Semantica e Concorrenza
Modulo di Semantica

Pietro Di Gianantonio

corso di laurea in informatica

Modulo di 6 crediti (48 ore),

Semantica dei linguaggi di programmazione
Definire in maniera formale e rigorosa il comportamento dei programmi.

- Vedremo 3 diversi approcci, metodi, alla semantica.
- Metodi applicati ad una serie linguaggi di programmazione via via più complessi.

Alcuni argomenti si sovrappongo al corso di Metodi Formali.

Propedeutico a:

- Modulo sulla Concorrenza (Lenisa),
- Interpretazione Astratta (Comini),
- una qualsiasi trattazione formale e rigorosa dei programmi.

Semantica dei linguaggi di programmazione

Obiettivo: definire in maniera formale il comportamento dei programmi, e dei costrutti di programmazione.
In contrapposizione alle definizioni informali, comunemente usate.

Utile per:

- evitare ambiguità nella definizione di un linguaggio di programmazione:
  - si mettono in evidenza i punti critici di un linguaggio di programmazione (es. ambiente statico – dinamico, valutazione e passaggio dei parametri)
  - utile nella costruzione di compilatori,
- ragionare sui singolo programmi (definire logiche, principi di ragionamento, sui programmi):
  provare che un certo programma soddisfa una certa proprietà, una specifica, che è corretto.

Approcci alla semantica

- Semantica Operazionale. Descrivere come avviene l'esecuzione del programma,
  non su un vero calcalalatore, troppo complesso,
  su una semplice macchina formale (nello stile della macchine di Turing).
- Semantica Operazionale Strutturata (SOS). La macchina formale costituita da un sistema di riscrittura: sistema di regole, sintattiche, di riscrittura.

Approcci alla semantica

- Semantica Denotazionale. Il significato di un programma descritto da un oggetto matematico.
  Funzione parziale, elemento in un insieme ordinato.
  Un elemento di un insieme ordinato rappresenta il significato di un programma, di una parte del programma, di un costrutto della programmazione.

  Alternative, action semantics, semantica a giochi, categoriale.
- Semantica Assiomatica. Il significato di un programma espresso in termini di pre-condizioni e post-condizioni.

Tante semantiche perché nessuna completamente soddisfacente.
Ognuna descrive un aspetto del comportamento dei programmi, ha un diverso obiettivo.

Semplice, sintattica, intuitiva.

Piuttosto flessibile.

- Può facilmente gestire linguaggi di programmazione complessi.
- La struttura delle regole resta costante nei diversi linguaggi.

SOS contra

La semantica dipende dalla sintassi, è formulata usando la sintassi.

Difficile correlare programmi scritti in linguaggi differenti.

La semantica non è composizionale (la semantica di un elemento dipende dalla semantica dei suoi componenti).

Induce una nozione di equivalenza tra programmi difficile da verificare (ed utilizzare).

Obiettivi:

- una semantica indipendente dalla sintassi, si possono confrontare programmi scritti in linguaggi differenti.
- una semantica composizionale (la semantica di un elemento dipende dalla semantica dei suoi componenti).
- più astratta, fornisce strumenti per il ragionamento sui programmi.

Caratteristica principale: descrivere il comportamento di un programma attraverso un oggetto matematico.

Diverse caratteristiche dei linguaggi di programmazione

- non terminazione,
- store (memoria, linguaggi imperativi)
- environment, (ambiente)
- non determinismo,
- concorrenza,
- funzioni di ordine superiore (linguaggi funzionali),
- eccezioni,
- ...

Al crescere della ricchezza del linguaggi cresce (rapidamente) la complessità della semantica denotazionale.

Descrizione indiretta di un programma,
mediante un insieme di asserzioni:

$$\{Pre\}\ p\ \{Post\}$$

Fornisce immediatamente una logica per ragionare sui programmi.

Complementare, e giustificata, dalle altre semantiche.

Libro di testo

Glynn Winskel:
The Formal Semantics of Programming Languages. An introduction.

Un classico, presentazione semplice, completa, con pochi fronzoli, e poche considerazioni generali.

Tre copie in biblioteca.

Presentiamo buona parte del libro, saltando le dimostrazioni. Argomenti aggiuntivi.

Propedeuticità

Alcuni argomenti della logica:

- calcolo dei predicati,
- costruzione insiemistiche: prodotto, somma disgiunta, spazio di funzione, insiemi delle parti,
- grammatiche (libere dal contesto),
- definizione induttive e principio di induzione,
- dualità: linguaggio e modello, (sintassi e semantica).

Un semplice linguaggio imperativo: IMP

Categorie sintattiche:

Numeri interi (**N**): n, valori booleani (**T**), locazioni (**Loc**): X,

Espressioni aritmetiche (**AExp**):

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

Espressioni booleane (**BExp**):

$$b ::= \textbf{true} \mid \textbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \textbf{not}\ b \mid b_0\ \textbf{or}\ b_1 \mid b_0\ \textbf{and}\ b_1$$

Comandi (**Com**):

$$c ::= \textbf{skip} \mid X := a \mid c_0; c_1 \mid \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1 \mid \textbf{while}\ b\ \textbf{do}\ c$$

Sintassi astratta, più semplice della sintassi concreta

Sintassi incompleta: non definiti la sintassi per le costanti intere, le locazioni.
Poco interessanti, ortogonali alla nostra trattazione.

Linguaggio minimale capace di calcolare tutte le funzioni computabili (Turing-completo), se le locazioni possono memorizzare interi arbitrariamente grandi.

Manca:

- le variabili (l'ambiente, (environment)),
- definizioni di procedure, funzioni,
- definizioni ricorsive.

Semantica operazionale per IMP

Un insieme di regole per descrivere il comportamento di espressive aritmetiche, booleane, comandi.

All'espressioni aritmetiche si associano asserzioni, giudizi (judgments)

$$\langle a, \sigma \rangle \ \Rightarrow \ n$$

dove $\sigma : \mathbf{Loc} \to \mathbf{N}$, stato (memoria, store).

Giudizi derivati attraverso regole in deduzione naturale,
Regole guidate dalla sintassi (stuctured operational semantics).

Alle espressioni base si associano assiomi:

$$\frac{}{\langle n, \sigma \rangle \ \Rightarrow \ n}$$

$$\frac{}{\langle X, \sigma \rangle \ \Rightarrow \ \sigma(X)}$$

IMP: regole SOS

Alle espressioni composte regole di derivazione:

$$\frac{\langle a_0, \sigma\rangle \Rightarrow n_0 \qquad \langle a_1, \sigma\rangle \Rightarrow n_1}{\langle a_0 + a_1, \sigma\rangle \Rightarrow n} \qquad n_0 + n_1 = n$$

$$\ldots$$

- Valutare un espressione aritmetica è banale, quindi: regole banali.
- Ad ogni espressione associata una, o più regole determinate dal suo connettivo principale.
- Regole sottintendono un algoritmo di valutazione (deterministico).
- Regole assumono che un preesistente meccanismo di rappresentazione dei numeri e di calcolo delle operazione aritmetiche.
  Si astrae dal problema di eseguire le operazioni.

Espressione booleane: regole SOS

Assiomi . . .

$$\frac{\langle a_0, \sigma\rangle \;\Rightarrow\; n \qquad \langle a_1, \sigma\rangle \;\Rightarrow\; n}{\langle a_0 = a_1, \sigma\rangle \;\Rightarrow\; \textbf{true}}$$

$$\frac{\langle a_0, \sigma\rangle \;\Rightarrow\; n \qquad \langle a_1, \sigma\rangle \;\Rightarrow\; m}{\langle a_0 = a_1, \sigma\rangle \;\Rightarrow\; \textbf{false}} \qquad n \neq m$$

. . .

$$\frac{\langle b_0, \sigma\rangle \;\Rightarrow\; t_0 \qquad \langle b_1, \sigma\rangle \;\Rightarrow\; t_1}{\langle b_0 \;\textbf{and}\; b_1, \sigma\rangle \;\Rightarrow\; \textbf{t}}$$

dove $t \equiv \textbf{true}$ se $t_0 \equiv \textbf{true}$ e $t_1 \equiv \textbf{true}$. Altrimenti $t \equiv \textbf{false}$.

Dare regole alternative per in connettivo **and** .

Attraverso le regole posso definire in maniera esplicite per le operazione aritmetiche. Senza demandare alle side condition.
Semplificazione: rappresentiamo i naturali e non gli interi

$$n ::= 0 \mid Sn$$

Regole per l'addizione, prodotto, confronto.

$$n ::= 0 \mid n : 0 \mid n : 1$$

Dove $[\![n : 0]\!] = 2 \times [\![n]\!]$
e $[\![n : 1]\!] = 2 \times [\![n]\!] + 1$

L'esecuzione di un comando ha l'effetto di modificare la memoria, store:

$$\langle c, \sigma \rangle \;\Rightarrow\; \sigma'$$

Per rappresentare lo store modificato si usa la notazione $\sigma[m/X]$

$$\begin{aligned} &\sigma[m/X](X) = m \\ &\sigma[m/X](Y) = \sigma(Y) \qquad \text{se } X \neq Y \end{aligned}$$

In un approccio completamente operazionale lo stato dovrebbe essere un oggetto sintattico:
grammatica per definire gli stati,
insieme di regole che ne definiscono il comportamento.

$$\langle \textbf{skip}, \sigma \rangle \;\Rightarrow\; \sigma$$

$$\frac{\langle a, \sigma \rangle \;\Rightarrow\; n}{\langle X := a,\; \sigma \rangle \;\Rightarrow\; \sigma[n/X]}$$

$$\frac{\langle c_0, \sigma \rangle \;\Rightarrow\; \sigma' \qquad \langle c_1, \sigma' \rangle \;\Rightarrow\; \sigma''}{\langle c_0; c_1,\; \sigma \rangle \;\Rightarrow\; \sigma''}$$

$$\frac{\langle b, \sigma \rangle \;\Rightarrow\; \textbf{true} \qquad \langle c_0, \sigma \rangle \;\Rightarrow\; \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\; \sigma \rangle \;\Rightarrow\; \sigma'}$$

$\ldots$

$$\frac{\langle b, \sigma \rangle \;\Rightarrow\; \textbf{true} \qquad \langle c, \sigma \rangle \;\Rightarrow\; \sigma'' \qquad \langle \textbf{while } b \textbf{ do } c\ , \sigma'' \rangle \;\Rightarrow\; \sigma'}{\langle \textbf{while } b \textbf{ do } c,\; \sigma \rangle \;\Rightarrow\; \sigma'}$$

$\ldots$

Dalla semantica, una nozione di equivalenza tra espressioni, comandi.

$$c_0 \sim c_1$$

se per ogni coppia di store σ, σ':

$$\langle c_0, \sigma \rangle \Rightarrow \sigma' \quad \text{se e solo se} \quad \langle c_1, \sigma \rangle \Rightarrow \sigma'$$

Più nozione di equivalenza:

$$c_0 \equiv c_1$$

Se i comandi c_0 e c_1, formulati nella sintassi astratta, sono uguali.

$$\sigma_0 = \sigma_1$$

se σ_0 e σ_1 sono la stessa funzione $\mathbf{Loc} \to \mathbf{N}$

- Codificare in Haskell le regole della semantica operazionale.
- Posto:

$$w \equiv \textbf{while } b \textbf{ do } c$$

dimostrare che:

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

e

$$w \sim \textbf{if } b \textbf{ then } w \textbf{ else skip}$$

Le regole sono deterministiche:

- Formulazione forte:
  Per ogni c, σ esiste al più un σ' ed una singola dimostrazione di:

$$\langle c, \sigma \rangle \Rightarrow \sigma'$$

- Formulazione debole:
  Per ogni c, σ esiste al più un σ' per cui valga

$$\langle c, \sigma \rangle \Rightarrow \sigma'$$

Formulazione alternativa: descrive un passo di computazione.

$$\langle c, \sigma \rangle \ \rightarrow \ \langle c', \sigma' \rangle$$

Nuove regole per il while

$$\frac{\langle b, \sigma \rangle \ \rightarrow \ \textbf{true}}{\langle \textbf{while } b \textbf{ do } c, \ \sigma \rangle \ \rightarrow \ \langle c; \textbf{while } b \textbf{ do } c, \ \sigma \rangle}$$

Seconda regole per il while ...

Vengono usate entrambe le formulazioni:

- Per alcuni linguaggi è più semplice fornire la big-step semantics. Più astratta.
- Nei linguaggi per la concorrenza è fondamentale considerare la small-step semantics. Contiene informazioni aggiuntive sui passi di computazione, e sull'ordine di esecuzione.
- può essere non banale dimostrare l'equivalenza tra le due formulazioni.

In matematica e informatica molte definizione induttive:

- numeri naturali,
- liste,
- grammatiche,
- derivazioni, dimostrazioni

Insiemi di costruttori:

- zero, successore;
- lista vuota, concantenazione;
- costrutti della grammatica;
- assiomi, regol di derivazione.

Sulle strutture induttive:

- definizioni per ricorsione,
- dimostrazioni per induzione.

Rafforzamenti, estensioni:

- induzione generalizzata,

$$\forall n \,.\, (\forall m < n \,.\, P(m)) \Rightarrow P(n)$$

allora

$$\forall n \,.\, P(n)$$

- insiemi ben fondati.

Associo a espressioni di IMP oggetti matematici (funzioni (parziali))
in maniera composizionale.

Elementi basi della costruzione:

- $N = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, l'insieme dei numeri interi.
- $T = \{\mathit{true}, \mathit{false}\}$, l'insieme dei valori booleani.
- $\Sigma = \mathbf{Loc} \to N$, l'insieme dei possibili stati (configurazioni della memoria,dello store).

Notare la differenza tra $\mathbf{N}$ e N.

Funzioni di interpretazione

A ogni categoria sintattica si associa una funzione di interpretazione:

- $\mathcal{A}[\![\ ]\!] : \mathbf{AExp} \ \to (\Sigma \to N)$
  un'espressione aritmetica rappresenta una funzione da stato a numero intero.
- $\mathcal{B}[\![\ ]\!] : \mathbf{BExp} \ \to (\Sigma \to T)$
- $\mathcal{C}[\![\ ]\!] : \mathbf{Com} \ \to (\Sigma \rightharpoonup \Sigma)$
  un comando rappresenta una funzione parziale, da stato in stato.

Riprendo le idee della semantica operazionale ma le esprimo in maniera differente.

Uso la doppia parentesi $[\![\ ]\!]$ per racchiudere gli elementi della sintassi

Semantica delle espressioni (aritmetiche, booleane)

Le funzioni di interpretazione vengono definite per induzione sulla grammatica (sulla struttura del termine).

$$\mathcal{A}[\![\mathbf{n}]\!](\sigma) = n$$

$$\mathcal{A}[\![\mathbf{X}]\!](\sigma) = \sigma(X)$$

$$\mathcal{A}[\![a_o + a_1]\!](\sigma) = (\mathcal{A}[\![a_o]\!](\sigma)) + (\mathcal{A}[\![a_1]\!](\sigma))$$

...

$$\mathcal{B}[\![a_o \leq a_1]\!](\sigma) = (\mathcal{A}[\![a_o]\!](\sigma)) \leq (\mathcal{A}[\![a_1]\!](\sigma))$$

Ogni elemento, ogni operatore, viene interpretato con il suo corrispondente semantico.

Comandi rappresentano funzioni parziali.

Nelle definizioni, rappresento le funzioni parziali attraverso il loro grafi (l'insieme di coppie "argomento, valore").

$$\mathcal{C}[\![\textbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\mathcal{C}[\![X := a]\!] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma, \quad \mathcal{A}[\![a]\!](\sigma) = n\}$$

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\begin{aligned}\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] = \quad & \{(\sigma, \sigma') \in \mathcal{C}[\![c_0]\!] \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\} \quad \cup \\ & \{(\sigma, \sigma') \in \mathcal{C}[\![c_1]\!] \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\}\end{aligned}$$

Il caso difficile: il costruttore **while**

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \mathcal{C}[\![\textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}]\!] =$$
$$\{(\sigma,\sigma) \mid \sigma \in \Sigma,\ \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \ \cup$$
$$\{(\sigma,\sigma') \in (\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\}$$

Definizione ricorsiva.

Esistenza delle soluzione:

- riduco il problema ad un problema di punto fisso,
- considero l'operatore:

$$\Gamma(R) = \begin{array}{l} \{(\sigma,\sigma) \mid \sigma \in \Sigma,\ \ \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \cup \\ \{(\sigma,\sigma') \in (R \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\} \end{array}$$

  Γ trasforma una relazione tra Σ e Σ in un' altra relazione.

$$\Gamma : \mathit{Rel}(\Sigma,\Sigma) \ \rightarrow \ \mathit{Rel}(\Sigma,\Sigma)$$

- Definisco $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ come minimo punto fisso per Γ.

Posto

$$\Omega \equiv \textbf{while true do skip}$$

Definisco

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$$

Come limite delle sue approssimazioni:

$$\mathcal{C}[\![\Omega]\!]$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c; \Omega \textbf{ else skip}]\!]$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c; (\textbf{if } b \textbf{ then } c; \Omega \textbf{ else skip}) \textbf{ else skip}]\!]$$

Teoremi di punto fisso (su ordini)

Forniscono soluzione al problema precedente.

- teorema di Knaster-Tarski: in un reticolo completo, ogni funzione monotona ha un minimo (e un massimo) punto fisso, (importante in matematica)
- in un ordine parziale completo ogni funzione monotona e continua ha un minimo punto fisso.
  (fondamentale nella semantica denotazionale)
  (si indeboliscono le proprietà dell'ordine, si rafforzano le proprietà della funzione).
- in uno spazio metrico completo, ogni funzione contrattiva ha un unico punto fisso,
  (usata in analisi)

Un ordine parziale $(P, \sqsubseteq)$ è formato da un insieme P e una relazione binaria $\sqsubseteq$ su P t.c. per ogni $p, q, r \in P$

- $p \sqsubseteq p$, (riflessiva)
- $p \sqsubseteq q$ e $q \sqsubseteq r$ allora $p \sqsubseteq r$, (transitiva)
- $p \sqsubseteq q$ e $q \sqsubseteq p$ allora $p = q$, (antisimmetrica)

Dato sottoinsieme X di P,

- X ha un maggiorante (upper bound) se esiste $p \in P$ t.c per ogni $q \in X$, $q \sqsubseteq p$.
- l'estremo superiore di X, $\bigsqcup X$ se esiste, è un maggiorante più piccolo di tutti gli altri,
  - $q \in X$, $q \sqsubseteq \bigsqcup X$, inoltre
  - per ogni p se $\forall q \in X \;.\; q \sqsubseteq p$ allora $\bigsqcup X \;\sqsubseteq\; p$

  l'estremo superiore è unico (dimostrare per esercizio).

Definizione

- Un reticolo (lattice) è un ordine parziale in cui ogni coppia di elementi ha un estremo superiore, ed estremo inferiore $\sqcap$.
  Possiamo scrivere $\bigsqcup\{p, q\}$ come $p \sqcup q$.
  Segue che in un reticolo ogni insieme finito ha insieme superiore.
- Un reticolo completo è un ordine parziale in cui ogni sottoinsieme ha estremo superiore.

Esercizio. Mostrare che in un reticolo completo ogni sottoinsieme ha anche estremo inferiore.

Definizione

Una funzione $f : P \to Q$ tra ordini è monotona se rispetta l'ordine. Se $p_1 \sqsubseteq p_2$ allora $f(p_1) \sqsubseteq f(p_2)$

Teorema (Knaster-Tarski)

Ogni funzione monotona f su un reticolo completo P possiede un minimo (massimo) punto fisso.

Si mostra che $\bigsqcap\{p \mid f(p) \sqsubseteq p\}$ è un punto fisso per f (ossia $\bigsqcap\{p \mid f(p) \sqsubseteq p\} = f(\bigsqcap\{p \mid f(p) \sqsubseteq p\})$

Le funzioni parziali non formano un reticolo.
Le relazioni sono un estensione delle funzioni parziali e formano un reticolo.
Funzioni e funzioni parziali possono essere viste come casi particolari di relazioni.

Definizione

- Una relazione tra X e Y è caratterizzata da un sottoinsieme di $X \times Y$.
- Una funzione parziale $f : X \rightharpoonup Y$ è un relazione tale che $\forall x \in X, y, y' \in Y \; . \; (x, y) \in f \wedge (x, y') \in f \;\; \rightarrow \;\; y = y'$
- una funzione totale è una funzione parziale tale che $\forall x \exists y (x, y) \in f$

$(x, y) \in f$ si scrive anche $y = f(x)$

Definizione (Composizione)

Data due relazione R tra X e Y e S tra Y e Z definiamo $S \circ R$ come

$$(x, z) \in S \circ R \Leftrightarrow \exists y \ . \ (x, y) \in R \wedge (y, z) \in S$$

Notare l'inversione nell'ordine.

Nel caso che le relazioni siano funzione, questa composizione coincide con la composizione di funzioni.

- l'insieme di relazioni su $\Sigma \times \Sigma$ forma un reticolo completo,
- l'operatore:

$$\begin{aligned}\Gamma(R) = \quad & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{false}\} \cup \\ & \{(\sigma,\sigma') \in (R \circ \mathcal{C}[\![c]\!]) \mid \mathcal{B}[\![b]\!](\sigma) = \mathit{true}\}\end{aligned}$$

  è monotono,
- il teorema di Knaster-Tarski prova l'esistenza un punto fisso per l'operatore Γ (soluzione delle definizione ricorsiva della semantica).

La definizione ricorsiva possiede più soluzioni, la soluzione minima è quella che descrive correttamente il comportamento del programma.

- È preferibile usare funzioni parziali e non relazioni.
  La soluzione precedente non prova che il punto fisso sia una funzione parziale, potrebbe essere una relazione generica.
- In IMP la semantica di un comando : $\Sigma \rightharpoonup \Sigma$.
- Lo spazio delle funzioni parziali forma un ordine, $f \sqsubseteq g$ quando le seguenti condizioni equivalenti sono soddisfatte:
  - g è più definita di f,
  - $\forall x. f(x) \downarrow \Rightarrow f(x) = g(x)$
  - come insieme di coppie (argomento, risultato) $f \subseteq g$

  questo spazio non è un reticolo.
- È necessario usare un secondo teorema di punto fisso.

.

Definizione

- In un ordine parziale P, una catena è una sequenza di elementi $p_1 \sqsubseteq p_2 \sqsubseteq p_3, \ldots$, ciascuno maggiore del precedente.
- Un ordine parziale completo (CPO) P è un ordine parziale in cui ogni catena possiede un estremo superiore.
- Un ordine parziale completo con bottom è un ordine parziale completo contenente un elemento minimo $\bot$.

CPO sono le strutture tipiche i cui interpretare programmi.

Un esempio di CPO le funzioni parziali da N in N.

Definizione (Continuità)

Una funzione $f : D \to E$ tra CPO è continua se preserva l'estremo superiore delle catene.
Per ogni catena $p_1, p_2, p_3, \ldots$, $\bigsqcup_{i \in N} f(p_i) = f(\bigsqcup_{i \in N} p_i)$.

Teorema

Ogni funzione continua su un CPO con bottom $f : D \to D$ possiede un minimo punto fisso.

Proof.

Il minimo punto fisso è l'estremo superiore della catena $\bot, f(\bot), f(f(\bot)), f^3(\bot), \ldots$.
Completare per esercizio. □

Definizione per approssimazioni del while

Applicando il teorema precedente, la funzione

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c\ ]\!]$$

è il limite della seguente catena di funzioni parziali:

$$\begin{array}{l}
\mathcal{C}[\![\bot]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \bot)]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \textbf{if } b \textbf{ then } (c; \bot))]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } (c; \textbf{if } b \textbf{ then } (c; \textbf{if } b \textbf{ then } (c; \bot)))]\!] \\
\ldots
\end{array}$$

dove indichiamo con $\bot$ un programma sempre divergente.

con **if** b **then** c si indica il comando **if** b **then** c **else skip**
zucchero sintattico, arricchisce il linguaggio senza aggiunge nuove definizioni semantiche.

Una categoria per la semantica denotazionale

Astraendo dall'esempio precedente: si interpretano i linguaggi di programmazione (tipi di dato, classi di elementi) utilizzando:

1. ordini parziali completi (CPO)
   - ordine: l'information order
   - $\perp$ rappresenta l'assenza di informazioni, il programma che diverge sempre,
   - completi per dare soluzione alle equazioni ricorsive, di punto fisso.
2. funzioni tra CPO che sono
   - monotone
   - continue: preservano l'estremo superiore di catene crescenti.

Esempi di CPO

- N con l'ordine piatto: $n \sqsubseteq m \Leftrightarrow n = m$ (ordine completo per catene).
- $N_\perp = N \cup \{\perp\}$, $n \sqsubseteq m \Leftrightarrow (n = m \vee n = \perp)$, l'insieme dei naturali con $\perp$,
- $T = \{\mathit{true}, \mathit{false}\}$, $T_\perp$
- $\mathbf{O} = \{\perp, \top\}$ con $\perp \sqsubseteq \top$, il CPO con due elementi.
- $N \to N_\perp$ con l'ordine puntale: $f \sqsubseteq g$ sse per ogni n, $f(n) \sqsubseteq g(n)$,
  isomorfo a $N \rightharpoonup N$ (con l'ordine visto in precedenza)
  si internalizza la divergenza: $f(n) = \perp$ indica che $f(n)$ non termina,
  si evitano le funzioni parziali
- $\mathit{Nat} \cup \{\infty\}$, i numeri naturali con l'ordine lineare.
- streams di valori booleani: stringhe parziali, arbitrariamente lunghe.

Significato intuitivo.

Consideriamo un programma con tipo funzionale
$F : (N \to N) \to N$

La sua semantica $[\![F]\!] : (N \to N_\perp) \to N_\perp$ è una funzione:

- monotona, perciò se $F(f) \Rightarrow 3$ allora $F(g) \to 3$ per ogni funzione g più definita di f.
  Preserva l'information order.
- Continua perciò se $F(f) = 3$ allora F genera 3 dopo aver valutato f su di un numero finito di valori,
  ossia, esiste un funzione parziale g, definita su di un numero finito di elementi tale che $g \sqsubseteq [\![f]\!]$ e $[\![F]\!](g) = 3$
  Finitarietà: per ottenere una parte finita di informazione sul risultato è sufficiente esaminare una parte finita di informazione dell'argomento.

Esercizio: mostrare che la composizione preserva la continuità

Lambda (λ) notazione

In matematica definisco le funzioni mediante equazioni nella forma: $f(x) = \sin(x) + \cos(x)$.

Con la λ notazione scrivo direttamente:
$\lambda x\ .\ \sin(x) + \cos(x)$, oppure $\lambda x \in R\ .\ \sin(x) + \cos(x)$
oppure $f = \lambda x.\sin(x) + \cos(x)$

In Haskell:

```
\ x -> (sin x ) + (cos x)
```

Vantaggi:

- name less functions, posso definire una funzione senza darle un nome,
- definizione funzioni più sintetica,
- funzioni assimilate agli altri elementi,
- concettualmente più chiara: $\int \sin(x)dx$ diventa $\int \lambda x.\sin(x)$ oppure $\int \sin$.

Costruttori di CPO

Per dare semantica a linguaggi complessi si associa a

- tipi: CPO opportunamente strutturati
- programmi e sottoespressioni di programmi: elementi di CPO, funzioni su CPO

nel fare questo

- costruiamo CPO complessi a partire ad CPO,
- usiamo funzioni e operatori standard su questi costruzioni.

CPO senza bottom

Ad un insieme di valori D, (es. N l'insieme dei numeri interi, B) associo il CPO D con ordine piatto $d_1 \sqsubseteq d_2$ sse $d_1 = d_2$.

Dal punto di vista informativo, elementi inconfrontabili: informazioni diverse, tutte completamente definite, nessuna più definita dell'altra.

Insieme di valori tutti completamente definiti.

Definizione

$D_\perp = D \cup \{\perp\}$ con relazione d'ordine:
$d_1 \sqsubseteq d_2$ sse $d_1 = \perp$ oppure $(d_1, d_2 \in D \ \wedge \ d_1 \sqsubseteq d_2)$

Dal CPO D, costruisco il CPO $D_\perp$ delle computazioni, eventualmente divergenti, che generano elementi di D.

Funzioni associate:

- $\lfloor \_ \rfloor : D \to D_\perp$,
  dato un valore d, costruisce la computazione $\lfloor d \rfloor$ che restituisce l'elemento d.
- da una funzione $f : D \to E$ definita su valori (E CPO con bottom),
  si deriva la funzione (stretta) $f^* : D_\perp \to E$ definita su computazioni.

Notazione. $(\lambda x.e)^*(d)$ si scrive anche $let\ x \Leftarrow d\ .\ e$.

Osservazioni

- si usa approccio ispirato dalla teoria delle categorie:
  per ogni costruttore si definisce le funzioni che lo caratterizzano.
  ingredienti base per la costruzione di altre funzioni continue
- i costruttori di CPO hanno una costruzione corrispondente in Haskell
- in teoria delle categorie l'operazione di Lifting definisce una monade,
  corrispondente Haskell: la monade Maybe
  corrispondenza non perfetta:
  in Haskell posso definire

```
test ::  Maybe a -> Bool
test Nothing = True
test (Just _)  = False
```

Definizione

$D \times E$ l'insieme delle coppie con la relazione d'ordine puntuale:
$\langle d_1, e_1 \rangle \sqsubseteq \langle d_2, e_2 \rangle$ sse $d_1 \sqsubseteq d_2$ e $e_1 \sqsubseteq e_2$

Si generalizza al prodotto finito.

Costruisce i CPO associati a coppie, record, vettori.

Funzioni associate:

- proiezioni $\pi_1 : (D \times E) \to D$, $\pi_1(\langle d, e \rangle) = d$,
  $\pi_2 : (D \times E) \to E$
- da una coppia di funzioni $f : C \to D,\ g : C \to E$
  si deriva la funzione $\langle f, g \rangle : C \to (D \times E)$
  $\langle f, g \rangle(c) = \langle f(c), g(c) \rangle$
  (restituisce coppie di elementi).

Definiscono un isomorfismo tra $(C \to D) \times (C \to E)$ e
$C \to (D \times E)$
Dato da ...

- Mostrare che le definizioni sono ben date: l'ordine $D \times E$ è un CPO, le funzioni π_i, $\langle f, g \rangle$ sono continue.
- Costruire $O \times O (= O^2)$, O^3. A quali ordini sono isomorfi?
- Costruire $(T_\perp)^2$

Proposizione

Una funzione $f : (C \times D) \to E$ è continua sse è continua in ciascuno degli argomenti.

Definizione

$[D \to E]$ l'insieme funzioni continue da D in E con l'ordine puntuale
$f \sqsubseteq g$ sse per ogni $d \in D$, $f(d) \sqsubseteq g(d)$.

Costruisco CPO per linguaggi funzionali.

Mostrare che $[D \to E]$ è un CPO.

Funzioni associate.

- applicazione $app : ([D \to E] \times D) \to E$
  $app(\langle f, d \rangle) = f(d)$
- currying da (Haskell Curry), una funzione
  $f : (D \times E) \to F$ induce una funzione
  $curry(f) : D \to [E \to F]$
  $curry(f)(d)(e) = f(\langle d, e \rangle)$

Definiscono un isomorfismo tra $C \to [D \to E]$ e $(C \times D) \to E)$, dato da ...

- Mostrare che l'operatore di punto fisso $Y : [D \to D] \to D$ è continuo.
- Mostrare che $[T \to D]$ è isomorfo a D^2.
- Disegnare i CPO:
  $[0 \to T]$, $[N_\perp \to T]$, $[0 \to T_\perp]$, $[N_\perp \to T_\perp]$.

Definizione

$D + E = \{\langle 1, d\rangle \mid d \in D\} \cup \{\langle 2, e\rangle \mid e \in E\}$
con ordine:

- $\langle 1, d\rangle \sqsubseteq \langle 1, d'\rangle$ sse $d \sqsubseteq d'$
- $\langle 2, e\rangle \sqsubseteq \langle 2, e'\rangle$ sse $e \sqsubseteq e'$
- $\langle 1, d\rangle$ incomparabile con $\langle 2, e\rangle$

Mostrare che $D + E$ è un CPO.

CPO associati ai variant type.

- inserzioni: $in_1 : D \to (D + E)$, $in_1(d) = \langle 1, d\rangle$,
  $in_2 : E \to (D + E)$
- dalle funzioni $f : D \to C$, $g : E \to C$
  si deriva la funzione $[f, g] : (D + E) \to C$
  $[f, g](\langle 1, d\rangle) = f(d), \quad [f, g](\langle 2, e\rangle) = g(e),$

Definiscono un isomorfismo tra $(D \to C) \times (E \to C)$ e
$[D + E] \to C$

- Definire la somma n-aria (di n CPO).
  Posso ridurre la somma n-aria ad applicazioni ripetute della somma binaria?
- Definire, tramite funzioni e costruttori standard, la semantica dell'**if then else**
- Definire, tramite funzioni e costruttori standard, la semantica delle funzioni booleane.

Funzioni su CPO possono essere costruite da un linguaggio facente uso di:

- variabili con tipo un dominio: $x_1 : D_1, \ldots, x_i : D_i$
- costanti: *true*, *false*, $-1, 0, 1, \ldots$
- funzioni base: $\lfloor \_ \rfloor$, π_i, *app*, in_i, *fix*
- costruttori: $(\_)^*$, $\langle \_, \_ \rangle$, $curry(\_)$, $[\_, \_]$,
- applicazione, lambda astrazione, composizione di funzioni.

Esempi:

- $\langle \pi_2, \pi_1 \rangle$
- $\lambda x \,.\, f(gx) \quad \lambda x \,.\, f(\pi_1 x)(\pi_2 x)$

Proposizione

Ogni espressione del metalinguaggio di tipo E e con solo le variabili $x_1 : D_1, \ldots, x_i : D_i$, denota un elemento nel CPO $[(D_1 \times \ldots \times D_i) \to E]$, ossia una funzione continua.

Proof.

Per induzione sulla struttura dell'espressione e, si definisce il significato di e e da qui la continuità. □

Metalinguaggi permette di definire oggetti matematici, con una sintassi simile alla definizione di programmi Haskell.

Metalinguaggi e Haskell definiscono oggetti di natura diversa.

Semantica (operazionale e denotazionale) di due semplici linguaggi funzionali

con due differenti meccanismi di valutazione

- call-by-value (eager) come: Standard ML, OCaml.
- call-by-name (lazy) come Haskell, Miranda.

$$
\begin{array}{rl}
t ::= & x \\
& \mathbf{n} \mid t_1 \,\mathbf{op}\, t_2 \mid \\
& (t_1, t_2) \mid \mathbf{fst}\, t \mid \mathbf{snd}\, t \mid \\
& \lambda x.t \mid (t_1\, t_2) \mid \\
& \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \\
& \mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \mid \\
& \mathbf{rec}\, x.t
\end{array}
$$

Non tutte le espressioni hanno senso,
esempio $(1 \;\; 3)$, $((\lambda x.x + 1)(2, 3))$
Controllo di tipo:

- determina le espressioni corrette,
- si deriva $t : \tau$
- tipi:

$$\tau ::= \textbf{int} \;\mid\; \tau_1 * \tau_2 \;\mid\; \tau_1 \to \tau_2$$

- regole induzione sulla struttura del termine
  ogni costrutto sintattico ha una regola del tipo:

$$\frac{x : \tau_1 \quad t_1 : \tau_1 \quad t_2 : \tau_2}{\textbf{let}\, x \Leftarrow t_1 \,\textbf{in}\, t_2 : \tau_2}$$

Nessun polimorfismo

- ogni variabile ha associato un unico tipo,
- proprietà: ogni espressione ha tipo unico,
- nessun polimorfismo: sistema di tipi e semantica denotazionale più semplice,

Mediante un sistema di regole descrivo come un termine riduce.
Due alternative:

- big-step reduction: $t \to c$ ($t \Rightarrow c$) descrive il valore c generato dalla computazione di t
  più vicina alla semantica denotazionale, meno semplice da definire.
- small-step reduction $t_1 \to t_2$ descrive come un passo di computazione trasforma il termine t_1 nel termine t_2.

  Nelle semantiche small step per linguaggi funzionali, spesso, un passo di computazione è definito come una sostituzione "termine per variabile". Questo non è un passo di computazione elementare.

Per ogni tipo, un insieme dei valori o forme canoniche:
risultati della computazione,
termini che non posso essere ridotti ulteriormente.

- **int** (tipi ground), le costanti numeriche $\ldots -1, 0, 1, 2 \ldots$
- $\tau_1 * \tau_2$, le coppie (v_1, v_2), con v_i valore.
  Riduzione eager, elementi completamente definiti.
- $\tau_1 \to \tau_2$, le λ-astrazioni $\lambda x.t$, con t non necessaria mente un valore.
  Sono possibili definizione alternative: $\lambda x.v$ con v termine non riducibile, esempi $x + 3$, $(x\ 1)$

Per definizione valori sono termini chiusi: posso valutare solo termini chiusi; questa restrizione semplifica le definizioni.

Regole di riduzione: per induzione su struttura termine

$$\frac{t_0 \Rightarrow m \quad t_1 \Rightarrow n}{t_0 + t_1 \Rightarrow o} \qquad o = m + n$$

...

$$\frac{t_0 \Rightarrow 0 \quad t_1 \Rightarrow c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \Rightarrow c}$$

$$\frac{t_0 \Rightarrow n \quad t_2 \Rightarrow c}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \Rightarrow c} \qquad n \neq 0$$

$$\frac{t_1 \Rightarrow c_1 \quad t_2 \Rightarrow c_2}{(t_1, t_2) \Rightarrow (c_1, c_2)}$$

$$\frac{t \Rightarrow (c_1, c_2)}{\textbf{fst } t \Rightarrow c_1}$$

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_2 \;\Rightarrow\; v_2 \quad t_1'[v_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

$$\frac{t_1 \;\Rightarrow\; c_1 \quad t_2[c_1/x] \;\Rightarrow\; c}{\mathbf{let}\, x \Leftarrow t_1\, \mathbf{in}\, t_2 \;\Rightarrow\; c}$$

$$\mathbf{rec}\, x.\lambda y\, .t \;\Rightarrow\; \lambda y\, .t[\mathbf{rec}\, x.\lambda y\, .t\, /\, x]$$

Proprietà della riduzione

- riduzione deterministica, ogni termine riduce al più ad un valore, esiste al più una regola applicabile;
- le riduzioni preservano il tipo: subject reduction.
- **rec** derivazioni infinite corrispondono a computazioni infinite. Esempio: $((\mathbf{rec}\, f^{\mathbf{N}\to\mathbf{N}}.\, \lambda y.\, (f\ \ 1))\ 2)$
- Esercizio:
  $((\mathbf{rec}\, f.\lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ (f(x-1))+2)\ 2)$

Domini per la semantica denotazionale

Si distingue tra:

- domini per interpretare valori;
- domini per interpretare computazioni.

Domini per valori: per induzione sul struttura del tipo.

- $V_{\mathbf{int}} = N$
- $V_{\tau_1 * \tau_2} = V_{\tau_1} * V_{\tau_2}$
- $V_{\tau_1 \to \tau_2} = [V_{\tau_1} \to (V_{\tau_2})_\perp]$

Domini per le computazioni:

$$(V_\tau)_\perp$$

Ambiente

L'interpretazione di un termine aperto dipende da come interpretiamo le variabili (dall'ambiente).
Nei linguaggi call-by-value, le variabili denotano valori.
Env, l'insieme delle funzioni da variabili nei domini per le computazioni

$$\rho : \mathbf{Var} \to \sum_{\tau \in \mathcal{T}} V_\tau$$

che rispettano i tipi $x : \tau$ implica $\rho(x) \in V_\tau$.

L'interpretazione di un termine $t : \tau$,

$$[\![t]\!] : \mathbf{Env} \to (V_\tau)_\perp$$

Definizioni per induzione sul termine

$$[\![x]\!] = \lambda\rho.\, \lfloor \rho(x) \rfloor$$

$$[\![\mathbf{n}]\!] = \lambda\rho.\, \lfloor n \rfloor$$

$$[\![t_1 \,\mathbf{op}\, t_2]\!] \;=\; \lambda\rho.\, [\![t_1]\!]\rho \;op_\perp\; [\![t_2]\!]\rho$$

$$[\![(t_1, t_2)]\!] = \lambda\rho.\, let\ v_1 \Leftarrow [\![t_1]\!]\rho.\, let\ v_2 \Leftarrow [\![t_2]\!]\rho.\, \lfloor (v_1, v_2) \rfloor$$

$$[\![(\mathbf{fst}\, t)]\!] = \lambda\rho.\, let\ v \Leftarrow [\![t]\!]\rho.\, \lfloor \pi_1(v) \rfloor$$

$$[\![\lambda x.t]\!] = \lambda\rho.\, \lfloor \lambda v : V_\sigma.\, [\![t]\!](\rho[v/x]) \rfloor$$

$$[\![(t_1\, t_2)]\!] = \lambda\rho.\, let\ v_1 \Leftarrow [\![t_1]\!]\rho.\, let\ v_2 \Leftarrow [\![t_2]\!]\rho.\, v_1(v_2)$$

$$[\![\mathbf{rec}\, x.\lambda y.\, t]\!] = \lambda\rho.\, Fix(\lambda v_1 : V_{\sigma\to\tau}.\, \lambda v_2 : V_\sigma.\, [\![t]\!]\, \rho[v_1/x, v_2/y])$$

Proprietà della semantica denotazionale

- substitution lemma
  Se $[\![s]\!]\rho = v$ allora $[\![t[s/x]]\!]\rho = [\![t]\!]\rho[v/x]$
  Si dimostra per induzione sulla struttura del termine t
- per ogni valore c, $[\![c]\!] \neq \bot$

Correttezza della semantica operazionale

Proposizione

Per ogni termine t, e valore c,

$$t \Rightarrow c \text{ implica } [\![t]\!] = [\![c]\!]$$

Si dimostra per induzione sulla derivazione di $t \Rightarrow c$.
Le regole della semantica operazionale rispettano quella denotazionale.

L'implicazione opposta non è valida,
perché esistono valori diversi con la stessa semantica denotazionale.
$[\![c]\!] = [\![c']\!]$ ma $c \not\Rightarrow c'$

Vale una relazione più debole:

Proposizione

Per ogni termine t, e valore c,

$$[\![t]\!] = [\![c]\!] \text{ implica l'esistenza di } c' \text{ tale che } t \Rightarrow c'.$$

Se $[\![t]\!]$ è diversa da $\perp$ allora le computazioni su t terminano.

Dimostrazione non ovvia; usa tecniche sofisticate, "logical relations".

Corollario

Per ogni $t : int$ e valore intero n

$$t \Rightarrow n \text{ sse } [\![t]\!] = [\![n]\!]$$

Differenti insiemi di valori, sui vari tipi.

- **int** (tipi ground), le costanti numeriche $\ldots -1, 0, 1, 2 \ldots$
- $\tau_1 * \tau_2$, le coppie (t_1, t_2), con t_i chiusi, non necessariamente valori.
  Riduzione lazy? Definendo valori in questo modo posso gestire liste infinite, indipendentemente dal meccanismo di riduzione.
- $\tau_1 \to \tau_2$, le λ-astrazioni $\lambda x.t$, con t chiuso non necessariamente un valore.

Differenze rispetto alla semantica call-by-name:

$$\frac{t_1 \;\Rightarrow\; \lambda x.t_1' \quad t_1'[t_2/x] \;\Rightarrow\; c}{(t_1\, t_2) \;\Rightarrow\; c}$$

$$\frac{t_2[t_1/x] \;\Rightarrow\; c}{\mathbf{let}\, x \Leftarrow t_1\, \mathbf{in}\, t_2 \;\Rightarrow\; c}$$

$$\frac{t[\mathbf{rec}\, x.t \,/\, x] \;\Rightarrow\; c}{\mathbf{rec}\, x.t \;\Rightarrow\; c}$$

Manca la regola per le coppie

Principali differenze

- valutazione degli argomenti call-by-name,
- per uniformità, valutazione call-by-name anche per il costrutto **let**,
- la ricorsione applicabile tutti gli elementi,
- il diverso insieme di valori per il tipo coppia forza regole diverse.

Proprietà preservate:

- riduzione deterministica, ogni termine riduce al più ad un valore, esiste al più una regola applicabile;
- le riduzioni preservano il tipo; (subject reduction).

Domini per la semantica denotazionale

Domini per valori:

- $V_{\mathbf{int}} = N$
- $V_{\tau_1 * \tau_2} = (V_{\tau_1})_\perp * (V_{\tau_2})_\perp$
- $V_{\tau_1 \Rightarrow \tau_2} = [(V_{\tau_1})_\perp \Rightarrow (V_{\tau_2})_\perp]$

Domini per le computazioni.

$$(V_\tau)_\perp$$

Appunto: nei linguaggi "call-by-name" è possibile semplificare i domini non distinguendo domini per valori e per espressioni.
Ambiente. Nei linguaggi call-by-name, le variabili denotano computazioni.
Env, l'insieme delle funzioni da variabili nei domini per le computazioni

$$\rho : \mathbf{Var} \to \sum_{\tau \text{ tipo}} (V_\tau)_\perp$$

che rispettano i tipi.

Definizioni per induzione sul termine

Insiemistica e categoriale

$$[\![x]\!] = \lambda\rho.\, \rho(x) \;=\; \pi_x$$

$$[\![\mathbf{n}]\!] = \lambda\rho.\, \lfloor n \rfloor \;=\; \lfloor \_ \rfloor \circ n \circ 1$$

$$[\![t_1 \,\mathbf{op}\, t_2]\!] \;=\; \lambda\rho.\, [\![t_1]\!]\rho \; op_\perp \; [\![t_2]\!]\rho \;=\; op_\perp \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle$$

$$[\![(t_1, t_2)]\!] = \lambda\rho.\, \lfloor ([\![t_1]\!]\rho, [\![t_2]\!]\rho) \rfloor \;=\; \lfloor \_ \rfloor \circ \langle [\![t_1]\!], [\![t_2]\!] \rangle$$

$$[\![(\mathbf{fst}\; t)]\!] = \lambda\rho.\, let\; v \Leftarrow [\![t]\!]\rho.\, \pi_1(v) \qquad [\![\mathbf{fst}]\!] = \lfloor ((\pi_1)^*) \rfloor$$

$$[\![\lambda x.t]\!] = \lambda\rho.\, \lfloor \lambda v : (V_\sigma)_\perp .\, [\![t]\!](\rho[v/x]) \rfloor \;=\; \lfloor \_ \rfloor \circ curry([\![t]\!] \circ \prod_{y \in \mathbf{Var}} (in_y \circ f_y))$$

$$[\![(t_1\, t_2)]\!] = \lambda\rho.\, let\; v \Leftarrow [\![t_1]\!]\rho.\, v([\![t_2]\!]\rho) \;=\; app \;\circ \langle (id)^* \circ [\![t_1]\!], [\![t_2]\!] \rangle$$

$$[\![\mathbf{rec}\, x.t]\!] = \lambda\rho.\, Fix(\lambda v : (V_\sigma)_\perp [\![t]\!](\rho[v/x])) \;=\; Fix \circ curry([\![t]\!] \circ \prod_{y \in \mathbf{Var}} (in_y \circ f_y$$

dove $f_x = \pi_1$ e $f_y = \pi_y \circ \pi_2$ se $x \neq y$

Proprietà della semantica denotazionale

- substitution lemma
  Se $[\![s]\!]\rho = v$ allora $[\![t[s/x]]\!]\rho = [\![t]\!]\rho[v/x]$
  Si dimostra per induzione sulla struttura del termine t
- per ogni valore c, $[\![c]\!] \neq \bot$

Confronto tra semantiche: correttezza e adeguatezza

Proposizione (Correttezza)

Per ogni termine t, e valore c,

$$t \Rightarrow c \text{ implica } [\![t]\!] = [\![c]\!]$$

Proposizione (Adeguatezza)

Per ogni termine t, e valore c,

$$[\![t]\!] = [\![c]\!] \text{ implica l'esistenza di } c' \text{ tale che } t \Rightarrow c'.$$

Corollario

Per ogni $t : \mathit{int}$ e valore intero n

$$t \Rightarrow n \text{ sse } [\![t]\!] = [\![n]\!]$$

Uguaglianza osservazionale e full-abstraction

Due termini t_1, t_2 sono uguali per la semantica operazionale, $t_1 \sim t_2$, se:
per ogni contesto $C[\ ]$

$$C[t_1]\downarrow \ \Leftrightarrow C[t_2]\downarrow$$

Per il teorema di adeguatezza:

$$[\![t_1]\!] = [\![t_2]\!] \text{ implica } t_1 \sim t_2$$

L'implicazione contraria (full abstraction) è vera solo per poche semantiche denotazionali.

Tipi somma

Possiamo arricchire il linguaggi funzionale con tipi somma

$$\tau ::= \ldots \mid \tau_1 + \tau_2$$

Costruttori e distruttori

$$t ::= \ldots \mathbf{inl}(t) \mid \mathbf{inr}(t) \mid \mathbf{case}\ t\ \mathbf{of}\ \mathbf{inl}(x_1).t_1,\ \mathbf{inr}(x_2).t_2$$

Semantica denotazionale di Haskell

Estendiamo le semantiche operazionale e denotazionale del linguaggio funzionale lazy sino a comprendere buona parte di Haskell.
Punti critici.

- Definizioni di tipi (non ricorsive).
- Definizioni ricorsive per pattern matching.
- Definizioni ricorsive di tipi.
- Polimorfismo

Core (Kernel) Language

Nel definire la semantica di un linguaggio, o la sua implementazione e utile considerare i core Language:

- linguaggio semplice, insieme ridotto di costruttori primitive, sottoinsieme ridotto del linguaggio principale,
- linguaggio principale può essere (facilmente) tradotto nel linguaggio core,
- approccio modulare all'implementazione, alla semantica

Dal punto di vista teorico i linguaggi core risaltano:

- i meccanismi fondamentali della computazione,
- similitudini: differenze tra linguaggi.

System FC

- Lambda calcolo tipato :
  applicazione e λ-astrazione;
- tipi di dato algebrici (ricorsivi):
  un insieme di funzioni basi, costruttori e distruttori;
- tipi polimorfi
- equivalenze tra tipi: operatori di coercizione.

Definizione di tipi (non ricorsive)

Ad ogni tipo definito associamo un tipo Fun.
Ad ogni costruttore una funzione.

```
data TypeId  =  Const1  TypeExp11 ... TypeExp1M |
                Const2  TypeExp21 ... TypeExp2N |
                ...
```

Dominio di valori:

$$\begin{aligned} V_{TId} = \quad & ((V_{TExp11})_\perp \times \ldots \times (V_{TExp1M})_\perp) + \\ & (TExp21)_\perp \times \ldots \times (V_{TExp2N})_\perp) + \\ & \ldots \end{aligned}$$

$$[\![\texttt{Const1}]\!]_\rho = \lfloor \lambda v_1 : (V_{TExp11})_\perp \ldots \lambda v_N : (V_{TExp1N})_\perp . \lfloor in_i \langle v_1, \ldots, v_N \langle \rfloor \rfloor$$

Distruttore costrutto case.

```
data Nat = O | S Nat
```

La definizione genera due funzioni costruttori di tipo:

```
O :: Nat
S :: Nat -> Nat
```

una funzione distruttore di tipo case.

Definizioni per pattern matching ridotte a uso di funzioni case su argomenti singoli

Esempio:

```
add x O     = x
add x (S y) = S (add x y)
```

viene tradotta nel core language in:

```
let add = rec add' . \ x y -> case y of O -> x |
                                        S y1 -> S (add' x y1)
in
```

Pattern con più argomenti

La definizione

```
and True  True   = True
and x     False  = False
and False x      = False
```

diventa:

```
let and = \x y -> case x of True -> case y of True -> True
                                              False -> False
                            False -> False
```

Il costrutto case rende esplicito rende esplicito l'ordine di valutazione degli argomenti.
Pattern matching porta a definizione più compatte leggibili eleganti.

Pattern matching: altro esempio

La definizione :

```
leq O     x     = True
leq (S x) O     = False
leq (S x) (S y) = leq x y
```

Tradotta in:

```
let leq =  rec leq' .
  \ x \ y  -> case x of O -> True
                       S x1 -> case y of O -> False
                                         S y1 ->  leq' x1 y1
in ...
```

Più complesso definire funzioni mutuamente ricorsive:
si definisce ricorsivamente un enupla di funzioni, un elemento in un tipo prodotto.

Necessario definire soluzioni di equazione.

```
data Nat = O | S Nat
```

$$LazyNat \cong 1 + (LazyNat)_\perp$$

```
data ListNat = Empty | Cons Nat ListNat
```

$$ListLazyNat \cong 1 + (LazyNat)_\perp \times (ListLazyNat)_\perp$$

È sufficiente trovare CPO che soddisfino l'equazione a meno di isomorfismi, ossia la parte sinistra e destra dell'equazione sono isomorfe tra loro.

In Haskell posso definire tipi ricorsivi a cui non è banale associare un CPO

```
data FunBool = Fun [FunBool -> Bool]
```

$FunBool \cong (Funbool_{\perp} \rightarrow T_{\perp})$

```
data Untyped = Abst [Untyped -> Untyped]
```

$Un \cong (Un_{\perp} \rightarrow Un_{\perp})$

Esistono diverse tecniche per la soluzione di equazioni di dominio.
Tutte richiedo un arricchimento della teoria dei CPO.
Presenteremo quella basata sugli Information System

Diversa presentazione di un CPO e dei suoi elementi.
Un elemento di un CPO è descritto come un sottoinsieme di informazioni elementari consistenti tra di loro.

Un sistema di informazioni elementari (Information System) è dato da:

- un insieme A di token (informazioni elementari),
- $Con \subseteq \wp_f(A)$, un predicato di consistenza, Con definisce gli insiemi finiti di token consistenti.
- una relazione di entailment (implicazione) $\vdash \subseteq (Con \times A)$
  $X \vdash a$ indica che le informazioni contenute in X implicano l'informazione a

Esempi: N, B, 1, $1_\perp$, $B_\perp$, $[B_\perp \to B_\perp]$, $[B_\perp \to [B_\perp \to B_\perp]]$

Information System: definizione

Un information system $A, Con, \vdash$ deve soddisfare le seguenti condizioni:

1. $Con(X)$ e $Y \subseteq X$ implica $Con(Y)$,
2. per ogni $a \in A$ si ha $Con(\{a\})$,
3. $X \vdash a$ implica $Con(X \cup \{a\})$ e $X \neq \emptyset$,
4. $Con(X)$ e $a \in X$ implica $X \vdash a$,
5. $Con(X), Con(Y)$, $X \vdash^* Y$ e $Y \vdash a$ implica $X \vdash a$

   dove $X \vdash^* Y$ indica $\forall b \in Y . X \vdash b$

Da Information System a CPO

Il CPO $|\mathcal{A}|$ associato all'information system $\mathcal{A} = (A, Con, \vdash)$ è formato da quei sottoinsieme x di A tali che:

1. $Con(x)$, x è consistente, ossia: per ogni $Y \subseteq_f x$, $Con(Y)$,
2. x è chiuso per la relazione di entailment, ossia: per ogni $Y \subseteq_f x$, $Y \vdash a$ implica $a \in x$

L'ordine su $|\mathcal{A}|$ è l'information order
$x \sqsubseteq y$ se e solo se $x \subseteq y$

$|\mathcal{A}|$ è un CPO con elemento minimo.

Definizione alternativa:
Definisco il CPO $\lceil \mathcal{A} \rceil$ aggiungendo la condizione

- $x \neq \emptyset$, diverso da vuoto.
  - Winskel usa questa seconda definizione.
  - $|\mathcal{A}| \cong (\lceil \mathcal{A} \rceil)\perp$
  - $\lceil \mathcal{A} \rceil$ può CPO senza elemento minimo $\perp$.
  - $|\mathcal{A}|$ genera il CPO delle computazioni, $\lceil \mathcal{A} \rceil$ il CPO dei valori.

Considerazioni

- Information system differenti possono indurre lo stesso CPO.
- Esistono CPO non definibili attraverso information system
- La classe dei CPO definibili da information system prende il nome di domini di Scott (Dana Scott).
  Consistently complete, ω-algebraic CPO.
- Esistono varianti degli information system senza la relazione di entailment, e in cui la relazione di coerenza è binaria.
  Questi danno luogo agli spazi coerenti.
- La corrispondenza Domini di Scott, Information System è un esempio si Stone Dualità, gli elementi di uno spazio posso essere descritti dalle loro proprietà.

Costruzioni sugli Information System: Lifting

L'information system vuoto:

$$\mathcal{O} = (\emptyset, \{\emptyset\}, \emptyset)$$

$$|\mathcal{O}| = \{\bot\}$$

Dato l'information system: $\mathcal{A} = (A, Con, \vdash)$
definiamo $\mathcal{A}_\bot = (A_\bot, Con_\bot, \vdash_\bot)$
come:

- $A_\bot = A \uplus \{*\}$
- $Con_\bot(X)$ sse $\exists Con(Y) \,.\, (X = in_l\, Y \ \vee \ X = (in_l\, Y \cup \{in_r\, *\}))$
- $X \vdash_\bot a$ sse $a = in_r * \ \vee$
  $\exists (Y \vdash b) \,.\, a = in_l b \ \wedge \ (X = in_l Y \ \vee \ X = in_l Y \cup \{in_r *\}$

Si ha:

$$|\mathcal{A}_\bot| \cong |\mathcal{A}|_\bot \quad \text{e} \quad \lceil \mathcal{A}_\bot \rceil \cong \lceil \mathcal{A} \rceil_\bot$$

Dati: $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$
definiamo $\mathcal{A} \times \mathcal{B} = (A \uplus B, Con_{A\times B}, \vdash_{A\times B})$
come:

- $Con_{A\times B}(in_l\, X \cup in_r\, Y)$ sse $Con_A(X)$ e $Con_B(Y)$
- $(in_l\, X \cup in_r\, Y) \vdash_{A\times B} (in_l\, a)$ sse $X \vdash_A a$
  $(in_l\, X \cup in_r\, Y) \vdash_{A\times B} (in_r\, b)$ sse $Y \vdash_B b$

Si ha:

$$|\mathcal{A} \times \mathcal{B}| \cong |\mathcal{A}| \times |\mathcal{B}|$$

Dati: $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$
definiamo $\mathcal{A} \otimes \mathcal{B} = (A \times B, Con_{A\otimes B}, \vdash_{A\otimes B})$
come:

- $Con_{A\otimes B}(Z)$ sse $Con_A(\pi_1 Z)$ e $Con_B(\pi_2 Z)$
- $Z \vdash_{A\otimes B} c$ sse $\pi_1 Z \vdash_A \pi_1 c \quad \wedge \quad \pi_2 Z \vdash_A \pi_2 c$

Si ha:

$$\lceil \mathcal{A} \otimes \mathcal{B} \rceil \cong \lceil \mathcal{A} \rceil \times \lceil \mathcal{B} \rceil$$

Dati gli information system $\mathcal{A} = (A, Con_A, \vdash_A)$ e
$\mathcal{B} = (B, Con_B, \vdash_B)$
definiamo $\mathcal{A} + \mathcal{B} = (A \uplus B, Con_{A+B}, \vdash_{A+B})$
come

- $Con_{A\times B}(X)$ sse $X = in_l\, Y \ \wedge \ Con_A(Y)$
  o $X = in_r\, Y \ \wedge \ Con_B(Y)$
- $(in_l\, X) \vdash_{A+B} (in_l a)$ sse $X \vdash_A a$
  $(in_r\, Y) \vdash_{A_B} (in_r b)$ sse $Y \vdash_B b$

Si ha:

$$|\mathcal{A} + \mathcal{B}| \cong \text{somma coalescente di} |\mathcal{A}|, |\mathcal{B}|$$

$$\lceil \mathcal{A} + \mathcal{B} \rceil \cong \lceil \mathcal{A} \rceil + \lceil \mathcal{B} \rceil$$

Dati $\mathcal{A} = (A, Con_A, \vdash_A)$ e $\mathcal{B} = (B, Con_B, \vdash_B)$
definiamo $\mathcal{A} \to \mathcal{B} = (C, Con_C, \vdash_C)$
come:

- $C = \{(X, b) \mid X \in Con_A, b \in B\}$
- $Con_C(\{(X_1, b_1), \ldots, (X_n, b_n)\})$ sse $\forall I \subseteq \{1, \ldots, n\}$.
  $Con_A(\bigcup_{i \in I} X_i) \Rightarrow Con_B(\{b_i \mid i \in I\})$
- $\{(X_1, b_1), \ldots (X_n, b_n)\} \vdash_C (X, b)$ sse $\{b_i \mid (X \vdash_A^* X_i)\} \vdash_B b$

Proposizione:

$$|\mathcal{A} \to \mathcal{B}| \cong [|\mathcal{A}| \to |\mathcal{B}|]$$

Lo spazio delle funzioni (continue) strette $[C \to_\perp D]$
ossia le funzioni f t.c. $f(\perp) = \perp$
viene descritto dal IS che ripete la costruzione precedente
modificando solo la definizione del primo punto:
$C = \{(X, b) \mid X \in Con_A, X \neq \emptyset, b \in B\}$

Proposizione:

$$|\mathcal{A} \to_\perp \mathcal{B}| \cong [|\mathcal{A}| \to_\perp |\mathcal{B}|] \cong [\lceil \mathcal{A} \rceil \to \lceil (\mathcal{B})_\perp \rceil]$$

Definiamo una relazione d'ordine, $\trianglelefteq$, sugli IS come:

$$(A, Con_A, \vdash_A) \trianglelefteq (B, Con_B, \vdash_B)$$

sse

- $A \subseteq B$
- $Con_A(X)$ sse $X \subseteq A \ \wedge \ Con_B(X)$
- $X \vdash_A a$ sse $(X \cup \{a\}) \subseteq A \ \wedge \ X \vdash_B a$

Proposizione

L'insieme degli IS con $\trianglelefteq$ forma un CPO con $\bot$, l'elemento minimo è $\prime$, il sup di una catena $\mathcal{A}_0 \trianglelefteq \mathcal{A}_1 \trianglelefteq \mathcal{A}_2 \ldots$ con $\mathcal{A}_i = (A_i, Con_i, \vdash_i)$ si ottiene attraverso l'operazione unione
$\bigsqcup_{i\in\omega} \mathcal{A}_i = (\bigcup_{i\in\omega} A_i, \ \bigcup_{i\in\omega} C_i, \ \bigcup_{i\in\omega} \vdash_i \ )$

Proposizione

I costruttori di IS $\bot$, $+$, $\times$, $\to$, $\to_\bot$ sono continui rispetto a $\trianglelefteq$.

Corollario

Ogni costruttore composizione dei costruttori base ammette punto fisso.

Linguaggio eager con tipi ricorsivi

Tipi:

$$\tau ::= \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid X \mid \mu X.\tau$$

Espressioni

$$\begin{aligned} t ::= \quad & x \\ & () \mid (t_1, t_2) \mid \mathbf{fst}\, t \mid \mathbf{snd}\, t \mid \\ & \lambda x.t \mid (t_1\, t_2) \mid \\ & \mathbf{inl}(t) \mid \mathbf{inr}(t) \mid \mathbf{case}\ t\ \mathbf{of}\ \mathbf{inl}(x_1).t_1,\ \mathbf{inr}(x_2).t_2 \mid \\ & \mathbf{abs}\,(t) \mid \mathbf{rep}\,(t) \mid \\ & \mathbf{rec}\, f.\lambda x.t \end{aligned}$$

Un unico tipo base, Nat definito per ricorsione di tipo.

Type checking

- ogni variabile ha associato un unico tipo, $x_\tau \Rightarrow x$;
- regole induzione sulla struttura del termine,
  ogni costrutto sintattico ha una regola di tipo;
- nuove regole: unit:

$$() : \mathbf{1}$$

  abstraction:

$$\frac{t : \tau[\mu X.\tau/X]}{\mathbf{abs}\,(t) : \mu X.\tau}$$

  representation:

$$\frac{t : \mu X.\tau}{\mathbf{rep}\,(t) : \tau[\mu X.\tau/X]}$$

- type conversion, type casting esplicito;
- necessarie assicurare l'unicità del tipo;
- utili se il CPO associato a $\mu X.\tau$ è isomorfo a quello associato a $\tau[\mu X.\tau/X]$,

A causa dei tipi ricorsivi
deve essere definito in mediante un insieme regole induttive,
e non per induzione sulla struttura del tipo

$$\frac{c : C_{\tau[\mu X.\tau/X]}}{\mathbf{abs}\,(c) : C_{\mu X.\tau}}$$

Nuove regole per termini nei tipi ricorsivi.

$$\frac{t \;\Rightarrow\; c}{\mathbf{abs}\,(t) \;\Rightarrow\; \mathbf{abs}\,(c)}$$

$$\frac{t \;\Rightarrow\; \mathbf{abs}\,(c)}{\mathbf{rep}\,(t) \;\Rightarrow\; c}$$

Mediante information system

Associazione tipi – information system fatta mediante:

- un ambiente di tipo (type environment);
- un insieme di definizioni, per induzione sulla struttura di tipo;

$$\mathcal{V}[\![\mathbf{1}]\!]_\chi = \mathcal{O}_\perp \cong (\{*\}, \{\{*\}, \emptyset\}, \{\{*\} \vdash *\})$$
$$\mathcal{V}[\![\tau_1 * \tau_2]\!]_\chi = \mathcal{V}[\![\tau_1]\!]_\chi \otimes \mathcal{V}[\![\tau_2]\!]_\chi$$
$$\mathcal{V}[\![\tau_1 + \tau_2]\!]_\chi = \mathcal{V}[\![\tau_1]\!]_\chi + \mathcal{V}[\![\tau_2]\!]_\chi$$
$$\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi \to_\perp \mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$
$$\mathcal{V}[\![X]\!]_\chi = \chi(X)$$
$$\mathcal{V}[\![\mu X.\tau]\!]_\chi = \mu I.\mathcal{V}[\![\tau]\!]_{\chi[I/X]}$$

Nota: $|\mathcal{V}[\![\tau]\!]|$ definisce il CPO delle computazioni associate a τ, e $\lceil\mathcal{V}[\![\tau]\!]\rceil$ definisce il CPO dei valori associate a τ,
Per semplificare non definiamo due information system (delle computazioni e quello dei valori),

Le equazioni vanno riformulate per adattarsi ai domini definiti in termini di information systems.

abs e **rep** sono rappresentati dalla funzione identità.

Valgono le usuali proprietà di correttezza e adeguatezza.

Tipi:

$$\tau ::= \mathbf{0} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid X \mid \mu X.\tau$$

Espressioni

$$\begin{aligned} t ::= \quad & x \\ & \bullet \mid (t_1, t_2) \mid \mathbf{fst}\, t \mid \mathbf{snd}\, t \mid \\ & \lambda x.t \mid (t_1\, t_2) \mid \\ & \mathbf{inl}(t) \mid \mathbf{inr}(t) \mid \mathbf{case}\ t\ \mathbf{of}\ \mathbf{inl}(x_1).t_1,\ \mathbf{inr}(x_2).t_2 \mid \\ & \mathbf{abs}\,(t) \mid \mathbf{rep}\,(t) \mid \\ & \mathbf{rec}\, x.t \end{aligned}$$

0 non contiene nessun valore, $\bullet$ termine divergente, nessuna regola di riduzione.

Vale ancora la regola eager.

$$\frac{c : C_{\tau[\mu X.\tau/X]}}{\mathbf{abs}\,(c) : C_{\mu X.\tau}}$$

Mentre, sono valori:

$$(t_1, t_2),\ \mathbf{inl}(t),\ \mathbf{inr}(t)$$

Valgono le regole eager per termini in tipi ricorsivi:

$$\frac{t \ \Rightarrow\ c}{\mathbf{abs}\,(t) \ \Rightarrow\ \mathbf{abs}\,(c)}$$

$$\frac{t \ \Rightarrow\ \mathbf{abs}\,(c)}{\mathbf{rep}\,(t) \ \Rightarrow\ c}$$

$$\mathcal{V}[\![\mathbf{0}]\!]_\chi = \mathcal{O}$$

$$\mathcal{V}[\![\tau_1 * \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi \times \mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$

$$\mathcal{V}[\![\tau_1 + \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi)_\perp + (\mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\chi = (\mathcal{V}[\![\tau_1]\!]_\chi \to \mathcal{V}[\![\tau_2]\!]_\chi)_\perp$$

$$\mathcal{V}[\![X]\!]_\chi = \chi(X)$$

$$\mathcal{V}[\![\mu X.\tau]\!]_\chi = \mu I.\mathcal{V}[\![\tau]\!]_{\chi[I/X]}$$

Semantica denotazionale linguaggi imperativi

IMP linguaggio ridotto semantica denotazionale semplice, banale.

Linguaggi più complessi portano a semantiche più complesse.

Presentiamo una rassegna delle tecniche idea usate nella semantica di linguaggi realistici.

Problemi e soluzioni

1. Input/Output. Allo store si aggiunge la lista dei valori di input e output
2. Comandi che generano errori. Il valore "errore" come risultato di una computazione.
3. Espressioni che modificano lo stato, che non terminano. Semantica delle espressione più complessa, simile a quella dei comandi.
4. Due variabili posso denotare la stessa locazione. Si separa il concetto di "store" da quello di "environment".
5. Identificatori che denotato costanti, funzioni. Si allarga il codominio dell'environment.
6. Espressioni a sinistra dell'assegnazione. Due tipi di valutazione.
7. Comandi che non passano il controllo al comando successivo: Eccezioni, goto, generazione di errore. Continuazioni.

Estensioni del linguaggio IMP

Espressioni (**Exp**) su un insieme di tipi più ricco di valori (si omettono i tipi)

$$e ::= \ldots \mid \textbf{read} \mid e(e)$$

Lettura dall'input, chiamata di funzione.

Comandi (**Com**):

$$c ::= \ldots \mid \textbf{output}\ e \mid e_1 := e_2 \mid e_1(e_2) \mid \textbf{begin}\ d; c\ \textbf{end}$$

genera un valore in uscita, assegnazione più complessa, chiamata di procedure, blocco con dichiarazione

Dichiarazioni (**Dec**)

$$d ::= \textbf{const}\ I = e \mid \textbf{var}\ I = e \mid \textbf{proc}\ I(I_1); c \mid \textbf{fun}\ I(I_1); e \mid d_1; d_2$$

dichiarazione di costanti, variabili, procedure, funzioni, composizione di dichiarazioni.

Per semplificare la presentazione, omettiamo dichiarazioni e controlli di tipo.

Domini per la semantica

Bisogna distinguere tra diversi valori:

- valori denotabili, **Dv**, sono denotati da un identificatore. Tipicamente: interi, reali, booleani, vettori, locazioni **Loc**, procedure, funzioni.
- valori esprimibili, **Ev**, che sono rappresentati dalle espressioni. Un sovrainsieme dei valori denotabili, spesso coincide.
- valori memorizzabili **Sv** che possono essere inseriti in memoria. procedure e funzioni non rientrano in questa categoria.

Domini per gli identificatori

L'environment **Env** definisce l'associazione identificatore–valore

$$\mathbf{Env} = \mathbf{Ide} \to (\mathbf{Dv} + \{\mathbf{unbound}\})$$

Lo store, memoria, definisce l'associazione locazione–valore

$$\mathbf{Store} = \mathbf{Loc} \to (\mathbf{Sv} + \{\mathbf{unused}\})$$

La semantica delle variabili definita in due passi.

Stato: memoria più input:

$$\Sigma = \textbf{Store} \times \textbf{BasicValue}^*$$

Risposta di un programma:

$$\textbf{Answer} = (\{\textbf{error}\} + \{\textbf{stop}\} + \textbf{BasicValue} \times \textbf{Answer})_\perp$$

definizione ricorsiva, il domino soluzione contiene anche stream infininiti di BasicValue.

Nella versione più diretta la semantica di un comando diventa.

$$\mathcal{C}^{dir}[\![\ ]\!] : \mathbf{Com} \to \mathbf{Env} \to \Sigma \to \mathbf{Answer}$$

- Σ, stato
- **Answer** in questo caso $\Sigma + \mathbf{error}$

$$\mathcal{E}^{dir}[\![\ ]\!] : \mathbf{Exp} \to \mathbf{Env} \to \Sigma \to (\mathbf{EVal} \times \mathbf{Answer})$$

Problematica per istruzioni di salto, eccezzioni, errori.

Semantica del linguaggio IMP mediante continuazioni.

$$\mathcal{C}[\![\ ]\!] : \mathit{Com} \to \Sigma \to (\Sigma \to \mathit{Answer}) \to \mathit{Answer}$$

- Σ, stato
- *Answer*, il risultato dell'esecuzione (non necessariamente il nuovo stato).
- $(\Sigma \to \mathit{Answer})$ la continuazione, il comportamento della restante parte del programma.

Semantica per continuazioni: stesse idee del CPS, tecnica di programmazione per linguaggi funzionali.

La funzione:

```
 showTree::  (Show a) => Tree a -> String
showTree (Leaf x) = show x
showTree (Branch l r) = "<" ++ showTree l ++
                        "|" ++ showTree r ++ ">"
```

In CPS diventa:

```
 showsTree ::  (show a) => Tree a -> String -> String
showsTree (Leaf x) s = (show x) :  s
showsTree (Branch l r) s= '<' :  showsTree l
                          ('|' :  showsTree r ('>' :  s))
```

Diversi tipi di continuazioni.
Continuazione di un comando:

$$\mathbf{Cont} = \Sigma \to \mathbf{Answer}$$

rappresenta il comportamento di ciò che segue un comando: riceve uno stato e restituisce una risposta,

Continuazione di un espressione:

$$\mathbf{ECont} = \mathbf{Ev} \to \Sigma \to \mathbf{Answer} \;=\; \mathbf{Ev} \to \mathbf{Cont}$$

La parte di programma che segue un espressione: riceve il valore dell'espressione, lo stato, restituisce una risposta.

Continuazione di una dichiarazione:

$$\mathbf{DCont} = \mathbf{Env} \to \Sigma \to \mathbf{Answer} \;=\; \mathbf{Env} \to \mathbf{Cont}$$

La parte di programma che segue una dichiarazione: riceve l'ambiente definito dalla dichiarazione, lo stato, e restituisce una risposta.

$$\mathbf{Proc} = \mathbf{Cont} \to \mathbf{ECont}$$

Una procedura riceve: una continuazione (il resto del programma dove viene chiamata) e un valore (il parametro); restituisce una continuazione.

$$\mathbf{Fun} = \mathbf{ECont} \to \mathbf{ECont}$$

Una funzione riceve: una E-continuazione (il resto del programma dove viene chiamata) e un valore (il parametro); restituisce una continuazione.

$$\mathcal{C}[\![\ ]\!] : \textbf{Com} \to \textbf{Env} \to \textbf{Cont} \to \textbf{Cont}$$

$$\mathcal{E}[\![\ ]\!] : \textbf{Exp} \to \textbf{Env} \to \textbf{ECont} \to \textbf{Cont}$$

$$\mathcal{R}[\![\ ]\!] : \textbf{Exp} \to \textbf{Env} \to \textbf{ECont} \to \textbf{Cont}$$

valutazione a destra dell'assegnazione

$$\mathcal{D}[\![\ ]\!] : \textbf{Dec} \to \textbf{Env} \to \textbf{DCont} \to \textbf{Cont}$$

Semplificata: senza controllo condizioni di errore.

$$\mathcal{E}[\![\textbf{true}]\!]\rho\,\epsilon = \epsilon\; \mathit{true}$$

$$\mathcal{E}[\![\textbf{read}]\!]\rho\,\epsilon\,\sigma = \epsilon(\mathit{head}(\pi_2\,\sigma))(\langle\pi_1(\sigma), \mathit{tail}(\pi_2\,\sigma)\rangle)$$

$$\mathcal{E}[\![I]\!]\rho\,\epsilon = \epsilon\,(\rho(I))$$

$$\mathcal{R}[\![e]\!]\rho\,\epsilon = \mathcal{E}[\![e]\!]\rho\,(\lambda v.\lambda\sigma.\mathit{cond}\,((\mathit{loc}?\,v),\ \epsilon\,(\sigma\,v)\,\sigma,\ \epsilon\,v\,\sigma))$$

$$\mathcal{E}[\![e_0 + e_1]\!]\rho\epsilon = \mathcal{R}[\![e_0]\!]\rho\,(\lambda v_0.\mathcal{R}[\![e_1]\!]\rho\,(\lambda v_1.\,\epsilon(v_0 + v_1)))$$

$$\mathcal{E}[\![e_1(e_2)]\!]\rho\epsilon = \mathcal{E}[\![e_1]\!]\rho\,(\lambda f.\mathcal{E}[\![e_2]\!]\rho\,(f\,\epsilon))$$

$$\mathcal{C}[\![\textbf{skip}]\!]\rho\,\kappa \;=\; \kappa$$

$$\mathcal{C}[\![\textbf{output } e]\!]\rho\,\kappa \;=\; \mathcal{R}[\![e]\!]\rho\,(\lambda v.\lambda\sigma.\langle v, \kappa\sigma\rangle)$$

$$\mathcal{C}[\![c_0; c_1]\!]\rho\,\kappa \;=\; \mathcal{C}[\![c_0]\!]\rho\,(\mathcal{C}[\![c_1]\!]\rho\,\kappa)$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\rho\,\kappa \;=\; \mathcal{R}[\![b]\!]\rho\,(\lambda v.\mathit{cond}\,(v,\, \mathcal{C}[\![c_0]\!]\rho\,\kappa,\, \mathcal{C}[\![c_1]\!]\rho\,\kappa)$$

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]\rho \;=\; \mathit{Fix}(\lambda F.\lambda\kappa.\mathcal{R}[\![b]\!]\rho\,(\lambda v.\mathit{cond}\,(v,\, \mathcal{C}[\![c]\!]\rho\,(F\kappa),\, \kappa)$$

$$\mathcal{C}[\![e_0 := e_1]\!]\rho\,\kappa \;=\; \mathcal{E}[\![e_0]\!]\rho\,(\lambda l_0.\mathcal{R}[\![e_1]\!]\rho\,(\lambda v_1.\lambda\sigma_1.\kappa([v_1/l_0]\sigma_1)))$$

$$\mathcal{C}[\![e_0(e_1)]\!]\rho\kappa \;=\; \mathcal{E}[\![e_0]\!]\rho\,(\lambda\, p.\, \mathcal{E}[\![e_1]\!]\rho(p\kappa))$$

$$\mathcal{C}[\![\textbf{begin } d\,;\, c \textbf{ end}]\!]\rho\kappa \;=\; \mathcal{D}[\![d]\!]\rho\,(\lambda\,\rho'.\mathcal{C}[\![c]\!]\rho[\rho']\,\kappa)$$

$$\mathcal{D}[\![\textbf{const } I = e]\!]\rho\,\delta \;=\; \mathcal{R}[\![e]\!]\rho\,(\lambda v.\delta([v/I]))$$

$$\mathcal{D}[\![\textbf{var } I = e]\!]\rho\,\delta \;=\; \mathcal{R}[\![e]\!]\rho\,(\lambda v.\lambda\sigma.\delta([new(\sigma)/I])(\sigma[v/new(\sigma)]))$$

$$\mathcal{D}[\![\textbf{proc } I(I_1); c]\!]\rho\,\delta \;=\; \delta([\lambda\kappa.\lambda v.\mathcal{C}[\![c]\!](\rho[v/I_1])\kappa\,/I])$$

$$\mathcal{D}[\![\textbf{fun } I(I_1); e]\!]\rho\,\delta \;=\; \delta([\lambda\epsilon.\lambda v.\mathcal{E}[\![e]\!](\rho[v/I_1])\epsilon\,/I])$$

$$\mathcal{D}[\![d_0; d_1]\!]\rho\,\delta \;=\; \mathcal{D}[\![d_0]\!]\rho(\lambda\rho_0.\mathcal{D}[\![d_1]\!](\rho[\rho_0])(\lambda\rho_1.\delta(\rho_o[\rho_1])))$$

Permettono l'uscita da comandi, una versione strutturata del costrutto di salto.
Si aggiunge alla sintassi dei comandi, la dichiarazione dei gestori eccezioni (exception handler) e la chiamata di eccezione (exception raising):

$$c ::= \ldots \mid \textbf{trap}\ c\ \ l_1 : c_1, l_2 : c_2, \ldots, l_n : c_n\ \textbf{end} \mid \textbf{escapeto}\ l \mid \ldots$$

Si associa ad una eccezione, l, una continuazione $\textbf{Ev} := \ldots + \textbf{Cont}$

$$\mathcal{C}[\![\textbf{trap}\ c\ \ l_1 : c_1, \ldots, l_n : c_n\ \textbf{end}]\!]\rho\kappa$$
$$= \mathcal{C}[\![c]\!](\rho[\mathcal{C}[\![c_1]\!]\rho\kappa/l_1]\ldots[\mathcal{C}[\![c_n]\!]\rho\kappa/l_n])\kappa$$

La chiamata di eccezione passa il controllo al gestore:

$$\mathcal{C}[\![\textbf{escapeto}\ l]\!]\rho\kappa = \mathcal{E}[\![l]\!]\rho(\lambda\kappa' \,.\, \kappa')$$

Etichette e istruzioni di salto

$$c ::= \ldots \mid l : c \mid \textbf{goto } l \mid \ldots$$

Etichette sono identificatori, bisogna definire una funzione semantica che costruisce l'environment per le etichette.

$$\mathcal{J} : Com \to Env \to Cont \to Env$$

Si associa ad etichette continuazioni.

$$\mathcal{J}[\![e_0 := e_1]\!]\rho\kappa = ()$$

$$\mathcal{J}[\![l : c]\!]\rho\kappa = \mathcal{J}[\![c]\!]\rho\kappa[\mathcal{C}[\![c]\!]\rho\kappa/l]$$

$$\mathcal{J}[\![c_0; c_1]\!]\rho\kappa = \mathcal{J}[\![c_0]\!]\rho(\mathcal{C}[\![c_1]\!]\rho\,\kappa)[\mathcal{J}[\![c_1]\!]\rho\,\kappa]$$

$$\mathcal{C}[\![\textbf{begin } d\,;\, c \textbf{ end}]\!]\rho\kappa = \mathcal{D}[\![d]\!]\rho\,(\lambda\,\rho'.\mathcal{C}[\![c]\!](\rho[\rho'][\rho''])\,\kappa)$$

dove

$$\rho'' = Fix.\lambda\rho_0.\mathcal{J}[\![c]\!](\rho[\rho'][\rho_0])\kappa$$

Diversi aspetti:

- numero di parametri, procedure con tipi espliciti.
  più domini per funzioni, procedure.

$$\mathbf{Proc_n} = \mathbf{Cont} \to \mathbf{Ev}^n \to \mathbf{Cont}$$

- procedure ricorsive:
  environment definito tramite punto fisso.

$$\mathcal{D}[\![\mathbf{proc}I(I_1); C]\!]\rho\upsilon = \upsilon(p/I)$$
$$\text{dove} \quad p = \lambda\kappa.e.\mathcal{C}[\![C]\!]\rho[e/I_1]\kappa$$

$$\mathcal{D}[\![\mathbf{rec\ proc}I(I_1); C]\!]\rho\upsilon = \upsilon(p/I)$$
$$\text{dove} \quad p = Fix\lambda p_0.\lambda\kappa.e.\mathcal{C}[\![C]\!]\rho[p_0/I, e/I_1]\kappa$$

Le definizioni precedenti usano legame statico: procedure valutate nell'ambiente della dichiarazione.
Alcuni linguaggi usano legame dinamico: l'ambiente di valutazione è quello della chiamata.
Diversi domini semantici:

$$\mathbf{Proc} = \mathbf{Env} \to \mathbf{Cont} \to \mathbf{Ev} \to \mathbf{Cont}$$

$$\mathcal{D}[\![\mathbf{proc}\, I(I_1);\, C]\!]\rho\upsilon = \upsilon((\lambda\rho'.\kappa.e.\mathcal{C}[\![C]\!]\rho'[e/I_1]\kappa)/I)$$

$$\mathcal{C}[\![e_1(e_2)]\!]\rho\kappa = \mathcal{E}[\![e_1]\!]\rho\ \lambda p.\mathcal{E}[\![e_2]\!]\rho(p\ \rho\ \kappa)$$

Sono possibili anche metodi misti, si combina l'ambiente della dichiarazione con quello della chiamata:

$$\mathcal{D}[\![\mathbf{proc}\, I(I_1);\, C]\!]\rho\upsilon = \upsilon((\lambda\rho'.\kappa.e.\mathcal{C}[\![C]\!](mix(\rho)(\rho'))[e/I_1]\kappa)/I)$$

Vantaggi:

- procedure sono automaticamente ricorsive;
- approccio top down, posso iniziare a scrivere il programma principale senza definire tutte le procedure;
- più facilmente modificabili.

Svantaggi:

- impossibile il type checking statico: non so a cosa verranno legati le variabili libere di una procedura;
- comportamenti imprevedibili;
- si perde la differenze tra parametri e variabili libere;

Passaggio dei parametri, call by reference

$$\mathcal{D}[\![\textbf{proc}\, I(I_1); C]\!]\rho\upsilon = \upsilon(p/I)$$
$$\text{dove} \quad p = \lambda\kappa.\lambda l.(\mathit{cond}(\mathit{loc}?l)(\mathcal{C}[\![C]\!]\rho[l/I_1]\kappa), \mathit{error}))$$

Controllo che il parametro formale sia una locazione.
Sull'argomento viene fatta la valtazione sinistra.

Passaggio dei parametri, Call by Value

$$\mathcal{D}[\![\textbf{proc}\, I(I_1); C]\!]\rho\upsilon = \upsilon(p/I)$$
$$\text{dove} \quad p = \lambda\kappa.\lambda v.\lambda\sigma.\mathcal{C}[\![C]\!]\rho[new(\sigma)/I_1]\kappa(\sigma)[v/new(\sigma)]$$

Quindi viene creata una locazione, il valore viene memorizzato in essa, il parametro formale viene legato alla locazione.

$$\mathcal{C}[\![e_1(e_2)]\!]\rho\kappa = \mathcal{E}[\![e_1]\!]\rho(\lambda p.\mathcal{R}[\![e_2]\!]\rho(p\kappa))$$

Il parametro viene valutato, se una locazione si restituisce il valore.

$new(\sigma)$ una locazione libera in σ.
Per semplicità, σ denota lo store, si omette la componente input.

Valutazione degli argomenti

Non necessariamente l'argomento viene al momento della chiamata.

Call by closure (by name): argomento valutato durante l'esecuzione della procedura.
Al parametro non associo un valore, ma una chiusura:
Closure = **ECont** $\rightarrow$ **Cont**,
ad una procedura associo il tipo:

$$\textbf{Proc} = \textbf{Cont} \rightarrow (\textbf{ECont} \rightarrow \textbf{Cont}) \rightarrow \textbf{Cont}$$

$$\mathcal{C}[\![e_1(e_2)]\!]\rho\kappa = \mathcal{E}[\![e_1]\!]\rho(\lambda p.p\kappa(\mathcal{E}[\![e_2]\!]\rho))$$

$$\mathcal{E}[\![I]\!]\rho\epsilon = \mathit{cond}(\mathit{isClosure}(\rho(I)), \rho(I)(\epsilon), \epsilon(\rho(I))$$

Altre possibilità, call by text, call by denotation.

Linguaggi e semantica della concorrenza

Due paradigmi di comunicazione tra processi concorrenti:

- memoria condivisa: multiprocessori;
- scambio di messaggi: multicomputer, sistemi distribuiti.

Semantica:

- non determinismo: programmi concorrenti sono intrinsecamente non-deterministici;
- un programma non può essere descritto come una funzione da ingresso a uscita.

- semantica operazionale:
  - memoria condivisa: small-step reduction, scambio di messaggi: label transition system
- semantica denotazione: powerdomains, programmi come alberi,

Dijkstra guarded commands: scelta non deterministica

$$c := \textbf{skip} \mid \textbf{abort} \mid X := a \mid c_0; c_1 \mid \textbf{if } gc \textbf{ fi} \mid \textbf{do } gc \textbf{ od}$$

Guarded commands: lista di programmi con guardia,

$$gc ::= b \to c \mid gc \,[\!]\, gc$$

Dove b è un espressione booleana.
L'esecuzione di una lista di guarded commands comporta la selezione di una guardia vera e l'esecuzione del relativo commando.
L'esecuzione fallisce se nessuna guardia è vera.

Hoare, CSP Communicating Sequential Processes

Dijkstra guarded command con:

- parallelismo $\|$: $\quad c := c_1 \parallel c_2 \mid \ldots$
- comunicazioni (sincrone) su canali: comandi in parallelo comunicano attraverso canali, non posso condividere variabili

$$c := \alpha!a \mid \alpha?X \mid \ldots$$

Restrizione di canali: alcuni canali usabili solo all'interno del comando:

$$c := c_1 \backslash \alpha \mid \ldots$$

Guarded commands con comunicazione: il relativo comando eseguito solo se la comunicazione

$$gc ::= b \wedge \alpha!a \to c \mid b \wedge \alpha?X \to c \mid \ldots$$

Nel CSP originari le comunicazioni avvengono specificando il nome di un processo, e non di un canale.

Robin Milner, CCS Calculus of Concurrent Systems

Versione ristretta del CSP, viene lasciato l'essenziale per studiare la concorrenza.
Si eliminano assegnazione, ristretti i comandi test e cicli.

$p :=$	$0 \mid$		processo vuoto
	$b \to p \mid$		guardia booleana
	$\alpha!a \to p \mid$	$\alpha?x \to p \mid$	output - input
	$\tau \to p \mid$		azione vuota
	$p_0 + p_1 \mid$		scelta
	$p_0 \parallel p_1 \mid$		parallelismo
	$p \backslash \alpha \mid$		restrizione
	$p[f] \mid$		relabelling
	$P(a_1, \ldots, a_n)$		identificatori di processo

Non si vuol definire un linguaggi di programmazione, ma un sistema per la descrizione di sistemi concorrenti.

Semantica Operazionale del CCS

Label transition system, due tipi di transazioni:

- $\rightarrow$ ($\xrightarrow{\tau}$) descrive un passo di computazione,
- $\xrightarrow{\alpha!n}$ $\xrightarrow{\alpha?n}$, descrive una potenziale interazione con l'esterno.

Motivazioni per il secondo tipo di transazioni:

- Presentazione più sintetica del transition system.
- Semantica composizionale. Descrivo il singolo processo.

Regole di semantica operazionale:

Esercizi: definire un processo che simuli il comportamento di una variabile e un processo che simuli una pila.

Elimino dal CCS numeri naturali e variabili

$p :=$	$\alpha \to p \mid \overline{\alpha} \to p \mid$	comunicazione
	$\tau \to p \mid$	azione vuota
	$\Sigma_i p_i \mid$	scelta potenzialmente infinita
	$p_0 \parallel p_1 \mid$	parallelismo
	$p \backslash \alpha \mid$	restrizione
	$p[f] \mid$	relabelling
	P	identificatori di processo

Si perde la distinzione tra ingresso uscita, resta un meccanismo di sincronizzazione, α, $\overline{\alpha}$.

Sistema più astratto.
Numeri naturali e guardie simulabili, in parte tramite canali e la somma infinita (rappresentazione estensionale)

Quando due processi sono equivalenti?
Quando i loro alberi di derivazione coincidono a meno di duplicazioni.

Definizione

- Una bisimulazione forte (strong bisimulation) è una relazione *simmetrica* R tale che:
  se $p\,R\,q \ \wedge \ p \xrightarrow{\lambda} p'$
  allora esiste q' t.c. $q \xrightarrow{\lambda} q' \ \wedge \ p'\,R\,q'$
- $p \sim q$, p è fortemente bisimile a q se esiste un simulazione R t.c. $p\,R\,q$

Esercizio: mostrare che $\sim$ è una bisimulazione (la massima).

È una buona nozione di equivalenza?

- Aspetti positivi. È una congruenza. È possibile ragionare in maniera composizionale.
- Aspetti negativi. Si osserva la transazione muta $\xrightarrow{\tau}$, si distingue più del strettamente necessario.

Definizione

- Una bisimulazione debole (weak bisimulation) è una relazione *simmetrica* R tale che:
  se $p\,R\,q \ \wedge \ p \xrightarrow{a} p'$
  allora esiste q', q'', q''' t.c. $q \xrightarrow{\tau}{}^{*} q' \xrightarrow{a} q'' \xrightarrow{\tau}{}^{*} q''' \ \wedge \ p'\,R\,q'''$.
  E se $p\,R\,q \ \wedge \ p \xrightarrow{\tau} p'$
  allora esiste q' t.c. $q \xrightarrow{\tau}{}^{*} q' \ \wedge \ p'\,R\,q'$.
- $p \sim q$, p è debolmente bisimile a q se esiste un simulazione R t.c. $p\,R\,q$

- Si identificano processi che sono naturalmente equivalenti.
- La bisimulazione debole non è una congruenza.
  Controesempio $\tau.a$, a, e $\tau.a + b$, $a + b$

Trace equivalence

Due processi sono equivalenti se generano le stesse tracce (alberi di derivazione con gli stessi cammini)

Una logica modale per transazioni etichettate. Proposizioni:

$$A := T \mid A_o \wedge A_1 \mid \neg A \mid [\lambda]A$$

Connettivi definibili attraverso la negazione:

$$A := F \mid A_o \vee A_1 \mid \langle\lambda\rangle A$$

Soddisfacibilità di una formula;

$$p \models A$$

Definiamo due processi logicamente equivalenti se soddisfano lo stesso insieme di proposizioni.
Mostrare che due processi fortemente bisimili sono logicamente equivalenti.
É vero il contrario?

Per aumentare il poter espressivo del linguaggio, aggiungo alla logica di Hennessy-Milner

- costanti S per insieme di processi (insiemi eventualmente definibili in base alla sintassi),
- variabili di insieme e definizioni medianti (minimo) punto fisso,
- operatori modali generici,

$$A := S \mid X \mid \mu X.A \mid [.]A \mid \ldots$$

Posso definire il massimo punto fisso come:

$$\nu X.A = \neg(\mu X.\neg(A[\neg X/X]))$$

Connettivi definibili:
Possibly $B = \mu X.(B \vee \langle . \rangle X)$
Eventually $B = \mu X.(B \vee (\langle . \rangle T \wedge [.]X))$
Always $B = \nu X.(B \wedge \langle . \rangle X)$