Astrazioni

- Sebbene le definizioni ricorsive possono essere espresse attraverso la replicazione, a volte può fare comodo utilizzarle per descrivere i nostri esempi (un ragionamento analogo vale per la poliadicità).

- Perciò se aggiungessimo al π-calcolo le definizioni ricorsive (che possiamo vedere come comode macro) avremmo tre modi differenti per legare nomi:

  – In $x(y).P$ il nome y è legato dall'input, ed il suo scope è P

  – In $(\nu y)P$ il nome y è legato dalla restrizione, ed il suo scope è P

  – In $A(\vec{y}) \stackrel{\text{def}}{=} Q_A$ i nomi $\vec{y}$ sono legati dalla definizione, ed il loro scope è Q_A.

- È possibile comunque ricondurre queste tre forme differenti di 'binding' ad una sola

- Scriveremo allora $(x).P$ per denotare una **astrazione** di **arietà** 1. Più in generale

$$F \stackrel{\text{def}}{=} (x_1).(x_2).\ldots.(x_n).P$$

è un'astrazione di arietà n, e scriveremo $F{:}n$. Nota che un qualsiasi processo P è un'astrazione di arietà 0.

- Definiamo $((x).F)\langle y\rangle \stackrel{\text{def}}{=} \{y/x\}F$.

- Se $F{:}n$ allora $(x).F : n+1$ e viceversa se $F : n+1$ allora $F\langle x\rangle{:}n$.

- Usiamo le seguenti abbreviazioni:

$$\begin{array}{rcll}
(x_1,\ldots,x_2).P & \stackrel{\text{def}}{=} & (x_1).(x_2)\ldots(x_n).P & \quad \text{per } x_1,\ldots,x_n \text{ distinti} \\
F\langle y_1,\ldots,y_n\rangle & \stackrel{\text{def}}{=} & (\ldots(F\langle y_1\rangle)\langle y_2\rangle\ldots)\langle y_n\rangle &
\end{array}$$

- Tornando ai nostri tre costrutti che legano nomi. Possiamo allora riscriverli usando le astrazioni:

$$\begin{array}{rcl} x(\vec{y}).P & \text{come} & x((\vec{y}).P) \\ (\nu x)P & \text{come} & (\nu)((x).P) \\ A(\vec{x}) \stackrel{\text{def}}{=} Q_A & \text{come} & A \stackrel{\text{def}}{=} (\vec{x}).Q_A \end{array}$$

- **Esempio 89** *Abbiamo già visto come definire l'***operatore di link** *tra due processi P e Q rispetto ad una porta sinistra l ed una destra r:*

  $$P \frown Q \stackrel{def}{=} (\nu m)(\{^m/_r\}P \mid \{^m/_l\}Q)$$

  Questa definizione **funziona per specifici** *l ed r che bisogna conoscere a priori. Generalizzando è possibile dare la medesima definizione su processi parametrici su due porte, cioè su astrazioni di arietà 2. Così se F:2 e G:2 allora*

  $$F \frown G \stackrel{def}{=} (l, r).(\nu m)(F\langle l, m\rangle \mid G\langle m, r\rangle)$$

  Applicando l'operatore di link tra astrazioni di processi buffer B e C visti precedentemente, e definiti come:

$$
\begin{aligned}
B &\stackrel{def}{=} (l,r).l(x).C\langle x,l,r\rangle \\
C &\stackrel{def}{=} (x,l,r).\overline{r}\langle x\rangle.B\langle l,r\rangle
\end{aligned}
$$

poichè $B{:}2$ *e* $C{:}3$*, la catena*

$$J = C\langle x_1\rangle \frown B \frown C\langle x_2\rangle$$

è un'astrazione di arietà 2*, parametrica nella porte di input ed output. Mentre* $J\langle in_1, out_4\rangle$ *è un processo in cui le porte di input e di output sono state istanziate.*

Strutture dati in π-calcolo

- Negli esempi precedenti abbiamo spesso usato valori all'interno di un dominio finito $V = \{v_1, \ldots, v_n\}$ come nel caso di

$$Buff^{(n)}_{v_1,\ldots,v_k}$$

- In pratica abbiamo sempre assunto che il nostro linguaggio fosse esteso con tali valori. Questo perchè nella definizione del π-calcolo gli unici valori che si possono trasmettere sono **nomi** e non booleani, interi, ect.

- Comunque, la trasmissione di nomi è sufficientemente espressiva per codificare vari tipi di dato in termini di processi.

- Iniziamo con tipi di dato cosidetti **effimeri** che possono essere usati una sola volta. Essi corrispondono ad una **istanza** di un valore. Per esempio nell'espressione

$$12 \times (x + 12)$$

abbiamo due copie del valore 12. Una volta a conoscenza del valore di x è possibile calcolare il risultato dell'espressione **consumando** le due occorrenze dell'intero 12. Si noti che il valore associato ad x è **persistente** in quanto riusabile. Sarà evidente in seguito che il modo più semplice per rendere persistente un valore è quello di definirlo all'interno di una replicazione.

- Iniziamo coi **booleani**. **Staticamente**, l'elemento $true$ non è altro che un elemento dell'insieme $\{true, false\}$. **Dinamicamente**, è uno **switch** che ci dice in quale modo procedere. In π-calcolo i booleani sono rappresentati attraverso astrazioni:

$$True \stackrel{\text{def}}{=} (l).l(t,f).\bar{t} \qquad\qquad False \stackrel{\text{def}}{=} (l).l(t,f).\overline{f}$$

Scrivendo $True\langle l\rangle$ e $False\langle l\rangle$ denotiamo i valori vero e falso **allocati** in l. Ma allora come possiamo utilizzare questi valori, ad esempio, in combinazione con una condizionale?

$$\begin{array}{rcl} True\langle l\rangle \;\mid\; \bar{l}\langle t', f'\rangle.(t'.P + f'.Q) & \xrightarrow{\tau}^{*} & P \\ False\langle l\rangle \;\mid\; \bar{l}\langle t', f'\rangle.(t'.P + f'.Q) & \xrightarrow{\tau}^{*} & Q \end{array}$$

Notate come, per potere interagire, sia i valori che il condizionale devono essere **allocati** sullo stesso canale l.

- Si noti anche che se si suppone di avere più condizionali in parallelo con differenti valori potremmo avere interferenze. **Tali interferenze possono essere evitate usando canali privati**:

$$Cond(P,Q) \stackrel{\text{def}}{=} (l).(\nu t, f)\bar{l}\langle t, f\rangle.(t.P + f.Q)$$

E quindi adesso

$$\begin{array}{rcl} True\langle l\rangle \mid Cond(P,Q)\langle l\rangle & \xrightarrow{\tau}^{*} & P \\ False\langle l\rangle \mid Cond(P,Q)\langle l\rangle & \xrightarrow{\tau}^{*} & Q \end{array}$$

- Anzi, più precisamente vale:

$$\begin{array}{rcl} True\langle l\rangle \mid Cond(P,Q)\langle l\rangle & \xrightarrow{\tau}\approx & P \\ False\langle l\rangle \mid Cond(P,Q)\langle l\rangle & \xrightarrow{\tau}\approx & Q \end{array}$$

- Notate che l'operatore $Cond(P, Q)(l)$, così come l'operatore di link $\frown$, è un operatore derivato. Tali operatori fanno intuire come sia possibile costruire un vero e proprio **linguaggio di programmazione concorrente** che abbia come nucleo il π-calcolo (Es: JOIN (Fournet e Gonthier 1996); PICT (Pierce and Turner 1997), ect.)

- Come rappresentiamo il tipo enumerato

  $\{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday\}$?

  anche qui con un'astrazione contenente un menù con sette componenti:

  $$Monday \stackrel{\text{def}}{=} (l).l(mo, tu, we, th, fr, sa, su).\overline{mo}$$

  In questo modo possiamo rappresentare ogni valore con una astrazione unaria. Questo significa che possiamo definire strutture di dato, ad esempio liste, in maniera del tutto indipendente da ciò che la struttura andrà a contenere (booleani, interi, ect..).

- Quindi più in generale com'è definito un tipo di dato in π-calcolo? Li defininiremo in termini di **costanti** e **costruttori**. Ad esempio,

– Un valore booleano o è $True$ oppure $False$;
– Un valore intero o è $Zero$ oppure $Succ$ di un intero;
– Una lista o è Nil oppure un $Cons$ di un valore con una lista;
– Un albero binario.....

Liste in π-calcolo

- Vediamo adesso come sia possibile rappresentare liste di valori in π-calcolo.
- Le nostre definizioni sono **polimorfe**; ovvero, sono indipendenti dai valori contenuti nelle liste.
- Definiamo i seguenti costruttori Nil e $Node$.

$$Nil \stackrel{\text{def}}{=} (k).k(n,c).\overline{n} \qquad\qquad Node \stackrel{\text{def}}{=} (k,v,l).k(n,c).\overline{c}\langle v,l\rangle$$

- Dove $Nil\langle k\rangle$ rappresenta una istanza della lista vuota; mentre il processo $Node\langle k, v, l\rangle$ è un connettore per costruire liste (vedi figura pag. 105).

- Entrambi i costrutti sono degli switch che una volta interrogati (attraverso un output su k) ritornano:

  – nel primo caso, un ouput su $\overline{n}$

  – nel secondo caso, un output $\overline{c}\langle v, l\rangle$ dove v è il link al valore in quella posizione, mentre l è il link alla lista rimanente.

- Sia V un certo tipo di dato, già rappresentato, e sia L una lista nella rappresentazione che ci apprestiamo a dare, allora

  – la costante Nil

  – il costrutto $Cons(V, L)$

  – ed una lista di n valori $[V_1, \ldots, V_n]$

vengono così rappresentate:

$$\begin{array}{rcl} Nil & \stackrel{\text{def}}{=} & (k).k(n,c).\overline{n} \\ Node & \stackrel{\text{def}}{=} & (k,v,l).k(n,c).\overline{c}\langle v,l\rangle \\ Cons(V,L) & \stackrel{\text{def}}{=} & (k).(\nu vl)(Node\langle k,v,l\rangle \mid V\langle v\rangle \mid L\langle l\rangle) \\ [V_1,\ldots,V_n] & \stackrel{\text{def}}{=} & Cons(V_1,Cons(\ldots Cons(V_n,Nil)\ldots)) \end{array}$$

- Vediamo ad esempio come si rappresenta in π-calcolo la lista $[True, False]\langle k\rangle$:

$$\begin{array}{rcl} [True,False]\langle k\rangle & \stackrel{\text{def}}{=} & Cons(True,Cons(False,Nil))\langle k\rangle \\ & \stackrel{\text{def}}{=} & (\nu vl)(Node\langle k,v,l\rangle \mid True\langle v\rangle \mid Cons(False,Nil)\langle l\rangle) \\ & \stackrel{\text{def}}{=} & (\nu vl)(Node\langle k,v,l\rangle \mid True\langle v\rangle \\ & & \qquad \mid (\nu v'l')(Node\langle l,v',l'\rangle) \mid False\langle v'\rangle \mid Nil\langle l'\rangle)) \\ & \stackrel{\text{def}}{=} & (\nu vl)(k(n,c).\overline{c}\langle v,l\rangle \mid v(t,f).\overline{t} \\ & & \qquad \mid (\nu v'l')(l(n',c').\overline{c'}\langle v',l'\rangle \mid v'(t,f).\overline{f} \mid l'(n',c').\overline{n'})) \end{array}$$

- Adesso vediamo alcuni esempi di **programmazione funzionale** con liste.

- Cominciamo con un costrutto simile al $Cond(P, Q)$ visto per i booelani.

$$Listcase(P, F) \stackrel{\text{def}}{=} (k).(\nu nc)\overline{k}\langle n, c\rangle.(n.P + c.F)$$

- Tale operatore ritorna il processo P in caso di lista vuota, altrimenti ritorna un'astrazione F:2 che verrà istanziata con i puntatori al valore di testa ed alla restante lista.

Proposizione 90

$$\begin{array}{lll} Listcase(P, F)\langle k\rangle \ \mid \ Nil\langle k\rangle & \xrightarrow{\tau}^{\star} & P \\ Listcase(P, F)\langle k\rangle \ \mid \ Cons(V, L)\langle k\rangle & \xrightarrow{\tau}^{\star} & (\nu vl)(F\langle v, l\rangle \mid V\langle v\rangle \mid L\langle l\rangle) \end{array}$$

Prova Da fare a casa.

- Per comodità possiamo riscrivere la *Listcase* in una notazione di più alto livello. Se supponiamo $F = (v, l).Q$ allora:

$$\text{case } k \text{ of} \quad \begin{array}{lcl} Nil? & \Longrightarrow & P \\ Cons?(v, l) & \Longrightarrow & Q \end{array}$$

- Vediamo adesso un costrutto che copia una lista da una locazione (cioè canale) l ad un'altra locazione m:

$$\begin{array}{rcl} Copy(l, m) & \stackrel{\text{def}}{=} & \text{case } l \text{ of} \\ & & \begin{array}{lcl} Nil? & \Longrightarrow & Nil\langle m \rangle \\ Cons?(v, l') & \Longrightarrow & (\nu m')(Node\langle m, v, m' \rangle \mid Copy\langle l', m' \rangle) \end{array} \end{array}$$

- La $Copy\langle l, m \rangle$ non contiene la lista da spostare. In un certo qual modo è simile ad una funzione, infatti, quando posta in parallelo con un lista L allocata in l, ritorna una copia di L allocata in m.

Proposizione 91 $L\langle l \rangle \mid Copy\langle l, m \rangle \xrightarrow{\tau}^{\star} L\langle m \rangle$.

- La prova è per induzione sulla struttura della lista L.
- Si noti che la lista originale, allocata in l, viene distrutta (infatti le nostre liste sono tipi di dato **effimeri**).
- Vediamo adesso un altro operatore $Join(k, l, m)$ che date due liste allocate in k ed l, rispettivamente, concatena le due liste e alloca la lista risultante in m.
- Tale operatore scorre la prima lista, copiando un elemento alla volta, e poi aggancia la seconda lista in un colpo solo.

$$
\begin{array}{lcll}
Join(k, l, m) & \stackrel{\text{def}}{=} & \text{case } k \text{ of} & \\
 & & Nil? & \Longrightarrow \quad Copy\langle l, m\rangle \\
 & & Cons?(v, k') & \Longrightarrow \quad (\nu m')(Node\langle m, v, m'\rangle \mid Join\langle k', l, m'\rangle)
\end{array}
$$

- Per formulare la correttezza dell'operatore di $Join$ defininiamo la concatenazione di due liste $K = [U_1, \ldots, U_r]$ e $L = [V_1, \ldots, V_s]$ come
$$K \oplus L \stackrel{\text{def}}{=} [U_1, \ldots, U_r, V_1, \ldots, V_s]$$

L'operatore $\oplus$ soddisfa delle poprietà ovvie:

$$\begin{array}{rcl} Nil \oplus L & = & L \\ Cons(V, K \oplus L) & = & Cons(V, K) \oplus L \end{array}$$

- Possiamo allora provare il seguente risultato:

  Proposizione 92 $K\langle k\rangle \mid L\langle l\rangle \mid Join\langle k, l, m\rangle \xrightarrow{\tau}^{\star} (K \oplus L)\langle m\rangle$

  Prova La prova è per induzione sulla struttura di K.

- **Importante:** Sebbene *Join* e *Copy* sembrino essere funzioni (da liste in liste), in realtà si comportano in maniera leggermente differente. Si consideri ad esempio il sistema:

$$S = Join\langle k, l, m\rangle \mid K_1\langle k\rangle \mid K_2\langle k\rangle \mid L\langle l\rangle$$

  allora vi sono due possibili reazioni:

$$\begin{array}{rcl} S & \xrightarrow{\tau}^{\star} & K_1\langle k\rangle \mid (K_2 \oplus L)\langle m\rangle \\ S & \xrightarrow{\tau}^{\star} & K_2\langle k\rangle \mid (K_1 \oplus L)\langle m\rangle \end{array}$$

- Questo perchè l'operatore di $Join$ può scegliere tra due liste differenti entrambe allocate in k.
- Per rendere tali operatori delle vere e proprie funzioni è sufficiente farli operare su astrazioni di liste, cioè liste non ancora allocate in un qualche canale.
- In tal modo gli operatori possono accedere in maniera privata alle liste a cui vengono applicate.

$$\begin{aligned} Move(L) &\stackrel{\text{def}}{=} (m).(\nu l)(L\langle l\rangle \mid Copy\langle l, m\rangle) \\ Append(K, L) &\stackrel{\text{def}}{=} (m).(\nu kl)(K\langle k\rangle \mid L\langle l\rangle \mid Join\langle k, l, m\rangle) \end{aligned}$$

Adesso possiamo dire veramente che facciamo programmazione funzionale su liste. In seguito proveremo i seguenti risultati:

Teorema 93

$$\begin{aligned} Move(L) &\approx L \\ Append(K, L) &\approx K \oplus L. \end{aligned}$$

Tipi di dato persistenti

- Nelle definizioni date fino adesso i tipi di dato sono **effimeri**, cioè una volta usati vengono consumati: Ad esempio in:

$$True\langle l\rangle \mid Cond(P,Q)\langle l\rangle \quad \xrightarrow{\tau}^{*} \quad P$$

- Il valore $True\langle l\rangle$, una volta testato, viene consumato. E se volessimo invece riutilizzare tale valore per successivi testing?

- In maniera analoga quando operiamo su liste le distruggiamo creandone di nuove.

- Mostriamo adesso come definire dei booelani e delle liste **persistenti**. Per comodità useremo gli stessi nomi preceduti da un asterisco.

- I booleani diventano allora:

$$\star True \stackrel{\text{def}}{=} (l).!l(t,f).\bar{t} \qquad\qquad \star False \stackrel{\text{def}}{=} (l).!l(t,f).\overline{f}$$

- Mentre per quanto riguarda le liste è sufficiente replicare i connettori basilari:

$$\star Nil \stackrel{\text{def}}{=} (k).!k(n,c).\overline{n} \qquad\qquad \star Node \stackrel{\text{def}}{=} (k,v,l).!k(n,c).\overline{c}\langle v,l\rangle$$

- La definizione completa di liste persistenti è

$$\begin{array}{rcl}
\star Nil & \stackrel{\text{def}}{=} & (k).!k(n,c).\overline{n} \\
\star Node & \stackrel{\text{def}}{=} & (k,v,l).!k(n,c).\overline{c}\langle v,l\rangle \\
\star Cons(V,L) & \stackrel{\text{def}}{=} & (k).(\nu vl)(\star Node\langle k,v,l\rangle \mid V\langle v\rangle \mid L\langle l\rangle) \\
\star[V_1,\ldots,V_n] & \stackrel{\text{def}}{=} & \star Cons(V_1,\star Cons(\ldots\star Cons(V_n,\star Nil)\ldots))
\end{array}$$

- Le definizioni di $\star Copy$, $\star Join$, $\star Move$, e $\star Append$, sono simili alle definizioni prive di asterisco in cui $Node$ è rimpiazzato da $\star Node$.

- Adesso che abbiamo liste persistenti ha senso chiedersi se gli elementi di una lista possano essere delle **celle variabili**, cioè delle celle il cui contenuto può cambiare nel tempo.

- Ovviamente, avrebbe poco senso definire una lista effimera di variabili, poichè in tal caso il valore contenuto in una variabile potrebbe essere acceduto una sola volta.

- Nel seguito vi verrà mostrato come sia possibile implementare una lista di variabili.

- Una cella variabile in π-calcolo ha le seguenti proprietà:

  1. O è vuota oppure contiene un valore;
  2. se ne può testare lo stato ed eventualmente leggere;
  3. può essere svuotata o le si può assegnare un valore.

  Indicheremo con $Nullref$ una cella vuota e con Ref una cella contenente un valore.

$$
\begin{aligned}
Nullref(r) &\stackrel{\text{def}}{=} r(n,c).(\overline{n}.Nullref\langle r\rangle + c(v').Ref\langle r,v'\rangle + n.Nullref\langle r\rangle)\\
Ref(r,v) &\stackrel{\text{def}}{=} r(n,c).(\overline{c}\langle v\rangle.Ref\langle r,v\rangle + c(v').Ref\langle r,v'\rangle + n.Nullref\langle r\rangle)
\end{aligned}
$$

Spieghiamo cosa avviene.

– Una $Nullref$ può essere:
  * *testata* attraverso $\overline{n}$ rimanendo una $Nullref$
  * *riempita* attraverso c con un valore v' evolvendo in una Ref
  * *svuotata* attraverso n rimanendo sempre $Nullref$.
– Una Ref può essere:
  * *letta* attraverso $\overline{c}$ rimanendo sempre Ref
  * *modificata* con un nuovo valore attraverso c ed in tal caso evolve in una nuova Ref
  * *svuotata*, attraverso n ed evolve in $Nullref$.

- Così una cella variabile contenente un valore V (ad esempio un booleano) può essere così rappresentato:

$$Store(V) \stackrel{\text{def}}{=} (r).(\nu v)(Ref\langle r, v\rangle \mid V\langle v\rangle)$$

- Ovviamente, essendo *Store* una astrazione unaria possiamo usarla come elemento di una lista.
- Creando cioè una lista di puntatori di memoria come evidenziato in figura a pag. 111.
- La struttura di una lista non può essere cambiata a tempo di esecuzione, ma gli elementi della lista, essendo variabili, possono puntare ad oggetti differenti nell'arco del tempo.
- Quali sono le operazioni possibili su celle di memoria?

$$
\begin{array}{rcl}
Refcases(P,F) & \stackrel{\text{def}}{=} & (r).(\nu nc)\overline{r}\langle n,c\rangle.(n.P + c.F) \\
Nullify(P) & \stackrel{\text{def}}{=} & (r).(\nu nc)\overline{r}\langle n,c\rangle.\overline{n}.P \\
Assign(P) & \stackrel{\text{def}}{=} & (r,v).(\nu nc)\overline{r}\langle n,c\rangle.\overline{c}\langle v\rangle.P
\end{array}
$$

- $Refcases$ analizza il contenuto di una cella di memoria
- $Nullify$ svuota il contenuto di una cella di memoria
- $Assign$ assegna un nuovo valore alla cella di memoria.

• Verificare allora le seguenti riduzioni

$$
\begin{array}{rcl}
Refcases(P,F)\langle r\rangle \mid Nullref\langle r\rangle & \stackrel{\tau}{\longrightarrow}^{\star} & P \mid Nullref\langle v\rangle \\
Refcases(P,F)\langle r\rangle \mid Ref\langle r,v\rangle & \stackrel{\tau}{\longrightarrow}^{\star} & F\langle v\rangle \mid Ref\langle r,v\rangle \\
Nullify(P)\langle r\rangle \mid Ref\langle r,v\rangle & \stackrel{\tau}{\longrightarrow}^{\star} & P \mid Nullref\langle r\rangle \\
Assign(P)\langle r,v'\rangle \mid Ref\langle r,v\rangle & \stackrel{\tau}{\longrightarrow}^{\star} & P \mid Ref\langle r,v'\rangle
\end{array}
$$

• Fare Esercizio 10.25.

- Concludendo, abbiamo visto come implementare vari tipi di dato in π-calcolo. Nel caso delle liste tali strutture, in generale, possono avere come elementi dei processi. Ciò vuol dire, ad esempio, che potremmo allocare i nostri processi in una struttura ad albero, per rappresentare, possibilmente, la nozione di locazione o sito.