura ($defun$, $defun-inline$, $defmacro$ y $defconstant$):

```
(mod (arg_one arg_two)
  (defconstant const_name value)
  (defun function_name (parameter_one parameter_two) *function_code*)
  (defun another_function (param_one param_two param_three) *function_code*)
  (defun-inline utility_function (param_one param_two) *function_code*)
  (defmacro macro_name (param_one param_two) *macro_code*)

  (main *program*)
)
```

- Las funciones pueden hacer referencia a sí mismas en su código, pero las macros y las funciones en línea ($defun - inline$) no pueden, ya que se insertan en tiempo de compilación.
- Tanto las funciones como las macros pueden hacer referencia a otras funciones, macros y constantes.
- Las macros que hacen referencia a sus parámetros deben estar entre comillas con los parámetros sin comillas.
- Tenga cuidado con los bucles infinitos en macros que hacen referencia a otras macros.
- Los comentarios se pueden escribir con punto y coma.
- Las funciones en línea ($defun - inline$) son regularmente más costosas computacionalmente que las funciones regulares ($defun$), excepto cuando se reutilizan argumentos calculados: (defun-inline foo (X) (+ X X)) (foo (* 200 300)) realizará la costosa multiplicación dos veces.

4.4. Ejemplos

4.4.1. Factorial

```
(mod (arg_one)
  ; function definitions
  (defun factorial (input)
    (if (= input 1) 1 (* (factorial (- input 1)) input))
  )

  ; main
  (factorial arg_one)
)
```

```
$ run factorial.clvm
(a (q 2 2 (c 2 (c 5 ()))) (c (q 2 (i (= 5 (q . 1)) (q 1 . 1) (q 18 (a 2 (c 2 (c (- 5 (q
    ↪ . 1)) ()))) 5)) 1) 1))
$ brun '(a (q 2 2 (c 2 (c 5 ()))) (c (q 2 (i (= 5 (q . 1)) (q 1 . 1) (q 18 (a 2 (c 2 (c
    ↪ (- 5 (q . 1)) ()))) 5)) 1) 1))' '(5)'
120
```

4.4.2. Cuadrado de Lista

```
(mod (args)

  (defmacro square (input)
    (qq (* (unquote input) (unquote input)))
  )

  (defun sqre_list (my_list)
    (if my_list
      (c (square (f my_list)) (sqre_list (r my_list)))
      my_list
    )
  )

  (sqre_list args)
)
```

```
$ run square_list.clvm
(a (q 2 2 (c 2 (c 5 ()))) (c (q 2 (i 5 (q 4 (* 9 9) (a 2 (c 2 (c 13 ())))) (q . 5)) 1)
    ↪ 1))
$ brun '(a (q 2 2 (c 2 (c 5 ()))) (c (q 2 (i 5 (q 4 (* 9 9) (a 2 (c 2 (c 13 ())))) (q .
    ↪ 5)) 1) 1))' '((10 9 8 7))'
(100 81 64 49)
```

5. Funciones de Chialisp

5.1. include

Sirve para importar funcionalidades sin tener que copiar y pegar código entre archivos. Cuando ejecute *main.clvm* con *run*, asegúrese de usar la opción $-i$ para especificar en qué directorios buscar archivos que se pueden incluir. Por ejemplo:

```
$ run -i ./libraries/chialisp/ main.clvm
(a (q 4 (c 4 (c 5 (c 11 ()))) (c (c 6 (c 23 (c 47 ()))) ())) (c (q 50 . 51) 1))
```

Los archivos que se van a incluir deben tener el formato de paréntesis del ejemplo "condition_codes.clvm".

```
;; condition_codes.clvm
(
  (defconstant AGG_SIG_ME 50)
  (defconstant CREATE_COIN 51)
)
```

Se debe usar *include*, como muestra el ejemplo del archivo *main.clvm*, para importar parámetros.

```
;;main.clvm
(mod (pubkey msg puzzle_hash amount)

  (include "condition_codes.clvm")

  (list (list AGG_SIG_ME pubkey msg) (list CREATE_COIN puzzle_hash amount))

)
```

El compilador solo incluirá cosas que use, así que no se preocupe por incluir un archivo de biblioteca grande cuando intente optimizar el tamaño de su programa.

5.2. sha256tree1

Cuando los puzzles se procesan, no se serializan simplemente y se pasan a sha256. En su lugar, tomamos el hash del árbol binario. Cuando hacemos un hash comenzamos en las hojas del árbol y subimos, concatenando con un 1 ó 2 para indicar que es un atom ó cons box.

```
(defun sha256tree1
   (TREE)
   (if (l TREE)
       (sha256 2 (sha256tree1 (f TREE)) (sha256tree1 (r TREE)))
       (sha256 1 TREE)
   )
)
```

El sha256tree1 permite verificar puzzles de otras monedas, condensar puzzles para facilitar la firma y hacer condiciones CREATE_COIN que dependan de algunos datos pasados.

5.3. Currying

Es responsable, en casi su totalidad, del almacenamiento del estado de las monedas. La idea es pasar argumentos al puzzle antes de que se cree el hash. El curry se compromete con los valores de la solución para que la persona que resuelve el puzzle no pueda cambiarlos.

```
;; utility function used by curry
(defun fix_curry_args (items core)
 (if items
     (qq (c (q . (unquote (f items))) (unquote (fix_curry_args (r items) core))))
     core
 )
)

; (curry sum (list 50 60)) => returns a function that is like (sum 50 60 ...)
(defun curry (func list_of_args) (qq (a (q . (unquote func)) (unquote (fix_curry_args
    ↪ list_of_args (q . 1))))))
```

Esto será útil cuando desee crear un modelo de puzzle específico, pero con diferentes valores en ciertos parámetros cada vez que se lo cree. El código anterior se reduce a algo como esto:

```
(a func (c curry_arg_1 (c curry_arg_2 1)))
```

También puede realizar la operación inversa. Dado un programa, puede deshacer la lista de argumentos, con un simple:

```
(f (r (r curried_func)))
; (c curry_arg_1 (c curry_arg_2 1))
```

Del ejemplo de moneda bloqueada con contraseña en Chialisp tendríamos:

```
(mod (password new_puzhash amount)
  (defconstant CREATE_COIN 51)

  (defun check_password (password new_puzhash amount)

    (if (= (sha256 password) (q . 0
        ↪ x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824))
      (list (list CREATE_COIN new_puzhash amount))
      (x)
    )
  )

  ; main
  (check_password password new_puzhash amount)
)
```

Puede ver que el hash de la contraseña está integrado en el puzzle. Esto significa que cada vez que desee bloquear una moneda con una nueva contraseña, debe volver a crear el archivo que contiene la fuente del código.

```
(mod (PASSWORD_HASH password new_puzhash amount)
  (defconstant CREATE_COIN 51)

  (defun check_password (PASSWORD_HASH password new_puzhash amount)

    (if (= (sha256 password) PASSWORD_HASH)
      (list (list CREATE_COIN new_puzhash amount))
      (x)
    )
  )

```

```
  ; main
  (check_password PASSWORD_HASH password new_puzhash amount)
)
```

Sin embargo, ahora tenemos el problema de que cualquiera puede pasar la combinación de hashes que desee y desbloquear esta moneda, por este motivo, cuando creamos esta moneda necesitamos que se comprometa el hash de la contraseña. Entonces, antes de determinar el hash del puzzle de la moneda que vamos a crear necesitamos usar el curry como se muestra a continuación.

```
; curry_password_coin.clvm
(mod (password_hash password_coin_mod)
  (include "curry.clvm") ; From above

  (curry password_coin_mod (list password_hash))
)
```

Si compilamos esta función y le pasamos parámetros como este:

```
brun <curry password coin mod> '((q . 0xcafef00d) (q . <password coin mod>))'
```

Recibiremos un puzzle similar al modulo de la contraseña de la moneda pero que incluye el hash que hemos pasado. Ahora se puede ejecutar un nuevo curring con otro hash diferente obtenido de una nueva contraseña para generar un nuevo puzzle cada vez. Esto fue realizado en python pero a veces necesitaremos que ocurra en otro entorno como lo veremos en la siguiente sub-sección.

5.4. Puzzles externos e internos

Una de las características más poderosas de Chialisp, es la capacidad de tener un puzzle externo que "envuelve" un puzzle interno. Este concepto es extremadamente útil porque permite que una moneda retenga toda su funcionalidad y programabilidad estándar dentro del puzzle interno, pero que esté sujeta a un conjunto adicional de reglas por el puzzle externo.

Para este ejemplo, vamos a continuar con nuestro bloqueo de contraseña, pero esta vez vamos a requerir que cada vez que se gaste la moneda, se requiera establecer una nueva contraseña.

```
(mod (
    MOD_HASH        ;; curried in
    PASSWORD_HASH   ;; curried in
    INNER_PUZZLE    ;; curried in
    inner_solution
    password
    new_password_hash
  )

  (include "condition_codes.clvm")
  (include "sha256tree1.clvm")
  (include "curry-and-treehash.clvm")

  (defun pw-puzzle-hash (MOD_HASH mod_hash_hash new_password_hash_hash inner_puzzle_hash
      ↪ )
    (puzzle-hash-of-curried-function
      MOD_HASH
      inner_puzzle_hash new_password_hash_hash mod_hash_hash ; parameters must be
          ↪ passed in reverse order
    )
   )

  ;; tweak 'CREATE_COIN' condition by wrapping the puzzle hash, forcing it to be a
      ↪ password locked coin
  (defun-inline morph-condition (condition new_password_hash MOD_HASH)
   (if (= (f condition) CREATE_COIN)
     (list CREATE_COIN
```

```
      (pw-puzzle-hash MOD_HASH (sha256tree1 MOD_HASH) (sha256tree1 new_password_hash) (
          ↪ f (r condition)))
      (f (r (r condition)))
    )
    condition
  )
 )

 ;; tweak all 'CREATE_COIN' conditions, enforcing created coins to be locked by
     ↪ passwords
 (defun morph-conditions (conditions new_password_hash MOD_HASH)
  (if conditions
    (c
      (morph-condition (f conditions) new_password_hash MOD_HASH)
      (morph-conditions (r conditions) new_password_hash MOD_HASH)
    )
    ()
  )
 )

 ; main
 (if (= (sha256 password) PASSWORD_HASH)
   (morph-conditions (a INNER_PUZZLE inner_solution) new_password_hash MOD_HASH)
   (x "wrong password")
 )

)
```

Puede notar que importamos una nueva biblioteca llamada curry-and-treehash. Cuando crea este puzzle por primera vez, necesita curry en 3 cosas:

- MOD_HASH: que es el hash del árbol del código sin argumentos curry.
- PASSWORD_HASH: es el hash de la contraseña que desbloqueará esta moneda
- INNER_PUZZLE: que es un puzzle completamente separado que tendrá sus propias reglas sobre cómo se puede gastar la moneda.

Los puzzles de Chialisp tienden a leerse de abajo hacia arriba, así que comencemos con este fragmento:

```
; main
(if (= (sha256 password) PASSWORD_HASH)
  (morph-conditions (a INNER_PUZZLE inner_solution) new_password_hash MOD_HASH)
  (x "wrong password")
)
```

Todo lo que está sucediendo aquí es que nos estamos asegurando de que la contraseña sea correcta y, si lo es, vamos a ejecutar el curry en $INNER\_PUZZLE$ con el pasado en $inner\_solution$

Esto devolverá una lista de condiciones que pasaremos a la siguiente función junto con el nuevo hash de contraseña y MOD_HASH.

```
;; tweak all 'CREATE_COIN' conditions, enforcing created coins to be locked by passwords
(defun morph-conditions (conditions new_password_hash MOD_HASH)
 (if conditions
   (c
     (morph-condition (f conditions) new_password_hash MOD_HASH)
     (morph-conditions (r conditions) new_password_hash MOD_HASH)
   )
   ()
 )
)
```

La recursividad es la base de Chialisp y funciones como estas aparecen con mucha frecuencia al escribirlo. Para recorrer la lista de condiciones, primero verificamos si todavía quedan elementos (recuerde que una lista vacía () o nil se evalúa como falso). Luego, transformamos la primera condición y la concatenamos con la salida recursiva del resto de la lista. Al final, tendremos la misma lista de elementos en el mismo orden, pero todos habrán pasado por $morph - condition$.

```
;; tweak 'CREATE_COIN' condition by wrapping the puzzle hash, forcing it to be a
    ↪ password locked coin
(defun-inline morph-condition (condition new_password_hash MOD_HASH)
 (if (= (f condition) CREATE_COIN)
   (list CREATE_COIN
     (pw-puzzle-hash MOD_HASH (sha256tree1 MOD_HASH) (sha256tree1 new_password_hash) (f
         ↪ (r condition)))
     (f (r (r condition)))
   )
   condition
 )
)
```

Primero comprobamos si el código de operación (primer elemento de la lista) es CREATE_COIN. Si no es así, devuelva la condición como de costumbre. Si es así, devuelve una condición que es casi exactamente la misma, excepto que estamos pasando el hash del puzzle a una función que lo modificará:

```
(defun pw-puzzle-hash (MOD_HASH mod_hash_hash new_password_hash_hash inner_puzzle_hasr
   (puzzle-hash-of-curried-function
     MOD_HASH
     inner_puzzle_hash new_password_hash_hash mod_hash_hash ; parameters must be passed
         ↪ in reverse order
   )
)
```

Como no conocemos el puzzle interno, solo su hash, es imposible convertirlo directamente en el siguiente puzzle que queremos crear. Además, lo único que nos importa es generar el hash correcto del actual puzzle para el siguiente puzzle, y tenemos los hash del árbol tanto para este módulo como para el puzzle interno. Entonces, podemos usar puzzle-hash-of-curried-function que nos permite crear el hash de una función dada:

- Se creará el hash del puzzle de esa función.
- Se creará el hash del puzzle de todos sus argumentos en orden inverso como si fueran parte de un hash de árbol.

Esto significa que se espera que los argumentos que son átomos y números estén en forma de hash de árbol, con un prefijo 1 como $(sha256(q,1)my - argument - value)$ y la salida de $sha256tree$ es adecuada para cualquier cosa que involucre celdas *cons*. Sería posible que $puzzle - hash - of - curried - function$ adivinara estos valores si tomara los valores de los parámetros en sí mismos, pero eso podría requerir volver a calcular los hash costosos.

Otros detalles de implementación de esta biblioteca son un poco extensos en esta parte del tutorial pero, en esencia, nos permite reanudar un hash de árbol que hemos completado excepto el último paso.

Cuando se crea esta moneda, solo se puede gastar con una contraseña que se transfiera al curry en $PASSWORD\_HASH$. El puzzle interior puede ser cualquier cosa que desee, incluidos otros puzzles externos que tienen sus propios puzzles internos. Cualquier moneda que se cree como resultado de ese puzzle interno será "envuelto" por este mismo puzzle externo asegurando que cada hijo de esta moneda esté bloqueado por una contraseña para siempre.

Creamos una moneda simple, pero puedes ver el potencial de esto. Puede hacer cumplir un conjunto de reglas no solo en una moneda que bloquee, sino en cada moneda descendiente. En el ecosistema de Chialisp, todas las monedas inteligentes son interoperables entre sí a menos que uno de los puzzle de la pila especifique lo contrario. Las posibilidades son infinitas y representan la gran capacidad de programación que permite Chialisp para las monedas.

6. Transacción Estandard

6.1. Pagar "Puzzle Delegado" o "Puzle Oculto"

6.1.1.

El puzzle Delegado permite al solucionador pasar un puzzle y una solución para crear sus propias condiciones para la salida. Sin embargo, también queremos la capacidad de comprometernos previamente con un puzzle sin revelarlo, y dejar que cualquiera con el conocimiento del puzzle "oculto" lo gaste. Pero si realizamos el caso de gasto delegado, tendremos que revelar el rompecabezas completo, incluido el curry en el rompecabezas oculto, y ya no estará oculto. Ya no podemos guardar una moneda con el mismo rompecabezas, o de lo contrario la gente podrá decir que el hash del rompecabezas es el mismo y gastarlo sin nuestro consentimiento. Es posible que nuestro gasto delegado ni siquiera llegue a la red; un nodo malicioso puede simplemente denegar nuestra transacción después de verla y luego publicar el caso de gasto oculto por su cuenta.

Podemos intentar resolver esto mediante el hash del puzzle oculto. Esto tiene algunos problemas similares. Si realiza un gasto, las personas pueden ver cualquier hash de rompecabezas idéntico más tarde y gastarlos sin su consentimiento. Además, muchas personas pueden intentar usar el mismo rompecabezas oculto. Si alguien lo revela, todas las monedas encerradas con ese mismo rompecabezas también se pueden identificar y gastar. Necesitamos que el rompecabezas esté oculto, pero también tener algo de entropía que lo mantenga exclusivo para nosotros. La solución que utiliza la transacción estándar es:

- Puzzle Oculto
- Clave pública que puede firmar para el caso de gastos delegados

```
1 synthetic_offset == sha256(hidden_puzzle_hash + original_public_key)
```

Luego calculamos la clave pública de esta nueva clave privada y la agregamos a nuestra clave pública original existente.

```
1 synthentic_public_key == original_public_key + synthetic_offset_pubkey
```

Si el solucionador puede revelar correctamente AMBOS el rompecabezas oculto y la clave pública original, entonces nuestro rompecabezas puede derivar la clave pública sintética y asegurarse de que coincida con la que tiene curry.

Quizás se pregunte por qué agregamos la clave pública de nuestra clave privada derivada a la clave pública original cuando ya es parte de la derivación. Esto se debe a que también usamos la clave pública sintética para firmar nuestros gastos delegados. Cuando agrega dos claves públicas, la clave privada para la clave pública resultante es la suma de las claves privadas originales. Si no agregamos la clave pública original, cualquiera que conozca el acertijo oculto podría derivar la clave privada sintética y luego realizar gastos delegados. Agregar la clave pública original asegura que todavía haya un componente secreto de la clave privada sintética, aunque se pueda conocer la mitad.

Esta técnica también es ingeniosa porque nos permite esconder el rompecabezas oculto en una pieza de información que ya era necesaria para el gasto delegado. Es imposible adivinar cuál es el rompecabezas oculto, ¡incluso si es un rompecabezas oculto estándar! Incluso es difícil saber si hay un rompecabezas oculto. Esto también puede contribuir a la privacidad. Por ejemplo, si dos partes acuerdan encerrar juntas algunas monedas con un rompecabezas oculto, puede compartir claves públicas y verificar esa información en la cadena de bloques sin revelar nada a la red. Además, se podría gastar la moneda si una de las dos partes es deshonesta, se puede delegar el gasto de monedas.Además, es imposible decirle que son gastos normales del día. Veremos el código en un momento, pero aquí hay algunos términos que debe conocer antes de mirarlo:

- hidden puzzle: un "puzzle oculto"que se puede revelar y utilizar como una forma alternativa de desbloquear los fondos subyacentes
- synthetic key offset: una clave privada generada criptográficamente usando el rompecabezas oculto y $original\_public\_key$ como entradas

- synthetic public key: la clave pública (curry in) que es la suma del $original\_public\_key$ y la clave pública correspondiente a $Synthetic\_key\_offset$
- original public key: una clave pública, donde el conocimiento de la clave privada correspondiente representa la propiedad de la moneda
- delegated puzzle: un rompecabezas delegado, como en "graftroot", que debería devolver las condiciones deseadas.
- solution: la solución al rompecabezas delegado u oculto.

6.2. Chialisp

Aquí está la fuente completa y luego la desglosaremos:

```
(mod

    (SYNTHETIC_PUBLIC_KEY original_public_key delegated_puzzle solution)

    ; "assert" is a macro that wraps repeated instances of "if"
    ; usage: (assert A0 A1 ... An R)
    ; all of A0, A1, ... An must evaluate to non-null, or an exception is raised
    ; return the last item (if we get that far)

    (defmacro assert (items)
        (if (r items)
            (list if (f items) (c assert (r items)) (q . (x)))
            (f items)
        )
    )

    (include condition_codes.clvm)
    (include sha256tree1.clvm)

    ; "is_hidden_puzzle_correct" returns true iff the hidden puzzle is correctly encoded

    (defun-inline is_hidden_puzzle_correct (SYNTHETIC_PUBLIC_KEY original_public_key
        ↪ delegated_puzzle)
      (=
          SYNTHETIC_PUBLIC_KEY
          (point_add
              original_public_key
              (pubkey_for_exp (sha256 original_public_key (sha256tree1 delegated_puzzle)
                  ↪ ))
          )
      )
    )

    ; "possibly_prepend_aggsig" is the main entry point

    (defun-inline possibly_prepend_aggsig (SYNTHETIC_PUBLIC_KEY original_public_key
        ↪ delegated_puzzle conditions)
      (if original_public_key
          (assert
              (is_hidden_puzzle_correct SYNTHETIC_PUBLIC_KEY original_public_key
                  ↪ delegated_puzzle)
              conditions
          )
          (c (list AGG_SIG_ME SYNTHETIC_PUBLIC_KEY (sha256tree1 delegated_puzzle))
              ↪ conditions)
```

```
      )
    )

    ; main entry point

    (possibly_prepend_aggsig
        SYNTHETIC_PUBLIC_KEY original_public_key delegated_puzzle
        (a delegated_puzzle solution))
)
```

Primero, hablemos de los argumentos:

```
(SYNTHETIC_PUBLIC_KEY original_public_key delegated_puzzle solution)
```

Todos estos términos se definen anteriormente. Cuando resolvimos este acertijo:

- *SYNTHETIC_PUBLIC_KEY*: es un curry.
- Pasamos *original_public_key* si es el gasto oculto o () si es el gasto delegado.
- *delegated_puzzle* es el rompecabezas oculto si es el gasto oculto, ó el rompecabezas delegado si es el gasto delegado.
- *solution* es la solución a todo lo que se pasa a *delegated_puzzle*.

Al igual que con la mayoría de los programas de Chialisp, comenzaremos a ver la implementación desde abajo:

```
(possibly_prepend_aggsig
    SYNTHETIC_PUBLIC_KEY original_public_key delegated_puzzle
    (a delegated_puzzle solution))
```

Pasamos argumentos a *possibly_prepend_aggsig*. Lo único que hay que tener en cuenta es que estamos evaluando el rompecabezas delegado con la solución (antes de pasarlo). Esto dará como resultado una lista de condiciones que generaremos siempre que el resto del rompecabezas se verifique.

```
(defun-inline possibly_prepend_aggsig (SYNTHETIC_PUBLIC_KEY original_public_key
    ↪ delegated_puzzle conditions)
  (if original_public_key
      (assert
          (is_hidden_puzzle_correct SYNTHETIC_PUBLIC_KEY original_public_key
              ↪ delegated_puzzle) ; hidden case
          conditions
      )
      (c (list AGG_SIG_ME SYNTHETIC_PUBLIC_KEY (sha256tree1 delegated_puzzle))
          ↪ conditions) ; delegated case
  )
)
```

Esta función es la lógica del flujo de control principal que determina si estamos haciendo el gasto "oculto" o "delegado". La primera línea solo verifica si se pasó una *original_public_key*. En el gasto delegado, pasamos () para ese argumento, y dado que se evalúa como falso, funciona muy bien como un interruptor para determinar lo que estamos haciendo.

Si el gasto es el gasto oculto, pasamos la mayoría de nuestros parámetros a *is_hidden_puzzle_correct* y, siempre que no falle, simplemente devolvemos las condiciones que se nos den. Si el gasto es el gasto delegado, anteponemos un requisito de firma del curry en clave pública en el hash del rompecabezas delegado.

```
(defun-inline is_hidden_puzzle_correct (SYNTHETIC_PUBLIC_KEY original_public_key
    ↪ delegated_puzzle)
  (=
      SYNTHETIC_PUBLIC_KEY
```

```
      (point_add
          original_public_key
          (pubkey_for_exp (sha256 original_public_key (sha256tree1 delegated_puzzle)))
      )
  )
)
```

Esta es la representación de Chialisp de lo que se explicó en la sección anterior. Una clave privada es de 32 bytes, por lo que vamos a usar $sha256$ (cuya salida es de 32 bytes) para asegurarnos de que nuestra clave privada se deriva de $original\_public\_key$ y el hash del rompecabezas oculto.

6.3. Conclusion

Este rompecabezas asegura casi todas las monedas de la red Chia. Cuando usa el software de billetera Chia Network, está rastreando la cadena de bloques en busca de monedas bloqueadas con este formato específico. $SYNTHETIC\_PUBLIC\_KEY$ está buscando un puzzle oculto inválido que fallará automáticamente Esto se debe a que la mayoría de los usuarios de Chia no necesitan la funcionalidad de rompecabezas ocultos para las transacciones. Este puzzle también lo convierte en un fantástico rompecabezas interno de cualquier moneda inteligente que pueda escribir.

7. Ciclo de Vida de una Moneda

Quizás se pregunte cómo se guarda una moneda, dónde se almacena, cómo y cuándo se gasta y quién puede gastarla. Echemos un vistazo a los pasos por los que pasa una moneda desde su creación hasta su destrucción.

7.1. Modelo de Conjunto de Monedas

El modelo de Chia de cómo se almacenan las monedas se llama modelo de conjunto de monedas y se basa en el modelo UTXO de Bitcoin. La idea es simple, cada nodo completo tiene una base de datos de registros de monedas:

```
class Coin:
  parent_coin_info: bytes32
  puzzle_hash: bytes32
  amount: uint64

class CoinRecord:
  coin: Coin
  confirmed_block_index: uint32
  spent_block_index: uint32
  spent: boolean
  coinbase: boolean
  timestamp: uint64
```

Tenga en cuenta que el objeto Coin es parte del formato blockchain y no se puede cambiar, mientras que el objeto CoinRecord es parte del nodo completo y se puede modificar mediante implementaciones alternativas.

Esta base de datos se genera observando la cadena de bloques desde la altura del bloque cero y procesando todas las transacciones hasta que esté sincronizada con el pico actual. Cuando un nodo completo procesa un gasto, primero se asegura de que la moneda que se gasta exista en su base de datos, ejecuta y valida las condiciones de Chialisp que se producen, marca la moneda que se estaba gastando como gastada y agrega las monedas nuevas que se crearon como resultado.

Este enfoque hace que la base de datos sea muy liviana a medida que crece la cadena de bloques, pero aún permite la complejidad y las monedas inteligentes a través de Chialisp, que está comprometido con el hash del rompecabezas. También es una distinción importante de algunas otras cadenas de bloques en que todo el estado de Chialisp se almacena en las monedas. No existe una forma especial de almacenar datos para que las personas puedan acceder a ellos; la mayor

parte de esa funcionalidad proviene de encontrar formas de revelar la información adecuada en las revelaciones de acertijos y de comprometerse con hashes predecibles de acertijos.

7.2. Farmeo de Recompensas

Como probablemente sepa, los agricultores crean la totalidad del nuevo valor en Chia. Cada 18,75 segundos aproximadamente, aparece un nuevo bloque que permite a un granjero crear una moneda de la nada. Aquí hay dos recompensas agrícolas reales que se generaron en el bloque 10,000 de la cadena de bloques:

```
[
  {
    "amount" : 1750000000000,
    "parent_coin_info" : "0
        ↪ xccd5bb71183532bff220ba46c268991a00000000000000000000000000000270b",
    "puzzle_hash" : "0x0b42a11b76d276f191026ae1a01c711cc0637e63d8ce0c2f62d6d079cc974920"
  },
  {
    "amount" : 250000000000,
    "parent_coin_info" : "0
        ↪ x3ff07eb358e8255a65c30a2dce0e5fbb00000000000000000000000000000270b",
    "puzzle_hash" : "0x0b42a11b76d276f191026ae1a01c711cc0637e63d8ce0c2f62d6d079cc974920"
  }
]
```

Tenga en cuenta que $parent\_coin\_info$ para ambos es un poco extraño. Debido a que se crean de la nada, a las monedas se les asigna un valor especial como moneda principal: la mitad del "desafío de génesis", algunos ceros de relleno y el índice del bloque en el que se cultivan.

```
pool_parent_id = bytes32(genesis_challenge[:16] + block_height.to_bytes(16, "big"))
farmer_parent_id = bytes32(genesis_challenge[16:] + block_height.to_bytes(16, "big"))
```

Esta información no suele ser muy relevante, pero se utiliza. Por ejemplo, en los puzzles de agrupación, la moneda de recompensa debe hacer un anuncio a medida que se reclama y otro puzzle debe afirmar ese anuncio. También es importante señalar que dado que el desafío de génesis es parte de la información de los padres, y dado que el desafío de génesis es diferente entre la red principal y cada red de prueba, es casi imposible terminar con dos ID de moneda idénticos. Esto es, en parte, para evitar ataques de reproducción de firmas: las monedas que están firmadas en testnet no podrán usar la misma firma en mainnet.

7.3. Transacción (Spend Bundles)

Muy bien, entonces ha recibido algo de XCH y desea implementar su primera moneda inteligente. Para hacer eso, primero tendrá que crear un paquete de gastos. A esto a veces se le llama coloquialmente una "transacción". Un paquete de gastos es una construcción simple. Es un objeto que contiene exactamente dos cosas: una lista de monedas gastadas y una firma agregada.

```
class SpendBundle:
  coin_solutions: List[CoinSpend]
  aggregated_signature: G2Element
```

El gasto de una moneda contiene exactamente tres cosas: la moneda que está tratando de gastar ($parent\_coin\_info$, $amount$, $puzzle\_hash$), la revelación del rompecabezas (necesita un árbol hash para el hash del rompecabezas) y la solución.

```
class CoinSolution:
  coin: Coin
  puzzle_reveal: SerializedProgram
  solution: SerializedProgram
```

La firma agregada es una firma BLS y debe ser una firma de exactamente los pares de *clave_publica/mensaje* que se generan por las condiciones de los acertijos en el paquete. No puede agregar firmas adicionales ni omitir ninguna. Si no se emiten condiciones AGG_SIG_ME o AGG_SIG_UNSAFE, la firma agregada debe ser una firma vacía que es una ç"seguida de 191 ceros.

Una vez que haya creado un paquete de gastos, puede enviarlo al punto final de $RPC/push\_tx$ en un nodo completo de Chia. Sin embargo, tenga en cuenta algunas cosas:

- A menos que cultive el bloque, no puede garantizar si su paquete de gastos llegará a la red o cuándo lo hará. Si dos gastos dependen el uno del otro de alguna manera, deben estar en el mismo paquete de gastos y vinculados con firmas.
- Cuando utilice bloqueos de tiempo absolutos cortos, asegúrese de dejar un tiempo razonable para que el paquete llegue a la red. A veces, debido a la alta presión de las tarifas, es posible que una transacción de tarifa baja tarde un poco en entrar. Si su gasto depende de tener la moneda disponible en la cadena de bloques antes de que pase el bloqueo de tiempo, no es una garantía, así que asegúrese de irse en una buena cantidad de tiempo por si acaso.
- Técnicamente, nada mantiene juntos los gastos en un paquete de gastos, excepto por la firma agregada y la lógica del anuncio. Si realiza un gasto que no afirma un anuncio de otra moneda en el gasto Y no requiere una firma, ese gasto puede ser excluido del paquete por un nodo o agricultor malintencionado. También tenga en cuenta que un paquete de gastos se puede agregar a otro sin su permiso. De hecho, el código de creación de bloques hace algo similar cuando crea el generador, del que hablaremos pronto.

Recuerde que otros nodos también están manejando este dato, y es mejor asumir que intentarán cambiar todo lo que puedan. Asegúrese de que si algo cambia de una manera que no le gusta, todo el paquete fallará.

7.4. Fees y Mempool

Una vez que cree y envíe un paquete de gastos, su nodo lo validará y luego comenzará a enviar la información a sus pares. Cuando cada nodo recibe una transacción, intenta validarla por sí solo y la rechazará si considera que el gasto no es válido. Es extremadamente importante tener en cuenta que antes de validarlo, el nodo puede ejecutar software que intente cambiarlo. Por eso es muy importante que cree sus paquetes de gastos de manera que cualquier cambio invalide automáticamente la firma o el hash del rompecabezas. El nodo malicioso puede enviar la transacción modificada a otros nodos honestos y, si no es inválida, los nodos honestos la aceptarán.

Una vez que haya validado el paquete de gastos, incluirá la información relevante en su mempool. El mempool es una lista de transacciones a las que un nodo se aferra y espera poner en un bloque. Los elementos del mempool están ordenados por tarifas. Se crea una tarifa cuando la suma de las salidas $CREATE\_COIN$ es menor que la cantidad de la moneda que se está gastando. Esto significa que ningún valor puede ser destruido en el ecosistema de Chia: si gasta una moneda y no crea ninguna moneda nueva, el agricultor puede agregar el valor de la moneda existente a sus recompensas agrícolas.

Cuando el mempool decide qué transacciones colocar en un bloque, busca colocar las transacciones con la relación $tarifa/costomsalta$. Todavía no hemos hablado mucho sobre el costo, pero puede obtener más información al respecto aquí y hablaremos más sobre la optimización en una sección posterior. La idea principal es: si realiza una transacción compleja y difícil de ejecutar con una gran cantidad de datos, probablemente terminará pagando una tarifa más alta para colocarla en un bloque en un período de tiempo razonable.

7.5. Generadores de transacciones

Cuando un agricultor decide incluir transacciones en un bloque, las agregará todas en algo llamado generador de transacciones. Un generador es un programa clvm que genera la información en el paquete de gastos. Dará como resultado una lista de listas que contienen:

- La identificación principal de la moneda que se está gastando
- El rompecabezas y la solución revelan

- El monto de la moneda

Luego, ese programa se ejecuta a través de otro programa Chialisp que es parte del consenso y se usa para validar cada bloque que recibe un nodo. Ese programa analiza el generador en busca de revelaciones de puzzles y soluciones, y devuelve una lista total de condiciones para cada transacción en el bloque.

Este es un concepto importante: todas las transacciones de un bloque se ejecutan en paralelo. El hecho de que todas estas transacciones se evalúen como un gran programa proporciona algunos beneficios valiosos:

Los agricultores no pueden ejecutar una transacción en el momento del farming porque no hay un orden para las transacciones. Pueden patear transacciones y gastar las monedas que las transacciones pateadas planeaban gastar si lo deseaban, pero luego no pueden volver a agregar la transacción al conjunto porque la moneda que estaba gastando ya está gastada.

Además, dado que todas las transacciones se devuelven como una única lista de condiciones, sus condiciones pueden depender potencialmente de anuncios de otras transacciones. Si sabe que con frecuencia ocurre un gasto que crea un anuncio, ¡sus acertijos pueden afirmar ese anuncio sin ser el propietario de esa transacción! Esta es una implementación potencial para oráculos en la Red de Chia. An entity can create a singleton, or a coin that there is verifiably only one of, and can make announcements every block that puzzles reliant on the announcements can assert.

Además, los generadores también permiten la compresión de ciertos puzzles en un bloque para reducir su costo. Esto depende completamente de los agricultores para que se produzca esta compresión y, como desarrollador de Chialisp, no tiene mucho control sobre ella. Sin embargo, incentiva el uso de esos puzzles comprimibles específicos. Por ejemplo, cuando los agricultores crean un bloque con el formato de transacción estándar en ellos, pueden comprimir esas transacciones en una única clave pública y una referencia de un bloque para buscar el puzzle completo sin comprimir. Si observa el rompecabezas estándar y decide que es demasiado complejo para usarlo como un rompecabezas interno para un rompecabezas que está escribiendo, puede intentar hacer un rompecabezas más simple para ahorrar costos. Sin embargo, es posible que termine pagando más costos porque es posible que su rompecabezas no se comprima, ¡mientras que la transacción estándar sí lo hará!

7.6. Conclusión

Ahora debería tener una comprensión bastante decente del entorno en el que se ejecutarán sus puzzle. Escribir rompecabezas seguros de Chialisp requiere una forma un tanto única de ver las cosas que no comparten muchos otros lenguajes de programación o cadenas de bloques. La comprensión de la red está inextricablemente entrelazada con la creación de acertijos y, si bien eso crea una curva de aprendizaje empinada, también permite una funcionalidad increíble en un formato increíblemente compacto.

Se recomienda encarecidamente que comprenda completamente esta sección antes de decidir implementar cualquier moneda inteligente para que pueda asegurarse de que sean seguras y funcionales. Simplemente ejecutar el rompecabezas en su computadora no es un sustituto de ejecutarlo en millones de nodos con una intención indeterminada.

8. Seguridad

Al escribir Chialisp, las preocupaciones de seguridad deben estar en primer plano. El lenguaje está diseñado específicamente para asegurar dinero en una red sin autoridad centralizada para hacer cumplir las reglas. La única persona que se interponga en el camino de los atacantes y potencialmente grandes sumas de dinero serás tú.

8.1. Firmar y Afirmar la Verdad de la Solución

Recuerde de nuestra discusión sobre los ciclos de vida de las monedas que cuando empuja una transacción, se transmite a otros nodos hasta que encuentra uno que la pone en un bloque. Cada nodo elige lo que se transmitirá al siguiente nodo. Si lo desea, puede cambiar algunos datos antes de reenviarlos.

Es por eso que la firma agregada es parte del paquete de gastos. Le permite marcar los datos como válidos solo si también hay una firma que avala su corrección. Las firmas son la forma en que evita que los nodos cambien su transacción de manera maliciosa; si lo hacen, el gasto ya no será válido.

La firma es especialmente importante cuando se analizan los valores de la solución. El puzzle esta asegurado por el hash del rompecabezas en la moneda. La solución, sin embargo, puede ser cualquier cosa. La mayoría de las veces, cuando gasta una moneda, las condiciones de salida se transmiten de alguna manera a través de la solución. La mayoría de las veces, cuando gasta una moneda, las condiciones de salida se transmiten de alguna manera a través de la solución. Si no firma esas condiciones (o el rompecabezas delegado que las genera) debe asumir que un atacante se dará cuenta e intentará sustituir sus propios valores.

A veces, es necesario tener valores de solución que logísticamente no se pueden firmar, pero que tampoco deben cambiarse. En escenarios como estos, debe intentar tener anuncios de uso de monedas firmados para afirmar que la moneda se está gastando con la información correcta.

8.2. Afirmar la Información de la Moneda

La firma es la forma en que evita que los nodos se metan con sus propios gastos, pero a veces desea crear monedas que se intercambiarán con reglas específicas. Como resultado, no sabe quién gastará la moneda y no sabe si serán honestos. En nuestra discusión sobre puzzles externos, vimos que puede hacer cumplir las reglas en las monedas de su hijo usando el curry y envolviendo hashes de árbol, pero hay ocasiones en las que también desea hacer cumplir las verdades sobre usted o sus padres.

Aquí es donde entra en juego la familia de códigos de operación $ASSERT\_MY\_*$. Cuando necesite información ($parent\_coin\_info$, $puzzle\_hash$, $amount$) sobre su moneda para usar en el puzzle, una parte honesta no siempre puede obtenerla. A veces, será necesario pasarlo a través de la solución. La solución siempre debe tratarse como si la resolvieran partes malintencionadas o descuidadas. Si se está transmitiendo alguna información de moneda, debe afirmarse con códigos de operación para garantizar que la red, que puede ver esa información, pueda confirmarla.

Tenga en cuenta que $ASSERT\_MY\_COIN\_ID$ en realidad afirmará implícitamente las tres piezas de información en una moneda. Lo mismo ocurre con $ASSERT\_MY\_PARENT\_ID$ para las monedas principales, lo cual es particularmente útil ya que no existe por ejemplo: $ASSERT\_MY\_PARENT\_PUZZLE\_HASH$.

8.3. Repetir Ataques

Otra gran preocupación al crear sus gastos es si serán válidos si se excluyen o se reutilizan partes de ellos. Este tipo de ataque es la razón por la que AGG_SIG_UNSAFE está etiquetado como está.

Si firmas algo con AGG_SIG_UNSAFE, los únicos datos que se están firmando es el mensaje que estás intentando firmar. Una vez que lo firma y lo presiona, esa firma vive en la cadena de bloques para siempre. Si luego crea un puzzle que está bloqueado con la necesidad de la misma firma, un atacante puede encontrar la firma que usó la última vez y reutilizarla. Es por eso que debe intentar usar siempre AGG_SIG_ME si es posible. No solo lo obliga a comprometerse con el ID de la moneda en la firma (algo que es único para cada gasto), sino que también se compromete con el desafío de la génesis de la red en la que se encuentra. De lo contrario, una firma revelada para una moneda en testnet podría reproducirse en mainnet.

La exclusión también debe ser una preocupación primordial en su mente. A menudo, gastará varias monedas en el mismo paquete, y todas deben estar unidas en una firma agregada. Si tiene una buena razón para no firmar uno de ellos, asegúrese de saber qué sucede si se excluye del paquete. Además, las firmas agregadas no se pueden desglosar en firmas más pequeñas a menos que haya firmado previamente una de las combinaciones más pequeñas de pares de mensajes de clave pública en el paquete. El atacante puede excluir el resto de transacciones que contienen condiciones AGG_SIG y reutilizar la firma más pequeña nuevamente en las transacciones restantes. También pueden calcular la firma agregada restante y quizás firmar todos los gastos excepto el que excluyen. Esto se conoce como resta de firmas y es otra gran razón para usar AGG_SIG_ME tanto como sea posible.

8.4. El ataque "Flash Loan from God"

Un ángulo interesante que también debe tenerse en cuenta durante la construcción de sus monedas es cómo se mantiene su seguridad si una parte que las está gastando tiene dinero infinito. Esto puede parecer ridículo, excepto que la criptomoneda permite que existan préstamos flash, que son préstamos instantáneos de dinero sin condiciones, excepto que se devuelven al propietario dentro del mismo bloque.

Tomemos, por ejemplo, una moneda de alcancía que solo le permite retirar fondos una vez que el monto de la alcancía ha crecido hasta un objetivo de ahorro determinado. Si una persona quiere recuperar sus fondos antes de tiempo, puede pedir prestado dinero equivalente a su objetivo de ahorro, retirar la alcancía y luego devolver el dinero que pidió prestado.

También existe la posibilidad de utilizar grandes sumas de dinero prestado para influir en el precio de algo, si ese precio se calcula mediante programación. Si tiene suficiente dinero, puede simular singularmente un montón de operaciones para influir en el cálculo del precio al precio que desea, realice una transacción a ese precio y luego devuelva todo el dinero que pidió prestado para simular el comercio y conservar las ganancias.

Afortunadamente, este ataque tiene una solución relativamente fácil, y es agregar una condición $(ASSERT\_HEIGHT\_RELATIVE\ 1)$ para evitar que el dinero se devuelva en el mismo bloque

8.5. Revelaciones de Puzzle y Soluciones

Recuerde pensar en cuándo se revelan los puzzles y las soluciones. Se revelan solo al gastar tiempo de la moneda que se les asigna. Lo único que ve la red antes de eso es la moneda principal y el hash del puzzle. Esto puede ser una ventaja, ya que puede ocultar información confidencial para gastar la moneda dentro del hash del puzzle antes de que se revele. Sin embargo, una vez que se revela el puzzle, se revela para siempre, por lo que la información confidencial no se puede volver a considerar confidencial.

También tenga en cuenta que si una moneda principal está obteniendo información para su moneda secundaria antes de crearla, será pública antes de que se gaste la moneda secundaria. Para algunas billeteras, esto es una ventaja, ya que es posible que desee ciertos datos sobre el puzzle de una moneda para calcular si es suya o no. Sin embargo, si intenta utilizar una contraseña de texto sin formato, no será muy seguro. En su lugar, asegúrese de comprometerse previamente con las cosas con hash y luego afirme que se revelan correctamente más adelante.

8.6. Seguridad de Monedas Bloqueadas con Contraseña

Vale la pena señalar que la moneda bloqueada con contraseña que hemos estado construyendo en realidad no es muy segura. Cuando resuelvas el puzzle, debes revelar la contraseña. Dado que todos los nodos completos a los que les dé su gasto ahora podrán ver su contraseña, ¡pueden cambiar la solución y pagar ellos mismos todo el dinero!

Para solucionarlo, probablemente sea mejor utilizar una clave pública que también tenga que firmar para obtener la solución. El nuevo puzzle podrá ser gastado solo con una contraseña y solo por la persona que haya decidido que posee esta moneda. Por supuesto, esto no es particularmente útil la mayor parte del tiempo y generalmente es tan bueno como una moneda bloqueada con firma con pasos adicionales. Las firmas son, con mucho, la forma más segura de guardar sus monedas.

8.7. Conclusiones

Es de esperar que tenga una mejor idea de los riesgos involucrados al crear un rompecabezas de Chialisp. Vale la pena dedicar su tiempo a intentar explotar sus acertijos pasando soluciones peligrosas o omitiendo transacciones/firmas. No solo está tratando de protegerse contra los malos actores, sino también contra las personas que accidentalmente bloquean sus monedas. Los rompecabezas suelen ser bastante permanentes, por lo que vale la pena el tiempo extra.

9. Debugging

Debido a la naturaleza de los programas de Chialisp, a menudo puede ser difícil determinar dónde exactamente algo va mal. Dado que Chialisp se serializa en CLVM antes de ejecutarse, los errores que reciba a menudo parecerán tener poco sentido dentro del contexto en el que escribió el

código defectuoso. Repasemos ahora algunos trucos que puede utilizar para facilitar la detección de errores en su programa.

9.1. Salida Detallada

Tanto *run* como *brun* tienen un indicador $-v$ para imprimir salidas detalladas. Esta salida es muy detallada y muestra cada evaluación que hizo el programa antes de terminar o salir. Echemos un vistazo a un ejemplo:

```
brun '(c (sha256 0xdeadbeef) ())' '()' -v

FAIL: path into atom ()

(a 2 3) [((c (sha256 0xdeadbeef) ()))] => (didn't finish)

3 [((c (sha256 0xdeadbeef) ()))] => ()

2 [((c (sha256 0xdeadbeef) ()))] => (c (sha256 0xdeadbeef) ())

(c (sha256 0xdeadbeef) ()) [()] => (didn't finish)

() [()] => ()

(sha256 0xdeadbeef) [()] => (didn't finish)

0xdeadbeef [()] => (didn't finish)
```

Cada resultado detallado comienza con $(a23)$ que simplemente representa el rompecabezas completo que se ejecuta con la solución completa. Si está depurando, es probable que esto tenga una salida de (no terminó). Podemos rastrear las apariciones de (no terminó) hasta que encontremos la falla más profunda para evaluar. En este ejemplo, vemos que está intentando ejecutar 0xdeadbeef como un programa para acceder a un valor en la solución. La solución es solo () que obviamente no es lo suficientemente profunda, por lo que arroja un error. Deberíamos haber citado el átomo antes de pasarlo a sha256.

9.2. Errores Comunes

9.2.1. Path into atom

Este error es quizás el error más común que aparecerá cuando ejecute un programa nuevo. Significa que ha intentado atravesar un árbol con un índice que es más profundo que el árbol.

Lo que esto suele intentar transmitir es que algo anda mal con una variable a la que está intentando hacer referencia. Asegúrese de verificar que sus argumentos se pasen correctamente de una función a la siguiente y que todo su código los haga referencia dentro del alcance correcto. Tal vez llamó a una función y no le pasó suficientes parámetros. Puede mirar el resultado detallado para ver qué evaluaciones no terminaron para tener una idea de qué parte podría estar fallando.

9.2.2. first/rest of non-cons

Con este error, clvm está tratando de decirle que ha intentado usar f o r en un átomo en lugar de un cons box. Esto se debe, nuevamente, generalmente a una falta de alineación de argumentos. Asegúrese de saber qué puede ser cada variable cuando se pasa a otra función: un atom, un cons box u otro. Si puede ser cualquiera, asegúrese de verificar si es una desventaja antes de realizar operaciones de lista en él. A veces, esto puede deberse a la evaluación de una lista de una longitud inesperada y al encontrarse con () antes de lo esperado. Además, verifique que toda la evaluación de su programa se esté realizando en el momento adecuado. Quizás un programa se evaluó en un atom demasiado pronto.

9.2.3. sha256 on list

Este error es bastante descriptivo, pero es importante resaltar cuándo ocurre con más frecuencia. A menudo, al crear un programa, querrá compartir un compromiso con algún tipo de programa CLVM con algunos otros datos. Por lo general, esto se hace mediante el hash de árbol del programa usando sha256tree y luego comprometiéndose con él de esa manera. Sin embargo, con la complejidad y las piezas móviles de muchas aplicaciones, es posible que pierda la noción de qué elementos son programas y qué elementos son solo hashes de árbol. Este error a menudo indica que está pasando un programa cuando debería pasar un hash de árbol. Vaya a todas las referencias de sha256 en su aplicación y probablemente pueda encontrar al culpable.

9.2.4. Using (x) to log

A menudo, le gustaría poder ver los valores de una variable en medio de la ejecución de un programa. La mayoría de los lenguajes tienen algún tipo de declaración de registro con la que hacer esto, pero es algo imposible de implementar en Chialisp ya que se evalúa en lugar de ejecutarse. Una de las soluciones que puede utilizar es envolver la declaración que busca depurar en x. El operador de aumento toma un argumento opcional para registrar cuando aumenta. Digamos que está intentando depurar esta línea de código:

```
(list CREATE_COIN_ANNOUNCEMENT (sha256tree (list coin-info coin-data)))
```

Puede intentar comentar esa línea y crear un nuevo aumento para salir con algo de información:

```
; (list CREATE_COIN_ANNOUNCEMENT (sha256tree (list coin-info coin-data)))
(x (list CREATE_COIN_ANNOUNCEMENT (sha256tree (list coin-info coin-data))))
```

Tenga en cuenta que la evaluación se realizará antes de que se cree el mensaje de aumento. A veces es mejor simplemente generar una lista de los argumentos:

```
; (list CREATE_COIN_ANNOUNCEMENT (sha256tree (list coin-info coin-data)))
(x (list coin-info coin-data))
```

También hay una advertencia que se produce cuando intenta depurar una serie de gastos que ocurren de forma consecutiva. Quizás el puzzle se ejecute la primera vez y falle la segunda vez. Si subes durante la ejecución, es posible que también provoques el error del primer puzzle, lo que no te llevará al segundo puzzle. En escenarios como estos, intente averiguar la diferencia entre los gastos y envuelva el aumento en un if para que pueda pasar de manera segura por el primer puzzle.

9.2.5. main.sym

Cuando usa *run* en un *mod* que contiene constantes o funciones, el compilador generará automáticamente un archivo llamado *main.sym*. Este archivo contiene asignaciones de los nombres de funciones o constantes a sus representaciones en el código de bytes. Cuando está ejecutando el programa con *brun*, puede especificar el archivo de símbolo con la bandera $-y$. Luego, cuando vea errores o imprima resultados detallados, verá texto legible por humanos en lugar del entero o código de bytes que se está utilizando para referirse a él.

Esto es particularmente útil cuando se trata de salidas largas y detalladas. Puede desplazarse hacia arriba en el registro hasta que reconozca un fragmento de código que no está terminando. Sin la tabla de símbolos, puede ser mucho más difícil de reconocer.

Es importante destacar que la tabla de símbolos no podrá identificar funciones o macros en línea, ya que se insertan en el momento de la compilación. Si está depurando, probablemente sea una buena idea cambiar las funciones en línea en funciones para que pueda reconocerlas en la tabla de símbolos.

9.2.6. opd y opc

Hay dos comandos más en el repositorio *clvm_tools* que están relacionados con la serialización de CLVM. Cuando el programa se ejecuta en la cadena de bloques, se ejecuta en su forma serializada. A veces puede resultar útil ver esa compilación serializada. A veces puede resultar útil ver esa compilación serializada. Por ejemplo, cuando se evalúa el costo de un programa, se le cobra el costo

por cada byte en la revelación del puzzle. Está incentivado para asegurarse de que la revelación del puzzle sea lo más pequeña posible.

Si desea ver la salida serializada, puede usar opc para compilar o ensamblar el CLVM:

```
$ opc '(q "hello" . "world")'
ff01ff8568656c6c6f85776f726c64
```

Además, otros lenguajes como Python también suelen manejar CLVM en su formato serializado. Si está escribiendo código de controlador para sus rompecabezas, es posible que deba depurar un paquete de gastos que contenga algunos CLVM serializados. En este escenario, puede ser útil desensamblar el programa serializado en la forma legible por humanos.

```
$ opd ff01ff8568656c6c6f85776f726c64
(q "hello" . "world")
```

Con programas grandes, puede que no sea mucho más claro en la forma legible por humanos, pero a menudo aún puede distinguir ciertos patrones. Los argumentos curry, por ejemplo, son relativamente fáciles de elegir y, a menudo, pueden brindarle la información crucial que necesita para depurar sus programas.

9.3. Conclusiones

En ocasiones, depurar Chialisp puede resultar frustrante. Debido a la naturaleza de cómo lisp maneja las estructuras de datos, los programas a menudo continúan con valores incorrectos solo para producir un error en un punto posterior que no da pistas sobre la ruptura inicial. Por ejemplo, un error tipográfico de variable a menudo dará como resultado que la variable se evalúe como una cadena, y si eso se convierte en algo, ¡es imposible saberlo!

Se recomienda que tenga un conocimiento sólido de CLVM, ya que es la base de todos los procesos que ocurren en Chialisp. Hará que sea más fácil construir una imagen en su cabeza de las evaluaciones que están sucediendo y por qué pueden estar sucediendo inesperadamente.

Con suerte, con estos trucos puedes ahorrarte un poco de tiempo y sacar tus monedas inteligentes más rápido.

10. Optimización

Antes de implementar una moneda inteligente en la red, debe examinar de cerca el código para encontrar formas de optimizar su ejecución. Recuerde, el código que escriba se implementará en millones de nodos; si es lento, ralentiza toda la red. Esta es la razón por la que la restricción de bloque en la red Chia depende del costo de ejecución del programa de Chialisp que debe ejecutarse en ese bloque. Si desea escribir una moneda más grande y de ejecución lenta, tendrá que pagar más tarifas cada vez que quiera gastarla. Repasemos algunas técnicas que puede utilizar para optimizar sus puzzles.

10.1. Minimizar el Número de Gastos

Al final del día, una de las mayores pérdidas de costos será la frecuencia con la que tenga que gastar la moneda. Con bastante frecuencia, encontrará formas de construir monedas donde los participantes deben gastar la moneda para interactuar con ella. La moneda puede atravesar varios estados mientras lo hacen. Cada vez que tiene que gastar la moneda, actúa como un multiplicador del costo del programa base. Incluso si no está atravesando un camino costoso a través del código, aún debe revelarse el rompecabezas completo y lo más probable es que haya condiciones $CREATE\_COIN$ y AGG_SIG_ME que a menudo representan una gran parte del costo.

También es importante que tenga la menor cantidad posible de firmas y operaciones de firma. Por lo general, es una buena práctica recopilar todo lo que necesita firmarse en su programa, hacer un hash y pedir una sola firma en ese hash. A veces, también puede ser inteligente con anuncios en los que una moneda sin firmar puede afirmar su información relevante de una moneda firmada. Sea creativo, pero recuerde siempre verificar que toda la información importante esté firmada o confirmada.

10.2. *defun* vs *defun − inline*

En la mayoría de los casos, es mejor utilizar funciones en línea en lugar de funciones normales. Las funciones en línea se insertan donde se llaman en tiempo de compilación, lo que eliminará la sobrecarga de llamadas a la función y no almacenará la función por separado en el código.

Existe un escenario potencial donde esto no es cierto. Si está utilizando una función en línea con un argumento que se ha calculado, terminará pagando por ese cálculo cada vez que se haga referencia al argumento:

```
(defun-inline add_to_self (x) (+ x x))
(add_to_self (* 200 200))
```

El fragmento de código anterior dará como resultado la siguiente expansión:

```
(+ (* 200 200) (* 200 200))
```

Como puede ver, ¡la costosa operación de multiplicación ahora se ha realizado dos veces!

10.3. Familiarízate con todos los operadores

Asegúrese de consultar la sección de referencia para averiguar todos los operadores que puede utilizar y cuánto cuestan (https://chialisp.com/docs/ref/clvm/). Muchos operadores comunes que podría tener la tentación de utilizar tienen un costo sorprendentemente alto y es mejor evitarlos.

Por ejemplo, es posible que desee evaluar de manera diferente en función de si un número es par o impar:

```
(if (r (divmod value 2))
  ; do odd things
  ; do even things
)
```

Tenga en cuenta que el if aprovecha el hecho de que 0 == (). Esta técnica también es útil cuando se recurre a listas. El último elemento de una lista es siempre (), que se evalúa como falso, por lo que en ese caso puede romper la recursividad.

Sin embargo, *divmod* es una operación bastante cara, y tenemos que agregar una *r* para acceder al resto una vez que la operación se haya completado. En su lugar, podemos usar *logand* para evaluar solo el último bit:

```
(if (logand value 1)
  ; do odd things
  ; do even things
)
```

¡Ahora nos hemos ahorrado al menos el 50

10.4. Mantenga el Número de Argumentos Pequeño

Este consejo es bueno tanto para la optimización como para la legibilidad. Mientras el programa se está ejecutando, debe pagar un costo para buscar un valor en el entorno. No es un costo grande, pero aumenta cuanto más profundo tiene que ir al árbol del entorno para buscar el valor. Si puede mantener el número de argumentos pequeño, puede recortar el costo cada vez que su programa usa un argumento en su evaluación.

Una forma de hacerlo es agrupar los argumentos que siempre terminan juntos en el mismo lugar. He aquí un ejemplo:

```
(mod (
      CURRIED_PUBKEY
      some_data
      some_other_data
      some_more_data
      even_more_data
      pubkey
```

```
      my_amount
      my_id
     )

     (import "condition_codes.clvm")
     (import "sha256tree.clvm")

     (defun-inline agg_sig (CURRIED_PUBKEY some_data some_other_data some_more_data
         ↪ even_more_data)
        (AGG_SIG_ME CURRIED_PUBKEY (sha256tree (list some_data some_other_data
            ↪ some_more_data even_more_data)))
     )

     (defun-inline assert_amount_and_sig (CURRIED_PUBKEY some_data some_other_data
         ↪ some_more_data even_more_data my_amount)
        (c (ASSERT_MY_AMOUNT my_amount) (agg_sig CURRIED_PUBKEY some_data
            ↪ some_other_data some_more_data even_more_data))
     )

     (defun-inline assert_id_and_amount_and_sig (CURRIED_PUBKEY some_data
         ↪ some_other_data some_more_data even_more_data my_amount my_id)
        (c ASSERT_MY_ID my_id (assert_amount_and_sig CURRIED_PUBKEY some_data
            ↪ some_other_data some_more_data even_more_data my_amount))
     )

     (assert_id_and_amount_and_sig CURRIED_PUBKEY some_data some_other_data
         ↪ some_more_data even_more_data my_amount my_id)
)
```

Puede ver que el código está un poco fuera de control desde el punto de vista de la legibilidad, y al acceder a my_amount y my_id tenemos que profundizar en el árbol del entorno para leer sus valores. En su lugar, deberíamos simplemente agrupar todos nuestros datos en una lista para empezar.

```
(mod (
      CURRIED_PUBKEY
      all_data
      pubkey
      my_amount
      my_id
     )

     (import "condition_codes.clvm")
     (import "sha256tree.clvm")

     (defun-inline agg_sig (CURRIED_PUBKEY all_data)
        (AGG_SIG_ME CURRIED_PUBKEY (sha256tree all_data))
     )

     (defun-inline assert_amount_and_sig (CURRIED_PUBKEY all_data my_amount)
        (c (ASSERT_MY_AMOUNT my_amount) (agg_sig CURRIED_PUBKEY all_data))
     )

     (defun-inline assert_id_and_amount_and_sig (CURRIED_PUBKEY all_data my_amount my_id
         ↪ )
        (c ASSERT_MY_ID my_id (assert_amount_and_sig CURRIED_PUBKEY all_data my_amount))
     )

     (assert_id_and_amount_and_sig CURRIED_PUBKEY all_data my_amount my_id)
```

```
25 )
```

10.5. No uses Funciones por Reflejo

A menudo, usar una función común puede convertirse en una cuestión de hábito y puede terminar usándola donde en realidad crea más complejidad de la necesaria. Un buen ejemplo es sha256tree. Dado que la función funciona en cons box o atoms, puede tener la tentación de usarla en un solo átomo (tal vez lo esté convirtiendo en una función). La función necesita trabajar de esta manera porque se repite y siempre se encontrará con átomos mientras lo hace. Sin embargo, usarlo para hash solo un atom en realidad agrega costos innecesarios al programa. No solo agrega la sobrecarga de llamada a la función, sino que también agrega la verificación para ver si es un atom o una lista, ¡aunque sepa que es un atom! Un método más rentable es aplicar el hash manualmente como si se hiciera un hash en un árbol: (sha256 1 $some\_atom$).

10.6. Conclusiones

Muchas de las optimizaciones que puede hacer pueden parecer tontas por la pequeña cantidad de costo que ahorran. Sin embargo, si espera que su moneda se use ampliamente, habrá miles de usuarios que pagarán por eso en tarifas todos los días. Con el tiempo, puede suponer una gran cantidad de dinero desperdiciado. Es importante tomarse el tiempo para revisar su código y asegurarse de que puede ahorrar tanto costo como sea posible antes de implementarlo en la red.

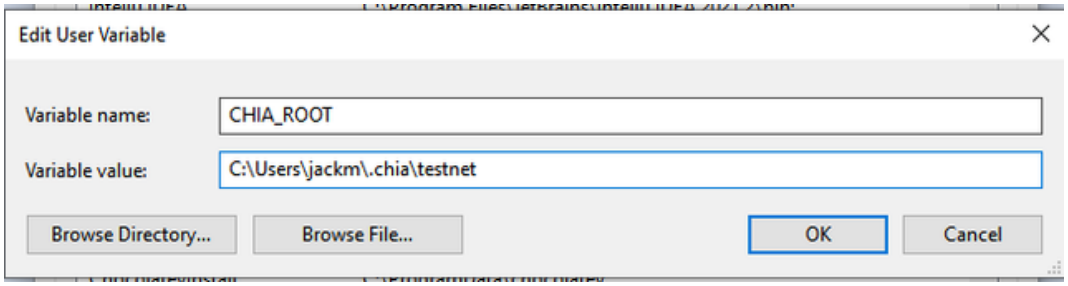
11. Configuración de la App de Chia en Testnet

11.1. Testnet7

- Agregar la variable de entorno $CHIA\_ROOT$ En Mac y linux ejecutar:

```
1 $ export CHIA_ROOT="~/.chia/testnet"
```

En Windows:



- Ejecutar en la linea de comandos

```
1 $ chia init
2 $ chia configure --testnet true
```

- Crear el directorio db en el directorio de testnet (../.chia/testnet/db)
- En el directorio (../*.chia/testnet/db/*) insertar la db:

```
1 $ wget https://download.chia.net/testnet7/blockchain_v1_testnet7.sqlite
```

- Ejecutar además:

```
1 $ chia keys generate
2 $ chia start farmer
3 $ chia show -a testnet-node.chia.net:58444
4 $ chia wallet show
```

- Recibir la moneda TXCH desde faucet de chia

11.2. Billetera Ligera Testnet10

Si deseamos usar la billetera ligera en testnet10 entonces ejecutar:

```
$ git clone https://github.com/Chia-Network/chia-blockchain.git -b
    ↪ protocol_and_cats_rebased --recurse-submodules
$ cd chia-blockchain/
$ sh install.sh
$ . ./activate
$ sh install-gui.sh
$ cd chia-blockchain-gui
$ npm run electron &
```

Recibir la moneda TXCH desde el faucet de chia

11.3. Nodo Completo Testnet10

Si deseamos ejecutar el full node testnet10 entonces ejecutar:

- Descargar el archivo blockchain_v1_testnet10.sqlite
- Mover el archivo "blockchain_v1_testnet10.sqlite" a la carpeta "../.chia/standalone_wallet/db/" (la carpeta "db" hay que crearla)
- Ir al directorio "chia-blockchain" en donde se instaló la billetera ligera y ejecutar:

```
$ cd chia-blockchain/
$ . ./activate
$ chia start node
```

- Puede revisar el progreso de la sincronización ejecutando:

```
$ chia show -s
```

11.4. Añadir CAD admin tool

- Activar el entorno virtual creado en testnet10 y realizar las instalaciones correspondientes:

```
$ cd chia-blockchain/
$ . ./activate
$ git clone https://github.com/Chia-Network/CAT-admin-tool.git -b main --
    ↪ recurse-submodules
$ cd CAT-admin-tool
$ pip install .
$ pip install chia-dev-tools --no-deps
$ pip install pytest
```

- Verificar que todo esta funcionando ejecutando

```
$ cats --help
$ cdv --help
$ chia show -s
```

Además, podemos probar que funciona correctamente creando un CAT.

11.5. Creación de un CAT

- Comprobar que el nodo esta actualizado

```
$ cd chia-blockchain/
$ . ./activate
$ chia start node
$ chia show -s
```

- Una vez que esta sincronizado abrir la wallet ligera y copiar nuestra wallet en testnet10 la cual debe tener mojos (1000 por cada CAT que se desea crear).
- Ir a la carpeta "CAT-admin-tool"
- Ejecutar los siguientes comandos cambiando las variables "<your receive address>" por la wallet que tenemos y "<XCH mojos>" por la cantidad de mojos a utilizar por ejemplo 100.

```
$ cats --tail ./reference_tails/genesis_by_coin_id.clsp.hex --send-to <your receive
    ↪  address> --amount <XCH mojos> --as-bytes --select-coin
```

- Usar los datos insertados en el paso anterior y añadir un <Coin ID>que se obtiene en el paso anterior como "Name:"

```
cats --tail ./reference_tails/genesis_by_coin_id.clsp.hex --send-to <your receive
    ↪ address> --amount <XCH mojos> --as-bytes --curry 0x<Coin ID>
```

- Guardar el <Asset ID>y <Spend Bundle>que salen como outputs del anterior paso y ejecutar:

```
cdv rpc pushtx <Spend Bundle>
```

Obteniendo como resultado

```
{
    "status": "SUCCESS",
    "success": true
}
```

11.6. Añadir ID de la billetera al CAT

- Abir la wallet ligera y observar que se ha disminuido el balance por la creación del CAT (por default cada CAT = 1000 mojos). Además ver si se agregó automaticamente dando click en "CHIA WALLET" en caso de no haberlo hecho o en otra billetera añadir el token.

```
$ cd chia-blockchain/
$ . ./activate
$ cd chia-blockchain-gui
$ npm run electron &
```

- En la wallet ligera hacer click en "+ ADD TOKEN" luego click en "- Custom". Insertar un nombre cualquiera en el primer campo y el "<Asset ID>" para el segundo campo.

12. Proyecto Piggybank Coin

12.1. Requerimientos

- Va a permitir sacar dinero a una dirección predeterminada
- No se podrán realizar cambios una vez creada

- Cualquiera puede realizar un depósito (sin firmas, ni permisos)
- Servirá para servicios de crowd funding
- Existirá por siempre este servicio de crowd funding

12.2. Creación del Entorno Virtual de Trabajo

```
$ mkdir testProject
$ cd testProject
$ python3 -m venv venv
$ ln -s venv/bin/activate
$ source activate
$ pip install chia-dev-tools
$ brun --help
$ cdv --help
```

12.3. Carpeta Include

Obtener las constantes de Chialisp (testProject/include/condition_codes.clib) ejecutando:

```
$ cdv clsp retrieve condition_codes
```

12.4. Moneda del Piggybank

Luego se puede incluir el directorio obtenido (en el paso anterior) en piggybank.clsp (testProject/piggybank/piggybank.clsp)

```
(mod    (
            my_amount
            new_amount
            my_puzzlehash
        )

        (include condition_codes.clib)

        (defconstant TARGET_AMOUNT 500)
        (defconstant CASH_OUT_PUZZLE_HASH 0xcafef00d)

        (defun-inline cash_out (CASH_OUT_PUZZLE_HASH my_amount new_amount my_puzzle_hash
            ↪ )
            (list
                (list CREATE_COIN CASH_OUT_PUZZLE_HASH new_amount)
                (list CREATE_COIN 0)
            )
        )

        (defun-inline recreate_self (my_amount new_amount my_puzzlehash)
            (list
                (list CREATE_COIN my_puzzlehash new_amount)
            )
        )

        ;main
        (if (> new_amount my_amount)
            (if (> new_amount TARGET_AMOUNT)
                (cash_out CASH_OUT_PUZZLE_HASH my_amount new_amount my_puzzle_hash)
                (recreate_self my_amount new_amount my_puzzlehash)
```

```
            )
            (x)
        )

)
```

Con este código tenemos prácticamente una moneda, incluso podemos obervar que compila correctamente si usamos el comando:

```
$ cdv clsp build ./piggybank/piggybank.clsp
```

Esta es una manera de comprobar que nuestro código es válido a pesar que tal vez no haga lo que queramos o tenga problemas de seguridad. Cuando queremos gastar una moneda tenemos que crear un "spend bundle" y enviarlo a la red (es decir a diferentes nodos de CHIA). Los nodos de la red podrían tratar de cambiar el código (para ganar más fees por ejemplo), por esta podemos crear una moneda más segura añadiendo en las funciones $cash\_out$ y $recreate\_self$:

```
(list ASSERT_MY_AMOUNT my_amount)
(list ASSERT_MY_PUZZLEHASH my_puzzlehash)
(list CREATE_COIN_ANNOUNCEMENT new_amount)
```

Obteniendo que el código de ../testProject/piggybank/piggybank.clsp sería:

```
(mod    (
            TARGET_AMOUNT
            CASH_OUT_PUZZLE_HASH
            my_amount
            new_amount
            my_puzzlehash
        )

        (include condition_codes.clib)

        (defun-inline cash_out (CASH_OUT_PUZZLE_HASH my_amount new_amount my_puzzle_hash
            ↪ )
            (list
                (list CREATE_COIN CASH_OUT_PUZZLE_HASH new_amount)
                (list CREATE_COIN 0)
                (list ASSERT_MY_AMOUNT my_amount)
                (list ASSERT_MY_PUZZLEHASH my_puzzlehash)
                (list CREATE_COIN_ANNOUNCEMENT new_amount)
            )
        )

        (defun-inline recreate_self (my_amount new_amount my_puzzlehash)
            (list
                (list CREATE_COIN my_puzzlehash new_amount)
                (list ASSERT_MY_AMOUNT my_amount)
                (list ASSERT_MY_PUZZLEHASH my_puzzlehash)
                (list CREATE_COIN_ANNOUNCEMENT new_amount)
            )
        )

        ;main
        (if (> new_amount my_amount)
            (if (> new_amount TARGET_AMOUNT)
                (cash_out CASH_OUT_PUZZLE_HASH my_amount new_amount my_puzzle_hash)
                (recreate_self my_amount new_amount my_puzzlehash)
            )
            (x)
        )
```

```
38
39 )
```

Para compilar el proyecto y obtener el archivo piggybank.clsp.hex ejecutar:

```
1 $ cdv clsp build ./piggybank/piggybank.clsp
```

Ahora para obtener la versión compilada en CLV ejecutar:

```
1 $ cdv clsp disassemble ./piggybank/piggybank.clsp.hex
```

En caso de querer ejecutar el comando curry con la finalidad de usarlo antes de hacer el puzzle ejecutar:

```
1 $ cdv clsp curry ./piggybank/piggybank.clsp.hex -a 500 -a 0xcafef00d
```

En dónde: 500 es el argumento TARGET_AMOUNT y 0xcafef00d representa el argumento de CASH_OUT_PUZZLE_HASH.

12.5. Chialisp Driver Code

Chialisp Driver Code es código necesario para correr nuestra aplicación ya que la misma no esta escrita puramente usando Chialisp ya que no es buena haciendo llamadas RPC o escribiendo en las bases de datos. Por tanto siempre será necesario usar otros lenguages en la parte superior de los puzzles. Al momento Python soporta muy bien esta integración con Chialisp por está razón se lo utiliza aquí junto al puzzle driver code del repositorio de la blockchain de Chia.Tomar en cuenta que:

- Chialisp Driver Code es la única interface entre el puzzle y el resto de la aplicación
- Es necesario actualizar este archivo si se quiere cambiar cualquier cosa del puzzle
- En este caso el Chialisp Driver Code es sencillo ya que solamente podemos hacer dos cosas con nuestra moneda: crearla o contribuir.
- Es necesario un archivo llamado "_init_.py" además del Chialisp Driver Code "piggybank_drivers.py"

El archivo "../testProject/piggybank/_init_.py" esta vacío, pero sirve para indicar que la carpeta piggibank es un paquete de Python, mientras que el "../testProject/piggybank/piggybank_drivers.py" contiene:

```
1  from chia.types.blockchain_format.coin import Coin
2  from chia.types.blockchain_format.sized_bytes import bytes32
3  from chia.types.blockchain_format.program import Program
4  from chia.types.condition_opcodes import ConditionOpcode
5  from chia.util.ints import uint64
6  from chia.util.hash import std_hash
7
8  from clvm.casts import int_to_bytes
9
10 from cdv.util.load_clvm import load_clvm
11
12 PIGGYBANK_MOD = load_clvm("piggybank.clsp","cdv.examples.clsp")
13
14 #Create a piggybank
15 def create_piggybank_puzzle(amount, cash_out_puzhash):
16     return PIGGYBANK_MOD.curry(amount, cash_out_puzhash)
17
18 #Generate a solution to contribute a piggybank
19 def solution_for_piggybank(pb_coin, contribution_amount):
20     return Program.to([pb_coin.amount, (pb_coin.amount + contribution_amount), pb_coin.
       ↪ puzzle_hash])
21
```

```
# Return the condition to assert the announcement
def piggybank_announcement_assertion(pb_coin, contribution_amount):
    return [ConditionOpcode.ASSERT_COIN_ANNOUNCEMENT, std_hash(pb_coin.name() +
        ↪ int_to_bytes(pb_coin.amount + contribution_amount))]
```

En donde el comando *load_clvm* toma el archivo "piggybank.clsp" y su paquete "cdv.examples.clsp" (ejemplo obtenido de las herramientas de desarrollador). La función "piggybank_announcement_assertion" va a sacar un hash usando el ID de la moneda con el mensaje que hemos enviado (hash del padre + hash(amount+contribution amount)), en dónde *pb_coin.name*() representa el hash del padre.

12.6. Test

Con la finalidad de testear el código, obtener la carpeta para el testeo, ejecutando:

```
$ cdv test --init
```

Crear el archivo ../tests/test_skeleton.py que contiene el código básico para testear nuestra moneda piggybank

```
import pytest

from cdv.test import setup as setup_test


class TestSomething:
    @pytest.fixture(scope="function")
    async def setup(self):
        network, alice, bob = await setup_test()
        await network.farm_block()
        yield network, alice, bob

    @pytest.mark.asyncio
    async def test_something(self, setup):
        network, alice, bob = setup
        try:
            pass
        finally:
            await network.close()
```

Sin embargo, el testeo realmente se hace como muestra el archivo ../tests/test_piggybank.py, que contiene el código:

```
import pytest

from typing import Dict, List, Optional

from chia.types.blockchain_format.coin import Coin
from chia.types.spend_bundle import SpendBundle
from chia.types.condition_opcodes import ConditionOpcode
from chia.util.ints import uint64

from cdv.examples.drivers.piggybank_drivers import (
    create_piggybank_puzzle,
    solution_for_piggybank,
    piggybank_announcement_assertion,
)

from cdv.test import CoinWrapper
from cdv.test import setup as setup_test


```

```python
class TestStandardTransaction:
    @pytest.fixture(scope="function")
    async def setup(self):
        network, alice, bob = await setup_test()
        await network.farm_block()
        yield network, alice, bob

    async def make_and_spend_piggybank(self, network, alice, bob, CONTRIBUTION_AMOUNT) -> Dict[str, List[Coin]]:
        # Get our alice wallet some money
        await network.farm_block(farmer=alice)

        # This will use one mojo to create our piggybank on the blockchain.
        piggybank_coin: Optional[CoinWrapper] = await alice.launch_smart_coin(
            create_piggybank_puzzle(uint64(1000000000000), bob.puzzle_hash)
        )
        # This retrieves us a coin that is at least 500 mojos.
        contribution_coin: Optional[CoinWrapper] = await alice.choose_coin(
            CONTRIBUTION_AMOUNT)

        # Make sure everything succeeded
        if not piggybank_coin or not contribution_coin:
            raise ValueError("Something went wrong launching/choosing a coin")

        # This is the spend of the piggy bank coin.  We use the driver code to create the solution.
        piggybank_spend: SpendBundle = await alice.spend_coin(
            piggybank_coin,
            pushtx=False,
            args=solution_for_piggybank(piggybank_coin.as_coin(), CONTRIBUTION_AMOUNT),
        )
        # This is the spend of a standard coin.  We simply spend to ourselves but minus the CONTRIBUTION_AMOUNT.
        contribution_spend: SpendBundle = await alice.spend_coin(
            contribution_coin,
            pushtx=False,
            amt=(contribution_coin.amount - CONTRIBUTION_AMOUNT),
            custom_conditions=[
                [
                    ConditionOpcode.CREATE_COIN,
                    contribution_coin.puzzle_hash,
                    (contribution_coin.amount - CONTRIBUTION_AMOUNT),
                ],
                piggybank_announcement_assertion(piggybank_coin.as_coin(), CONTRIBUTION_AMOUNT),
            ],
        )

        # Aggregate them to make sure they are spent together
        combined_spend = SpendBundle.aggregate([contribution_spend, piggybank_spend])

        result = await network.push_tx(combined_spend)
        return result

    @pytest.mark.asyncio
    async def test_piggybank_contribution(self, setup):
        network, alice, bob = setup
        try:
```

```
            result: Dict[str, List[Coin]] = await self.make_and_spend_piggybank(network,
                ↪  alice, bob, 500)

            assert "error" not in result

            # Make sure there is exactly one piggybank with the new amount
            filtered_result: List[Coin] = list(
                filter(
                    lambda addition: (addition.amount == 501)
                    and (
                        addition.puzzle_hash == create_piggybank_puzzle(1000000000000,
                            ↪ bob.puzzle_hash).get_tree_hash()
                    ),
                    result["additions"],
                )
            )
            assert len(filtered_result) == 1
        finally:
            await network.close()

    @pytest.mark.asyncio
    async def test_piggybank_completion(self, setup):
        network, alice, bob = setup
        try:
            result: Dict[str, List[Coin]] = await self.make_and_spend_piggybank(network,
                ↪  alice, bob, 1000000000000)

            assert "error" not in result

            # Make sure there is exactly one piggybank with value 0
            filtered_result: List[Coin] = list(
                filter(
                    lambda addition: (addition.amount == 0)
                    and (
                        addition.puzzle_hash == create_piggybank_puzzle(1000000000000,
                            ↪ bob.puzzle_hash).get_tree_hash()
                    ),
                    result["additions"],
                )
            )
            assert len(filtered_result) == 1

            # Make sure there is exactly one coin that has been cashed out to bob
            filtered_result: List[Coin] = list(
                filter(
                    lambda addition: (addition.amount == 1000000000001) and (addition.
                        ↪ puzzle_hash == bob.puzzle_hash),
                    result["additions"],
                )
            )
            assert len(filtered_result) == 1
        finally:
            await network.close()

    @pytest.mark.asyncio
    async def test_piggybank_stealing(self, setup):
        network, alice, bob = setup
        try:
```

```
            result: Dict[str, List[Coin]] = await self.make_and_spend_piggybank(network,
                ↪  alice, bob, -100)
            assert "error" in result
            assert (
                "GENERATOR_RUNTIME_ERROR" in result["error"]
            )  # This fails during puzzle execution, not in driver code
        finally:
            await network.close()
```

Los tests que se van a realizarce pueden observarse ejecutando:

```
$ cdv test --discover
```

En donde pushtx envía la transacción directamente a la red, amt especifíca la cantidad que vamos a gastar. Finalmente, para testear los archivos de la carpeta tests, ejectutar:

```
$ cdv test
```

12.7. Deploy on Testnet

El deploy va a ser mucho más simple en el futuro, sin embargo es una buena manera de ver como funcionan la construcción de bloques en la blockchain de CHIA. Para hacer el deploy se utilizará los comandos de compilación y curry. Primeramente, obtendremos la versión compilada de ../piggybank/piggybank.clsp

```
$ cdv clsp build piggybank/piggybank.clsp
```

Obtener el $puzzleHashOfYourWallet$ con el comando

```
brun '(sha256 2)' '([yourWallet])'
```

Obteniendo el $puzzleHashOfThisPuzzle$ que se lo usará con el comando:

```
$ cdv  clsp curry piggybank/piggybank.clsp.hex -a 500 -a [puzzleHashOfYourWallet] --
    ↪ treehash
```

Luego, con el comando:

```
$ cdv encode [puzzleHashOfThisPuzzle] --prefix txch
```

Se obtendrá, una billetera a la que deberemos enviar dinero para que se cree la alcancía (no para darle dinero).

```
txch18464wupg7wz9z5sere6ssjraj93ga4wpzxu0qjupd69lqcq2r8lqkt3vkj
```

Ahora, para crear la alcancía en nuestro nodo de Chia ejecutar:

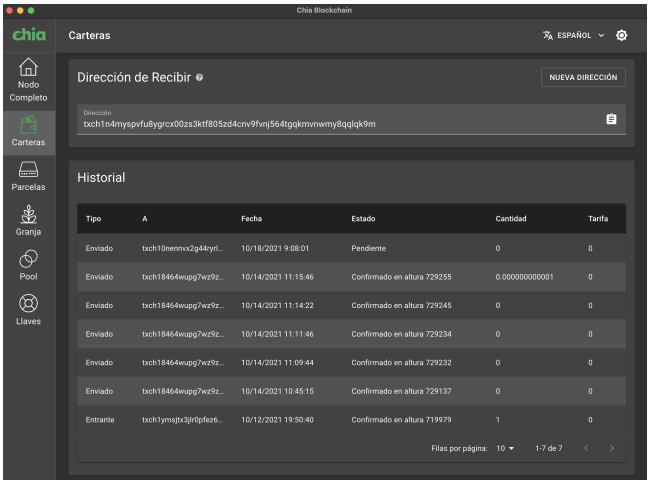
```
chia show -s
chia wallet send -a 0 -t [billetera] --override
```

Ahora bien hay que tomar en cuenta que actualmente hay que crear una billetera nueva para cada moneda que escribimos. Además, se debe agregar a la GUI

```
cdv rpc coinrecords --by puzhash [puzzleHashOfThisPuzzle]
```

Una ves que se confirme el registro de moneda, vamos a obtener una respuesta como la que se muestra a continuación:

```
[{'coin': {'amount': 0,
           'parent_coin_info': '0
               ↪ x3a065e71c2a826a23b83c9087b43651b70bf17a30fea70645cb68ae66480f16c',
           'puzzle_hash': '0
               ↪ x3d75577028f3845152191e7508487d91628ed5c111b8f04b816e8bf0600a19fe'}]
```



Tomar en cuenta que hay que esperar hasta que se confirme la transacción en la blockchain. Para ello revisar el historial.

Caso contrario la respuesta obtenida al ejecutar "cdv rpc coinrecords –by puzhash [puzzleHashOfThisPuzzle]" será

```
[]
```

12.8. Contribuir al Piggybank

Con la finalidad de contibuir al Piggybank vamos a utilizar el archivo "../piggybank/contribution.clsp". Que tiene el código:

```
(mod (
        PUBKEY
        coin_id
        new_amount
    )

    (include condition_codes.clib)

    (list
        (list ASSERT_COIN_ANNOUNCEMENT (sha256 coin_id new_amount))
        (list AGG_SIG_UNSAFE PUBKEY (sha256 coin_id new_amount))
    )
)
```

Además, se utilizará el archivo "../piggybank/sign_contribution.py" para firmar la contribución:

```
from blspy import PrivateKey, AugSchemeMPL

from chia.util.hash import std_hash

from clvm.casts import int_to_bytes

SK = PrivateKey.from_bytes(bytes.fromhex("3
    ↪ d4237d9383a7b6e60d1bfe551139ec2d6e5468205bf179ed381e66bed7b9788"))
COIN_ID = bytes.fromhex("")
NEW_AMOUNT = int_to_bytes(1000)

signature = AugSchemeMPL.sign(SK, std_hash(COIN_ID + NEW_AMOUNT))

print(str(signature))
```

También se utilizará el archivo Json "../spend_bundle.json", el cuál contendrá los gastos realizados.

```
{
  "coin_spends": [
    {
      "coin": {
        "parent_coin_info": "",
        "puzzle_hash": "",
        "amount": 0
      },
      "puzzle_reveal": "",
      "solution": ""
    },
    {
      "coin": {
        "parent_coin_info": "",
        "puzzle_hash": "",
        "amount": 1000
      },
      "puzzle_reveal": "",
      "solution": ""
    }
  ],
  "aggregated_signature": ""
}
```

El dato de *parent_coin_info* y *puzzle_hash*, de la primera parte del archivo Json, se obtiene quitándole el prefijo 0x a los valores obtenidos con el comando

```
$ cdv rpc coinrecords --by puzhash [puzzleHashOfThisPuzzle]
```

Ahora, para obtener el primer *puzzle_reveal* ejecutar:

```
$ cdv  clsp curry piggybank/piggybank.clsp.hex -a 500 -a [puzzleHashOfYourWallet] -x
```

La *solucion* tomando en cuenta que la wallet tiene 0 txch y vamos a depositar 1000 txch se obtendría ejecutando:

```
$ opc '(0 1000 0x[puzzleHashOfThisPuzzle])'
```

Asi finalizaría nuestro primer gasto. Ahora, para el segundo que sería de contribución. Creamos otro Piggybank con 1000 txch y completamos la información. Como se lo hizo anteriormente para obtener: *parent_coin_info*, *puzzle_hash* y *puzzle_reveal*.

```
$ cdv rpc coinrecords --by puzhash [puzzleHashOfThisPuzzle_2]
$ cdv  clsp curry piggybank/piggybank.clsp.hex -a [puzzleHashOfYourWallet_2] -x
```

En el caso de la solución primero obtener *coin_id* y luego la *solucion*:

```
$ cdv rpc coinrecords --by puzhash [puzzleHashOfThisPuzzle\_1] -nd
$ opc '(0x[coin\_id] 1000)'
```

Ahora, agregamos nuestra firma agregada *aggregated_signature* ejecutando:

```
$ python3 piggybank/sign_contribution.py
```

Completar los valores del archivo *sign_contribution* con el *coind_id* obtenido anteriormente y enviar todo a la red.

```
$ cdv rpc pushtx spend_bundle.json
```