

Teoria della Concorrenza modulo di semantica

Pietro Di Gianantonio, Fabio Alessi

corso di laurea in informatica

Modulo di 6 crediti (48 ore),

viene presentata la Semantica dei linguaggi di programmazione, propedeutico a

- la restante parte del corso di Teoria della Concorrenza,
- il corso di Interpretazione Astratta (Comini),
- una qualsiasi trattazione formale e rigorosa dei programmi.

negli scorsi anni esisteva un corso separato di Semantica dei linguaggi di programmazione.

- Presenteremo i diversi metodi (3) utilizzati per descrivere in maniera formale i linguaggi di programmazione.
- Cominceremo con il considerare un linguaggio di programmazione semplicissimo, in seguito arricchito con nuovi costrutti.

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 le 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)
 - fondamentale nella costruzione dei compilatori,
- ragionare sui singoli 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, in una semplice macchina formale. Parallelo con le macchine di Turing.
- **Semantica Operazionale Strutturata (SOS)**. La macchina formale sostituita da regole, sintattiche, di riscrittura.
- **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 Assiomatrica**. 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.

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).

Semantica denotazionale

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 alle altre semantiche.

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**): b , 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**):

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

Comandi (**Com**):

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

Sintassi astratta e sintassi concreta

IMP

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

Manca:

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

Semantica operativa per IMP

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

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

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

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

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

Alle espressioni base si associano assiomi:

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

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

IMP: regole SOS

Alle espressioni composte regole di derivazione:

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

...

- 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 operazioni aritmetiche. Si astrae dal problema di eseguire le operazioni..

Espressione booleana: regole SOS

Assiomi ...

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow n}{\langle a_0 = a_1, \sigma \rangle \Rightarrow \mathbf{true}}$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 = a_1, \sigma \rangle \Rightarrow \mathbf{false}} \quad n \neq m$$

...

$$\frac{\langle b_0, \sigma \rangle \Rightarrow t_0 \quad \langle b_1, \sigma \rangle \Rightarrow t_1}{\langle b_0 \mathbf{and} b_1, \sigma \rangle \Rightarrow \mathbf{t}}$$

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

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 $\llbracket n : 0 \rrbracket = 2 \times \llbracket n \rrbracket$
e $\llbracket n : 1 \rrbracket = 2 \times \llbracket n \rrbracket + 1$

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 $\sigma[m/X]$

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

In un approccio completamente operativo lo stato dovrebbe essere un oggetto sintattico:

grammatica per definire gli stati,

insieme di regole che ne definiscono il comportamento.

Comandi, regole

$$\langle \text{skip}, \sigma \rangle \Rightarrow \sigma$$

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

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

$$\frac{\langle b, \sigma \rangle \Rightarrow \text{true} \quad \langle c_0, \sigma \rangle \Rightarrow \sigma'}{\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \Rightarrow \sigma'}$$

...

$$\frac{\langle b, \sigma \rangle \Rightarrow \text{true} \quad \langle c, \sigma \rangle \Rightarrow \sigma'' \quad \langle \text{while } b \text{ do } c, \sigma'' \rangle \Rightarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \Rightarrow \sigma'}$$

Equivalenze

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 $\text{Loc} \rightarrow \mathbf{N}$

Dimostrare che posto:

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

si ha:

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

e

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

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

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

Big-step, small-step SOS

Formulazione alternativa: descrive un passo di computazione.

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

Nuove regole per il while

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

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.

Induzione

In matematica e informatica molte definizioni induttive:

- numeri naturali,
- grammatiche
- derivazioni, dimostrazioni

Insiemi di costruttori:

Sulle strutture induttive:

- definizione per ricorsione,
- dimostrazioni per induzione.

Rafforzamenti, estensioni:

- induzione generalizzata,
- insiemi ben fondati.

Semantica delle espressioni (aritmetiche, booleane)

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

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

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

$$\mathcal{A}[a_0 + a_1](\sigma) = (\mathcal{A}[a_0](\sigma)) + (\mathcal{A}[a_1](\sigma))$$

...

$$\mathcal{B}[a_0 \leq a_1](\sigma) = (\mathcal{A}[a_0](\sigma)) \leq (\mathcal{A}[a_1](\sigma))$$

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

Definiti,

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

a ogni categoria sintattica si associa una funzione di interpretazione:

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

Relazioni

Definizione

- Una relazione tra X e Y è caratterizzata da un sottinsieme di $X \times Y$.
- Una funzione parziale $f : X \rightarrow Y$ 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.

Semantica dei comandi

Comandi rappresentano funzioni parziali,
Conviene considerare le funzioni parziali come relazioni.
(funzioni e funzioni parziali possono essere viste come casi particolari di relazioni)

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

$$\mathcal{C}[X := a] = \{(\sigma, \sigma[n/X]) \mid \text{if } \mathcal{A}[a](\sigma) = n\}$$

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

$$\mathcal{C}[\text{if } b \text{ then } c_0 \text{ else } c_1] = \{(\sigma, \sigma') \in \mathcal{C}[c_0] \mid \mathcal{B}[b](\sigma) = \text{true}\} \cup \{(\sigma, \sigma') \in \mathcal{C}[c_1] \mid \mathcal{B}[b](\sigma) = \text{false}\}$$

Il caso difficile: il costruttore while

$$\begin{aligned} \mathcal{C}[\text{while } b \text{ do } c] &= \mathcal{C}[\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip}] = \\ &= \{(\sigma, \sigma) \mid \mathcal{B}[b](\sigma) = \text{false}\} \cup \\ &= \{(\sigma, \sigma') \in (\mathcal{C}[\text{while } b \text{ do } c] \circ \mathcal{C}[c]) \mid \mathcal{B}[b](\sigma) = \text{true}\} \end{aligned}$$

Definizione ricorsiva.

Esistenza delle soluzioni:

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

$$\Gamma(R) = \{(\sigma, \sigma) \mid \mathcal{B}[b](\sigma) = \text{false}\} \cup \{(\sigma, \sigma') \in (R \circ \mathcal{C}[c]) \mid \mathcal{B}[b](\sigma) = \text{true}\}$$

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

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

- Cerco il **minimo** punto fisso per Γ .

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)

Strutture ordinate

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 sottinsieme 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 $\sqcup\{p, q\}$ 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 sottinsieme ha estremo superiore.

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

Applicazione alla semantica

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

$$\Gamma(R) = \{(\sigma, \sigma) \mid \mathcal{B}[[b]](\sigma) = false\} \cup \{(\sigma, \sigma') \in (R \circ \mathcal{C}[[c]]) \mid \mathcal{B}[[b]](\sigma) = true\}$$

è monotono,

- il teorema di Knaster-Tarski prova l'esistenza un punto fisso per l'operatore Γ (soluzione delle definizioni ricorsive della semantica).

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

Definizione

Una funzione $f : P \rightarrow Q$ tra ordini complete 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 $\sqcap\{p \mid f(p) \sqsubseteq p\}$ è un punto fisso per f (ossia $\sqcap\{p \mid f(p) \sqsubseteq p\} = f(\sqcap\{p \mid f(p) \sqsubseteq p\})$)

Strutture per la semantica denotazionale

- 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 : $\Sigma \rightarrow \Sigma$.
- 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) $f \sqsubseteq g$
 questo spazio non è un reticolo.
- È necessario usare un secondo teorema di punto fisso.

Ordini parziali completi

Definizione

- In un ordine parziale P , una **catena** è una sequenza di elementi $p_1 \sqsubseteq p_2 \sqsubseteq p_3, \dots$, 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 \perp .

CPO sono le strutture tipiche i cui interpretare programmi.

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

Definizione (Continuità)

Una funzione $f : D \rightarrow E$ tra cpo è **continua** se preserva l'estremo superiore delle catene.

Per ogni catena p_1, p_2, p_3, \dots , $\bigsqcup_{i \in N} f(p_i) = f(\bigsqcup_{i \in N} p_i)$.

Nuovo teorema di punto fisso

Teorema

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

Proof.

Il minimo punto fisso è l'estremo superiore della catena

$\perp, f(\perp), f(f(\perp)), f^3(\perp), \dots$

Completare per esercizio. □

Definizione per approssimazioni del while

Applicando il teorema precedente, la funzione

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

è il limite della seguente catena di funzioni parziali:

$$\begin{aligned} &\mathcal{C}[\perp] \\ &\mathcal{C}[\text{if } b \text{ then } (c; \perp)] \\ &\mathcal{C}[\text{if } b \text{ then } (c; \text{if } b \text{ then } (c; \perp))] \\ &\mathcal{C}[\text{if } b \text{ then } (c; \text{if } b \text{ then } (c; \text{if } b \text{ then } (c; \perp)))] \\ &\dots \end{aligned}$$

dove indichiamo con \perp 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.

Un categoria per la semantica denotazionale

Astraendo dall'esempio precedente: si interpretano i linguaggi di programmazione 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 ,
- $B_{\perp} = \{true, false, \perp\}$,
- $O = \{\perp, \top\}$ con $\perp \sqsubseteq \top$, il cpo con due elementi.
- $N \rightarrow N_{\perp}$ con l'ordine puntuale: $f \sqsubseteq g$ sse per ogni n , $f(n) \sqsubseteq g(n)$, isomorfo a $N \rightarrow N$ (con l'ordine visto in precedenza) si internalizza la divergenza: $f(n) = \perp$ indica che $f(n)$ non termina, si evitano le funzioni parziali
- streams: stringhe parziali, arbitrariamente lunghe.

Monotonia e continuità

Significato intuitivo.

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

La sua semantica $\llbracket F \rrbracket : (N \rightarrow N_{\perp}) \rightarrow N_{\perp}$ è una funzione:

- **monotona**, perciò se $F(f) \Rightarrow 3$ allora $F(g) \rightarrow 3$ 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 $g \sqsubseteq \llbracket f \rrbracket$ e $\llbracket F \rrbracket(g) = 3$
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:

$$\lambda x . \sin(x) + \cos(x), \text{ oppure } \lambda x \in R . \sin(x) + \cos(x)$$

$$\text{oppure } f = \lambda x . \sin(x) + \cos(x)$$

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 incofrontabili: 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 $\vee (d_1, d_2 \in D \wedge d_1 \sqsubseteq d_2)$

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

Funzioni associate:

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

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

Prodotto

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

Definiscono un isomorfismo tra $(C \rightarrow D) \times (C \rightarrow E)$ e $C \rightarrow (D \times E)$

Dato da ...

Esercizi e proprietà

- Costruire $O \times O (= O^2)$, O^3 . A quali altri CPO sono isomorfi?
- Costruire T_{\perp}^2
- 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,

Proposizione

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

Definizione

$[D \rightarrow E]$ 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 $[D \rightarrow E]$ è un cpo.

Funzioni associate.

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

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

- Mostrare che app è continua.
- Mostrare che se f è continua allora $curry(f)$ è continua.
- Mostrare che l'operatore di punto fisso $Y : [D \rightarrow D] \rightarrow D$ è continuo.
- Mostrare che $[T \rightarrow D]$ è isomorfo a D^2 .

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

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

- Definire, tramite funzioni e costruttori standard, la semantica dell'**if then else**
- Definire, tramite funzioni e costruttori standard, la semantica delle funzioni booleane.

Metalinguaggio

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

- variabili con tipo un dominio: $x_1 : D_1, \dots, x_i : D_i$
- costanti: $true, false, -1, 0, 1, \dots$
- funzioni base: $[-], \pi_i, app, in_i, fix$
- costruttori: $(-)^*, \langle -, - \rangle, curry(-), [-, -],$
- applicazione e lambda astrazione.

Esempi:

Metalinguaggio

Proposizione

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

Proof.

Per induzione sulla struttura dell'espressione e , si definisce il significato di e e da qui la correttezza. \square

Linguaggi funzionali

Semantica (operazionale e denotazionale) di un semplice linguaggio funzionali

con due differenti meccanismi di valutazione

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

Tipi:

$\tau ::= \mathbf{int} \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2$

Espressioni

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

Non tutte le espressioni hanno senso, esempio $(1\ 3)$, $((\lambda x.x + 1)(2, 3))$

Controllo di tipo:

- determina le espressioni corrette,
- si deriva $t : \tau$
- per induzione sulla struttura del termine ogni costrutto sintattico ha una regola del tipo:

$$\frac{x : \tau_1 \quad t_1 : \tau_1 \quad t_2 : \tau_2}{\mathbf{let}\ x \leftarrow t_1 \mathbf{in}\ t_2 : \tau_2}$$

- a ogni variabile è associato un unico tipo, conseguenza: ogni espressione ha tipo unico,
- nessun polimorfismo: semantica più semplice,

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

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

I linguaggi funzionali si basano sulla chiamata di funzioni; non è semplice definire cosa sia un passo di computazione.

Call-by-value, Big-step reduction

Per ogni tipo, un insieme dei **valori**.

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

Per definizione valori sono termini chiusi, questo semplifica le definizioni della semantica operativa.

Regole di riduzione

Definite sulla struttura del termine (chiuso).

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

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

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

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

Regole di riduzione

$$\frac{t_1 \Rightarrow \lambda x. t'_1 \quad t_2 \Rightarrow v_2 \quad t'_1[v_2/x] \Rightarrow c}{(t_1 t_2) \Rightarrow c}$$

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

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

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: $(\mathbf{rec} \ x^{\mathbf{N} \rightarrow \mathbf{N}}. \lambda y. (x - 1) \ 2)$
- 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.

- Domini per interpretare valori.
- Domini per interpretare computazioni.

Domini per valori: per induzione sul struttura del tipo.

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

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

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

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

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

$$\llbracket t \rrbracket : \mathbf{Env} \rightarrow (V_{\tau})_{\perp}$$

Definizioni per induzione sul termine

$$\begin{aligned}\llbracket x \rrbracket &= \lambda \rho. [\rho(x)] \\ \llbracket n \rrbracket &= \lambda \rho. [n] \\ \llbracket t_1 \text{ op } t_2 \rrbracket &= \lambda \rho. \llbracket t_1 \rrbracket \rho \text{ op } \llbracket t_2 \rrbracket \rho \\ \llbracket (t_1, t_2) \rrbracket &= \lambda \rho. \text{let } v_1 \leftarrow \llbracket t_1 \rrbracket \rho. \text{let } v_2 \leftarrow \llbracket t_2 \rrbracket \rho. [(v_1, v_2)] \\ \llbracket (\text{fst } t) \rrbracket &= \lambda \rho. \text{let } v \leftarrow \llbracket t \rrbracket \rho. [\pi_1(v)] \\ \llbracket \lambda x. t \rrbracket &= \lambda \rho. [\lambda v : V_\sigma. \llbracket t \rrbracket (\rho[v/x])] \\ \llbracket (t_1 t_2) \rrbracket &= \lambda \rho. \text{let } v_1 \leftarrow \llbracket t_1 \rrbracket \rho. \text{let } v_2 \leftarrow \llbracket t_2 \rrbracket \rho. v_1(v_2) \\ \llbracket \text{rec } x. \lambda y. t \rrbracket &= \lambda \rho. \text{Fix}(\lambda v_1 : V_{\sigma \rightarrow \tau}. \lambda v_2 : V_\sigma. \llbracket t \rrbracket \rho[v_1/x, v_2/y])\end{aligned}$$

Proprietà della semantica denotazionale

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

Confronto tra semantiche

Correttezza della semantica operativa

Proposizione

Per ogni termine t , e valore c ,

$$t \Rightarrow c \text{ implica } \llbracket t \rrbracket = \llbracket c \rrbracket$$

Si dimostra per induzione sulla derivazione di $t \Rightarrow c$.

Le regole della semantica operativa rispettano quella denotazionale.

L'implicazione opposta non è valida,

perché esistono valori diversi con la stessa semantica denotazionale.

$$\llbracket c \rrbracket = \llbracket c' \rrbracket \text{ ma } c \not\Rightarrow c'$$

Confronto tra semantiche

Vale una relazione più debole:

Proposizione

Per ogni termine t , e valore c ,

$$\llbracket t \rrbracket = \llbracket c \rrbracket \text{ implica l'esistenza di } c' \text{ tale che } t \Rightarrow c'.$$

Se $\llbracket t \rrbracket$ è diversa da \perp allora le computazioni su t terminano.

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

Corollario

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

$$t \Rightarrow n \text{ sse } \llbracket t \rrbracket = \llbracket n \rrbracket$$

Differenti insiemi di **valori**, sui vari tipi.

- **int** (tipi ground), le costanti numeriche $\dots - 1, 0, 1, 2 \dots$
- $\tau_1 * \tau_2$, 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.
- $\tau_1 \rightarrow \tau_2$, le λ -astrazioni $\lambda x.t$, con t chiuso non necessariamente un valore.

Differenze rispetto alla semantica call-by-name:

$$\frac{t_1 \Rightarrow \lambda x.t'_1 \quad t'_1[t_2/x] \Rightarrow c}{(t_1 t_2) \Rightarrow c}$$

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

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

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:

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

Domini per le computazioni.

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

Appunto: nei linguaggi "call-by-name", non è strettamente indispensabile separare 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

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

che rispettano i tipi.

Definizioni per induzione sul termine

Insiemistica e categoriale

$$\llbracket x \rrbracket = \lambda \rho. \rho(x) = \pi_x$$

$$\llbracket n \rrbracket = \lambda \rho. \llbracket n \rrbracket = \llbracket _ \rrbracket \circ n \circ 1$$

$$\llbracket t_1 \text{ op } t_2 \rrbracket = \lambda \rho. \llbracket t_1 \rrbracket \rho \text{ op } \llbracket t_2 \rrbracket \rho = \text{op} \circ \langle \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket \rangle$$

$$\llbracket (t_1, t_2) \rrbracket = \lambda \rho. \llbracket (t_1, t_2) \rrbracket \rho = \llbracket _ \rrbracket \circ \langle \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket \rangle$$

$$\llbracket (\text{fst } t) \rrbracket = \lambda \rho. \text{let } v \Leftarrow \llbracket t \rrbracket \rho. \pi_1(v) \quad \llbracket \text{fst} \rrbracket = \llbracket ((\pi_1)^*) \rrbracket$$

$$\llbracket \lambda x. t \rrbracket = \lambda \rho. \llbracket \lambda v : (V_\sigma)_\perp. \llbracket t \rrbracket (\rho[v/x]) \rrbracket = \llbracket _ \rrbracket \circ \text{curry}(\llbracket t \rrbracket \circ \prod_{y \in \text{Var}} (in_y \circ f_y))$$

$$\llbracket (t_1 t_2) \rrbracket = \lambda \rho. \text{let } v \Leftarrow \llbracket t_1 \rrbracket \rho. v(\llbracket t_2 \rrbracket \rho) = \text{app} \circ \langle (id)^* \circ \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket \rangle$$

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

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



Confronto tra semantiche: correttezza e adeguatezza

Proposizione (Correttezza)

Per ogni termine t , e valore c ,

$$t \Rightarrow c \text{ implica } \llbracket t \rrbracket = \llbracket c \rrbracket$$

Proposizione (Adeguatezza)

Per ogni termine t , e valore c ,

$$\llbracket t \rrbracket = \llbracket c \rrbracket \text{ implica l'esistenza di } c' \text{ tale che } t \Rightarrow c'.$$

Corollario

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

$$t \Rightarrow n \text{ sse } \llbracket t \rrbracket = \llbracket n \rrbracket$$



Proprietà della semantica denotazionale

- substitution lemma

Se $\llbracket s \rrbracket \rho = v$ allora $\llbracket t[s/x] \rrbracket \rho = \llbracket t \rrbracket \rho[v/x]$

Si dimostra per induzione sulla struttura del termine t

- per ogni valore c , $\llbracket c \rrbracket \neq \perp$



Uguaglianza osservazionale e full-abstraction

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

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

Per il teorema di adeguatezza:

$$\llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket \text{ implica } t_1 \sim t_2$$

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



IMP linguaggio ridotto semantica denotazionale semplice, banale.

Linguaggi più complessi portano a semantiche più complesse.

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

- 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 espressioni più complessa, simile a quella dei comandi.
- 4 Due variabili possono denotare la stessa locazione. Si separi il concetto di "store" da quello di "environment".
- 5 Identificatori che denotano 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.

Continuazioni

Semantica del linguaggio IMP mediante continuazioni.

$$\mathcal{C}[\] : Com \rightarrow \Sigma \rightarrow (\Sigma \rightarrow Answer) \rightarrow Answer$$

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

CPS: Continuation Passing Style

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))
```

Estensioni del linguaggio IMP

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

$$e ::= \dots \mid \mathbf{read} \mid e(e)$$

Lettura dall'input, chiamata di funzione.

Comandi (**Com**):

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

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

Dichiarazioni (**Dec**)

$$d ::= \mathbf{const} \ l = e \mid \mathbf{var} \ l = e \mid \mathbf{proc} \ l(l_1); c \mid \mathbf{fun} \ l(l_1); e \mid d_1; d_2$$

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

$$\mathbf{Env} = \mathbf{Ide} \rightarrow (\mathbf{Ev} + \{\mathbf{unbound}\})$$

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

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

La semantica delle variabili definita in due passi.



Domini per l'Input-Output

Stato: memoria più input:

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

Risposta di un programma:

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

definizione ricorsiva, la soluzione massima considera programmi non terminanti.



Continuazioni

Diversi tipi di continuazioni.

Continuazione di un comando:

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

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

Continuazione di un'espressione:

$$\mathbf{ECont} = \mathbf{Ev} \rightarrow \Sigma \rightarrow \mathbf{Answer} = \mathbf{Ev} \rightarrow \mathbf{Cont}$$

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

Continuazione di una dichiarazione:

$$\mathbf{DCont} = \mathbf{Env} \rightarrow \Sigma \rightarrow \mathbf{Answer} = \mathbf{Env} \rightarrow \mathbf{Cont}$$

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

Domini per procedure e funzioni

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

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

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

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

Funzioni semantiche

$$\mathcal{C}[\] : \mathbf{Com} \rightarrow \mathbf{Env} \rightarrow \mathbf{Cont} \rightarrow \mathbf{Cont}$$

$$\mathcal{E}[\] : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{ECont} \rightarrow \mathbf{Cont}$$

$$\mathcal{R}[\] : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{ECont} \rightarrow \mathbf{Cont}$$

valutazione a destra dell'assegnazione

$$\mathcal{D}[\] : \mathbf{Dec} \rightarrow \mathbf{Env} \rightarrow \mathbf{DCont} \rightarrow \mathbf{Cont}$$

Alcune definizioni

Semplificate: senza controllo condizioni di errore.

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

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

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

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

Eccezioni, si aggiunge alla sintassi dei comandi

$$c ::= \dots \mid \mathbf{trap} \ C \ l_1 : C_1, l_2 : C_2, \dots, l_n : C_n \ \mathbf{end} \mid \mathbf{escapeto} \ l \mid \dots$$

Facile assegnare semantica tramite continuazioni.

Salti

$$c ::= \dots \mid l : C \mid \mathbf{goto} \ l \mid \dots$$

Semantica più complessa, tramite definizioni ricorsive dell'ambiente e del comando.

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

Diversi aspetti:

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

$$\mathbf{Proc}_n = \mathbf{Cont} \rightarrow \mathbf{Ev}^n \rightarrow \mathbf{Cont}$$

- procedure ricorsive: environment definito tramite punto fisso.

$$\mathcal{D}[\mathbf{procl}(l_1); C]\rho v = v(p/l)$$

dove $p = \lambda\kappa.e.\mathcal{C}[C]\rho[e/l_1]\kappa$

$$\mathcal{D}[\mathbf{rec} \ \mathbf{procl}(l_1); C]\rho v = v(p/l)$$

dove $p = \lambda\kappa.e.\mathcal{C}[C]\rho[p/l, e/l_1]\kappa$

Legame statico e dinamico

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:

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

$$\mathcal{D}[\mathbf{procl}(l_1); C]\rho v = v((\lambda\rho'.\kappa.e.\mathcal{C}[C]\rho'[e/l_1]\kappa)/l)$$

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

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

$$\mathcal{D}[\mathbf{procl}(l_1); C]\rho v = v((\lambda\rho'.\kappa.e.\mathcal{C}[C](\mathit{mix}(\rho)(\rho')))[e/l_1]\kappa)/l)$$

Passaggio dei parametri, call by reference

$$\mathcal{D}[\mathbf{procl}(l_1); C]\rho v = v(p/l)$$

dove $p = \lambda\kappa.e.\mathcal{C}[C]\rho[e/l_1]\kappa$

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

Passaggio dei parametri, Call by Value

$$\mathcal{D}[\text{proc } l(l_1); C] \rho v = v(p/l)$$

dove $p = \lambda \kappa. \text{deref}(\text{ref}(\lambda e. \mathcal{C}[\mathcal{C}] \rho [e/l_1] \kappa))$

Il parametro viene valutato, se una locazione si restituisce il valore. Quindi viene creata una locazione, il valore viene memorizzato in essa, si passa alla procedura la locazione.

$$\text{ref, deref} : \text{ECont} \rightarrow \text{ECont}$$

$$\text{deref}(\epsilon)(e)(\sigma) = \text{cond}(\text{isLoc}(e), \epsilon(\sigma(e))(\sigma), \epsilon(e)(\sigma))$$

$$\text{deref}(\epsilon)(e)(\sigma) = \epsilon(\text{new}(\sigma))(\sigma)[e/\text{new}(\sigma)]$$

Per semplicità, σ denota lo store, si omette la componente input.

$\text{new}(\sigma)$ una locazione libera in σ .



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:

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

$$\mathcal{C}[\mathcal{E}_1(\mathcal{E}_2)] \rho \kappa = \mathcal{E}[\mathcal{E}_1] \rho (\lambda p. p \kappa (\mathcal{E}[\mathcal{E}_2] \rho))$$

$$\mathcal{E}[l] \rho \epsilon = \text{cond}(\text{isClosure}(\rho(l)), \rho(l)(\epsilon), \epsilon(\rho(l)))$$

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 operativa: small-step reduction,
- semantica denotazione: programmi come alberi,



Linguaggio non deterministico

Dijkstra guarded commands, nella versione parallela:

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

$$gc ::= b \rightarrow c \mid gc \parallel gc$$



CSP Concurrent Sequential Processes

Dijkstra guarded command + parallelismo \parallel

$$c := c_1 \parallel c_2 \mid \dots$$

Comandi, processi, in parallelo comunicano attraverso canali, non posso condividere variabili

$$c := \alpha!a \mid b\alpha?X \mid \dots$$

Guardie con comunicazione

$$gc ::= b \vee \alpha!a \rightarrow c \mid b \vee \alpha?X \rightarrow c \mid \dots$$

Restrizione dei canali

$$c := c_1 \setminus \alpha \mid \dots$$

CCS Calculus of Concurrent Systems

Robin Milner,

Versione sintetica del CSP, l'essenziale per studiare la concorrenza. Viene eliminato l'assegnazione, ristretti i comandi test e cicli.

$$p ::= 0 \mid \\ b \rightarrow p \mid \alpha!a \rightarrow p \mid \alpha?x \rightarrow p \mid \tau \rightarrow p \mid \\ p_0 + p_1 \mid p_0 \parallel p_1 \mid \\ p \setminus \alpha \mid p[f] \\ P(a_1, \dots, a_n)$$

Semantica Operazionale

Label transition system, due tipi di transazioni:

- \rightarrow descrive un passo di computazione,
- $\xrightarrow{\alpha!n} \xrightarrow{\alpha?n}$, descrive come un processo (comando) interagisce con l'esterno.

Motivazioni:

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

Regole di semantica operazionale:

Esercizio: definire un processo che simuli il comportamento di una variabile.

Pure CCS

Elimino dal CCS numeri naturali e variabili

$$p ::= \\ \sum_i p_i \\ \alpha! \rightarrow p \mid \alpha? \rightarrow p \mid \tau \rightarrow p \mid \\ P$$

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

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

Nozione bisimulazione

Quando due processi sono equivalenti?

Quando il loro albero di derivazione coincide.

Formalmente, una simulazione è una relazione R tale che:

...

$p \lesssim q$ se esiste una simulazione R t.c. pRq

\lesssim è la bisimulazione massima.

È una buona nozione di equivalenza?

È una congruenza?