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, deve prendere in input una lista di caratteri e determinarne la correttezza, basandosi su una certa grammatica formale, generando quindi un albero di sintassi astratta per tale lista. Il parser in considerazione è implementato in OCaml, questa scelta risulta abbastanza ovvia dal momento che lo stesso scl è implementato in questo linguaggio.

Si tiene inoltre a precisare che l'albero di derivazione generato dal parser è direttamente codice scl: siccome il parser non è un compilatore successivamente all'analisi sintattica non si hanno altre fasi di generazione del codice.

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.

TODO: Mostrare esempio di codice smart calculus

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 coppia 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

Una volta costruita la lista di token il parser deve cercare di costruire l'albero di sintassi astratta, che dovrà essere un albero di derivazione in una determinata grammatica per scl che necessita di essere definita. La definizione di questa sintassi è fondamentale in quanto costituisce l'utilità e lo scopo dello stesso parser: fornisce al programmatore una maniera di scrivere codice scl semplice e leggibile. È quindi necessario che la grammatica debba coprire tutti i costrutti del linguaggio scl, cercando di rimanere fedele e conforme alla sua struttura.

```
configuration ::= act*
act ::= (Human | Contract) ?( (Int) ) { decl* meth* }
stm ::= decl | 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
Bool ::= true | false
iexpr ::= Int | Var | fail | (iexpr)? - iexpr | iexpr + iexpr
        | Int * iexpr | max iexpr iexpr  | symbol string | ( iexp )
bexpr ::= Bool | Var | fail | iexpr > iexpr | iexpr >= iexpr
        | iexpr < iexpr | iexpr <= iexpr | e == e | !bexpr
        | bexpr && bexpr | bexpr || bexpr | ( bexpr )
sexpr ::= String | Var | fail
cexpr ::= this | Var | fail | contr_addr String
hexpr ::= Var | fail | hum_addr String
e ::= iexpr | bexpr | sexpr | cexpr | hexpr
t ::= int | bool | string | Contract | Human
```

[2]

In figura viene mostrata la grammatica dell'analizzatore sintattico, dove i simboli terminali sono i token generati nell'analisi lessicale: in figura che in figura vengono denominati Int (ovvero l'insieme dei numeri interi), String (l'insieme delle stringhe) e Var (l'insieme di tutte le variabili), a cui si aggiungono tutte le parole chiave (ad esempio return, if, +, >, …) che devono essere specificate al lexer.

Questa sintassi permette di definire un insieme di attori (a cui si può specificare il `balance` iniziale, equivalentemente ad un assegnamento), ognuno dei quali ha dei campi che devono essere dichiarati e una lista di metodi, che consistono in dichiarazione (con tipo in input e in output, dove quest'ultimo è sempre presente), statement (lista di comandi di assegnamento, istruzione condizionale e per gli umani scelta) e espressione di ritorno.

Si può facilmente notare che la grammatica scelta è ambigua, come viene mostrato in questo breve esempio di un assegnamento: `x = y` che nel contesto di uno statement

```
stm => Var = rhs => x = rhs => x = e =>

Caso 1:
    x = iexpr => x = y

Caso 2:
    x = bexpr => x = y
```

L'ambiguità è quindi dovuta al fatto che tutti i tipi di espressioni possono accettare indistintamente delle variabili, diventerebbe quindi necessario introdurre un sistema di tipi. Il sistema di tipi in realtà è già presente in scl, quindi l'unica cosa di cui si deve preoccupare il parser è di fare un controllo sui tipi, che in

[2] Sintassi per il parser, dove configuration è il simbolo iniziale

questo esempio diventa: se y è stata dichiarata come intera allora sono nel Caso 1 se invece è dichiarata come booleana sono nel Caso 2. Il controllo dei tipi verrà esaminato successivamente più in dettaglio.

TODO: Esempio con sintassi

Implementazione del codice con parser combinator

Una volta definita la grammatica si hanno tutte le basi necessarie per l'implementazione OCaml del parser. Il parser può essere implementato con diverse tecniche: similmente a quanto fatto per l'analisi lessicale si possono usare librerie di OCaml che generano analizzatori sintattici, oppure si può usare la tecnica del parser combinator. In questo caso usata quest'ultima tecnica che consiste nel considerare un parser come una funzione di ordine superiore che avendo uno o più parser in input restituisce un nuovo parser. Dove a livello di codice un parser si intende una funzione che prende una lista di token e restituisce la lista rimanente di token e l'albero di derivazione generato.

```
type 'ast parser =
token t -> (vartable * bool) -> token t * 'ast * (vartable * bool)
```

[3]

Per l'implementazione è necessario definire un'eccezione che ogni funzione `parser` solleverà qualora non dovesse essere in grado di riconoscere l'input, questa eccezione è stata denominata `Fail`.

Con l'ausilio dei parser combinator si possono tradurre facilmente le produzioni della grammatica definita in precedenza: innanzitutto è necessario descrivere gli operatori sintattici in questione ovvero come la concatenazione, la stella di Kleene, l'unione, la costante e la possibilità. L'implementazione di ogni operatore risulta semplice, ad ognuno di questi è fatta corrispondere una funzione, che nella logica dei parser combinator mi genera nuovi parser, per l'operatore costante –che riconosce i simboli terminali– viene generato un parser che verifica se il simbolo richiesto corrisponde al simbolo in input, per l'unione si hanno due parser in input e in maniera non deterministica si deve scegliere il parser che esegue, e nel caso venga sollevata la `Fail` si fa backtracking e sceglie il parser rimanente, la concatenazione invece consiste nel prendere in input due parser e eseguirli sequenzialmente unendo successivamente i due output, la stella di Kleene vede l'esecuzione dello stesso parser finché questo non solleva una `Fail` –caso per cui l'esecuzione dell'operatore termina– infine all'operatore di possibilità corrisponde un parser che semplicemente prevede che possa fallire.

```
let const : token -> (token -> 'ast) -> 'ast parser =
 fun t1 f t2 tbl ->
  if (List.length t2 > 0) && (t1 = (List.hd t2)) then
```

[3]Definizione del tipo parser nel caso discusso. Nell'input e nell'output del parser si hanno inoltre una tabella che contiene i campi dichiarati e un tipo boolean per distinguere se l'attore di cui si sta facendo il parsing sia umano o contratto

```
   (junk t2), f t1, tbl
  else
   raise Fail

let choice : 'ast parser -> 'ast parser -> 'ast parser
= fun p1 p2 s tbl ->
 try p1 s tbl with Fail -> p2 s tbl

let concat :
'ast1 parser -> 'ast2 parser -> ('ast1 -> 'ast2 -> 'ast3) -> 'ast3 parser =
fun p1 p2 f s tbl ->
  let rest1,ast1,tbl1 = p1 s tbl in
  let rest2,ast2,tbl2 = p2 rest1 tbl1 in
  rest2,f ast1 ast2,tbl2

let kleenestar :
'ast2 parser -> 'ast1 -> ('ast1 -> 'ast2 -> 'ast1) -> 'ast1 parser =
 fun p empty_ast f s t ->
  let rec aux p1 s1 acc tbl=
  try
   let (rest1, ast1, ntbl) = p1 s1 tbl in
   aux p1 rest1 (f acc ast1) ntbl
  with Fail -> (s1, acc, tbl)
  in aux p s empty_ast t

let option : 'ast parser -> 'ast option parser =
 fun p s tbl -> try
  let next,res,ntbl = p s tbl in next,Some res,ntbl
 with Fail -> s,None,tbl
```

[Implementazione in codice OCaml dei parser per gli operatori sintattici]

Dopo aver definito questi operatori diventa banale scrivere un parser per ogni non terminale della grammatica: basta combinare tra loro questi operatori coerentemente con quanto definito dalle produzioni della grammatica.

Quindi con l'ausilio dei parser combinator si ha un parser LL(1), di conseguenza è top-down, quindi inizia a costruire l'albero dalla radice, cioè definisce inizialmente una configuration vuota che verrà riempita dall'attività dell'analizzatore sintattico, l'input viene consumato da sinistra a destra e con solo un simbolo di lookahead, che significa che il parser prende in considerazione solo un token alla volta. Il problema, ora, è che la grammatica non è LL(1), perché, come visto in precedenza, presenta delle ambiguità e in più è ricorsiva sinistra, infatti se il parser cercasse di risolvere con la ricorsione sinistra l'esecuzione andrebbe in loop infinito. Sarà quindi necessario eliminare ricorsione sinistra e ambiguità.

Eliminazione ricorsione sinistra

Per eliminare la ricorsione sinistra bisogna prima agire sulla grammatica senza andarne a cambiare la sintassi e poi andare a trascrivere il relativo codice del parser. Si prenderà in considerazione soltanto il caso dell'eliminazione nel non terminale `iexpr`, per gli altri non terminali la risoluzione è analoga. Concettualmente si dividono in due non terminali le produzioni senza ricorsione sinistra da quelle in cui è presente, che chiameremo rispettivamente `atomic_iexpr` e `cont_iexpr` in questo modo

```
atomic_iexpr ::= Int | Var | fail | "-" iexpr | max iexpr iexpr
                | symbol string | "(" iexpr ")"
cont_iexpr ::= "+" iexpr | "-" iexpr | "*" iexpr
```

Rimane quindi solo da definire `iexpr` che è semplicemente

```
iexpr ::= atomic_iexpr (cont_iexpr)?
```

E adesso per la trascrizione nel codice OCaml si tratta solo di trascrivere quanto definito componendo i parser già definiti

```
let rec atomic_int_expr s =
 choice_list [
   comb_parser (base Int) (fun expr -> AnyExpr(Int,expr));
   concat (kwd "-") atomic_int_expr (fun _ -> minus) ;
   concat (concat (kwd "(") int_expr scd) (kwd ")") fst ;
   concat (concat (kwd "max") int_expr scd) int_expr max;
   concat (kwd "symbol") symbol_pars scd;
 ] s
and int_expr s =
 concat atomic_int_expr (option cont_int_expr)
 (fun x f -> match f with Some funct -> funct x | _ -> x) s
and binop s =
 choice_list [
  const (Kwd "+") (fun _ -> plus) ;
  const (Kwd "*") (fun _ -> mult) ;
  const (Kwd "-") (fun _ -> subtract)
 ] s
and cont_int_expr s = concat binop int_expr (fun f x -> f x) s
```

Controllo dei tipi e tabella delle variabili

Come mostrato in precedenza le ambiguità della grammatica, stando alla sua definizione, sono dovute al fatto che in espressioni di tipo diverso può comparire la stessa variabile. Questo però è solo vero a livello di grammatica, invece il codice scl ha un sistema di tipi per cui ovviamente non è possibile che compaia una variabile di un determinato tipo in un'espressione di un altro tipo. Compito del parser comunque è di verificare che i tipi siano coerenti tra di loro in modo da essere in grado di generare il codice scl. Ciò che invece è possibile nel linguaggio

scl è che siccome le variabili sono una coppia tipo-nome, allora due variabili diverse possono lo stesso nome. Ovviamente si è interessati a una sintassi per cui non si debba specificare il tipo ogni volta che venga usata la variabile, si vuole quindi che solo il nome sia l'identificativo per la variabile (non esisteranno quindi due variabili con nome diverso) e che il tipo gli venga assegnato al momento della dichiarazione, e rimanga associato al nome nel suo scope.

Viene allora introdotta una tabella per ogni attore che contiene la lista delle coppie tipo-nome di tutte le sue variabili. Questa tabella contiene informazioni necessarie al parser per tutta la sua analisi e che lo stesso parser può modificare, allora come mostrato in figura ogni oggetto di tipo `parser` deve avere questa tabella in input e in output. Si ricorda che non è possibile avere una tabella come una variabile globale poiché OCaml è un linguaggio funzionale e le variabili non possono cambiare il loro contenuto.

Le operazioni che chiamano in causa le variabili sono le dichiarazioni, l'assegnamento e le espressioni. Nelle dichiarazioni è necessario specificare il tipo di variabile, quindi ad ogni dichiarazione viene aggiunta una variabile con il tipo specificato nella tabella, se è già presente una variabile dichiarata con quel nome allora viene sollevato `Fail`. Nell'assegnamento si ha che l-valore è una variabile dove non viene specificato il tipo, viene quindi fatta ricerca nella tabella per determinare il tipo della variabile, una volta determinato viene fatto il parsing del r-valore e viene controllato se il tipo di quest'ultimo combacia col tipo della variabile. Però può essere che l-valore non sia stato dichiarato, in questo caso si è deciso che venga valutato solo r-valore e che poi la variabile del l-valore venga aggiunta alla tabella col tipo del r-valore. Nel caso ci fosse una variabile in un'espressione […]

Le variabili oltre a poter essere aggiunte alla tabella devono anche poter essere rimosse. Si prendano in considerazione una variabili locale all'interno di una funzione, essa non deve essere visibile in una funzione esterna, quando verrà quindi fatto il parsing di questa seconda funzione la variabile non deve essere presente nella tabella. Diventa allora necessario definire lo scope per le variabili, in modo che determini la presenza o meno della variabile nella tabella. Per come è definito il linguaggio questa è un'operazione banale, infatti in scl non ci sono funzioni annidate, quindi gli unici blocchi su cui può far parte una variabile sono il blocco globale dell'attore e il blocco locale della funzione. Quindi durante il parsing, o la variabile è una variabile globale, o è una variabile locale della funzione presa in esame: non è possibile avere una variabile attiva che sia allo stesso tempo variabile locale di un'altra funzione. Quindi basta associare ad ogni variabile nella tabella un valore booleano che denota se la variabile è locale o meno, per far sì che quando viene finito il parsing di una funzione tutte le variabili locali vengano rimosse.

Tabella delle funzioni

Analogamente alle variabili anche le funzioni possono essere aggiunte nella stessa tabella: quando viene dichiarato un metodo, la firma di questo –nome, lista dei tipi dei parametri e tipo di ritorno– viene salvata nella tabella dell'attore. Così ogni volta che ci sarà una chiamata di funzione dal nome di questa viene cercata la firma del metodo nella tabella, e si verifica che la lista delle espressioni nella chiamata sia coerente con la lista dei parametri presa dalla firma appena ritornata. In scl però si possono chiamare i metodi di uno specifico contratto, in questo caso non verrà fatto nessun controllo perché il contratto potrebbe essere esterno alla configurazione e, in questo caso, non si saprebbe niente sui suoi metodi.

Distinzione tra umani e contratti

Un ultimo accorgimento da fare riguarda la differenza tra il codice di contratti e umani, come mostrato in precedenza. Avendo diverse operazioni i due hanno una sintassi differente (nella figura non vi è distinzione per ragioni di semplicità) e quindi il parser nella sua analisi deve sapere se sta facendo il parsing per un umano o per un contratto. Questa informazione (che si traduce in un booleano) sarà insieme alla tabella come input e output di ogni funzione di tipo `parser`.

Conclusione

Si è quindi visto che per il parser LL(1) implementato, è stato necessario definire in primo luogo una sintassi per scl. Con la tecnica dei parser combinator poi è bastato trascrivere le stesse regole della grammatica come combinazione di più parser, facendo però alcuni accorgimenti: si è tolta la ricorsione sinistra per evitare che il parser divergesse, e poi è stato messo a punto un controllo sui tipi, che grazie all'ausilio di una tabella, toglie le ambiguità.