Contratti, Composizione e Scalatura

La specifica di Rholang

Bozza della versione 0.2

27 Feb 2018

Lucius Gregory Meredith
Jack Pettersson
Gary Stephenson
Michael Stay
Kent Shikama
Joseph Denman

Destinatari: Sviluppatori che utilizzano la piattaforma RChain; sviluppatori che creano o effettuano la manutenzione della piattaforma RChain; progettisti di linguaggi; teorici della concorrenza; appassionati della blockchain e di criptovalute

Per quanto sia giocoso il mondo naturale, le azioni naturali hanno conseguenze naturali.

Il contratto tra cliente e fornitore di servizi

può significare la differenza tra il prosperare ed il fallire.

Eppure, con delle poste in gioco così elevate, la natura scala in maniera naturale. Con grazia. Splendidamente.

Istintivamente, sappiamo come la natura faccia questo. Ogni bambino riesce a vedere l'albero nella foglia. Eppure, noi ce lo scordiamo, e sembra che ce lo ricordiamo, quando riusciamo a ricordarcelo, nel momento cruciale, quando abbiamo bisogno di costruire sistemi su larga scala, e i sistemi che abbiamo costruito stanno fallendo. Come adesso.

Introduzione e motivazione

La necessità di contratti sociali

Bitcoin e la sua blockchain basata sulla proof-of-work hanno indicato la strada verso un medium privo di fiducia per lo scambio finanziario e basato sulle informazioni attraverso il quale questa generazione può affrontare i gravi fallimenti dei sistemi finanziari e degli stati che presumibilmente li regolano e li governano. Eliminando il ruolo della terza parte fidata, questa tecnologia e la generazione che la adotta non trae alcuna necessità di istituzioni finanziarie troppo grandi per fallire, né si deve inginocchiare alle agenzie governative che sovvenzionano questa irresponsabilità sistemica con i soldi dei contribuenti. Essi possono realizzare delle alternative efficaci senza dover sopportare l'assurdo concetto che non vi sia un altro modo, né aspettare che le istituzioni pubbliche si aggiornino sul reale potenziale delle tecnologie Internet. Tuttavia, l'incombente strato di servizi finanziari è ricco
di strumenti finanziari complessi, un fatto non sorvolato da Ethereum e dagli altri individui che lavorano sulle tecnologie emergenti degli smart contracts.

Queste nuove offerte arricchiscono le basi del set di strumenti della blockchain con delle piattaforme computazionalmente complete in cui riprogettare, ed ancora più importante, reimmaginare, lo strato di strumenti finanziari. Nel frattempo, sforzi come Factom riconoscono che la blockchain non è solo per mantenere i registri su chi ha quanti quatloo. I registri dei land trust sono spesso abusati massicciamente in molti paesi, e Factom sta mettendo queste informazioni sulla blockchain in modo che tutti possano vedere un registro pubblico ed un attestato di proprietà.

Allo stesso modo, RChain riconosce che il flusso di informazioni va di pari passo con la strumentazione finanziaria. Inoltre, non si tratta solo di provenienza. Gli obblighi contrattuali nel mondo abilitato alla blockchain riguardano comunque il quando, il dove e il modo relativi alla divulgazione di informazioni sensibili. RChain introduce queste idee in un contesto di social network a rischio relativamente basso, ma questo è principalmente perché crediamo che stabilire fiducia nella tecnologia
in contesti a basso rischio, su una scala a livello di Internet, sia la strada giusta per l'adozione nei mercati in cui c'è più rischio, come il lavoro freelance, o gli incontri online, lo scambio di cartelle cliniche, o l'autodeterminazione attraverso Internet.

Inoltre, RChain riconosce che una piattaforma computazionalmente completa necessita di due elementi chiave per fornire dei servizi realistici e scalabili. Prima di tutto ha bisogno di una chiara semantica computazionale basata su metodi formali. ML e Haskell sono due prominenti e popolari linguaggi che godono di questo tipo di basi e il loro successo non è in poca misura derivato dalle loro basi. Naturalmente, queste lingue sono state concepite quando l'informatica era in gran parte un fenomeno desktop, e non il tessuto della società globale, che si sta facendo spazio

[1] Più precisamente, Factom sta inserendo gli hash dei registri dei land trust nella loro blockchain e ancorando questi alla blockchain bitcoin.

su miliardi di dispositivi interconnessi, con una topologia di comunicazione che si riconfigura dinamicamente in continuazione. In quanto tale, né la lingua, né i modelli computazionali sui quali sono basati sono particolarmente adatti per questo nuovo scopo. No, le semantiche computazionali richieste devono essere quelle che si adattano alle esigenze di questo tempo e luogo - ecco perché il linguaggio contrattuale di RChain si basa sui calcoli del processo mobile, che fornisce, in base alla sua progettazione, un modello matematico dell'infrastruttura informatica di oggi.

In secondo luogo, ma con uguale importanza, la piattaforma computazionale deve fornire un mezzo per limitare, sondare ed esaminare i contratti, cosa che è diventata particolarmente evidente sulla scia dell'exploit di "The DAO". Il modo accettato di farle questo nella progettazione del linguaggio di programmazione moderno e nella semantica del linguaggio di programmazione è con i tipi. Il linguaggio contrattuale di RChain è dotato di un moderno sistema di battitura comportamentale. Questo sistema fornisce alle parti che intendono impegnarsi contrattualmente un mezzo per sondare gli obblighi e le garanzie contrattuali in modo automatico. Ciò offre ai potenziali partecipanti la possibilità di determinare se diventare parte di un particolare contratto è davvero la scelta giusta.

Inoltre, quando una nuova tecnologia facilita la forgiatura o la produzione di un artefatto utile, quello che segue poco dopo è una proliferazione di tali artefatti. Facilitare la creazione e l'esecuzione degli smart contracts genererà inevitabilmente in un'abbondanza degli stessi. Pensaci un attimo. Smart contract è solo un altro nome per il codice. Se non abbiamo imparato nient'altro dai sistemi di controllo del codice sorgente da CVS a SVN a github, dovremmo aver imparato che il codice è la materia oscura di Internet. Che sia all'interno del firewall aziendale o all'esterno,
nella terra dell'open source, la ricerca del codice che fa esattamente ciò che deve essere fatto è un'arte oscura, che è meglio lasciare ai maghi. Le tecnologie come Hoogle sono già mirate alla ricerca sulla base del tipo di informazioni. I tipi comportamentali di RChain permettono di portare quell'idea molto, molto più lontano - cercare codice sulla base di come è modellato e di cosa fa - non solo quali stringhe, parole chiave o frasi potrebbero apparire da qualche parte nel suo testo.

Entrambe queste funzionalità si riferiscono alla scalatura in diverse dimensioni. Scegliere una semantica che rifletta naturalmente come l'elaborazione effettivamente accada su Internet è di fondamentale importanza. La scelta di VM di Ethereum, che impone la serializzazione globale sui contratti su scala di Internet sono precisamente il motivo per cui Ethereum V 1.0 non verrà scalato. Beh, questo e il fatto che neanche la proof-of-work scali. Al DevCon1, il discorso di Vitalik sulla scalatura di Ethereum ammette proprio questo. Entrambe le sue proposte sul cambiare l'architettura e sulle corrispondenti modifiche al linguaggio contrattuale che codifica esso sono di diversi passi nella direzione che il modello RChain ha avuto sin dall'inizio.

Dove le scelte di RChain cominciano davvero a brillare, tuttavia, sono nelle caratteristiche che emergono dall’avere un sistema di tipo comportamentale. Uno degli aspetti chiave della scalatura non è quanti o quanto velocemente, ma chi e perché. Quali agenti useranno il sistema, e per quale scopo? Non passera molto tempo prima che le principali società realizzino che la blockchain - come attualmente resa dal
codice in esecuzione - è semplicemente un modo molto più lento di fare le cose rispetto ai loro sistemi interni.
Non ha praticamente alcun senso dietro al firewall aziendale. Il suo valore reale è nel dominio pubblico tra

agenti privati. Aggiungi a questo il fatto che è una tecnologia nuova di zecca con ogni sorta di difetto e di rischio, e l'interesse svanirà molto presto. Risolvere solo la questione sulla velocità ed i problemi di throughput non risolverà quest'ultima preoccupazione.

La tipizzazione comportamentale dei contratti sociali significa che - per la prima volta - abbiamo una tecnologia che può imporre il flusso di informazioni e le proprietà di vividezza su scala Internet. Questo è il naturale passo successivo nel demolire la mancanza di fiducia. Agenti che non hanno mai avuto una relazione possono diventare parte di un contratto automatizzato con la certezza che tale contratto non divulgherà informazioni sensibili, e che tale contratto, sotto opportune garanzie, eseguirà in modo affidabile le transazioni mission-critical. Questo tipo di garanzie porterà le società e le agenzie governative alla blockchain - perché queste agenzie si basano sul fornire servizi mission-critical. Ma portare questi giocatori al tavolo sposta in modo efficace l'equilibro di potere verso il pubblico che mantiene un'infrastruttura così preziosa. Quella è una scala di un colore completamente diverso, per mischiare le metafore.

Esempi di contratti sociali

Riportiamo queste idee sulla terra, e iniziamo dal basso con un contratto che detiene una risorsa, come un saldo, e consente ai clienti di ottenere ed impostare il valore della risorsa.
Tale comportamento simile ad una cellula è come l'"hello world" della blockchain.

```
contract Cell( get, set, state ) = {
  for( rtn <- get; @v <- state ) {
    rtn!( v ) | state!( v ) | Cell( *get, *set, *state )
  } |
  for( @newValue <- set; _ <- state ) {
    state( newValue ) | Cell( *get, *set, *state )
  }
}
```

Questo richiede un canale per ottenere le richieste, un canale per le richieste impostate e un canale di stato dove terremo la risorsa. In parallelo, attende i canali get e set per le richieste del cliente. Esso si unisce ad un cliente in arrivo con una richiesta contro il canale di stato. Questo join fa due cose.
In primo luogo, rimuove lo stato interno dall'accesso mentre questo, a sua volta, serializza le azioni get e set, in modo che stiano sempre operando contro una singola copia coerente della risorsa.
Naturalmente, la serializzazione è ancora soggetta all'ordine di arrivo non deterministico delle richieste del cliente. Ma ci saranno alcune serializzazioni di quelle richieste che forniscono una singola cronologia di accessi e di aggiornamenti contro lo stato.

Una richiesta di ottenere arriva con un canale rtn in cui verrà inviato il valore in stato. Dal momento che il valore è stato preso dal canale di stato, viene rimandato dentro, ed il comportamento della Cella è invocato in

modo ricorsivo.. Una richiesta di impostazione arriva fornita con un nuovo valore, che viene pubblicato sul canale di stato (il vecchio valore è stato rubato dal join). Nel frattempo, il comportamento della cella arriva richiamato in modo ricorsivo.

Possiamo instanziare ed avviare la Cella con un canale di stato privato, chiamato corrente, un valore iniziale, iniziale di

```
new current in { Cell( *get, *set, *current ) | current!( initial ) }
```

e possiamo completarlo in un contratto Wallet parametrico nei canali get e set e nel valore iniziale.

```
contract Wallet( get, set, @initial ) = {
  new current in { Cell( *get, *set, *current ) | current!( initial ) }
}
```

Infine, possiamo creare un'istanza di un Wallet.

```
Wallet( *get, *set, 1000.00 )
```

Questo è un semplice contratto Wallet che può contenere un saldo (o un altro tipo di risorsa) e consentire al saldo di essere aggiornato.

Un aspetto meno desiderabile di questa implementazione è che accumulerà i thread. Per vedere questo, considera cosa succede quando servi la richiesta di un cliente sul canale get. La Cella è invocata in modo ricorsivo; tuttavia, c'è ancora un thread in attesa di servire una richiesta impostata, e un altro thread simile verrà lanciato sulla chiamata ricorsiva alla Cella. Un grande squilibrio di richieste
(o set) get, e questa implementazione esaurirà la memoria. Un'implementazione più sicura userebbe il costrutto di selezione

```
contract Cell( get, set, state ) = {
  select {
    case rtn <- get; @v <- state => {
      rtn!( v ) | state!( v ) | Cell( *get, *set, *state )
    }
    case @newValue <- set; _ <- state => {
      state!( newValue ) | Cell( *get, *set, *state )
    }
  }
}
```

Questa implementazione può essere sostituita nel contratto Wallet senza alcuna perturbazione a quel contesto di codice. Tuttavia, quando viene eseguito solo uno dei thread nella Cella può rispondere alla richiesta del cliente. È una gara, ed il thread perdente, che sia getter o setter, viene ucciso. In questo modo,

quando viene chiamata l'invocazione ricorsiva della Cella, il thread perdente non è ancora in giro, ma il nuovo processo della Cella è ancora in grado di rispondere a entrambi i tipi di richiesta del cliente.

Per i programmatori che preferiscono uno stile più orientato agli oggetti con una ricca struttura di messaggi, c'è una ulteriore terza opzione che utilizza solo un canale di richiesta del cliente e invia in base al tipo di messaggio ricevuto sul canale.

```
contract Cell( client, state ) = {
  for( request <- client; @v <- state ) {
    request match {
      case @("get", rtn) => {
        rtn!( v ) | state!( v ) | Cell( *client, *state )
      }
      case @("set", @newvalue) => {
        state!( newValue ) | Cell( *client, *state )
      }
    }
  }
}
```

Questa implementazione richiederebbe una modifica al contratto Wallet. O il contratto Wallet deve attivare le richieste sui canali get e set in nei messaggi

```
contract Adapter( get, set, client ) = {
  select {
    case rtn <- get => {
      client!( ("get", *rtn) ) |
      Adapter( *get, *set, *client )
    }
    case @newValue <- set => {
      client!( ("set", newValue) ) |
      Adapter( *get, *set, *client )
    }
  }
}

contract Wallet( get, set, @initial ) = {
  new client, current in {
    Adapter( *get, *set, *client ) |
    current!( initial ) |
    Cell( *client, *current )
  }
}
```

o deve passare ai clienti il cambiamento nell'interfaccia contrattuale.

```
contract Wallet( client, @initial ) = {
  new current in { Cell( *client, *current ) | current!( initial ) }
}
```

Anche con questo esempio di base possiamo vedere molte delle caratteristiche salienti del linguaggio.
L'esecuzione simultanea, il passaggio di messaggi asincroni e la corrispondenza dei modelli sono intrecciati insieme in un linguaggio semplice e comprensibile.

In che modo i contratti sociali differiscono gli smart contracts di Ethereum

Per cominciare, i contratti di Ethereum sono internamente sequenziali. Infatti, l'intera catena di chiamate derivante da un punto di entrata in un singolo contratto avrà un ordine seriale globale. Pensate a questo in termini di gestione della catena di fornitura. Ford Motor Company vuole serializzare la fornitura dei pneumatici con la fornitura del telaio, del motore o dell'alimentazione elettrica? Boeing vuole
serializzare il sistema di fornitura del carburante con l'illuminazione o la fornitura dei sedili interni? Le imprese sono create o devastate dalle efficienze derivanti dall'essere capaci di gestire i processi in parallelo e coordinarli contemporaneamente. Eppure, sicuramente Ford e Boeing potrebbero trarre grandi vantaggi da un sistema di gestione della catena di fornitura basata sui smart contracts. Proprio come con Haskell o ML, il modello scelto non si adatta al dominio.

In realtà, una tecnologia precedente, la modellazione dei processi aziendali, ha già esplorato proprio questa applicazione. Biztalk di Microsoft, nonché standard come BPEL, BPML, W3C Coreography, per nominarne qualcuno, hanno tutti concluso che la concorrenza era la valuta, per così dire, e hanno optato per scegliere i calcoli del processo mobile come loro base semantica. Il paradigmatico esempio applicativo nella modellazione dei processi aziendali è la gestione della catena di fornitura.

Specifiche della lingua

Il linguaggio del contratto sociale di RChain

RChain offre un linguaggio di contratto tipizzato che fornisce una semantica naturalmente adatta per il calcolo decentralizzato e distribuito. È costruito attorno alla comunicazione dei processi mobili.

Sintassi

Le specifiche della sintassi di Rholang sono in fieri; aspettati che cambino prima della pubblicazione di Mercury. In particolare, è probabile che la sintassi utilizzata sul lato sinistro di una freccia cambi; potremmo includere una versione lineare di "contratto"; potremmo includere la sintassi per denotare l'intento di
pagare per il calcolo rispetto all'intenzione di pagare solo per lo spazio di archiviazione.

Rholang è una versione più amichevole di un linguaggio più piccolo, rho calculus,che a sua volta è una variante riflessiva di ordine superiore del asynchronous polyadic pi calculus. Inizieremo con la sintassi e la semantica del calcolo rho, e quindi daremo la sintassi e la semantica di Rholang in termini del calcolo più piccolo.

Sintassi del calcolo Rho

```
M,N ::= 0          // nil o processo bloccato

      |  for( x1 <- y1; ... ; xN <- yN ) P        // input guarded agent

      |  x!( P )    // output

      |  M + N     // sum o scelta

P,Q  ::= M         // processo "normale"

      |  *x         //nome dereferenziato o non quotato

      |   P | Q     // composizione parallela

x, y ::= @P          // nome o processo quotato
```

Nomi liberi e associati

FN(0) = {}

FN(*x) = { x }

FN(for(x1 <- y1; … ; xN <- yN)P) = { y1, … , yN } U FN(P) \ { x1, … , xN }

FN(x!(P)) = { x } U FN(P)

FN(M+N) = FN(M) U FN(N)

FN(P|Q) = FN(P) U FN(Q)

Equivalenza strutturale

L'equivalenza strutturale è la più piccola congruenza, $\equiv_S$, così che

- (P,|,0) formano un monoide commutativo
- (P,+,0) formano un monoide commutativo
- If $\equiv_N$ denota l'equivalenza del nome, quindi $\equiv_S$ include l'alfa-equivalenza usando $=_N$

Equivalenza del nome

L'equivalenza del nome è la più piccola equivalenza sui nomi tale che

$P \equiv_S Q \Rightarrow @P \equiv_N @Q$

Sostituzione semantica contro sostituzione sintattica

La sostituzione usata nell'equivalenza α è in realtà solo un dispositivo per riconoscere formalmente che le occorrenze leganti non dipendono dai nomi specifici. Non è il motore del calcolo. La proposta qui è che mentre la sincronizzazione è il driver di quel motore, il vero motore del calcolo è la nozione semantica di sostituzione che riconosce che un nome dereferenziato è una richiesta per eseguire il processo il cui codice è stato associato al nome che è stato deferenziato. Formalmente, questo equivale alla nozione di sostituzione, che differisce dalla sostituzione sintattica nella sua applicazione a un nome dereferenziato.

$$*x\{ @Q / @P \} = Q \text{ if } x \equiv_N @P \text{ and } *x \text{ otherwise}$$

Relazione di riduzione

In quanto segue, => è implicazione e -> è "riduce a":

comm: xi $=_N$ xi' => R + for(x1 <- y1; … ; xN <- yN)P + S | x1'!(Q1) | … | xN'!(QN) -> P{ @Q1/y1, … , @QN/yN }
par: P -> P' => P|Q -> P'|Q
struct: P = P', P' -> Q', Q' = Q => P -> Q

Sintassi di Rholang

Abbiamo aggiunto commenti alla sintassi per indicare qualcosa della semantica voluta; le versioni future di questo documento includeranno il dispiegamento esplicito di queste produzioni nel calcolo rho.

```
<process> ::= "Nil"
        // Raggruppamento
        | "{" <process> "}"

        // Equivalente a "for" "(" [ <names> ] "<=" <name> ")" <process>
        | "contract" <name> "(" [ <names> ] ")" = <process>

        // Ricezione; solo i processi booleani ammessi in clausola if.
        | "for" "(" <receipt> ")" <process>

        // Esegue esattamente un ramo. La scelta non è deterministica.
        | "select" "{" <branch>* "}"

        // Destrutturazione
        // Se il nome non corrisponde a nessuno dei casi, l'intero processo è equivalente a Nil
        | "match" <process> "{" <case>* "}"

        // Evaluates to a Boolean.  Desugars to
        // match <process> {
        //   case <process> => true
        //   case _ => false
        // }
        | <process> "matches" <process>

        // Boolean destructuring.  Desugars to
        // match <process> {
        //   case true => <process>
        //   case false => <process>
        // }
        | "if" "(" <process> ")" <process> [ "else" <process> ]

        //  Inviare
        | <name> <send>"(" [ <processlist> ] ")"

        // invocazione del contratto
        | <name> "(" [ <processlist> ] ")"
```

```
        | <process> "|" <process>
        | "*" <name>

        // Nuovo nome nello spazio dei nomi @private.
        // I nomi privati non hanno una struttura interna visibile.
        | "new" <vlist> "in" <process>

        | <primitive>
        // sintatticamente permesso con processi arbitrari, ma il compilatore si lamenta se
        //i processi non sono del tipo giusto.
        | <process> <binop> <process>
        | <unop> <process>

        | <collection>
        | <procfield>

<names> ::= <typedname> [ "," <typedName> ]*
<typedname> ::= <name> [ ":" <process> ]
<name> ::= <var> | <quote>
// Il trattino basso serve per scartare l'associazione dalla partita
<var> ::= <identifier> | "_"
<quote> ::= "@" <process>

// Aspetta un prodotto di canali
<receipt> ::= <linreceipt> | <nonlinreceipt>
<linreceipt> ::= <names> <linarr> <name>
                            [ ";" <names> <linarr> <name>]* [ <ifclause> ]
<nonlinreceipt> ::= <names> <nonlinarr> <name> [ <ifclause> ]
<arrow> ::= <linarr> | <nonlinarr>
<linarr> ::= "<-"
// Linear-receive-then-send,  ricezione sequenziale, ricezione replicata
<nonlinarr> ::= "<!" | "<<" | "<="
// La sintassi consente qualsiasi processo, ma il compilatore si lamenta se non è un booleano
<ifclause> ::= "if" <process>

<send> ::= "!" | "!!"

<branch> ::= <linreceipt> "=>" <process>

<case> ::= "case" <process> "=>" <process>

<processlist> ::= <process> [ "," <process> ]*
```

```
<vlist> ::= <vdecl> [ "," <vdecl>]*
<vdecl> ::= <simplevdecl> | <iopairvdecl>
<simplevdecl> := <var> [ ":" <type>]
<iopaorvdecl> := "(" <var>, <var> ")" ":" "iopair" [<type>]

<primitive> ::= <boolean>
        // Numero intero a 64 bit firmato
        | <int64>
        // Delimitato da apostrofi o virgolette
        | <string>
        | "DateTime" "(" <iso8601> ")"
        // Delimited with backticks
        | <uri>

<binop> ::=
        // Processi strutturalmente equivalenti
        "=="
        | "!="
        // Operatori booleani
        | "and" | "or"
        // Numeri interi, stringhe e data / ora
        |  "+"
        // Numeri interi e stringhe
        | "*" | "%"
        // Interi
        | "-" | "/"
        // Tutti i tipi ordinati
        |  "<="  |  "<"  |  ">="  |  ">"
        // Bitfields
        |  "bitand"  |  "bitor"

<unop> ::=
        //Booleano
        "not"
        // Bitfield
        | "bitnot"
        // Numero intero
        | "+" | "-"

<collection> ::= <list> | <tuple> | <set> | <map>
<list> ::= "[" [ <processlist> ] "]"
<tuple> ::= "(" [ <processlist> ] ")"
```

<set> ::= "Set" "(" [<processlist>] ")"
<map> ::= "{" [<kvlist>] "}"
<kvlist> ::= <kvpair> ["," <kvpair>]*
<kvpair> ::= <process> ":" <process>

<procmeth> ::= <process> "." <identifier> "(" [<processlist>] ")"

Tipi spaziali

La grammatica per i tipi spaziali è la grammatica per i processi ampliati di seguito produzioni:

<process> ::= "private"
| "~" <process>
| <process> "&&" <process>
| <process> "||" <process>
| "=" <process>

Zucchero sintattico di Rholang

Replicazione

Si noti che quando si usano le regole di riduzione della sezione semantica, il processo

x!(for(y <- x){ x!(*y) | *y } | P) | for(y <- x){ x!(*y) | *y }

riduce a

P | P | ...

In altre parole, questa espressione costituisce un'implementazione della replicazione, e quindi il contesto di codice per P rappresenta una versione concorrente del famoso combinatore Y per il calcolo lambda.

È noto che la ricorsione e la replica sono inter-definibili, e fornire una versione di ricorsione dalla replica è un esercizio utile e istruttivo. A questo proposito, nota che è sempre possibile convertire un'espressione di processo protetta in una che riceverà la sua continuazione da una fonte esterna. Questo è,

for(y <- x) { P } = { k(P) | for(y <- x; p <- k) { *p } }

Questa osservazione dà luogo a una conversione dei processi in una forma normale che chiamiamo continuazione saturata. La saturazione continua ha molti utilizzi, sia in termini di memorizzazione di processi per l'esecuzione

a lungo termine, ma anche in termini di supporto del calcolo reversibile.

La saturazione continua è la chiave per una strategia di compilazione che trasforma ogni processo di Rholang in un calcolo reversibile. Questo offre una semantica naturale per le transazioni con rollback. Ai fini di questa discussione, tuttavia, si noti che fattorizzando i processi nella loro forma normale saturata continua è possibile definire una versione di replica che funziona solo per i processi protetti da input. Per uno studente della lingua ambizioso, è utile ed illuminante esercitarsi con questa versione di replica.

Persistente I/O

Quando si inviano dati sui canali, il comportamento predefinito è che sia i dati che la continuazione del consumatore siano effimeri, cioè che i dati scompaiono dal canale dopo la lettura, e il consumatore dei dati legga dal canale esattamente una volta. Questo è il comportamento che abbiamo visto fino ad ora, ed è esattamente lo stesso nel calcolo sottostante. Però, forniamo anche zucchero sintattico per quando i dati, la continuazione, o entrambi sono persistenti, e ciò dà origine a modelli piuttosto interessanti e pratici che consentono ai canali di emulare, per esempio, flussi di dati e posizioni di memoria.

Questa sezione descrive le diverse combinazioni e spiega come vengono de-zuccherate verso il comportamento predefinito. Nell'interesse della chiarezza, consideriamo tutte le coppie della forma:

consumatore di dati | *produttore di dati*

Dati effimeri, continuazione effimera

Questa è l'espressione standard del calcolo del processo in cui sia la continuazione che i dati sono effimeri. Ovvero, il canale è usato esattamente una volta dal consumatore, e i dati sono rimossi dal canale una volta letti:

```
for(v <- channel) { P } | channel!(Q)
```

Dati persistenti, continuazione effimera

Ciò significa che il canale viene utilizzato come un insieme di valori di sola aggiunta, vale a dire ogni valore rimarrà nel canale anche dopo che è stato letto.

```
for(v <- channel) { P } | channel!!(Q)
```

Tuttavia, a volte è utile per il consumatore che questi diano solo un'occhiata al valore in un canale, anche se produttore non ha specificato che dovrebbe persistere. Questo si ottiene usando l'operatore receive-then-resend **<!** , come qui:

```
for(v <! channel) { P } | channel(Q)
```

È zucchero sintattico per

```
for(v <- channel) { channel!(*v) | P } | channel(Q)
```

Naturalmente, questi due possono essere combinati. Sia il produttore che il consumatore potrebbero volersi assicurare che il valore rimanga nel canale, cosa che esprimono tramite:

```
for(v <! channel) { P } | channel!!(Q)
```

De-zuccherando questo usando le regole date nei due esempi precedenti, vediamo che questo è lo zucchero per:

```
for(v <- channel) { channel!(*v) | P } | channel!!(Q)
```

Dati effimeri, continuazione persistente

Se il consumatore vuole ascoltare un canale come uno stream, vuole eseguire la stessa continuazione per ogni messaggio ricevuto. In altre parole, la continuazione dovrebbe persistere. La freccia <= indica che la produzione dovrebbe essere replicata.

```
for(v <= channel) { P } | channel!(Q)
```

Dati persistenti, continuazione persistente

Infine, arriviamo al caso in cui sia i dati che la continuazione dovrebbero persistere. Il seguente idioma esprime un calcolo illimitato:

```
for(v <= channel) { P } | channel!!(Q)
```

Vale a dire, lo stesso valore dovrebbe essere inviato sul canale infinitamente volte, e dovrebbe essere letto e passato alla continuazione infinite volte.

Questo potrebbe essere ciò che un programmatore intende, dal momento che non è un problema di vividezza. Nota che

```
for(v <= channel) { P } | channel!!(Q) | channel!!(Q')
```

è un processo in cui persistono sia i valori dei dati Q e Q' che la continuazione P.

D'altra parte, un programmatore potrebbe volere un processo in cui la continuazione attenda un segnale su qualche altro canale, e solo in seguito gestisca i dati:

```
for (_ <= signal) { for (v <- channel) { P } | channel!(Q) }
```

Ogni volta che viene ricevuto un messaggio sul canale del segnale, un nuovo valore di dati effimeri viene prodotto.

Comunicazione seriale

Se una lista finita è conosciuta in anticipo, può essere inviata in un unico messaggio:

```
for (@list <- channel) { print!(list) } | channel!([4, 5, 6])
```

Se la lista viene generata un termine alla volta, l'intero processo diventa molto più complicato:

```
new printEach, iterate, channel, ack in {
        contract printEach() = {
                for (@item, @done <- channel) {
                        print!(item, *ack) |
                        if (done) { printEach() } else { Nil }
                }
        } |

        contract iterate(@list, @i, @limit) = {
                channel!(list[i], i < limit) |
                for(_ <- ack) {
                        if (i < limit) { iterate(list, i + 1, limit) } else { Nil }
                }
        } |

        printEach() | iterate([4, 5, 6], 0, 3)
}
```

Questo schema è abbastanza comune da meritare un po' di zucchero. Il processo

```
for ( v, ack << channel ) { P }
```

desugars to

```
for (ack <- channel) {
        ack!() |
        new left in {
                for (_ <= left) = {
                        for (v, keepGoing <- channel) {
                                P |
                                if (keepGoing) { left!() } else { Nil }
                        }
                }
        }
}
```

dove "left" e "done" sono freschi rispetto a P. La de-zuccherazione aspetta prima un canale "ack" per confermare la ricezione dei messaggi, quindi attende le coppie costituite da un valore v ed un booleano. P quindi gestisce v e conferma la ricezione, mentre il resto del processo o aspetta un'altra coppia o si chiude, a seconda del flag "done".

Allo stesso modo, le Liste hanno un metodo iterato che interagisce con il codice sopra. Il processo

Coll.iterate

è definito essere

```
contract iterate(channel) = {
        new ack, right in {
                channel!(*ack) |
                for (_ <- ack) {
                        for(@i, @limit <= right) {
                                channel!(Coll[i], i < limit) |
                                for (_ <- ack) {
                                        if (i < limit) { right!(i + 1, limit) } else { Nil }
                                }
                        } |
                        right(0, Coll.size)
                }
        }
}
```

in modo che il processo

```
for ( v, ack << channel ) { print!(*v, *ack) } | [4, 5, 6].iterate(*channel)
```

invochi "print" tre volte, una volta ciascuna su 4, 5 e 6. Gli Insiemi e le Mappe hanno metodi che restituiscono liste di dati su cui è possibile chiamare il metodo iterativo.

Primitivi e funzioni estranee

Tutti i valori, come booleani, interi, stringhe, datatempo, URI e raccolte di questi, sono processi. Ciò significa che @ 57, @ "joe", @ (123, false), @ {"hello" => "world"}, ecc., sono tutti nomi pubblici su cui i processi possono comunicare. Si noti che le regole di sintassi impediscono a un processo P dall'ascolto su @P, quindi non c'è un senso in cui @ 57 comunica con il numero 57.

Le funzioni possono essere incorporate come processi che accettano l'input e un canale di ritorno, quindi inviano il risultato sul canale di ritorno. Ad esempio, la funzione JavaScript

```
function inc(x) { return x+1; }
```

potrebbe essere incorporata in Rholang come

```
contract inc(@x, ret) = { ret!(x + 1) }.
```

Non esiste una disposizione esplicita per le funzioni estranee nella sintassi di Rholang. Invece, le piattaforme potrebbero esporre funzionalità come nomi pubblici speciali. Nota che possiamo attenuare l'autorità che questi nomi pubblici forniscono utilizzando il sistema di tipi per limitare i nomi che un pezzo

codice può inviare. Ad esempio, se insistiamo sul fatto che un pezzo di codice comunichi solo sui nomi nello spazio dei nomi @private, allora l'unico modo in cui il codice potrebbe utilizzare un nome pubblico è attraverso un processo proxy in ascolto su un nome privato che inoltra al nome pubblico. Il proxy è quindi in una posizione per filtrare i messaggi dal codice confinato

Semantica

La semantica operazionale di Rholang può essere data come teoria nominale multisortata di Lawvere che descrive il grafico di termini e riscritture. I due tipi di teoria sono T per i termini e R per riscritture. Ogni produzione grammaticale corrisponde a un simbolo di funzione che prende alcuni numeri di sottotemi in un termine più ampio; per esempio, la produzione <process> | <Process> corrisponde a un simbolo di funzione

$$|:T^2 \to T$$

portando (P, Q) a P | Q. Ogni regola di riduzione corrisponde a un simbolo di funzione che prende alcuni numeri di sottotemi da riscrivere dal contesto iniziale di quei termini al contesto finale; ad esempio, la regola di riscrittura comm corrisponde a un simbolo di funzione

$$comm:T^4 \to R$$

prendendo (P, Q, R, S) per riscrivere

$$\text{for (@S <- @P) \{ Q \} | @P!(R) -> Q\{ @R / @S \}.}$$

Le regole di equivalenza strutturale corrispondono alle equazioni tra simboli di funzioni; per esempio,

$$P \mid Q \equiv Q \mid P.$$

Guida per le implementazioni

Ignorando le sfumature sulla struttura dei nomi, ecco un ragionevole rendering delle semantiche della concorrenza del core nel codice Scala utilizzando il framework di attori Akka. L'interpretazione è una funzione con la firma

$$[\![-]\!](-): \text{Rholang} \times \text{Map(Symbol, Queue)} \to \text{Scala}.$$

Rholang	Scala
0	{ }
x!(y1, …, yn)	[\[x\]](m) ! ([\[y1\]](m), …, [\[yn\]](m))

for ((y1, …, yn) <- x) { P }	for ((y1, …, yn) <- [\[x\]](m)) {
	  [\[P\]](m)(y1, …, ym)
	}

P | Q	spawn { [\[P\]](m) }; spawn { [\[Q\]](m) }

new x in P	{ val q = new Set(); ⟦P⟧(m + ("x" <- q)) }

Limitazione di velocità

Le risorse computazionali disponibili per un contratto devono essere limitate alle risorse che il contratto ha ottenuto giustamente. A tal fine, ogni azione significativa dal punto di vista computazionale che si possa esprimere in Rholang ha un costo corrispondente che il contratto deve pagare prima di eseguire l'azione.

Idealmente, vorremmo esprimere il modello di costo come un'estensione del calcolo di base, in modo da ragionare formalmente sulla complessità. Introduciamo un Sistema di Transizione Etichettato (LTS) per il calcolo RHO. Quindi estendiamo la LTS con la semantica dei costi e, infine, introduciamo un meccanismo di limitazione della velocità integrato che "dinamizza" i contratti di Rholang.

Semantica della transizione etichettata

I Sistemi di Transizione etichettati sono degli strumenti utili per esaminare l'evoluzione del tempo lineare di un sistema mantenendo le informazioni sulla cronologia del sistema. La relazione di transazione etichettata per il calcolo rho è della forma

$$A \vdash P \xrightarrow{\ell} Q$$

dove c'A insieme finito di nomi e $fn(P) \subseteq A$. La relazione sopra è letta: "In uno stato dove i nomi A sono noti P, P possono fare ℓ per diventare Q. Le etichette, $\ell \in \mathcal{L}$ sono definite da la grammatica

$\ell ::= \tau$	internal transition
ρ	unquote $@Q$ to Q
$x!(@Q)$	output $@Q$ on x
$for(@Q \leftarrow x)$	input $@Q$ on x

I nomi gratuiti delle etichette sono $fn(\tau) = \varnothing = fn(\rho)$, $fn(x!(@Q)) = \{x, @Q\} = fn(for(@Q \leftarrow x))$.
I lettori familiari riconosceranno l'aggiunta dell'etichetta di rilascio, ρ,come la nostra prima partenza dalla semantica di transizione come formulata per il π-calculus. Si noti che l'etichetta di rilascio non abbia nomi vincolati. bound names.

$$\text{LIFT}: \frac{}{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0} \qquad \text{IN}: \frac{}{A \vdash for(z \leftarrow x)P \xrightarrow{for(@Q \leftarrow x)} P\{@Q/z\}}$$

$$\text{SUM}: \frac{A \vdash P \xrightarrow{\ell} P'}{A \vdash P + Q \xrightarrow{\ell} P'} \qquad \text{TAU}: \frac{}{A \vdash \tau.P \xrightarrow{\tau} P}$$

$$\text{COMM}: \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P' \qquad A \vdash Q \xrightarrow{x!(@Q)} 0}{A \vdash P \xrightarrow{\ell} P'}$$

$$A \vdash P|Q \xrightarrow{\tau} P' \qquad \text{PAR} : \frac{}{A \vdash P|Q \xrightarrow{\ell} P'|Q}$$

$$\text{RHO} : \frac{}{A \vdash *q \xrightarrow{\rho} Q}$$

Con le etichette come punti di riferimento, alcune relazioni spaziotemporali diventano più semplici da esprimere.Ad esempio, una traccia parziale può essere scritta

$$A_1 \vdash P_1 \xrightarrow{\ell_1} ... \xrightarrow{\ell_n} P_{n+1}$$

ed espressa formalmente,

$$\exists P_2, ..., P_n, A_2, ..., A_n. \forall i \in 1, ..., n. A_{i+1} = A_i \cup fn(\ell_i) \wedge A_i \vdash P_i \xrightarrow{\ell_i} P_{i+1}$$

Nel caso $fn(P) \subseteq A$, una traccia parziale P rispetto a A is just

$$\text{ptrn}_A(P) = \{\ell_1, ..., \ell_2 \mid \exists P'. A \vdash P \xrightarrow{\ell_1} ... \xrightarrow{\ell_n} P'\}$$

Semantica dei costi

In generale, un modello di costo è una funzione $\mathcal{M} : \mathcal{L} \to \mathbb{Q}^+$ che assegna a ciascuna etichetta un numero razionale positivo che riflette un'interpretazione astratta della complessità del calcolo rappresentato dall'etichetta. Il costo di una singola transizione è quindi espresso come $\mathcal{M}(\ell) = k_\ell \in \mathbb{Q}^+$Ricaviamo una semantica dei costi dalle regole presentate sopra, dove i giudizi sono nella forma

$$\text{NORM} : \frac{A \vdash P \xrightarrow{\ell} Q}{A \vdash k_P = \mathcal{M}(\ell) + k_Q}$$

e possono essere letti: "In uno stato in cui i nomi A possono essere conosciuti P, se P può farlo ℓ diventare Q quindi il costo di P è il costo di ℓ più il costo di Q. Il costo di P è calcolato ricorsivamente sommando il costo di ℓ e il costo della continuazione Q.

$$\text{LIFT} : \frac{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0}{A \vdash k_{x!(Q)} = \mathcal{M}(x!(@Q))} \qquad \text{IN} : \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P'}{A \vdash k_P = \mathcal{M}(for(@Q \leftarrow x)) + k_{P'}}$$

$$\text{SUM} : \frac{A \vdash P + Q \xrightarrow{\ell} P'}{A \vdash k_{P+Q} = \mathcal{M}(\ell) + k_{P'}} \qquad \text{TAU} : \frac{A \vdash \tau.P \xrightarrow{\tau} P}{A \vdash k_{\tau.P} = \mathcal{M}(\tau) + k_P}$$

$$\text{COMM} : \frac{A \vdash P|Q \xrightarrow{\tau} P'}{A \vdash k_{P|Q} = \mathcal{M}(\tau) + k_{P'}} \qquad \text{PAR} : \frac{A \vdash P|Q \xrightarrow{\ell} P'|Q}{A \vdash k_{P|Q} = \mathcal{M}(\ell) + k_{P'|Q}}$$

$$\text{RHO} : \frac{A \vdash *q \xrightarrow{\rho} Q}{A \vdash k_{*q} = \mathcal{M}(\rho) + k_Q}$$

Il costo di una A-parziale traccia n con transizioni è semplicemente la somma del costo delle etichette che delimitano la traccia.

$$\text{ptr}_A(P)_k = \sum_{i=1}^{n} \mathcal{M}(\ell_i)$$

Questi particolari giudizi di costo non sono regole per la stima statica del costo di un contratto, ma regole per dedurre il costo dinamico di un contratto. Con questi in atto, una nozione formale di risorsa può essere introdotta.

Semantica limitante della velocità

Su RChain, le risorse disponibili per un contratto sono denominate in "phlogiston" - una metrica simile al "gas" di Ethereum. Durante l'inizializzazione, una quantità iniziale di phlogiston, $ph \in \mathbb{Q}^+$ è assegnata al contratto. Dal saldo iniziale, ad ogni sottoprocesso nel contratto è assegnata una certa quantità di phlogiston.

Il saldo del phlogiston di P è denotato P_{ph} e, per ogni transizione in P una certa quantità di di phlogiston che rappresenta il costo della transizione viene dedotta. Il consumo di risorse è quantificato utilizzando i giudizi della forma

$$\text{NORM} : \frac{A \vdash P \xrightarrow{\ell} Q}{A \vdash Q_{ph} = P_{ph} - \mathcal{M}(\ell)}$$

dov'è P_{ph} il saldo del phlogiston P prima della transizione ed Q_{ph} è il bilancio di phlogiston di Q di dopo la transizione. Avendo stabilito la semantica dei costi presentata nella sezione precedente, le regole semantiche per la limitazione della velocità sono semplici:

$$\text{LIFT}: \frac{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0}{A \vdash 0_{ph} = x!(Q)_{ph} - \mathcal{M}(x!(@Q))} \qquad \text{IN}: \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P'}{A \vdash P'_{ph} = P_{ph} - \mathcal{M}(for(@Q \leftarrow x))}$$

$$\text{SUM}: \frac{A \vdash P + Q \xrightarrow{\ell} P'}{A \vdash P'_{ph} = P_{ph} + Q_{ph} - \mathcal{M}(\ell)} \qquad \text{TAU}: \frac{A \vdash \tau.P \xrightarrow{\tau} P}{A \vdash P_{ph} = \tau.P_{ph} - \mathcal{M}(\tau)}$$

$$\text{COMM}: \frac{A \vdash P|Q \xrightarrow{\tau} P'}{A \vdash P'_{ph} = P_{ph} + Q_{ph} - \mathcal{M}(\tau)} \qquad \text{PAR}: \frac{A \vdash P|Q \xrightarrow{\ell} P'|Q}{A \vdash P'_{ph} = P_{ph} - \mathcal{M}(\ell)}$$

$$\text{RHO}: \frac{A \vdash *q \xrightarrow{\rho} Q}{A \vdash Q_{ph} = *q_{ph} - \mathcal{M}(\rho)}$$

Condizioni di interruzione

Le regole di cui sopra hanno un'ipotesi implicita che per qualsiasi transizione si verifichi lungo $A_1 \vdash P_1 \xrightarrow{\ell_1} ... \xrightarrow{\ell_n} P_{n+1}$,la transizione contrassegnata da ℓ è abilitata perché il saldo di phlogiston di of P_{i+1} è maggiore o uguale a zero.In attuazione, questa condizione è considerata come una una invariante che deve essere verificata e soddisfatta prima di eseguire la transizione successiva.
Le seguenti regole definiscono l'interruzione delle condizioni in base al bilancio dei flussi di phlogiston:

$$\text{ERR LIFT}: \frac{A \vdash x!(Q) \xrightarrow{x!(@Q)} 0 \qquad 0_{ph} < 0}{A \vdash x!(Q) \to 0} \qquad \text{ERR IN}: \frac{A \vdash P \xrightarrow{for(@Q \leftarrow x)} P' \qquad P'_{ph} < 0}{A \vdash P \to 0}$$

$$\text{ERR SUM}: \frac{A \vdash P + Q \xrightarrow{\ell} P' \qquad P'_{ph} < 0}{A \vdash P + Q \to 0} \qquad \text{ERR TAU}: \frac{A \vdash \tau.P \xrightarrow{\tau} P \qquad P_{ph} < 0}{A \vdash \tau.P \to 0}$$

$$\text{ERR PAR}: \frac{A \vdash P|Q \xrightarrow{\ell} P'|Q \qquad P'_{ph} < 0}{A \vdash P|Q \to Q} \qquad \text{ERR COMM}: \frac{A \vdash P|Q \xrightarrow{\tau} P' \qquad P'_{ph} < 0}{A \vdash P|Q \to 0}$$

$$\text{ERR RHO}: \frac{A \vdash *q \xrightarrow{\rho} Q \qquad Q_{ph} < 0}{A \vdash *q \to 0}$$

Le regole di cui sopra richiedono l'interruzione di qualsiasi processo se il suo saldo di phlogiston non è sufficiente per coprire

il costo della transizione successiva. Per quanto riguarda$\mathrm{ERR\ LIFT}$ notare la distinzione tra il saldo di phlogiston del processo null 0_{ph} e il processo null stesso, anche (sfortunatamente) denotato da 0. Inoltre, l'iscrizione aggiuntiva rappresenta$\mathrm{ERR\ SUM}$ ancora una volta una scelta non deterministica, non una aggiunta come nelle regole della sezione precedente.

Rispetto a $\mathrm{ERR\ PAR}$, si noti che se P e Q procedono indipendentemente, e $P'_{ph} \not< 0$ l'esecuzione di Q è inalterata. Intuitivamente, ha senso che il bilancio di phlogiston di un processo non abbia influenza su un altro processo indipendente.

Le pubblicazioni future definiranno i costrutti di programmazione più imperativi di Rholang, cioè espressioni aritmetiche, case statement, ecc. in modo simile. Questi sono relativamente semplice da definire. Inoltre, le formulazioni future includeranno annotazioni di tipo e quantificazione di memoria, archiviazione e attività di rete nel modello di costo.

Conclusione

Facendo un passo indietro possiamo vedere come i contratti sono davvero solo zucchero sintattico per processi ".", ed i processi sono fatti di processi "." La composizionalità che vediamo nel mondo naturale, questo principio di "come sopra, così in basso" che riflette l'albero nella struttura della foglia, questo è evidente nella struttura degli smart contract sociali come quelli sviluppati da startup come Resonate Coop; e il trucco della Natura di lasciare andare l'autorità centrale a favore di autonomia e indipendenza, ciò è evidente anche nella struttura degli smart contract sociali sulla piattaforma RChain. Questo è proprio ciò che intendiamo quando diciamo che un processo è l'esecuzione concomitante di diversi processi.