Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il management

UNA PROPOSTA DI SEMANTICA DI GO

Relatore:
Chiar.mo Prof.
DAVIDE SANGIORGI

Presentata da:
ARNALDO CESCO

Sessione II
Anno Accademico 2017-2018

Agli 01011001 ed all' α-Team ...

Introduzione

La concorrenza è una delle materie più importanti dell' informatica odierna: sono sistemi caratterizzati da concorrenza reti di sensori, sistemi distribuiti, database ecc... La sua gestione non è semplice, perchè gli umani sono abituati a ragionare in modo lineare; analogamente, anche nel campo dei linguaggi di programmazione i leader di mercato non la includono fra le loro caratteristiche di base. Da una decina di anni a questa parte però si assiste al rapido sviluppo di nuovi linguaggi con particolare focus sulla concorrenza, come Rust[1], Jolie[2] o Go[3].
Di questi, Go sembra essere il più promettente a causa del supporto di Google (casa madre del linguaggio), scala (è già utilizzato in molti server dell'azienda di Mountain View), distribuzione open-source.

Tuttavia, la letteratura non si è ancora concentrata nella formalizzazione della gestione della concorrenza nei nuovi linguaggi che stanno nascendo; ad esempio, riguardo a Go compare solo un generico riferimento al process calculus "Communicating sequential processes" di C.A.R. Hoare[4] all'interno delle specifiche del linguaggio[5].

Questo lavoro propone una semantica di Go formalmente specificata, partendo da un sottoinsieme ristretto ma Turing-completo di istruzioni del linguaggio, "ChamGo", che include alcune primitive del linguaggio giudicate interessanti per quanto riguarda la programmazione concorrente. L'approccio utilizzato nella definizione è quello della Reduction Semantics à la Milner[6] nella variante "chemical abstract machine"[7].

Indice

Introduzione **i**

1 Go e ChamGo **2**

1.1 Introduzione a Go 2

1.1.1 Cenni sulla struttura di un programma Go 3

1.1.2 Cenni riguardo alla concorrenza in Go 3

1.1.3 Un esempio di programma Go 4

1.2 ChamGo 5

1.2.1 Costrutti sulla concorrenza 5

1.2.2 Altri costrutti 6

1.2.3 Alcune considerazioni 7

1.3 La grammatica di ChamGo 7

2 Semantica di ChamGo **8**

2.1 Chemical abstract machine 8

2.2 La chemical abstract machine di ChamGo 9

2.2.1 Le molecole 9

2.2.2 Grammatica della chemical abstract machine di ChamGo 10

2.3 Regole della cham 10

2.3.1 Regole di riscrittura 11

2.3.2 Regole di membrana 11

2.3.3 Regole di computazione 11

2.4 Un esempio: Fibonacci parallelo 14

2.4.1 Fibonacci parallelo in Go 14
2.4.2 Fibonacci parallelo in ChamGo 15

3 Lavoro futuro e conclusioni **19**

Bibliografia **22**

Ringraziamenti **23**

Capitolo 1

Go e ChamGo

1.1 Introduzione a Go

Go nasce nel 2007 dal lavoro di un team coordinato da R. Pike all'interno di Google con l'obiettivo di essere "an open source programming language that makes it easy to build simple, reliable, and efficient software"[3]. È un linguaggio compilato, a tipizzazione forte e statica, con una sintassi simile a C.

Lo sviluppo di Go ha seguito due direttrici: l'efficienza - il linguaggio consente performance vicine al C++ ma gestisce autonomamente la memoria e possiede un garbage collector - e la semplicità. Infatti in Go sono presenti caratteristiche di diversi paradigmi di programmazione: oggetti e metodi (*OOP*), funzioni di prima classe e strutture non modificabili(*funzionale*), gestione avanzata della concorrenza attraverso *canali*. Molte caratteristiche sono dovute alla necessità di semplicità per un programmatore che viene da un altro linguaggio. Ad esempio, esistono *de facto* due modelli di concorrenza: uno basato sui canali secondo il modello CSP, ed uno basato su lock e semafori, che non sarà considerato in questo lavoro perchè simile ad altri già noti in letteratura.
Ancora per ragioni di semplicità, Go non consente ereditarietà, programma-

zione con *generics*, coversioni di tipo implicite ed un meccanismo simile alle eccezioni è presente ma viene suggerito di usarlo solo in casi estremi[8].

1.1.1 Cenni sulla struttura di un programma Go

Un programma go consiste nella dichiarazione di un package a cui il programma appartiene, eventuali `import` da altri package, dichiarazione di funzioni tra cui, obbligatoriamente, la funzione `func main()`.
I package sono librerie di codice; attraverso il costrutto `import` si possono importare package dalla directory. Importare un package significa introdurre nelle funzioni utilizzabili dal programma quelle che nel package sono dichiarate con una lettera maiuscola a inizio nome.
Le funzioni sono di prima classe (in ChamGo non trattate) e possono essere associate a degli oggetti, diventando così dei metodi. Non è necessario che un programma dichiari altre funzioni oltre al main. Il main esegue all'interno della `main-goroutine`; quando questa goroutine termina, termina l'esecuzione dell'intero programma, comprese altre goroutine.

1.1.2 Cenni riguardo alla concorrenza in Go

"Concurrency and multi-threaded programming have a reputation for difficulty. [...] One of the most successful models for providing high-level linguistic support for concurrency comes from Hoare's Communicating Sequential Processes, or CSP. [...] Experience with several earlier languages has shown that the CSP model fits well into a procedural language framework."[8]

La concorrenza nel linguaggio è gestita secondo un modello simile a CSP i cui elementi base sono canali e goroutine. Le goroutine sono funzioni che eseguono indipendentemente e concorrentemente con le altre goroutine. Ogni goroutine ha il proprio stack, ma non è un thread: la sua gestione è affidata al runtime di Go e non al sistema operativo.
Un canale permette a due o più goroutine di comunicare. Come ultimo

linguaggio, in ordine di uscita, della famiglia Newsqueak-Alef-Limbo [9] i canali sono oggetti di prima classe, cioè possono essere assegnati a variabili, passati come parametri, ritornati come risultato di una computazione, inclusi in strutture dati. I canali sono tipati ed hanno scope a livello di blocco. Un canale può essere *buffered* o *unbuffered*, cioè avere o meno un numero massimo di valori che può contenere. L'output su canale generalmente non è bloccante, ad eccezione di quando un canale buffered ha già raggiunto la sua capienza massima; l'input si comporta in maniera analoga, cioè è bloccante solamente quando su un canale non sono presenti valori.

1.1.3 Un esempio di programma Go

```
package main                          //dichiarazione del package

import "fmt"                          //import della standard library
                                      //per l'output
func add(wait chan bool, x*int){ //dichiarazione di funzione con parametri
                                      //un canale e un puntatore
 *x++  //incrementa di uno la variabile  x
 wait <- true                         //invia true sul canale wait
}

func main(){                         //funzione main
 wait := make(chan bool, 1)          //crea un canale di bool con size 1
 x:=0                                //dichiara una variabile x inizializzata a 0
 go add(wait, &x)                    //spawn della goroutine add
 <-wait                              //attende un output su wait
 x++                                 //incrementa x di 1
 fmt.Println(x)                      //stampa x usando la funzione della standard library
}
```

Non si ha race condition nell'incremento di x perchè l'input da canale è

bloccante, ottenendo così una specie di semaforo. L'esecuzione di questo programma stamperà sempre 2.

1.2 ChamGo

ChamGo contiene tutti i costrutti principali riguardanti canali e goroutine, più alcuni costrutti necessari per la creazione di programmi minimamente significativi. Non sono inclusi import e package, tuttavia i costrutti scelti sono un sottoinsieme turing completo perchè permettono la ricorsione. Di seguito ne è mostrata una rassegna dettagliata. I costrutti riguardanti i canali (e le variabili) sono scritti utilizzando una sintassi simile a CCS/π-calcolo [10] [6]. Il type checking è dato per assunto.

1.2.1 Costrutti sulla concorrenza

- `ch := make(chan <tipo>, <taglia>)` crea un nuovo canale *ch* ha tipo $< type >$ e capienza massima $< size >$. In ChamGo l'istruzione analoga è νch[size], essendo il type checking assunto. Se $< size >= 0$ il canale è considerato *unbuffered*.

- `<-c` legge un valore dal canale *c*. In ChamGo l'istruzione analoga è *c*.

- `ch<- <valore>` inserisce il valore *c* nel canale *ch*.In ChamGo l'istruzione analoga è $\overline{ch}\ v$.

- `go f(<valore1>, <valore2>...)` genera una nuova *goroutine* f. I parametri sono calcolati nella goroutine generatrice e l'esecuzione di *f* si ha nella generata. Un programma Go termina quando termina il main, quindi eventuali *goroutine* che stanno ancora eseguendo possono essere terminate ex abrupto. In ChamGo l'istruzione analoga è *go* $f(\overrightarrow{v})$.

1.2.2 Altri costrutti

- `x := <valore>` crea una variabile x inizializzata con il valore indicato. In ChamGo l'istruzione analoga è νx[v].

- `func f(`$< nome >_i$ `<`$tipo_i$`> ){ ... }` dichiara la funzione f. Una funzione è un insieme di istruzioni parametrizzato. Inoltre, le funzioni sono *cittadini di prima classe*, in questo lavoro non trattate.In ChamGo l'istruzione analoga è $f\langle\overrightarrow{z}\rangle$.

- `f(<valore1>, <valore2>...)` esegue la funzione *f*. È possibile la dichiarazione di funzioni anonime come di funzioni standard. In ChamGo l'istruzione analoga è $f(\overrightarrow{v})$.

- `if <operazione booleana>` {`<istr1>`} `else` {`<istr2>`} consente l'esecuzione condizionale di blocchi di codice in base al risultato (booleano) di una operazione. In ChamGo l'istruzione analoga è if $op(\overrightarrow{z})$ then $\{p_1\}$ else $\{p_2\}$.

- `x= <valore>` assegna un valore ad una variabile. Il valore può essere hard-coded oppure il risultato di una operazione su valori (eventualmente anche su variabili), oppure l'input da un canale. In ChamGo l'istruzione analoga è $x := v$.

A questi costrutti è necessario aggiungere l'*operatore sequenziale* e il processo vuoto. Il primo permette di eseguire prima l'istruzione alla sua sinistra e poi l'istruzione alla sua destra. In Go esso è indicato con un "a capo", in ChamGo si utilizza ;. Una semantica più dettagliata dell'operatore sequenziale è fornita nel capitolo successivo, poichè la scelta di usare reduction semantics non permette un dettaglio semplice di questo costrutto. Il secondo costrutto è un semplice artificio per indicare la fine della computazione, non presente in Go ed indicato in ChamGo con 0.

1.2.3 Alcune considerazioni

Questo sottoinsieme di costrutti è Turing completo, poichè permette la ricorsione. Tuttavia grazie alla presenza dell'operatore sequenziale è possibile ed interessante aggiungere un altro costrutto relativo alla concorrenza: la select, che aggiunge possibilità di comportamento parametrico in base al canale di input.

- `select` consente di eseguire blocchi diversi a seconda del canale da cui si è ricevuto un input. In ChamGo l'istruzione analoga è $\sum_{i=1}^{n} c_i; p_i + p'$.

Per distinguere le istruzioni del main dalle altre si usa una sottolineatura, ad esempio $\underline{\nu \mathrm{c}[5]}$.

1.3 La grammatica di ChamGo

Indicati tutti i costrutti di ChamGo , se ne fornisce la grammatica. Per indicare valori concreti si usa l'insieme $\mathbb{V} = \{v, w \ldots\}$, per le variabili l'insieme $\aleph = \{x, y \ldots\}$, per i canali l'insieme $\Gamma = \{c, d, ...\}$, per le funzioni un insieme infinito $\mathbb{F}$ di *nomi di funzione*.

$$Program ::= [Function^*]Main$$
$$Function ::= f\langle \overrightarrow{z} \rangle = p$$
$$Main ::= \underline{p}$$
$$p ::= \nu \mathrm{c}[\mathrm{v}] \mid \overline{c}\ z \mid \nu \mathrm{x}[\mathrm{v}] \mid x := A \mid f(\overrightarrow{v}) \mid go\ f(\overrightarrow{v}) \mid \text{if } op(\overrightarrow{z}) \text{ then } \{p_1\} \text{ else } \{p_2\} \mid \sum_{i=1}^{n} c_i; p_i + p' | p; p | 0$$
$$A ::= v | op\overrightarrow{v} | c$$
$$z ::= x | v$$

Capitolo 2

Semantica di ChamGo

In questo capitolo si fornisce la semantica del sottoinsieme di Go proposto al capitolo 1 utilizzando una sintassi CCS-like [10] e basandosi sulla Reduction Semantics di Milner et al. [6] nella variante della Chemical Abstract Machine [7]. Inizialmente il lavoro si è focalizzato nella realizzazione di una semantica di tipo *structural operational*, procedendo molto avanti; successivamente questo approccio è stato scartato per ragioni di eleganza, semplicità e chiarezza. La reduction semantics infatti permette regole molto più concise e consente di rendere meglio processi che possono evolvere contemporaneamente in direzioni diverse grazie alla possibilità di rompere la struttura rigida della sintassi.

2.1 Chemical abstract machine

La chemical abstract machine non è basata su un modello rigidamente geometrico: i componenti della computazione possono muoversi liberamente nel sistema e comunicare tra loro. Lo stato di un sistema è descritto come una soluzione chimica dove delle *molecole* possono interagire con altre attraverso regole di reazione (computazione vera e propria) ed esistono meccanismi meccanici che, come fa il moto browniano, consentono di muovere la soluzione spostando le molecole (riscrittura). La trasformazione della so-

luzione è ovviamente parallela: un numero qualunque di reazioni può essere eseguito parallelamente, se queste coinvolgono molecole diverse.Le soluzioni sono multiinsiemi di molecole, così da garantire implicitamente associatività e commutatività della composizione parallela.
Per modellare la programmazione gerarchica, si introduce la possibilità che alcune molecole siano contenute in una sottosoluzione racchiusa da una membrana porosa; esistono regole (di membrana) che regolano il modo in cui può avvenire la comunicazione con la soluzione esterna. Infine, si specifica un insieme di regole che permette di produrre nuove molecole a partire da quelle esistenti: al contrario delle classiche regole di inferenza, per esempio della *structural operational semantics*, queste regole non hanno premesse e sono puramente locali.

2.2 La chemical abstract machine di ChamGo

2.2.1 Le molecole

All'interno della soluzione, le molecole possono essere:

- **Composizione parallela di molecole**: consente alle molecole di spostarsi all'interno della soluzione ed interagire. P_1, P_2
- **Variabili**: ognuna delle quali rappresentante un valore concreto. $x = 2$
- **Canali**: ogni canale è costituito da una lista ordinata (FIFO) di valori che possono essere prelevati o immessi attraverso le operazioni di input ed output. È possibile che il canale abbia un limite massimo di lunghezza, nel caso esso è indicato dal valore in apice. $c = \{\ 2,3,4\ \}^3$
- **Funzioni**: ogni funzione ha un nome, un vettore di parametri ed è composta da dei costrutti. Le funzioni sono molecole sempre presenti all'interno della soluzione; quando invocate i parametri sono sostituiti da valori concreti e le funzioni replicate.

- **Membrane**: possono essere di due tipi: relative a variabili (νx) o relative a canali (νch). Le regole *di membrana* ne consentono la porosità e sono usate principalmente per modellare lo scope di un oggetto.
- **Costrutti**: i costrutti mostrati nel cap.1

2.2.2 Grammatica della chemical abstract machine di ChamGo

Alla grammatica di un programma ChamGo definita in precedenza, si aggiungono gli elementi che modellano molecole, membrane e soluzione.

$$Program ::= [Function^*]Main$$
$$Function ::= f\langle \overrightarrow{z} \rangle = p$$
$$Main ::= \underline{p}$$
$$p ::= \nu\text{c[v]} \mid \overline{c}\ z \mid \nu\text{x[v]} \mid x := A \mid f(\overrightarrow{v}) \mid go\ f(\overrightarrow{v}) \mid \text{if } op(\overrightarrow{z}) \text{ then } \{p_1\} \text{ else } \{p_2\} \mid \sum_{i=1}^{n} c_i; p_i + p' \mid p; p \mid 0$$
$$A ::= v \mid op(\overrightarrow{z}) \mid c$$
$$z ::= x \mid v$$
$$S ::= x = v \mid c = \{\ \overrightarrow{v}\ \}^v \mid Function \mid p \mid \underline{p} \mid S, S \mid \nu\text{S} \mid \nu\text{S}$$

2.3 Regole della cham

Le regole sono di tre tipi: regole di semplice riscrittura, che possono essere applicate in entrambi i sensi; regole di ampliamento di membrane, particolari regole di riscrittura che regolano il rapporto tra la sottosoluzione contenuta in una membrana e la soluzione esterna; regole di computazione, applicabili in un unico verso, che esprimono come le istruzioni interagiscono con la soluzione. È bene notare che buona parte di queste regole hanno carattere locale, grazie alle caratteristiche della cham. Le regole sono da intendersi relative sia all'esecuzione del main sia all'esecuzione di altre goroutine, tranne dove specificato (ad esempio in **Goroutine call** e **Main goroutine call**).

2.3.1 Regole di riscrittura

$$p_1; (p_2; p_3) \leftrightarrows (p_1; p_2); p_3 \qquad \textbf{Sequential associativity}$$

introdotta per evitare ambiguità nell'uso dell'operatore sequenziale.

$$\nu c(S) \leftrightarrows \nu c'(S[c|c']) \qquad \textbf{Channel } \alpha\textbf{-conversion}$$

per consentire la rinomina di canali.

$$\nu x(S) \leftrightarrows \nu x'(S[x|x']) \qquad \textbf{Variable } \alpha\textbf{-conversion}$$

per consentire la rinomina di variabili.

2.3.2 Regole di membrana

In entrambi i casi servono a includere o portare fuori da una membrana parti di soluzione.
L'insieme dei nomi di una soluzione S $names(S)$ è l'insieme dei canali o variabili che compaiono in S senza essere ristretti dalla propria membrana.

$$\nu c(S), S' \leftrightarrows \nu c(S, S') \; se \; c \notin names(S') \qquad \textbf{Channel membrane}$$

$$\nu x(S), S' \leftrightarrows \nu x(S, S') \; se \; x \notin names(S') \qquad \textbf{Variable membrane}$$

2.3.3 Regole di computazione

Queste regole mostrano come la soluzione può continuare a reagire fino a terminazione.

$$\frac{S \to S' \quad \exists \underline{p}}{S \to S'} \qquad \textbf{Main execution}$$

un soluzione può continuare le proprie reazioni solo se il main non è terminato.

$$\frac{S \to S'}{\nu c(S) \to \nu c(S')} \qquad \textbf{Channel membrane execution}$$

$$\frac{S \to S'}{\nu x(S) \to \nu x(S')} \qquad \textbf{Variable membrane execution}$$

le membrane consentono reazioni al proprio interno.

$$S, 0 \rightarrow S \qquad \textbf{End}$$

Un programma vuoto termina scomparendo dalla soluzione.

La creazione di una membrana cattura le istruzioni successive in essa, consentendo così di utilizzare la variabile o il canale creato

$$\nu c[v]; p \rightarrow \nu c(c = \{\emptyset\}^v, p) \qquad \textbf{New channel}$$

la creazione di un canale introduce la membrana relativa e la lista di valori ad esso associati (inizializzata vuota).

$$\nu x[v]; p \rightarrow \nu x(x = v, p) \qquad \textbf{New variable}$$

la creazione di una variabile introduce la membrana relativa al cui interno si trova la molecola con il valore della variabile.

La possibilità che l'input o l'output su un canale sia bloccante è data dalla condizione necessaria indicata. **Channel input** si rifà a **Channel assignment** perchè anzichè "perdere" il valore ricevuto, lo mette in una nuova variabile che non sarà utilizzata: questo per semplificare ulteriormente la semantica proposta.

$c = \{\ \overrightarrow{v}\ \}^v\ , \bar{c}z; p \rightarrow c = \{\ \overrightarrow{v}, v_z\ \}^v, p \quad se|\overrightarrow{v}| < v$ **Channel output**

l'output su canale è bloccante se il canale ha raggiunto la dimensione massima.

$c = \{\ \overrightarrow{v}, w\ \}^v, c; p \rightarrow \nu \mathrm{x}[\ c = \{\overrightarrow{v}, w\}^v,\ x = 0,\ x := c;\ p]$ **Channel input**

$x = v, c = \{\ \overrightarrow{v}, w\ \}^v, x := c; p \rightarrow x = w, c = \{\ \overrightarrow{v}\ \}^v, \mathrm{p} \quad \text{se } |\overrightarrow{v}| > 0$ **Channel assignment**

Per eseguire una funzione, devono venire a contatto dichiarazione della funzione ed invocazione; per farlo generalmente si sfruttano le regole di membrana.

$$f\langle\overrightarrow{z}\rangle = p, f(\overrightarrow{v}); p' \rightarrow f\langle\overrightarrow{z}\rangle = p, p[\overrightarrow{z}|\overrightarrow{v}]; p' \qquad \textbf{Function call}$$

Per l'esecuzione di goroutine sono presenti due regole: una riguarda lo spawn da parte del main, l'altra da parte di altre goroutine, per garantire una corretta applicazione di **Main execution**.

$$f\langle\overrightarrow{z}\rangle = p, go\ f(\overrightarrow{v}); p' \rightarrow f\langle\overrightarrow{z}\rangle = p, p[\overrightarrow{z}|\overrightarrow{v}], p' \qquad \textbf{Goroutine call}$$

Una goroutine può essere creata in qualunque punto del programma.

$$f\langle\overrightarrow{z}\rangle = p, \underline{go f(\overrightarrow{v})}; p' \rightarrow f\langle\overrightarrow{z}\rangle = p, p[\overrightarrow{z}|\overrightarrow{v}], \underline{p}' \quad \textbf{Main goroutine call}$$

Il main può generare goroutine ma queste non saranno considerate come main.

Nel caso della select, si utilizza il costrutto che non tiene in memoria il valore ricevuto dal canale.

$c = \{\overrightarrow{v}, w\}^v\ \sum_{i=1}^{n} c_i; p_i + p' \rightarrow c = \{\overrightarrow{v}\}^v,\ p_j \quad$ dove $0 \leq j \leq n$ **Select ok**

$c_0 = \{\emptyset\}^v, \ldots, c_n = \{\emptyset\}^v, \sum_{i=1}^{n} c_i; p_i + p' \rightarrow c_0 = \{\emptyset\}^v, \ldots, c_n = \{\emptyset\}^v, p'$
dove $0 \leq j \leq n$ **Select fail**

Per non appesantire la semantica proposta, Il calcolo delle operazioni è affidato ad un oracolo O il quale riceve dei parametri in ingresso e restituisce il risultato.

$$\text{if } op(\overrightarrow{z}) \text{ then } \{p\} \text{ else } \{p'\} \rightarrow \begin{cases} p \text{ se } O(\overrightarrow{z}) \vdash op(\overrightarrow{z}) = T \\ p' \text{ se } O(\overrightarrow{z}) \vdash op(\overrightarrow{z}) = F \end{cases} \qquad \textbf{Ifelse}$$

$$x = v, x := w; p \rightarrow x = w, \mathrm{p} \qquad \textbf{Assignment from value}$$

$$x = v, x := op(\overrightarrow{z}); p \rightarrow x = w, \mathrm{p} \quad \textit{dove } O(\overrightarrow{z}) \vdash op(\overrightarrow{z}) = w$$

Assignment from operation

op è una operazione aritmetica o booleana.

2.4 Un esempio: Fibonacci parallelo

Nel seguente esempio, si calcola il valore dell' x-esimo numero della successione di fibonacci parallelizzando le chiamate ricorsive attraverso l'utilizzo di goroutine diverse. Ad una groutine viene passata la coppia di parametri numero da calcolare - canale su cui inviare l'output.

2.4.1 Fibonacci parallelo in Go

```
package main

import "fmt"

func main() {
        resultChan := make(chan int)
        x := 4
        go fibonacci(x, resultChan)
        result := <-resultChan
        fmt.Println(result)
}

func fibonacci(number int, channel chan int) {
        if number > 1 {
                closeFirst := make(chan int)
                go fibonacci(number-1, closeFirst)
                closeSecond := make(chan int)
                go fibonacci(number-2, closeSecond)
                f1, f2 := <-closeFirst, <-closeSecond
                channel <- (f1 + f2)
        } else if number == 1 {
                channel <- 1
        } else {
```

```
                channel <- 0
        }
}
```

2.4.2 Fibonacci parallelo in ChamGo

Per non appesantire la lettura, la dimensione del buffer dei canali è omessa; inoltre si introduce una nuova istruzione,$\overline{c}\ op(\overrightarrow{z})$ come abbreviazione di: $c = \{\overrightarrow{v}\ \}^v\ , \overline{c}\ op(\overrightarrow{z}); p \rightarrow c = \{\overrightarrow{v}, w\ \}^v, p \quad$ se $|\overrightarrow{v}| < v$ e $O(\overrightarrow{z} \vdash op(\overrightarrow{z}) = w)$
Si aggiunge anche l'istruzione *print*, equivalente di `fmt.Println`, che stampa un valore sullo standard output poi evapora.
L'esecuzione è indicata con frecce; alcuni passaggi banali o simili a alcuni già svolti sono omessi, così come alcuni elementi della soluzione che non sono interessanti nel passaggio.
La soluzione S che rappresenta il programma è formata da una funzione ($fibonacci$) e le istruzioni del main:

$S = \{$
$\quad fibonacci\langle number, channel\rangle = \{$
$\quad\quad if\ (number > 1)\ then$
$\quad\quad\quad \nu closeFirst[]; go\ fibonacci(number - 1, closeFirst);$
$\quad\quad\quad \nu closeSecond[]; go\ fibonacci(number - 2, closeSecond);$
$\quad\quad\quad \nu f1[0];\ \nu f2[0];\ f1 := closeFirst;\ f2 := closeSecond;$
$\quad\quad\quad \overline{channel}(f1 + f2)$
$\quad\quad else$
$\quad\quad\quad if\ (number == 1)\ then\ \overline{channel}\ 1$
$\quad\quad\quad else\ \overline{channel}\ 0$
$\quad \},$
$\quad \underline{\nu resultChan[]; \nu x[4]; go\ fibonacci(x, resultChan); \nu result[resultChan]; print(result)}$
$\}$

$\downarrow$

Applicazione di **New channel** e **New variable** fino alla chiamata della goroutine

$\downarrow$

$S = \{$
$fibonacci\langle number, channel\rangle, \nu resultChan(resultChan = \{\emptyset\}, \nu x(x = 4,$
$\underline{go\ fibonacci(x, resultChan); \nu result[resultChan]; print(result)}))\}$

$\downarrow$

Per potere eseguire $fibonacci$ si sposta la dichiarazione della funzione all'interno delle membrane attraverso le regole di membrana

$\downarrow$

$S = \{$
$\nu resultChan(resultChan = \{\emptyset\}, \nu x(x = 4, fibonacci\langle number, channel\rangle,$
$\underline{gofibonacci(x, resultChan); \nu result[resultChan]; print(result)}))\}$

$\downarrow$

La goroutine può iniziare ad eseguire secondo **Goroutine execution** e l'esecuzione di main proseguire

$\downarrow$

$S = \{$
$\nu resultChan(resultChan = \{\emptyset\}, \nu x(x = 4, fibonacci\langle number, channel\rangle,$
$fibonacci[x, channel|4, result], \nu result(result = 0, \underline{result := channel; print(result)}))\}$

$\downarrow$

Il main è fermo perchè attende un output su $channel$, si mostra l'esecuzione della goroutine di $fibonacci$

$\downarrow$

$S = \{$
$\ldots \nu closeFirst(closeFirst = \{\emptyset\}, gofibonacci(3, closeFirst); \nu closeSecond[];$
$go\ fibonacci(number - 2, closeSecond); \nu f1[0];\ \nu f2[0];\ f1 := closeFirst;$
$f2 := closeSecond; \overline{channel}(f1 + f2)) \ldots\}$

$\downarrow$

Per portare la dichiarazione di $fibonacci$ all'interno della membrana di $closeFirst$ evitando confusione sui canali è necessario usare **Channel α-conversion**

$\downarrow$

$S = \{$
$\ldots \nu closeFirst'(closeFirst' = \{\emptyset\}, fibonacci\langle number, channel\rangle,$
$gofibonacci(3, closeFirst'); \nu closeSecond(closeSecond = \{\emptyset\},$
$go\ fibonacci(number - 2, closeSecond); \nu f1[0];\ \nu f2[0];\ f1 := closeFirst;$
$f2 := closeSecond; \overline{channel}(f1 + f2))) \ldots\}$

$\downarrow$

Si procede in maniera analoga per le computazioni successive, si mostra solo un caso

$\downarrow$

$S = \{$
$\ldots \nu closeFirst'''(closeFirst' = \{1\}\nu closeSecond''(closeSecond'' = \{0\},$
$\nu f1'''(f1''' = 0, \nu f2''(f2'' = 0, f1''' := closeFirst'''; f2'' := closeSecond'';$
$\overline{closeFirst''}(f1''' + f2''))))) \ldots\}$

$\downarrow$

Per applicare **Channel input** si spostano $closeFirst''' = \{\emptyset\}$ e $f1''' = 0$

$\downarrow$

$S = \{$
$\ldots \nu closeFirst'''(\nu closeSecond''(closeSecond'' = \{0\}, \nu f1'''(\nu f2''($
$f2'' = 0, f1''' = 0, closeFirst' = \{1\}, f1''' := closeFirst'''; f2'' := closeSecond'';$
$\overline{closeFirst''}(f1''' + f2''))))) \ldots\}$

$\downarrow$

Si applica **Channel input** ed analogamente si procede per $closeSecond'' = \{\emptyset\}$ e $f2'' = 0$

$\downarrow$

$S = \{$
$\ldots \nu closeFirst'''(\nu closeSecond''(\nu f1'''(\nu f2''(closeFirst' = \{\emptyset\},$
$closeSecond'' = \{\emptyset\}, f1''' = 1, f2' = 0, ; \overline{closeFirst''}(f1''' + f2''))))) \ldots\}$

$\downarrow$

Si sposta la molecola $closeFirst'' = \{\emptyset\}$ convenientemente e si applica

Channel output

$\downarrow$

$S = \{$
$\ldots \nu closeFirst'''(\nu closeSecond''(\nu f1'''(\nu f2''(closeFirst''' = \{\emptyset\},$
$closeSecond'' = \{\emptyset\}, f1''' = 1, f2' = 0, closeFirst'' = \{1\})))) \ldots\}$

$\downarrow$

Le reazioni successive sono analoghe a quanto mostrato. Dopo le chiamate ricorsive, si ha:

$\downarrow$

$S = \{$
$\nu resultChan(resultChan = \{3\}, \ldots, \nu result(result = 0, \underline{result := resultChan; print(result)})$
$\ldots)\}$

$\downarrow$

Proseguendo con le reazioni:

$\downarrow$

$S = \{\ \nu resultChan(\ldots, \nu result(resultChan = \{\emptyset\}, result = 3, \underline{print(result)}) \ldots)\}$

$\downarrow$

Applicando print

$\downarrow$

$S = \{\ \nu resultChan(\ldots, \nu result(resultChan = \{\emptyset\}, result = 3) \ldots)\}$

```
Output:  3
```

Non possono eseguire ulteriori reazioni per via della regola **Main execution**, quindi il programma è terminato.

Capitolo 3

Lavoro futuro e conclusioni

Per quanto Turing-completo e funzionante, ChamGo è ancora piuttosto limitato. La trattazione della concorrenza si rifà a solamente uno dei due modelli presenti nel linguaggio e non include strutture che li combinano, ad esempio `sync.WaitGroup`. Inoltre, non sono trattati due costrutti del linguaggio di grande utilità nell'uso dei canali: `close`, che permette di dichiarare chiuso un canale per evitare che sia utilizzato in futuro, e `for - range`, che permette di eseguire operazioni fino a quando il canale di input non viene chiuso. Il `close`, può agire in maniera "globale", cioè su più molecole, quindi la sua inclusione all'interno di ChamGo avrebbe portato a pesanti modifiche nello stile della semantica. Il `for - range` non è stato incluso a causa della sua forte dipendenza da `close`.
Non è stato trattata la gestione FIFO degli input da canale. Per includerla, si potrebbe utilizzare una soluzione simile a quella usata per distinguere il main dalle altre goroutine: segnando ogni goroutine in una maniera univoca (ad esempio, assegnando un nuovo colore ad ogni spawn), si potrebbe introdurre un oracolo che tenga traccia dei colori che fanno richiesta di input e ne gestisca l'accesso al canale. Altre aggiunte alla semantica proposta possono essere i costrutti `for` e `while`, la costruzione di oggetti, il passaggio di oggetti per valore o per riferimento (aggiunta banale nel caso di variabili o canali) e le funzioni come cittadini di prima classe, introducendone scope e closure.

Si è provato a creare un eseguibile partendo dalla semantica proposta provando vari approcci: utilizzando tool basati su *matching logic* come K [11], provando a creare da zero un linguaggio utilizzando Racket con la libreria Redex [12], cercando un mapping tra ChamGo e linguaggi che implementano il π-calcolo (Pict [13]). In tutti e tre i casi, il compito si è rivelato molto più arduo del previsto, o per la difficoltà dei tool o per l'assenza di supporto dei linguaggi.
L'uso di Chemical Abstract Machine può essere una delle chiavi per ampliare la comprensione della concorrenza ed una efficace arma per l'implementazione. Un esempio in questo senso è Chemlambda [14].

Bibliografia

[1] The Rust Team, "The Rust Programming Language." `https://www.rust-lang.org/`.

[2] The Jolie Team, "Jolie Programming Language - Official Website." `https://www.jolie-lang.org/`.

[3] R. Pike, K. Thompson, and R. Griesemer, "The Go Programming Language." `https://golang.org/`.

[4] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, 1985.

[5] "The go programming language specification." `https://golang.org/ref/spec`.

[6] R. Milner, *Communicating and mobile systems: the pi-calculus*. Cambridge Univ. Press, 1999.

[7] G. Berry and G. Boudol, "The chemical abstract machine," *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 90*, 1990.

[8] "The go programming language frequently asked questions." `https://golang.org/doc/faq`.

[9] R. Pike, "Go concurrency patterns." Google I/O, Jun 2012.

[10] R. Milner, *A Calculus of Communicating Systems*. Springer Berlin Heidelberg, 1980.

[11] G. Rosu, "K Framework." `http://www.kframework.org/`.

[12] R. B. Findler, C. Klein, B. Fetscher, and M. Felleisen, "Redex: Practical Semantics Engineering." `https://docs.racket-lang.org/redex/`.

[13] B. C. Pierce and D. N. Turner, " The Pict Programming Language." `http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html`.

[14] "Chemlambda - Chemical Concrete Machine." `https://chorasimilarity.wordpress.com/chemical-concrete-machine/`, Mar 2018.

Ringraziamenti

Ringrazio gli amici, in particolare la balotta dello Stop, che mi hanno sopportato mentre deliravo di semantica, canali, molecole.

Ringrazio Giulia, che più di tutti ha retto i miei deliri.

Ringrazio la mia famiglia, che mi ha consentito i detti deliri.

Ringrazio il prof. Sangiorgi e il dott. Giallorenzo, che mi hanno trasmesso la passione per la *computer science*.