ISABELLA MASTROENI CORRADO PRIAMI

SEMANTICA OPERAZIONALE: STRUMENTI E APPLICAZIONI

Linguaggi Imperativi e Funzionali

Indice

Prefazione

La semantica dei linguaggi di programmazione è quella parte della teoria che studia il significato dei programmi. Le tecniche operazionali che sono affrontate nel testo poi cercano di formalizzare il significato di un programma in termini delle azioni che una descrizione astratta di un generico mezzo di calcolo deve compiere per eseguirli.

Abbiamo adottato nel testo la semantica operazionale strutturale poiché è vicina all'intuizione e quindi può essere compresa senza approfondite conoscenze matematiche come invece accade per altre tecniche di definizione semantica. Nonostante la semplicità dei concetti matematici su cui si basa, l'approccio strutturale supporta tecniche formali di dimostrazione poiché è strettamente legato alla logica.

Gli aspetti formali nello studio dei linguaggi di programmazione costituiscono un aspetto fondamentale della preparazione di un informatico che voglia saper progettare o usare professionalmente un linguaggio. Lo studio teorico dei linguaggi caratterizza inoltre l'informatico rispetto a figure affini come quella dell'ingegnere informatico. Inoltre, la necessità di insegnare tecniche formali sin dai primi anni dei corsi di laurea in informatica è ormai un requisito essenziale.

L'esigenza di scrivere un testo ricco di esercizi sulla semantica operazionale strutturale dei linguaggi di programmazione imperativi e funzionali nasce dalla quasi totale assenza di materiale analogo in lingua italiana. Anche i testi pubblicati in altre lingue prediligono la descrizione dei fondamenti della tecnica senza dedicare uno spazio adeguato alle applicazioni di tali fondamenti ai costrutti dei linguaggi di programmazione. La pubblicazione alla quale maggiormente ci siamo ispirati è il rapporto tecnico dell'Università di Aharus di Plotkin in cui viene introdotta la semantica

operazionale strutturale. Infatti parte degli esercizi che qui abbiamo risolto sono proposti da Plotkin come lavoro per il lettore.

La parte sui linguaggi funzionali invece presenta prima un breve richiamo di λ-calcolo come fondamento del paradigma. Poi, seguendo ancora parzialmente il lavoro di Plotkin, questo viene esteso fino ad ottenere un semplice linguaggio funzionale. Abbiamo scelto questa strada per introdurre il paradigma funzionale poiché è quella adottata nella maggior parte dei corsi tenuti nelle università europee.

Abbiamo preferito presentare sotto forma di esercizi anche alcuni aspetti importanti dei linguaggi, come ad esempio certi tipi di passaggio dei parametri, per stimolare il lettore a comprendere come la semantica operazionale strutturale necessiti di pochissimi pre-requisiti per affrontare anche problemi relativamente complessi. Inoltre questo dovrebbe evidenziare come la semantica operazionale sia uno strumento che deve poi essere applicato nella formalizzazione del significato dei costrutti dei linguaggi, ma che non dipende da nessun linguaggio in particolare. In conclusione gli esercizi possono anche essere interpretati come approfondimenti di quegli aspetti minimali riguardanti la semantica dei linguaggi che sono presentati nei vari capitoli prima delle sezioni dedicate agli esercizi.

Questo testo è adottato nel corso di *Linguaggi: semantica, paradigmi e macchine astratte* tenuto al terzo anno del corso laurea in Informatica dell'Università di Verona. Esso può essere usato in qualunque corso in cui si vogliano presentare aspetti formali dei linguaggi di programmazione. La sua struttura modulare basata su approfondimenti presentati come esercizi lo rende versatile per una trattazione a diversi livelli di dettaglio.

Il materiale contenuto nel testo è stato presentato in due corsi sui linguaggi di programmazione, e la sua forma finale è frutto di una mediazione tra gli aspetti formali che devono essere trattati e le esigenze degli studenti che devono apprenderli. Per questo motivo ringraziamo tutti gli studenti dei corsi di Linguaggi III e Linguaggi IV dell'anno accademico '98/'99 del corso laurea in Informatica dell'Università di Verona per i loro commenti e le loro critiche costruttive.

Verona, 2 luglio 1999

ISABELLA MASTROENI
CORRADO PRIAMI

Capitolo 1

Introduzione

In questo testo richiamiamo le nozioni elementari di semantica operazionale strutturale [18] dei linguaggi imperativi e funzionali. Poi presentiamo sotto forma di esercizi approfondimenti ed estensioni dei concetti richiamati.

Prima di addentrarci nello studio dei linguaggi imperativi e funzionali, presentiamo una breve introduzione alle principali caratteristiche della semantica operazionale.

1.1 Semantica operazionale

Sin dalla nascita dell'informatica, il comportamento dinamico delle macchine è stato definito attraverso approcci operazionali che descrivono le transizioni tra gli stati che queste compiono durante il processo di calcolo.

La semantica operazionale fa la sua comparsa nella letteratura negli anni '60 grazie ai lavori di McCarthy [13] e Lucas [12]. In tale semantica un programma viene considerato come una sequenza di istruzioni atomiche che modificano gli stati delle macchine. Questi sono costituiti dai programmi stessi, dai dati su cui essi operano e da alcune strutture ausiliarie che rappresentano la memoria della macchina. Il passaggio da uno stato (*configurazione*) ad un altro è rappresentato mediante funzioni matematiche da stati a stati, dette transizione della macchina. Se necessario, le transizioni possono essere etichettate da informazioni che descrivono l'attività che ha

causato il passaggio di stato in modo da rendere più chiara l'evoluzione del sistema. Possiamo quindi definire l'esecuzione di un programma (*computazione*) come una sequenza di stati connessi l'uno all'altro da transizioni. PL/I e Algol60 sono stati i primi linguaggi di programmazione reali ad avere una definizione mediante semantica operazionale.

La semantica operazionale è molto vicina all'intuizione poiché descrive i passi che una macchina astratta compie durante il calcolo. Essa fornisce inoltre linee guida per i progettisti dei linguaggi senza però fissare soluzioni implementative a priori. Tale semantica descrive una macchina astratta facilmente simulabile da un interprete consentendo di costruire rapidamente prototipi dei linguaggi che si vogliono definire per poterli provare sul campo nel più breve tempo possibile.

La semantica operazionale di un linguaggio necessita di strutture dati molto semplici e di alcune operazioni definite su esse per descrivere in modo formale, e quindi non ambiguo, il significato dei programmi scritti nel suddetto linguaggio. Quindi le tecniche operazionali sono utilizzabili da una larga classe di utenti e sono anche adatte a scopi didattici. La tecnica maggiormente utilizzata per dimostrare proprietà e per definire costrutti mediante semantica operazionale è una forma di induzione detta strutturale che generalizza l'induzione matematica. Perciò per definire il significato di un programma in termini del significato dei suoi componenti elementari (*composizionalità*), dopo i lavori di de Bakker e de Roever [6], Plotkin [18] propose un approccio strutturale (*SOS*). La sua novità sta nell'usare come base la logica per la derivazione delle transizioni. Tali derivazioni sono ottenute inducendo sulla struttura sintattica dei linguaggi espressa in notazione BNF. Inoltre anche le transizioni sono definite induttivamente mediante assiomi e regole di inferenza quindi è possibile costruire un sistema di dimostrazione formale basato sulla logica.

L'approccio *SOS* è stato usato con successo per descrivere non solo linguaggi imperativi e funzionali, ma anche logici, orientati agli oggetti, concorrenti e distribuiti. Quindi la padronanza della tecnica consente di poter affrontare lo studio di nuovi linguaggi senza dover acquisire nuove conoscenze matematiche di base. Inoltre la versatilità del metodo lo rende adatto a descrivere linguaggi eterogenei che sempre più si rendono necessari con il diffondersi delle reti di calcolatori.

Fino ad ora abbiamo assunto implicitamente di descrivere il comportamento dinamico dei sistemi. Comunque la semantica operazionale strutturale è adatta anche a descrivere aspetti statici legati ai linguaggi. Gli aspetti statici riguardano tutte quelle proprietà che possono essere dedotte semplicemente dalla rappresentazione dei programmi e che non coinvolgono l'esecuzione (o la simulazione) dei costrutti che questi contengono. Un buon esempio di utilizzo di questa tecnica è la descrizione dei sistemi di tipi dei linguaggi per verificare staticamente se un programma applica i suoi costrutti ad operandi dei tipi corretti [4]. Questi aspetti sono di solito genericamente classificati come semantica statica dei linguaggi. Noi presenteremo nei prossimi capitoli, oltre la semantica dinamica, anche la semantica statica dei linguaggi imperativi e funzionali utilizzando questo approccio.

1.2 Sistemi di transizione

Cerchiamo qui di raffinare il concetto di descrizione operazionale di un sistema. Poiché il comportamento dinamico di un programma è definito dalle transizioni tra stati, la struttura matematica più adeguata a modellarlo è quella dei grafi. I grafi che definiamo mediante le regole di inferenza della semantica operazionale di un linguaggio sono detti *sistemi di transizione* e li trattiamo brevemente in questa sezione.

Un sistema è un modello matematico del fenomeno investigato ed è caratterizzato, ad un certo momento t, dal suo stato interno (*configurazione*) costituito da un *programma di controllo* e da alcuni *dati*. Quella appena fornita è una descrizione statica di sistema. Esso cambia il suo stato interno in base agli stimoli che riceve da un ambiente in cui opera o in base a particolari valori dei suoi dati. L'insieme delle configurazioni attraverso cui può transire un sistema caratterizza il suo comportamento dinamico. Per questo motivo parliamo dei sistemi di transizione come di modelli del comportamento dinamico dei programmi.

Definizione 1.1 (sistema di transizione). *Un* sistema di transizione *è una struttura* $(\Gamma, \rightarrow)$*, dove* Γ *è un insieme di elementi* γ *chiamati* configurazioni *e la relazione binaria* $\rightarrow \subseteq \Gamma \times \Gamma$ *è chiamata* relazione di transizione.

Nel seguito scriveremo $\gamma \longrightarrow \gamma'$ per $\langle \gamma, \gamma' \rangle \in \longrightarrow$. Alcune volte useremo la chiusura riflessiva e transitiva $\longrightarrow^*$ di una relazione di transizione definita da

$$\gamma_0 \longrightarrow^* \gamma_n \;\Leftrightarrow\; \exists \gamma_1 \ldots \gamma_n \,.\, \gamma_i \longrightarrow \gamma_{i+1},\; i \in [0, n].$$

Individuando particolari sottoinsiemi di Γ che interpretiamo come insiemi degli stati finali T o degli stati iniziali I, otteniamo rispettivamente i sistemi di transizione *iniziali* e *terminali*. Inoltre possiamo decidere di etichettare le transizioni con elementi di un certo alfabeto per descrivere l'azione del sistema che ne causa la transizione di stato. In questo caso parliamo di sistemi di transizione etichettati.

Un sistema di transizione può essere rappresentato convenientemente mediante un grafo in cui i nodi rappresentano le configurazioni e gli archi orientati le transizioni. I possibili comportamenti di un sistema sono quindi ottenuti visitando il corrispondente grafo del sistema di transizione. Per un esempio di costruzione di un sistema di transizione si veda l'Esercizio 3.3.

1.3 Induzione

Richiamiamo brevemente una generalizzazione del principio di induzione matematica detta induzione strutturale. Anziché prendere l'insieme $\mathbb{N}$ dei numeri naturali come base, si considerano gli elementi di una categoria sintattica e l'ordinamento sui naturali è sostituito dalla relazione *componente immediato di* sugli elementi sintattici.

Consideriamo ad esempio la seguente grammatica che definisce i numeri binari

$$b ::= 0 \mid 1 \mid b0 \mid b1$$

Gli elementi base sono costituiti dai letterali 0 e 1 che rappresentano i numeri binari 0 e 1 rispettivamente. Gli elementi composti sono invece $b0$ e $b1$. Vale la relazione 0 è componente immediato di $b0$ e 1 è componente immediato di $b1$. Se si vuole dimostrare una proprietà che vale per tutti i numeri binari, si può dimostrarla per 0 e 1 come base. Se poi riusciamo a dimostrarla anche per $b0$ e $b1$ assumendola valida per b, siamo in grado di affermare che tutti i numeri binari verificano la proprietà.

Generalizzando l'applicazione dell'induzione strutturale vista per i numeri binari ad un qualunque linguaggio definito mediante grammatiche o notazione BNF otteniamo il seguente principio.

Principio 1.2 (induzione strutturale). *Per dimostrare che una proprietà è valida per tutti gli elementi di una categoria sintattica*

1. *si dimostra la proprietà per tutti gli elementi base della categoria (quelli che non hanno nessun elemento come componente);*

2. *si dimostra la proprietà per tutti gli elementi composti assumendo che la proprietà sia verificata da tutti i loro componenti immediati (ipotesi induttiva).*

La tecnica di dimostrazione per induzione strutturale è strettamente correlata allo stile composizionale di definizione della semantica operazionale strutturale. Infatti il significato dei costrutti dei linguaggi di programmazione è definito in termini del significato dei loro immediati componenti.

Un'altra tecnica di dimostrazione che si deriva direttamente dallo stile logico di definizione dei sistemi di transizione è l'induzione sulla forma degli alberi di derivazione delle transizioni.

Principio 1.3 (induzione sugli alberi di derivazione). *Per dimostrare che una proprietà vale per tutte le transizioni di un certo linguaggio*

1. *si dimostra che essa vale per gli alberi di derivazione semplici (costituiti dagli assiomi della semantica operazionale strutturale per il linguaggio considerato);*

2. *si dimostra che la proprietà vale per tutti gli alberi di derivazione composti. Cioè si dimostra la proprietà per tutte le regole di inferenza (definite per le categorie sintattiche su cui si vuole dimostrare la proprietà) della semantica operazionale supponendo, per ipotesi induttiva, che essa sia vera per le premesse delle regole stesse.*

Per un esempio di applicazione di questo principio vedi Esercizio 4.3.

1.4 Prospettive

Con l'evoluzione dell'informatica la complessità dei sistemi studiati e progettati cresce enormemente. I progettisti devono di volta in volta considerare contemporaneamente una grande quantità e varietà di informazioni per guidare le loro scelte. Per esempio l'esplosione del Web e di Internet forniscono un ambiente globale di calcolo dove programmi allocati su siti distinti, potenzialmente in movimento (si pensi ai computer di bordo di un aereo), possono cooperare o persino migrare durante il calcolo. Questo impone l'integrazione di diverse applicazioni residenti in diversi siti e, potenzialmente, scritte in linguaggi distinti basati su paradigmi distinti. Quindi un requisito indispensabile è che le tecniche di specifica e di verifica siano quanto più possibile indipendenti dai linguaggi e dalle architetture.

L'attenzione principale delle tecniche *SOS* è rivolta al modo in cui viene definito il significato di un costrutto di programmazione piuttosto che ad uno specifico linguaggio. Vedremo infatti come sia possibile definire il significato di costrutti di linguaggi diversi senza dover cambiare l'impianto formale che utilizziamo. Quindi la specifica di un linguaggio può essere modificata durante il progetto o sintonizzata alle particolari esigenze del progettista con pochissimo sforzo e con un ridotto spreco di risorse. In conclusione la tecnica *SOS* rappresenta uno stile di definizione della semantica dei linguaggi.

Altro aspetto interessante dell'approccio utilizzato in questo testo è la possibilità di vedere le definizioni *SOS* come il nucleo di un generatore di interpreti. Dato un particolare dominio applicativo il progettista individua i costrutti essenziali per descrivere le soluzioni dei problemi e li definisce utilizzando la semantica *SOS*. Il generatore automatico di interpreti (o compilatori) produce un linguaggio orientato al dominio con un piccolo (e quindi efficiente) supporto a tempo di esecuzione.

1.5 Piano del lavoro

Nel prossimo capitolo richiamiano i concetti di base dei linguaggi imperativi come ambienti e memorie. Inoltre riportiamo la sintassi di un semplice linguaggio che useremo come base per gli esercizi proposti.

Il Capitolo 3 dopo aver introdotto la semantica statica e dinamica delle espressioni affronta la soluzione di alcuni problemi. La stessa struttura è seguita nei Capitoli 4, 5, 6 per le dichiarazioni, i comandi e le procedure rispettivamente.

Il Capitolo 7 richiama le nozioni fondamentali del λ-calcolo visto come base del paradigma di programmazione funzionale. Quindi affronta la soluzione di alcuni esercizi.

Il Capitolo 8 arricchisce la sintassi del λ-calcolo ottenendo un linguaggio funzionale *ML*-like. Dopo aver definito la semantica statica e dinamica del nuovo linguaggio, forniamo la soluzione di alcuni esercizi proposti.

Il Capitolo 9 fornisce alcune considerazioni conclusive e inquadra alcuni aspetti studiati nel panorama dei vari linguaggi di programmazione.

Capitolo 2

Un semplice linguaggio imperativo

In questo capitolo richiamiamo brevemente le caratteristiche essenziali dei linguaggi imperativi e forniamo la sintassi del linguaggio che assumiamo definito per la soluzione degli esercizi presentati nei prossimi capitoli. La semantica statica e dinamica di questo nucleo che chiameremo IMP è definita suddividendo i costrutti in base alle categorie sintattiche individuate (espressioni, dichiarazioni e comandi) ed è riportata nei corrispondenti capitoli. Per motivi didattici le procedure sono trattate in un capitolo a parte anche se la loro sintassi rientra in parte nella categoria sintattica delle dichiarazioni e in parte in quella dei comandi. Infine richiamiamo i concetti di induzione strutturale e di regole di inferenza.
Nella prossima sezione richiamiamo la definizione delle strutture che utilizzeremo nelle definizioni semantiche: ambienti statici e dinamici e memorie ed in quella successiva introduciamo la sintassi di IMP. Nell'ultima sezione richiamiamo brevemente alcuni principi di induzione utilizzati nel testo.

2.1 Ambienti e memorie

Per tener traccia dei valori assegnati agli identificatori nei linguaggi imperativi facciamo ricorso al concetto di *memoria*. Una memoria è una col-

lezione di *locazioni* che possono essere aggiornate dinamicamente. Una locazione può essere in tre stati distinti: inutilizzata, indefinita o definita. La locazione è *inutilizzata* quando non è ancora stata allocata per qualche identificatore. Si trova nello stato *indefinito* se è già stata allocata, ma non contiene ancora un valore (il corrispondente identificatore è stato dichiarato, ma non ancora inizializzato). Infine, la locazione è *definita* quando contiene un valore. Nel seguito utilizzeremo il termine memoria per denotare un'immagine dell'insieme di locazioni ad un istante particolare.

Poiché il valore degli identificatori, così come quello delle locazioni, può variare dinamicamente è ragionevole associare identificatori a locazioni. Il valore di un identificatore sarà poi quello contenuto nella corrispondente locazione. Le locazioni sono un meccanismo di indirizzamento indiretto tra gli identificatori e i valori che possono memorizzare. Per formalizzare questa associazione definiamo prima le locazioni.

Definizione 2.1 (locazione). *Per ogni valore memorizzabile di tipo τ, sia Loc_τ un insieme infinito di celle di memoria che possono memorizzare valori di tipo τ. Quindi la collezione di* locazioni *è*

$$Loc = \cup_{\tau \in \text{STyp}} Loc_\tau$$

con metavariabile l, dove STyp *è l'insieme dei tipi composti da elementi memorizzabili (vedi prossima sezione).*

A questo punto siamo in grado di definire il concetto di memoria.

Definizione 2.2 (memoria). *Una* memoria *è un elemento dello spazio di funzioni*

$$Stores = \cup_{L \subseteq_f Loc} Store_L$$

dove $\subseteq_f$ significa "un sottoinsieme finito di" e $Store_L : L \to SVal \cup \{?, \bot\}$ ha metavariabile σ. I simboli $?$ e $\bot$ denotano rispettivamente lo stato inutilizzato e indefinito. Imponiamo alle memorie la condizione

$$\forall l \in L \,.\, \sigma(l) \in \tau \;\Leftrightarrow\; l \in L \cap Loc_\tau$$

per assicurare che le locazioni siano utilizzate correttamente dal punto di vista dei tipi.

L'operazione di aggiornamento delle memorie è definita come segue. Siano $L, L' \subseteq_f Loc$, $\sigma \in \text{Store}_L$ *e* $\sigma' \in \text{Store}_{L'}$. *Definiamo* $\sigma[\sigma'] \in \text{Store}_{L \cup L'}$ *come*

$$\sigma[\sigma'](l) = \begin{cases} \sigma'(l) & l \in L' \\ \sigma(l) & l \in (L - L') \end{cases}$$

Se $L \cap L' = \emptyset$ *scriviamo* σ, σ' *al posto di* $\sigma[\sigma']$. *La notazione* $\sigma : L$ *significa che il dominio di* σ *è* L.

A questo punto possiamo introdurre il concetto di *ambiente* $(dinamico)$ *Env* che associa gli identificatori con le locazioni della memoria. Quindi, componendo ambienti e memorie recuperiamo il valore di un identificatore. Più precisamente, un ambiente lega gli identificatori con i *valori denotabili DVal*. Lo speciale valore indefinito (ancora scritto $\perp$) è necessario per modellare il fatto che un identificatore non è legato a nessun valore denotabile nell'ambiente corrente.

Definizione 2.3 (ambiente). *Un* ambiente *dinamico è un elemento dello spazio di funzioni*

$$Env = \cup_{I \subseteq_f Id} Env_I$$

dove $Env_I : I \to DVal \cup Loc \cup \{\perp\}$ *ha metavariabile* ρ. *Le operazioni di aggiornamento degli ambienti* $\rho[\rho'] \in Env_{I \cup I'}$ *e* $\rho, \rho' \in Env_{I \cup I'}$ *sono definite come per le memorie. Anche la notazione* $\rho : I$ *ha lo stesso significato.*

Infine, come vedremo nei prossimi capitoli, questi ambienti vengono generati e modificati dalle *dichiarazioni* che a nuovi identificatori associano un valore (nel caso di dichiarazioni di costante) o una locazione (nel caso di dichiarazioni di variabili). Per quest'ultimo caso dobbiamo definire una funzione che ad ogni nuovo identificatore definito associ una locazione non ancora utilizzata. Per fare ciò dobbiamo supporre di poter ordinare le locazioni dello stesso tipo in modo da poter associare loro un valore naturale nel momento in cui vengono utilizzate, in tal modo possiamo distinguerle da quelle non ancora usate.

Definizione 2.4 (generatore di locazioni). *Un* generatore di locazioni *è una funzione* $New : Loc \times DTyp \rightarrow Loc \times \mathbb{N}$ *definita come la funzione che esegue la seguente associazione:*

$$New(L, \tau) = \langle l, m \rangle,\ L \subseteq_f Loc_\tau, l \in Loc_\tau,$$
$$m = (max\{n \mid \exists \langle l, n \rangle \,.\, l \in L\}) + 1$$

In generale, della coppia risultante, considereremo solo la locazione (vista la sola utilità tecnica della numerazione) usando la notazione $l \in New_\tau(L)$ al posto di quella formalmente corretta $\langle l, m \rangle \in New(L, \tau)$.

A questo punto, in modo analogo agli ambienti introdotti sopra per associare dinamicamente identificatori e locazioni, possiamo definire degli ambienti statici Δ che associano agli identificatori il tipo dei valori che poi denoteranno a tempo di esecuzione. Questi ambienti saranno la base per la definizione della semantica statica.

Definizione 2.5 (ambiente statico). *Un* ambiente statico *(o di tipi) è un elemento dello spazio di funzioni TEnv definito da*

$$TEnv = \cup_{I \subseteq_f Id} TEnv_I$$

dove $TEnv_I : I \rightarrow$ DTyp *ha metavariabile* Δ *e* DTyp *è l'insieme dei tipi* denotabili. *Le operazioni di aggiornamento degli ambienti* $\Delta[\Delta'] \in TEnv_{I \cup I'}$ *e* $\Delta, \Delta' \in TEnv_{I \cup I'}$ *sono definite come per le memorie. Anche la notazione* $\Delta : I$ *ha lo stesso significato.*

Per essere precisi, scriveremo $\Delta \vdash_I$ per rendere esplicito l'insieme finito di identificatori I che costituisce il dominio di Δ.

La seguente definizione stabilisce la compatibilità tra gli ambienti dinamici ρ e gli ambienti statici Δ.

Definizione 2.6 (compatibilità di ambienti). *Sia* $\rho : I$ *un ambiente dinamico e* $\Delta : I$ *un ambiente statico con* $I \subseteq_f Id$. *Gli ambienti* ρ *e* Δ *sono* compatibili *(scritto* $\rho : \Delta$*) se e soltanto se*

$$\forall id \in I \,.\, (\Delta(id) = \tau \ \wedge \ \rho(id) \in \tau) \vee$$
$$\exists \tau \,.\, (\Delta(id) = \tau loc \wedge \rho(id) \in Loc_\tau)$$

Nelle regole di inferenza contenute negli esercizi proposti di seguito usiamo sempre come ambienti Δ e ρ invece degli ambienti vuoti perché supponiamo che le regole fornite costistuiscano un frammento di codice e che quindi siano inserite in un contesto più ampio. Infatti, se si deve esaminare un intero programma gli ambienti (statico e dinamico) con cui iniziamo l'applicazione delle regole sono quelli vuoti.

2.2 Sintassi

Qui richiamiamo brevemente la sintassi del linguaggio di programmazione IMP utilizzando la notazione BNF. Gli insiemi sintattici di base sono

TIPI. *Typ* $= \{bool, int\}$ con metavariabile τ.

VALORI DI VERITÀ. *bool* $= \{tt, ff\}$ con metavariabile t.

NUMERI. *int* $= \{\ldots, -1, 0, 1, \ldots\}$ con metavariabili m, n.

IDENTIFICATORI. *Id* $= \{rate, a25, b, x, \ldots\}$ con metavariabile id, x.

OPERATORI UNARI. *Uop* $= \{\mathbf{not}\}$ con metavariabile $\mathtt{uop}$.

OPERATORI BINARI. *Bop* $= \{+, -, \times, =, \mathbf{or}\}$ con metavariabile $\mathtt{bop}$.

e le categorie sintattiche derivate di IMP sono

COSTANTI. *Con* con metavariabile k definita da

$$k ::= n \mid t$$

ESPRESSIONI. *Exp* con metavariabile e definite come

$$e ::= k \mid id \mid e\,\mathtt{bop}\,e' \mid \mathtt{uop}\,e.$$

DICHIARAZIONI. *Dic* con metavariabile d definite come

$$d ::= \mathbf{nil} \mid \mathbf{const}\ x : \tau = e \mid \mathbf{var}\ x : \tau = e \mid d; d \mid d\ \mathbf{and}\ d \mid d\ \mathbf{in}\ d \mid form = ae \mid \mathbf{procedure}\ p(form)\ c \mid \rho$$

COMANDI. *Com* con metavariabile c definiti come

$$c ::= \mathbf{nil} \mid id := e \mid c; c \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid d; c \mid p(ae)$$

ATTUALI. *AExp* con metavariabile ae definiti come

$$ae ::= \bullet \mid e, ae$$

ATTUALI VALUTATI. *ACon* con metavariabile ak definiti come

$$ak ::= \bullet \mid k, ak$$

FORMALI. *Form* con metavariabile *form* definiti come

$$\mathit{form} ::= \bullet \mid \texttt{const}\ x : \tau, \mathit{form} \mid \texttt{var}\ x : \tau, \mathit{form}$$

CHIUSURE. *Abs* con metavariabile abs definite come

$$abs ::= \lambda\, form.c$$

VALORI ESPRIMIBILI. *Eval* $=$ *bool* $\cup$ *int* con metavariabile ev definiti come

$$ev ::= k$$

TIPI ESPRIMIBILI. *ETyp* $=$ *Typ* con metavariabile et definito come

$$et ::= \tau$$

VALORI DENOTABILI. *DVal* $=$ *bool* $\cup$ *int* $\cup$ *Loc* $\cup$ *Abs* con metavariabile dv definiti come

$$dv ::= k \mid l \mid abs$$

TIPI DENOTABILI. $\mathit{DTyp} = \mathit{Typ} \cup (\mathit{Typ} \times loc) \cup (\mathit{ATyp} \times proc)$ con metavariabile dt definiti come

$$dt ::= \tau \mid \tau loc \mid aetproc$$

VALORI MEMORIZZABILI. $\mathit{SVal} = \mathit{bool} \cup \mathit{int}$ con metavariabile sv definiti come

$$sv ::= k$$

TIPI MEMORIZZABILI. $\mathit{STyp} = \mathit{bool} \cup \mathit{int}$ con metavariabile st definiti come

$$st ::= \tau$$

TIPI DEGLI ATTUALI. *ATyp* con metavariabile aet definiti come

$$aet ::= \bullet \mid et, aet$$

Alcuni commenti sono necessari per le categorie sintattiche derivate. Le costanti costituiscono i valori a cui sono valutate le espressioni che in IMP possono solo essere interi (n) o booleani (t).

Le espressioni sono costituite componendo costanti o identificatori attraverso gli operatori unari e binari.

Le dichiarazioni contengono la dichiarazione di costante (**const**) e di identificatore modificabile (**var**) che possono essere composte nelle dichiarazioni sequenziali (;), simultanee (**and**) o private (**in**). Inoltre è possibile dichiarare identificatori come nomi di procedure. La dichiarazione $form = ae$ serve per implementare il passaggio dei parametri operando il legame tra i formali introdotti nella dichiarazione di una procedura e gli attuali presenti nella chiamata della procedura. Infine, l'ambiente ρ è introdotto nella sintassi delle dichiarazioni per motivi tecnici che saranno chiari quando definiremo la semantica dinamica delle dichiarazioni.

I comandi, oltre all'assegnamento, la composizione sequenziale, il comando condizionale e il comando iterativo, contengono il costrutto di blocco $d; c$. Questo serve a delimitare il campo di azione delle dichiarazioni

ed è strettamente legato al concetto di scoping che tratteremo in dettaglio nel capitolo sulle procedure. Infine abbiamo la chiamata di procedura che individua i parametri attuali ae.

I parametri attuali sono una lista di espressioni che quindi viene valutata ad una lista di costanti (vedi la categoria sintattica degli attuali valutati). Abbiamo introdotto anche gli attuali valutati poiché IMP implementa il passaggio dei parametri per valore e quindi gli attuali vengono valutati completamente prima di legarli ai formali.

I parametri formali sono dichiarati in modo analogo alla dichiarazione degli identificatori con la sola differenza che non sono inizializzati. La loro inizializzazione avverrà al momento della chiamata della procedura che contiene i corrispondenti attuali.

La categoria sintattica delle chiusure è stata introdotta per avere una struttura da legare nell'ambiente dinamico agli identificatori di procedura. Infatti la chiusura contiene tutta l'informazione che ci occorre: i parametri formali ed il codice del corpo della procedura.

Abbiamo poi tre coppie di categorie sintattiche simili che definiscono i valori esprimibili, denotabili e memorizzabili ed i loro corrispondenti tipi. I valori esprimibili sono quelli che possono essere restituiti come valutazione di un'espressione ed abbiamo già detto essere le costanti.

I valori denotabili sono quelli che possono essere legati agli identificatori negli ambienti dinamici. Oltre i valori esprimibili abbiamo quindi le locazioni e le chiusure. Il tipo corrispondente alle locazioni è τloc dove τ indica il tipo memorizzabile nella locazione ed il suffisso loc esplicita il fatto che si tratta di una locazione. Il tipo di una chiusura corrisponde alla lista di tipi associata ai formali (che corrisponde poi alla lista dei tipi associata agli attuali) seguita dal suffisso $proc$.

Infine i valori memorizzabili sono quelli che possono essere registrati in una locazione di memoria che nel caso di IMP coincidono con quelli esprimibili.

L'ultima categoria sintattica che abbiamo descrive i tipi dei parametri attuali. Questi sono una lista di tipi esprimibili in quanto possiamo passare solo espressioni come attuali che poi sono interamente valutate poiché implementiamo il passaggio dei parametri per valore.

Capitolo 3

Espressioni

Le espressioni sono una categoria sintattica che compare in tutti i linguaggi di programmazione. Esse sono *valutate* per ottenere un valore. Il tipo di un valore ottenuto dalla valutazione delle espressioni è uno dei tipi esprimibili (*ETyp*) del linguaggio. L'insieme di tipi *ETyp* definisce naturalmente l'insieme dei valori esprimibili *EVal*.

I costituenti elementari delle espressioni sono i letterali (nel nostro linguaggio IMP costanti ed identificatori) che sono poi composti mediante gli operatori.

In questo capitolo definiremo le espressioni aritmetiche e booleane fornendo la loro semantica statica e dinamica nella prima sezione. Quindi presenteremo alcuni esercizi risolti nella sezione successiva.

3.1 Semantica di EXP

Introduciamo per prima cosa una funzione che individua l'insieme degli identificatori che hanno occorrenze libere all'interno di un'espressione. Nel seguito ometteremo il termine occorrenze riferendoci a loro come identificatori. Il contesto consente di eliminare l'ambiguità tra il riferimento alle occorrenze e quello agli identificatori.

Definizione 3.1 (identificatori liberi). *La funzione* $FV : Exp \to Id$*, che ad ogni espressione associa l'insieme degli identificatori liberi in essa*

contenuti, è definita per induzione da

$$FI(k) = \emptyset$$
$$FI(id) = \{id\}$$
$$FI(e_0 \texttt{ bop } e_1) = FI(e_0) \cup FI(e_1)$$
$$FI(\texttt{uop } e) = FI(e)$$

Per le epressioni la funzione $DI : Exp \rightarrow Id$ che individua gli identificatori in posizione di definizione è la costante $\emptyset$ in quanto nel nostro linguaggio IMP le espressioni non definiscono identificatori.
Definiamo adesso la semantica statica per le espressioni del nostro linguaggio. Lo scopo della semantica statica è quello di associare un tipo a ciascuna espressione corretta; se questo accade diremo che l'espressione è *ben formata*. Le regole sono riportate in Tab. 3.1.

$$\emptyset \vdash_\emptyset t : \textit{bool} \quad \emptyset \vdash_\emptyset n : \textit{int} \quad \Delta \vdash_I id : \tau, \;\; \Delta(id) \in \{\tau, \tau loc\}$$

$$\frac{\Delta \vdash_I e : \tau_0, e' : \tau_1}{\Delta \vdash_I e \texttt{ bop } e' : \tau_{bop}(\tau_0, \tau_1)} \qquad \frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I \texttt{uop } e : \tau_{uop}}$$

Tabella 3.1: Semantica statica per le espressioni di IMP.

I tre assiomi assegnano un tipo agli elementi base su cui sono costruite le espressioni. I primi due hanno ambiente dei tipi vuoto poiché non occorre alcuna informazione per associare il tipo τ agli elementi stessi. L'assioma per gli identificatori associa alle occorrenze di id il tipo registrato in Δ (vedremo che saranno le dichiarazioni a costruire gli ambienti di tipo) purché non sia il tipo di una procedura. Le due regole servono per assegnare il tipo alle espresioni composte. La prima considera gli operatori binari $\texttt{bop} = \{=, +, -, *, or\}$. La funzione $\tau_{bop} : ETyp \times ETyp \rightarrow ETyp \cup \{\perp\}$, definita in Tab. 3.2 (abbiamo usato il simbolo $\perp$ per i casi in cui la funzione non è definita), determina il tipo dell'espressione composta a partire dal tipo delle espressioni componenti, disponibile nelle premesse

$+, -, *$	int	bool
int	int	$\bot$
bool	$\bot$	$\bot$

$=$	int	bool
int	$bool$	$\bot$
bool	$\bot$	$bool$

or	int	bool
int	$\bot$	$\bot$
bool	$\bot$	$bool$

not	int	bool
	$\bot$	$bool$

Tabella 3.2: Definizione delle funzioni τ_{bop} e τ_{uop}

della regola. L'ultima regola è analoga alle precedenti e gestisce gli operatori unari. La funzione $\tau_{uop} : ETyp \rightarrow ETyp \cup \{\bot\}$ è definita in Tab. 3.2.

Le regole della semantica dinamica descrivono come vengono valutate le espressioni di IMP considerando che le configurazioni terminali per le espressioni sono del tipo $\langle k, \sigma \rangle$. Le regole sono riportate in Tab. 3.3. Il primo assioma descrive la valutazione degli identificatori. La semantica associa a ciascun id il valore registrato in ρ se id è una costante oppure il valore contenuto nella locazione di memoria l associata a id in ρ se id è un identificatore modificabile. Notiamo che non possiamo avere identificatori di procedure poiché la semantica statica non riuscirebbe ad assegnare un tipo all'espressione. Le successive due regole permettono la valutazione delle epressioni coinvolte in operazioni mediante un operatore binario `bop`; si nota che la valutazione va da sinistra a destra, cioè si valuta l'espressione a destra dell'operatore solo dopo aver finito la valutazione dell'espressione a sinistra. La terza regola ci dice che i passi di valutazione di una espressione possono essere fatti anche nel contesto di un operatore unario. Infine, i due assiomi permettono di completare l'operazione associando all'operatore sintattico applicato ai suoi operandi il valore risultato del corrispondente operatore semantico applicato all'interpretazione semantica degli operandi.

Per caratterizzare il comportamento dinamico della valutazione di un'espressione, definiamo una funzione che restituisce il valore associato all'e-

$$\rho \vdash_\Delta \langle id, \sigma\rangle \rightarrow_e \langle k, \sigma\rangle, k = \rho(id) \ \vee \ (\rho(id) = l \ \wedge \ k = \sigma(l))$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e_0, \sigma\rangle}{\rho \vdash_\Delta \langle e \ \mathtt{bop} \ e', \sigma\rangle \rightarrow_e \langle e_0 \ \mathtt{bop} \ e', \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e', \sigma\rangle \rightarrow_e \langle e_1, \sigma\rangle}{\rho \vdash_\Delta \langle k \ \mathtt{bop} \ e', \sigma\rangle \rightarrow_e \langle k \ \mathtt{bop} \ e_1, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{uop} \ e, \sigma\rangle \rightarrow_e \langle \mathtt{uop} \ e', \sigma\rangle}$$

$$\rho \vdash_\Delta \langle k \ \mathtt{bop} \ k', \sigma\rangle \rightarrow_e \langle k'', \sigma\rangle, k'' = k \ \mathtt{bop} \ k'$$

$$\rho \vdash_\Delta \langle \mathtt{uop} \ t, \sigma\rangle \rightarrow_e \langle t', \sigma\rangle, t' = \mathtt{uop} \ t$$

Tabella 3.3: Semantica dinamica per le espressioni di IMP.

spressione nelle configurazioni finali.

Definizione 3.2 (valutazione). *La funzione*

$$Eval : Exp \times Store \rightarrow Con,$$

che descrive il comportamento dinamico delle espressioni a partire da una memoria σ restituendo il valore in cui esse sono valutate, è definita da

$$Eval(e, \sigma) = k \ \Leftrightarrow \ \langle e, \sigma\rangle \rightarrow_e^* \langle k, \sigma\rangle$$

A questo punto possiamo utilizzare il comportamento dinamico delle espressioni per definire una relazione di equivalenza sugli elementi di Exp

Definizione 3.3 (equivalenza). *L'equivalenza di espressioni $\equiv \subseteq Exp \times Exp$ è definita da*

$$e_0 \equiv e_1 \ \Leftrightarrow \ \forall \sigma. \, (Eval(e_0, \sigma) = Eval(e_1, \sigma))$$

3.2 Esercizi

Esercizio 3.1. *Si scrivano le regole per la valutazione delle espressioni da destra a sinistra, contrapposta alla valutazione da sinistra a destra definita nella semantica di* IMP.

SOLUZIONE. Nella valutazione da destra a sinistra un'espressione transisce quando si effettua una transizione nel suo sottotermine più a destra e solo quando a destra si ha una costante si procede col ridurre il termine a sinistra. Le regole della semantica dinamica sono

1. $$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e \langle e', \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \mathtt{bop}\ e, \sigma\rangle \to_e \langle e_0 \mathtt{bop}\ e', \sigma\rangle}$$

2. $$\frac{\rho \vdash_\Delta \langle e_0, \sigma\rangle \to_e \langle e_0', \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \mathtt{bop}\ k, \sigma\rangle \to_e \langle e_0' \mathtt{bop}\ k, \sigma\rangle}$$

3. $\rho \vdash_\Delta \langle k_0 \mathtt{bop}\ k, \sigma\rangle \to_e \langle k', \sigma\rangle$ dove $k' = k_0 \mathtt{bop}\ k$

Le regole per gli operatori unari e gli identificatori non devono essere modificate poiché hanno una sola espressione come componente. □

Esercizio 3.2. *Definire le regole per la valutazione* parallela *delle espressioni, in modo che la seguente sequenza di transizioni sia possibile:*

$$(1 + (2 + 3)) + ((4 + 5) + 6) \to_e (1 + (2 + 3)) + (9 + 6) \to_e$$
$$(1 + 5)(9 + 6) \to_e 6 + (9 + 6) \to_e 6 + 15 \to_e 21$$

Mostrare, quindi, che la sequenza di transizioni può essere derivata dalle regole fornite.

SOLUZIONE. Per risolvere l'esercizio bisogna notare che per permettere il parallelismo delle esecuzioni delle transizioni bisogna avere regole che permettano sia la valutazione da sinistra a destra che quella da destra a sinistra (come suggerisce la sequenza fornita dall'esercizio). Le regole della semantica dinamica sono quindi

1. $$\frac{\rho \vdash_\Delta \langle e_0, \sigma\rangle \to_e \langle e_0', \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \mathtt{bop}\ e_1, \sigma\rangle \to_e \langle e_0' \mathtt{bop}\ e_1, \sigma\rangle}$$

$$2.\quad \frac{\rho \vdash_\Delta \langle e_1, \sigma\rangle \to_e \langle e_1', \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \,\mathtt{bop}\, e_1, \sigma\rangle \to_e \langle e_0 \,\mathtt{bop}\, e_1', \sigma\rangle}$$

$$3.\quad \rho \vdash_\Delta \langle k_0 \,\mathtt{bop}\, k_1, \sigma\rangle \to_e \langle k', \sigma\rangle \text{ dove } k' = k_0 \,\mathtt{bop}\, k_1$$

Nel seguito etichetteremo le transizioni con dei numeri per identificarle univocamente all'interno di computazioni (sequenze di transizioni). Inoltre annoteremo gli alberi di derivazione delle transizioni con i nomi delle regole applicate. Per esempio nella derivazione della transizione $\xrightarrow{1}$ applichiamo prima l'assioma (3.), poi la regola (1.) ed infine la regola (2.). Si noti che il nome della regola è riportato a destra della conclusione dell'istanza dell'applicazione della regola.

A questo punto vediamo mediante quale sequenza di regole sopra definite si riesce a derivare la computazione data:

$$(1+(2+3))+((4+5)+6) \xrightarrow{1} (1+(2+3))+(9+6) \xrightarrow{2} (1+5)+(9+6) \xrightarrow{3} 6+(9+6) \xrightarrow{4} 6+15 \xrightarrow{5} 21$$

$$\xrightarrow{1}: \frac{\dfrac{}{\rho \vdash_\Delta \langle (4+5), \sigma\rangle \to_e \langle 9, \sigma\rangle\ (3.)}}{\rho \vdash_\Delta \langle (4+5)+6, \sigma\rangle \to_e \langle 9+6, \sigma\rangle\ (1.)}}{\rho \vdash_\Delta \langle (1+(2+3))+((4+5)+6), \sigma\rangle \to_e \langle (1+(2+3))+(9+6), \sigma\rangle\ (2.)}$$

$$\xrightarrow{2}: \frac{\dfrac{}{\rho \vdash_\Delta \langle (2+3), \sigma\rangle \to_e \langle 5, \sigma\rangle\ (3.)}}{\rho \vdash_\Delta \langle 1+(2+3)\sigma\rangle \to_e \langle 1+5, \sigma\rangle\ (2.)}}{\rho \vdash_\Delta \langle (1+(2+3))+(9+6), \sigma\rangle \to_e \langle (1+5)+(9+6), \sigma\rangle\ (1.)}$$

$$\xrightarrow{3}: \frac{\rho \vdash_\Delta \langle (1+5), \sigma\rangle \to_e \langle 6, \sigma\rangle\ (3.)}{\rho \vdash_\Delta \langle (1+5)+(9+6), \sigma\rangle \to_e \langle 6+(9+6), \sigma\rangle\ (1.)}$$

$$\xrightarrow{4}: \frac{\rho \vdash_\Delta \langle (9+6), \sigma\rangle \to_e \langle 15, \sigma\rangle\ (3.)}{\rho \vdash_\Delta \langle 6+(9+6), \sigma\rangle \to_e \langle 6+15, \sigma\rangle\ (2.)}$$

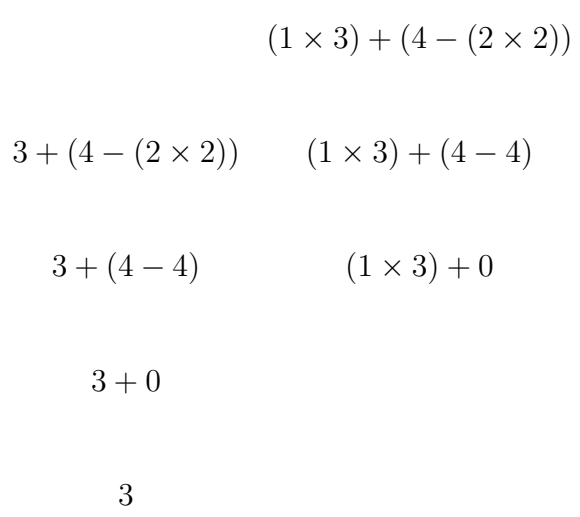

Figura 3.1: Sistema di transizione di $(1 \times 3) + (4 - (2 \times 2))$.

$$\xrightarrow{5}: \rho \vdash_{\Delta} \langle (6 + 15), \sigma \rangle \rightarrow_e \langle 21, \sigma \rangle \ (3.)$$

Si noti che, poiché non ci sono identificatori nelle espressioni, possiamo assumere sia ρ che σ vuoti. □

Esercizio 3.3. *Assumendo le regole introdotte nell'Esercizio 3.2 per la valutazione parallela delle espressioni, costruire il sistema di transizione di* $(1 \times 3) + (4 - (2 \times 2))$.

SOLUZIONE. Non riportiamo ambiente e memoria nella costruzione del sistema di transizione poiché l'espressione è costituita solo da costanti. Lo stato iniziale è quindi l'espressione stessa da cui sono possibili due transizioni a seconda che si valuti prima la moltiplicazione di sinistra o di destra.

Le derivazioni corrispondenti alle prime due transizioni sono

$$\frac{\rho \vdash_\Delta \langle (1 \times 3), \sigma\rangle \rightarrow_e \langle 3, \sigma\rangle \ (3.)}{\rho \vdash_\Delta \langle (1 \times 3) + (4 - (2 \times 2)), \sigma\rangle \rightarrow_e \langle 3 + (4 - (2 \times 2)), \sigma\rangle \ (1.)}$$

per la valutazione da sinistra, e

$$\frac{\dfrac{\rho \vdash_\Delta \langle (2 \times 2), \sigma\rangle \rightarrow_e \langle 4, \sigma\rangle \ (3.)}{\rho \vdash_\Delta \langle (4 - (2 \times 2)), \sigma\rangle \rightarrow_e \langle 4 - 4, \sigma\rangle \ (2.)}}{\rho \vdash_\Delta \langle (1 \times 3) + (4 - (2 \times 2)), \sigma\rangle \rightarrow_e \langle (1 \times 3) + (4 - 4), \sigma\rangle \ (2.)}$$

per la valutazione da destra. Proseguendo con la costruzione di tutte le possibili derivazioni fino alla configurazione finale $\langle 3, \sigma\rangle$ otteniamo il sistema di transizione in Fig. 3.1. □

Esercizio 3.4. *Si modifichi la sintassi di* IMP *aggiungendo alla categoria sintattica delle espressioni il costrutto:*

$$e ::= (id := e)$$

dove l'espressione viene valutata al valore di e *contemporaneamente all'esecuzione dell'assegnamento che, in generale, causa side-effects.*

SOLUZIONE. È importante capire cosa deve essere modificato nel linguaggio base da cui partiamo (IMP) in relazione ai costrutti aggiunti. Si nota infatti che tale costrutto introduce la possibilità che si verifichino side-effects nella valutazione delle espressioni poiché le nuove espressioni modificano la memoria. Esse però non alterano l'insieme di identificatori in posizione di definizione in quanto comunque non contengono dichiarazioni. Allora dobbiamo solo aggiungere la seguente definizione per gli identificatori liberi:

$$FI(id := e) = \{id\} \cup FI(e)$$

Definiamo ora la semantica statica e dinamica del nuovo costrutto.

Semantica statica
Consideriamo l'ambiente di tipi iniziale Δ sull'insieme I di identificatori, in base al quale determiniamo il tipo della nuova espressione

$$\frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I id := e : \tau},\ \Delta(id) = \tau loc$$

Quindi decidiamo di assegnare all'espressione $id := e$ il tipo che la semantica statica assegna ad e.

Semantica dinamica
Valutiamo prima l'espressione e registrando in σ le eventuali modifiche

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma'\rangle}{\rho \vdash_\Delta \langle id := e, \sigma\rangle \rightarrow_e \langle id := e', \sigma'\rangle}$$

e poi assegnamo il valore k ottenuto alla locazione di id restituendolo contemporaneamente come risultato della valutazione dell'espressione.

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle k, \sigma'\rangle}{\rho \vdash_\Delta \langle id := k, \sigma\rangle \rightarrow_e \langle k, \sigma[k/l]\rangle},\ l = \rho(id)$$

Inoltre poiché la valutazione delle espressioni può modificare la memoria, dobbiamo rimpiazzare ciascuna transizione $\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma\rangle$ in Tab. 3.3 con $\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma'\rangle$. □

Esercizio 3.5. *Definire semantica statica e dinamica della seguente nuova espressione*

$$e ::= \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$$

con la semantica intuitiva di valere e_1 se e_0 è vera ed e_2 altrimenti.
SOLUZIONE. Come nei precedenti esercizi, la prima cosa da fare è definire gli insiemi degli identificatori liberi (non ci sono identificatori in posizione di definizione perché le espressioni non contengono dichiarazioni)

$$FI(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) = FI(e_0) \cup FI(e_1) \cup FI(c_2)$$

Semantica statica
Consideriamo l'ambiente di tipi iniziale Δ definito sull'insieme I di identificatori, in base al quale definiamo una semantica statica. Si nota che, essendoci una condizione, l'espressione è ben formata solo nel caso in cui e_0 sia booleano. Le altre espressioni possono essere di qualunque tipo purché lo stesso per entrambe in quanto l'uso dell'una o dell'altra in ogni contesto deve essere indifferente.

$$\frac{\Delta \vdash_I e_0 : \mathit{bool},\ \Delta \vdash_I e_1 : \tau_0,\ \Delta \vdash_I e_2 : \tau_0}{\Delta \vdash_I \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau_0}$$

Se permettessimo ad e_1 di essere di tipo τ_1 e a e_2 di essere di tipo τ_2 non potremmo determinare staticamente la correttezza del frammento

$$4 = \mathbf{if}\ e_0\ \mathbf{then}\ \mathit{tt}\ \mathbf{else}\ 3$$

Infatti l'espressione è corretta solo se e_0 è falsa.

Semantica dinamica
La valutazione di tale espressione deve coincidere con la valutazione di e_1 se e_0 vale *tt*, mentre deve coincidere con la valutazione di e_2 altrimenti. La prima regola sotto esprime il fatto che la valutazione della guardia e può avvenire anche nel contesto **if** e che questa può richiedere più passi.

$$\frac{\rho \vdash_\Delta \langle e_0, \sigma\rangle \rightarrow_e \langle e_0', \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \sigma\rangle \rightarrow_e \langle \mathbf{if}\ e_0'\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e_0, \sigma\rangle \rightarrow_e \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \sigma\rangle \rightarrow_e \langle e_1, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e_0, \sigma\rangle \rightarrow_e \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \sigma\rangle \rightarrow_e \langle e_2, \sigma\rangle}$$

□

Esercizio 3.6. *Dimostrare la validità della seguente implicazione che garantisce la corrispondenza tra semantica statica e dinamica durante la computazione chiamata* subject reduction*.*

$$\Delta \vdash_I e : \tau \ \wedge \ \rho \vdash_\Delta \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle \ \Rightarrow \Delta \vdash_I e' : \tau$$

SOLUZIONE. Dimostriamo l'implicazione per induzione sulla struttura della categoria sintattica delle espressioni.

BASE
Consideriamo il caso in cui $e = k$, a cui possiamo applicare l'assioma

$$\Delta \vdash_I k : \tau$$

Poiché non c'è nessuna regola della semantica dinamica applicabile alla configurazione $\langle k, \sigma \rangle$, l'ipotesi dell'implicazione è falsa e quindi l'implicazione è vera.
Consideriamo ora $e = x$. La regola di semantica statica che usiamo è

$$\Delta \vdash_I x : \tau, \Delta(x) \in \{\tau, \tau loc\}$$

Inoltre l'unica regola di semantica dinamica applicabile è

$$\rho \vdash_\Delta \langle x, \sigma \rangle \rightarrow_e \langle k, \sigma \rangle$$

dove $k = \rho(x)$ oppure $k = \sigma(\rho(x))$. Poiché $\rho : \Delta$ e $\Delta(x) = \tau$ implica $k = \rho(x)$ con $k \in \tau$ (vedi il Capitolo 4, dichiarazione di identificatori costanti), la semantica statica al valore k non può che assegnare lo stesso tipo τ

$$\Delta \vdash_I k : \tau$$

Se invece $\Delta(x) = \tau loc$ abbiamo che $\rho_x \in Loc_\tau$ (vedi il Capitolo 4, dichiarazione di identificatori modificabili) e quindi $\sigma(\rho(x)) \in \tau$ da cui ancora $\Delta \vdash_I k : \tau$.

PASSO INDUTTIVO
Supponiamo adesso che l'ipotesi induttiva valga per le espressioni e_0 ed e_1

ad esaminiamo il caso in cui $e = e_0 \texttt{ bop } e_1$. Per ipotesi tale espressione è ben formata e $\Delta \vdash_I e : \tau$. Questo, però, avviene solo se sia e_0 che e_1 sono ben formate ed hanno tipi τ_0 e τ_1 tali che $\tau_{bop}(\tau_0, \tau_1) = \tau$. Questo per la regola della semantica statica

$$\frac{\Delta \vdash_I e_0 : \tau_0,\ \Delta \vdash_I e_1 : \tau_1}{\Delta \vdash_I e_0 \texttt{ bop } e_1 : \tau_{bop}(\tau_0, \tau_1)}$$

Vediamo ora la semantica dinamica, l'espressione e può transire nel caso in cui venga valutata e_0 o nel caso in cui venga valutata e_1. Consideriamo solo un caso, l'altro è poi analogo. Applichiamo dunque la regola della semantica dinamica

$$\frac{\rho \vdash_\Delta \langle e_0, \sigma\rangle \rightarrow_e \langle e_0', \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \texttt{ bop } e_1, \sigma\rangle \rightarrow_e \langle e_0' \texttt{ bop } e_1, \sigma\rangle}$$

A questo punto vogliamo dimostrare che l'espressione $e_0' \texttt{ bop } e_1$ è ben formata ed ha tipo $\tau_{bop}(\tau_0, \tau_1)$. Questo deriva immediatamente dall'ipotesi induttiva $e_0' : \tau_0$.
Dobbiamo considerare anche il caso in cui applichiamo l'assioma

$$\rho \vdash_\Delta \langle k \texttt{ bop } k', \sigma\rangle \rightarrow_e \langle k'', \sigma\rangle, k'' = k \texttt{ bop } k'$$

Poiché i due argomenti di `bop` sono ormai completamente valutati, il tipo che assegnamo a k'' è proprio $\tau_{bop}(\tau_0, \tau_1)$.
Del tutto analogo è il caso in cui $e = \texttt{uop } e_0$, applicando l'ipotesi induttiva su e_0. □

Esercizio 3.7. *Data un'espressione e ed un ambiente statico Δ, dimostrare che vale la seguente implicazione*

$$x_0 \in FI(e) \setminus Dom(\Delta) \ \Rightarrow\ \Delta \nvdash_I e : \tau$$

SOLUZIONE. La proprietà $FI(e) \setminus Dom(\Delta) = \emptyset$ garantisce che ogni identificatore utilizzato in e è stato precedentemente dichiarato oppure è stato importato nell'ambiente attraverso delle interfacce. In questo senso generalizziamo il concetto di termine chiuso, solitamente definito solo mediante

FI dicendo che un'espressione è chiusa se $FI(e) = \emptyset$. La generalizzazione di chiusura appena introdotta è necessaria per garantire la compilazione separata dei moduli di un programma. Ciascun modulo suppone dichiarati gli identificatori che importa dagli altri moduli. Nel seguito parleremo di termini chiusi riferendoci alla generalizzazione introdotta.

Eseguiamo ora la dimostrazione per induzione sulla struttura delle espressioni.

BASE
Consideriamo $e = k$. In questo caso si ha che $x_0 \notin FI(e) = \emptyset$ dunque l'implicazione è sempre vera.
Vediamo ora cosa succede se $e = x$; consideriamo $x = x_0$ perché nell'altro caso non ci sono problemi essendo l'implicazione sempre vera. La regola della semantica statica che possiamo applicare è

$$\Delta \vdash_I x : \tau, \;\; \Delta(x) \in \{\tau, \tau loc\}$$

ma poiché $x_0 \notin Dom(\Delta)$ per ipotesi tale regola non è applicabile e dunque non riusciamo ad assegnare il tipo all'espressione.

PASSO INDUTTIVO
Supponiamo ora $e = e_0 \; \texttt{bop} \; e_1$. Per ipotesi $x_0 \in FI(e)$ dunque per definizione di $FI(e) = FI(e_0) \cup FI(e_1)$ x_0 appartiene almeno a uno dei due insiemi dell'unione. Senza perdere di genericità possiamo supporre $x_0 \in FI(e_0)$, e quindi per ipotesi induttiva $\Delta \not\vdash_I e_0 : \tau_0$. Possiamo perciò concludere che anche ad e non riusciamo ad assegnare un tipo poiché l'unica regola che potremmo applicare è

$$\frac{\Delta \vdash_I e_0 : \tau_0, \; \Delta \vdash_I e_1 : \tau_1}{\Delta \vdash_I e_0 \; \texttt{bop} \; e_1 : \tau_{bop}},$$

ma una delle due premesse non è verificata.
Il caso in cui $e = \texttt{uop} \; e_0$ è del tutto analogo. □

Capitolo 4

Dichiarazioni

Le dichiarazioni sono una categoria sintattica dei linguaggi di programmazione i cui elementi sono *elaborati* per produrre dei legami. Un legame è una associazione tra un identificatore ed un valore denotabile (ambiente dinamico) oppure un identificatore ed un tipo denotabile (ambiente statico). Secondo questa definizione possiamo pensare agli ambienti come insiemi di legami.

Le dichiarazioni sono la categoria sintattica dei linguaggi che determina quindi quali sono le occorrenze libere ed in posizione di definizione degli identificatori. Un altro concetto fondamentale per questo scopo è quello di blocco che vedremo nel capitolo sui comandi.

Legato alle dichiarazioni che sono preposte a costruire l'ambiente in cui eseguire i programmi è anche il concetto di scoping. Questo può essere statico o dinamico e la sua scelta può determinare esecuzioni con risultati distinti per lo stesso programma. Riprenderemo questo concetto nel capitolo delle procedure.

Esaminiamo qui le dichiarazioni che sono permesse dalla definizione di IMP fornendo la loro semantica statica e dinamica. Proseguiamo quindi fornendo la soluzione di alcuni esercizi proposti.

4.1 Semantica di DIC

Come nel caso delle espressioni definiamo per prima cosa quali sono gli identificatori liberi ed in posizione di definizione nelle dichiarazioni.

Definizione 4.1 (identificatori liberi). *La funzione $FI : Dic \to Id$ che ad ogni dichiarazione associa l'insieme degli identificatori liberi in essa contenuti è definita per induzione da*

$$
\begin{aligned}
&FI(\mathbf{nil}) = \emptyset \\
&FI(\mathbf{const}\ x : \tau = e) = FI(\mathbf{var}\ x : \tau = e) = FI(e) \\
&FI(d_0; d_1) = FI(d_0\ \mathbf{in}\ d_1) = FI(d_0) \cup (FI(d_1) \setminus DI(d_0)) \\
&FI(d_0\ \mathbf{and}\ d_1) = FI(d_0) \cup FI(d_1) \\
&FI(\rho) = \emptyset
\end{aligned}
$$

Notiamo che la funzione $FI : Dic \to Id$ è definita in termini dell'analoga funzione per le espressioni che per semplicità ha lo stesso nome. Nel testo useremo lo stesso nome per funzioni analoghe su domini diversi per non appesantire la notazione, ma teniamo presente che in realtà abbiamo un insieme di funzioni mutuamente ricorsive.

Definizione 4.2 (identificatori in posizione di definizione). *La funzione $DI : Dic \to Id$ che ad ogni dichiarazione associa l'insieme degli identificatori in posizione di definizione in essa contenuti è definita per induzione da*

$$
\begin{aligned}
&DI(\mathbf{nil}) = \emptyset \\
&DI(\mathbf{const}\ x : \tau = e) = DI(\mathbf{var}\ x : \tau = e) = \{x\} \\
&DI(d_0; d_1) = DI(d_0\ \mathbf{and}\ d_1) = DI(d_0) \cup DI(d_1) \\
&DI(d_0\ \mathbf{in}\ d_1) = DI(d_1) \\
&DI(\rho) = I, \quad \rho : I
\end{aligned}
$$

Definiamo adesso la semantica statica il cui scopo è quello di associare ad ogni dichiarazione un ambiente di tipi che registra il tipo dei valori che verranno associati agli identificatori durante l'esecuzione dei programmi.

$$\Delta \vdash_I \textbf{nil} \ : \ \emptyset$$

$$\frac{\Delta \vdash_I e : \tau}{\Delta_I \vdash \textbf{const}\ x : \tau = e \ : \ [x = \tau]}$$

$$\frac{\Delta \vdash_I e : \tau}{\Delta_I \vdash \textbf{var}\ x : \tau = e \ : \ [x = \tau loc]}$$

$$\frac{\Delta \vdash_I d_0 : \Delta_0, \ \ \Delta[\Delta_0] \vdash_{I \cup I_0} d_1 : \Delta_1}{\Delta \vdash_I d_0; d_1 : \ \Delta_0[\Delta_1]}, \Delta_0 : I_0$$

$$\frac{\Delta \vdash_I d_0 : \Delta_0, \ \ \Delta[\Delta_0] \vdash_{I \cup I_0} d_1 : \Delta_1}{\Delta \vdash_I d_0 \ \textbf{in}\ d_1 : \ \Delta_1}, \Delta_0 : I_0$$

$$\frac{\Delta \vdash_I d_0 : \Delta_0, \ \ \Delta \vdash_I d_1 : \Delta_1}{\Delta \vdash_I d_0 \ \textbf{and}\ d_1 : \ \Delta_0, \Delta_1}, I_0 \cap I_1 = \emptyset$$

Tabella 4.1: Semantica statica per le dichiarazioni di IMP.

Se tale associazione avviene diremo che la dichiarazione è *ben formata*. Le regole sono riportate in Tab. 4.1.

Dopo l'assioma per il **nil**, la prima regola considera le dichiarazioni di costante. Essa ci dice che dobbiamo valutare l'espressione e per ottenere il tipo τ corrispondente al valore espresso da e. Se questo tipo coincide con quello indicato nella dichiarazione stessa si modifica l'ambiente associando all'identificatore x il tipo τ. La regola successiva, invece, è per la dichiarazione di variabili e all'identificatore associa il tipo τloc. Questo indica che l'identificatore corrisponde ad una locazione che può contenere solo un valore di tipo τ. Le tre regole successive servono ad assegnare un ambiente di tipi a dichiarazioni composte. In particolare, la prima serve per le dichiarazioni *sequenziali* per le quali alla fine è visibile l'ambiente modificato da entrambe le dichiarazioni d_0 e d_1 con la proprietà che se

$$\rho \vdash_\Delta \langle \mathbf{nil}, \sigma \rangle \rightarrow_d \langle \emptyset, \sigma \rangle$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_\Delta \langle \mathbf{const}\ x : \tau = e, \sigma \rangle \rightarrow_d \langle \mathbf{const}\ x : \tau = e', \sigma \rangle}$$

$$\rho \vdash_\Delta \langle \mathbf{const}\ x : \tau = k, \sigma \rangle \rightarrow_d \langle [x = k], \sigma \rangle$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_\Delta \langle \mathbf{var}\ x : \tau = e, \sigma \rangle \rightarrow_d \langle \mathbf{var}\ x : \tau = e', \sigma \rangle}$$

$$\rho \vdash_\Delta \langle \mathbf{var}\ x : \tau = k, \sigma \rangle \rightarrow_d \langle [x = l], \sigma[l = k] \rangle, l = New_\tau(L), \sigma : L$$

Tabella 4.2: Semantica dinamica per le dichiarazioni semplici di IMP.

un identificatore è definito sia in d_0 che in d_1 allora è visibile solo l'ultima modifica; per rappresentare l'ambiente risultante si usa la notazione $\Delta[\Delta_0]$ che si legge Δ modificato da Δ_0. La seconda è quella delle dichiarazioni *private* dove le modifiche fatte dalla dichiarazione a sinistra dell'**in** sono visibili solo all'altra dichiarazione e non all'esterno. Quindi l'ambiente finale è solo quello associato alla seconda dichiarazione. Infine la terza regola è quella delle dichiarazioni *simultanee* dove ogni dichiarazione è indipendente dalle modifiche effettuate dall'altra, a condizione che definiscano identificatori differenti (condizione $I_0 \cap I_1 = \emptyset$); in tal caso all'esterno sono visibili entrambi gli ambienti generati dalle dichiarazioni.

In Tab. 4.2 sono riportate le regole per la semantica dinamica che descrivono come vengono elaborate le dichiarazioni semplici considerando che le configurazioni terminali sono del tipo $\langle \rho, \sigma \rangle$. Dopo l'assioma per il **nil**, le prime due regole servono ad associare alla dichiarazione di costante l'ambiente in cui all'identificatore definito viene associato il valore ottenuto dalla valutazione dell'espressione indicata nella dichiarazione stessa. Le successive due, invece, servono per elaborare la dichiarazione di variabile; in tal caso l'ambiente dinamico viene modificato associando all'identificatore una nuova locazione del tipo richiesto, mentre la memo-

ria viene modificata associando alla nuova locazione il valore determinato dalla valutazione dell'espressione presente nella dichiarazione. Il fatto che la locazione l sia nuova ci viene assicurato dal fatto che $l \notin L$, il dominio di σ (vedi Def. 2.4).

In Tab. 4.3 abbiamo le regole che permettono di elaborare le dichiarazioni composte eseguendo una elaborazione da sinistra a destra, e gli assiomi che indicano come i diversi tipi di composizione di dichiarazione generano l'ambiente finale a partire dagli ambienti ottenuti elaborando le due dichiarazioni d_0 e d_1 componenti. Nel caso delle dichiarazioni *sequenziali* iniziamo elaborando d_0. In seguito elaboriamo d_1 nell'ambiente originale modificato con i legami introdotti da d_0. Infine associamo alla dichiarazione composta l'ambiente ottenuto unendo gli ambienti associati alle due dichiarazioni dando priorità, se ci fossero uguali identificatori definiti da entrambe, ai legami introdotti dalla seconda dichiarazione. Per quel che riguarda le dichiarazioni *private* si elabora sempre la dichiarazione d_0 e, anche in tal caso, viene poi elaborata d_1 nell'ambiente modificato da d_0. Alla fine, però, alla dichiarazione composta viene associato solo l'ambiente associato a d_1. Nelle dichiarazioni *simultanee* elaboriamo sempre, per prima, d_0 e poi, però, elaboriamo d_1 ancora nell'ambiente originale, non modificato da d_0. In tal caso alla dichiarazione composta viene associato l'ambiente unione dei due ambienti generati, nell'ipotesi che d_0 *e* d_1 definiscano identificatori diversi $(I_0 \cap I_1 = \emptyset)$.

Per descrivere il comportamento dinamico delle dichiarazioni definiamo una funzione che associa ogni dichiarazione con l'ambiente dinamico che questa genera.

Definizione 4.3 (elaborazione). *La funzione* $Elab : Dic \times Store \to Env$ *che descrive il comportamento dinamico delle dichiarazioni a partire da una memoria* σ *restituendo l' ambiente che esse generano, è definita da*

$$Elab(d, \sigma) = \rho \quad \Leftrightarrow \quad \langle d, \sigma \rangle \to_d^* \langle \rho, \sigma' \rangle$$

A questo punto possiamo utilizzare il comportamento dinamico delle dichiarazioni per definire una equivalenza sugli elementi di Dic.

Definizione 4.4 (equivalenza). *L'equivalenza di dichiarazioni,* $\equiv \subseteq Dic \times Dic$*, è definita da*

$$d_0 \equiv d_1 \quad \Leftrightarrow \quad \forall \sigma.\, (Elab(d_0, \sigma) = Elab(d_1, \sigma))$$

$$\frac{\rho \vdash_\Delta \langle d_0, \sigma\rangle \rightarrow_d \langle d_0', \sigma'\rangle}{\rho \vdash_\Delta \langle d_0; d_1, \sigma\rangle \rightarrow_d \langle d_0'; d_1, \sigma'\rangle}$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle d_1, \sigma\rangle \rightarrow_d \langle d_1', \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0; d_1, \sigma\rangle \rightarrow_d \langle \rho_0; d_1', \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\rho \vdash_\Delta \langle \rho_0; \rho_1, \sigma\rangle \rightarrow_d \langle \rho_0[\rho_1], \sigma\rangle$$

$$\frac{\rho \vdash_\Delta \langle d_0, \sigma\rangle \rightarrow_d \langle d_0', \sigma'\rangle}{\rho \vdash_\Delta \langle d_0 \textbf{ in } d_1, \sigma\rangle \rightarrow_d \langle d_0' \textbf{ in } d_1, \sigma'\rangle}$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle d_1, \sigma\rangle \rightarrow_d \langle d_1', \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0 \textbf{ in } d_1, \sigma\rangle \rightarrow_d \langle \rho_0 \textbf{ in } d_1', \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\rho \vdash_\Delta \langle \rho_0 \textbf{ in } \rho_1, \sigma\rangle \rightarrow_d \langle \rho_1, \sigma\rangle$$

$$\frac{\rho \vdash_\Delta \langle d_0, \sigma\rangle \rightarrow_d \langle d_0', \sigma'\rangle}{\rho \vdash_\Delta \langle d_0 \textbf{ and } d_1, \sigma\rangle \rightarrow_d \langle d_0' \textbf{ and } d_1, \sigma'\rangle}$$

$$\frac{\rho \vdash_\Delta \langle d_1, \sigma\rangle \rightarrow_d \langle d_1', \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0 \textbf{ and } d_1, \sigma\rangle \rightarrow_d \langle \rho_0 \textbf{ and } d_1', \sigma'\rangle}$$

$$\rho \vdash_\Delta \langle \rho_0 \textbf{ and } \rho_1, \sigma\rangle \rightarrow_d \langle \rho_0, \rho_1, \sigma\rangle$$

Tabella 4.3: Semantica dinamica per le dichiarazioni composte di IMP.

4.2 Esercizi

Esercizio 4.1. *Aggiungere ad* IMP *la dichiarazione*

$$d ::= x == y$$

il cui significato è quello di associare x alla stessa locazione di y. Definirne semantica statica e dinamica.

SOLUZIONE. Cerchiamo, prima di tutto, di capire di che cosa si tratta. Dopo tale dichiarazione si ottengono due identificatori differenti che però sono associati alla stessa locazione, si verifica cioè un fenomeno chiamato di *aliasing* a causa del quale possiamo modificare il contenuto di una locazione mediante due identificatori che esistono indipendentemente l'uno dall'altro. Definiamo gli identificatori liberi e quelli in posizione di definizione della nuova dichiarazione:

$$FI(x == y) = \{y\}$$
$$DI(x == y) = \{x\}$$

Semantica statica
Poiché l'identificatore x che stiamo definendo deve riferirsi alla stessa locazione associata a y, dovrà avere lo stesso tipo

$$\frac{\Delta \vdash_I y : \tau loc}{\Delta \vdash_I (x == y) : [x = \tau loc]}$$

Semantica dinamica
L'unica operazione da fare è quella di associare la locazione che corrisponde a y anche a x.

$$\rho \vdash_\Delta \langle x == y, \sigma\rangle \rightarrow_d \langle \rho[x = l], \sigma\rangle \ \ , \ \ l = \rho(y)$$

□

Esercizio 4.2. *Sostituire la dichiarazione* $\mathbf{var}\ x : \tau = e$ *con*

$$d ::= \mathbf{var}\ x : \tau$$

che non esegue l'inizializzazione di x fino al primo assegnamento che coinvolge l'identificatore.

SOLUZIONE. Definiamo prima gli identificatori liberi ed in posizione di definizione nella nuova dichiarazione

$$FI(\mathbf{var}\ x : \tau) = \emptyset$$
$$DI(\mathbf{var}\ x : \tau) = \{x\}$$

Semantica statica
Si nota che quando dichiariamo l'identificatore dobbiamo solo assegnargli un tipo senza nessun controllo

$$\Delta \vdash_I (\mathbf{var}\ x : \tau) : [x = \tau loc]$$

Semantica dinamica
Nella semantica dinamica dobbiamo controllare cosa succede nelle regole in cui vengono aggiornati gli identificatori: dichiarazione e assegnamento. In particolare, in questo caso, cambia solo la dichiarazione.
Per essa infatti dobbiamo trovare un modo per poter individuare a tempo di esecuzione quali identificatori sono stati dichiarati ma non inizializzati, in modo da poter generare un errore, se necessario; usiamo per tale scopo il simbolo $\perp$.

$$\rho \vdash_\Delta \langle \mathbf{var}\ x : \tau, \sigma \rangle \rightarrow_d \langle [x = l], \sigma[l = \perp] \rangle,\ l \in New_\tau(L), \sigma : L$$

Però a questo punto bisogna modificare la regola di valutazione degli identificatori aggiungendo la condizione che x sia stato inizializzato assegnando un valore diverso da $\perp$ alla locazione l corrispondente

$$\rho \vdash_\Delta \langle x, \sigma \rangle \rightarrow_e \langle k, \sigma \rangle,\ \ (k = \rho(x) \vee (l = \rho(x) \wedge k = \sigma(l) \wedge k \neq \perp))$$

□

Esercizio 4.3. *Definire semantica statica e dinamica della dichiarazione*

$$d ::= \mathbf{perv}\ x : \tau$$

con semantica intuitiva che x non può essere dichiarata di nuovo dentro un blocco contenuto in quello in cui appare la dichiarazione d.

SOLUZIONE. Aggiungiamo al linguaggio IMP la dichiarazione fornita dal testo dell'esercizio. Gli identificatori liberi e gli identificatori in posizione di definizione sono:

$$FI(\mathbf{perv}\ x : \tau) = \emptyset$$
$$DI(\mathbf{perv}\ x : \tau) = \{x\}$$

Anche in questo caso dobbiamo essere in grado di distinguere a tempo di esecuzione quali variabili sono state dichiarate normalmente e quali sono **perv**. Intrduciamo allora un nuovo tipo denotabile:

$$DTyp ::= \ldots \mid \tau perv$$

Semantica statica
Vediamo come funziona la semantica statica per la nuova dichiarazione

$$\Delta \vdash_I (\mathbf{perv}\ x : \tau) : [x = \tau perv],\ x \notin I_{|perv}$$

dove $I_{|perv} = \{x \in I \mid \Delta(x) = \tau perv\}$. La condizione indica che l'iterse-zione tra gli identificatori su cui è definito Δ, ristrette al caso in cui siano di tipo $\tau perv$, e x deve essere vuota proprio perché la semantica intuitiva della dichiarazione dice che gli identificatori definiti $\mathbf{perv}$ non possono essere ridefiniti.
Analogamente dobbiamo aggiungere la condizione $x \notin I_{|perv}$ anche alle regole per le dichiarazioni di tipo **const** e **var** per impedire ridefinizioni.
Non dobbiamo modificare le regole per le dichiarazioni composte poiché la modifica dei casi base garantisce la proprietà per tutte le dichiarazioni. Questo possiamo dimostrarlo per induzione sulla forma degli alberi di derivazione. Consideriamo ad esempio la regola per la dichiarazione sequenziale. Negli altri casi è analogo. Se in d_0 e d_1 non vengono ridefiniti gli identificatori dichiarati **perv**, questo non accadrà neppure in $d_0; d_1$. Infatti per ipotesi induttiva $x \notin I_{|perv}$ in $\Delta \vdash_I d_0 : \Delta_0$ per ogni dichiarazione di x in d_0. Ma ogni dichiarazione di x in $\Delta[\Delta_0] \vdash_{I \cup I_0} d_1 : \Delta_1$ verifica per ipotesi induttiva $x \notin (I \cup I_0)_{|perv}$ e quindi le dichiarazioni in $d_0; d_1$ verificano la proprietà.

Semantica dinamica
Consideriamo il linguaggio IMP modificato nell'Esercizio 4.2; a questo

aggiungiamo le nuove regole

$$\rho \vdash_{\Delta} \langle \mathbf{perv}\ x : \tau, \sigma\rangle \rightarrow_d \langle [x = l], \sigma[l = \bot]\rangle,$$
$$\rho(x) \notin Loc_{\tau perv} \ \wedge\ l \in New_{\tau perv}(L), \sigma : L$$

Tale regola associa l'identificatore ad una locazione se esso non era precedentemente stato dichiarato nello stesso blocco come **perv**. A questo punto, avendo modificato i casi base delle dichiarazioni non c'è bisogno di modificare le regole di inferenza per le dichiarazioni composte.
Notiamo che la semantica statica ci assicura che le dichiarazioni che elaboriamo non ridefiniscono gli identificatori di tipo **perv** nei blocchi che le contengono. □

Esercizio 4.4. *Definire semantica statica e dinamica della seguente dichiarazione*

$$d ::= \mathbf{external}\ x : \tau$$

con semantica intuitiva: se dentro un blocco un identificatore è definito **external** *allora viene inizializzato con il valore che aveva fuori dal blocco.*
SOLUZIONE. La prima considerazione da fare è che per poter inserire questo tipo di modifica bisogna fare riferimento al linguaggio IMP modificato nell'Esercizio 4.2 che non inizializza gli identificatori quando li dichiara.

Prima di definire la semantica del nuovo costrutto aggiungiamo un tipo denotabile per distinguere gli identificatori **external** dagli altri

$$DTyp ::= \ldots \mid \tau ext$$

Notiamo inoltre che per questa nuova dichiarazione gli insiemi FI e DI sono gli stessi insiemi definiti per tutti gli altri tipi di dichiarazione.

Semantica statica
La semantica statica per la dichiarazione **external** è

$$\Delta \vdash_I (\mathbf{external}\ x : \tau) : [x = \tau ext]$$

Quindi, in base alla semantica intuitiva fornita dal testo, a livello statico le dichiarazioni **external** si comportano esattamente come le altre associando semplicemente il tipo corretto alla variabile.

Semantica dinamica

$$\rho \vdash_{\Delta} \langle \mathbf{external}\ x : \tau, \sigma\rangle \rightarrow_d \langle [x = l], \sigma[l = \bot]\rangle,$$
$$x \notin Dom(\rho)\ \vee\ \rho(x) = l_{\bot}, l = New_{\tau}(L), \sigma : L$$

cioè se x non è già stata dichiarata (e/o inizializzata) la dichiarazione **external** è equivalente alle altre dichiarazioni. Invece se x era già stata inizializzata la allochiamo nuovamente dandole, però, il valore che aveva precedentemente.

$$\rho \vdash_{\Delta} \langle \mathbf{external}\ x : \tau, \sigma\rangle \rightarrow_d \langle [x = l], \sigma[l = k]\rangle,$$
$$x \in Dom(\rho)\ \wedge\ k = \sigma(l')\ \wedge\ l' = \rho(x) \neq l_{\bot},$$
$$l = New_{\tau}(L), \sigma : L$$

Si noti che in tal modo creiamo una variabile del tutto identica a quella esterna con l'effetto, però, che, uscendo dal blocco, la variabile esterna è esattamente come era prima dell'attivazione del blocco, pur avendo lavorato con il valore che aveva (l'entrata nel blocco simula una chiamata di procedura con passaggio dei parametri per valore; vedi Capitolo 6).
A questo punto, non avendo alterato l'effetto della dichiarazione sui comandi successivi, non ci sono altre modifiche da fare.

Nella semantica intuitiva delle dichiarazioni **external** la frase "viene inizializzato con il valore che aveva fuori dal blocco" si può interpretare come *deve* avere un valore fuori dal blocco anziché *può* avere un valore come abbiamo fatto nella soluzione proposta. Nel caso del *deve* la semantica statica deve garantire che x sia stato già dichiarato al momento della sua dichiarazione **external** e quindi avremmo

$$\Delta \vdash_I (\mathbf{external}\ x : \tau) : [x = \tau ext], \Delta(x) \in \{\tau, \tau loc, \tau ext\}$$

La semantica dinamica può ora essere definita utilizzando solo il secondo assioma. Infatti $x \in Dom(\rho)$ nella condizione del secondo assioma è garantito dalla semantica statica, mentre $\rho(x) = l_{\bot}$ nel primo assioma deve generare un errore a tempo di esecuzione in quanto l'identificatore x non era stato inizializzato.
Questo è un esempio in cui diversi implementatori del costrutto potrebbero fare assunzioni diverse e quindi realizzazioni diverse con la nascita di

numerosi dialetti dello stesso linguaggio. Uno degli scopi della semantica formale dei linguaggi è proprio quello di evitare le ambiguità del linguaggio naturale per garantire che tutte le relizzazioni dello stesso linguaggio siano identiche con notevole beneficio per la portabilità, manutenibilità e chiarezza delle applicazioni. □

Esercizio 4.5. *Estendere* IMP *con il blocco per le espressioni la cui sintassi è*

$$e ::= \textbf{let}\ d\ \textbf{into}\ e$$

con semantica intuitiva: valutare l'espressione e nell'ambiente corrente esteso con i legami generati da d. Definire semantica statica e dinamica. Dimostrare poi che l'espressione sotto è corretta dal punto di vista dei tipi

$$\begin{array}{l}\textbf{let const}\ x : int = 3\\ \textbf{into let const}\ x : int = 5\ \&\ \textbf{const}\ y : int = 6 * x\\ \quad \textbf{into}\ x + y\end{array}$$

dove & può essere ; (per le dichiarazioni sequenziali), **and** *(per le dichiarazioni simultanee) oppure* **in** *(per le dichiarazioni private). Determinare quindi il risultato delle valutazioni dell'espressione nel caso in cui & sia*

$$i.)\ ;\ \ ii.)\ \textbf{and}\ \ iii)\ \textbf{in}$$

fornendo le derivazioni principali delle transizioni delle valutazioni.

SOLUZIONE. Partiamo inserendo la nuova espressione nel nostro linguaggio e definendo gli identificatori liberi e in posizione di definizione

$$\begin{array}{l}FI(\textbf{let}\ d\ \textbf{into}\ e) = (FI(e) \setminus DI(d)) \cup FI(d)\\ DI(\textbf{let}\ d\ \textbf{into}\ e) = DI(d) \cup DI(e)\end{array}$$

Poiché nelle espressioni ci possono essere identificatori in posizione di definizione, bisogna modificare DI in modo che, se applicato ad una espressione, ad un comando o ad una dichiarazione contenente una espressione, tenga conto anche degli identificatori in essa definiti (es. $DI(x := e) = DI(e), \ldots$).

Semantica statica
Dobbiamo valutare il tipo di e nell'ambiente di tipi modificato dalla dichiarazione d e assegnamo il tipo di e alla nuova espressione

$$\frac{\Delta \vdash_I d : \Delta_0,\ \Delta[\Delta_0] \vdash_{I \cup I_0} e : \tau}{\Delta \vdash_I (\mathbf{let}\ d\ \mathbf{into}\ e) : \tau}, \quad \Delta_0 : I_0 \tag{4.1}$$

Semantica dinamica
Con questa nuova espressione anche la valutazione delle espressioni causa modifiche nella memoria generando side-effects. Quindi dobbiamo modificare le regole in Tab. 3.3 considerando che ogni volta che valutiamo una epressione possiamo modificare anche la memoria. Per quanto riguarda il blocco, prima valutiamo la dichiarazione d e poi procediamo con la valutazione di e assegnando, infine, il valore associato ad e all'intera espressione

$$\frac{\rho \vdash_\Delta \langle d, \sigma\rangle \rightarrow_d \langle d', \sigma'\rangle}{\rho \vdash_\Delta \langle \mathbf{let}\ d\ \mathbf{into}\ e, \sigma\rangle \rightarrow_e \langle \mathbf{let}\ d'\ \mathbf{into}\ e, \sigma'\rangle}$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma'\rangle}{\rho \vdash_\Delta \langle \mathbf{let}\ \rho_0\ \mathbf{into}\ e, \sigma\rangle \rightarrow_e \langle \mathbf{let}\ \rho_0\ \mathbf{into}\ e', \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle e, \sigma\rangle \rightarrow_e \langle k, \sigma'\rangle}{\rho \vdash_\Delta \langle \mathbf{let}\ \rho_0\ \mathbf{into}\ e, \sigma\rangle \rightarrow_e \langle k, \sigma'\rangle}, \rho_0 : \Delta_0$$

Vista la semantica della nuova espressione passiamo alla dimostrazione che il frammento fornito è corretto dal punto di vista dei tipi. Per maggiore chiarezza e semplicità conviene dare dei nomi alle varie dichiarazioni e sottoespressioni

$$d_1 = \mathbf{const}\ x : int = 3$$
$$d_2 = \mathbf{const}\ x : int = 5$$
$$d_3 = \mathbf{const}\ y : int = 6 * x$$
$$e_1 = x + y$$
$$e_2 = \mathbf{let}\ d_1\ \mathbf{into}\ \mathbf{let}\ d_2\ \&\ d_3\ \mathbf{into}\ e_1$$

Vediamo ora nei differenti casi cosa succede assumendo di aver già determinato il tipo delle espressioni di inizializzazione.

i. Consideriamo $\& = ;$. Dobbiamo usare la regola (4.1) partendo da un ambiente di tipi iniziale $\Delta : I$ e dalla dichiarazione d_1 i cui legami serviranno per l'analisi di $d_2; d_3$

$$\Delta \vdash_I d_1 : [x = int]$$

Bisogna poi assegnare il tipo ad e_2 in $\Delta[x = int]$ analizzando inizialmente la dichiarazione $d_2;d_3$

$$\frac{\Delta[x = int] \vdash_{I\cup\{x\}} d_2 : [x = int], \quad (\Delta[x = int])[x = int] \vdash_{I\cup\{x\}} d_3 : [y = int]}{\Delta[x = int] \vdash_{I\cup\{x\}} (d_2;d_3) : [x = int][y = int]}$$

Valutiamo e_1 in $\Delta[x = int, y = int]$ $(= \Delta[x = int][y = int]$ visto che le due modifiche riguardano identificatori diversi)

$$\frac{\Delta[x = int, y = int] \vdash_{I\cup\{x,y\}} x : int, \quad \Delta[x = int, y = int] \vdash_{I\cup\{x,y\}} y : int}{\Delta[x = int, y = int] \vdash_{I\cup\{x,y\}} (x + y) : int}$$

Quindi unendo le due valutazioni fatte

$$\frac{\Delta[x = int] \vdash_{I\cup\{x\}} (d_2;d_3) : [x = int, y = int], \quad \Delta[x = int, y = int] \vdash_{I\cup\{x,y\}} e_1 : int}{\Delta[x = int] \vdash_{I\cup\{x\}} e_2 : int}$$

Infine possiamo concludere

$$\frac{\Delta \vdash_I d_1 : [x = int],\ \Delta[x = int] \vdash_{I\cup\{x\}} e_2 : int}{\Delta \vdash_I (\mathbf{let}\ d_1\ \mathbf{into}\ e_2) : int}$$

e quindi dal punto di vista statico non ci sono problemi.

ii. Consideriamo $\& = \mathbf{and}$. Rispetto al caso precedente cambia solo la valutazione dell'espressione e_2 sempre nell'ambiente $\Delta[x = int]$. Valutiamo inizialmente d_2 **and** d_3

$$\frac{\Delta[x = int] \vdash_{I \cup \{x\}} d_2 : [x = int],\ \Delta[x = int] \vdash_{I \cup \{x\}} d_3 : [y = int]}{\Delta[x = int] \vdash_{I \cup \{x\}} (d_2 \ \mathbf{and}\ d_3) : [x = int], [y = int]}$$

Poiché e_1 deve essere valutato nell'ambiente in cui è stato valutato precedentemente, anche in questo caso non ci sono problemi.

iii. Consideriamo $\& = \mathbf{in}$. Anche qui dobbiamo valutare solo e_2 nell'ambiente $\Delta[x = int]$ perché il resto è analogo ai casi precedenti. Valutiamo d_2 **in** d_3

$$\frac{\begin{array}{c}\Delta[x = int] \vdash_{I \cup \{x\}} d_2 : [x = int], \\ (\Delta[x = int])[x = int] \vdash_{I \cup \{x\}} d_3 : [y = int]\end{array}}{\Delta[x = int] \vdash_{I \cup \{x\}} (d_2 \ \mathbf{in}\ d_3) : [y = int]}$$

Ma allora per attribuire un tipo a e_1 dobbiamo valutarla nell'ambiente $(\Delta[x = int])[y = int]$ che è esattamente l'ambiente in cui abbiamo valutato l'espressione nei casi precedenti e dunque anche qui non ci sono problemi.

Dobbiamo infine determinare il risultato della valutazione dell'espressione nei tre diversi casi usando, per semplicità, le stesse notazioni usate prima.

i. Consideriamo $\& = ;$. Valutiamo subito d_1 nell'ambiente iniziale ρ, compatibile con l'ambiente di tipi Δ, e una memoria iniziale σ

$$\rho \vdash_{\Delta} \langle \mathbf{const}\ x = 3, \sigma \rangle \rightarrow_d \langle [x = 3], \sigma \rangle$$

Vediamo ora che valore viene attribuito ad e_2 nell'ambiente $\rho[x = 3]$ compatibile con $\Delta' = \Delta[x = int]$

$$\frac{\rho[x = 3] \vdash_{\Delta'} \langle \mathbf{const}\ x = 5, \sigma \rangle \rightarrow_d \langle [x = 5], \sigma \rangle}{\begin{array}{c}\rho[x = 3] \vdash_{\Delta'} \langle \mathbf{const}\ x = 5\ ;\ \mathbf{const}\ y = 6 * x, \sigma \rangle \rightarrow_d \\ \langle [x = 5]; \mathbf{const}\ y = 6 * x, \sigma \rangle\end{array}}$$

$$\dfrac{\dfrac{(\rho[x=3])[x=5] \vdash_{\Delta'} \langle x, \sigma\rangle \rightarrow_e \langle 5, \sigma\rangle}{(\rho[x=3])[x=5] \vdash_{\Delta'} \langle 6 * x, \sigma\rangle \rightarrow_e \langle 30, \sigma\rangle}}{\rho[x=3] \vdash_{\Delta'} \langle [x=5]; \mathbf{const}\ y = 6 * x, \sigma\rangle \rightarrow_d \langle [x=5][y=30], \sigma\rangle}$$

Dobbiamo ora valutare e_1 in $(\rho[x=3])[x=5, y=30]$ compatibile con $\Delta'' = \Delta'[y = int]$

$$\dfrac{(\rho[x=3])[x=5, y=30] \vdash_{\Delta''} \langle x, \sigma\rangle \rightarrow_e \langle 5, \sigma\rangle}{(\rho[x=3])[x=5, y=30] \vdash_{\Delta''} \langle x + y, \sigma\rangle \rightarrow_e \langle 5 + y, \sigma\rangle}$$

ed infine

$$\dfrac{\dfrac{\dfrac{(\rho[x=3])[x=5, y=30] \vdash_{\Delta''} \langle y, \sigma\rangle \rightarrow_e \langle 30, \sigma\rangle}{(\rho[x=3])[x=5, y=30] \vdash_{\Delta''} \langle 5 + y, \sigma\rangle \rightarrow_e \langle 35, \sigma\rangle}}{\rho[x=3] \vdash_{\Delta'} \langle e_2, \sigma\rangle \rightarrow_e \langle 35, \sigma\rangle}}{\rho \vdash_{\Delta} \langle \mathbf{let}\ d_1\ \mathbf{into}\ e_2, \sigma\rangle \rightarrow_e \langle 35, \sigma\rangle}$$

Da cui il valore finale 35 dell'espressione.

ii. Consideriamo $\& = \mathbf{and}$. Vediamo ora solo le transizioni che sono diverse rispetto al caso precedente; questa volta y non viene valutato nell'ambiente modificato da d_2 poichè $d_2\ e\ d_3$ sono simultanee

$$\dfrac{\dfrac{\rho[x=3] \vdash_{\Delta'} \langle x, \sigma\rangle \rightarrow_e \langle 3, \sigma\rangle}{\rho[x=3] \vdash_{\Delta'} \langle 6 * x, \sigma\rangle \rightarrow_e \langle 18, \sigma\rangle}}{\begin{array}{c}\rho[x=3] \vdash_{\Delta'} \langle [x=5]\ \mathbf{and\ const}\ y = 6 * x, \sigma\rangle \rightarrow_d \\ \langle [x=5], [y=18], \sigma\rangle\end{array}}$$

Valutiamo e_1 in $(\rho[x=3])[x=5, y=18]$ $(= \rho[x=5], [y=18])$ compatibile con $\Delta'' = \Delta'[y = int]$

$$\dfrac{(\rho[x=3])[x=5, y=18] \vdash_{\Delta''} \langle x, \sigma\rangle \rightarrow_e \langle 5, \sigma\rangle}{(\rho[x=3])[x=5, y=18] \vdash_{\Delta''} \langle x + y, \sigma\rangle \rightarrow_e \langle 5 + y, \sigma\rangle}$$

ed infine

$$\dfrac{\dfrac{\dfrac{(\rho[x=3])[x=5,y=18] \vdash_{\Delta''} \langle y,\sigma\rangle \rightarrow_e \langle 18,\sigma\rangle}{(\rho[x=3])[x=5,y=18] \vdash_{\Delta''} \langle 5+y,\sigma\rangle \rightarrow_e \langle 23,\sigma\rangle}}{\rho[x=3] \vdash_{\Delta'} \langle e_2,\sigma\rangle \rightarrow_e \langle 23,\sigma\rangle}}{\rho \vdash_{\Delta} \langle \mathbf{let}\ d_1\ \mathbf{into}\ e_2,\sigma\rangle \rightarrow_e \langle 23,\sigma\rangle}$$

Da cui il valore finale 23.

iii. Consideriamo $\& = \mathbf{in}$. Anche qui vediamo solo le transizioni che cambiano

$$\dfrac{\dfrac{(\rho[x=3])[x=5] \vdash_{\Delta'} \langle x,\sigma\rangle \rightarrow_e \langle 5,\sigma\rangle}{(\rho[x=3])[x=5] \vdash_{\Delta'} \langle 6*x,\sigma\rangle \rightarrow_e \langle 30,\sigma\rangle}}{\rho[x=3] \vdash_{\Delta'} \langle [x=5]\ \mathbf{in\ const}\ y=6*x,\sigma\rangle \rightarrow_d \langle [y=30],\sigma\rangle}$$

Questa volta dobbiamo valutare e_1 in $\rho[x=3][y=30]$, in quanto all'esterno della valutazione della dichiarazione si vede solo l'ambiente generato da d_3

$$\dfrac{(\rho[x=3])[y=30] \vdash_{\Delta''} \langle x,\sigma\rangle \rightarrow_e \langle 3,\sigma\rangle}{(\rho[x=3])[y=30] \vdash_{\Delta''} \langle x+y,\sigma\rangle \rightarrow_e \langle 3+y,\sigma\rangle}$$

ed infine

$$\dfrac{\dfrac{\dfrac{(\rho[x=3])[y=30] \vdash_{\Delta''} \langle y,\sigma\rangle \rightarrow_e \langle 30,\sigma\rangle}{(\rho[x=3])[y=30] \vdash_{\Delta''} \langle 3+y,\sigma\rangle \rightarrow_e \langle 33,\sigma\rangle}}{\rho[x=3] \vdash_{\Delta'} \langle e_2,\sigma\rangle \rightarrow_e \langle 33,\sigma\rangle}}{\rho \vdash_{\Delta} \langle \mathbf{let}\ d_1\ \mathbf{into}\ e_2,\sigma\rangle \rightarrow_e \langle 33,\sigma\rangle}$$

Da cui il valore finale 33.

□

Esercizio 4.6. *Dimostrare che vale la seguente implicazione*

$$\Delta \vdash_I d : \Delta' \ \wedge \ \rho \vdash_\Delta \langle d, \sigma \rangle \rightarrow_d \langle d', \sigma' \rangle \ \Rightarrow \Delta \vdash_I d' : \Delta'$$

Poiché la categoria sintattica delle espressioni è usata da quella delle dichiarazioni, assumiamo il risultato dimostrato nell'Esercizio 3.6.
SOLUZIONE. Dimostriamo l'implicazione per induzione sulla struttura della categoria sintattica delle dichiarazioni.

BASE
Consideriamo il caso in cui $d = \mathbf{const}\ x : \tau = e$. Per ipotesi la dichiarazione è ben formata e $\Delta \vdash_I d : [x = \tau]$. Allora anche l'espressione e è ben formata ed ha tipo τ per la regola della semantica statica

$$\frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I (\mathbf{const}\ x : \tau = e) : [x = \tau]} \tag{4.2}$$

La regola della semantica dinamica che possiamo applicare è

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_\Delta \langle \mathbf{const}\ x : \tau = e, \sigma \rangle \rightarrow_d \langle \mathbf{const}\ x : \tau = e', \sigma \rangle}$$

Vogliamo dunque vedere se la dichiarazione $d' = \mathbf{const}\ x : \tau = e'$ è ben formata e la semantica statica le associa l'ambiente statico $\Delta' = [x = \tau]$. Applicando la regola (4.2) a d' vediamo che è possibile associarle un ambiente solo se e' è ben formata ed ha tipo τ. Ma questo è vero per il risultato dimostrato nell'Esercizio 3.6.
Il caso in cui $d = \mathbf{var}\ x : \tau = e$ è analogo a quello appena visto.

PASSO INDUTTIVO
Supponiamo che l'ipotesi induttiva valga per le dichiarazioni d_0 e d_1 ed esaminiamo la dichiarazione composta $d = d_0; d_1$. Per ipotesi tale dichiarazione è ben formata, quindi la semantica statica riesce ad assegnarle un ambiente statico $\Delta' = \Delta_0[\Delta_1]$. Ma allora lo sono anche d_0 e d_1 perché se almeno una delle due non lo fosse non lo sarebbe neanche d per la regola

$$\frac{\Delta \vdash_I d_0 : \Delta_0,\ \Delta[\Delta_0] \vdash_I d_1 : \Delta_1}{\Delta \vdash_I (d_0; d_1) : \Delta_0[\Delta_1]} \tag{4.3}$$

Per quel che riguarda la semantica dinamica, la transizione che può fare una tale dichiarazione consiste in una transizione di d_0 o di d_1. Vediamo cosa succede solo quando transisce d_0, l'altro caso è analogo.

$$\frac{\rho \vdash_\Delta \langle d_0, \sigma\rangle \rightarrow_d \langle d_0', \sigma'\rangle}{\rho \vdash_\Delta \langle d_0 : d_1, \sigma\rangle \rightarrow_d \langle d_0'; d_1, \sigma'\rangle}$$

Vogliamo vedere allora se $\Delta \vdash_I d_0'; d_1 : \Delta'$. Applicando la regola (4.3) a $d' = d_0'; d_1$ questo vale se sono ben formate sia d_0' che d_1 e generano ambienti statici Δ_0 e Δ_1 rispettivamente. Ma $d_1 : \Delta_1$ per ipotesi e $d_0' : \Delta_0$ per ipotesi induttiva.
Ci rimane da considerare il caso in cui applichiamo l'assioma

$$\rho \vdash_\Delta \langle \rho_0; \rho_1, \sigma\rangle \rightarrow_d \langle \rho_0[\rho_1], \sigma\rangle$$

Poiché $\rho_0 : \Delta_0$ e $\rho_1 : \Delta_1$ abbiamo $\rho_0[\rho_1] : \Delta_0[\Delta_1] = \Delta'$.
Nel caso di dichiarazioni simultanee e private la dimostrazione è analoga.
□

Esercizio 4.7. *Data una dichiarazione d ed un anbiente statico Δ, dimostrare la seguente implicazione*

$$x_0 \in FI(d) \setminus Dom(\Delta) \;\Rightarrow\; \Delta \nvdash_I d : \Delta'$$

assumendo il risultato dell'Esercizio 3.7.

SOLUZIONE. Dimostriamo per induzione, quindi, che non riusciamo ad associare un ambiente alla dichiarazione nel caso ci trovassimo nelle ipotesi dell'implicazione.

BASE
Consideriamo $d = \mathbf{const}\ x : \tau = e$. In tal caso, $x_0 \in FI(d)$ implica che, per definizione di $FI(d)$, $x_0 \in FI(e)$. Infatti $x \neq x_0$ poiché $x \in DI(d)$. L'unica regola che potremmo applicare per determinare l'ambiente statico da associare a d è

$$\frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I \mathbf{const} x : \tau = e \ : \ [x = \tau]}$$

Ma per quanto dimostrato nell'Esercizio 3.7 non riusciamo ad assegnare un tipo ad e e di conseguenza non possiamo applicare la regola poiché la premessa è falsa.
Il caso in cui $d = \mathbf{var}\ x : \tau = e$ è analogo a quello appena dimostrato.

PASSO INDUTTIVO
Consideriamo la dichiarazione composta $d = d_0; d_1$. Per ipotesi $x_0 \in FI(d_0; d_1) = FI(d_0) \cup (FI(d_1) \setminus DI(d_0))$. Questo significa che x_0 appartiene alle variabili libere di almeno una delle due dichiarazioni componenti e nel caso occorra in d_1 non viene dichiarata in d_0. Senza perdere di generalità possiamo supporre che $x_0 \in FI(d_0)$. Per ipotesi induttiva non riusciamo ad associare un ambiente a d_0, e dunque non riusciamo ad associarlo neanche a d poiché l'unica regola utile

$$\frac{\Delta \vdash_I d_0 : \Delta_0,\ \Delta[\Delta_0] \vdash_{I \cup I_0} d_1 : \Delta_1}{\Delta \vdash_I d_0; d_1 : \Delta_0[\Delta_1]},\ \ \Delta_0 : I_0$$

ha una delle due premesse non verificata. Si noti che il fatto che d_1 venga elaborata in un ambiente più ampio non crea problemi alla dimostrazione perché per ipotesi $x_0 \notin DI(d_0)$ e dunque l'ipotesi induttiva vale anche per d_1 che viene elaborata nell'ambiente $\Delta[\Delta_0]$ tale che $x_0 \notin Dom(\Delta[\Delta_0])$.
Nei casi di composizione simultanea e privata la dimostrazione è analoga.
□

Capitolo 5

Comandi

I comandi sono una categoria sintattica dei linguaggi imperativi i cui elementi sono *eseguiti* per aggiornare la memoria della macchina astratta che implementa il supporto del linguaggio. I comandi sono la categoria sintattica che distingue il paradigma imperativo da quello funzionale (e più in generale da tutti gli altri). Infatti la parola imperativo deriva dal latino imperare che vuol dire comandare. Poiché lo scopo dei comandi è quello di aggiornare la memoria, la loro esecuzione non restituisce nessun valore.

I comandi sono costituiti a partire dall'assegnamento che modifica il contenuto di una locazione di memoria e poi contengono costrutti per la composizione ed il controllo del flusso di esecuzione.

Esaminiamo qui i comandi che compaiono nella definizione di IMP fornendo la loro semantica statica e dinamica. Proseguiamo quindi fornendo la soluzione di alcuni esercizi proposti.

5.1 Semantica di COM

Come per le altre categorie sintattiche definiamo, per prima cosa, gli identificatori liberi e in posizione di definizione che occorrono nei comandi.

Definizione 5.1 (identificatori liberi). *La funzione* $FI : Com \to Id$*, che ad ogni comando associa l'insieme degli identificatori liberi in esso*

contenuti, è definita per induzione da

$$
\begin{aligned}
&FI(\mathbf{nil}) = \emptyset \\
&FI(x := e) = FI(e) \cup \{x\} \\
&FI(c_0; c_1) = FI(c_0) \cup FI(c_1) \\
&FI(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1) = FI(c_0) \cup FI(c_1) \cup FI(e) \\
&FI(\mathbf{while}\ e\ \mathbf{do}\ c) = FI(c) \cup FI(e) \\
&FI(d; c) = FI(d) \cup (FI(c) \setminus DI(d))
\end{aligned}
$$

Definizione 5.2 (identificatori in posizione di definizione). *La funzione $DI : Com \to Id$, che ad ogni comando associa l'insieme degli identificatori in posizione di definizione in esso contenuti, è definita per induzione da*

$$
\begin{aligned}
&DI(\mathbf{nil}) = \emptyset \\
&DI(x := e) = \emptyset \\
&DI(c_0; c_1) = DI(c_0) \cup DI(c_1) \\
&DI(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1) = DI(c_0) \cup DI(c_1) \\
&DI(\mathbf{while}\ e\ \mathbf{do}\ c) = DI(c) \\
&DI(d; c) = DI(c) \cup DI(d)
\end{aligned}
$$

Come nei capitoli precedenti, commentiamo adesso la semantica statica spiegando le regole in Tab. 5.1.

In generale il compito della semantica statica è quello di vedere se un comando è *ben formato*; questo avviene nel caso in cui le sue componenti sono ben formate, cioè quando alle espressioni componenti riusciamo ad assegnare un tipo, e alle dichiarazioni riusciamo ad assegnare un ambiente statico; in tal modo deduciamo la ben formatezza del comando. Vediamo, dunque, i comandi uno per volta. Il **nil** non ha sottocomponenti e per questo è sempre ben formato. Per quel che riguarda l'assegnamento, le condizioni perché sia ben formato è che l'espressione e abbia lo stesso tipo dell'identificatore x che deve essere stato dichiarato nell'ambiente Δ.

$$\Delta \vdash_I \textbf{nil} \qquad \frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I x := e},\ \Delta(x) = \tau loc$$

$$\frac{\Delta \vdash_I c_0,\ \Delta \vdash_I c_1}{\Delta \vdash_I c_0; c_1} \qquad \frac{\Delta \vdash_I e : bool,\ \Delta \vdash_I c_0,\ \Delta \vdash_I c_1}{\Delta \vdash_I \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1}$$

$$\frac{\Delta \vdash_I e : bool,\ \Delta \vdash_I c}{\Delta \vdash_I \textbf{while } e \textbf{ do } c} \qquad \frac{\Delta \vdash_I d : \Delta',\ \Delta[\Delta'] \vdash_{I \cup I'} c}{\Delta \vdash_I d; c}, \Delta' : I'$$

Tabella 5.1: Semantica statica per i comandi di IMP.

Invece la sequenzializzazione $c_1; c_2$ è ben formata se lo sono i comandi che la compongono. I comandi **if** e **while**, sono ben formati se l'espressione guardia e, in entrambi, ha tipo booleano e se i sottocomandi che contengono sono ben formati. Infine per la semantica del blocco, si ha che $d; c$ è ben formato se elaborando d nell'ambiente di tipi iniziale Δ riusciamo ad assegnargli un ambiente Δ' e se il comando c è ben formato quando viene valutato in Δ modificato dall'ambiente Δ' generato da d.

Esaminiamo ora le regole della semantica dinamica per i comandi riportate in Tab. 5.2 considerando che un comando può transire in configurazioni del tipo $\langle c, \sigma \rangle$ oppure σ, dove quest'ultima è terminale. Nel caso dell'assegnamento, appena valutata l'espressione e, modifichiamo la memoria associando il valore ottenuto da e alla locazione che l'ambiente ρ associa all'identificatore considerato. Il comando **nil** è, invece, un comando neutro che non ha effetto sulla memoria. Il comando **if** è un comando condizionale, dunque a seconda del valore booleano dell'espressione e viene eseguito il comando del ramo **then** o quello del ramo **else**. In particolare se e vale *tt* allora il sistema transisce nella configurazione che permette di eseguire il comando posizionato dopo **then**; se invece e vale *ff* allora si esegue l'altro comando. Per quel che riguarda il comando $c_1; c_2$ viene inizialmente eseguito c_1. Dopo, nella memoria modificata dal primo comando, viene eseguito il secondo comando c_2. Il **while** è, invece, un comando iterativo

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle k, \sigma\rangle}{\rho \vdash_\Delta \langle x \ := \ e, \sigma\rangle \to_c \sigma[l = k]}, \rho(x) = l \qquad \rho \vdash_\Delta \langle \textbf{nil}, \sigma\rangle \to_c \sigma$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to_c \langle c_1, \sigma\rangle} \qquad \frac{\rho \vdash_\Delta \langle c_0, \sigma\rangle \to_c \langle c_0', \sigma'\rangle}{\rho \vdash_\Delta \langle c_0; c_1, \sigma\rangle \to_c \langle c_0'; c_1, \sigma'\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to_c \langle c_0, \sigma\rangle} \qquad \frac{\rho \vdash_\Delta \langle c_0, \sigma\rangle \to_c \sigma'}{\rho \vdash_\Delta \langle c_0; c_1, \sigma\rangle \to_c \langle c_1, \sigma'\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \textbf{while } e \textbf{ do } c, \sigma\rangle \to_c \langle c; \textbf{while } e \textbf{ do } c, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \textbf{while } e \textbf{ do } c, \sigma\rangle \to_c \sigma}$$

$$\frac{\rho \vdash_\Delta \langle d, \sigma\rangle \to_d \langle d', \sigma'\rangle}{\rho \vdash_\Delta \langle d; c, \sigma\rangle \to_c \langle d'; c, \sigma'\rangle} \qquad \frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle c, \sigma\rangle \to_c \langle c', \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0; c, \sigma\rangle \to_c \langle \rho_0; c', \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle c, \sigma\rangle \to_c \sigma'}{\rho \vdash_\Delta \langle \rho_0; c, \sigma\rangle \to_c \sigma'}, \rho_0 : \Delta_0$$

Tabella 5.2: Semantica dinamica per i comandi di IMP.

che esegue il corpo c fino al momento in cui la condizione in esso contenuta diventa falsa. Quindi, nel caso in cui la valutazione di e sia *tt* allora il comando transisce nella configurazione in cui si ha il comando c da eseguire e nuovamente il **while** che permette di iterare l'esecuzione del comando c. Se la valutazione di e è *ff*, invece, il comando non fa nulla e restistuisce la memoria inalterata. Il comando **while** viene anche chiamato di iterazione indefinita perché non è noto a priori il numero di volte in cui viene eseguito il corpo c. Questo per contrapporlo a comandi stile **for** chiamati di iterazione finita per i quali invece il numero di volte in cui viene eseguito il corpo è noto all'inizio dell'esecuzione (vedi Esercizio 5.8). Infine, per ciò che riguarda il blocco, si valuta inizialmente la dichiarazione d e poi si esegue c nell'ambiente iniziale modificato da quello generato da d.

Per formalizzare il comportamento dinamico dei comandi introduciamo una funzione che restituisce la memoria ottenuta dopo l'esecuzione di un comando.

Definizione 5.3 (esecuzione). *La funzione $Exec : Com \times Store \to Store$, che descrive il comportamento dinamico dei comandi a partire da una memoria σ restituendo la memoria della configurazione finale, è definita da*

$$Exec(c, \sigma) = \sigma' \quad \Leftrightarrow \quad \langle c, \sigma \rangle \to_c^* \sigma'$$

A questo punto possiamo utilizzare il comportamento dinamico dei comandi per definire una equivalenza sugli elementi di Com.

Definizione 5.4 (equivalenza). *L'equivalenza di comandi $\equiv \subseteq Com \times Com$ è definita da*

$$c_0 \equiv c_1 \quad \Leftrightarrow \quad \forall \sigma.\, (Exec(c_0, \sigma) = Exec(c_1, \sigma))$$

5.2 Esercizi

Esercizio 5.1. *Definire una funzione $Mod : Com \to Id$ che associa ad ogni comando l'insieme degli identificatori che possono potenzialmente essere modificati.*

SOLUZIONE. L'unico comando che può alterare il contenuto di una locazione associata ad un identificatore è l'assegnamento. Poiché questo può comparire in molti contesti sintattici distinti, procediamo per induzione strutturale sulla sintassi dei comandi partendo dalle basi **nil** e $x := e$.

$$Mod(\mathbf{nil}) = \emptyset$$
$$Mod(id := e) = \{id\}$$

A questo punto applico il passo induttivo definendo Mod sui comandi composti.

$$Mod(c_1; c_2) = Mod(c_1) \cup Mod(c_2)$$
$$Mod(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = Mod(c_1) \cup Mod(c_2)$$
$$Mod(\mathbf{while}\ e\ \mathbf{do}\ c) = Mod(c)$$
$$Mod(d; c) = Mod(c)$$

Nel caso del blocco, gli identificatori dichiarati in d non sono considerati *modificabili* poiché essi non saranno più visibili all'uscita del blocco. □

Esercizio 5.2. *Definire la semantica statica e dinamica del nuovo comando $x+ := e$ in modo che valga l'equivalenza*

$$(x+ := e) \ \equiv \ (x := x + e)$$

e considerando come valori memorizzabili solo gli interi. Infine dimostrare che effettivamente l'equivalenza vale.
SOLUZIONE. Il primo passo è capire come gli identificatori vengono modificati da questa aggiunta:

$$FI(x+ := e) = \{x\} \cup FI(e)$$
$$DI(x+ := e) = \emptyset$$

Semantica statica
Consideriamo l'ambiente di tipi iniziale Δ definito sull'insieme I di identificatori, in base al quale definiamo una nuova regola per la semantica statica (nei seguenti esercizi tale considerazione verrà sottointesa)

$$\frac{\Delta \vdash_I e : int}{\Delta \vdash_I x+ := e}\ , \Delta(x) = intloc$$

Si noti che consideriamo corretto solo il caso in cui x sia di tipo intero e denoti una locazione.

Semantica dinamica
Vediamo quali regole aggiungere alla semantica dinamica considerando un ambiente ρ compatibile con l'ambiente di tipi Δ generato dalle regole di semantica statica (anche tale considerazione verrà d'ora in poi sottointesa)

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e \langle e', \sigma\rangle}{\rho \vdash_\Delta \langle x+ := e, \sigma\rangle \to_c \langle x+ := e', \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e \langle m, \sigma\rangle}{\rho \vdash_\Delta \langle x+ := e, \sigma\rangle \to_c \langle x+ := m, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle x, \sigma\rangle \to_e \langle n, \sigma\rangle}{\rho \vdash_\Delta \langle x+ := m, \sigma\rangle \to_c \sigma[m'/l]}, n = \sigma(l),\ l = \rho(x), m' = m + n$$

Si noti che è stato eliminato il caso in cui x sia dichiarata costante ($n = \rho(x)$) perché un assegnamento è ben formato soltanto se la sua parte sinistra è dichiarata variabile.

L'esercizio chiede inoltre di dimostrare che effettivamente l'equivalenza sia valida. In base alla definizione di equivalenza di comandi dobbiamo vedere come i due comandi modificano la memoria a partire dalla stessa memoria σ_0 e dallo stesso ambiente ρ iniziali.

i. Consideriamo prima $x+ := e$. La valutazione di e non modifica σ_0, quindi possiamo scrivere

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \to_e^* \langle m, \sigma_0\rangle}{\rho \vdash_\Delta \langle x+ := e, \sigma_0\rangle \to_c \langle x+ := m, \sigma_0\rangle}$$

A questo punto è sufficiente applicare l'ultima regola scritta sopra per ottenere $Exec(x+ := e, \sigma_0) = \sigma_0[m'/l] = \sigma_1$.

ii. Consideriamo ora $x := x + e$. La regola

$$\frac{\rho \vdash_\Delta \langle x, \sigma_0\rangle \to_e \langle n, \sigma_0\rangle}{\rho \vdash_\Delta \langle x := x + e, \sigma_0\rangle \to_c \langle x := n + e, \sigma_0\rangle}, l = \rho(x), n = \sigma_0(l)$$

rappresenta una valutazione parziale della parte destra dell'assegnamento. Essa consente di recuperare il valore contenuto nella locazione l associata all'identificatore x nell'ambiente corrente ρ. Nota che il valore inizialmente associato a x coincide nei due casi poiché partiamo dallo stesso ambiente ρ e la stessa memoria σ_0. Per questo stesso motivo la valutazione di e ottenuta mediante la regola

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \to_e^* \langle m, \sigma_0\rangle}{\rho \vdash_\Delta \langle x := n + e, \sigma_0\rangle \to_c \langle x := n + m, \sigma_0\rangle}$$

genera lo stesso valore m del primo caso. L'ultimo passo di valutazione dell'espressione $x + e$ è descritto dalla regola

$$\frac{\rho \vdash_\Delta \langle m + n, \sigma_0\rangle \to_e \langle m', \sigma_0\rangle}{\rho \vdash_\Delta \langle x := m + n, \sigma_0\rangle \to_c \ \sigma_0[m'/l]}, l = \rho(x), m' = m + n$$

che somma i due interi ottenuti nei passi precedenti. Quindi la modifica operata sulla memoria σ_0 coincide con la modifica operata dal nuovo comando $x+ := e$. Infatti, la memoria risultante è

$$Exec(x := x + e, \sigma_0) = \sigma_0[m'/l] = Exec(x+ := e, \sigma_0)$$

ed è così dimostrata l'equivalenza.

Una soluzione alternativa e forse più immediata dell'esercizio sarebbe quella di definire la semantica dinamica mediante l'assioma

$$\rho \vdash_\Delta \langle x+ := e, \sigma\rangle \to_c \langle x := x + e, \sigma\rangle.$$

L'inconveniente di questa scelta è che non viene conservato formalmente il principio di composizionalità della semantica. Infatti il significato del nuovo costrutto è definito in termini del normale assegnamento che non compare nella sua sintassi. □

Esercizio 5.3. *Definire la semantica statica e dinamica del nuovo comando* $x_1 := (x_2 := (\cdots (x_n := e) \cdots))$ *in modo che valga l'equivalenza*

$$(x_1 := (x_2 := (\cdots (x_n := e) \cdots))) \equiv (x_1 := e')$$

dove $e' = (x_2 := (\cdots (x_n := e) \cdots))$.

SOLUZIONE. Prima di tutto si suppone che il linguaggio su cui lavoriamo contenga le modifiche fatte nell'Esercizio 3.4.
Vediamo allora come tale comando agisce sugli identificatori liberi e in posizione di definizione

$$\begin{aligned} &FI(x_1 := (x_2 := (\cdots (x_n := e) \cdots))) = \\ &\quad \{x_1, x_2, \cdots, x_n\} \cup FI(e) \\ &DI(x_1 := (x_2 := (\cdots (x_n := e) \cdots))) = \emptyset \end{aligned}$$

Semantica statica

$$\frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I (x_1 := (x_2 := (\cdots (x_n := e) \cdots)))}, \Delta(x_i) = \tau loc, i = 1..n$$

Semantica dinamica
Si nota che considerando le espressioni introdotte nell'Esercizio 3.4 non è necessario fare modifiche alla semantica dinamica in quanto il comando diventa un assegnamento di una espressione ad una variabile che è già implementato nel nostro linguaggio. Eventualmente l'unica cosa da evidenziare è l'esecuzione di tale comando:

$$\frac{\rho \vdash_\Delta \langle x_n := e, \sigma\rangle \rightarrow_e^* \langle k, \sigma[k/l]\rangle}{\begin{array}{c}\rho \vdash_\Delta \langle x_1 := (\cdots (x_n := e) \cdots), \sigma\rangle \rightarrow_c \\ \langle x_1 := (\cdots (x_{n-1} := e) \cdots), \sigma[k/l]\rangle\end{array}}, l = \rho(x_n)$$

Nella regola appena definita invece di $x_{n-1} := e$ potevamo mettere $x_{n-1} := k$; in tal modo si otteneva un comando diverso solo nel caso in cui $\exists i > 1.\ x_i \in Id(e)$ ($Id(e)$ indica l'insieme degli identificatori contenuti in e) poiché assegnando ad ogni passo agli identificatori la valutazione di e fatta all'inizio non si tiene conto della successiva modifica eseguita su tale x_i.
□

Esercizio 5.4. *Definire la semantica statica e dinamica dell'assegnamento parallelo (o collaterale)*

$$x_1 := e_1 \textbf{ and } x_2 := e_2 \textbf{ and} \cdots \textbf{and } x_n := e_n$$

dove gli identificatori x_i devono essere tutti differenti. L'esecuzione di questo comando consiste nel valutare prima tutte le espressioni e poi eseguire gli assegnamenti.

SOLUZIONE. Iniziamo subito col definire gli identificatori liberi e in posizione di definizione:

$$\begin{aligned} &FI(x_1 := e_1 \textbf{ and } x_2 := e_2 \textbf{ and} \cdots \textbf{and } x_n := e_n) = \\ &\qquad \{x_1, x_2, \ldots, x_n\} \cup FI(e_1) \cup FI(e_2) \cup \ldots \cup FI(e_n) \\ &DI(x_1 := e_1 \textbf{ and } x_2 := e_2 \textbf{ and} \cdots \textbf{and } x_n := e_n) = \emptyset \end{aligned}$$

Semantica statica
Dobbiamo verificare che il tipo associato a ciascun identificatore x_i sia una locazione che può memorizzare valori del tipo dell'espressione corrispondente e_i.

$$\frac{\Delta \vdash_I e_1 : \tau_1, \Delta \vdash_I e_2 : \tau_2, \ldots, \Delta \vdash_I e_n : \tau_n}{\Delta \vdash_I x_1 := e_1 \textbf{ and} \cdots \textbf{and } x_n := e_n}, \Delta(x_i) = \tau_i loc,\ i = 1..n$$

Semantica dinamica
Per eseguire il nuovo comando prima valutiamo tutte le espressioni e_i

$$\frac{\begin{array}{c}\rho \vdash_\Delta \langle e_1, \sigma\rangle \to_e^* \langle m_1, \sigma\rangle \\ \vdots \\ \rho \vdash_\Delta \langle e_n, \sigma\rangle \to_e^* \langle m_n, \sigma\rangle\end{array}}{\begin{array}{r}\rho \vdash_\Delta \langle x_1 := m_1 \textbf{ and} \cdots \textbf{and } x_n := e_n, \sigma\rangle \to_c \\ \langle x_1 := m_1 \textbf{ and} \cdots \textbf{and } x_n := m_n, \sigma\rangle\end{array}}$$

e poi assegnamo i valori ottenuti ai corrispondenti identificatori $\forall i \in [1, n]$

$$\frac{\rho \vdash_\Delta \langle x_i := m_i, \sigma\rangle \to_c \sigma[m_i/l_i]}{\begin{array}{c}\rho \vdash_\Delta \langle x_i := m_i \textbf{ and } \cdots \textbf{ and } x_n := m_n, \sigma\rangle \to_c \\ \langle x_{i+1} := m_{i+1} \textbf{ and } \cdots \textbf{ and } x_n := m_n, \sigma_i\rangle\end{array}}, \begin{cases} \sigma_i = \sigma[m_i/l_i] \\ l_i = \rho(v_i) \end{cases}$$

Si noti che, in assenza di specifiche più precise, nello scrivere le regole abbiamo scelto di valutare le espressioni ed eseguire gli assegnamenti da sinistra a destra. Di fatto questo può, in certe condizioni, essere differente da una valutazione casuale degli assegnamenti descritta dalla regola di inferenza

$$(*)\ \frac{\rho \vdash_\Delta \langle e_i, \sigma\rangle \to_e^* \langle m_i, \sigma\rangle}{\begin{array}{c}\rho \vdash_\Delta \langle x_1 := e_1 \textbf{ and } \cdots \textbf{ and } x_i := e_i \textbf{ and } \cdots \textbf{ and } x_n := e_n, \sigma\rangle \to_c \\ \langle x_1 := e_1 \textbf{ and } \cdots \textbf{ and } x_i := m_i \textbf{ and } \cdots \textbf{ and } x_n := e_n, \sigma\rangle\end{array}}$$

In particolare tali regole non sono equivalenti nel caso in cui il nostro linguaggio ammetta espressioni con side-effects ed $\exists i, j.\ Mod(e_i) \cap Id(e_j) \neq \emptyset$, dove Mod è l'estensione alle espressioni con effetti collaterali della funzione definita nell'Esercizio 5.1. Consideriamo ad esempio la dichiarazione

$$x := (y := 1) \textbf{ and } y := y + 1$$

assumendo che inizialmente l'ambiente ρ associ y ad una locazione l_y contenente 4. Con la valutazione da sinistra a destra otteniamo alla fine $x = 1$ e $y = 2$. Inizialmente valutiamo $(y := 1)$ che ha come effetto collaterale quello di assegnare il valore 1 alla locazione l_y.

$$\frac{\rho \vdash_\Delta \langle y := 1, \sigma_0[4/l_y]\rangle \to_c \sigma_0[1/l_y]}{\begin{array}{c}\rho \vdash_\Delta \langle x := (y := 1) \textbf{ and } y := y + 1, \sigma_0\rangle \to_c \\ \langle x := 1 \textbf{ and } y := y + 1, \sigma_0[1/l_y]\rangle\end{array}}$$

Il secondo passo consiste nel valutare l'espressione $y + 1$ nella memoria modificata nel passo precedente e quindi con il valore 1 contenuto in l_y.

$$\frac{\dfrac{\rho \vdash_\Delta \langle y, \sigma_0[1/l_y]\rangle \to_e \langle 1, \sigma_0[1/l_y]\rangle}{\rho \vdash_\Delta \langle y + 1, \sigma_0[1/l_y]\rangle \to_e \langle 2, \sigma_0[1/l_y]\rangle}}{\rho \vdash_\Delta \langle x := 1 \textbf{ and } y := y + 1, \sigma_0\rangle \to_c \langle x := 1 \textbf{ and } y := 2, \sigma_0[1/l_y]\rangle}$$

A questo punto eseguendo gli assegnamenti otteniamo che x varrà 1 e y 2. Ora valutiamo da destra a sinistra e dimostriamo che alla fine avremo $x = 1$ e $y = 5$. In questo caso valutiamo quindi $y + 1$ a partire da una configurazione in cui l_y contiene 4.

$$\dfrac{\dfrac{\rho \vdash_\Delta \langle y, \sigma_0[4/l_y]\rangle \to_e \langle 4, \sigma_0[4/l_y]\rangle}{\rho \vdash_\Delta \langle y+1, \sigma_0[4/l_y]\rangle \to_e \langle 5, \sigma_0[4/l_y]\rangle}}{\begin{array}{c}\rho \vdash_\Delta \langle x := (y := 1)\ \mathbf{and}\ y := y+1, \sigma_0\rangle \to_c \\ \langle x := (y := 1)\ \mathbf{and}\ y := 5, \sigma_0[4/l_y]\rangle\end{array}}$$

La valutazione di $y := 1$ coincide con il caso precedente.

$$\dfrac{\rho \vdash_\Delta \langle y := 1, \sigma_0[4/l_y]\rangle \to_c \sigma_0[1/l_y]}{\rho \vdash_\Delta \langle x := (y := 1)\ \mathbf{and}\ y := 5, \sigma_0\rangle \to_c \langle x := 1\ \mathbf{and}\ y := 5, \sigma_0[1/l_y]\rangle}$$

Come ci aspettavamo, effettivamente x vale ancora 1, ma y vale 5.
Si noti infine che, poiché la regola (*) consente entrambe le derivazioni, essa introduce non determinismo nell'esecuzione dell'assegnamento. □

Esercizio 5.5. *Definire semantica statica e dinamica del nuovo comando*

$$\mathbf{if}\ e\ \mathbf{then}\ c.$$

Mostrare poi che il nuovo comando può essere simulato mediante il comando condizionale di IMP.

SOLUZIONE. Analogamente a ciò che è stato fatto negli esercizi precedenti vediamo subito come variano gli insiemi FI e DI degli identificatori liberi e in posizione di definizione:

$$FI(\mathbf{if}\ e\ \mathbf{then}\ c) = FI(e) \cup FI(c)$$
$$DI(\mathbf{if}\ e\ \mathbf{then}\ c) = DI(c)$$

Semantica statica
Come per il comando condizionale di IMP dobbiamo solo controllare che il tipo dell'espressione e sia un booleano e che il comando c sia ben formato.

$$\dfrac{\Delta \vdash_I e : \mathit{bool}, \Delta \vdash_I c}{\Delta \vdash_I \mathbf{if}\ e\ \mathbf{then}\ c}$$

Semantica dinamica
Per eseguire il nuovo comando dobbiamo prima valutare l'espressione e e poi passare ad eseguire c se otteniamo il valore tt

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle tt, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c, \sigma\rangle \to_c \langle c, \sigma\rangle}$$

oppure terminare l'esecuzione del comando se e vale *ff*

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle ff, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c, \sigma\rangle \to_c \sigma}$$

Per simulare questo costrutto con il costrutto condizionale di IMP è sufficiente scrivere quello nuovo in funzione di quello vecchio e poi dimostrarne l'equivalenza mediante $Exec$. Consideriamo

$$\mathbf{if}\ e\ \mathbf{then}\ c \ \equiv\ \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}$$

Per dimostrare l'equivalenza consideriamo una memoria iniziale σ_0. Abbiamo il caso in cui e vale *tt* e quello in cui e vale *ff* per ciascuno dei due comandi.

i. Per eseguire $\mathbf{if}\ e\ \mathbf{then}\ c$, prima valutiamo e

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \to_e^* \langle tt, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c, \sigma_0\rangle \to_c \langle c, \sigma_0\rangle}$$

e poi eseguiamo il comando c

$$\rho \vdash_\Delta \langle c, \sigma_0\rangle \to_c^* \sigma_1.$$

Dunque se e è valutata *tt*, $Exec(\mathbf{if}\ e\ \mathbf{then}\ c, \sigma_0) = \sigma_1$. Nel caso e rappresenti *ff* la memoria finale coincide con quella iniziale, infatti

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \to_e^* \langle ff, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c, \sigma_0\rangle \to_c \sigma_0}$$

Dunque $Exec(\mathbf{if}\ e\ \mathbf{then}\ c, \sigma_0) = \sigma_0$.

ii. Per eseguire $\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}$ prima valutiamo e. Se otteniamo il valore tt selezioniamo il comando c

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \to_e^* \langle tt, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}, \sigma_0\rangle \to_c \langle c, \sigma_0\rangle}$$

e lo eseguiamo a partire dalla stessa memoria iniziale σ_0 del caso precedente ottenendo quindi anche la stessa memoria finale

$$\rho \vdash_\Delta \langle c, \sigma_0\rangle \to_c^* \sigma_1.$$

Dunque se e vale *tt*, $Exec(\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}, \sigma_0) = \sigma_1$. Se invece e vale *ff* selezioniamo il comando $\mathbf{nil}$

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \to_e^* \langle \mathit{ff}, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}, \sigma_0\rangle \to_c \langle \mathbf{nil}, \sigma_0\rangle}$$

la cui esecuzione non produce modifiche sulla memoria

$$\rho \vdash_\Delta \langle \mathbf{nil}, \sigma_0\rangle \to_c \sigma_0.$$

Dunque se e rappresenta *ff*, $Exec(\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}, \sigma_0) = \sigma_0$, e quindi

$$\forall e. Exec(\mathbf{if}\ e\ \mathbf{then}\ c, \sigma_0) = Exec(\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else\ nil}, \sigma_0).$$

□

Esercizio 5.6. *Dimostrare che l'equivalenza*

$$\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \ \equiv\ \mathbf{if}\ e\ \mathbf{then}\ c_0;\ \mathbf{if\ not}\ e\ \mathbf{then}\ c_1$$

vale se e *vale* ff*, oppure se* $Id(e) \cap Mod(c_0) = \emptyset$*, dove* Mod *è definita come nell'Esercizio 5.1.*

SOLUZIONE. Per dimostrare tale equivalenza dobbiamo vedere cosa succede applicando le regole di inferenza ai due diversi casi partendo dalla stessa memoria iniziale σ_0. Visto che il problema richiede di dimostrare come prima cosa che l'equivalenza vale se e vale *ff*, dividiamo la dimostrazione in base al valore di e.

i. Consideriamo che il valore associato a e sia falso

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathit{ff}, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma_0\rangle \rightarrow_c \langle c_1, \sigma_0\rangle}$$

questo per la definizione delle regole di inferenza per l'**if**. Perciò se $\rho \vdash_\Delta \langle c_1, \sigma_0\rangle \rightarrow_c^* \sigma_1$, si avrà $Exec(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma_0) = \sigma_1$. Vediamo ora cosa accade nell'altro comando.

$$\frac{\dfrac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathit{ff}, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0, \sigma_0\rangle \rightarrow_c \sigma_0}}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0; \mathbf{if\ not}\ e\ \mathbf{then}\ c_1, \sigma_0\rangle \rightarrow_c \langle \mathbf{if\ not}\ e\ \mathbf{then}\ c_1, \sigma_0\rangle}$$

Poiché gli identificatori di e non sono stati modificati (abbiamo ancora la memoria σ_0), avremo

$$\frac{\dfrac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathit{ff}, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{not}\ e, \sigma_0\rangle \rightarrow_e \langle \mathit{tt}, \sigma_0\rangle}}{\rho \vdash_\Delta \langle \mathbf{if\ not}\ e\ \mathbf{then}\ c_1, \sigma_0\rangle \rightarrow_c \langle c_1, \sigma_0\rangle}$$

Come nel caso precedente arriviamo alla configurazione $\langle c_1, \sigma_0\rangle$ e quindi alla fine terminiamo nella stessa memoria. Possiamo dunque affermare che se il valore booleano della guardia e è falso i comandi sono equivalenti.

ii. Vediamo ora il caso in cui e vale *tt* e $Id(e) \cap Mod(c_0) = \emptyset$.

$$\frac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathit{tt}, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma_0\rangle \rightarrow_c \langle c_0, \sigma_0\rangle}$$

questo per la definizione delle regole di inferenza per l'**if**. Perciò se $\rho \vdash_\Delta \langle c_0, \sigma_0\rangle \rightarrow_c^* \sigma_1$, si avrà $Exec(\mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma_0) = \sigma_1$. Vediamo adesso l'altro caso

$$\frac{\dfrac{\rho \vdash_\Delta \langle e, \sigma_0\rangle \rightarrow_e^* \langle \mathit{tt}, \sigma_0\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0, \sigma_0\rangle \rightarrow_c \langle c_0, \sigma_0\rangle}}{\begin{array}{c}\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0; \mathbf{if\ not}\ e\ \mathbf{then}\ c_1, \sigma_0\rangle \rightarrow_c \\ \langle c_0; \mathbf{if\ not}\ e\ \mathbf{then}\ c_1, \sigma_0\rangle\end{array}}$$

Per quello che è stato detto nel comando precedente, vale la seguente

$$\frac{\rho \vdash_\Delta \langle c_0, \sigma_0\rangle \to_c^* \sigma_1}{\rho \vdash_\Delta \langle c_0; \textbf{if not } e \textbf{ then } c_1, \sigma_0\rangle \to_c \langle \textbf{if not } e \textbf{ then } c_1, \sigma_1\rangle}$$

A questo punto notiamo che il comando c_0 non altera i valori associati agli identificatori contenuti in e poiché $Id(e) \cap Mod(c_0) = \emptyset$. Quindi

$$\frac{\rho \vdash_\Delta \langle \textbf{not } e, \sigma_1\rangle \to_e^* \langle \mathit{ff}, \sigma_1\rangle}{\rho \vdash_\Delta \langle \textbf{if not } e \textbf{ then } c_1, \sigma_1\rangle \to_c \sigma_1}$$

che conclude la dimostrazione.

□

Esercizio 5.7. *Definire semantica statica e dinamica del nuovo comando*

$$\begin{array}{l}\textbf{if } e_1 \textbf{ then } c_1\\ \textbf{elseif } e_2 \textbf{ then } c_2\\ \vdots\\ \textbf{elseif } e_n \textbf{ then } c_n\\ \textbf{else } c_{n+1}\end{array}$$

inserito al posto del costrutto condizionale tradizionale. Mostrare poi che il nuovo comando può essere definito in termini del comando condizionale di IMP.

SOLUZIONE. Definiamo, come al solito, gli insiemi FI e DI per il nuovo comando:

$$\begin{array}{l} FI(\textbf{if } e_1 \textbf{ then } c_1 \textbf{ elseif } e_2 \textbf{ then } c_2 \ \ldots \ \textbf{else } c_{n+1}) =\\ \quad FI(c_1) \cup FI(c_2) \cup \ldots \cup FI(c_{n+1}) \cup\\ \quad FI(e_1) \cup FI(e_2) \cup \ldots \cup FI(e_n)\\ DI(\textbf{if } e_1 \textbf{ then } c_1 \textbf{ elseif } e_2 \textbf{ then } c_2 \ \ldots \ \textbf{else } c_{n+1}) =\\ \quad DI(c_1) \cup DI(c_2) \cup \ldots \cup DI(c_{n+1})\end{array}$$

Semantica statica
Dobbiamo verificare che tutte le guardie e_i sono di tipo booleano e che i comandi c_i sono ben formati nell'ambiente statico in cui valuto il nuovo comando

$$\frac{\Delta \vdash_I e_1 : \mathit{bool}, \ldots, \Delta \vdash_I e_n : \mathit{bool}, \Delta \vdash_I c_1, \ldots, \Delta \vdash_I c_{n+1}}{\Delta \vdash_I \textbf{if } e_1 \textbf{ then } c_1 \textbf{ elseif } e_2 \textbf{ then } c_2 \ \ldots \ \textbf{else } c_{n+1}}$$

Semantica dinamica
Iniziamo a valutare le guardie booleane partendo dalla prima. Se questa è vera eseguiamo il comando corrispondente e terminiamo. Altrimenti scartiamo il ramo **then** ed eseguiamo il sottocomando

$$\textbf{if } e_2 \textbf{ then } c_2 \textbf{ elseif } \ldots \textbf{ else } c_{n+1}$$

Perciò le regole generica sono

$$\frac{\rho \vdash_\Delta \langle e_i, \sigma\rangle \rightarrow_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \textbf{if } e_i \textbf{ then } c_i \ \ldots \ \textbf{else } c_{n+1}, \sigma\rangle \rightarrow_c \langle c_i, \sigma\rangle}, \ i \in [1, n]$$

$$\frac{\rho \vdash_\Delta \langle e_i, \sigma\rangle \rightarrow_e^* \langle \mathit{ff}, \sigma\rangle}{\begin{array}{c}\rho \vdash_\Delta \langle \textbf{if } e_i \textbf{ then } c_i \ \ldots \ \textbf{else } c_{n+1}, \sigma\rangle \rightarrow_c \\ \langle \textbf{if } e_{i+1} \textbf{ then } c_{i+1} \ \ldots \ \textbf{else } c_{n+1}, \sigma\rangle\end{array}}, \ i \in [1, n]$$

Data la semantica del costrutto possiamo dimostare che è simulabile dal costrutto tradizionale semplicemente scrivendolo

$$\begin{array}{l}\textbf{if } e_1 \textbf{ then } c_1 \textbf{ else} \\ \quad \textbf{if } e_2 \textbf{ then } c_2 \textbf{ else} \\ \quad \vdots \\ \quad\quad \textbf{if } e_n \textbf{ then } c_n \textbf{ else } c_{n+1}\end{array}$$

Come si può notare il cambiamento è solo sintattico dunque non è necessaria una esplicita dimostrazione di equivalenza. □

Esercizio 5.8. *Definire semantica statica e dinamica del nuovo comando*

$$\mathbf{do}\ e\ \mathbf{times}\ c$$

con la semantica intuitiva di ripetere c fino a quando l'espressione e è diversa da 0. Tale comando è detto stile **for** *in quanto può eseguire c solo un numero finito di volte determinato a priori.*

SOLUZIONE. Definiamo prima di tutto gli insiemi FI e DI:

$$FI(\mathbf{do}\ e\ \mathbf{times}\ c) = FI(e) \cup FI(c)$$
$$DI(\mathbf{do}\ e\ \mathbf{times}\ c) = DI(c)$$

Semantica statica
Per come è definita la semantica intuitiva, il comando è ben formato se l'espressione e è di tipo intero e se il comando c è ben formato

$$\frac{\Delta \vdash_I e : int, \Delta \vdash_I c}{\Delta \vdash_I \mathbf{do}\ e\ \mathbf{times}\ c}$$

Semantica dinamica
Per eseguire tale comando valutiamo, per prima cosa, l'espressione determinando così il valore numerico m ad essa corrispondente. A questo punto eseguiamo m volte il comando c decrementando, ad ogni esecuzione di c il valore m. Poiché la semantica statica non ci assicura che il valore di e sia positivo, dobbiamo inserire nella semantica dinamica un controllo a tempo di esecuzione ($m > 0$). Infatti, se m fosse negativo il comando non terminerebbe mai.

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e \langle e', \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{do}\ e\ \mathbf{times}\ c, \sigma\rangle \to_c \langle \mathbf{do}\ e'\ \mathbf{times}\ c, \sigma\rangle}$$

$$\rho \vdash_\Delta \langle \mathbf{do}\ m\ \mathbf{times}\ c, \sigma\rangle \to_c \langle c; \mathbf{do}\ m-1\ \mathbf{times}\ c, \sigma\rangle, m > 0$$

$$\rho \vdash_\Delta \langle \mathbf{do}\ 0\ \mathbf{times}\ c, \sigma\rangle \to_c \sigma$$

Notiamo che si poteva scegliere di valutare l'epressione ad ogni passo eseguendo il decremento, non su m, ma su e. Questa scelta però è equivalente alla precedente solo nel caso in cui sia verificata la condizione $Id(e) \cap Mod(c) = \emptyset$, dove $Mod(c)$ è la funzione che restituisce gli identificatori che possono essere modificati da c definita nell'Esercizio 5.1 estesa opportunamente al presente comando. Infatti, in caso contrario, la terminazione del ciclo non sarebbe garantita perchè l'esecuzione di c potrebbe alterare la condizione di terminazione non facendo mai arrivare il valore di e a 0. □

Esercizio 5.9. *Definire semantica statica e dinamica del nuovo comando di iterazione*

$$
\begin{array}{l}
\textbf{loop} \\
c_1 \\
\textbf{when } e_1 \textbf{ do } c_1' \textbf{ exit} \\
c_2 \\
\vdots \\
\textbf{when } e_n \textbf{ do } c_n' \textbf{ exit} \\
c_{n+1} \\
\textbf{repeat}
\end{array}
$$

con la semantica intuitiva di eseguire ripetutamente $c_1, c_2, \ldots, c_{n+1}$ fino a quando non sia verificata una delle condizioni e_i di uscita. A questo punto si esegue il corrispondente comando e si termina.

SOLUZIONE. Per semplicità identifichiamo l'intero nuovo comando con $\overline{c}_1$. Definiamo allora gli insiemi FI e DI:

$$FI(\overline{c}_1) = FI(c_1) \cup \bigcup_{i=1}^{n}((FI(c_{i+1}) \cup FI(c_i') \cup FI(e_i)) \setminus DI(c_i))$$

$$DI(\overline{c}_1) = \bigcup_{i=1}^{n}(DI(c_i) \cup DI(c_i')) \cup DI(c_{n+1})$$

Semantica statica
Perché il comando sia ben formato tutte le espressioni devono essere booleane e i comandi devono essere ben formati

$$\frac{\Delta \vdash_I \forall i \in [1,n]\,.\,e_i : \mathit{bool}, \Delta \vdash_I \forall i \in [1,n]\,.\,c'_i, \Delta \vdash_I \forall i \in [1,n+1]\,.\,c_i}{\Delta \vdash_I \overline{c}_1}$$

Semantica dinamica
L'esecuzione del comando di **loop** consiste nell'eseguire, ad ogni passo, un comando c_i, valutare una condizione e_i e, se questa è falsa, andare avanti, altrimenti eseguire il comando c'_i e uscire dal ciclo. I comandi c_i devono essere eseguiti secondo l'ordine sequenziale in cui compaiono nel corpo del **loop**. Quindi per permettere di eseguire il comando giusto e per permettere di poter ripetere l'intero **loop** se nessuna condizione risulta vera dobbiamo portarci dietro, nelle regole, rispettivamente il comando originale, privato dei comandi già eseguiti e delle condizioni già controllate ($\overline{c}_i$), e l'intero comando originale ($\overline{c}_1$). Utilizzeremo la notazione

$$\overline{c}_i = c_i; \mathbf{when}\ e_i\ \mathbf{do}\ c'_i\ \mathbf{exit}\ c_{i+1}\ \cdots\ c_{n+1},\ i \in [1,n]$$

assumendo $\overline{c}_{n+1} = c_{n+1}$. Quindi interpretiamo i componenti del **loop** come una composizione sequenziale di comandi.

$$\rho \vdash_\Delta \langle \overline{c}_1, \sigma\rangle \to_c \langle c_1; \mathbf{when}\ e_1\ \mathbf{then}\ c'_1\ \mathbf{exit}; \overline{c}_2; \overline{c}_1, \sigma\rangle$$

$$\rho \vdash_\Delta \langle \overline{c}_i, \sigma\rangle \to_c \langle c_i; \mathbf{when}\ e_i\ \mathbf{then}\ c'_i\ \mathbf{exit}; \overline{c}_{i+1}, \sigma\rangle$$

Abbiamo introdotto due regole per distinguere il primo passo dai successivi per evitare di accumulare copie di c_1 in coda. Per poter specificare la semantica del **loop** ci rimane quindi da definire la semantica di **when**. Questa ci dice che se la condizione e_i è vera, eseguiamo c'_i nella memoria corrente e scartiamo la continuazione del **loop** ($\overline{c}_{i+1}$) e la copia dell'intero comando ($\overline{c}_1$) delimitata da **loop** e **repeat** che abbiamo introdotto con la prima regola.

$$\frac{\rho \vdash_\Delta \langle e_i, \sigma\rangle \to_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{when}\ e_i\ \mathbf{do}\ c'_i\ \mathbf{exit}; \overline{c}_{i+1}; \overline{c}_1, \sigma\rangle \to_c \langle c'_i, \sigma\rangle}$$

Nel caso in cui la condizione e_i sia falsa, scartiamo il comando corrispondente e procediamo con la continuazione $\overline{c}_{i+1}$ del **loop**.

$$\frac{\rho \vdash_\Delta \langle e_i, \sigma\rangle \to_e^* \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{when}\ e_i\ \mathbf{do}\ c_i'\ \mathbf{exit}; \overline{c}_{i+1}; \overline{c}_1, \sigma\rangle \to_c \langle \overline{c}_{i+1}; \overline{c}_1, \sigma\rangle}$$

□

Esercizio 5.10. *Si consideri la possibilità di dichiarare identificatori senza allocarli fino a quando non viene fatta una richiesta esplicita. Si dia allora semantica statica e dinamica al nuovo comando*

$$\mathbf{allocate}(x)$$

in modo che il seguente frammento di programma

$$\begin{array}{l}\mathbf{var}\ x : int;\\ \mathbf{begin}\\ x := 1;\ \mathbf{allocate}(x)\\ \mathbf{end}\end{array}$$

dia errore a tempo di esecuzione.

SOLUZIONE. Definiamo gli identificatori liberi e in posizione di definizione del nuovo comando. Poiché non possiamo allocare un identificatore prima di averlo dichiarato, abbiamo

$$FI(\mathbf{allocate}(x)) = \{x\} \qquad DI(\mathbf{allocate}(x)) = \emptyset$$

Notiamo che allocando dinamicamente gli identificatori bisogna trovare un modo per distinguere quegli identificatori che sono stati dichiarati, ma non ancora allocati. Aggiungiamo per questo un simbolo alle locazioni da associare agli identificatori quando vengono dichiarati in modo da identificare a tempo di esecuzione se un identificatore è stato allocato oppure no. Il nuovo insieme di locazioni è

$$Loc' = Loc \cup \{l_\perp\}.$$

Per poter evidenziare la situazione di errore nel caso in cui venga utilizzata una variabile non ancora allocata definiamo $\sigma(l_\perp) = \perp$.

Semantica statica
Dal punto di vista statico il nuovo comando è corretto se l'identificatore x che si vuole allocare è stato precedentemente dichiarato e quindi compare nel dominio di Δ.

$$\Delta \vdash_I \mathbf{allocate}(x), \;\; x \in I$$

Semantica dinamica
Per la semantica dinamica del nuovo comando, dobbiamo modificare anche la semantica dinamica dell'assegnamento e della dichiarazione di variabile.

$$\rho \vdash_\Delta \langle \mathbf{allocate}(x), \sigma \rangle \rightarrow_c \langle [x = l]; \mathbf{nil}, \sigma \rangle, \begin{cases} \rho(x) = l_\perp \\ \Delta(x) = \tau loc \\ l = New_\tau(L), \sigma : L \end{cases}$$

Si noti che l'unica modifica che il nuovo comando deve fare è quella di associare all'identificatore x una nuova locazione, quindi deve modificare ρ con $[x = l]$. La condizione $\rho(x) = l_\perp$ ci assicura che allochiamo un identificatore solo una volta e $\Delta(x) = \tau loc$ è necessaria per scegliere una nuova locazione del tipo dichiarato per x. Notiamo inoltre che è possibile allocare solo identificatori dichiarati di tipo **var**. Inoltre si noti che tecnicamente stiamo eseguendo un comando, dunque non possiamo farlo transire in una configurazione del tipo $\langle \rho_0, \sigma \rangle$ perché le configurazioni dei comandi sono del tipo $\langle c, \sigma \rangle$ o σ. Dobbiamo allora fare in modo che il nuovo comando transisca in un altro comando. Avendo a disposizione un ambiente possiamo pensare di renderlo un blocco sequenzializzandolo col comando **nil** che non esegue alcuna modifica.
Vediamo adesso come modifichiamo la semantica dell'assegnamento

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \rightarrow_e^* \langle k, \sigma \rangle}{\rho \vdash_\Delta \langle x := e, \sigma \rangle \rightarrow_c \sigma[k/l]}, l = \rho(x) \neq l_\perp \tag{5.1}$$

Dunque dinamicamente controlliamo che l'assegnamento non venga fatto su una variabile dichiarata ma ancora non allocata.

Consideriamo infine la dichiarazione di variabile. Associamo ad x la locazione speciale $l_\perp$ e scriviamo $\perp$ in σ in corrispondenza di $l_\perp$.

$$\rho \vdash_\Delta \langle \mathbf{var}\ x : \tau, \sigma\rangle \rightarrow_d \langle [x = l_\perp], \sigma[l_\perp = \perp]\rangle$$

Per completezza possiamo scrivere le regole che generano l'errore (poi da propagare a tutti i contesti)

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e^* \langle k, \sigma\rangle}{\rho \vdash_\Delta \langle x := e, \sigma\rangle \rightarrow_c ERRORE}, \rho(x) = l_\perp$$

$$\rho \vdash_\Delta \langle x, \sigma\rangle \rightarrow_e ERRORE, \sigma(\rho(x)) = \perp$$

Si noti che è proprio la prima di queste due regole a causare un errore dinamico nel frammento fornito dal testo. Infatti, applicando le regole definite nell'Esercizio 4.2 a partire da un ambiente ρ_0 e una memoria σ_0 si ha che

$$\rho_0 \vdash_\Delta \langle \mathbf{var}\ x : int, \sigma_0\rangle \rightarrow_d \langle [x = l_\perp], \sigma_0[l_\perp = \perp]\rangle$$

Adesso dobbiamo valutare il corpo del frammento nell'ambiente modificato dalla dichiarazione fatta. Siano $\rho_1 = \rho_0[x = l_\perp]$ e $\sigma_1 = \sigma_0[l_\perp = \perp]$ l'ambiente e la memoria in cui dobbiamo valutare l'assegnamento; allora $\rho(x) = l_\perp$ e dunque la regola (5.1) non è applicabile, mentre lo è la prima tra quelle che generano errore e quindi

$$\rho_1 \vdash_\Delta \langle x := 1, \sigma_1\rangle \rightarrow_c ERRORE\ ,\ \rho_1(x) = l_\perp$$

□

Esercizio 5.11. *Aggiungere ad* IMP *la composizione collaterale di comandi con sintassi*

$$c \parallel c$$

e semantica intuitiva: "eseguire i due sottocomandi indipendentemente ed in un qualunque ordine". Definire semantica statica e dinamica e dimostrare che $c_0 \parallel c_1$ è equivalente a $c_1 \parallel c_0$. Determinare sotto quali condizioni $c_1 \parallel c_0$ è equivalente a $c_1; c_0$.

SOLUZIONE. Prima di tutto bisogna determinare quali sono gli identificatori liberi e quelli in posizione di definizione estendendo le definizioni di DI ed FI per il nuovo comando:

$$FI(c_0 \parallel c_1) = FI(c_0) \cup FI(c_1)$$
$$DI(c_0 \parallel c_1) = DI(c_0) \cup DI(c_1)$$

Semantica statica
A questo punto dobbiamo definire le regole di semantica statica che determinano quando il comando è ben formato; poiché i due comandi sono eseguiti in modo indipendente l'ambiente di tipi in cui si controllano entrambi i comandi è lo stesso

$$\frac{\Delta \vdash_I c_0, \Delta \vdash_I c_1}{\Delta \vdash_I c_0 \parallel c_1}$$

Si noti che le eventuali dichiarazioni presenti in un comando non influenzano gli identificatori liberi dell'altro poiché c_0 e c_1 devono essere indipendenti. Non ci sono altri aspetti statici che vengono influenzati.

Semantica dinamica
Le prime due regole esplicitano il fatto che non esiste un ordine prefissato per l'esecuzione dei due comandi. Le altre due regole ci dicono che una volta terminata l'esecuzione di un comando si prosegue con il residuo dell'altro.

1. $$\frac{\rho \vdash_\Delta \langle c_0, \sigma\rangle \rightarrow_c \langle c_0', \sigma'\rangle}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma\rangle \rightarrow_c \langle c_0' \parallel c_1, \sigma'\rangle}$$

2. $$\frac{\rho \vdash_\Delta \langle c_1, \sigma\rangle \rightarrow_c \langle c_1', \sigma'\rangle}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma\rangle \rightarrow_c \langle c_0 \parallel c_1', \sigma'\rangle}$$

3. $$\frac{\rho \vdash_\Delta \langle c_0, \sigma\rangle \rightarrow_c \sigma'}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma\rangle \rightarrow_c \langle c_1, \sigma'\rangle}$$

4. $$\frac{\rho \vdash_\Delta \langle c_1, \sigma\rangle \rightarrow_c \sigma'}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma\rangle \rightarrow_c \langle c_0, \sigma'\rangle}$$

Per dimostrare che $c_0 \parallel c_1$ è equivalente a $c_1 \parallel c_0$ dobbiamo dimostrare che $Exec(c_0 \parallel c_1, \sigma_0) = Exec(c_1 \parallel c_0, \sigma_0)$.

In seguito utilizzeremo la scrittura $Exec(c, \sigma) =^k \sigma'$, dove il k denota il numero di transizioni effettuate, intendendo dire che $\langle c, \sigma\rangle \rightarrow_c^k \langle c', \sigma'\rangle$ oppure $\langle c, \sigma\rangle \rightarrow_c^k \sigma'$.
Tale dimostrazione può essere fatta per induzione sul numero k di passi che vengono eseguiti. La dimostrazione si basa sul fatto che per ogni transizione che possiamo fare su $c_0 \parallel c_1$ riusciamo a farne una su $c_1 \parallel c_0$ che esegue le stesse modifiche e viceversa (infatti le regole sono simmetriche).

BASE $k = 1$
Se eseguiamo prima c_0 ed un passo di esecuzione non conduce ad una configurazione terminale, applichiamo la regola $(1.)$ oppure $(2.)$ a seconda che c_0 sia alla sinistra o alla destra di $\parallel$

$$(1.)\quad \frac{\rho \vdash_\Delta \langle c_0, \sigma_0\rangle \rightarrow_c \langle c_0', \sigma_1\rangle}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma_0\rangle \rightarrow_c \langle c_0 \parallel c_1, \sigma_1\rangle}$$

$$(2.)\quad \frac{\rho \vdash_\Delta \langle c_0, \sigma_0\rangle \rightarrow_c \langle c_0', \sigma_1\rangle}{\rho \vdash_\Delta \langle c_1 \parallel c_0, \sigma_0\rangle \rightarrow_c \langle c_1 \parallel c_0', \sigma_1\rangle}$$

Se invece il passo di esecuzione di c_0 produce una memoria finale σ_1, le due regole che possiamo applicare sono $(3.)$ o $(4.)$.

$$(3.)\quad \frac{\rho \vdash_\Delta \langle c_0, \sigma_0\rangle \rightarrow_c \sigma_1}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma\rangle \rightarrow_c \langle c_1, \sigma_1\rangle}$$

$$(4.)\quad \frac{\rho \vdash_\Delta \langle c_0, \sigma_0\rangle \rightarrow_c \sigma_1}{\rho \vdash_\Delta \langle c_1 \parallel c_0, \sigma_0\rangle \rightarrow_c \langle c_1, \sigma_1\rangle}$$

Il numero (i.) alla sinistra indica la regola usata. Analogamente, se eseguiamo prima c_1 abbiamo

$$(2.)\quad \frac{\rho \vdash_\Delta \langle c_1, \sigma_0\rangle \rightarrow_c \langle c_1', \sigma_1\rangle}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma_0\rangle \rightarrow_c \langle c_0 \parallel c_1', \sigma_1\rangle}$$

$$(1.)\quad \frac{\rho \vdash_\Delta \langle c_1, \sigma_0\rangle \to_c \langle c_1', \sigma_1\rangle}{\rho \vdash_\Delta \langle c_1 \parallel c_0, \sigma_0\rangle \to_c \langle c_1' \parallel c_0, \sigma_1\rangle}$$

oppure

$$(4.)\quad \frac{\rho \vdash_\Delta \langle c_1, \sigma_0\rangle \to_c \sigma_1}{\rho \vdash_\Delta \langle c_0 \parallel c_1, \sigma_0\rangle \to_c \langle c_0, \sigma_1\rangle}$$

$$(3.)\quad \frac{\rho \vdash_\Delta \langle c_1, \sigma_0\rangle \to_c \sigma_1}{\rho \vdash_\Delta \langle c_1 \parallel c_0, \sigma_0\rangle \to_c \langle c_0, \sigma_1\rangle}$$

Quindi dopo un passo di esecuzione

$$Exec(c_0 \parallel c_1, \sigma_0) =^1 Exec(c_1 \parallel c_0, \sigma_0) =^1 \sigma_1.$$

Inoltre, per come sono definite le regole, c_0', c_1' e σ_1 ottenuti dopo un passo di computazione sono uguali sia se si parte da $c_0 \parallel c_1$ sia se si parte da $c_1 \parallel c_0$.

PASSO INDUTTIVO
Supponiamo che dopo k passi

$$Exec(c_0 \parallel c_1, \sigma_0) =^k Exec(c_1 \parallel c_0, \sigma_0) =^k \sigma_k$$

e dobbiamo dimostrare che tale relazione vale ancora dopo $k+1$ passi.
Se eseguiamo prima c_0 e questo non conduce direttamente ad una configurazione finale, abbiamo

$$(1.)\quad \frac{\rho \vdash_\Delta \langle c_0', \sigma_k\rangle \to_c \langle c_0'', \sigma_{k+1}\rangle}{\rho \vdash_\Delta \langle c_0' \parallel c_1', \sigma_k\rangle \to_c \langle c_0'' \parallel c_1', \sigma_{k+1}\rangle}$$

$$(2.)\quad \frac{\rho \vdash_\Delta \langle c_0', \sigma_k\rangle \to_c \langle c_0'', \sigma_{k+1}\rangle}{\rho \vdash_\Delta \langle c_1' \parallel c_0', \sigma_0\rangle \to_c \langle c_1' \parallel c_0'', \sigma_{k+1}\rangle}$$

Se eseguiamo prima c_1 invece possiamo ottenere

$$(2.)\quad \frac{\rho \vdash_\Delta \langle c_1', \sigma_k\rangle \to_c \langle c_1'', \sigma_{k+1}\rangle}{\rho \vdash_\Delta \langle c_0' \parallel c_1', \sigma_k\rangle \to_c \langle c_0' \parallel c_1'', \sigma_{k+1}\rangle}$$

$$(1.)\quad \frac{\rho \vdash_\Delta \langle c_1', \sigma_k\rangle \to_c \langle c_1'', \sigma_{k+1}\rangle}{\rho \vdash_\Delta \langle c_1' \parallel c_0', \sigma_k\rangle \to_c \langle c_1'' \parallel c_0', \sigma_{k+1}\rangle}$$

Analogo nei casi in cui i comandi vadano in una memoria.
In queste regole abbiamo considerato σ_k, c_0' e c_1' rispettivamente la memoria, c_0 e c_1 dopo k passi di computazione; si noti che c_0', c_1' e σ_k ottenuti lavorando a partire da $c_0 \parallel c_1$ sono uguali a quelli ottenuti a partire dall'altro comando per ipotesi induttiva. Allora $Exec(c_0' \parallel c_1', \sigma_k) =^1 Exec(c_1' \parallel c_0', \sigma_k) =^1 \sigma_{k+1}$ dopo un passo di computazione, perciò si può concludere che

$$Exec(c_0 \parallel c_1, \sigma_0) =^{k+1} Exec(c_1 \parallel c_0, \sigma_0) =^{k+1} \sigma_{k+1}$$

Poiché le transizioni che eseguono $c_0 \parallel c_1$ e $c_1 \parallel c_0$ sono le stesse, la dimostrazione è conclusa. Infine, per rispondere all'ultima richesta dell'esercizio, bisogna capire in quale condizione l'esecuzione indipendente dei comandi c_0 e c_1 è equivalente all'esecuzione sequenziale, ad esempio di c_1 prima di c_0. Se i due comandi lavorano sulle stesse variabili l'eseguire prima uno o prima l'altro non può essere indifferente, mentre se lavorano su variabili distinte i due comandi sono totalmente indipendenti. Allora la condizione da imporre dovrà garantire che gli insiemi degli identificatori modificati da un comando ed utilizzati dall'altro siano disgiunti. Quindi

$$Mod(c_0) \cap Id(c_1) = \emptyset \ \wedge \ Mod(c_1) \cap Id(c_0) = \emptyset$$

dove $Id(c)$ indica l'insieme di identificatori utilizzati dal comando c e $Mod(c)$ quelli potenzialmente modificabili da c (vedi Esercizio 5.1). □

Esercizio 5.12. *Si consideri il linguaggio* IMP *modificato nel seguente modo:*

- *il tipo* int *coincide con i naturali;*
- *ogni comando non contiene sottoespressioni del tipo* $e - e'$*;*
- *come comando iterativo non si ha il* **while** *ma il comando definito nell'Esercizio 5.8.*

Dimostrare che qualsiasi sequenza di comandi termina sempre.

SOLUZIONE. Questa dimostrazione va fatta per induzione strutturale sulle espressioni e sui comandi del linguaggio la cui sintassi è riportata sotto per comodità.

$$c ::= \mathbf{nil} \mid id := e \mid c; c \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{do}\ e\ \mathbf{times}\ c$$
$$e ::= k \mid id \mid e\ \texttt{bop}\ e \mid \texttt{uop}\ e$$

Poiché la definizione dei comandi si fonda sulla definizione delle espressioni dobbiamo prima dimostrare induttivamente che la valutazione delle espressioni porta sempre in configurazioni terminali. Supponiamo inoltre che tutti i costrutti su cui facciamo la dimostrazione siano sintatticamente corretti, cioè supponiamo che la semantica statica non abbia generato errori.

i. Dimostriamo per induzione sulla definizione delle espressioni che la loro valutazione termina sempre in configurazioni del tipo $\langle k, \sigma \rangle$.

   BASE
   Per le costanti non dobbiamo fare nulla poiché $\langle k, \sigma \rangle$ è terminale. Rimangono da considerare gli identificatori.

$$\rho \vdash_{\Delta} \langle id, \sigma \rangle \rightarrow_e \langle k, \sigma \rangle,$$

$$k = \sigma(\rho(id))\ se\ \rho(id) \in Loc\ o\ k = \rho(id)\ se\ \rho(id) \in EVal$$

   PASSO INDUTTIVO
   Supponiamo, per ipotesi induttiva, che la valutazione di e_1 ed e_2 conduca sempre ad una configurazione terminale e dunque possiamo

sempre assegnare loro un valore. Allora le premesse delle seguenti regole di inferenza valgono per ipotesi induttiva:

$$\frac{\rho \vdash_\Delta \langle e_1, \sigma\rangle \rightarrow_e^* \langle m_1, \sigma\rangle}{\rho \vdash_\Delta \langle e_1 \texttt{ bop } e_2, \sigma\rangle \rightarrow_e \langle k_1 \texttt{ bop } e_2, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e_2, \sigma\rangle \rightarrow_e^* \langle k_2, \sigma\rangle}{\rho \vdash_\Delta \langle k_1 \texttt{ bop } k_2, \sigma\rangle \rightarrow_e \langle k_1 \texttt{ bop } k_2, \sigma\rangle}$$

$$\rho \vdash_\Delta \langle k_1 \texttt{ bop } k_2, \sigma\rangle \rightarrow_e \langle k', \sigma\rangle, k' = k_1 \texttt{ bop } k_2$$

Si noti che questo assioma è sempre applicabile perché le `bop` definite nel nostro linguaggio sono totali e non abbiamo la sottrazione che potrebbe generare valori negativi non ammessi (seconda ipotesi). Dunque raggiungiamo sempre una configurazione terminale. Naturalmente, però, se avessimo anche la divisione tra le `bop`, questo sarebbe vero solo se $m_2 \neq 0$ che non può essere deciso staticamente. Vediamo ora cosa succede con l'operatore unario `uop`.

$$\frac{\rho \vdash_\Delta \langle e_1, \sigma\rangle \rightarrow_e^* \langle t_1, \sigma\rangle}{\rho \vdash_\Delta \langle \texttt{uop } e_1, \sigma\rangle \rightarrow_e \langle t', \sigma\rangle}, t' = \texttt{uop } t_1$$

Si noti che anche la valutazione di questa espressione termina. Possiamo allora concludere che ad ogni espressione riusciamo ad associare un valore intero o booleano e cioè raggiungiamo una configurazione terminale.

ii. Presupponendo valido ciò che abbiamo appena dimostrato per le espressioni, dimostriamo per induzione sulla definizione dei comandi che l'esecuzione di ogni comando termina in configurazioni terminali del tipo σ.

BASE

Gli unici comandi che costituiscono la base sono **nil** e l'assegnamento che non contengono sottocomandi

$$\rho \vdash_\Delta \langle \mathbf{nil}, \sigma \rangle \to_c \sigma$$

Dunque tale comando termina. Per l'assegnamento abbiamo

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \to_e^* \langle k, \sigma \rangle}{\rho \vdash_\Delta \langle x := e, \sigma \rangle \to_c \sigma[l = k]}, l = \rho(x)$$

Anche questo comando porta ad una configurazione terminale.

PASSO INDUTTIVO
Supponiamo per ipotesi induttiva che c_1 e c_2 siano due comandi che terminano. Quindi le premesse delle seguenti regole di inferenza valgono o per ipotesi induttiva o per la dimostrazione precedente. Consideriamo prima la sequenzializzazione

$$\frac{\rho \vdash_\Delta \langle c_1, \sigma \rangle \to_c^* \sigma'}{\rho \vdash_\Delta \langle c_1; c_2, \sigma \rangle \to_c \langle c_2, \sigma' \rangle}$$

$$\rho \vdash_\Delta \langle c_2, \sigma' \rangle \to_c^* \sigma''$$

Dunque, poiché questo ultimo assioma vale per ipotesi induttiva, il comando $c_1; c_2$ termina sempre. Vediamo adesso il costrutto condizionale

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \to_e^* \langle \mathit{tt}, \sigma \rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_c \langle c_1, \sigma \rangle}$$

$$\rho \vdash_\Delta \langle c_1, \sigma \rangle \to_c^* \sigma'$$

Anche in questo caso l'esecuzione di c_1 termina per ipotesi induttiva.

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \to_e^* \langle \mathit{ff}, \sigma \rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_c \langle c_2, \sigma \rangle}$$

$$\rho \vdash_\Delta \langle c_2, \sigma \rangle \to_c^* \sigma''$$

Allora, visto che anche questo comando termina, ci resta da dimostrare che termina anche il comando iterativo

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle m, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{do}\ e\ \mathbf{times}\ c_1, \sigma\rangle \to_c \langle c_1; \mathbf{do}\ m-1\ \mathbf{times}\ c_1, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle c_1, \sigma\rangle \to_c^* \sigma'}{\rho \vdash_\Delta \langle c_1; \mathbf{do}\ m-1\ \mathbf{times}\ c_1, \sigma\rangle \to_c^* \langle \mathbf{do}\ m-1\ \mathbf{times}\ c_1, \sigma'\rangle}$$

Il numero $m \in \mathbb{N}$ poiché non abbiamo né numeri negativi né sottoespressioni del tipo $e - e'$ nei comandi. Poiché l'insieme dei numeri naturali è un insieme ben fondato, raggiungiamo sicuramente la seguente configurazione dopo m esecuzioni di c_1

$$\rho \vdash_\Delta \langle \mathbf{do}\ 0\ \mathbf{times}\ c_1, \sigma\rangle \to_c \sigma$$

Visto che anche questo comando termina, la dimostrazione è conclusa.

□

Esercizio 5.13. *Dimostrare che*

$$\rho \vdash_\Delta \langle c, \sigma\rangle \to_c \langle c', \sigma'\rangle \Rightarrow$$
$$\forall x \in Dom(\rho) \setminus Id(c).\ \sigma(l) = \sigma'(l), \rho(x) = l$$

dove $Id(c)$ indica l'insieme degli identificatori che occorrono in c.
SOLUZIONE. Dimostriamo l'implicazione per induzione strutturale sulla definizione dei comandi del nostro linguaggio.

BASE
Il comando **nil** non modifica la memoria

$$\rho \vdash_\Delta \langle \mathbf{nil}, \sigma\rangle \to_c \sigma$$

e quindi l'implicazione vale. Vediamo ora l'assegnamento

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle k, \sigma\rangle}{\rho \vdash_\Delta \langle x_0 := e, \sigma\rangle \to_c \sigma[l_0 = k]}, l_0 = \rho(x_0)$$

Poiché x_0 è un identificatore del comando che stiamo eseguendo e poiché per ipotesi $x \notin Id(c)$ allora sicuramente $x_0 \neq x$. Inoltre per definizione $\sigma[l_0 = k] = \sigma'$ è tale che (sapendo che l_0 è la locazione associata a x_0) $\forall x \neq x_0 \,.\, \sigma(x) = \sigma'(x)$.

PASSO INDUTTIVO
Supponiamo a questo punto di prendere un comando c i cui sottocomandi possono essere c_1 e/o c_2. Per ipotesi abbiamo che $x \notin Id(c)$ dunque, poiché l'insieme degli identificatori di un comando contiene gli identificatori dei suoi sottocomandi, sicuramente $x \notin Id(c_1) \cup Id(c_2)$ e quindi possiamo applicare caso per caso l'ipotesi induttiva.
Vediamo come funziona la dimostrazione per il comando $c = c_1; c_2$ dove supponiamo che, nell'ambiente ρ, $\langle c_1, \sigma\rangle \rightarrow_c^* \sigma_1$ e $\langle c_2, \sigma_1\rangle \rightarrow_c^* \sigma'$. Per ipotesi induttiva si ha che $\sigma_1(x) = \sigma(x)$ e che $\sigma_1(x) = \sigma'(x)$, perciò

$$\frac{\rho \vdash_\Delta \langle c_1, \sigma\rangle \rightarrow_c^* \sigma_1}{\rho \vdash \Delta\langle c_1; c_2, \sigma\rangle \rightarrow_c^* \langle c_2, \sigma_1\rangle}$$

$$\rho \vdash_\Delta \langle c_2, \sigma_1\rangle \rightarrow_c^* \sigma'$$

da cui

$$\rho \vdash_\Delta \langle c_1; c_2, \sigma\rangle \rightarrow_c^* \sigma'.$$

Quindi sappiamo che, per le ipotesi induttive e per la proprietà transitiva dell'uguaglianza, $\sigma(x) = \sigma'(x)$.
Vediamo adesso cosa succede per il comando condizionale. Supponiamo che, nell'ambiente ρ, $\langle c_1, \sigma\rangle \rightarrow_c^* \sigma_1$ e $\langle c_2, \sigma\rangle \rightarrow_c^* \sigma_2$; allora le ipotesi induttive sono $\sigma_1(x) = \sigma(x) = \sigma_2(x)$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e^* \langle tt, \sigma\rangle}{\rho \vdash_\Delta \langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \rightarrow_c \langle c_1, \sigma\rangle}$$

Per ipotesi induttiva sappiamo che tale comando termina in una memoria σ_1 che non altera il valore di x. Cioè $\sigma(x) = \sigma_1(x)$. Vediamo l'altro caso

dell'**if**

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle \to_c \langle c_2, \sigma\rangle}$$

Anche qui per ipotesi induttiva sappiamo che il comando c_2 termina in una memoria σ_2 che non altera il valore di x, cioè $\sigma(x) = \sigma_2(x)$.
Vediamo infine il caso del **while**. Supponiamo che, in ρ, $\langle c_1, \sigma\rangle \to_c^* \sigma_1$, e dunque per ipotesi induttiva vale $\sigma(x) = \sigma_1(x)$. Poiché nel comando iterativo si esegue c_1 ripetutamente le ipotesi possono non essere conservate. Dunque per completare la dimostrazione bisogna dimostrare, per induzione sul numero di cicli eseguiti dal **while**, che, se dopo n passi la memoria è σ_n allora vale $\sigma_n(x) = \sigma(x)$. Questo conclude l'induzione strutturale.

BASE Vediamo cosa succede con un passo di esecuzione:

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{while}\ e\ \mathbf{do}\ c_1, \sigma\rangle \to_c \sigma}$$

La memoria non viene modificata e quindi le ipotesi fatte rimangono valide. Vediamo ora l'altro caso

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{while}\ e\ \mathbf{do}\ c_1, \sigma\rangle \to_c^* \langle \mathbf{while}\ e\ \mathbf{do}\ c_1, \sigma_1\rangle} \tag{5.2}$$

Per ipotesi $\sigma(x) = \sigma_1(x)$ e quindi l'enunciato da dimostrare vale nel caso base. Si nota, inoltre, che la regola di inferenza corretta per $\mathbf{while}\ e\ \mathbf{do}\ c_1$ è differente da quella qui riportata per il fatto che prima di eseguire nuovamente il **while** si esegue c_1. Abbiamo condensato questi passi in un'unica regola per semplicità e maggiore chiarezza.

PASSO INDUTTIVO Supponiamo che dopo n applicazioni della regola (5.2) la memoria risultante, a partire dalla memoria iniziale σ, sia σ_n. Per ipotesi induttiva (sui cicli) assumiamo $\sigma(x) = \sigma_n(x)$. Vediamo, dunque, cosa succede dopo $n+1$ passi del **while**

$$\frac{\rho \vdash_\Delta \langle c_1, \sigma_n\rangle \to_c^* \sigma_{n+1}}{\rho \vdash_\Delta \langle \mathbf{while}\ e\ \mathbf{do}\ c_1, \sigma_n\rangle \to_c \langle \mathbf{while}\ e\ \mathbf{do}\ c_1, \sigma_{n+1}\rangle}$$

Per ipotesi $x \notin Id(\mathbf{while}\ e\ \mathbf{do}\ c_1)$ e quindi$x \notin Id(c_1)$. Per ipotesi induttiva strutturale vale che $\sigma_{n+1}(x) = \sigma_n(x)$, mentre per ipotesi induttiva sui cicli $\sigma_n(x) = \sigma(x)$ e dunque per transitività dell'uguaglianza si ha $\sigma_{n+1}(x) = \sigma(x)$.

□

Esercizio 5.14. *Aggiungere ad* IMP *il comando*

$$c ::= d\ \mathbf{init}\ c\ \mathbf{end}$$

il cui significato è quello di eseguire i comandi c di inizializzazione di d prima dei comandi del blocco in cui d compare.

SOLUZIONE. Questo esercizio può essere velocemente risolto se si capisce bene cosa fa esattamente tale comando; la sua semantica intuitiva è costituita dai seguenti passi:

1. elaborare d;
2. eseguire c;
3. proseguire oltre.

Questa è esattamente la semantica intuitiva del comando $d; c$ quindi possiamo dire che se IMP ha semantica SEM il nostro nuovo linguaggio ha semantica SEM/$_\equiv$ dove $(d\ \mathbf{init}\ c\ \mathbf{end}, c; d) \in \equiv$. In tal modo la risoluzione dell'esercizio sarebbe finita; comunque per completezza diamo le regole di inferenza per il comando.

Definiamo gli identificatori liberi e in posizione di definizione:

$$FI(d\ \mathbf{init}\ c\ \mathbf{end}) = FI(c) \setminus DI(d)$$
$$DI(d\ \mathbf{init}\ c\ \mathbf{end}) = DI(d)$$

Semantica statica
Dobbiamo controllare che il nuovo comando sia ben formato. Questo avviene se la dichiarazione d genera un ambiente di tipi Δ_0 corretto ed il

comando c è ben formato tenendo conto anche dei legami introdotti da d.

$$\frac{\Delta \vdash_I d : \Delta_0,\ \Delta[\Delta_0] \vdash_{I \cup I_0} c}{\Delta \vdash_I d\ \mathbf{init}\ c\ \mathbf{end}}, \Delta_0 : I_0$$

Semantica dinamica
Per descrivere il comportamento dinamico del nuovo costrutto dobbiamo definire delle regole che ricalcano quelle del blocco. Questo garantisce anche che l'equivalenza $d\ \mathbf{init}\ c\ \mathbf{end} \equiv d;c$ sia verificata.

$$\frac{\rho \vdash_\Delta \langle d, \sigma\rangle \to_d \langle d', \sigma'\rangle}{\rho \vdash_\Delta \langle d\ \mathbf{init}\ c\ \mathbf{end}, \sigma\rangle \to_c \langle d'\ \mathbf{init}\ c\ \mathbf{end}, \sigma'\rangle}$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle c, \sigma\rangle \to_c \langle c', \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0\ \mathbf{init}\ c\ \mathbf{end}, \sigma\rangle \to_c \langle \rho_0\ \mathbf{init}\ c'\ \mathbf{end}, \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle c, \sigma\rangle \to_c \sigma'}{\rho \vdash_\Delta \langle \rho_0\ \mathbf{init}\ c\ \mathbf{end}, \sigma\rangle \to_c \sigma'}, \rho_0 : \Delta_0$$

□

Esercizio 5.15. *Si modifichi la sintassi di* IMP *aggiungendo alla categoria sintattica delle espressioni il costrutto*

$$e ::= \mathbf{begin}\ c\ \mathbf{result}\ e$$

con il significato intuitivo di eseguire c e poi valutare e.

SOLUZIONE. Per quel che riguarda gli identificatori liberi dobbiamo aggiungere (alla definizione di FI)

$$FI(\mathbf{begin}\ c\ \mathbf{result}\ e) = FI(c) \cup (FI(e) \setminus DI(c))$$

Diverso è il discorso per gli identificatori in posizione di definizione. Infatti le nuove espressioni contengono i comandi che possono modificare il

valore degli identificatori con assegnamenti, e possono contenere dichiarazioni che definiscono identificatori. Quindi dobbiamo aggiungere al nostro linguaggio la definizione dell'insieme di identificatori definiti (DI) da un'espressione:

$$DI(m) = DI(t) = \emptyset$$
$$DI(x) = \emptyset$$
$$DI(\textbf{begin } c \textbf{ result } e) = DI(c) \cup DI(e)$$
$$DI(e_0 \texttt{ bop } e_1) = DI(e_0) \cup DI(e_1)$$
$$DI(\textbf{not } e) = DI(e)$$

A questo punto dobbiamo definire la semantica statica e dinamica del nuovo costrutto.

Semantica statica
Consideriamo l'ambiente di tipi iniziale Δ definito sull'insieme I di identificatori, in base al quale determiniamo il tipo della nuova espressione. La regola più naturale per la semantica statica è

$$\frac{\Delta \vdash_I c, \Delta \vdash_I e : \tau}{\Delta \vdash_I \textbf{begin } c \textbf{ result } e \ : \ \tau}$$

cioè la nuova espressione è *ben formata* se lo sono i suoi elementi componenti. Questa regola purtroppo non è corretta. Consideriamo infatti l'espressione

$$\textit{tt} \textbf{ or begin var } z : int = 4; \textbf{nil end } (z = 4)$$

che è corretta poiché l'identificatore z è dichiarato nel corpo del **begin**. Se applichiamo la regola data quando andiamo a valutare $(z = 4)$ non abbiamo nessun legame per z in Δ e quindi otteniamo un errore. Infatti, dovremmo valutare l'espressione nell'ambiente Δ esteso con i legami generati dalle eventuali dichiarazioni presenti in c. In effetti, questo è ciò che avevamo in mente quando abbiamo esteso FI e definito DI. Per questo motivo modifichiamo le regole della semantica statica dei comandi in modo che questi restituiscano l'ambiente di tipi generato dalle eventuali dichiarazioni che contengono. Le nuove regole per i comandi sono riportate in Tab. 5.3.

$$\Delta \vdash_I \textbf{nil} : \emptyset \qquad \frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I x := e : \emptyset},\ \Delta(x) = \tau loc$$

$$\frac{\Delta \vdash_I c_0 : \Delta_0,\ \Delta \vdash_I c_1 : \Delta_1}{\Delta \vdash_I c_0; c_1 : \Delta_0[\Delta_1]} \qquad \frac{\Delta \vdash_I e : \mathit{bool},\ \Delta \vdash_I c : \Delta'}{\Delta \vdash_I \textbf{while } e \textbf{ do } c : \Delta'}$$

$$\frac{\Delta \vdash_I e : \mathit{bool},\ \Delta \vdash_I c_0 : \Delta_0,\ \Delta \vdash_I c_1 : \Delta_0}{\Delta \vdash_I \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1 : \Delta_0}$$

$$\frac{\Delta \vdash_I d : \Delta',\ \Delta[\Delta'] \vdash_{I \cup I'} c : \Delta''}{\Delta \vdash_I d; c : \Delta'[\Delta'']}, \Delta' : I'$$

Tabella 5.3: Semantica statica per i comandi di IMP.

Si noti che abbiamo imposto ai due rami del comando condizionale di generare lo stesso ambiente di tipi. Se così non fosse, non saremmo in grado di determinare staticamente la correttezza di certi frammenti come

$$\begin{array}{l} \mathit{tt} \textbf{ or begin if } e \textbf{ then var } x : \mathit{int} = 4; \textbf{nil} \\ \quad \textbf{else var } z : \mathit{bool} = \mathit{tt}; \textbf{nil end}(x = 3) \end{array}$$

Se la condizione e è falsa l'identificatore x non è definito. Se avessimo permesso nelle regole la generazione di Δ_0 da c_0 e Δ_1 da c_1 mettendo nella conclusione $\Delta_0[\Delta_1]$ non avremmo scoperto l'errore staticamente.

Semantica dinamica
Nella valutazione della nuova espressione si esegue inizialmente c e poi, considerando le modifiche che esso ha apportato alla memoria, si valuta l'espressione e.

$$\frac{\rho \vdash_\Delta \langle c, \sigma \rangle \rightarrow_c \langle c', \sigma' \rangle}{\rho \vdash_\Delta \langle \textbf{begin } c \textbf{ result } e, \sigma \rangle \rightarrow_e \langle \textbf{begin } c' \textbf{ result } e, \sigma' \rangle}$$

$$\frac{\rho \vdash_\Delta \langle c, \sigma\rangle \to_c \sigma'}{\rho \vdash_\Delta \langle \mathbf{begin}\ c\ \mathbf{result}\ e, \sigma\rangle \to_e \langle e, \sigma'\rangle}$$

Inoltre poiché la valutazione delle espressioni può modificare la memoria, dobbiamo rimpiazzare ciascuna transizione $\rho \vdash_\Delta \langle e, \sigma\rangle \to_e \langle e', \sigma\rangle$ in Tab. 3.3 con $\rho \vdash_\Delta \langle e, \sigma\rangle \to_e \langle e', \sigma'\rangle$. □

Esercizio 5.16. *Modificare il linguaggio permettendo solo la dichiarazione*

$$d ::= \mathbf{var}\ x$$

Definire quindi un type-checker dinamico.

SOLUZIONE. Supponiamo di aver apportato ad IMP le modifiche fatte nell'Esercizio 4.2. Definiamo allora gli identificatori liberi e in posizione di definizione nel seguente modo

$$FI(\mathbf{var}\ x) = \emptyset$$
$$DI(\mathbf{var}\ x) = \{x\}$$

A questo punto si noti che non introducendo i tipi nelle sintassi delle dichiarazioni, non riusciamo a definire l'ambiente di tipi per gli identificatori durante l'analisi statica che quindi scompare. Allora dobbiamo definire l'ambiente Δ a tempo di esecuzione portandocelo dietro nelle configurazioni della semantica dinamica che saranno del tipo

$$\rho \vdash_I \langle d, \Delta, \sigma\rangle \to_d \langle d', \Delta', \sigma'\rangle$$

Aggiungiamo inoltre un nuovo tipo denotabile che individua le variabili per le quali non è ancora stato definito il tipo:

$$DTyp ::= \ldots \mid \top$$

Allora la semantica dinamica della dichiarazione sarà

$$\rho \vdash_I \langle \mathbf{var}\ x, \Delta, \sigma\rangle \to_d \langle [x = l_\perp], [x = \top], \sigma\rangle$$

dove $l_\perp$ indica che la variabile non è stata allocata. Vediamo adesso le dichiarazioni composte e consideriamo quella sequenziale (quelle privata e simultanea sono analoghe). Le regole che dobbiamo aggiungere sono

$$\frac{\rho \vdash_I \langle d_0, \Delta, \sigma\rangle \rightarrow_d \langle d_0', \Delta', \sigma'\rangle}{\rho \vdash_I \langle d_0; d_1, \Delta, \sigma\rangle \rightarrow_d \langle d_0'; d_1, \Delta[\Delta'], \sigma'\rangle}$$

$$\frac{\rho[\rho_0] \vdash_{I \cup I_0} \langle d_1, \Delta, \sigma\rangle \rightarrow_d \langle d_1', \Delta', \sigma'\rangle}{\rho \vdash_I \langle \rho_0; d_1, \Delta, \sigma\rangle \rightarrow_d \langle \rho_0; d_1', \Delta[\Delta'], \sigma'\rangle}, \rho_0 : I_0$$

$$\rho \vdash_I \langle \rho_0; \rho_1, \Delta, \sigma\rangle \rightarrow_d \langle \rho_0[\rho_1], \Delta, \sigma\rangle$$

Notiamo che la composizione di ambienti statici avviene solo nelle regole. Nell'assioma è già stata effettuata induttivamente.
Per quel che riguarda i comandi, la derivazione sarà del tipo

$$\rho \vdash_I \langle c, \Delta, \sigma\rangle \rightarrow_c \langle c', \Delta', \sigma'\rangle$$

oppure

$$\rho \vdash_I \langle c, \Delta, \sigma\rangle \rightarrow_c \langle \Delta', \sigma'\rangle$$

L'assegnamento rimane pressoché inalterato, però le condizioni di applicabilità si complicano notevolmente

$$\frac{\rho \vdash_I \langle e, \sigma\rangle \rightarrow_e^* \langle k, \sigma\rangle}{\rho \vdash_I \langle x := e, \Delta, \sigma\rangle \rightarrow_c \langle \Delta[x = \tau loc], \sigma[l = k]\rangle}$$

questo se valgono le seguenti condizioni:

$$\Delta(e) = \tau \ \wedge \ (\Delta(x) = \tau loc \ \vee \ \Delta(x) = \top)$$

con

$$l = \begin{cases} \rho(x) & \text{se } \rho(x) \neq l_\perp \wedge \rho(x) \in Loc_\tau \\ New_\tau(L) & \text{se } \rho(x) = l_\perp \wedge \Delta(x) = \top \wedge \sigma : L \end{cases}$$

La condizione $\Delta(x) = \top$ serve per assicurare che l'identificatore comunque era stato precedentemente dichiarato.
Per concludere, esaminiamo il blocco. Le nuove regole sono

$$\frac{\rho \vdash_I \langle d, \Delta, \sigma\rangle \rightarrow_d^* \langle \rho_0, \Delta[\Delta_0], \sigma'\rangle}{\rho \vdash_I \langle d; c, \Delta, \sigma\rangle \rightarrow_c \langle \rho_0; c, \Delta[\Delta_0], \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\frac{\rho[\rho_0] \vdash_{I \cup I_0} \langle c, \Delta[\Delta_0], \sigma\rangle \rightarrow_c \langle c', \Delta', \sigma'\rangle}{\rho \vdash_I \langle \rho_0; c, \Delta[\Delta_0], \sigma'\rangle \rightarrow_c \langle \rho_0; c', \Delta([\Delta_0][\Delta']), \sigma'\rangle}, \rho_0 : \Delta_0$$

$$\frac{\rho[\rho_0] \vdash_{I \cup I_0} \langle c, \Delta[\Delta_0], \sigma\rangle \rightarrow_c \langle \Delta', \sigma'\rangle}{\rho \vdash_I \langle \rho_0; c, \Delta[\Delta_0], \sigma\rangle \rightarrow_c \langle \Delta, \sigma'\rangle}, \rho_0 : \Delta_0$$

□

Esercizio 5.17. *Definire semantica statica e dinamica della seguente dichiarazione*

$$d ::= \textbf{external}\ x : \tau$$

con semantica intuitiva: se fuori da un blocco un identificatore è definito **external** *allora in qualsiasi blocco in cui venga ridefinito il valore a cui si fa riferimento è quello che l'identificatore aveva fuori dal blocco.*

SOLUZIONE. La prima considerazione da fare è che per poter inserire questo tipo di modifica bisogna fare riferimento al linguaggio IMP modificato nell'Esercizio 4.2 che non inizializza gli identificatori quando li dichiara. Inoltre, come nell'Esercizio 4.4, occorre un nuovo tipo che permetta di distinguere gli identificatori **external** dagli altri:

$$DTyp ::= \ldots \mid \tau ext$$

Gli insiemi DI e FI sono uguali a quelli delle altre dichiarazioni. Vediamo quindi la semantica statica e dinamica.

Semantica statica

$$\Delta \vdash_I (\mathbf{external}\ x : \tau) : [x = \tau ext]$$

L'assegnamento non cambia, mentre cambia la semantica statica del blocco.

$$\frac{\Delta \vdash_I d : \Delta_0,\ \Delta' \vdash_{I \cup I_0} c}{\Delta \vdash_I d; c},\ \ \Delta_0 : I_0$$

dove $\Delta' = (\Delta[\Delta_0])[\Delta_{|ext}]$ con $I_{|ext} = \{x | \Delta(x) = \tau ext\}$ e $\Delta_{|ext} : I_{|ext}$. Quindi c viene valutato nell'ambiente di tipi modificato dalla dichiarazione d tranne per quegli identificatori che fuori dal blocco erano stati dichiarati **external** per i quali si tiene valida la dichiarazione fatta in precedenza.

Semantica dinamica
Vediamo come viene valutata la nuova dichiarazione:

$$\rho \vdash_\Delta \langle \mathbf{external}\ x : \tau, \sigma \rangle \rightarrow_d \langle [x = l], \sigma[l = \bot] \rangle,\ l \in New_{\tau ext}(L),\ \sigma : L$$

Il concetto di blocco cambia nello stesso modo in cui è cambiato a livello statico

$$\frac{\rho \vdash_\Delta \langle d, \sigma \rangle \rightarrow_d \langle d', \sigma' \rangle}{\rho \vdash_\Delta \langle d; c, \sigma \rangle \rightarrow_c \langle d'; c, \sigma' \rangle}$$

$$\frac{\rho' \vdash_{\Delta'} \langle c, \sigma \rangle \rightarrow_c \langle c', \sigma' \rangle}{\rho \vdash_\Delta \langle \rho_0; c, \sigma \rangle \rightarrow_c \langle \rho_0; c', \sigma' \rangle}$$

$$\frac{\rho' \vdash_{\Delta'} \langle c, \sigma \rangle \rightarrow_c \sigma'}{\rho \vdash_\Delta \langle \rho_0; c, \sigma \rangle \rightarrow_c \sigma'}$$

dove Δ' è quello definito nella semantica statica e $\rho' = (\rho[\rho_0])[\rho_{|ext}]$ con $\rho_{|ext}$ definito in modo analogo a $\Delta_{|ext}$.

□

Esercizio 5.18. *Dimostrare che vale la seguente implicazione*

$$\Delta \vdash_I c \ \wedge \ \rho \vdash_\Delta \langle c, \sigma \rangle \to_c \langle c', \sigma' \rangle \ \Rightarrow \ \Delta \vdash_I c'$$

Poiché la categoria sintattica dei comandi utilizza quella delle espressioni e quella delle dichiarazioni, assumiamo validi i risultati degli Esercizi 3.6 e 4.6.

SOLUZIONE. Dimostriamo l'implicazione per induzione sulla struttura della categoria dei comandi.

BASE
Se $c = \mathbf{nil}$, nessuna transizione della semantica dinamica del tipo richiesto dal testo è applicabile e dunque l'implicazione è vera.
L'altro caso base è $c = x := e$. Per ipotesi il comando è ben formato e quindi sia a e che a x riusciamo ad assegnare un tipo, cioè sono entrambe ben formate. Infatti se solo una delle due non lo fosse non lo sarebbe neanche c per la regola

$$\frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I x := e}, \Delta(x) = \tau loc \tag{5.3}$$

La regola della semantica dinamica che possiamo applicare è

$$\frac{\rho \vdash_\Delta \langle e, \sigma \rangle \to_e \langle e', \sigma \rangle}{\rho \vdash_\Delta \langle x := e, \sigma \rangle \to_c \langle x := e', \sigma \rangle}$$

Vogliamo quindi dimostrare che $c' = x := e'$ è ben formato. Applicando la regola (5.3) a c', deduciamo che questo è ben formato poiché x non è stata modificata quindi per la regola (5.3) $\Delta(x) = \tau loc$, ed $e' : \tau$ per il risultato dimostrato nell'Esercizio 3.6. Infatti e è ben formata per quanto detto inizialmente ed e' è ottenuta da e mediante una applicazione di una regola della semantica dinamica.

PASSO INDUTTIVO
Supponiamo che valga l'ipotesi induttiva per i comandi c_0 e c_1. Vediamo il caso in cui $c = c_0; c_1$. Per ipotesi tale comando è ben formato, e allora lo

sono anche c_0 e c_1, perché se anche uno solo dei due non lo fosse, non lo sarebbe neanche c per la regola

$$\frac{\Delta \vdash_I c_0,\ \Delta \vdash_I c_1}{\Delta \vdash_I c_0; c_1} \tag{5.4}$$

Per quel che riguarda la semantica dinamica, l'esecuzione di tale comando consiste nell'esecuzione sequenziale dei suoi sottocomandi. Vediamo il caso in cui viene eseguito c_0, poi l'altro è analogo.

$$\frac{\rho \vdash_\Delta \langle c_0, \sigma\rangle \to_c \langle c_0', \sigma'\rangle}{\rho \vdash_\Delta \langle c_0; c_1, \sigma\rangle \to_c \langle c_0'; c_1, \sigma'\rangle}$$

Dunque vogliamo dimostrare che $c' = c_0'; c_1$ è ben formato. Applicando la regola (5.4) a c', questo è ben formato se sono ben formati c_0' e c_1. Ma c_1 lo è per ipotesi mentre c_0' lo è per ipotesi induttiva.
Esaminiamo ora $c = \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$ e supponiamo che sia ben formato, perciò, come nei casi precedenti, lo sono anche e, c_0 e c_1, perché se almeno uno non lo fosse non lo sarebbe neanche c per la regola

$$\frac{\Delta \vdash_I e : \mathit{bool},\ \Delta \vdash_I c_0,\ \Delta \vdash_I c_1}{\Delta \vdash_I \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1}$$

Per quel che riguarda la semantica dinamica, ci sono due possibilità a seconda del valore associato ad e

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to_c \langle c_0, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \to_c \langle c_1, \sigma\rangle}$$

In entrambi i casi il comando transisce in comandi che, per ipotesi, sono ben formati.
Consideriamo $c = \mathbf{while}\ e\ \mathbf{do}\ c_0$. Supponiamo che sia ben formato allora

per le solite considerazioni lo sono anche e e c_0. Se il valore di e è falso, la configurazione in cui il comando transisce è terminale, dunque non è una transizione del tipo richiesto delle ipotesi dell'implicazione, per questo l'implicazione è vera. Vediamo ora la regola nel caso in cui e valga vero

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \to_e^* \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{while}\ e\ \mathbf{do}\ c_0, \sigma\rangle \to_c \langle c_0; \mathbf{while}\ e\ \mathbf{do}\ c_0, \sigma\rangle}$$

per vedere che adesso $c' = c_0; \mathbf{while}\ e\ \mathbf{do}\ c_0$ è ben formato basta dimostrare che lo sono i comandi che lo compongono. Notiamo che sia c_0 che il **while** lo sono per ipotesi.
Vediamo infine il caso del blocco, $c = d; c_0$. Per ipotesi il blocco è ben formato e per le solite considerazioni lo sono anche d e c_0. Per quel che riguarda la semantica dinamica l'esecuzione del comando consiste nell'elaborazione di d e poi nell'esecuzione di c_0. Vediamo i due casi distinti

$$\frac{\rho \vdash_\Delta \langle d, \sigma\rangle \to_d \langle d', \sigma\rangle}{\rho \vdash_\Delta \langle d; c_0, \sigma\rangle \to_c \langle d'; c_0, \sigma\rangle}$$

In questo caso il comando $c' = d'; c_0$ è ben formato. Infatti d' lo è per il risultato ottenuto nell'Esercizio 4.6, in quanto d è ben formata per ipotesi e d' è ottenuta da d mediante l'applicazione di una regola della semantica dinamica; c_0 lo è per ipotesi. Vediamo l'altro caso

$$\frac{\rho \vdash_\Delta \langle c_0, \sigma\rangle \to_c \langle c_0', \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0; c_0, \sigma\rangle \to_c \langle \rho_0; c_0', \sigma'\rangle}$$

In tal caso, perché il nuovo comando sia ben formato, è sufficiente che lo sia c_0' in quanto l'altra componente è un ambiente al quale possiamo sempre associare l'ambiente statico compatibile. Ma c_0' è ben formato per ipotesi induttiva essendo stato ottenuto da c_0 mediante una applicazione di una regola della semantica dinamica ed essendo c_0 ben formato per ipotesi induttiva. □

Esercizio 5.19. *Dato un comado c e un ambiente statico Δ dimostrare che vale la seguente implicazione*

$$x_0 \in FI(c) \setminus Dom(\Delta) \;\Rightarrow\; \Delta \nvdash_I c$$

assumendo i risultati dimostrati negli esercizi 3.7 e 4.7.

SOLUZIONE. Dimostriamo l'implicazione per induzione sulla struttura dei comandi.

BASE
Consideriamo $c = \mathbf{nil}$. In tal caso $x_0 \notin FI(\mathbf{nil})$ dunque l'implicazione è sempre vera.
L'altro caso base consiste in $c = x := e$. Per ipotesi $x_0 \in FI(x := e) = \{x\} \cup FI(e)$. In questa situazione il comando c non è ben formato poiché l'unica regola che potremmo applicare

$$\frac{\Delta \vdash_I e : \tau}{\Delta \vdash_I x := e},\;\; \Delta(x) = \tau loc$$

ha la condizione a margine non verificata se $x = x_0$, poiché $x_0 \notin Dom(\Delta)$, oppure ha la premessa non verificata se $x_0 \in FI(e)$ per il risultato dimostrato nell'Esercizio 3.7.

PASSO INDUTTIVO
Consideriamo $c = c_0; c_1$. Per ipotesi $x_0 \in FI(c_0; c_1) = FI(c_0) \cup FI(c_1)$. Dunque per almeno uno dei due comandi vale l'ipotesi induttiva e dunque almeno uno dei comandi non è ben formato implicando che anche c non è ben formato in base all'unica regola applicabile per la composizione sequenziale

$$\frac{\Delta \vdash_I c_0,\; \Delta \vdash_I c_1}{\Delta \vdash_I c_0; c_1}$$

dove almeno una delle due premesse è falsa.
Consideriamo ora $c = \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$ e supponiamo che $x_0 \in FI(c) = FI(e) \cup FI(c_0) \cup FI(c_1)$. In tal caso x_0 appartiene ad almeno uno dei tre insiemi e quindi l'ipotesi induttiva vale per almeno uno dei tre

sottotermini. Per questo almeno una delle premesse della seguente regola è falsa

$$\frac{\Delta \vdash_I e : \mathit{bool},\ \Delta \vdash_I c_0,\ \Delta \vdash_I c_1}{\Delta \vdash_I \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1}$$

quindi il comando c non è ben formato.
Analoga è la dimostrazione per $c = \mathbf{while}\ e\ \mathbf{do}\ c_0$.

Consideriamo infine il caso in cui $c = d; c_0$ e supponiamo che $x_0 \in FI(d; c_0) = FI(d) \cup (FI(c_0) \setminus DI(d))$. In questo contesto x_0 appartiene ad almeno uno dei due insiemi dell'unione. Se $x_0 \in FI(d)$, per il risultato dimostrato nell'Esercizio 4.7 non riusciamo ad associare un ambiente statico al blocco. Invece se $x_0 \in FI(c_0) \setminus DI(d)$ si ha che $x_0 \notin Dom(\Delta[\Delta'])$, dove Δ' è l'ambiente associato a d, e dunque per ipotesi induttiva c_0 non è ben formato nell'ambiente $\Delta[\Delta']$. Dunque l'unica regola che possiamo applicare, anche in questo caso, ha una premessa non verificata ben formato

$$\frac{\Delta \vdash_I d : \Delta',\ \Delta[\Delta'] \vdash_{I \cup I'} c_0}{\Delta \vdash_I d; c_0}, \quad \Delta' : I'$$

e possiamo concludere che il blocco non è ben formato. □

Capitolo 6

Procedure

L'idea che sta alla base delle procedure è quella di abbreviare la scrittura di programmi che contengono sequenze ripetute di comandi uguali e che differiscono al più per i dati su cui operano. L'astrazione sui dati viene implementata mediante il meccanismo di passaggio dei parametri. Essenzialmente, leghiamo negli ambienti una sequenza di comandi (il corpo della procedura) ad un identificatore (il nome della procedura). Le procedure implementano il principio di astrazione sui comandi. Trattiamo in questo capitolo le procedure così come definite in IMP fornendo la loro semantica statica e dinamica. Proseguiamo quindi fornendo la soluzione di alcuni esercizi proposti.

6.1 Semantica delle procedure

Riportiamo prima le definizioni di FI e DI.

Definizione 6.1 (identificatori liberi). *La funzione che ad ogni dichiarazione e comando riguardante le procedure associa l'insieme degli identifi-*

catori liberi in essi contenuti è definita per induzione da

$$FI(\textbf{procedure}\ p(form)c) = FI(c) \setminus DI(form)$$
$$FI(form = ae) = FI(ae)$$
$$FI(p(ae)) = FI(ae) \cup \{p\}$$
$$FI(\bullet) = \emptyset$$
$$FI(form) = \emptyset$$
$$FI(e, ae) = FI(e) \cup FI(ae)$$

Definizione 6.2 (identificatori in posizione di definizione). *La funzione che ad ogni dichiarazione e comando riguardante le procedure associa l'insieme degli identificatori in posizione di definizione in essi contenuti è definita per induzione da*

$$DI(\textbf{procedure}\ p(form)c) = \{p\} \cup DI(form) \cup DI(c)$$
$$DI(form = ae) = DI(form)$$
$$DI(p(ae)) = \emptyset$$
$$DI(\bullet) = \emptyset$$
$$DI(ae) = \emptyset$$
$$DI(\texttt{const}\ x : \tau) = DI(\texttt{var}\ x : \tau) = \{x\} \cup DI(form)$$

Definiamo quindi la semantica statica il cui scopo è quello di associare un ambiente di tipi alla lista dei parametri formali, una lista di tipi ai parametri attuali, di verificare che i tipi dei formali e degli attuali siano coincidenti nella chiamata di procedura e, infine, di controllare se il corpo della procedura è ben formato. Le regole sono riportate in Tab. 6.1.

Per discutere la semantica statica delle procedure partiamo dalla dichiarazione di procedura. La prima regola associa all'identificatore di procedura p il tipo $\mathcal{T}(form)proc$ nel caso in cui la lista di parametri formali, così come il corpo della procedura siano ben formati. La funzione $\mathcal{T} : Form \rightarrow ATyp$ restituisce la lista dei tipi dei parametri formali. Il tipo di p è quindi $aetproc$ (si noti che il tipo degli attuali e dei formali è lo stesso). La funzione $\mathcal{T}$ è definita per induzione strutturale sulla sintassi di *Form* considerando come caso base la lista col solo elemento $\bullet$ che è definito di tipo $\bullet$. La seconda regola in Tab. 6.1 riguarda la chiamata di

$$\frac{form : \Delta_0,\ \ \Delta[\Delta_0] \vdash_{I \cup I_0} c}{\Delta \vdash_I \textbf{procedure}\ p(form)c : [p = \mathcal{T}(form)proc]}$$

$$\frac{\Delta \vdash_I ae : aet}{\Delta \vdash_I p(ae)},\ \ \Delta(p) = aetproc$$

$$\frac{form : \Delta_0,\ \ \Delta[\Delta_0] \vdash_{I \cup I_0} c}{\Delta \vdash_I c},\ \ \Delta_0 : I_0$$

$$\frac{form : \Delta_0,\ \ \Delta \vdash_I ae : \mathcal{T}(form)}{\Delta \vdash_I form = ae : \Delta_0}$$

$$\begin{cases} \mathcal{T}(\bullet) = \bullet \\ \mathcal{T}(\texttt{const}\ x : \tau, form) = \tau, \mathcal{T}(form) \\ \mathcal{T}(\texttt{var}\ x : \tau, form) = \tau loc, \mathcal{T}(form) \end{cases}$$

$$\begin{cases} \Delta \vdash_I \bullet : \bullet \\ \dfrac{\Delta \vdash_I e : \tau, \Delta \vdash_I ae : aet}{\Delta \vdash_I e, ae : \tau, aet} \end{cases}$$

$$\begin{cases} \bullet : \emptyset \\ \dfrac{form : \Delta_0}{\texttt{const}\ x : \tau, form\ :\ \Delta_0[x = \tau]},\ \ \Delta_0 : I_0,\ x \notin I_0 \\ \dfrac{form : \Delta_0}{\texttt{var}\ x : \tau, form\ :\ \Delta_0[x = \tau loc]}\ \ \Delta_0 : I_0,\ x \notin I_0 \end{cases}$$

Tabella 6.1: Semantica statica per le procedure di IMP.

procedura e determina la sua ben formatezza. Essa esegue un controllo implicito della corrispondenza dei tipi tra parametri attuali e formali chiedendo che $\Delta(p) = aetproc$ dove la lista di tipi aet è quella associata ai parametri attuali nella premessa, mentre la lista dei tipi associata a p da Δ è determinata al momento della dichiarazione della procedura dalla lista dei parametri formali. Quindi la condizione imposta per l'applicazione della regola garantisce che queste due liste di tipi siano uguali, cioè che $aet = \mathcal{T}(form)$. Per poter applicare la regola della chiamata dobbiamo deteminare la lista aet dei tipi dei parametri attuali. Tale lista è costruita induttivamente partendo dal suo terminatore $\bullet$ e l'assioma $\Delta \vdash_I \bullet : \bullet$ mediante il quinto gruppo di regole. La terza regola controlla che il corpo c della procedura sia ben formato nell'ambiente corrente Δ. Per far questo tiene conto dell'ambiente statico Δ_0 generato dalla lista $form$ dei parametri formali. La quarta regola, infine, associa alla dichiarazione $form = ae$ l'ambiente statico generato da $form$ nel caso in cui le liste $form$ ed ae siano ben formate.

Ci rimane da descrivere come $form$ genera l'ambiente Δ_0. Questo avviene per induzione su $form$ partendo dal suo terminatore $\bullet$ mediante il sesto gruppo di regole. Si noti che non abbiamo prefissato $\Delta \vdash_I$ alla premessa ed alla conclusione delle regole per evidenziare che $FV(form) = \emptyset$ e quindi non dobbiamo mai ricorrere a Δ nella sua elaborazione. L'assioma associa l'ambiente vuoto al terminatore, mentre le altre due regole associano x a τ o τloc a seconda che si abbia una dichiarazione di tipo `const` o di tipo `var`.

In Tab. 6.2 sono riportate le regole per la semantica dinamica delle procedure. Consideriamo prima l'unica regola che riguarda la dichiarazione della procedura che genera l'ambiente dinamico $[p = \lambda form.c']$ senza modificare la memoria. Il valore denotabile che associamo a p è chiamato chiusura o astrazione. Notiamo che, a seconda del tipo di scoping con cui si lavora, c' è definito in modo diverso. In caso di *scoping statico* il corpo c' dell'astrazione è $\rho'; c$ dove c è il corpo della procedura e $\rho' = \rho_{|FI(c) \setminus DI(form)}$. L'ambiente ρ' consente di risolvere i riferimenti agli identificatori liberi in c nella chiamata, utilizzando i legami presenti nell'ambiente dinamico quando la procedura era stata dichiarata. Nel caso dello *scoping dinamico* invece $c' = c$ con c corpo della procedura;

$$\rho \vdash_\Delta \langle \textbf{procedure}\ p(form)c, \sigma\rangle \rightarrow_d \langle [p = \lambda form.c'], \sigma\rangle,$$

$$\begin{cases} c' = \rho_{|FI(c)\setminus DI(form)}; c & \text{scoping statico} \\ c' = c & \text{scoping dinamico} \end{cases}$$

$$\rho \vdash_\Delta \langle p(ae), \sigma\rangle \rightarrow_c \langle form = ae; c', \sigma\rangle, \quad \rho(p) = \lambda form.c'$$

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma\rangle}{\rho \vdash_\Delta \langle (e, ae), \sigma\rangle \rightarrow_{ae} \langle (e', ae), \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma\rangle}{\rho \vdash_\Delta \langle (k, ae), \sigma\rangle \rightarrow_{ae} \langle (k, ae'), \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma\rangle}{\rho \vdash_\Delta \langle form = ae, \sigma\rangle \rightarrow_d \langle form = ae', \sigma\rangle}$$

$$\frac{ak, L \vdash form : \rho_0, \sigma_0}{\rho \vdash_\Delta \langle form = ak, \sigma\rangle \rightarrow_d \langle \rho_0, \sigma[\sigma_0]\rangle}$$

$$\begin{cases} \bullet, L \vdash \bullet : \emptyset, \emptyset \\ \dfrac{ak, L \vdash form : \rho_0, \sigma_0}{(k, ak), L \vdash \texttt{const}\ x : \tau, form : \rho_0[x = k], \sigma_0} \\ \dfrac{ak, L \cup \{l\} \vdash form : \rho_0, \sigma_0}{(k, ak), L \vdash \texttt{var}\ x : \tau, form : \rho_0[x = l], \sigma_0[l = k]}, l \in New_\tau(L_0), \sigma_0 : L_0 \end{cases}$$

Tabella 6.2: Semantica dinamica per le procedure di IMP.

in tal modo al momento dell'esecuzione di c si utilizzeranno solo i valori associati agli identificatori al momento della chiamata della procedura.

La regola successiva definisce la chiamata permettendo di effettuare sia l'associazione tra i parametri formali ed attuali sia di eseguire il corpo dell'astrazione associata alla procedura.

Le due regole successive sono quelle che eseguono la valutazione delle espressioni contenute nella lista dei parametri attuali ottenendo la lista ak, lista dei valori corrispondenti alle espressioni in ae.

Le ulteriori due regole riguardano l'elaborazione di $form = ae$ introdotta dalla regola per la chiamata di procedura. La prima di queste permette di effettuare la valutazione di ae all'interno della dichiarazione $form = ae$. L'altra regola determina l'ambiente ρ_0 e la nuova memoria $\sigma[\sigma_0]$ ottenuti dall'elaborazione di $form = ae$. Per poter applicare queste regole abbiamo bisogno di un predicato $ak, L \vdash form : \rho, \sigma$, con $L \subseteq Loc$, che determina l'ambiente ρ e la memoria σ ottenuti elaborando $form = ae$ isolatamente. Intuitivamente, L è l'insieme di locazioni disponibili per implementare il passaggio dei parametri. L'assioma per il nuovo predicato associa l'ambiente $\emptyset$ e la memoria $\emptyset$ con il terminatore della lista dei formali $\bullet$ indipendentemente dall'insieme L di locazioni. Nel caso di un formale di tipo `const` estendiamo l'ambiente della premessa con il nuovo legame $x = k$ e lasciamo invariata la memoria σ_0. Infine se il parametro è di tipo `var` estendiamo l'ambiente nella premessa con il legame $x = l$, dove l è una nuova locazione di tipo τ contenuta in L che andiamo ad eliminare da L nella conclusione perché non venga più considerata disponobile. Inoltre modifichiamo la memoria inizializzando a k la nuova locazione. Si noti che in entrambi i casi (`const` e `var`) facciamo una copia del parametro attuale nel nuovo ambiente e nella nuova memoria. Quindi ogni modifica eseguita sui parametri all'interno del corpo della procedura non sarà visibile all'esterno. Questo meccanismo di passaggio dei parametri è noto come *passaggio per valore*. La sola differenza tra i parametri `const` ed i parametri `var` è che i primi non possono nemmeno essere modificati nel corpo della procedura.

6.2 Esercizi

Esercizio 6.1. *Dato il programma*

$$
\begin{array}{l}
\textbf{const}\ x : int = 4; \\
\textbf{procedure}\ p(\texttt{var}\ y : int)\ y := y * x; \textbf{print}(y); \\
\textbf{procedure}\ q(\bullet)\ \textbf{const}\ x : int = 2;\ p(2); \\
\textbf{begin} \\
q(\bullet); p(2); \\
\textbf{end}
\end{array}
$$

dove le parole chiave **begin** *e* **end** *servono solo da parentesi sintattiche e la sintassi del comando* **print***, assunta come data, è corrispondente all'intuizione. Dire qual è il suo output in caso di scoping statico e in caso di scoping dinamico e dimostrare la risposta.*

SOLUZIONE. Innanzitutto ricordiamo la differenza tra scoping statico e dinamico; nel primo si risolvono i legami a tempo di compilazione, quindi per le procedure questo significa che esse vengono valutate nell'ambiente della chiamata esteso con l'ambiente presente al momento della loro dichiarazione per risolvere i riferimenti alle variabili libere del corpo. Nello scoping dinamico, invece, i legami vengono risolti a tempo di esecuzione, quindi, per ciò che riguarda le procedure, l'ambiente che si considera è quello presente al momento della loro chiamata.

Scoping statico
Vediamo le derivazioni che vengono eseguite nel caso dello scoping statico, considerando un ambiente dinamico iniziale ρ compatibile con un ambiente di tipi Δ. Per rendere più compatta la scrittura diamo dei nomi

alle dichiarazioni e ai comandi del frammento considerato

$$
\begin{aligned}
&d_1 = \mathbf{const}\ x : int = 4;\\
&d_2 = \mathbf{procedure}\ p(\mathtt{var}\ y : int)\ y := y * x; \mathbf{print}(y);\\
&c_2 = y := y * x; \mathbf{print}(y);\\
&d_3 = \mathbf{procedure}\ q(\bullet)\ \mathbf{const}\ x : int = 2;\ p(2);\\
&c_3 = \mathbf{const}\ x : int = 2;\ p(2);\\
&d = d_2; d_3\\
&c_1 = q(\bullet); p(2);
\end{aligned}
$$

La prima cosa da fare è valutare la dichiarazione d_1

$$
\frac{\dfrac{\rho \vdash_\Delta \langle \mathbf{const}\ x : int = 4, \sigma\rangle \rightarrow_d \langle [x = 4], \sigma\rangle}{\rho \vdash_\Delta \langle \mathbf{const}\ x : int = 4; d, \sigma\rangle \rightarrow_d \langle [x = 4]; d, \sigma\rangle}}{\rho \vdash_\Delta \langle \mathbf{const}\ x : int = 4; d; c_1, \sigma\rangle \rightarrow_c \langle [x = 4]; d; c_1, \sigma\rangle}
$$

A questo punto $\rho_0 = \rho[x = 4]$ è l'ambiente in cui verrà elaborata la dichiarazione della procedura p (compatibile con $\Delta_0 = \Delta[x = int]$)

$$
\frac{\dfrac{\dfrac{\rho_0 \vdash_{\Delta_0} \langle \mathbf{procedure}\ p(\mathtt{var}\ y : int)\ c_2, \sigma\rangle \rightarrow_d \langle [p = \lambda\ \mathtt{var}\ y : int.[x = 4]; c_2], \sigma\rangle}{\rho \vdash_\Delta \langle \rho_0; \mathbf{procedure}\ p(\mathtt{var}\ y : int)\ c_2, \sigma\rangle \rightarrow_d \langle \rho_0; [p = \lambda\ \mathtt{var}\ y : int.[x = 4]; c_2], \sigma\rangle}}{\rho \vdash_\Delta \langle \rho_0; \mathbf{procedure}\ p(\mathtt{var}\ y : int)\ c_2; d_3, \sigma\rangle \rightarrow_d \langle \rho_0; [p = \lambda\ \mathtt{var}\ y : int.[x = 4]; c_2]; d_3, \sigma\rangle}}{\rho \vdash_\Delta \langle \rho_0; \mathbf{procedure}\ p(\mathtt{var}\ y : int)\ c_2; d_3; c_1, \sigma\rangle \rightarrow_c \langle \rho_0; [p = \lambda\ \mathtt{var}\ y : int.[x = 4]; c_2]; d_3; c_1, \sigma\rangle}
$$

Nella astrazione che asociamo a p nell'ambiente dinamico abbiamo inserito il legame $[x = 4]$ perché x è un identificatore libero del corpo c_2 della procedura. Chiamiamo $\rho' = [p = \lambda\ \mathtt{var}\ y : int.[x = 4]; c_2]$, ed essendo le dichiarazioni sequenziali l'ambiente in cui devo valutare la successiva dichiarazione della procedura è $\rho_1 = \rho[\rho_0[\rho']]$ il cui ambiente di tipi è

$\Delta_1 = \Delta_0[p = intproc]$.

$$\frac{\dfrac{\rho_1 \vdash_{\Delta_1} \langle \mathbf{procedure}\ q(\bullet)\ c_3, \sigma\rangle \rightarrow_d \langle [q = \lambda\ \bullet\ .\rho'; c_3], \sigma\rangle}{\rho \vdash_{\Delta} \langle \rho_0[\rho']; \mathbf{procedure}\ q(\bullet)\ c_3, \sigma\rangle \rightarrow_d \langle \rho_0[\rho']; \rho'', \sigma\rangle}}{\rho \vdash_{\Delta} \langle \rho_0[\rho']; \mathbf{procedure}\ q(\bullet)\ c_3; c_1, \sigma\rangle \rightarrow_c \langle \rho_0[\rho']; \rho''; c_1, \sigma\rangle}$$

dove $\rho'' = [q = \lambda\ \bullet\ .\rho'; c_3]$. Qui abbiamo incluso ρ' nell'astrazione associata a q poiché $p = Dom(\rho)$ è un identificatore libero in c_3. Adesso devo eseguire c_1 nell'ambiente $\rho[\rho_0[\rho'[\rho'']]] = \rho_2$. In questo ambiente viene chiamata la procedura q

$$\rho_2 \vdash_{\Delta_2} \langle q(\bullet), \sigma\rangle \rightarrow_c \langle \bullet = \bullet; \rho''; c_3, \sigma\rangle$$

dove $\Delta_2 = \Delta_1[q = \bullet proc]$ è l'ambiente di tipi compatibile con ρ_2, mentre ρ'' è l'ambiente associato a q dalla sua dichiarazione. Adesso quindi in $\rho_3 = \rho_2[\rho'']$ (la dichiarazione $\bullet = \bullet$ non ha alcuna influenza né sull'ambiente né sulla memoria) eseguiamo c_3

$$\frac{\rho_3 \vdash_{\Delta_3} \langle \mathbf{const}\ x : int = 2, \sigma\rangle \rightarrow_d \langle [x = 2], \sigma\rangle}{\rho_3 \vdash_{\Delta_3} \langle \mathbf{const}\ x : int = 2; p(2), \sigma\rangle \rightarrow_c \langle [x = 2]; p(2), \sigma\rangle}$$

dove Δ_3 è l'ambiente di tipi corrispondente a ρ_3, ed è uguale a Δ_2 avendo definito un identificatore già esistente. Notiamo che la memoria σ da cui eseguiamo c_3 è ancora quella iniziale poiché né le dichiarazioni di identificatori costanti né quelle di procedura la influenzano. A questo punto, anche se la chiamata $p(2)$ viene eseguita nell'ambiente $\rho_3[x = 2]$, l'esecuzione della chiamata effettua l'associazione tra formali ed attuali e poi esegue il corpo del comando risolvendo gli identificatori liberi nel blocco individuato dall'ambiente associato alla procedura nel momento della dichiarazione, $[x = 4]$.

$$\rho_3[x = 2] \vdash_{\Delta_3} \langle p(2), \sigma\rangle \rightarrow_c \langle \texttt{var}\ y : int = 2; [x = 4]; c_2, \sigma\rangle$$

dove $[x = 4]$ è l'ambiente registrato nell'astrazione associata a p al momento della sua dichiarazione. Perciò dal corpo di q possiamo notare che al momento della chiamata si ha $y = 2$ ma $x = 4$ e, allora, **print**(y) stampa

in output il valore 8. Tornando al corpo del programma si ha la chiamata $p(2)$ nell'ambiente ρ_3 che assegna il valore 4 a x e quindi il comando **print**(y) restituisce il valore 8.

Scoping dinamico
Riportiamo esplicitamente solo le derivazioni che cambiano in modo rilevante sottolineando la differenza con lo scoping statico.

$$\dfrac{\dfrac{\dfrac{\rho_0 \vdash_{\Delta_0} \langle \textbf{procedure}\ p(\texttt{var}\ y : int)\ c_2, \sigma\rangle \rightarrow_d \langle [p = \lambda\ \texttt{var}\ y : int.c_2], \sigma\rangle}{\rho \vdash_{\Delta} \langle \rho_0; \textbf{procedure}\ p(\texttt{var}\ y : int)\ c_2, \sigma\rangle \rightarrow_d \langle \rho_0; [p = \lambda\ \texttt{var}\ y : int.c_2], \sigma\rangle}}{\rho \vdash_{\Delta} \langle \rho_0; \textbf{procedure}\ p(\texttt{var}\ y : int)\ c_2; d_3, \sigma\rangle \rightarrow_d \langle \rho_0; [p = \lambda\ \texttt{var}\ y : int.c_2]; d_3, \sigma\rangle}}{\rho \vdash_{\Delta} \langle \rho_0; \textbf{procedure}\ p(\texttt{var}\ y : int)\ c_2; d_3; c_1, \sigma\rangle \rightarrow_c \langle \rho_0; [p = \lambda\ \texttt{var}\ y : int.c_2]; d_3; c_1, \sigma\rangle}$$

Notiamo che la dichiarazione della procedura non registra nell'astrazione legata al nome della procedura nessun ambiente e dunque, al momento della chiamata, i suoi identificatori liberi verranno risolti nell'ambiente corrente. Vediamo l'elaborazione della dichiarazione di q, sempre nell'ambiente modificato in modo opportuno dalle dichiarazioni precedenti, cioè in $\rho_1 = \rho[\rho_0[\rho']]$ dove ρ_0 è quello definito in precedenza e adesso $\rho' = [p = \lambda\ \texttt{var}\ y : int.c_2]$

$$\dfrac{\dfrac{\rho_1 \vdash_{\Delta_1} \langle \textbf{procedure}\ q(\bullet)\ c_3, \sigma\rangle \rightarrow_d \langle [q = \lambda\ \bullet.c_3], \sigma\rangle}{\rho \vdash_{\Delta} \langle \rho_0[\rho']; \textbf{procedure}\ q(\bullet)\ c_3, \sigma\rangle \rightarrow_d \langle \rho_0[\rho']; \rho'', \sigma\rangle}}{\rho \vdash_{\Delta} \langle \rho_0[\rho']; \textbf{procedure}\ q(\bullet)\ c_3; c_1, \sigma\rangle \rightarrow_c \langle \rho_0[\rho']; \rho''; c_1, \sigma\rangle}$$

dove $\rho'' = [q = \lambda\ \bullet.c_3]$. In questo caso le procedure non si portano dietro ambienti, dunque quando, dentro q, chiamiamo p si ha $x = 2$ e perciò l'output è $y = 4$. Invece quando p viene chiamata dal corpo del programma abbiamo $x = 4$ e quindi l'output è $y = 8$. Vediamo in particolare le regole delle chiamate di q e p

$$\rho_2 \vdash_{\Delta_2} \langle q(\bullet), \sigma\rangle \rightarrow_c \langle \bullet = \bullet; c_3, \sigma\rangle$$

$$\rho_3[x = 2] \vdash_{\Delta_3} \langle p(2), \sigma\rangle \rightarrow_c \langle \texttt{var}\ y : int = 2; c_2, \sigma\rangle$$

dove $\rho_2 = \rho[\rho_0[\rho'[\rho'']]]$ e $\rho_3 = \rho_2[\rho'']$. □

Esercizio 6.2. *Si consideri il seguente programma*

$$\begin{array}{l}\textbf{var}\ x : bool = tt;\\ \textbf{const}\ y : bool = tt;\\ \textbf{procedure}\ p(\bullet)\ x := z\ \textbf{or}\ y;\\ \textbf{const}\ y : bool = \mathit{ff};\\ p(\bullet)\end{array}$$

Dire quale errore determina la semantica statica fornendo le derivazioni. Ignorando l'errore rilevato staticamente, sotto quali condizioni l'esecuzione del programma non produce un errore a tempo di esecuzione?

SOLUZIONE. Per maggiore chiarezza, assegnamo dei nomi ad ogni dichiarazione:

$$\begin{array}{l}d_1 = \textbf{var}\ x : bool = \mathit{tt};\\ d_2 = \textbf{const}\ y : bool = \mathit{tt};\\ d_3 = \textbf{procedure}\ p(\bullet)\ x := z\ \textbf{or}\ y;\\ d_4 = \textbf{const}\ y : bool = \mathit{ff};\\ d = d_1; d_2; d_3; d_4\end{array}$$

Dobbiamo eseguire un'analisi statica di $d; p(\bullet)$, che rappresenta il programma fornito dal testo. Consideriamo un ambiente di tipi iniziale $\emptyset$ a partire dal quale applichiamo le regole della semantica statica.

$$\frac{\emptyset \vdash_\emptyset d_1 : [x = boolloc],\ [x = boolloc] \vdash_{\{x\}} d_2 : [y = bool]}{\emptyset \vdash_\emptyset (d_1; d_2) : [x = boolloc][y = bool]}$$

$$\frac{\begin{array}{c}\emptyset \vdash_\emptyset (d_1; d_2) : [x = boolloc][y = bool],\\ [x = boolloc][y = bool] \vdash_{\{x,y\}} d_3 : [p = \bullet proc]\end{array}}{\emptyset \vdash_\emptyset (d_1; d_2; d_3) : [x = boolloc][y = bool][p = \bullet proc]}$$

Chiamiamo $\Delta = [x = boolloc, y = bool, p = \bullet proc] = [x = boolloc][y = bool][p = \bullet proc]$ e $I = \{x, y, p\}$; in questo ambiente di tipi dobbiamo esaminare staticamente il corpo della procedura p, che chiamiamo c. La regola di inferenza da utilizzare è

$$\frac{\bullet : \emptyset,\ \Delta \vdash_I c}{\Delta \vdash_I c}, \Delta(p) = \bullet proc$$

Però non riusciamo ad assegnare un tipo a z in quanto non viene definita nel frammento fornito. Dunque il comando c non è ben formato poiché non è ben formato l'assegnamento in esso contenuto. A questo punto dell'analisi statica viene generato un errore.
Vediamo ora come può essere evitato l'errore a livello dinamico. L'errore è generato quando si tenta di valutare la variabile z inserita in una operazione di **or**. Sappiamo però che, in tale operazione, se uno dei due sottotermini che viene valutato è vero, allora il risultato restituito è *tt*, indipendentemente dal valore dell'altro sottotermine che, quindi, può non essere valutato. Perciò per evitare l'errore è necessario sostituire le regole per la valutazione di **or** del nostro linguaggio che impongono un ordine di valutazione da destra a sinistra con le seguenti che impongono l'ordine inverso

$$\frac{\rho \vdash_\Delta \langle e_1, \sigma\rangle \rightarrow_e \langle e_1', \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \textbf{ or } e_1, \sigma\rangle \rightarrow_e \langle e_0 \textbf{ or } e_1', \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e_1, \sigma\rangle \rightarrow_e \langle \mathit{tt}, \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \textbf{ or } e_1, \sigma\rangle \rightarrow_e \langle \mathit{tt}, \sigma\rangle}$$

$$\frac{\rho \vdash_\Delta \langle e_1, \sigma\rangle \rightarrow_e \langle \mathit{ff}, \sigma\rangle}{\rho \vdash_\Delta \langle e_0 \textbf{ or } e_1, \sigma\rangle \rightarrow_e \langle e_0, \sigma\rangle}$$

Le nuove regole non sono sufficienti ad evitare l'errore. Infatti se l'esecuzione della procedura risolve i riferimenti alle variabili libere a partire dall'ambiente dinamico definito nel momento della sua dichiarazione (scoping statico) il valore di y è *tt* e dunque la variabile z non viene valutata

evitando la generazione dell'errore. Se, invece, la procedura viene eseguita con scoping dinamico risulta che y vale *ff* e dunque, quando si valuta l'**or** si cerca di valutare la variabile z generando un errore. Quindi si rende necessario nella nostra situazione utilizzare lo scoping statico. □

Esercizio 6.3. *Modificare* IMP *in modo da poter passare le procedure come parametri di altre procedure. Definire, poi, semantica statica e dinamica delle modifiche apportate.*

SOLUZIONE. Per aggiungere questa nuova possibilità dobbiamo modificare la definizione dei parametri formali ed attuali in modo che ammettano anche procedure. Per quel che riguarda i parametri formali, dobbiamo aggiungere un costrutto che corrisponda alla dichiarazione di procedura; quest'operazione implementa il *Principio di Corrispondenza* in base al quale un linguaggio, per essere il più chiaro possibile deve avere una corrispondenza tra i meccanismi di passaggio dei parametri e i meccanismi di dichiarazione degli identificatori. Quindi aggiungiamo alla sintassi la seguente produzione

$$form ::= \ldots | \texttt{proc}\ p : aetproc, form$$

A questo punto, per mantenere la corrispondenza tra parametri formali e attuali, dobbiamo fare un'operazione analoga nella definizione dei parametri attuali valutati. Il problema che si presenta è quello di trovare un modo adeguato per rappresentare una procedura che si trova tra i parametri di un'altra procedura; per fare questo usiamo le astrazioni, introdotte proprio come "valore" di una procedura

$$ak ::= \ldots | \lambda\ form.c, ak$$

Avendo aggiunto un valore agli attuali dobbiamo modificare anche la definizione dei tipi degli attuali aet

$$aet ::= \ldots \mid aetproc, aet$$

Non dobbiamo invece modificare la definizione degli attuali ae poiché passiamo come parametro procedurale solo l'identificatore di procedura che

poi verrà valutato ad una astrazione. Infine dobbiamo modificare la funzione $\mathcal{T}$ che determina la lista di tipi di $form$ aggiungendo alla definizione per induzione strutturale in Tab. 6.1 il caso

$$\mathcal{T}(\texttt{proc}\ p : aetproc, form) = aetproc, \mathcal{T}(form)$$

Fatte tutte le modifiche sintattiche al linguaggio che permettono di passare le procedure come parametri di altre procedure, definiamo, per i nuovi costrutti, gli identificatori liberi, quelli in posizione di definizione e le nuove regole di inferenza della semantica statica e dinamica.

$$FI(\texttt{proc}\ p : aetproc, form) = \emptyset$$
$$DI(\texttt{proc}\ p : aetproc, form) = \{p\} \cup DI(form)$$

Semantica statica
Prima di tutto dobbiamo assegnare un ambiente di tipi alle dichiarazioni contenute nei parametri formali e quindi dobbiamo dire come viene modificato l'ambiente da un parametro formale che rappresenta una procedura

$$\frac{form : \Delta_0}{(\texttt{proc}\ p : aetproc, form) : \Delta_0[p = aetproc]}, \quad \Delta_0 : I_0,\ p \notin I_0$$

La condizione $p \notin I_0$ assicura che l'identificatore p non compare tra i formali già esaminati.

Semantica dinamica
Vediamo, ora, come viene modificato il predicato sui parametri attuali che definisce l'ambiente dinamico e la memoria associata alla lista degli attuali. La regola che dobbiamo aggiungere consente di estendere l'ambiente nella premessa con il legame tra un identificatore di procedura ed una astrazione lasciando inalterata la memoria.

$$\frac{ak, L \vdash form : \rho_0, \sigma_0}{(\lambda\ form.c, ak), L \vdash (\texttt{proc}\ p : aetproc, form) : \rho_0[p = \lambda\ form.c], \sigma_0}$$

Infine tra le regole di valutazione dei parametri attuali dobbiamo aggiungere quella che valuta i parametri procedurali

$$\rho \vdash_\Delta \langle (x, ae), \sigma \rangle \rightarrow_{ae} \langle (\lambda\ form.c, ae), \sigma \rangle, \quad \rho(x) = \lambda form.c$$

e quella che consente di proseguire la valutazione dopo aver trovato un parametro procedurale

$$\frac{\rho \vdash_{\Delta} \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma'\rangle}{\rho \vdash_{\Delta} \langle(\lambda\ form.c, ae), \sigma\rangle \rightarrow_{ae} \langle(\lambda\ form.c, ae'), \sigma'\rangle}$$

□

Esercizio 6.4. *Modificare* IMP *per consentire il passaggio dei parametri per riferimento; definire la semantica statica e dinamica dei costrutti che vengono aggiunti.*

SOLUZIONE. Ricordiamo che il passaggio per riferimento consiste nel passare come parametri attuali delle locazioni e quindi le modifiche fatte nel corpo della procedura sui parametri attuali saranno visibili anche all'esterno al ritorno dalla chiamata.

Dobbiamo prima modificare la definizione dei parametri formali e di quelli attuali. Ai parametri formali dobbiamo aggiungere una forma che ci permetta di identificare i parametri passati per riferimento; in quelli attuali specifichiamo il fatto che nella lista dei parametri possiamo anche trovare una locazione

$$form ::= \ldots | \texttt{ref}\ x : \tau, form$$
$$ae ::= \ldots | l, ae$$

Definiamo, quindi, il tipo della lista degli attuali

$$aet ::= \ldots | \tau loc, aet$$

A questo punto dobbiamo vedere quali sono gli identificatori liberi e quali sono quelli in posizione di definizione nel nuovo costrutto

$$FI(\texttt{ref}\ x : \tau, form) = \emptyset$$
$$DI(\texttt{ref}\ x : \tau, form) = \{x\} \cup DI(form)$$

Modifichiamo, infine, la funzione di assegnamento di tipo ai parametri formali

$$\mathcal{T}(\texttt{ref}\ x : \tau, form) = \tau loc, \mathcal{T}(form)$$

Possiamo, quindi, definire la semantica dei nuovi costrutti.

Semantica statica
Vediamo cosa succede per i formali. Dobbiamo aggiungere una regola alla definizione del predicato che associa un ambiente statico a $form$ per gestire i parametri per riferimento.

$$\frac{form : \Delta_0}{(\texttt{ref}\ x : \tau, form) : \Delta_0[x = \tau loc]},\ \ \Delta_0 : I_0;\ x \notin I_0$$

Vediamo ora le regole da aggiungere per gli attuali

$$\frac{\Delta \vdash_I ae : aet}{\Delta \vdash_I l, ae : \tau loc, aet},\ \ l \in Loc_\tau$$

Semantica dinamica
La semantica dinamica è un pò più complessa da definire in quanto, per determinare le configurazioni, abbiamo bisogno di capire in quale modo un parametro attuale deve essere utilizzato. Per questo introduciamo l'insieme sintattico MODI con metevariabile μ definito come

$$\mu ::= \bullet | val, \mu | ref, \mu$$

dove val indica il passaggio dei parametri per valore utilizzato in IMP e ref il nuovo passaggio per riferimento. Definiamo, ora la funzione $\mathcal{M} : aet \rightarrow$ MODI come segue

$$\mathcal{M}(\bullet) = \bullet$$
$$\mathcal{M}(\tau, aet) = val, \mathcal{M}(aet)$$
$$\mathcal{M}(\tau loc, aet) = ref, \mathcal{M}(aet)$$

la quale ci permette di associare ad una lista di tipi la lista dei modi in cui i corrispondenti parametri devono essere passati. Definiamo le configurazioni per i parametri attuali

$$\Gamma_{\Delta,\mu} = \{\langle ae, \sigma\rangle | \exists aet.\Delta \vdash_I ae : aet\} \cup \{\langle ak, \sigma\rangle\},\ \ \mathcal{M}(aet) = \mu$$

Definiamo, quindi, la semantica dinamica portandoci dietro, nelle regole, il modo μ nel caso in cui si stia valutando una lista di parametri attuali

$$\rho \vdash_{\Delta,\mu} \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma'\rangle$$

Adesso, se il passaggio è per valore le regole sono analoghe a quelle che abbiamo nel nostro linguaggio

$$\frac{\rho \vdash_{\Delta} \langle e, \sigma\rangle \rightarrow_{e} \langle e', \sigma'\rangle}{\rho \vdash_{\Delta,(val,\mu)} \langle (e, ae), \sigma\rangle \rightarrow_{ae} \langle (e', ae), \sigma'\rangle}$$

$$\frac{\rho \vdash_{\Delta} \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma'\rangle}{\rho \vdash_{\Delta,(val,\mu)} \langle (k, ae), \sigma\rangle \rightarrow_{ae} \langle (k, ae'), \sigma'\rangle}$$

Se invece il passaggio è per riferimento quando incontriamo un identificatore x dobbiamo assegnargli come valore una locazione

$$\rho \vdash_{\Delta,(ref,\mu)} \langle (x, ae), \sigma\rangle \rightarrow_{ae} \langle (l, ae), \sigma'\rangle, l = \rho(x)$$

$$\frac{\rho \vdash_{\Delta} \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma'\rangle}{\rho \vdash_{\Delta,(ref,\mu)} \langle (l, ae), \sigma\rangle \rightarrow_{ae} \langle (l, ae'), \sigma'\rangle}$$

A questo punto possiamo definire le regole che associano parametri formali e attuali. La prima consente di valutare gli attuali garantendo, attraverso i modi, le corrispondenze dei loro tipi con quelli dei formali.

$$\frac{\rho \vdash_{\Delta,\mu} \langle ae, \sigma\rangle \rightarrow_{e} \langle ae', \sigma'\rangle}{\rho \vdash_{\Delta} \langle form = ae, \sigma\rangle \rightarrow_{d} \langle form = ae', \sigma'\rangle}, \quad \mu = \mathcal{M}(\mathcal{T}(form))$$

La regola che consente invece di associare un ambiente alla dichiarazione $form = ae$ non cambia rispetto a quella fornita per IMP. Infine vediamo quale ambiente viene associato al nuovo tipo di parametro formale

$$\frac{ak, L \vdash form : \rho_0, \sigma_0}{(l, ak), L \vdash \texttt{ref}\ x : \tau, form : \rho_0[x = l], \sigma_0}$$

□

Esercizio 6.5. *Dimostrare che l'implicazione dell'Esercizio 4.6 vale anche in presenza di dichiarazioni di procedure e che l'implicazione dell'Esercizio 5.18 vale anche in presenza di chiamate di procedura.*

SOLUZIONE. Iniziamo dimostrando un risultato che servirà successivamente nella dimostrazione. Consideriamo la lista ae e dimostriamo che vale l'implicazione per induzione sulla costruzione della lista, assumendo il risultato dell'Esercizio 3.6. Per la base il risultato è immediato non essendo possibile nessuna transizione da $\bullet$. Consideriamo adesso e, ae. Per ipotesi tale lista è ben formata. Allora per la seguente regola lo sono anche e ed ae

$$\frac{\Delta \vdash_I e : \tau, \Delta \vdash_I ae : aet}{\Delta \vdash_I e, ae : \tau, aet} \tag{6.1}$$

altrimenti non lo sarebbe nemmeno e, ae. Per ciò che riguarda la semantica dinamica abbiamo due possibilità

$$\frac{\rho \vdash_\Delta \langle e, \sigma\rangle \rightarrow_e \langle e', \sigma\rangle}{\rho \vdash_\Delta \langle (e, ae), \sigma\rangle \rightarrow_{ae} \langle (e', ae), \sigma\rangle} \tag{6.2}$$

$$\frac{\rho \vdash_\Delta \langle ae, \sigma\rangle \rightarrow_{ae} \langle ae', \sigma\rangle}{\rho \vdash_\Delta \langle (k, ae), \sigma\rangle \rightarrow_{ae} \langle (k, ae'), \sigma\rangle} \tag{6.3}$$

Se applichiamo la regola (6.2) dobbiamo dimostrare che e', ae è ben formata. Per la regola (6.1) e', ae è ben formata se lo sono le sue componenti ed effettivamante e' lo è per la dimostrazione dell'Esercizio 3.6 mentre ae lo è per ipotesi. Se applichiamo la regola (6.3) dobbiamo dimostrare che k, ae' è ben formata, ma k è una costante e dunque è sempre ben formata, mentre ae' lo è per ipotesi induttiva poiché ae lo è per ipotesi e ae' è ottenuta da ae mediante l'applicazione di una regola dinamica.
Eseguiamo ora la dimostrazione per la dichiarazione di procedura considerando $d =$ **procedure** $p(form)c$. Per ipotesi essa è ben formata e l'unica transizione dinamica possibile è

$$\rho \vdash_\Delta \langle \textbf{procedure}\ p(form)c, \sigma\rangle \rightarrow_d \langle [p = \lambda form.c'], \sigma\rangle$$

che porta ad una configurazione terminale $\langle \rho, \sigma \rangle$ dove l'ambiente dinamico è sempre ben formato.
Consideriamo $d = (form = ae)$, con d ben formata. Allora per la regola

$$\frac{form : \Delta_0, \;\; \Delta \vdash_I ae : \mathcal{T}(form)}{\Delta \vdash_I form = ae : \Delta_0} \tag{6.4}$$

lo sono anche le liste ae e $form$, alla quale riusciamo ad associare l'ambiente Δ_0 (in seguito, per semplicità, quando riusciamo ad associare un ambiente statico alla lista $form$ diremo che $form$ è ben formata). L'unica regola dinamica applicabile è la seguente

$$\frac{\rho \vdash_\Delta \langle ae, \sigma \rangle \rightarrow_{ae} \langle ae', \sigma \rangle}{\rho \vdash_\Delta \langle form = ae, \sigma \rangle \rightarrow_d \langle form = ae', \sigma \rangle}$$

Dobbiamo ora dimostrare che $form = ae'$ è una dichiarazione ben formata e questo vale, per la regola (6.4), se lo sono $form$ e ae'. Ma $form$ lo è per ipotesi perché non è stata alterata, mentre ae' lo è in base alla dimostrazione fatta all'inizio.
Consideriamo infine l'unico comando che interessa le procedure: la chiamata. Consideriamo $c = p(ae)$. Per ipotesi c è ben formato e quindi

$$\frac{\Delta \vdash_I ae : aet}{\Delta \vdash_I p(ae)}, \;\; \Delta(p) = aetproc$$

Dunque ae è ben formato e inoltre $p \in Dom(\Delta)$. Quest'ultimo fatto implica che la dichiarazione di p è ben formata, perciò devono valere le premesse della regola

$$\frac{form : \Delta_0, \;\; \Delta[\Delta_0] \vdash_{I \cup I_0} c}{\Delta \vdash_I \textbf{procedure}\, p(form)c : [p = \mathcal{T}(form)proc]} \tag{6.5}$$

Quindi riusciamo ad associare l'ambiente Δ_0 alla lista $form$ e c, corpo della procedura, è ben formato nell'ambiente $\Delta[\Delta_0]$. Vediamo adesso l'unica regola della semantica dinamica che possiamo applicare

$$\rho \vdash_\Delta \langle p(ae), \sigma \rangle \rightarrow_c \langle form = ae; c', \sigma \rangle, \;\; \rho(p) = \lambda form.c'$$

Dobbiamo quindi dimostrare che il comando $form = ae; c'$ è ben formato, ed esso lo è se lo sono i suoi componenti. Ma $form = ae$ è ben formata grazie alla regola 6.4 perché per ipotesi ae è ben formata e a $form$ riusciamo ad associare un ambiente. Per ciò che riguarda c', sia nel caso di scoping statico che di scoping dinamico, esso è ben formato se lo è il corpo della procedura c, in quanto l'eventuale ambiente dinamico è sempre ben formato. Il corpo della procedura è ben formato se è verificata la seguente

$$\frac{form : \Delta_0,\ \ \Delta[\Delta_0] \vdash_{I \cup I_0} c}{\Delta \vdash_I c},\ \ \Delta_0 : I_0$$

che è vera visto che le premesse sono le stesse della regola (6.5), supposta vera per ipotesi. □

Esercizio 6.6. *Dimostrare che l'implicazione dell'Esercizio 4.7 vale anche in presenza di dichiarazioni di procedure e che quella dell'Esercizio 5.19 vale anche in presenza di chiamate di procedura.*

SOLUZIONE. Assumendo le dimostrazioni degli Esercizi 3.7, 4.7 e 5.19 eseguiamo la dimostrazione analoga cioè dimostriamo la seguente implicazione

$$x_0 \in FI(d)\ \wedge\ x_0 \notin Dom(\Delta)\ \ \Rightarrow\ \ \Delta \not\vdash_I d$$

Consideriamo adesso $d = (form = ae)$ e supponiamo $x_0 \in FI(form = ae) = FI(ae)$. Per definizione l'insieme $FI(ae)$ è l'unione di tutti gli identificatori liberi delle espressioni facenti parte della lista ae. Dunque il fatto che $x_0 \in FI(ae)$ implica che esiste almeno una e contenuta in ae tale che $x_0 \in FI(e)$. Allora per la dimostrazione dell'Esercizio 3.7 si ha che $\Delta \not\vdash_I e$, dunque ae non è ben formata e di conseguenza non lo è nemmeno d poiché non riusciamo ad applicare la regola

$$\frac{form : \Delta_0,\ \ \Delta \vdash_I ae : aet}{\Delta \vdash_I form = ae : \Delta_0}$$

in quanto una delle premesse è falsa.
Consideriamo ora la dichiarazione $d = \mathbf{procedure}\ p(form)c$ con $x_0 \in$

$FI(\mathbf{procedure}\ \ p(form)c) = FI(c) \setminus DI(form)$. Poiché $\Delta[\Delta_0]$ è ottenuto considerando Δ_0 l'ambiente generato da $form$, le condizioni $x_0 \notin DI(form)$ e $x_0 \notin Dom(\Delta)$ implicano che $x_0 \notin Dom(\Delta[\Delta_0])$. Perciò vale il risultato dimostrato nell'Esercizio 5.19 e dunque $\Delta[\Delta_0] \not\vdash_{I \cup I_0} c$ implicando la non ben formatezza della dichiarazione per la regola

$$\frac{form : \Delta_0,\ \ \Delta[\Delta_0] \vdash_{I \cup I_0} c}{\Delta \vdash_I \mathbf{procedure}\, p(form)c : [p = \mathcal{T}(form)proc]}$$

Consideriamo infine il comando $c = p(ae)$ e dimostriamo per induzione

$$x_0 \in FI(c) \setminus Dom(\Delta) \ \ \Rightarrow \ \ \Delta \not\vdash_I c$$

Supponiamo quindi che $x_0 \in FI(c) = FI(ae) \cup \{p\}$. Se $x_0 = p$ la tesi è immediata per la regola

$$\frac{\Delta \vdash_I ae : aet}{\Delta \vdash_I p(ae)},\ \ \Delta(p) = aetproc \tag{6.6}$$

poiché $\Delta(p)$ non è definito e dunque $p(ae)$ non può essere ben formato. Se $x_0 \in FI(ae)$ per i ragionamenti fatti in precedenza $\Delta \not\vdash_I ae : aet$ perciò $\Delta \not\vdash_I p(ae)$ essendo la premessa della regola (6.6) falsa. □

Capitolo 7

Il paradigma funzionale

In questo capitolo introduciamo il paradigma di programmazione funzionale. Al contrario dei linguaggi imperativi visti in precedenza in cui il passo elementare del calcolo è descritto da assegnamenti, nei linguaggi funzionali il passo elementare di computazione viene espresso con l'applicazione di funzione. L'idea intuitiva è quella di rendere la programmazione una attività più vicina alla definizione di funzioni matematiche.

In generale una funzione è una particolare relazione che ad ogni valore del dominio di definizione associa al più un valore del codominio. Intuitivamente una funzione è una regola di trasformazione degli elementi del dominio in quelli del codominio. L'applicazione di una funzione ad un elemento del dominio restituisce come risultato della sua valutazione il corrispondente valore del codominio ed è quindi l'esecuzione della regola di trasformazione.

Il concetto di memoria per descrivere le modifiche effettuate dagli assegnamenti diventa superfluo nel paradigma funzionale, poiché come abbiamo visto il passo elementare di calcolo restituisce un valore (il risultato della funzione). Questa è la differenza fondamentale tra linguaggi imperativi e linguaggi funzionali.

Nelle prossime sezioni richiameremo prima i fondamenti del paradigma funzionale descrivendo la semantica statica e dinamica del λ-calcolo e poi riporteremo la soluzione di alcuni esercizi proposti.

7.1 Il λ-calcolo

Il λ-calcolo fu sviluppato da Church negli anni '30 come base notazionale per descrivere le funzioni matematiche. Successivamente Kleene dimostrò che il λ-calcolo è un sistema di calcolo universale come la macchina di Turing. Infine, McCarthy implementò negli anni '50 il vero linguaggio di programmazione LISP sulla base del λ-calcolo. Adesso esistono numerosi linguaggi funzionali che hanno come nucleo il λ-calcolo. Tra questi ricordiamo tutti i linguaggi della famiglia *ML*, *Haskell*, *Scheme*.

La sintassi del λ-calcolo è molto semplice e comprende soltanto la categoria sintattica delle espressioni. Infatti un programma scritto in un linguaggio funzionale può essere visto come una funzione scritta mediante composizione di funzioni più semplici applicata ad alcuni parametri di ingresso. Il risultato della applicazione è il risultato dell'esecuzione del programma.

L'unico insieme sintattico di base è quello delle

VARIABILI. *Var* con metavariabili $x, y, z, \ldots$.

e l'unica categoria sintattica derivata è quella delle

ESPRESSIONI. *Exp* con metavariabile e definita come

$$e ::= x \mid (ee) \mid (\lambda\, x.e)$$

La concatenazione ee di due λ-termini è chiamata *applicazione* ed il costrutto $\lambda\, x.e$ è chiamato *astrazione*. Il significato intuitivo dell'applicazione $e'e''$ è il passaggio del parametro attuale e'' alla funzione e'. Il significato intuitivo dell'astrazione è quello della definizione di una funzione con parametro formale x e corpo e.

Per semplificare la scrittura dei λ-termini assumeremo nel seguito che l'applicazione associa a sinistra ed ha precedenza sull'astrazione. Inoltre assumiamo che il campo di azione della dichiarazione $\lambda.x$ si estenda a destra quanto più possibile.

7.2 Semantica statica

Come abbiamo fatto per i linguaggi imperativi, definiamo un sistema formale di assiomi e regole di inferenza che permettono di determinare se un λ-termine è corretto rispetto ad una qualche nozione di tipo che sia rilevante per i linguaggi funzionali. Prima però definiamo quali sono le variabili libere e legate dei λ-termini. Nel seguito di questo capitolo useremo la parola termine per riferirci ai λ-termini.

Definizione 7.1 (variabili libere). *L'insieme $FV(e)$ di tutte le* variabili libere *di un termine e è definito per induzione strutturale sulla sintassi dei termini come*

$$
\begin{aligned}
&FV(x) = \{x\} \\
&FV(\lambda x\,.\,e) = FV(e) \setminus \{x\} \\
&FV(ee') = FV(e) \cup FV(e')
\end{aligned}
$$

Una variabile del termine e che non è libera si dice legata*, e l'occorrenza della variabile legata x in $\lambda x\,.\,e$ si chiama* occorrenza di legame.

Per controllare la correttezza dei tipi dei termini associamo nella sintassi del λ-calcolo un tipo a tutte le occorrenze di legame delle variabili. Quindi modifichiamo la sintassi del calcolo sostituendo l'astrazione con $\lambda x : \tau\,.\,e$ che ricopre il ruolo delle dichiarazioni dei linguaggi imperativi. Dobbiamo inoltre definire l'insieme dei tipi che ci occorrono per determinare il tipo di ciascun termine che vogliamo considerare corretto.

Intuitivamente un'astrazione è come la dichiarazione di una procedura con parametro formale x e corpo costituito da una espressione anziché da un comando. Infatti nei linguaggi imperativi l'astrazione con corpo costituito da comandi è il valore denotabile che assegnamo nell'ambiente dinamico agli identificatori di procedura. Nel λ-calcolo la valutazione dell'astrazione deve restituire un valore poiché fa parte della categoria sintattica delle espressioni. Quindi l'astrazione qui è come una funzione matematica che prende certi argomenti in ingresso e restituisce dei valori. Pertanto decidiamo di assegnare alle astrazioni lo stesso tipo delle funzioni matematiche $\tau_0 \to \tau_1$, dove τ_0 è il dominio della funzione (e quindi un tipo in quanto insieme di valori) e τ_1 è il codominio. Possiamo a questo punto definire i tipi che utilizziamo come

TIPI. *Typ* con metavariabile τ definita in stile BNF da

$$\tau ::= int \mid bool \mid \tau \to \tau$$

dove $\to$ è chiamato *costruttore* di tipi.

Le regole della semantica statica che assegnano i tipi ai termini sono riportate in Tab. 7.1. Il simbolo Δ rappresenta un ambiente statico in cui le variabili sono legate ai loro tipi ed è definito come per i linguaggi imperativi. La prima regola assegna ad una variabile il tipo che questa ha associato nell'ambiente statico corrente Δ. La seconda regola ci dice che se il tipo del parametro formale di una astrazione è τ ed il tipo del corpo della astrazione è τ', allora il tipo dell'astrazione è il tipo funzionale $\tau \to \tau'$. Infine l'applicazione ha un tipo corretto se il componente sinistro è una astrazione ed il componente destro ha un tipo uguale al parametro formale dell'astrazione. In questo caso il tipo risultante dall'applicazione è il codominio dell'astrazione.

$$\Delta \vdash x : \Delta(x) \qquad \frac{\Delta[x:\tau] \vdash e : \tau'}{\Delta \vdash \lambda x : \tau . e : \tau \to \tau'}$$

$$\frac{\Delta \vdash e_0 : \tau \to \tau', \Delta \vdash e_1 : \tau}{\Delta \vdash e_0 e_1 : \tau'}$$

Tabella 7.1: Semantica statica del λ-calcolo.

7.3 Semantica dinamica

Per descrivere il comportamento dinamico dei termini dobbiamo per prima cosa introdurre il concetto di sostituzione.

Definizione 7.2 (sostituzione). *Una* sostituzione *è una funzione*

σ : Var $\to e$.

Scriveremo $\{e_1/x_1, \ldots, e_n/x_n\}$*, se* $\sigma(x_i) = e_i$ *per ogni* $i \in [1, n]$ *(e* $\sigma(x_j) = x_j$*,* $j \notin [1, n]$*). Scriveremo anche* $e\{e'/x\}$ *intendendo la sostituzione di* e' *al posto di tutte le occorrenze* libere *di* x *in* e*.*
La sostituzione $e\{e'/x\}$ *è* valida *se soddisfa* $BV(e) \cap FV(e') = \emptyset$*.*

La condizione sulla validità delle sostituzioni è necessaria per evitare il fenomeno della cattura delle variabili che consiste nel trasformare variabili libere in variabili legate come effetto indesiderato di una sostituzione. Nel seguito quando scriviamo una sostituzione assumiamo implicitamente che sia valida.

La semantica dinamica del λ-calcolo è definita in due passi. Prima individuiamo le mosse elementari che possono essere eseguite da alcuni sottotermini e le chiamiamo *conversioni*. Dopo utilizziamo le conversioni come gli assiomi di un sistema di regole che definisce le *riduzioni* che consentono di effettuare le conversioni in tutti i contesti sintattici. Sono quindi le riduzioni che definiscono le transizioni dei λ-termini. Nel seguito chiameremo *redex* (espressione riducibile) un sottotermine di un λ-termine a cui possono essere applicate delle conversioni.

Le conversioni del calcolo sono tre e furono introdotte da Curry che le chiamò α, β e η. Esse sono riportate in Tab. 7.2.

$$\alpha : \lambda x.e \xrightarrow{\alpha} \lambda y.e\{y/x\}, y \notin \mathit{FV}(e)$$

$$\beta : (\lambda x.e)e' \xrightarrow{\beta} e\{e'/x\}$$

$$\eta : \lambda x.(ex) \xrightarrow{\eta} e, x \notin \mathit{FV}(e)$$

Tabella 7.2: Regole di conversione per il λ-calcolo.

La prima regola implementa il cambio di nome delle variabili legate. Il significato semantico dei termini non è influenzato da questa regola che tecnicamente cambia il nome del parametro formale di una astrazione e di tutte le sue occorrenze nel corpo. La seconda regola è quella che implementa il passo di calcolo vero e proprio mediante l'istanziazione del parametro formale x con il parametro attuale e'. Il meccanismo di passaggio

del parametro è basato sulla sostituzione dell'attuale non valutato a tutte le occorrenze del formale (passaggio per nome). La regola β implementa quindi la riscrittura di un termine in un altro rappresentando l'applicazione di una funzione al suo argomento. La regola η implementa l'equivalenza estensionale delle funzioni. Questa ci dice che due funzioni sono equivalenti se restituiscono gli stessi risultati su tutti i possibili argomenti. Infatti sia la parte sinistra che quella destra della conversione una volta applicati ad un termine e' restituiscono ee' poiché il parametro formale x non occorre libero in e.

Il secondo passo nella definizione della semantica dinamica del λ-calcolo serve a permetterci di effettuare conversioni su sottotermini di termini generici. Inoltre, poiché siamo interessati al risultato della valutazione di un'espressione e non ai passi elementari del calcolo, astraiamo dalle particolari conversioni usate definendo una nuova relazione di transizione non etichettata in termini dei tre assiomi α, β e η. La nuova relazione di transizione si chiama *riduzione* ed è definita dalle regole in Tab. 7.3.

$$\frac{e \xrightarrow{\alpha} e'}{e \longrightarrow e'} \qquad \frac{e \xrightarrow{\beta} e'}{e \longrightarrow e'} \qquad \frac{e \xrightarrow{\eta} e'}{e \longrightarrow e'}$$

$$\frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'} \qquad \frac{e \longrightarrow e''}{ee' \longrightarrow e''e'} \qquad \frac{e' \longrightarrow e''}{ee' \longrightarrow ee''}$$

Tabella 7.3: Regole di riduzione per il λ-calcolo.

Il risultato della valutazione di un λ-termine mediante la relazione di riduzione è ancora un λ-termine al quale non è più possibile applicare riduzioni, eccetto, eventualmente, per α-conversioni. Infatti queste ultime non modificano il contenuto informativo di un termine e quindi non producono passi di calcolo effettivi. I termini che verificano questa proprietà sono detti in *forma normale*. Quindi il sistema di transizione del λ-calcolo ha come configurazioni i λ-termini e come relazione di transizione le riduzioni. Le configurazioni terminali sono i λ-termini in forma normale.

Esistono dei λ-termini per cui la scelta dei redex cui applicare le riduzioni puó non condurre ad una forma normale anche se questa esiste. Una strategia di riduzione che ci consente di raggiungere sempre una forma normale se esiste è detta *strategia normalizzante*. Una di queste è la strategia *lo* (leftmost-outermost) che impone di ridurre sempre il redex più a sinistra e più esterno del termine (cioé non contenuto in nessun altro redex).

Riportiamo adesso uno dei principali risultati del λ-calcolo che ci assicura la *confluenza* del calcolo. Essa ci dice che se da un termine è possibile raggiungere attraverso una sequenza di β ed η riduzioni due termini distinti e' ed e'', allora esiste un terzo termine raggiungibile sia da e' che da e''. Formalmente abbiamo il seguente teorema.

Teorema 7.3 (Church-Rosser). *Per tutti i λ-termini e, e' ed e'' tali che $e \twoheadrightarrow e'$ e $e \twoheadrightarrow e''$ esiste un λ-termine e''' tale che $e' \twoheadrightarrow e'''$ e $e'' \twoheadrightarrow e'''$, dove $\twoheadrightarrow$ è la chiusura riflessiva e transitiva della relazione di riduzione senza α-conversione.*

La relazione di riduzione è utilizzata anche per definire una relazione di equivalenza, detta di β-uguaglianza, che è utilizzata per controllare l'uguaglianza semantica di due termini. La β-uguaglianza è definita dalle regole in Tab. 7.4. Le prime tre regole definiscono le proprietà riflessiva, simmetrica e transitiva dell'uguaglianza. La quarta regola dice poi che le riduzioni mantengono la β-uguaglianza. Infine abbiamo le regole di congruenza della β-uguaglianza rispetto ad astrazione ed applicazione.

$$e = e \quad \frac{e = e'}{e' = e} \quad \frac{e = e', e' = e''}{e = e''} \quad \frac{e \rightarrow e'}{e = e'}$$

$$\frac{e = e'}{\lambda x.e = \lambda x.e'} \quad \frac{e_0 = e'_0}{e_0 e_1 = e'_0 e_1} \quad \frac{e_1 = e'_1}{e_0 e_1 = e_0 e'_1}$$

Tabella 7.4: β-uguaglianza.

La β-uguaglianza può direttamente essere definita in termini di riduzioni.

Definizione 7.4. *Due λ-termini e ed e' sono β-*uguali *($e = e'$) se e soltanto se esistono $e_0, \ldots, e_n$ ($n \geq 0$) tali che $e_0 \equiv e$, $e_n \equiv e'$ e*

$$\forall i < n, (e_i \rightarrow e_{i+1} \ \vee \ e_{i+1} \rightarrow e_i).$$

Concludiamo questa sezione trattando funzioni con più di un parametro formale. L'idea è quella di assumere che il risultato dell'applicazione di una funzione è a sua volta una funzione. In questo modo possiamo utilizzare *funzionali* per rappresentare funzioni con più argomenti. Quindi possiamo scrivere in λ-notazione una funzione $f(x, y) = e$ con $FV(e) = \{x, y\}$ come $\lambda x.(\lambda y.e)$. Per calcolare $f(e_0, e_1)$ abbiamo bisogno di due β-riduzioni

$$((\lambda x.(\lambda y.e))e_0)e_1 \xrightarrow{\beta} (\lambda y.e\{e_0/x\})e_1 \xrightarrow{\beta} (e\{e_0/x\})\{e_1/y\}.$$

Questa tecnica prende il nome di *carrizzazione* di funzioni. Per non appesantire la notazione scriveremo nel seguito $\lambda x_1 \ldots x_n.e$ intendendo $\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.e) \ldots)$.

7.4 Esercizi

Esercizio 7.1. *Definire una funzione $BV : e \rightarrow Var$ che determina le variabili legate di un termine e.*

SOLUZIONE. Procediamo per induzione strutturale sulla sintassi dei termini, tenendo conto che l'insieme ottenuto dall'applicazione di BV deve essere il complemento rispetto all'insieme determinato da FV per ogni occorrenza di ogni variabile in e

$$BV(x) = \emptyset$$
$$BV(\lambda x \,.\, e) = BV(e) \cup \{x\}$$
$$BV(ee') = BV(e) \cup BV(e')$$

□

Esercizio 7.2. *Sia e un λ-termine tale che $FV(e) \neq \emptyset$. Dimostrare che $\forall \Delta : V \,.\, FV(e) \not\subseteq V$ abbiamo $\Delta \not\vdash_V e : \tau$.*

SOLUZIONE. Dimostriamo questo risultato per induzione sulla struttura della categoria delle espressioni.

BASE
Consideriamo $e = x$ e supponiamo che $FI(e) = \{x\} \not\subseteq V$, cioè $x \notin V$. Sappiamo che ad un tale termine possiamo assegnare un tipo se vale la seguente regola

$$\Delta \vdash_V x : \Delta(x)$$

però per ipotesi x non appartiene al dominio di Δ e quindi non possiamo determinare il tipo $\Delta(x)$ da cui $\Delta \nvdash_V e : \tau$.

PASSO INDUTTIVO
Supponiamo di poter applicare l'ipotesi induttiva ai termini e_0 ed e_1. Consideriamo $e = e_0 e_1$ e supponiamo che $FI(e) \not\subseteq V$. Questo significa che, per definizione di $FI(e_0 e_1)$, $FI(e_0) \cup FI(e_1) \not\subseteq V$ cioè,

$$FI(e_0) \not\subseteq V \quad \vee \quad FI(e_1) \not\subseteq V$$

Possiamo quindi concludere che ad e non riusciamo ad assegnare un tipo in quanto almeno una delle premesse della seguente regola è falsa

$$\frac{\Delta \vdash_V e_0 : \tau_1 \rightarrow \tau_0, \;\; \Delta \vdash_V e_1 : \tau_1}{\Delta \vdash_V e_0 e_1 : \tau_0}$$

Consideriamo infine $e = \lambda x.e_0$ e supponiamo $FI(e) = FI(e_0) \backslash \{x\} \not\subseteq V$, questo implica che $FI(e_0) \not\subseteq V \cup \{x\}$. Quindi considerando l'ambiente statico $\Delta[x : \tau]$ e il suo dominio $V \cup \{x\}$, possiamo applicare l'ipotesi induttiva ad e_0, cioè $\Delta[x : \tau] \nvdash_{V \cup \{x\}} e_0 : \tau_0$. Possiamo perciò concludere che non riusciamo ad assegnare un tipo al termine e perché la premessa della seguente regola è falsa

$$\frac{\Delta[x : \tau] \vdash_{V \cup \{x\}} e_0 : \tau_0}{\Delta \vdash_V \lambda x : \tau.e_0 \; : \; \tau \rightarrow \tau_0}$$

□

Esercizio 7.3. *Dimostrare che la sostituzione definita per induzione strutturale sulla sintassi dei termini da*

$$y\{e'/x\} = \begin{cases} e' & \textit{se } x \equiv y \\ y & \textit{altrimenti} \end{cases}$$

$$(e_0 e_1)\{e'/x\} = e_0\{e'/x\}e_1\{e'/x\}$$

$$(\lambda y.e_0)\{e'/x\} = \begin{cases} \lambda y.e_0 & \textit{if } x \equiv y \\ \lambda y.(e_0\{e'/x\}) & \textit{se } x \not\equiv y \textit{ e } y \notin \text{FV}(e') \\ \lambda w.(e_0\{w/y\}\{e'/x\}) & \textit{se } x \not\equiv y \textit{ e } y \in \text{FV}(e') \end{cases}$$

dove $w \notin \text{FV}(e_0 \cup e')$ *è sempre valida.*

SOLUZIONE. Ricordiamo inizialmente che una sostituzione $e\{e'/x\}$è valida se $BV(e) \cap FV(e') = \emptyset$. Dunque dimostriamo che la sostituzione è valida per induzione sulla struttura delle espressioni verificando ad ogni passo che è valida la precedente condizione.

BASE
Sia $e = y$ allora $BV(e) = \emptyset$, dunque $\forall e' \,.\, BV(e) \cap FV(e') = \emptyset$, cioè la sostituzione è sempre valida.

PASSO INDUTTIVO
Sia $e = e_0 e_1$ e supponiamo che, per ipotesi induttiva, la sostituzione in e_0 ed e_1 sia valida, cioè $BV(e_0) \cap FV(e') = \emptyset$ e $BV(e_1) \cap FV(e') = \emptyset$. Quindi sapendo che $BV(e) = BV(e_0) \cup BV(e_1)$, vediamo a cosa è uguale $BV(e) \cap FV(e')$

$$\begin{aligned} & BV(e) \cap FV(e') = \text{(per definizione di } BV(e)) \\ & = (BV(e_0) \cup BV(e_1)) \cap FV(e') = \text{(per le proprietà degli insiemi)} \\ & = (BV(e_0) \cap FV(e')) \cup (BV(e_1) \cap FV(e')) = \text{(per ip. induttiva)} \\ & = \emptyset \cup \emptyset = \emptyset \end{aligned}$$

Perciò la sostituzione è valida.
Consideriamo infine $e = \lambda y.e_0$ e supponiamo per ipotesi induttiva che la sostituzione sia valida su e_0, cioè $BV(e_0) \cap FV(e') = \emptyset$. Considerando che per definizione $BV(\lambda y.e_0) = BV(e_0) \cup \{y\}$ vediamo cosa succede caso per caso nella sostituzione.

Se $x \equiv y$ la sostituzione non viene effettuata dunque il problema non si pone.
Se $x \not\equiv y$ distinguiamo ancora in due casi a seconda che y appartenga o meno alle variabili libere di e'. Se $y \notin FV(e')$ abbiamo

$$\begin{aligned}
&BV(e) \cap FV(e') = \text{(per definizione di } BV(e)\text{)} \\
&= (BV(e_0) \cup \{y\}) \cap FV(e') = \text{(per le proprietà degli insiemi)} \\
&= (BV(e_0) \cap FV(e')) \cup (\{y\} \cap FV(e')) = \text{(per ipotesi induttiva)} \\
&= \emptyset \cup (\{y\} \cap FV(e')) = \text{(per ipotesi)} \\
&= \emptyset \cup \emptyset = \emptyset
\end{aligned}$$

Infine se $y \in FV(e')$ per effettuare comunque una sostituzione valida viene applicata una α-conversione al termine $\lambda y.e_0$. In tal modo la sostituzione viene eseguita sul termine $\lambda w.(e_0\{w/y\})$ semanticamente uguale ad e ma con la differenza sintattica che ad y abbiamo sostituito w. A questo punto ricadiamo nel caso precedente in quanto w è stata scelta in modo che non appartenesse alle variabili libere di e' e rendesse valida l'α-conversione su e_0. Poiché $e_0\{w/y\}$ è un termine valido ed è contenuto in e possiamo applicare l'ipotesi induttiva e quindi dedurre cha l'intera sostituzione è valida. □

Esercizio 7.4. *Dato il termine* $(\lambda\, x.(\lambda\, y.xy))((\lambda\, z.zy)(\lambda\, w.w))$*, ridurlo in forma normale costruendo il sistema di transizione corrispondente.*
SOLUZIONE. Come primo passo possiamo applicare una η-conversione al sottotermine $\lambda\, y.xy$, ottenendo la derivazione

$$\dfrac{\dfrac{\dfrac{\lambda\, y.xy \xrightarrow{\eta} x}{\lambda\, y.xy \longrightarrow x}}{(\lambda\, x.(\lambda\, y.xy)) \longrightarrow (\lambda\, x.x)}}{(\lambda\, x.(\lambda\, y.xy))((\lambda\, z.zy)(\lambda\, w.w)) \longrightarrow (\lambda\, x.x)((\lambda\, z.zy)(\lambda\, w.w))}$$

Il primo passo di inferenza trasforma la η-conversione in riduzione in base alla terza regola in Tab. 7.3. Poi applichiamo la regola che consente di effettuare riduzioni nel contesto di una astrazione. Infine la quinta regola in Tab. 7.3 ci consente di ultimare la derivazione della transizione.

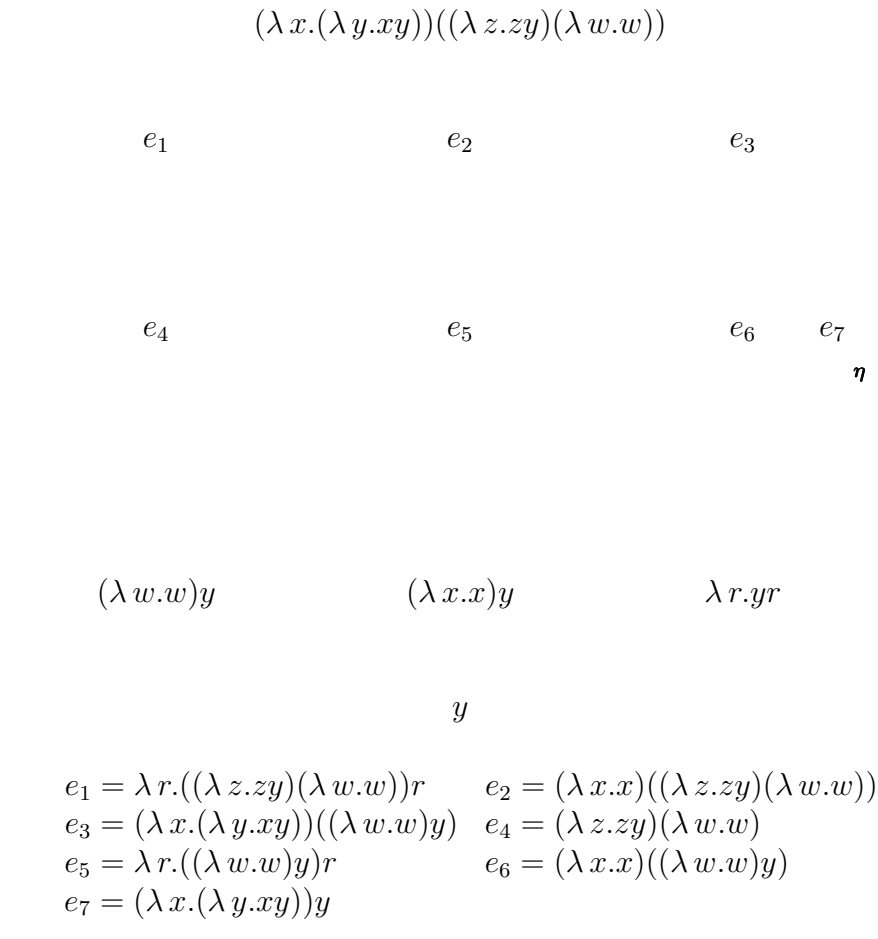

Figura 7.1: Sistema di transizione di $(\lambda\, x.(\lambda\, y.xy))((\lambda\, z.zy)(\lambda\, w.w))$.

Consideriamo adesso la β-conversione che possiamo applicare all'astrazione $\lambda\,.x$ più esterna e più a sinistra (notiamo che questo sarebbe il redex scelto dalla strategia normalizzante *lo*). La derivazione corrispondente è

$$\frac{(\lambda\, x.(\lambda\, y.xy))((\lambda\, z.zy)(\lambda\, w.w)) \xrightarrow{\beta} (\lambda\, y.xy)\{((\lambda\, z.zy)(\lambda\, w.w))/x\}}{(\lambda\, x.(\lambda\, y.xy))((\lambda\, z.zy)(\lambda\, w.w)) \longrightarrow (\lambda\, y.xy)\{((\lambda\, z.zy)(\lambda\, w.w))/x\}}$$

Poiché la variabile y occorre libera nel termine che dobbiamo sostituire alla x, in base alla definizione di sostituzione dell'Esercizio 7.3 dobbiamo ridenominare la y del termine in cui sostituiamo ottenendo

$$(\lambda\, r.((\lambda\, z.zy)(\lambda\, w.w))r).$$

In questo modo la sostituzione evita la cattura della variabile libera y. Notiamo che la sostituzione così definita implementa una α-conversione implicita. Infatti avremmo ottenuto lo stesso risultato effettuando prima una α-riduzione

$$\frac{\dfrac{\dfrac{(\lambda\, y.xy) \xrightarrow{\alpha} (\lambda\, r.xr)}{(\lambda\, y.xy) \longrightarrow (\lambda\, r.xr)}}{(\lambda\, x.(\lambda\, y.xy)) \longrightarrow (\lambda\, x.(\lambda\, r.xr))}}{(\lambda\, x.(\lambda\, y.xy))((\lambda\, z.zy)(\lambda\, w.w)) \longrightarrow (\lambda\, x.(\lambda\, r.xr))((\lambda\, z.zy)(\lambda\, w.w))}$$

e poi la β-riduzione a partire dalla configurazione

$$(\lambda\, x.(\lambda\, r.xr))((\lambda\, z.zy)(\lambda\, w.w)).$$

La sostituzione della β-riduzione è questa volta valida e può quindi essere effettuata normalmente. Proseguendo con le riduzioni otteniamo il sistema di transizione di Fig. 7.1 in cui non abbiamo riportato le α-conversioni. Le transizioni non etichettate sono β-riduzioni. □

Esercizio 7.5. *Sia e un termine e Δ un ambiente statico per cui esiste un tipo τ tale che $\Delta \vdash_V e : \tau$. Dimostrare che se*

1. *$e \xrightarrow{\alpha} e'$ allora esiste τ' tale che $\alpha(\Delta) \vdash_{\alpha(V)} e' : \tau'$;*

2. $e \xrightarrow{\beta} e'$ *allora esiste* τ' *tale che* $\Delta \vdash_V e' : \tau'$*;*

3. $e \xrightarrow{\eta} e'$ *allora esiste* τ' *tale che* $\Delta \vdash_V e' : \tau'$.

SOLUZIONE. Innanzitutto, per semplicità, quando $\exists \tau \,.\, \Delta \vdash_V e : \tau$ diremo che il termine e è ben formato. Inoltre, quando possibile, tralasceremo di specificare in quale ambiente statico il termine è ben formato per evitare di appesantire inutilmente lo svolgimento dell'esercizio. Supponiamo, quindi, che e sia ben formato in Δ e dimostriamo caso per caso che la relazione di riduzione non altera la condizione di ben formatezza dei termini a partire dall'ambiente statico originale. Si noti che tutte e tre le dimostrazioni vanno fatte per induzione sulla struttura delle espressioni ma in tutti i casi è immediata la dimostrazione per la base in quanto nessuna delle conversioni è applicabile ad una variabile quindi per ogni caso vediamo solo il passo induttivo.

1. Consideriamo $e = \lambda x.e_0$. Sapendo che ad e riusciamo ad associare un tipo, valgono le premesse della seguente regola

$$\frac{\Delta[x : \tau] \vdash_{V \cup \{x\}} e_0 : \tau'}{\Delta \vdash_V \lambda x : \tau.e_0 \; : \; \tau \rightarrow \tau'} \tag{7.1}$$

   Quindi ad e_0 riusciamo ad assegnare un tipo in $\Delta[x : \tau]$. La α-conversione esegue la seguente transizione

   $$\lambda x.e_0 \xrightarrow{\alpha} \lambda y.e_0\{y/x\}$$

   Vogliamo quindi dimostrare che il termine $\lambda y.e_0\{y/x\}$ è ben formato nell'ambiente statico $\alpha(\Delta)$. Ma questo è vero se vale la regola 7.1 riscritta per $\lambda y.e_0\{y/x\}$ e con ambiente $\alpha(\Delta)$, cioè è vero se $\alpha(\Delta)[y : \tau] \vdash_{\alpha(V) \cup \{y\}} e_0\{y/x\} : \tau'$. Ma effettivamente $\alpha(\Delta[x : \tau]) = \alpha(\Delta)[y : \tau]$, quindi se in $\Delta[x : \tau]$ associavamo ad x il tipo τ, analogamente in $\alpha(\Delta)[y : \tau]$ associamo il tipo τ a y. Perciò se e_0 è ben formato in $\Delta[x : \tau]$ allora $e_0\{y/x\}$ è ben formato in $\alpha(\Delta)[y : \tau]$.

   Consideriamo adesso $e = e_0 e_1$. Per ipotesi e è ben formata dunque vale la seguente regola

$$\frac{\Delta \vdash_V e_0 : \tau \rightarrow \tau', \;\; \Delta \vdash_V e_1 : \tau}{\Delta \vdash_V e_0 e_1 : \tau'} \tag{7.2}$$

Questo significa che e_0 è un'astrazione e che quindi è applicabile la regola della semantica dinamica

$$\frac{e_0 \xrightarrow{\alpha} e_0'}{e_0 e_1 \longrightarrow e_0' e_1}$$

Quindi se $e_0 = \lambda x.e_2$ si ha che $e_0' = \lambda y.e_2\{y/x\}$. Dimostrare che $e_0' e_1$ è ben formato consiste, per la regola 7.2 riscritta per $e_0' e_1$, nel dimostrare che lo sono le sue componenti. Ma e_1 lo è per ipotesi mentre e_0' lo è per quanto dimostrato in precedenza in quanto per ipotesi e_0 è ben formato.

2. Nel caso in cui $e = \lambda x.e_0$ la β-conversione non è applicabile dunque l'implicazione è sempre vera.
Consideriamo quindi $e = e_0 e_1$.Per ipotesi è ben formata, cioè vale la regola (7.2). Quindi sia e_0 che e_1 sono ben formate in Δ e in particolare e_0 è una astrazione. Per questo possiamo applicare la β-conversione considerando $e_0 = \lambda x.e_0'$ (si noti che essendo e_0 ben formato per ipotesi, allora lo è anche e_0' per la regola 7.1 nell'ambiente $\Delta[x : \tau]$)

$$(\lambda x.e_0')e_1 \xrightarrow{\beta} e_0'\{e_1/x\}$$

A questo punto vogliamo assegnare un tipo a $e_0'\{e_1/x\}$ in Δ, dove sia e_0' che e_1 sono ben formati per ipotesi. Quindi per induzione sulla definizione di sostituzione (vedi Esercizio 7.3) dimostriamo che al termine ottenuto mediante la sostituzione di un termine ben formato in un'altro ben formato riusciamo ad assegnare un tipo.

BASE
Consideriamo $e_0' = y$. Per definizione di sostituzione, se $x \equiv y$ la sostituzione non avviene e quindi il termine che risulta è lo stesso dal quale partiamo che è ben formato per ipotesi. Se, invece, $x \not\equiv y$ il termine risultante è y che, essendo una variabile, è sempre ben formato.

PASSO INDUTTIVO
Sia $e'_0 = e'e''$, dove per l'ipotesi di ben formatezza di e'_0 sia a e' che ad e'' riusciamo ad assegnare un tipo. In questo caso si ha per definizione che $e'e''\{e_1/x\} = e'\{e_1/x\}e''\{e_1/x\}$. Quest'ultimo termine è ben formato se lo sono le sue componenti, e ciò avviene per ipotesi induttiva essendo ben formati sia e' che e''.
Consideriamo infine $e'_0 = \lambda x.e'$ (si noti che l'ipotesi che e'_0 sia ben formato implica che lo sia anche e'), in tal caso si possono verificare tre situazioni differenti.
Se $x \equiv y$ allora non ci sono occorrenze libere di x in e'_0 e dunque la sostituzione non avviene lasciando il termine, ben formato per ipotesi, inalterato. Se invece le due variabili sono differenti dobbiamo ancora distinguere due casi; se $y \notin FV(e_1)$ allora dalla sostituzione si ottiene il termine $\lambda y.e'\{e_1/x\}$ al quale riusciamo ad assegnare un tipo perché $e'\{e_1/x\}$ è ben formato per ipotesi induttiva.
Invece se $y \in FV(e_1)$ il termine che si ottiene $\lambda w.(e'\{w/y\}\{e_1/x\})$ è ben formato se lo è il termine $e'\{w/y\}\{e_1/x\}$. Ma a tale termine riusciamo ad applicare l'ipotesi induttiva perché $e'\{w/y\}$ è ottenuto da un termine ben formato mediante un'α-conversione e, per quanto già dimostrato, il termine che si ottiene è ancora ben formato.

3. Consideriamo infine la η-conversione e prendiamo $e = \lambda x.(e_0 x)$ (in altri tipi di astrazione la conversione non è applicabile). Per ipotesi il termine e è ben formato, cioè vale la regola (7.1) riscritta per e. Questo significa che vale anche la regola (7.2) per l'applicazione $e_0 x$ nell'ambiente $\Delta[x : \tau]$. Riduciamo allora il termine con la η-conversione

$$\lambda x.(e_0 x) \xrightarrow{\eta} e_0 \ , \quad x \notin FV(e_0)$$

Vogliamo vedere se e_0 è ben formato in Δ essendolo per ipotesi in $\Delta[x : \tau]$, ma poiché $x \notin FV(e_0)$ nella valutazione del tipo di e_0 il tipo di x non viene preso dall'ambiente statico e quindi valutare e_0 in Δ o in $\Delta[x : \tau]$ è indifferente.

□

Capitolo 8

Un semplice linguaggio funzionale

In questo capitolo introduciamo la sintassi e la semantica di un semplice linguaggio funzionale mostrando come questo sia costituito essenzialmente dal λ-calcolo arricchito con alcuni zuccheri sintattici. Infatti l'intuizione alla base della programmazione funzionale è quella di vedere un programma come l'implementazione di una relazione tra i valori di ingresso e quelli di uscita. Quindi un programma è una funzione (tipicamente ottenuta per composizione da funzioni più semplici) che abbiamo visto può essere codificata mediante un λ-termine.

Nei linguaggi funzionali la relazione tra gli ingressi e le uscite è ottenuta direttamente mediante i risultati restituiti dalle funzioni. Al contrario, nei linguaggi imperativi i risultati sono ottenuti indirettamente ispezionando la memoria al termine di una esecuzione. Come conseguenza mostreremo che per descrivere il comportamento dinamico dei linguaggi funzionali è sufficiente utilizzare gli ambienti dinamici.

Il capitolo è organizzato riportando la sintassi di un semplice linguaggio funzionale che chiameremo FUN. Poi discuteremo le differenze principali con IMP e forniremo la semantica statica e dinamica di FUN. In questo caso non abbiamo un capitolo per ogni categoria sintattica rilevante perché mostriamo solo le differenze rispetto ad IMP e quindi necessitiamo di minor spazio. Infine riportiamo la soluzione di alcuni esercizi proposti.

8.1 Sintassi

I concetti su cui si fonda il paradigma funzionale sono le espressioni e le funzioni. In particolare, i linguaggi funzionali puri non contengono comandi e quindi non hanno bisogno di una memoria aggiornabile mediante assegnamenti.

Qui richiamiamo brevemente la sintassi del linguaggio di programmazione FUN utilizzando la notazione BNF. Gli insiemi sintattici di base coincidono con quelli di IMP (vedi Sezione 2.2) con la sola eccezione che gli identificatori *Id* sono sostituiti dalle variabili *Var* con metavariabile x (vedi Sezione 8.2).

Le categorie sintattiche delle costanti, dei parametri attuali e degli attuali valutati coincidono con quelle di IMP così come i valori ed i tipi esprimibili. Non avendo in FUN le memorie, le categorie sintattiche dei comandi, dei valori e dei tipi memorizzabili non sono presenti. Le altre categorie sintattiche derivate di FUN sono

ESPRESSIONI. *Exp* con metavariabile e definite come

$$e ::= k \mid x \mid e \,\mathtt{bop}\, e' \mid \mathtt{uop}\, e \mid \mathbf{let}\, d\, \mathbf{into}\, e \mid \\ \mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' \mid f(ae)$$

DEFINIZIONI. *Def* con metavariabile d definite come

$$d ::= \mathbf{nil} \mid x : \tau = e \mid d; d \mid d\ \mathbf{and}\ d \mid d\ \mathbf{in}\ d \mid \mathbf{rec}\ d \mid \\ form = ae \mid \mathbf{function}\ f(form) : \tau = e \mid \rho$$

FORMALI. *Form* con metavariabile $form$ definiti come

$$form\ ::= \bullet \mid x : \tau, form$$

CHIUSURE. *Abs* con metavariabile abs definite come

$$abs ::= \lambda\, form.e : \tau$$

VALORI DENOTABILI. *DVal* $=$ *bool* $\cup$ *int* $\cup$ *Abs* con metavariabile dv definiti come

$$dv ::= k \mid abs$$

TIPI DENOTABILI. *DTyp* $=$ *Typ* $\cup$ $\{$*ATyp* $\rightarrow$ *Typ*$\}$ con metavariabile dt definiti come

$$dt ::= \tau \mid aet \rightarrow \tau$$

TIPI DEGLI ATTUALI. *ATyp* con metavariabile aet definiti come

$$aet ::= \bullet \mid dt, aet$$

Rispetto alle espressioni di IMP abbiamo aggiunto in FUN il blocco per le espressioni, l'espressione condizionale e la chiamata di funzione. Essendo la funzione un'astrazione di espressioni, la sua chiamata cade in questa categoria così come la chiamata di procedura (astrazione di comandi) è nella categoria sintattica *Com*.

Le definizioni sostituiscono le dichiarazioni di IMP poiché non avendo più memoria è superfluo distinguere tra **const** e **var**. In questo contesto è come se tutte le variabili fossero **const** e quindi evitiamo di premettere qualunque parola chiave alla variabile x. L'altra novità è la definizione ricorsiva $\mathbf{rec}\ d$. La peculiarità di questa dichiarazione è che le variabili che si stanno definendo in d vengono considerate come già presenti nell'ambiente in cui si elabora d consentendo di definire variabili in termini di se stesse. Infine la definizione di funzione sostituisce poi la dichiarazione di procedura. Per le funzioni il corpo è un'espressione ed inoltre bisogna esplicitare il tipo del codominio nella definizione.

I formali riflettono le stesse modifiche fatte per le definizioni semplici e quindi non distinguiamo tra `const` e `var`. Inoltre le astrazioni adesso hanno come corpo una espressione e non un comando. Per simmetria con la definizione di funzione, nell'astrazione si registra anche il tipo del codominio.

Dai valori denotabili è scomparsa la locazione l e corrispondentemente τloc dai tipi denotabili. In questi ultimi il tipo funzionale $aet \rightarrow \tau$ sostituisce il tipo $aetproc$ che era necessario per gli identificatori di procedura.

Per concludere, i tipi degli attuali sono costituiti ora a partire dalla base dt anziché et perché prevediamo subito di poter passare funzioni come parametri di altre funzioni.

8.2 Confronto con IMP

Per confrontare IMP e FUN cerchiamo prima di individuare le origini delle caratteristiche fondamentali dei linguaggi imperativi. Questi sono stati progettati e realizzati come astrazioni di architetture convenzionali la cui componente fondamentale è la memoria. Da qui il concetto di identificatore e locazione come indirizzo di una cella di memoria. Questo rende molto diverso un identificatore di un linguaggio imperativo da una variabile di una formula matematica. Infatti gli identificatori (o meglio i valori delle celle di memoria che identificano) possono essere modificati mediante assegnamenti, mentre le variabili no. Una variabile rappresenta una quantità fissata, anche se non nota.

L'altro aspetto fondamentale dei linguaggi imperativi è legato all'iterazione di sequenze di comandi. L'iterazione deriva da una astrazione del ciclo di fetch-execute delle architetture tradizionali. Nella definizione di funzioni matematiche la ripetizione di trasformazioni si implementa mediante il concetto di ricorsione (**rec** d nella sintassi di FUN).

Il paradigma funzionale vuole essere una implementazione delle funzioni matematiche e quindi si basa sul concetto di variabile (e non identificatore e locazione) e di definizione ricorsiva (non iterazione). Non avendo gli identificatori e le locazioni, e quindi di conseguenza la memoria, possiamo definire la macchina astratta di FUN servendoci esclusivamente degli ambienti dinamici che associano variabili e valori (dobbiamo solo sostituire l'insieme degli identificatori con quello della variabili nella definizione di ρ data per i linguaggi imperativi).

Utilizzando il meccanismo della ricorsione per astrarre passi ripetuti del calcolo, i costrutti di controllo della sequenza tipo **while** o **for** sono superflui. Anche la composizione sequenziale non serve in quanto realizzata mediante composizione di funzioni. Ne consegue che la categoria sintattica dei comandi è superflua nei linguaggi funzionali.

Infine sostituiamo le procedure di IMP con le funzioni. Esse hanno

un'espressione come corpo e quindi la loro valutazione restituisce un valore il cui tipo deve essere esplicitato nella dichiarazione della funzione. I meccanismi di passaggio dei parametri sono analoghi a quelli di IMP.

8.3 Semantica di FUN

Come abbiamo fatto per IMP e per il λ-calcolo, cominciamo con la definizione delle variabili libere e legate delle espressioni di FUN.

Definizione 8.1 (variabili libere nelle espressioni). *La funzione*

$$FV : Exp \to Var$$

che ad ogni espressione associa l'insieme delle sue variabili libere è definita per induzione da

$$\begin{aligned}
&FV(k) = \emptyset \\
&FV(x) = \{x\} \\
&FV(e_0 \texttt{ bop } e_1) = FV(e_0) \cup FV(e_1) \\
&FV(\texttt{uop } e) = FV(e) \\
&FV(\textbf{let } d \textbf{ into } e) = FV(d) \cup (FV(e) \setminus BV(d)) \\
&FV(\textbf{if } e \textbf{ then } e_0 \textbf{ else } e_1) = FV(e) \cup FV(e_0) \cup FV(e_1) \\
&FV(f(ae)) = \{f\} \cup FV(ae)
\end{aligned}$$

Per determinare completamente la funzione FV definita sulle espressioni, dobbiamo definire anche le variabili libere che compaiono nelle definizioni.

Definizione 8.2 (variabili libere nelle definizioni). *La funzione*

$$FV : Def \to Var$$

che ad ogni definizione associa l'insieme delle sue variabili libere è defi-

nita per induzione da

$$
\begin{aligned}
&FV(\mathbf{nil}) = \emptyset \\
&FV(x : \tau = e) = FV(e) \\
&FV(d_0; d_1) = FV(d_0 \ \mathbf{in}\ d_1) = FV(d_0) \cup (FV(d_1) \setminus BV(d_0)) \\
&FV(d_0 \ \mathbf{and}\ d_1) = FV(d_0) \cup FV(d_1) \\
&FV(\mathbf{rec}\ d) = FV(d) \setminus BV(d) \\
&FV(form = ae) = FV(ae) \\
&FV(\mathbf{function}\ f(form) : \tau = e) = FV(e) \setminus BV(form) \\
&FV(\rho) = \bigcup_{x \in Dom(\rho)} FV(\rho(x))
\end{aligned}
$$

Notiamo che qui gli ambienti possono avere variabili libere (vedi nel seguito la discussione sulla semantica dinamica della ricorsione e delle funzioni). Poiché queste sono definite puntualmente, dobbiamo definire FV anche sui valori denotabili che sono legati alle variabili negli ambienti.

Definizione 8.3 (variabili libere dei valori denotabili). *La funzione*

$$FV : Dval \to Var$$

che ad ogni valore denotabile associa l'insieme delle sue variabili libere è definita per induzione da

$$FV(k) = \emptyset \qquad FV(\lambda\, form.e : \tau) = FV(e) \setminus BV(form)$$

Infine dobbiamo definire FV anche per i parametri formali ed attuali.

Definizione 8.4 (variabili libere nei parametri). *Le funzioni*

$$FV : Form \to Var \text{ e } FV : AExp \to Var$$

che ad ogni parametro formale ed attuale rispettivamente associano l'insieme delle loro variabili libere sono definite per induzione da

FORMALI	ATTUALI
$FV(\bullet) = \emptyset$	$FV(\bullet) = \emptyset$
$FV(form) = \emptyset$	$FV(e, ae) = FV(e) \cup FV(ae)$

$$\vdash_V n : int \quad \vdash_V t : bool \quad \vdash_V \mathbf{nil} : \emptyset \quad \vdash_V (x : \tau = e) : [x = \tau]$$

$$\frac{\vdash_V d_0 : \Delta_0, \vdash_V d_1 : \Delta_1}{\vdash_V d_0; d_1 : \Delta_0[\Delta_1]} \qquad \frac{\vdash_V d_0 : \Delta_0, \vdash_V d_1 : \Delta_1}{\vdash_V d_0 \ \mathbf{and}\ d_1 : \Delta_0, \Delta_1}$$

$$\frac{\vdash_V d_1 : \Delta_1}{\vdash_V d_0 \ \mathbf{in}\ d_1 : \Delta_1} \qquad \frac{\vdash_V d : \Delta'}{\vdash_V \ \mathbf{rec}\ d : \Delta'}$$

$$\frac{\mathit{form} : \Delta_0}{\vdash_V \mathit{form} = ae : \Delta_0} \qquad \frac{\vdash_V \rho(x) : \Delta(x)}{\vdash_V \rho : \Delta}, \forall x \in W, \rho : W, \Delta : W$$

Tabella 8.1: Semantica statica per le definizioni di FUN:ambienti.

8.3.1 Semantica statica

A questo punto possiamo definire la semantica statica dei costrutti di FUN. Dal punto di vista statico abbiamo ancora bisogno di un ambiente statico Δ definito come per IMP che associa le variabili dei termini con i loro tipi.

Le regole della semantica statica per le espressioni sono le stesse delle corrispondenti espressioni per IMP. Per il blocco la semantica statica è definita dalla regola introdotta nell'Esercizio 4.5. La regola per l'espressione condizionale si trova nell'Esercizio 3.5. Per la chiamata di funzione dobbiamo controllare che il tipo dei parametri attuali coincida con quello dei formali registrato al momento della definizione della funzione nell'ambiente statico e che anche il risultato della funzione sia corretto. La regola che effettua questi controlli è la terza in Tab. 8.3 (abbiamo raccolta in un'unica tabella le regole della semantica statica delle funzioni per omogeneità con la presentazione delle procedure). Per poter applicare la regola abbiamo bisogno di determinare il tipo degli attuali ae. A questo scopo utilizziamo il predicato definito dall'ultimo gruppo di regole in Tab. 8.3.

Cominciamo con la definizione ricorsiva che consente di definire variabili e funzioni in termini di loro stesse. Quindi per determinare il tipo delle variabili e funzioni coinvolte dobbiamo assumere di avere già a di-

sposizione l'ambiente statico generato dall'elaborazione della definizione ricorsiva. In realtà è sufficiente considerare il sottoinsieme dell'ambiente che assegna i tipi alle variabili e funzioni definite in d. A questo scopo una possibile regola è

$$\frac{\Delta[\Delta'_{|V_0}] \vdash_{V \cup V_0} d : \Delta'}{\Delta \vdash_V \mathbf{rec}\ d : \Delta_0}, V_0 = FV(d) \cap BV(d)$$

Questa regola è però di difficile applicazione poiché avendo a disposizione l'ambiente statico corrente Δ e la definizione ricorsiva d da elaborare, dobbiamo indovinare quale sia Δ' per soddisfare la premessa. Cerchiamo di capire a questo punto quale sia il ruolo dell'ambiente statico corrente nell'elaborazione di una definizione.

Per ogni tipo di definizione, il dominio dell'ambiente definito da d è determinato dalla sintassi di d e coincide con $BV(d)$. L'insieme dei tipi da associare alle variabili o funzioni definite in d è espressamente scritto nella sintassi e quindi anche questi possono essere determinati a partire soltanto da d. Notiamo infatti che assegnamo un tipo alle variabili con le definizioni semplici e le definizioni di funzioni. In questo modo possiamo determinare il Δ' generato da d senza utilizzare Δ. Il problema che si può però presentare riguarda la correttezza della definizione. Infatti il tipo di e nelle definizioni semplici o nelle definizioni di funzioni può non coincidere con quello esplicitamente riportato nel costrutto. Per determinare il tipo di e dobbiamo perciò far ricorso all'ambiente corrente Δ. In conclusione l'ambiente corrente Δ è utilizzato solo per controllare la validità di una definizione nel senso che il tipo esplicitamente riportato coincide con quello associato all'espressione di inizializzazione o al corpo della funzione. Quindi spezziamo ogni regola per le definizioni in due che definiscono due predicati distinti. Il primo è della forma $\vdash_V d : \Delta'$ con $FV(d) \subseteq V$ che determina sintatticamente l'ambiente che dovremmo associare alla definizione d se questa fosse valida. Il secondo predicato $\Delta \vdash_V d$ ci dice invece se d è valida oppure no nel senso descritto sopra. Le regole per il primo predicato che genera gli ambienti statici sono riportate in Tab. 8.1, mentre le regole per il predicato che controlla la validità delle definizioni sono in Tab. 8.2. Le prime due regole di ciascuna tabella sono per i valori denotabili $\rho(x)$ necessari nella premessa dell'ultima regola che gestisce il

$$\Delta \vdash_V k \quad \Delta \vdash_V \mathbf{nil} \quad \frac{\Delta \vdash_V e : \tau}{\Delta \vdash_V (x : \tau = e)}$$

$$\frac{\Delta \vdash_V d_0, \Delta \vdash_V d_0 : \Delta_0, \Delta[\Delta_0] \vdash_{V \cup V_0} d_1}{\Delta \vdash_V d_0; d_1}, \Delta_0 : V_0$$

$$\frac{\Delta \vdash_V d_0, \Delta \vdash_V d_1}{\Delta \vdash_V d_0 \ \mathbf{and}\ d_1}, BV(d_0) \cap BV(d_1) = \emptyset$$

$$\frac{\Delta \vdash_V d_0, \Delta \vdash_V d_0 : \Delta_0, \Delta[\Delta_0] \vdash_{V \cup V_0} d_1}{\Delta \vdash_V d_0 \ \mathbf{in}\ d_1}, \Delta_0 : V_0$$

$$\frac{\vdash_V d : \Delta', \Delta[\Delta'_{|V_0}] \vdash_{V \cup V_0} d}{\Delta \vdash_V \ \mathbf{rec}\ d}, V_0 = FV(d) \cap BV(d)$$

$$\frac{\mathit{form} : \Delta_0, \Delta \vdash_V ae : \mathcal{T}(\mathit{form})}{\Delta \vdash_V \mathit{form} = ae}$$

$$\frac{\Delta \vdash_V \rho(x)}{\Delta \vdash_V \rho}, \forall x \in W, \rho : W, \Delta : W$$

Tabella 8.2: Semantica statica per le definizioni di FUN:validità.

$$\vdash_V \ \textbf{function}\ f(\textit{form}) : \tau = e : [f = \mathcal{T}(\textit{form}) \to \tau]$$

$$\frac{\textit{form} : \Delta_0,\ \Delta[\Delta_0] \vdash_{V \cup V_0} e : \tau}{\Delta \vdash_V \ \textbf{function}\ f(\textit{form}) : \tau = e}, \Delta_0 : V_0$$

$$\frac{\Delta \vdash_V ae : aet}{\Delta \vdash_V f(ae) : \tau},\ \Delta(f) = aet \to \tau$$

$$\begin{cases} \mathcal{T}(\bullet) = \bullet \\ \mathcal{T}(x : \tau, \textit{form}) = \tau, \mathcal{T}(\textit{form}) \end{cases}$$

$$\begin{cases} \bullet : \emptyset \\ \dfrac{\textit{form} : \Delta_0}{x : \tau, \textit{form}\ :\ \Delta_0[x = \tau]},\ \ \Delta_0 : V_0,\ x \notin V_0 \end{cases}$$

$$\begin{cases} \Delta \vdash_V \bullet : \bullet \\ \dfrac{\Delta \vdash_V e : \tau, \Delta \vdash_V ae : aet}{\Delta \vdash_V e, ae : \tau, aet} \end{cases}$$

$$\vdash_V (\lambda \textit{form}.e : \tau) : \mathcal{T}(\textit{form}) \to \tau$$

$$\frac{form : \Delta_0, \Delta[\Delta_0] \vdash_{V \cup V_0} e : \tau}{\Delta \vdash_V (\lambda \textit{form}.e : \tau)}, \Delta_0 : V_0$$

Tabella 8.3: Semantica statica per le funzioni di FUN.

caso $d = \rho$ presente nella sintassi. Infine dobbiamo aggiungere alle regole della semantica statica per le espressioni la seguente

$$\frac{\vdash_V d : \Delta_0, \Delta \vdash_V d, \Delta[\Delta_0] \vdash_{V \cup V_0} e}{\Delta \vdash_V \ \mathbf{let}\ d\ \mathbf{into}\ e}$$

Esaminiamo la definizione di funzione. Anche qui, come per tutte le altre definizioni, abbiamo un predicato per la costruzione dell'ambiente statico ed uno per determinare la validità della definizione. Le regole corrispondenti sono le prime due in Tab. 8.3. La funzione $\mathcal{T}$ che associa una lista di tipi ai parametri formali è definita per induzione strutturale dal primo gruppo di regole. Il predicato che consente di associare un ambiente statico ad una lista *form* di parametri formali è invece definito dal secondo gruppo di regole.

Per concludere lo studio della semantica statica di FUN commentiamo le ultime due regole di Tab. 8.3. Esse servono per assegnare un tipo e determinare la buona formatezza delle astrazioni che sono i valori denotabili che associamo agli identificatori di funzione. Queste regole ci consentono di applicare quelle sugli ambienti ρ delle Tab. 8.1 e 8.2 quando vi sono funzioni.

8.3.2 Semantica dinamica

Vediamo adesso il comportamento dinamico di FUN. Come abbiamo già detto più volte, riusciamo a definire la semantica dei linguaggi funzionali senza utilizzare le memorie. Quindi avremo solo un ambiente dinamico che lega variabili e valori denotabili che non contengono piú le locazioni. Quindi dobbiamo riscrivere le regole della semantica dinamica delle espressioni tenendo conto delle nuove strutture. Le regole sono in Tab. 8.4 e 8.5. Esse sono analoghe a quelle per IMP con l'eccezione delle regole del blocco e dell'espressione condizionale che erano state rispettivamente introdotte negli Esercizi 4.5 e 3.5. Le chiamata di funzione è trattata insieme alle altre regole per la dichiarazione di funzione ed il passaggio dei parametri.

Prima di riportare le regole per le definizioni, discutiamo la definizione

$$\rho \vdash_\Delta x \rightarrow_e \rho(x) \qquad \frac{\rho \vdash_\Delta e \rightarrow_e e_0}{\rho \vdash_\Delta e \ \mathtt{bop}\ e' \rightarrow_e e_0 \ \mathtt{bop}\ e'}$$

$$\frac{\rho \vdash_\Delta e' \rightarrow_e e_1}{\rho \vdash_\Delta k \ \mathtt{bop}\ e' \rightarrow_e k \ \mathtt{bop}\ e_1} \qquad \frac{\rho \vdash_\Delta e \rightarrow_e e'}{\rho \vdash_\Delta \mathtt{uop}\ e \rightarrow_e \mathtt{uop}\ e'}$$

$$\rho \vdash_\Delta k \ \mathtt{bop}\ k' \rightarrow_e k'', k'' = k \ \mathtt{bop}\ k'$$

$$\rho \vdash_\Delta \mathtt{uop}\ t \rightarrow_e t', t' = \mathtt{uop}\ t$$

Tabella 8.4: Semantica dinamica per le espressioni di FUN.

ricorsiva. Quello che vorremmo avere è una regola del tipo

$$\frac{\rho[\rho'] \vdash_{\Delta[\Delta']} d \rightarrow_d^* \rho'}{\rho \vdash_\Delta \mathbf{rec}\ d \rightarrow_d^* \rho'}, \rho' : \Delta'$$

Per evitare di dover indovinare l'ambiente dinamico ρ' necessario a soddisfare le premesse della regola, come nel caso statico, utilizziamo due regole. La prima ci dice di elaborare la definizione ricorsiva supponendo di non conoscere niente delle variabili definite ricorsivamente. Questo fatto è espresso nella regola dalla notazione $\rho \setminus V_0$ che formalmente definisce $\rho_{|Dom(\rho)\setminus V_0}$ (vedi Tab. 8.6). Il risultato di tale elaborazione è un ambiente ρ_0 che puó contenere variabili libere. A questo punto possiamo applicare la seconda regola che ci permette di legare le variabili libere di ρ_0 con $\mathbf{rec}\ \rho_0$ (vedi la semantica dinamica della definizione di funzione).

Le altre regole (Tab. 8.7) per le definizioni sono analoghe a quelle già introdotte per le dichiarazioni di IMP.

Per concludere la descrizione della semantica dinamica di FUN ci rimane da trattare la definizione di funzione, la chiamata e l'associazione tra parametri formali ed attuali. Le regole sono tutte in Tab. 8.8.

La prima regola è per l'elaborazione della definizione di funzione. Come già per le procedure di IMP dobbiamo distinguere il caso dello scoping

$$\frac{\rho \vdash_\Delta d \rightarrow_d d'}{\rho \vdash_\Delta \textbf{let } d \textbf{ into } e \rightarrow_e \textbf{let } d' \textbf{ into } e}$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} e \rightarrow_e e'}{\rho \vdash_\Delta \textbf{let } \rho_0 \textbf{ into } e \rightarrow_e \textbf{let } \rho_0 \textbf{ into } e'}, \rho_0 : \Delta_0$$

$$\rho \vdash_\Delta \textbf{let } \rho_0 \textbf{ into } k \rightarrow_e k$$

$$\frac{\rho \vdash_\Delta e \rightarrow_e \hat{e}}{\rho \vdash_\Delta \textbf{if } e \textbf{ then } e' \textbf{ else } e'' \rightarrow_e \textbf{if } \hat{e} \textbf{ then } e' \textbf{ else } e''}$$

$$\frac{\rho \vdash_\Delta e \rightarrow_e \mathit{tt}}{\rho \vdash_\Delta \textbf{if } e \textbf{ then } e' \textbf{ else } e'' \rightarrow_e e'}$$

$$\frac{\rho \vdash_\Delta e \rightarrow_e \mathit{ff}}{\rho \vdash_\Delta \textbf{if } e \textbf{ then } e' \textbf{ else } e'' \rightarrow_e e''}$$

Tabella 8.5: Semantica dinamica per le espressioni di FUN (cont.).

$$\frac{\rho \setminus V_0 \vdash_{\Delta[\Delta_0]} d \rightarrow_d d'}{\rho \vdash_\Delta \textbf{rec } d \rightarrow_d \textbf{rec } d'}, \begin{cases} V_0 = BV(d) \cap FV(d) \\ \vdash_{FV(d)} d : \Delta' \\ \Delta_0 = \Delta'_{|V_0} \end{cases}$$

$$\rho \vdash_\Delta \textbf{rec } \rho_0 \rightarrow_d \{x = k \mid \rho_0(x) = k\} \cup \{f = (\lambda \mathit{form}.\textbf{let } (\textbf{rec } \rho_0) \setminus BV(\mathit{form}) \textbf{ into } e) \mid \rho_0(f) = (\lambda \mathit{form}.e)\}$$

Tabella 8.6: Semantica dinamica per le definizioni ricorsive di FUN.

$$\frac{\rho \vdash_\Delta e \rightarrow_e e'}{\rho \vdash_\Delta x : \tau = e \rightarrow_d x : \tau = e'} \qquad \rho \vdash_\Delta x : \tau = k \rightarrow_d [x = k]$$

$$\frac{\rho \vdash_\Delta d_0 \rightarrow_d d_0'}{\rho \vdash_\Delta d_0; d_1 \rightarrow_d d_0'; d_1} \qquad \frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} d_1 \rightarrow_d d_1'}{\rho \vdash_\Delta \rho_0; d_1 \rightarrow_d \rho_0; d_1'}, \rho_0 : \Delta_0$$

$$\rho \vdash_\Delta \rho_0; \rho_1 \rightarrow_d \rho_0[\rho_1] \qquad \frac{\rho \vdash_\Delta d_0 \rightarrow_d d_0'}{\rho \vdash_\Delta d_0 \textbf{ in } d_1 \rightarrow_d d_0' \textbf{ in } d_1}$$

$$\frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} d_1 \rightarrow_d d_1'}{\rho \vdash_\Delta \rho_0 \textbf{ in } d_1 \rightarrow_d \rho_0 \textbf{ in } d_1'}, \rho_0 : \Delta_0 \qquad \rho \vdash_\Delta \rho_0 \textbf{ in } \rho_1 \rightarrow_d \rho_1$$

$$\frac{\rho \vdash_\Delta d_0 \rightarrow_d d_0'}{\rho \vdash_\Delta d_0 \textbf{ and } d_1 \rightarrow_d d_0' \textbf{ and } d_1} \qquad \frac{\rho \vdash_\Delta d_1 \rightarrow_d d_1'}{\rho \vdash_\Delta \rho_0 \textbf{ and } d_1 \rightarrow_d \rho_0 \textbf{ and } d_1'}$$

$$\rho \vdash_\Delta \rho_0 \textbf{ and } \rho_1 \rightarrow_d \rho_0, \rho_1$$

Tabella 8.7: Semantica dinamica per le definizioni di FUN.

statico e dello scoping dinamico . Il corpo dell'astrazione che leghiamo al nome della funzione nell'ambiente dinamico è costituito dal blocco per le espressioni in cui la parte dichiarativa è un ambiente. Per lo scoping statico tale ambiente lega le variabili libere del corpo della funzione, mentre per lo scoping dinamico è vuoto (le variabili libere saranno risolte nell'ambiente attivo al momento della chiamata). Notiamo l'analogia con la dichiarazione di procedura in cui il corpo dell'astrazione è il blocco per i comandi in cui la parte dichiarativa ha lo stesso ruolo discusso sopra. Oltre al costrutto di blocco per espressioni o comandi, l'altra differenza con le procedure è che qui registriamo nell'astrazione anche il tipo che avrà il valore retituito dalla valutazione della funzione. Il tipo dei parametri è implicitamente presente in *form*. A causa delle definizioni ricorsive, le astrazioni possono contenere delle variabili libere (ricordiamo che la prima regola per la ricorsione non le lega nell'ambiente intermedio $\mathbf{rec}\ \rho_0$). Come conseguenza, anche gli ambienti dinamici possono contenere variabili libere.

La seconda regola è invece per la chiamata di funzione. Anche in questo caso facciamo ricorso al costrutto di blocco per le espressioni in cui la definizione d è istanziata con $form = ae$ per poter generare un ambiente in cui i formali sono associati ai valori ottenuti dalla valutazione degli attuali. Notiamo che il parametro attuale è istanziato nella definizione del blocco al momento della chiamata e quindi viene valutato dopo la chiamata ma prima della valutazione del corpo della funzione. Quindi il meccanismo di passaggio dei parametri è ancora per valore.

Le regole per la valutazione degli attuali sono analoghe a quelle fornite per IMP, così come quelle per i formali. La differenza sostanziale è che non si utilizzano memorie ed abbiamo un unico meccanismo di definizione dei formali.

Cerchiamo di capire il funzionamento dinamico delle definizioni ricorsive con un esempio. Consideriamo la classica definizione ricorsiva del fattoriale

$$\mathbf{rec}\ f(x : int) : int = \ \mathbf{if}\ x = 0\ \mathbf{then}\ 1\mathbf{else}\ x \times f(x-1)$$

Nel primo passo di elaborazione produciamo il legame

$$[f = \lambda x : int.(\mathbf{if}\ x = 0\ \mathbf{then}\ 1\mathbf{else}\ x \times f(x-1)) : int]$$

$$\rho \vdash_\Delta \textbf{function } f(\textit{form}) : \tau = e \rightarrow_d [f = \lambda \textit{form}.\textbf{let } \rho' \textbf{ into } e : \tau],$$

$$\begin{cases} \rho' = \rho_{|FV(e)\setminus BV(\textit{form})} & \text{scoping statico} \\ \rho' = \emptyset & \text{scoping dinamico} \end{cases}$$

$$\rho \vdash_\Delta f(ae) \rightarrow_e \textbf{let } \textit{form} = ae \textbf{ into } e,\ \rho(f) = \lambda \textit{form}.e : \tau$$

$$\frac{\rho \vdash_\Delta e \rightarrow_e e'}{\rho \vdash_\Delta (e, ae) \rightarrow_{ae} (e', ae)} \qquad \frac{\rho \vdash_\Delta ae \rightarrow_{ae} ae'}{\rho \vdash_\Delta (k, ae) \rightarrow_{ae} (k, ae')}$$

$$\frac{\rho \vdash_\Delta ae \rightarrow_{ae} ae'}{\rho \vdash_\Delta \textit{form} = ae \rightarrow_d \textit{form} = ae'}$$

$$\frac{ak \vdash \textit{form} : \rho_0}{\rho \vdash_\Delta \textit{form} = ak \rightarrow_d \rho_0}$$

$$\begin{cases} \bullet \vdash \bullet : \emptyset \\ \dfrac{ak \vdash \textit{form} : \rho}{k, ak \vdash x : \tau, \textit{form} : \rho[x = k]} \end{cases}$$

Tabella 8.8: Semantica dinamica per le funzioni di FUN.

che genera una chiusura per f che però contiene nel corpo la variabile libera f (quella della chiamata ricorsiva). Il secondo passo di elaborazione ci consente di risolvere il riferimento libero alla chiamata del fattoriale sfruttando l'ambiente ρ_0 appena ottenuto. Infatti l'ambiente finale dell'elaborazione del fattoriale è

$$[f = \lambda x : int.(\textbf{let rec}\\ [f = \lambda x : int.(\textbf{if } x = 0 \textbf{ then } 1\textbf{else } x \times f(x-1)) : int] \textbf{ into}\\ (\textbf{if } x = 0 \textbf{ then } 1\textbf{else } x \times f(x-1)) : int]$$

Quello che abbiamo fatto con questi passi di elaborazione è effettuare il primo passo della definizione ricorsiva dicendo come deve essere legata la variabile libera f nella chiusura. In effetti l'elaborazione di $\textbf{rec } \rho_0$ produce un nuovo ambiente ρ_1 in cui si va ad elaborare la definizione. I passi che faremo nell'applicazione della funzione consistono nel valutare l'argomento, nel generare un nuovo ambiente che sarà necessario per la chiamata successiva ed infine nel valutare il corpo.

8.4 Esercizi

Esercizio 8.1. *Definire per ogni categoria sintattica di* FUN *l'insieme delle variabili legate.*

SOLUZIONE. In modo analogo all'Esercizio 7.1 procediamo per induzione strutturale sulla sintassi delle categorie sintattiche di FUN. Definiamo quindi una funzione $BV : Dom \to Var$, $Dom \in \{Exp, Def, Form, AExp\}$, che deve essere il complemento rispetto alle variabili in Dom della corrsipondente FV. Quindi otteniamo

espressioni

$$
\begin{aligned}
&BV(k) = \emptyset \\
&BV(x) = \emptyset \\
&BV(e_0 \texttt{ bop } e_1) = BV(e_0) \cup BV(e_1) \\
&BV(\texttt{uop } e) = BV(e) \\
&BV(\textbf{let } d \textbf{ into } e) = BV(d) \cup BV(e) \\
&BV(\textbf{if } e \textbf{ then } e_0 \textbf{ else } e_1) = BV(e) \cup BV(e_0) \cup BV(e_1) \\
&BV(f(ae)) = \emptyset
\end{aligned}
$$

definizioni

$$
\begin{aligned}
&BV(\textbf{nil}) = \emptyset \\
&BV(x : \tau = e) = \{x\} \\
&BV(d_0; d_1) = BV(d_0 \textbf{ and } d_1) = BV(d_0) \cup BV(d_1) \\
&BV(d_0 \textbf{ in } d_1) = BV(d_1) \\
&BV(\textbf{rec } d) = BV(d) \\
&BV(\textit{form} = ae) = BV(\textit{form}) \\
&BV(\textbf{function } f(\textit{form}) : \tau = e) = \{f\} \\
&BV(\rho) = \bigcup_{x \in Dom(\rho)} BV(\rho(x))
\end{aligned}
$$

valori denotabili

$$BV(k) = \emptyset \qquad BV(\lambda \textit{form}.e : \tau) = BV(e) \cup BV(\textit{form})$$

parametri formali e attuali

FORMALI	ATTUALI
$BV(\bullet) = \emptyset$	$BV(\bullet) = \emptyset$
$BV(x : \tau, \textit{form}) = \{x\} \cup BV(\textit{form})$	$BV(ae) = \emptyset$

□

Esercizio 8.2. *Considerare il frammento di programma*

$$\begin{array}{l}\textbf{let rec function } g(x : int) : int = \\ \qquad\qquad \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } x + g(x-1) \\ \textbf{into } g(2)\end{array}$$

Se l'analisi statica non rileva errori, determinare il valore ottenuto dalla valutazione del frammento nel caso di scoping statico e dinamico, mostrando le derivazioni delle principali transizioni.

SOLUZIONE. Il frammento di programma fornito è un blocco espressione e quindi la regola per l'analisi statica che dobbiamo applicare è quella introdotta nell'Esercizio 4.5 modificata per tenere in considerazione le definizioni ricorsive

$$\frac{\vdash_V d : \Delta_0,\ \Delta \vdash_V d,\ \Delta[\Delta_0] \vdash_{V \cup V_0} e}{\Delta \vdash_V (\textbf{let } d \textbf{ into } e)}, \Delta_0 : V_0$$

Quindi come primo passo dobbiamo determinare l'ambiente statico Δ_0 generato dalla parte definizione. La definizione è una definizione ricorsiva di funzione e quindi prima determiniamo l'ambiente generato dalla funzione e controlliamo la sua validità usando le prime due regole di Tab. 8.3. Da queste premesse generiamo l'ambiente per la definizione ricorsiva e asseriamo la sua validità.

$$\frac{\vdash_V \textbf{function } g(x : int) : int = e : [g = int \to int]}{\vdash_V \textbf{rec function } g(x : int) : int = e : [g = int \to int]}$$

$$\frac{\dfrac{(x : int) : [x = int], \Delta[x = int] \vdash_{V \cup \{x\}} e}{\Delta \vdash_V \textbf{function } g(x : int) : int = e}}{\Delta \vdash_V \textbf{rec function } g(x : int) : int = e}$$

dove abbiamo posto per comodità

$$e = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } x + g(x-1)$$

ed abbiamo omesso la derivazione per $\Delta[x = int] \vdash_{V \cup \{x\}} e$ in quanto del tutto analoga a quelle simili viste per IMP. Adesso, per poter asserire la correttezza del blocco espressione dobbiamo determinare se il corpo della funzione è ben formato nell'ambiente corrente Δ esteso con i legami prodotti dall'elaborazione della definizione ricorsiva. Ma questo controllo è lo stesso della premessa della validità della definizione di funzione, da cui deriviamo la correttezza del frammento.
Adesso applichiamo quindi le regole della semantica dinamica per determinare il valore restituito dalla valutazione del blocco (sono anch'esse riportate nell'Esercizio 4.5 con l'omissione della memoria). Il primo passo di valutazione consiste nel determinare l'ambiente ρ_1 generato dall'elaborazione della definizione ricorsiva. Per questo applichiamo le regole di Tab. 8.6.

$$\frac{\dfrac{\rho \setminus \{g\} \vdash_\Delta \textbf{function}\ g(x : int) : int = e \rightarrowtail_d [g = \lambda\, x : int.\textbf{let}\ \emptyset\ \textbf{into}\ e : int]}{\rho \vdash_\Delta \textbf{rec function}\ g(x : int) : int = e \rightarrowtail_d \textbf{rec}\ [g = \lambda\, x : int.\textbf{let}\ \emptyset\ \textbf{into}\ e : int]}}{\rho \vdash_\Delta \textbf{let rec function}\ g(x : int) : int = e\ \textbf{into}\ g(2) \rightarrowtail_d \textbf{let rec}\ [g = \lambda\, x : int.\textbf{let}\ \emptyset\ \textbf{into}\ e : int]\ \textbf{into}\ g(2)}$$

Notiamo che l'ambiente nel corpo dell'astrazione è vuoto anche se la funzione contiene la variabile libera g. Ma questa è stata tolta dall'ambiente corrente ρ prima di elaborare la definizione in base alle condizioni della prima regola di Tab. 8.6. Poniamo per comodità

$$\rho_0 = [g = \lambda\, x : int.\textbf{let}\ \emptyset\ \textbf{into}\ e : int]$$

Il secondo passo di valutazione permette di determinare l'ambiente ρ_1 generato dalla dichiarazione ricorsiva in base alla seconda regola di Tab. 8.6.

$$\rho \vdash_\Delta \textbf{rec}\ \rho_0 \rightarrowtail_d \rho_1$$

dove

$$\rho_1 = [g = \lambda\, x : int.\textbf{let}\ (\textbf{rec}\ \rho_0) \setminus \{x\}\ \textbf{into}\ e : int]$$

A questo punto terminiamo la valutazione raccogliendo contesto nella conclusione con le regole del blocco in modo standard giungendo nella configurazione

$$\textbf{let } \rho_1 \textbf{ into } g(2).$$

Il passo successivo consiste nel valutare la chiamata di funzione nell'ambiente $\rho[\rho_1]$ (la premessa della regola del blocco che ci consente poi di dedurre il valore della chiamata dall'ambiente ρ). Per la regola della chiamata generiamo un ambiente $[x = 2]$ che va ad estendere quello corrente e in quello risultante si valuta il corpo di g ottenendo

$$\rho[\rho_1][x = 2] \vdash_{V'} \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } x + g(x - 1) \rightarrow_e^* 2 + g(1)$$

La derivazione delle transizioni sopra è analoga a quelle viste per IMP. L'unica differenza è che l'identificatore di funzione g definito ricorsivamente deve essere nuovamente istanziato e la nuova chiamata valutata. Essa sarà valutata nello stesso ambiente $\rho[\rho_1]$ questa volta estesa con il legame $[x = 1]$. Proseguendo le derivazioni si ottiene il valore 3. □

Esercizio 8.3. *Modificare* FUN *per consentire il passaggio dei parametri per nome; definire la semantica statica e dinamica dei costrutti che vengono aggiunti.*

SOLUZIONE. Ricordiamo che il passaggio dei parametri per nome consiste nel valutare solo parzialmente gli argomenti al momento della chiamata di funzione. Più precisamente si valutano gli attuali per legare le loro variabili libere nell'ambiente della chiamata della funzione. Questa soluzione prevede quindi la possibilità di valutare ulteriormente gli attuali durante la valutazione del corpo della funzione.
La soluzione di questo esercizio è simile a quella dell'Esercizio 6.4 in cui si definiscono i parametri passati per riferimento. Come prima cosa estendiamo la categoria sintattica dei formali per consentire di definire il nuovo tipo di passaggio dei parametri. Quindi

$$\textit{form} ::= \ldots \mid \texttt{name}\ x : \tau, \textit{form}$$

Dobbiamo prevedere un nuovo tipo denotabile per poter associare un tipo distinto ai parametri passati per nome da quelli passati per valore

$$dt ::= \ldots | \tau name$$

ed estendiamo conseguentemente la definizione della funzione $\mathcal{T}$ che associa una lista di tipi ai parametri formali con la clausola

$$\mathcal{T}(\texttt{name}\ x : \tau, \textit{form}) = \tau name, \mathcal{T}(\textit{form})$$

A questo punto dobbiamo vedere quali sono le variabile legate e libere nel nuovo costrutto

$$BV(\texttt{name}\ x : \tau, \textit{form}) = \{x\} \cup BV(\textit{form})$$
$$FV(\texttt{name}\ x : \tau, \textit{form}) = \emptyset$$

Possiamo, quindi, definire la semantica dei nuovi costrutti.

Semantica statica
Vediamo cosa succede per i formali. Dobbiamo aggiungere una regola alla definizione del predicato che associa un ambiente statico a *form* per gestire i parametri passati per nome. Il tipo che associamo al parametro è $\tau name$ appositamente introdotto.

$$\frac{\textit{form} : \Delta_0}{(\texttt{name}\ x : \tau, \textit{form}) : \Delta_0[x = \tau name]}, \Delta_0 : I_0;\ x \notin I_0$$

Dobbiamo anche aggiungere un assioma alle espressioni che ci dice che il valore associato a una variabile di tipo $\tau name$ è un valore di tipo τ

$$\Delta \vdash_I x : \tau, \Delta(x) = \tau name$$

Questo assioma ci serve per garantire che nell'ambiente della chiamata il valore di un parametro passato per nome è del tipo corrispondente.
Vediamo ora la regola da aggiungere per la valutazione degli attuali.

$$\frac{\Delta \vdash_I e : \tau, \Delta \vdash_I ae : aet}{\Delta \vdash_I e, ae : \tau name, aet}$$

Notiamo che la precedente regola deve essere *aggiunta* a quelle già presenti in Tab. 8.3 per consentire la valutazione di un attuale. Abbiamo quindi del non determinismo nella valutazione degli attuali poiché abbiamo due regole con le stesse premesse e diverse conclusioni. Tuttavia la sintassi dei formali ed il controllo nella chiamata di funzione consente solo una possibile soluzione e quindi globalmente il meccanismo di associazione tra formali ed attuali rimane deterministico.
Dobbiamo infine aggiungere le regole della semantica statica per i nuovi valori denotabili $e : \tau name$

$$\vdash_V (e : \tau name) : \tau name \qquad \frac{\Delta \vdash_V e : \tau}{\Delta \vdash_V (e : \tau name)}$$

a quelle di Tab. 8.1 e 8.2 rispettivamente.

Semantica dinamica
Come prima cosa dobbiamo estendere l'insieme dei valori denotabili con gli elementi che possono avere tipo $\tau name$. Aggiungiamo un insieme con elementi $e : \tau name$ per ammettere espressioni con variabili libere (quelle eventualmente presenti in e) in presenza di definizioni ricorsive. Notiamo che adesso possiamo avere nella sintassi del nostro linguaggio definizioni del tipo

rec `name` $x : int = \ldots$

Sempre per la presenza di definizioni ricorsive dobbiamo anche modificare le regole per la valutazione delle espressioni e la elaborazione delle definizioni esplicitando l'insieme D delle variabili che sono correntemente legate nell'ambiente dinamico. Avremo quindi regole del tipo

$$\rho \vdash_{\Delta,D} e \rightarrow_e e' \qquad \text{e} \qquad \rho \vdash_{\Delta,D} d \rightarrow_d d'$$

con $\Delta : V$ e $D \subseteq V$ e la compatibilità tra ambiente statico e dinamico vale per $\rho : \Delta_{|D}$. Le variabili in $V \setminus D$ sono quelle definite ricorsivamente.
Per distinguere il passaggio di parametri per nome da quello per valore introduciamo l'insieme sintattico MODI con metevariabile μ già utilizzato nell'Esercizio 6.4 e definito come

$$\mu ::= \bullet | val, \mu | name, \mu$$

$$\vdash_{D,\bullet} T(\bullet)$$

$$\frac{\vdash_{D,\mu} T(ae)}{\vdash_{D,(val,\mu)} T(k, ae)}$$

$$\frac{\vdash_{D,\mu} T(ae)}{\vdash_{D,(name,\mu)} T(e, ae)}, FV(e) \cap D = \emptyset$$

Tabella 8.9: Configurazioni terminali.

dove val indica il passaggio dei parametri per valore utilizzato in FUN e $name$ il nuovo passaggio per nome. Definiamo, ora la funzione $\mathcal{M} : aet \rightarrow$ MODI come segue

$$\mathcal{M}(\bullet) = \bullet$$
$$\mathcal{M}(\tau, aet) = val, \mathcal{M}(aet)$$
$$\mathcal{M}(\tau name, aet) = name, \mathcal{M}(aet)$$

la quale ci permette di associare ad una lista di tipi la lista dei modi in cui i corrispondenti parametri devono essere passati. Definiamo le configurazioni per i parametri attuali

$$\Gamma_{\Delta,D,\mu} = \{ae | \exists aet.\Delta \vdash_I ae : aet, \mathcal{M}(aet) = \mu\}$$

e le configurazioni terminali $T_{\Delta,D,\mu}$ definite mediante il predicato di Tab. 8.9.

Definiamo, quindi, la semantica dinamica portandoci dietro, nelle regole, il modo μ e l'insieme di variabili D nel caso in cui si stia valutando una lista di parametri attuali ottenendo relazioni del tipo

$$\rho \vdash_{\Delta,D,\mu} ae \rightarrow_{ae} ae'$$

Adesso, se il passaggio è per valore le regole sono analoghe a quelle che

abbiamo definito per FUN

$$\frac{\rho \vdash_{\Delta,D} e \rightarrow_e e'}{\rho \vdash_{\Delta,D,(val,\mu)} (e, ae) \rightarrow_{ae} (e', ae)}$$

$$\frac{\rho \vdash_{\Delta,D,\mu} ae \rightarrow_{ae} ae'}{\rho \vdash_{\Delta,D,(val,\mu)} (k, ae) \rightarrow_{ae} (k, ae')}$$

Se invece il passaggio è per nome, quando incontriamo un'espressione e dobbiamo legare le sue variabili libere nell'ambiente corrente ρ

$$\rho \vdash_{\Delta,D,(name,\mu)} (e, ae) \rightarrow_{ae} (\mathbf{let}\ \rho_{|FV(e)}\ \mathbf{into}\ e, ae)$$

La regola seguente ci dice come estendere la valutazione di una lista di attuali quando incontriamo un parametro passato per nome.

$$\frac{\rho \vdash_{\Delta,D,\mu} ae \rightarrow_{ae} ae'}{\rho \vdash_{\Delta,D,(name,\mu)} (e, ae) \rightarrow_{ae} (e, ae')}, FV(e) \cap D = \emptyset$$

Vediamo quale ambiente viene associato ai parametri formali

$$\bullet \vdash_{D,\bullet} \bullet : \emptyset$$

$$\frac{ae \vdash_{D,\mu} form : \rho_0}{k, ae \vdash_{D,(val,\mu)} x : \tau, form : \rho_0[x = k]}$$

$$\frac{ae \vdash_{D,\mu} \mathit{form} : \rho_0}{(e, ae) \vdash_{D,(name,\mu)} (\texttt{name}\ x : \tau, \mathit{form}) : \rho_0[x = e : \tau name]}$$

A questo punto possiamo definire le regole che associano parametri formali e attuali

$$\frac{\rho \vdash_{\Delta,D,\mu} ae \rightarrow_{ae} ae'}{\rho \vdash_{\Delta,D} \mathit{form} = ae \rightarrow_d \mathit{form} = ae'}, \mu = \mathcal{M}(\mathcal{T}(\mathit{form}))$$

$$\frac{ae \vdash_{D,\mu} form : \rho_0}{\rho \vdash_{\Delta,D} form = ae \rightarrow_d \rho_0}, \begin{cases} ae \in T_{\Delta,D,\mu} \\ \mu = \mathcal{M}(\mathcal{T}(form)) \end{cases}$$

□

Capitolo 9

Conclusioni

In questo testo abbiamo descritto i concetti fondamentali della semantica operazionale strutturale e li abbiamo applicati ai linguaggi imperativi e funzionali. La descrizione non ambigua dei costrutti principali dei linguaggi di programmazione dovrebbe consentire ai progettisti di prevedere l'effetto dei costrutti che stanno definendo, agli implementatori di produrre compilatori fedeli alla definizione dei linguaggi ed agli utenti di utilizzare in modo professionale le potenzialità offerte da un linguaggio.

La semantica operazionale tuttavia non è il solo modo di descrivere formalmente un linguaggio di programmazione. Esistono altre tecniche che richiameremo brevemente nella prossima sezione. Qui vogliamo solo ricordare che la nostra scelta è stata guidata dall'esigenza pressante per un testo didattico di utilizzare un formalismo matematicamente semplice e vicino all'intuizione di chi abbia almeno qualche volta programmato. Tuttavia la soluzione scelta consente di gestire tutti gli aspetti rilevanti dei linguaggi [7, 20].

Dopo aver brevemente richiamato altre tecniche di definizione semantica, nelle sezioni seguenti commentiamo le relazioni che intercorrono tra la nostra trattazione ed i linguaggi di programmazione reali. Infine riportiamo alcuni suggerimenti per chi voglia approfondire gli aspetti trattati nel testo ed alcuni ad essi correlati.

9.1 Altri approcci

Oltre la semantica operazionale che abbiamo trattato in questo testo, esistono altri approcci per definire formalmente il significato dei costrutti dei linguaggi. Tra le tecniche più diffuse ricordiamo la semantica denotazionale e quella assiomatica. Vediamo quali sono gli aspetti peculiari delle varie tecniche.

- *Semantica operazionale*. Il significato del costrutto di un linguaggio è dato dalle operazioni che questo induce sulla macchina astratta del linguaggio. Quindi la semantica operazionale descrive *come* è ottenuto l'effetto dell'esecuzione di un costrutto.

- *Semantica denotazionale*. Il significato di un costrutto di un linguaggio è dato da un oggetto matematico. Quindi la semantica denotazionale descrive solamente l'*effetto* dell'esecuzione del cosrtutto.

- *Semantica assiomatica*. Le proprietà salienti dell'esecuzione di un costrutto sono espresse mediante asserzioni logiche. Quindi, *alcuni aspetti* dell'esecuzione e dei suoi effetti possono essere ignorati da questa semantica.

Vogliamo ricordare che i differenti approcci per la definizione formale dei linguaggi di programmazione non sono mutuamente esclusivi e non sono in competizione l'uno con l'altro. Ad esempio la semantica operazionale è più adatta delle altre tecniche a fornire suggerimenti a chi deve effettivamente implementare un linguaggio e a gestire aspetti connessi con la concorrenza ed i sistemi distribuiti [19]. La semantica denotazionale è invece migliore per ragionare formalmente sui programmi e studiarne proprietà. La semantica assiomatica fornisce un sistema logico che può essere utilizzato per dimostrare proprietà in modo semi-automatico. Abbiamo però visto nel testo che la semantica operazionale può essere definita anch'essa in stile logico recuperando quindi tutti gli aspetti essenziali dell'approccio assiomatico.

9.2 I linguaggi reali

Nel presentare i linguaggi imperativi e funzionali abbiamo distinto gli aspetti caratterizzanti degli uni (assegnamento e procedure) e degli altri (espressioni e funzioni) trattandoli separatamente. Ancora una volta l'esigenza didattica ci ha guidato in questa scelta. La realtà dei linguaggi di programmazione è ben diversa.

Praticamente tutti i linguaggi imperativi dispongono di costrutti simili alle funzioni. Di solito sono astrazioni il cui corpo è costituito da comandi e che sono anche in grado di restituire un valore. Poiché le espressioni fanno parte di tutti i linguaggi, nessuno vieta di poter scrivere un programma di stile funzionale utilizzando un linguaggio tipo Pascal. Il paradigma di programmazione imperativo è pensato tuttavia per esprimere modifiche della memoria e quindi i programmi che si scrivono non sono certo funzionali.

I linguaggi funzionali dal canto loro sono più vicini al linguaggio FUN che abbiamo utilizzato. Infatti l'esigenza di avere l'assegnamento e strutture simili alla memoria è meno sentita e di solito è bene non utilizzarle in questo contesto. In generale i linguaggi funzionali della famiglia *ML* dispongono di costrutti tipo assegnamento ed iterativi, ma poiché il paradigma impone uno stile matematico di definizione dei programmi, la loro presenza nel linguaggio è molto più marginale di quanto non lo sia quella delle funzioni nei linguaggi imperativi.

In conclusione, anche se abbiamo separato i concetti tipici dei due paradigmi, il lettore dovrebbe aver acquisito a questo punto la familiarità con la semantica operazionale strutturale per poter definire linguaggi ibridi e studiarne il comportamento sia statico che dinamico.

9.3 Approfondimenti

In questa sezione riportiamo alcuni riferimenti ad altri testi sulla semantica dei linguaggi di programmazione che possono essere utilizzati per approfondire alcuni degli aspetti trattati. Una introduzione molto dettagliata agli aspetti semantici dei linguaggi di programmazione si trova in [22, 23], benché questi libri non siano prevalentemente orientati alla semantica ope-

razionale. Per quanto riguarda la semantica operazionale ricordiamo, oltre al lavoro di Plotkin [18] sul quale ci siamo prevalentemente basati nella stesura del presente lavoro, anche il libro [16] in cui si descrive la semantica operazionale di un linguaggio imperativo e si discutono alcune tecniche di verifica di proprietà dei programmi. Si può trovare in questo testo anche un'introduzione alla semantica assiomatica. Un altro testo che tratta la semantica operazionale dei linguaggi imperativi è [11]. In entrambi i libri citati non viene discusso l'aspetto statico legato ai linguaggi e nel secondo si tratta solo lo scoping dinamico.

Gli aspetti legati alla definizione operazionale dei sistemi di tipi per i linguaggi di programmazione sono affrontati in dettaglio in [8, 3].

La semantica denotazionale dei lingauggi di programmazione ed un confronto con la semantica operazionale e assiomatica sono trattati in [25] dove vi è anche una trattazione dell'induzione strutturale e sugli alberi di derivazione. Quest'ultima può essere ulteriormente approfondita in [1] e [5]. Altri testi in cui si affrontano tematiche di semantica denotazionale sono [14, 21, 9, 24].

I testi base per approfondire gli aspetti legati ai fondamenti della programmazione funzionale ed al λ-calcolo sono [2, 10]. Per quanto riguarda invece i veri linguaggi di programmazione funzionali, sono ottimi riferimenti per approfondimenti [17, 15].

Bibliografia

[1] P. Aczel. An introduction to inductive definitions. In *Handbook of mathematical logic*. North-Holland, 1977.

[2] H.P. Barengregt. *The Lambda Calculus*. North-Holland, 1984.

[3] L. Cardelli. *CRC Handbook of Computer Science and Engineering*, chapter 103, Type systems. CRC Press, 1997.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, december 1995.

[5] P Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of POPL'92*, pages 84–95, 1992.

[6] J.W. de Bakker and W.P. de Roever. A calculus for recursive program schemes. In *Proceedings of ICALP'72*, pages 167–196. North-Holland, 1972.

[7] P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, 1996.

[8] H. Goguen. *A typed operational semantics for type theory*. PhD thesis, University of Edinburgh, 1994.

[9] M.J.C. Gordon. *The denotational description of programming languages*. Springer-Verlag, 1979.

[10] J.R. Hindley and J.P. Seldin. *Introduction to combinators and λ-calculus*. Cambridge University Press, 1986.

[11] C. Laneve. *La descrizione operazionale dei linguaggi di programmazione*. Franco Angeli, 1998.

[12] P. Lucas. Formal definition of programming languages and systems. In *Proceedings of IFIP'71*, 1971.

[13] J. McCarthy. Towards a mathematical science of computation. In *Information Processing 1962*, pages 21–28, 1963.

[14] R.E. Milne and C. Strachey. *A theory of programming language semantics*. Chapman and Hall, 1976.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of standard ML (revised)*. MIT Press, 1997.

[16] F. Nielson and H.R. Nielson. *Semantics with applications: a formal introduction*. Wiley, 1992.

[17] S.L. Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.

[18] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[19] C. Priami. *Enhanced Operational Semantics for Concurrency*. PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1996. Available as Tech. Rep. TD-08/96.

[20] C. Priami. Operational methods in theoretical computer science. *ACM Computing Surveys*, 1999. To appear.

[21] J.E. Stoy. *The Scott-Strachey approach to programming language theory*. MIT Press, 1977.

[22] R.D. Tennent. *Principles of Programming Languages*. Prentice Hall, 1981.

[23] R.D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.

[24] D.A. Watt. *Programming Language Semantics*. Prentice Hall, 1990.

[25] G. Winskel. *The formal semantics of programming languages*. MIT Press, 1993.

Indice analitico