Indice

Introduzione

Da sempre, un elemento portante della società è l'idea di scambio di risorse, che ha condizionato ogni epoca. Normalmente concludere uno scambio ha sempre previsto la figura di un garante che impone delle regole affinché lo scambio sia valido. Per la prima volta nella storia si hanno a disposizione i mezzi per eliminare questa figura, perché si ha un sistema in cui l'esistenza stessa di un contratto valida la transazione.

Questi tipi di contratti che vengono definiti come smart contract, e devono garantire le proprietà di fiducia, affidabilità e di sicurezza, che in precedenza erano delegate al garante, ma che adesso diventano possibili grazie alle blockchain. Queste ultime sono code nelle quali è consentita l'operazione di lettura, mentre l'unica operazione di scrittura è l'aggiunta (o transazione): non è possibile quindi modificare un elemento già presente nella blockchain; di conseguenza mantiene un registro con la storia di tutte le transazioni eseguite. Inoltre questa struttura dati è distribuita, quindi va a far parte di una rete peer-to-peer in cui viene replicata per ogni nodo. Di conseguenza gli smart contract, se memorizzati nella blockchain, diventano programmi che possono essere eseguiti in modo distribuito e sicuro, e che controllano lo scambio di denaro tra diverse parti.

A livello di programmazione uno smart contract sono definiti da un identificativo, uno stato –cioè i dati– e del codice, inteso come insieme di metodi che modificano lo stato del contratto ed eseguono transazioni. Per implementare smart contract sono messi a disposizioni linguaggi di programmazione dedicati, a ognuno dei quali è associata la relativa blockchain, ad esempio troviamo il linguaggio Solidity per la blockchain Ethereum o Liquidity per Tezos.

Pertanto, il codice presente nei contratti è codice critico, in quanto possono gestire considerevoli somme di denaro, e è quindi interessante fare analisi statica sul comportamento degli smart contract, in merito sono stati fatti diversi lavori[1]. Di questi è particolarmente interessante un lavoro in cui emerge che nel momento in cui un utente razionale agisce per minimizzare le perdite di denaro (o equivalentemente per massimizzare il guadagno) il codice dello smart contract lo può forzare a determinate decisioni. In questo contesto per forzare si intende che da parte dell'utente l'aderenza al contratto avviene, perché se non ci fosse l'utente interessato perderebbe dei soldi. Questa analisi è stata fatta definendo un linguaggio, chiamato scl (smartCalculus), con un ristretto insieme di funzioni sugli smart contract, e che permettesse di modellare il comportamento di contratti e utenti (umani). Scl è un linguaggio ad attori minimale, così rende più semplice l'analisi di smart contract, ma allo stesso tempo espressivo abbastanza da essere Turing completo: permette l'invocazione di metodi, la modifica dei campi, comportamento condizionale, la ricorsione, il sollevamento di eccezioni.

[1] https://link.springer.com/content/pdf/10.1007%2F978-3-030-30985-5_23.pdf

Il comportamento del sistema poi è espresso come questi modelli si passa poi ad aritmetica di logica di Presburger

Parser

Introduzione

In questa sezione ci si concentra sull'implementazione del parser per scl: il parser, per come è definito, dovrà prendere in input in input una lista di caratteri ne determinerà la correttezza, basandosi su una certa grammatica formale, generando quindi un albero di sintassi astratta per tale lista. Verrà quindi mostrata la grammatica formale, in modo che possa rendere più facile la stesura del codice da parte del programmatore, mentre l'albero di derivazione sarà direttamente il codice scl. Il parser è implementato in OCaml, questa scelta risulta abbastanza ovvia dal momento che lo stesso scl è implementato in questo linguaggio.

Scl

Scl è un linguaggio imperativo ad attori che permette di descrivere il comportamento di contratti e umani (cioè gli utenti che interagiscono sui contratti). Scl è un linguaggio usato per l'analisi di smart contract, quindi col fine di fare un analisi più mirata, semplice ed essenziale è stato reso volutamente minimale; nonostante ciò scl è un linguaggio Turing completo in quanto permette l'assegnamento, l'istruzione condizionale, l'invocazione di funzioni, la ricorsione e il sollevamento di eccezioni.

Un programma scl consiste in una una configurazione, ovvero un insieme di attori (ovvero contratti o umani) che vengono definiti con i loro campi e i loro metodi, in maniera analoga ai linguaggi di programmazione con oggetti. Essendo un linguaggio ad attori si ha che ogni attore conosce tutti gli altri attori della configurazione, rendendo possibile –per esempio– che un umano chiami un metodo di uno specifico contratto, senza avere bisogno di parametri o campi aggiuntivi.

Sebbene siano due entità distinte, il codice di contratti e umani è molto simile, se non per il fatto che questi ultimi possono fallire –sollevano dunque un'eccezione– e hanno a disposizione l'operazione di scelta, che consiste un operatore non deterministico con il quale non si sa a priori quale codice l'umano andrà ad eseguire. L'idea è che nella realtà l'umano non ha un comportamento deterministico e quindi si devono prendere in considerazioni tutte le sue azioni possibili; questa diventa la parte interessante dell'analisi con scl in quanto si cerca di capire che comportamento sarà più vantaggioso per l'umano.

Analisi lessicale

Il parser non può fare l'analisi sintattica direttamente sul testo in input, per iniziare deve avere in ingresso una sequenza di token, dove ogni token è una cop-

pia nome valore, dove il nome rappresenta una determinata categoria sintattica, mentre il valore è una stringa del testo. Quindi per generare i token si inizia facendo l'analisi lessicale, dividendo le stringhe in input in diverse categorie di token.

Con questo obiettivo si è usato il modulo `Genlex` di OCaml, che permette di generare un analizzatore lessicale che in OCaml consiste in una funzione che prende in input uno stream di caratteri e restituisce in output una lista di token. Inoltre questo strumento è particolarmente vantaggioso rispetto a un'analisi lessicale senza uso di moduli aggiuntivi, perché toglie la preoccupazione di fornire un'implementazione per l'aggiunta di commenti nel testo, così come diventa automatica la rimozione degli spazi bianchi tra le varie stringhe. I token quindi sono riconosciuti e ad ognuno è associato una categoria (o nome) che sono: interi, stringhe, identificativi e parole chiave, dove queste ultime richiedono di essere esplicitate.

Concettualmente l'analizzatore sintattico richiederebbe uno stream di token, quindi ci si aspetterebbe che il parser implementato richieda un input del tipo `token Stream.t`. Infatti il tipo `Stream` di OCaml offrirebbe un vantaggio in termini di memorizzazione: non è necessario che tutta la sequenza di token sia in memoria, e sebbene l'analizzatore lessicale generato da `make_lexer` –la funzione di `GenLex` che lo genera– restituisca un tipo `token Stream.t`, è stato scelto di usare una lista di token. Questa è stata una scelta di natura implementativa: come sarà più chiaro successivamente, il parser ha bisogno di fare backtracking, e con gli `Stream` di OCaml l'operazione diventerebbe ardua siccome quando si esamina un elemento in uno stream, l'elemento precedente viene perso. Con la lista invece l'iterazione è molto più semplice e si può procedere in entrambe le direzioni senza preoccupazione.

Grammatica

La grammatica su cui si basa il parser cerca di essere conforme e fedele alla struttura di scl, e quindi cerca di coprire tutti i costrutti definiti dal linguaggio.

```
configuration ::= act*
act ::= (Human | Contract) ?( (nat) ) { decl* meth* }
stm ::= var = ( (rhs) | rhs) | if e then stm else stm | stm; stm
        | stm + stm | { stm }
decl ::= t var ?(= v)
meth ::=  var: ( (t * )* t)? -> t = fun var* stm return e
rhs ::= e | (var.)? (.value (e))? var e*
v ::= int | bool | string | contr_addr string | hum_addr string
iexpr ::= int | var | fail | (iexpr)? - iexpr | iexpr + iexpr
        | (int | string) * iexpr | max iexpr iexpr
        | symbol iexpr iexpr | ( iexp )
bexpr ::= true | false | var | fail | iexpr > iexpr | iexpr >= iexpr
        | iexpr < iexpr | iexpr <= iexpr | expr == expr | !bexpr
```

```
        | bexpr && bexpr | bexpr || bexpr | ( bexpr )
sexpr ::= string | var | fail
cexpr ::= this | var | fail | contr_addr string
hexpr ::= var | fail | hum_addr string
expr ::= iexpr | bexpr | sexpr | cexpr | hexpr
t ::= int | bool | string | Contract | Human
```