ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

MODELLAZIONE DELLA FAMIGLIA DI LINGUAGGI SOS

Tesi di Laurea in Paradigmi di programmazione

Relatore:
Chiar.mo Prof.
SIMONE MARTINI

Correlatore:
Ill.mo Prof.
RICCARDO SOLMI

Presentata da:
ENRICO RASPADORI

Sessione II
Anno Accademico 2009/2010

A i miei genitori per la comprensione,
A Bibi per l'amore,
Ai miei nonni per ciò che sono,
Ai buoni amici,
A Riccardo per la pazienza

Indice

1 La famiglia dei linguaggi SOS **7**
- 1.1 Panoramica storica 8
- 1.2 Labelled Transition System 10
  - 1.2.1 Generalità sui Labelled Transition System e utilizzo in SOS 10
  - 1.2.2 Termini 10
- 1.3 Varietà Sintattica 12
  - 1.3.1 Signature multi-sorta 13
  - 1.3.2 Labels 14
  - 1.3.3 Premesse negative 17
  - 1.3.4 Premesse infinite 19
  - 1.3.5 Predicati 20
  - 1.3.6 Appoggio ad altri framework 20
  - 1.3.7 Condizioni di regola 24
- 1.4 SOS nei linguaggi di programmazione 25
  - 1.4.1 Sistemi di transizione terminali 25
  - 1.4.2 Ambienti 27
- 1.5 Framework Derivati 31
  - 1.5.1 Semantica Naturale 31
  - 1.5.2 SOS Modulare 31
- 1.6 Utilizzo SOS 34
  - 1.6.1 Domini di ricerca 34
  - 1.6.2 Esempi 36
  - 1.6.3 Peso delle feature 48

2 Tool **54**
- 2.1 Panoramica sui tool esistenti 54
- 2.2 Il Sistema Centaur 54
  - 2.2.1 La storia 54
  - 2.2.2 Il linguaggio METAL 56
  - 2.2.3 Il linguaggio TYPOL 58

2.2.4 Riepilogo 60
2.3 Rml e tools 61
2.3.1 Il linguaggio rml 61
2.3.2 Sintassi RML 62
2.3.3 Structural Operational Semantics Development Tooling - Eclipse Plugin 64
2.3.4 Riepilogo 65
2.4 Maude e tools 65
2.4.1 Il sistema Maude 65
2.4.2 Sintassi Maude 66
2.4.3 Maude MSOS Tool 70
2.4.4 Riepilogo 75

Bibliografia **75**

Elenco delle figure

1.1 Grafo delle dipendenze con premesse negative 19
1.2 Sintassi Jinja-1 37
1.3 Predicati ammissibili come premesse all'interno delle regole di transizione in Jinja 37
1.4 Descrizione dello stato presente nelle configurazioni 38
1.5 Funzioni semantiche utilizzate nelle transizioni 38
1.6 Esempi di regole di transizione in Jinja 39
1.7 Grammatica stile BNF dei termini del calcolo 41
1.8 Le label ammissibili nelle transizioni del π-calcolo 41
1.9 Le regole di transizione del π-calcolo 42
1.10 Sintassi Astratta del core di ABS 44
1.11 Sintassi astratta della parte funzionale di ABS 45
1.12 Regole transizionali della parte funzionale del core di ABS . . 46
1.13 Sintassi dei termini del linguaggio object-oriented di core ABS presenti nelle regole di transizione 47
1.14 Regole di transizione per la parte object-oriented di ABS(1) . 47
1.15 Regole di transizione per la parte object-oriented di ABS(2) . 48
1.16 Riassunto della varietà sintattica dei linguaggi della famiglia SOS 53

2.1 Regole di transizione per gli operatori aritmetici in TYPOL . 59
2.2 Regole semantiche per l'ambiente 60
2.3 Uno screenshot del tool SOSDT 65

Elenco delle tabelle

1.1 Ambiti di utilizzo SOS 36
1.2 Riepilogo feature sintattiche presenti nelle regole di transizione di Jinja 40
1.3 Riepilogo feature sintattiche presenti nelle regole di transizione del π-calcolo 43
1.4 Riepilogo feature sintattiche presenti nelle regole di transizione di core ABS 49

2.1 Riepilogo feature sintattiche presenti nel sistema Centaur . . . 61
2.2 Riepilogo feature sintattiche presenti nel sistema RML 66
2.3 Riepilogo feature sintattiche presenti nel tool MSOS 75

Capitolo 1

La famiglia dei linguaggi SOS

La parola ‘semantica’ (dal greco semantiká), utilizzata negli ambiti più svariati, dalla psicologia alla logica, dalla linguistica all’informatica, e con le connotazioni più varie, dal popolare all’altamente tecnico, identifica lo studio dei rapporti che intercorrono fra i segni ed il loro significato. In particolare, all’interno della linguistica, la semantica si pone in relazione con la sintassi, la morfologia e la fonologia, che insieme formano la grammatica, legando le parole, frasi e sintagmi correttamente redatti secondo le regole grammaticali, a significati appartenenti al dominio della realtà extra-linguistica.

Se nel caso dei linguaggi naturali questo processo può essere assai complesso, nel caso dei linguaggi di programmazione la situazione è assai più semplice. In questo caso la semantica si configura come una relazione fra un insieme di frasi corrette ed entità autonome indipendenti che utilizziamo per descriverle, fra programmi composti da costrutti conformi grammaticalmente alla propria specifica ed una specifica descrizione del loro significato. L’attribuzione di semantica ad un linguaggio può avvenire attraverso politiche differenti.

Posso scegliere di descrivere il significato dei costrutti ‘a parole’, con una prosa, rimanendo in un ambito informale: nella maggior parte dei manuali di linguaggi di programmazione a fronte di una sintassi quasi sempre specificata formalmente, la parte semantica è solitamente lasciata ad una descrizione informale se non intuitiva, a scapito della chiarezza e della precisione della descrizione. La scelta di una definizione formale della semantica coinvolge la possibilità di poter ragionare matematicamente sulla correttezza dei programmi e può avvenire seguendo tre grandi categorie di metodo: la semantica *operazionale*, la semantica *denotazionale* e la semantica *assiomatica*. Le idee che stanno alla base dei tre approcci possono essere così brevemente

sintetizzate:

- **Semantica Operazionale:** il significato attibuito ai costrutti è specificato dai passi della sua computazione all'interno della macchina astratta che lo esegue;
- **Semantica Denotazionale:** il significato attribuito ai costrutti è modellato tramite oggetti matematici (funzioni) che ne esprimono il comportamento di input/output curando l'effetto della computazione e non lo svolgimento;
- **Semantica Assiomatica:** il significato attribuito ai costrutti è specificato asserendo una serie di predicati logici che descrivono gli effetti della loro ezecuzione.

Questo elaborato porrà la sua attenzione solamente sulla Semantica operazionale, fornendo una dapprima una breve panoramica storica riguardante il suo processo di sviluppo e strutturazione seguita da un lavoro di indagine sulla varietà presente all'interno della famiglia di linguaggi della semantica operazionale nella sua declinazione più strutturata (SOS). I capitoli successivi tratteranno il tentativo di modellare le entità del formalismo al fine di mettere a fattor comune molte delle differenze evidenziate durante il capitolo precedente e l'analisi di alcuni tool che hanno cercato di dare un implementazione delle regole di specifica semantica.

1.1 Panoramica storica

L'importanza di fornire una precisa semantica ai linguaggi di programmazione è stata riconosciuta sin dagli anni '60, al momento dello sviluppo del primo linguaggio di alto livello. L'uso di semantica operazionale, una semantica che esplicitasse il comportamento della computazione di un programma in ogni passo, era già stata supportata da McCarthy nel suo *'Towards a mathematical science of computation'*, e ha trovato anni dopo pieno utilizzo nella definizione di linguaggi come Algol60, PL/I e CSP.

Nel 1981 Gordon Plotkin, con il trattato seminale *'A Structural Approach to Operational Semantics'*, ha introdotto l'uso della Semantica Operazionale Strutturale (SOS) come modalità sistematica di definizione della semantica operazionale dei linguaggi di programmazione, partendo da un insieme di regole espresse in una certa forma. All'interno della SOS l'enfasi è posta sulla nozione di *passo individuale di esecuzione* con l'obiettivo di descrivere come

questo si realizza, tenendo traccia dei cambiamenti effettuati sullo *stato*, concetto che trova alloggiati al suo interno sia la *configurazione sintattica* di un costrutto ad un determinato momento della computazione, sia elementi extra sintattici come l'*ambiente*. La specifica semantica da un *astrazione* di come un programma è eseguito su una macchina (astratta). Vengono completamente ignorati dettagli relativi a registri, indirizzi di memoria, ecc... rendendo la formalizzazione indipendente da architetture e strategie implementative.

La formalizzazione della SOS poggia sui *Labelled transition system (LTS)*, un insieme di regole che illustrano le possibili transizioni entro le quali un frammento di sintassi può incanalarsi durante la sua 'esecuzione' ed i termini coinvolti al loro interno. Le transizioni possono inoltre essere etichettate (*label*), al fine di comunicare l'azione intrapresa, e affiancate da *predicati* sui termini. Grazie alla propria intuitività e flessibilità l'utilizzo della SOS ha trovato poi considerevole applicazione all'interno della teoria dei processi concorrenti, ambito da cui hanno preso piede alcuni dei maggiori sviluppi teorici legati ad essa.

Nel corso degli anni, il formato delle regole SOS è divenuto oggetto di studio. Facendo uso delle Specifiche di Sistemi Transizionali, diversi ricercatori e studiosi hanno declinato in vario modo i formati delle regole, allargandone o restringendone la sintassi secondo necessità e dominio di studio, ponendo inoltre l'attenzione verso i meta-teoremi associati ai differenti formati.

Il primo risultato meta-teorico, datato 1985, è apparso all'interno della tesi di dottorato di Robert de Simone, in cui viene sottolineato il legame fra le restrizioni sintattiche imposte ad un TSS per conformarsi ad uno specifico oggetto di studio e le proprietà della semantica operazionale esprimibile. Il formato sintattico proposto da De Simone risulta uno dei maggiormente ristretti apparsi in letteratura; successivamente altri autori come Bloom, Bol, Groote e Fokkink solo per citarne alcuni, hanno via via sempre più allargato i limiti sintattici cercando di inglobare una più ampia varietà di termini coinvolti all'interno della specifica delle regole.

Nella prossima sezione del presente capitolo verrà presentata una panoramica sintattica dei formati di regola che negli anni sono comparsi all'interno di manuali e articoli scientifici. Durante l'elaborato verrà posta l'attenzione solamente sull'aspetto sintattico rimandando ad altre pubblicazioni l'aspetto meta-teorico.

1.2 Labelled Transition System

1.2.1 Generalità sui Labelled Transition System e utilizzo in SOS

Come brevemente accennato in precedenza, l'ossatura della SOS poggia sui *Labelled Transition System (LTS)*, consistenti in relazioni binarie fra stati etichettate con un'azione. Intuitivamente, $s \xrightarrow{a} s'$ esprime la possibilità di transizione dallo stato s allo stato s' in seguito all'esecuzione dell'azione a. Il concetto di stato è specifico del dominio su cui si pone l'attenzione ed il suo significato può variare notevolmente con esso. Alcuni LTS permettono inoltre l'utilizzo di predicati in associazione agli stati. La scrittura sP sta a significare che P vale nello stato s. Per convenienza terminologica ci riferiremo sia alle relazioni che ai predicati con il termine *transizioni* allo scopo di allargare il più possibile in quanto a generalità la definizione di LTS.

Definizione 1.2.1. (Labelled Transition System) *Un* labelled transition system *è una quadrupla* $(\mathsf{States}, \mathsf{Act}, \left\{\xrightarrow{a} | a \in \mathsf{Act}\right\}, \mathsf{Pred})$*, dove:*

- States *è un insieme di stati, rappresentati da s;*
- Act *è un insieme di azioni, rappresentate da a,b;*
- $\xrightarrow{a} \subseteq \mathsf{States} \times \mathsf{States}$ *per ogni* $a \in \mathsf{Act}$*. Per comodità useremo la notazione* $s \xrightarrow{a} s'$ *al posto di* $(s, s') \in \xrightarrow{a}$*, e* $s \stackrel{a}{\nrightarrow}$ *se* $s \xrightarrow{a} s'$ *per nessun stato s';*
- Pred *è un insieme di predicati sugli stati* $s \in \mathsf{States}$*. Scriveremo sP (s¬P) qualora lo stato s soddisfi (non soddisfi) il predicato P.*

1.2.2 Termini

Definizione 1.2.2. (Signature) *Una* signature Σ *è un insieme si simboli di funzione, disgiunto da quello delle infinite variabili* Var*, ognuno con un numero di arietà assegnato.*

La signaure fornisce un insieme di termini, particolare per il dominio, che va a popolare concretamente l'insieme generico ***States*** *Una* costante *è un simbolo di funzione di* Σ *con arietà 0.*

Definizione 1.2.3. (Termine) *L'insieme* $\Upsilon(\Sigma)$ *dei* termini aperti *di una signature* Σ*, rappresentati con t,u, è il più piccolo insieme per cui:*

- *ogni variabile* $x \in \mathsf{Var}$ *è un termine;*

- *$f(t_1, \dots, t_{ar(f)})$) è un termine, se f è un simbolo di funzione di Σ e $t_1, \dots, t_{ar(f)}$ sono termini.*

$\mathrm{T}(\Sigma)$ *denota l'insieme dei* termini chiusi *su* Σ, *i termini che non contengono variabili.*

Le constanti $a()$ sono abbreviate in a.

Una sostituzione è una mappa $\sigma : \mathsf{Var} \to \Upsilon(\Sigma)$. Una sostituzione si definisce *chiusa* se ogni variabile viene mappata su un termine chiuso in $\mathrm{T}(\Sigma)$. La sostituzione da termini a termini è definita induttivamente sulla struttura del termine di partenza. Una sostituzione esplicita è delineata dalla scrittura $t[t'/x]$ con cui si intende il rimpiazzo del termine t' al posto di ogni occorrenza della variabile x all'interno del termine t.

All'interno di articoli scientifici sul calcolo di processi, viene spesso fatta la distinzione fra termini formali e attuali. La differenza poggia sulla diversa connotazione attribuita alle variabili. All'interno del mondo formale, una variabile (e per estensione induttiva un termine) è considerata un segnaposto per un termine aperto della signature a cui corrisponderà in una sostituzione. Una regola che presenta termini formali non è mai da utilizzare direttamente ma deve essere necessariamente istanzia mediante sostituzione(formale) con un termine aperto che ne concretizzi lo scheletro.

Le variabili attuali (e per estensione i termini) sono invece utilizzate come binder all'interno di costrutti particolari e all'interno di sostituzioni chiuse esplicite, sempre come segnaposto per termini chiusi.

Per chiarire più dettagliatamente la differenza mostriamo un esempio:

$$\frac{y * [\mu x.y * /x] \xrightarrow{a} z*}{\mu x.y* \xrightarrow{a} z*}$$

I termini formali compaiono accompagnati da un $*$. Notiamo la differenza fra il termine y e x. Il termine y non è altro che un segnaposto per un termine (anche aperto) con cui andiamo ad istanziare la regola. Il termine x invece è un binder all'interno del corpo del termine con cui verrà sostituito y.

Definizione 1.2.4. (Specifica di un sistema di transizioni)*(TSS) Sia Σ una signature, e t e t' meta-variabili di $\Upsilon(\Sigma)$. Una* regola di transizione *ρ, è formalizzata $\frac{H}{\alpha}$, con H insieme di premesse del tipo $t \xrightarrow{a} t'$ e tP (o anche $t \overset{a}{\not\to}$ e$\neg tP$ nel caso in cui sia concesso esprimere premesse negative) e α conclusione della forma $t \xrightarrow{a} t'$. Una regola di transizione si definisce* chiusa *qualora non contenga variabili.*

Le regole di transizione che presentano termini aperti all'interno dell'insieme delle premesse o delle conclusioni rappresentano schemi di regola e non regole vere e proprie. Necessitano infatti di una istanziazione ottenibile mediante l'utilizzo di una sostituzione che, a partire dalla regola comprendente termini aperti ne restituisca una corrispondente con solo termini chiusi. L'utilizzo di regole con termini aperti è necessario nel caso in cui si spazi all'interno di domini infiniti, ad esempio i numeri naturali. Una regola del tipo

$$\frac{}{m_0 + m_1 \longrightarrow m_2} \text{ se nell'aritmetica dei naturali } m_0+m_1 = m_2$$

che schematizza le transizioni dei costrutti sintattici che rapprensentano somme di naturali, fa utilizzo delle metavariabili m_0,m_1 e m_2 per identificare tutte le possibili coppie di numeri naturali e le relative somme:

$$\frac{}{0 + 0 \longrightarrow 0}$$

$$\frac{}{0 + 1 \longrightarrow 1}$$

$$\frac{}{1 + 1 \longrightarrow 2}$$

$$\vdots$$

Una transizione si dice *provabile* da un TSS T se è possibile costruire un albero, ramificato verso l'alto senza ramificazioni infinite, con le seguenti caratteristiche:

- La radice dell'albero è α;
- i nodi sono del tipo $t \xrightarrow{a} t'$ e tP (o anche $t \overset{a}{\nrightarrow}$ e$\neg tP$);
- se K è l'insieme dei nodi immediatamente sopra un nodo con etichetta β, o $K = \emptyset$ e $\beta \in H$ o $\frac{K}{\beta}$ è un'istanza di una regola di transizione in T.

1.3 Varietà Sintattica

La sezione corrente si occuperà di presentare in sequenza tutta una serie di differenziazioni sintattiche, più o meno sostanziali, che negli anni hanno visto la luce all'interno dei formati delle regole dei linguaggi della famiglia SOS.

1.3.1 Signature multi-sorta

Un formalismo SOS può differenziarsi, in base alla possibilità di ammettere diverse sorte per gli elementi dell'insieme degli stati. Basandosi sul numero di sorte permesse nella signature, un formalismo SOS può essere classificato in tre categorie differenti:

- **Multi-Sorta:** in cui non vi è restrizione sulle sorte permesse nella costruzione dei termini;
- **N-Sorta:** in cui solo un numero definito di sorte partecipa alla signature;
- **Sorta Singola:** in cui l'insieme dei termini è costruito a partire da una sola sorta ammessa all'interno della signature. Si tratta della categoria maggiormente frequente in letteratura in cui la singola sorta degli stati si accompagna solitamente ad un insieme di label costanti.

Esempio eloquente di signature a N-sorte (in questo caso specifico a 2-sorte) è il formato di regole plasmato da Mousavi, Reniers e Groote in *'Notion of bisimulation and congruence formats for SOS with data'* in cui processi e dati appartengono ad insiemi diversi ed esistono signature separate per processi e dati. Il formato delle regole segue il seguente schema:

$$\frac{\left\{(t_i, u_i) \xrightarrow{l_i}_{r_i} (y_i, u'_i) | i \in I\right\}}{(f(x_0, \ldots, x_{n-1}), u) \xrightarrow{l}_{r} (t', u')}$$

dove I è un insieme di indici, $\rightarrow_r$ un relazione, $l \in L$ insieme di label, e sono presenti differenti sorte per le variabili e per le signature di processi e dati, rispettivamente V_p, Σ_p e V_d, Σ_d, con $f \in \Sigma_p$, $x_0, \ldots, x_{n-1}$ e $y_i \in V_p$, t_i e t' $\in T(\Sigma_p)$ mentre $u, u', u_i, u'_i \in T(\Sigma_d)$.

Fokkink e Verhoef nel loro *'A Conservative Look at Operational Semantics with Variable Binding'* ammettono un insieme V di variabili e una formazione dei termini della signature di tipo multi-sorta. Le funzioni costruttrici dei termini di Σ sono operano secondo lo schema:

$$f : S_1 \text{ x } \ldots \text{ x } S_n \rightarrow S \text{ dove } S_1, \ldots, S_n \text{ e } S \text{ sono sorte.}$$

dove $S_1, \ldots, S_n$ e S sono sorte.

In *'A format for semantic equivalence comparison'* V. Galpin espone la definizione di un nuovo formato di regole (che qualifica con l'aggettivo 'esteso') che esibisce una singolare caratteristica nella signature. La particolare

connotazione è dovuta alla presenza di una troncatura netta fra l'insieme delle sorte ammesse nella signature e una sorta esclusiva per gli stati (P) non vuota. I termini aperti e chiusi generabili attraverso le funzioni che popolano la signature non faranno parte degli stati all'interno delle transizioni, in quanto l'insieme P conterrà elementi disgiunti da essi. In quest'ottica la sorta P è da considerarsi necessaria al fine di fornire un insieme di specifiche transizionali, a differenze delle restanti sorte su cui non è fatta alcuna ipotesi e nessun vincolo imposto. Lo scheletro delle regole viene formalizzato con:

$$\frac{\{p_i \xrightarrow{\lambda_i} p'_i | i \in I\}}{p \xrightarrow{\lambda} p'}$$

dove I è un insieme di indici, p, p', p_i, p'_i elementi dell'insieme P e λ_i e λ termini aperti delle restanti sorte.

1.3.2 Labels

La maggior parte dei framework SOS impiegati nella semantica operazionale di linguaggi di programmazione e calcolo di processi, riserva una particolare sorta per i termini che possono comparire come label, spesso prevedendo unicamente l'impiego di costanti per ricoprire il ruolo o omettendo l'etichetta della transizione nel caso in cui non abbia importanza specificarla(come succede nella maggior parte delle specifiche semantiche di linguaggi di programmazione). In alternativa possono essere contemplati anche generici termini, purchè chiusi, impiegati come label. Il punto rimane comunque impedire la valutazione dei termini che compaiono come label attraverso l'uso comune di variabili. Passiamo in rassegna una serie di formalismi SOS, al cui interno vengono rilassati i vincoli appena esposti.

Alcuni linguaggi SOS, come quello formalizzato in *'A Conservative Look to Operational Semantics with Variable Binding'* di Fokkink e Verhoef, permettono l'uso di termini aperti come label. Il formato delle regole viene allargato fino a comprendere transizioni del tipo $t \xrightarrow{x} t'$. Nell'intento di formalizzare la semantica del πI-calcolo (introdotto da D. Sangiorgi nel 1995) viene fatto uso di regole nel suddetto schema:

$$\text{PRE} \quad x(y).v^* \xrightarrow{x(y)} v^*$$

$$\text{SUM} \quad \frac{v^* \xrightarrow{x(y)} v'^*}{v^* \ + \ w^* \xrightarrow{x(y)} v'^*}$$

$$\text{PAR} \quad \frac{v^* \stackrel{x(y)}{\rightarrow} v'^* \qquad \neg F_y(w^*)}{v^*|w^* \stackrel{x(y)}{\rightarrow} v'^*|w^*}$$

La signature del πI-calcolo prevede due sorte, *Porte* e *Processi*. La regola PRE formalizza il comportamento di un processo $(x(y).v^*)$ che dopo aver comunicato il nome della porta y attraverso la porta x prosegue come il processo $v*$. Le regole SUM e PAR esprimono il medesimo concetto nel caso di composizione alternata o parallela di due processi v^* e w^*. Il termine aperto $x(y)$ compare come label e a riguardo è opportuno, per rigore di precisione, notare come x sia variabile libera, mentre l'occorrenza di y rappresenti un binder del corpo del processo con cui verrà istanziata la regola al momento della sostituzione formale del termine v^*.

Il calcolo di processi di ordine superiore è un campo della teoria dei sistemi concorrenti in cui è frequente l'utilizzo di termini aperti come label. I linguaggi di tale ambito, permettono infatti l'utilizzo di termini provenienti dalla sorta degli stati per comporre le label. A differenza del calcolo di processi del prim ordine, in cui l'intero calcolo è costruito attorno alla nozione di passaggio di nomi di porte di comunicazione fra agenti concorrenti, e abbiamo la netta separazione fra l'insieme *States* (in questo caso processi) e *Act* (in questo caso porte), in calcoli di processi di ordine superiore i processi stessi entrano a far parte dell'insieme *Act*.

Riportiamo alcuni esempi di regole esposte da B. Thomsen in *'A theory of higher order communicating systems'* in cui viene presentata un estensione del linguaggio CCS. Nel linguaggio in questione, denominato CHOCS, l'insieme *Act* dei termini che possono comparire in veste di label viene formalizzato come $Act = Names\ x\ \{?,!\}\ x\ Pr \cup \{\tau\}$, dove $Names$ identifica l'insieme dei nomi attribuibili alle porte mentre Pr quello dei processi.

$$\textbf{Restrizione:} \quad \frac{p \stackrel{a?p'}{\rightarrow} p''}{p/b \stackrel{a?p'}{\rightarrow} p''/b} \quad a \neq b$$

$$\textbf{Rinomina:} \quad \frac{p \stackrel{S(a)?p'}{\rightarrow} p''}{p[S] \stackrel{S(a)?p'}{\rightarrow} p''[S]} \quad a \neq b$$

Entrambe le regole presentano il processo p' a livello di label, mentre p e p'' occupano il ruolo di stati. La dicitura $a?p'$ denota la possibilità di ricevere sulla porta a il processo p'. p/b vieta al processo p la comunicazione per mezzo della porta b, mentre $p[S]$ palesa la rinomina delle porte del processo

p secondo la mappa S.

Mousavi in *'SOS for higher order processes'* svolge una carrellata dei formati delle regole SOS che negli anni si sono susseguiti nel formalizzare la semantica operazionale dei calcoli di processi di ordine superiore. La successione esplora nel dettaglio le restrizioni sulla natura dei termini disponibili ad occupare il ruolo di label, man mano rilassate durante l'evoluzione morfologica delle regole. In particolare Mousavi documenta come fondamentale, il passaggio fra il formato di regole *Tyft/tyxt* e la sua estensione *promoted Tyft/tyxt* proposta da Bernstein, in cui si è aperto alla possibilità di comprende generici termini della signature come label. Le regole in tal formato hanno la forma:

$$\frac{t_i \stackrel{t'_i}{\rightarrow} y_i | i \in I}{f(\overrightarrow{x_j}) \stackrel{g(\overrightarrow{z_k})}{\rightarrow} t}, \quad \frac{t_i \stackrel{t'_i}{\rightarrow} y_i | i \in I}{f(\overrightarrow{x_j}) \stackrel{z}{\rightarrow} t}$$

oppure

$$\frac{t_i \stackrel{t'_i}{\rightarrow} y_i | i \in I}{x \stackrel{g(\overrightarrow{z_k})}{\rightarrow} t}, \quad \frac{t_i \stackrel{t'_i}{\rightarrow} y_i | i \in I}{x \stackrel{z}{\rightarrow} t}$$

Oltre all'impiego di termini generici come label, all'interno di alcuni testi trovano spazio anche transizioni che prevedono l'uso di liste di termini aperti, secondo lo schema $t \stackrel{x,y}{\rightarrow} t'$. In 'A conservative look with variable binding'la specifica transizionale della semantica per real-time BPA presenta regole che alloggiano liste di termini come label. La lista contiene sia costanti a appartenenti all'insieme *Act*, sia generici termini t della signature. Riportiamo un esemplare proveniente da tale specifica:

$$\frac{p^* \stackrel{a,t}{\rightarrow} p'^*}{p^* \;+\; q^* \stackrel{a,t}{\rightarrow} p'^*}$$

Anche Mousavi nella specifica dei suoi formati *Promoted PANTH* e *Higher Order PANTH* ammette l'utilizzo di liste di termini (generici) all'interno delle label delle transizioni.

$$\frac{\{P(L_i t_i) \text{ or } t_i \stackrel{L_i}{\rightarrow}_{r_i} y_i | i \in I\} \quad \{\neg P(L_i t_i) t_i \stackrel{L_i}{\nrightarrow}_{r_i} y_i | i \in I\}}{P(L) f(\overrightarrow{x_i}) \text{ or } f(\overrightarrow{x_i}) \stackrel{L}{\rightarrow}_r t'}$$

Nella regola appena sopra le label L, L_i e L_j sono liste di termini generici della signature, del tipo $\overrightarrow{t}$.

Particolare trattamento delle label è quello riservato dalle specifiche della SOS Modulare di P. Mosses. La SOS Modulare prevede infatti una label con struttura, all'interno della quale incorporare componenti diverse, diverse per tipologie. Le label sono formulate secondo lo schema:

$$\{\rho,\ \sigma = \sigma_0,\ \sigma' = \sigma_0[x = t]\}$$

La SOS Modulare è trattata in dettaglio in uno specifico paragrafo del presente trattato. Label strutturate sono presenti anche all'interno delle specifiche della Enhanced SOS di P. Degano e C. Priami.

1.3.3 Premesse negative

In alcuni linguaggi della famiglia SOS è concesso esprimere all'interno delle transizioni contenute come premesse o conclusioni delle regole comprese all'interno di specifiche un sistema transizionale. Le premesse negative compaiono secondo la scrittura $t \nrightarrow$, $t \nrightarrow t'$ o $\neq Pt$ e stanno a significare rispettivamente l'impossibilità a transire in un qualunque stato(risulta una forma più compatta rispetto alla scrittura $(\forall t'(t \nrightarrow t')))$, l'impossibilità a transire verso uno stato specifico e il mancato soddisfacimento di una predicato in un determinato stato. Le premesse negative costituiscono un fattore di complicanza notevole all'interno di un linguaggio SOS, in quanto la loro presenza non rende chiaro immediatamente come possa essere costruita una prova di una formula negativa.

Il primo esempio di framework SOS con regole negative è apparso in *'Merge and termination in process algebra'* di Baeten e van Glabbeek nel tentativo di dare una semantica all'operatore di priorità.

$$\theta(_): \frac{x \xrightarrow{a} x' \qquad \forall_{b>a} x \stackrel{b}{\nrightarrow}}{\theta(x) \xrightarrow{a} \theta(t')}$$

La regola formalizza la possibilità di eseguire una transizione con la label a unicamente nel caso in cui sia verificata l'impossibilità di eseguirne una con ognuna delle label che, fornito un ordinamento, risulti $< a$. L'operatore di priorità sarebbe difficilmente esprimibile senza l'ausilio di transizioni negative, violando una delle caratteristiche fondamentali del formalismo SOS, la semplicità, motivo portante per cui si è mostrata una tendenza a rilassare i vincoli sintattici dei linguaggi SOS al fine di evitare inutili complicazioni.

Diversi studiosi hanno affrontato le problematiche connesse all'ammissione di transizioni negative. Tra questi si è distinto J. F. Groote, di passiamo brevemente in rassegna i risultati raggiunti. J. F. Groote in 'Transition system specifications with negative premises' esamina approfonditamente i

miglioramenti e le difficoltà legate all'introuduzione di regole con premesse negative all'interno di sistemi di specifichee transizionali. L'indagine parte dall'ammissione della necessità, nell'affrontare determinati problemi, di introdurre la possibilità di poter condizionare lo svolgimento di una transizione, in base all'assenza di disponibilità di altre. Ad esempio, esprimere un operatore di individuazione della situazione di deadlock di un processo (p), $D(p)$, può essere espresso in modo naturale nella seguente maniera:

$$\frac{p \overset{\forall a \in Act}{\nrightarrow}}{D(p)}$$

La definizione di un operatore di deadlock è anche fondmentale per esprimere sequenzialità di due processi:

$$\frac{D(p)}{p.q \overset{a}{\rightarrow} q}$$

in cui verificata l'impossibilità alla prosecuzione di p, q può cominciare la sua esecuzione. Groote sottolinea come due problemi principali compaiano qualora si scelga di allargare il formato di regola fino a comprendere transizioni negative(in questo contesto il formato considerato è il *tyft/tyxt*, modificato con l'addizione di premesse negative in *ntyft/ntyxt*):

- La possibilità di avere un insieme inconsistente di regole, occorre quando ho una transizione possibile solo a condizione che sia verificata la sua negata. In questo caso la semantica operazionale non è definita;

- Non è immediato verificare che un insieme anche inconsistente di regole che ammettano premesse negative definisca una semantica operazionale. La nozione consueta di provabilità, in cui le regole di una specifica transizionale (TSS) vengono utilizzate come regole di inferenza, non è più soddisfacente.

Groote tenta di fornire una soluzione al primo problema(anche al secondo ma la sua stesura varca gli obiettivi di questo testo), sviluppando una condizione che garantisca che l'insieme delle transizioni possibili per una specifica TSS sia non vuoto. Una relazione di transizione è costruita passo-passo, ed ogni volta che una transizione non risulta possibile in una relazione, deve essere garantita l'impossibilità di derivare il contrario da questa assunzione. Per assicurare ciò viene costruito con grafo che raffigura le dipendenze tra una transizione e l'altra della specifica TSS, avente archi etichettati con p fra la premessa positiva e la conclusione di una sostituzione chiusa di una regola di transizione del sistema e con n fra una premessa negativa e la sua conclusione.

Il verificarsi di una ciclo all'interno di tale grafo evidenzia la dipendenza fra due transizioni, in senso sia negativo che positivo, e decreta l'esistenza di una contraddizione all'interno della TSS in questione. A titolo di esempio consideriamo un TSS, con una signature che prevede due costanti a e δ e due funzioni unarie f e g, con l'insieme delle seguenti regole:

$$\frac{x \xrightarrow{a} y \quad y \xrightarrow{a} z}{f(x) \to \delta}$$

$$\frac{x \stackrel{a}{\nrightarrow}}{g(x) \to \delta}$$

$$a \to g(f(a))$$

Utilizzando la sostituzione chiusa σ, che mappa x in a, otterremmo il grafo delle dipendenze in Figura 1.1, facendo uso delle regole esposte poco sopra.

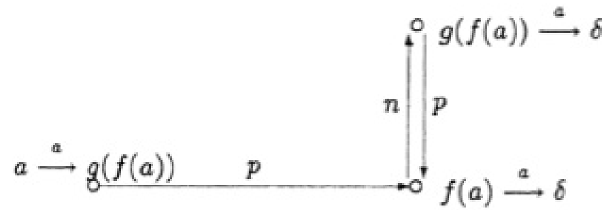

Figura 1.1: Grafo delle dipendenze con premesse negative

1.3.4 Premesse infinite

Alcuni linguaggi della famiglia SOS non pongono limitazioni alla cardinalità dell'insieme delle label. La possibilità di prevedere un numero infinito di elementi facenti parte dell'insieme *Act* costituisce uno dei due presupposti per la formazione di regole che presentino un numero infinito di premesse. L'altro aspetto riguarda l'eventualità di ammettere quantificatori logico universale in ambito di formazione delle regole.

L'operatore di priorità definito all'interno del paragrafo precedente, rappresenta un'esempio di regola con premesse infinite. Infatti, nel caso in cui l'insieme Act contenga un'infinità di azioni, la seconda premessa risulta infinita.

Un'altro esempio lo possiamo trovare in *'The algebra of timed processes ATP: theory and application'* di Nicollin e Sifakis in relazione all'operatore di incapsulamento:

$$\vartheta(_): \frac{\forall_{a \in A \setminus H} x \stackrel{a}{\nrightarrow}}{\vartheta_H(x) \stackrel{\chi}{\rightarrow} \vartheta(\delta)}$$

Il caso in cui l'insieme A sia composto da infiniti elementi costituisce presupposto per la presenza di premesse infinite.

1.3.5 Predicati

L'introduzione di predicati qualificati come transizioni nasce come stratagemma per semplificare lo studio di proprietà insistenti sui termini delle regole, come la terminazione.

Riportiamo di seguito un breve scampolo a scopo di chiarimento:

$$\frac{}{1 \downarrow} \qquad \frac{x \downarrow}{x + y \downarrow} \qquad \frac{y}{x + y \downarrow}$$

La regola esprime l'implicazione fra la terminazione di un processo e la terminazione della composizione alternativa in cui compare come uno dei termini.

1.3.6 Appoggio ad altri framework

Talvolta viene fatto utilizzo di costrutti esterni alla sintassi dei LTS per esprimere le regole SOS.

Si ricorre a framework complementari nell'espressione di determinati concetti semantici per ragioni di immediatezza e facilità, oltreche per l'avversione di determinati domini ad essere trattati con il formalismo transizionale.

Vediamo una serie di esempi di framework complementari:

- **Specifiche equazionali:**
  L'utilizzo di specifiche equazionali all'interno di regole di semantica operazionale, è stato introdotto da R. Milner nella specifica della semantica operazionale del π-calcolo, e trova utilizzo frequente nei casi in cui si debba focalizzare l'attenzione su alcune proprietà inerenti la congruenza strutturale di determinati termini o qualora si debbano definire termini in funzione di altri. La congruenza strutturale dei termini di una signature è la minima relazione che soddisfa i vincoli:

- $\forall_{p \in T(\Sigma)}\ p \equiv p$
- $\forall_{p,q \in T(\Sigma)}\ p \equiv q \ \Rightarrow\ q \equiv p$
- $\forall_{p,q,r \in T(\Sigma)}\ (p \equiv q\ \wedge\ pq \equiv r)\ \Rightarrow\ q \equiv r$
- $\forall_{f \in \Sigma} \forall_{p_i,q_i \in T(\Sigma)}\ (\forall_{0 \leq i \leq ar(f)}\ p_i \equiv q_i)\ \Rightarrow\ f(p_0, \ldots, p_{ar(f)-1}) \equiv f(q_0, \ldots, q_{ar(f)-1})$
- $\forall_{\sigma : V \to T(\Sigma)} \forall_{t,t' \in \Upsilon(\Sigma)}\ (t, t') \in \equiv\ \Rightarrow\ \sigma(t) \equiv \sigma(t')$

La combinazione di specifiche equazionali e regole SOS in stile tradizionale, semplifica la specifica SOS e le fornisce un aspetto più compatto. A titolo di esempio, la commutatività della composizione parallela di due termini, in *'Congruence for structural congruences'* di Mousavi e Reniers, viene espressa nella seguente forma:

$$x||y \equiv y||x$$

Le specifiche equazionali possono inoltre essere inserite all'interno di premesse di regole di transizione, combinate a transizioni canoniche. Alcuni linguaggi della famiglia SOS ammettono regole scritte secondo questa modalità, presentando schemi di regola nella forma seguente:

$$\frac{x \equiv y \qquad y \xrightarrow{l} y' y \equiv x}{x \xrightarrow{l} y} \quad \text{con } l \in Act$$

- **Funzioni Semantiche:**All'interno delle specifiche di semantica operazionale dei linguaggi programmativi, troviamo ampio utilizzo di un metodo agevole per mappare gli elementi appartenenti alle categorie sintattiche elementari in simboli semantici che fanno parte di domini differenti.

  La connessione fra regno sintattico e semantico avviene per mezzo di funzioni semantiche che rendono semplice e intuitivo il passaggio. Esteso sfruttamento di questo costrutto complementare viene fatto al momento di fornire una semantica alle operazioni aritmetiche e alle stringhe sintattiche che simboleggiano naturali e booleani, definendole solitamente per induzione strutturale. Fornita un espressione espressione ed uno stato (in questo caso considerabile come una semplice mappa fra variabili e valori) la funzione semantica è una funzione di ordine superiore che prende in input l'espressione e ci restituisce una funzione **Stato** $\to$ **Valore**. Da un'ottica differente possiamo invece considerarla una funzione di ordine superiore che presa in input uno stato restituisce

una funzione **Espressione** $\rightarrow$ **Valore**, notando quella che può essere chiamata una sua parametrizzazione simmetrica.

H. e F. Nielson in *'Semantics with applications'* descrivono la semantica delle stringhe sintattiche che rappresentano naturali nella seguente maniera(le metavariabili a_1 e a_2 spaziano all'interno della categoria sintattica delle espressioni aritmetiche):

- $A[n] = N[n]$,dove n a sinistra è sintassi mentre a destra numero naturale;
- $A[x]s = s\ x$,dove la funzione semantica A, mappa la variabile x al suo valore corrente, avvalendosi dell'ispezione dello stato;
- $A[a_1 + a_2]s$ =$A[a_1]s$+$A[a_2]s$,dove l'operatore di addizione + sulla sinistra è semplice sintassi mentre sulla destra aritmetica;
- $A[a_1 * a_2]s$ =$A[a_1]s$*$A[a_2]s$,dove l'operatore di moltiplcazione $*$ sulla sinistra è semplice sintassi mentre sulla destra aritmetica;
- $A[a_1 - a_2]s$ =$A[a_1]s$-$A[a_2]s$,dove l'operatore di sottrazione $-$ sulla sinistra è semplice sintassi mentre sulla destra aritmetica;

Il meccanismo delle funzioni semantiche può essere descritto avvalendosi di una comoda sintassi per casi, come avviene ad esempio in fase di definizione della funzione B che mappa semanticamente le espressioni booleane. Comprendendo il confronto fra espressioni aritmetiche, la funzione B non può evitare di chiamare all'interno dei propri casi la funzione semantica di mappature delle espressioni aritmetiche, per avere termini finali di confronto. Nell'estratto seguente $b1$, $b2$ spaziano all'interno della categoria delle espressioni booleane.

- $B[true] = \mathbf{tt}$,dove $true$ a sinistra è sintassi mentre a destra **tt** è valore di verità secondo l'algebra booleana;
- $B[false] = \mathbf{ff}$,dove $false$ a sinistra è sintassi mentre a destra **ff** è valore di falsità secondo l'algebra booleana;
- $B[a_1 = a_2]s = \left\{ \begin{array}{cc} \mathbf{tt} & if\ A[a_1]s = A[a_2 s] \\ \mathbf{ff} & if\ A[a_1]s \neq A[a_2 s] \end{array} \right\}$
- $B[a_1 \leq a_2]s = \left\{ \begin{array}{cc} \mathbf{tt} & if\ A[a_1]s \leq A[a_2 s] \\ \mathbf{ff} & if\ A[a_1]s \gg A[a_2 s] \end{array} \right\}$
- $B[\neg\ b]s = \left\{ \begin{array}{cc} \mathbf{tt} & if\ B[b]s = \mathbf{ff} \\ \mathbf{ff} & if\ B[b]s = \mathbf{tt} \end{array} \right\}$

– $B[b_1 \wedge b_2]s = \left\{ \begin{array}{ll} \mathbf{tt} & if\ B[b_1]s = \mathbf{tt}\ and\ B[b_2]s = \mathbf{tt} \\ \mathbf{ff} & if\ B[b_1]s = \mathbf{ff}\ or\ B[b_2]s = \mathbf{ff} \end{array} \right\}$

- **Semantica Algebrica:** La Semantica Algebrica è un formalismo le cui fondamenta poggiano sull'algebra astratta che affianca i LTS fornendo una specifica algebrica di dati e costrutti linguistici. L'approccio di base consiste nello specificare nomi e operazioni per oggetti di una certa sorta, usando assiomi algebrici per descrivere le loro proprietà caratteristiche. L'implementazione di dati e operazioni non fa parte delle specifiche della semantica algebrica.

  La signature di una semantica algebrica è una coppia $< Sorte, Operazioni >$, dove le *Sorte* sono i tipi necessari alla definizione delle operazioni. Poniamo ora l'attenzione su un esempio di modulo di semantica algebrica che fornisce semantica per i booleani. Il modulo è sudiviso in tre parti: una parte di dichiarazioni delle sorte, una di firme di operazioni, una di assiomi algebrici che specificano il funzionamento delle operazioni.

  module Booleans
  sorts Boolean
  operations
  true : Boolean
  false : Boolean
  errorBoolean : Boolean
  not (_) : Boolean → Boolean
  and (_ , _) : Boolean, Boolean → Boolean
  or (_ , _) : Boolean, Boolean → Boolean
  implies (_ , _) : Boolean, Boolean → Boolean
  eq? (_ , _) : Boolean, Boolean → Boolean
  xor (_ , _) : Boolean, Boolean → Boolean
  variables
  b, b1, b2 : Boolean
  equations
  [B1] and (true, b) = b
  [B2] and (false, true) = false
  [B3] and (false, false) = false
  [B4] not (true) = false
  [B5] not (false) = true
  [B6] or (b1, b2) = not (and (not (b1), not (b2)))
  [B7] implies (b1, b2) = or (not (b1), b2)
  [B8] xor (b1, b2) = and (or (b1, b2), not (and (b1, b2)))

[B9] eq? (b1, b2) = not (xor (b1, b2))
end Booleans

1.3.7 Condizioni di regola

Alcuni framework SOS si avvalgono di condizioni a livello di regola, per fornire semantica a quei costrutti sintattici in cui la conoscenza della conformazione di parte di essi, è essenziale per decretare il passo successivo di transizione. Classici esempi dell'utilizzo delle condizioni di regola sono i costrutti `if-then-else` o `while` dei linguaggi di programmazione.

Mostriamo di seguito un estratto da *'Semantics with applications'* di H. e F. Nielson a titolo di esempio, in cui B è la funzione semantica che mappa i booleani e s è lo stato:

$$< \text{if (b) then c1; else c2;}, s > \Rightarrow < \text{c1}, s > \textit{ where } B[b]s = \textit{true}$$

Da notare come Plotkin in *'A structural approach to operational semantics'* scelga di descrivere la semantica di `if-then-else` senza fare uso di funzioni semantiche o condizioni di regola, rimanendo completamente all'interno del dominio delle regole transizionali, sottolineando come, con questa connotazione, la condizione di regola non sia altro che una forma di zucchero sintattico, utile ad evitare il miscuglio di domini differenti a livello delle premesse delle regole con un guadagno in chiarezza e semplicità.

$$\frac{< b, \sigma > \rightarrow^* < tt, \sigma >}{< \text{if}(b)\text{then } c; \text{else } c'; , \sigma > \rightarrow < c, \sigma >}$$

L'utilizzo delle condizioni di regola è inoltre necessario all'interno di quei framework di semantica operazionale che stabiliscono un ordine all'interno dell'insieme di regole di transizione di una TSS. Una regola può essere applicata solo qualora non fosse applicabile alcuna regola che la preceda nell'ordinamento. Stabilito un ordinamento << fra le regole che compongono una specifica transizionale, la semantica di un operatore sequenziale verrebbe formalizzata come segue:

$$(r_x)\frac{x \stackrel{l}{\rightarrow} x'}{x; y \stackrel{l}{\rightarrow} x'; y}$$

$$(r_y)\frac{y \xrightarrow{l} y'}{x; y \xrightarrow{l} x; y'} \quad \text{if } r_y \text{ non può essere applicata.}$$

E' evidente come le due regole esposte sopra mantengano inalterata la consueta semantica per l'operatore sequenziale: il secondo argomento della composizione sequenziale può cominciare l'esecuzione a patto che il primo non possa compiere ulteriori transizioni. La condizione posta nella seconda regola è essenziale per rispettare l'ordine stabilito per le regole, in questo caso $(r_x) >> (r_y)$.

Diversa funzione della condizione di regola è quella di De Simone, utilizzata per raggruppare regole che rispettano la condizione espressa da un predicato sulle label, in un'unica scrittura:

$$\frac{\left\{x_i \xrightarrow{l_i} y_i | i \in I\right\}}{f(\vec{x}_{ar(f)}) \xrightarrow{l} t} \; [Cond(\vec{l_i}, l)]$$

1.4 SOS nei linguaggi di programmazione

Nel contesto dei linguaggi di programmazione i LTS assumono connotazioni particolari e vengono declinati secondo la necessità di utilizzo nel particolare contesto.

1.4.1 Sistemi di transizione terminali

Nel fornire semantica ai linguaggi di programmazione ci allontianiamo dal modello precedentemente introdotto di LTS e di TTS per fornirne uno più adatto alle particolari sembianze della specifica. Legando l'evoluzione di un frammento sintattico rappresentante un costrutto di programmazione all'interno di un sistema transizionale con i concetti di calcolabilità e terminazione nasce la necessità di poter discriminare in maniera discreta fra stadi in cui la transizione (e quindi la computazione) si trova in una fase in cui può ancora evolvere e altri in cui si deve considerare conclusa.

La signature viene sostituita con l'insieme di configurazioni(Γ), al cui interno vengono alloggiati sia i termini sintattici di cui si va a definire la semantica, sia lo stato, ente che raccoglie tutte le variazioni dei componenti semantici di cui si tiene traccia durante le transizioni. L'insieme delle azioni è solitamente istanziato con un singoletto, privo di etichetta. Al fine di semplificare la specifica di un linguaggio viene aggiunto un insieme $T \subseteq \Gamma$ contenente quelle configurazioni per cui non rimangono ulteriori evoluzioni o

passi di esecuzione, chiamate *terminali*.

Viene fatto un utilizzo più libero dei predicati, inserendoli indifferentemente in forme if o $where$ a livello di condizione di regola, oppure direttamente all'interno dell'insieme delle premesse. Si tratta frequentemente di predicati di uguaglianza, che fanno uso di funzioni semantiche per limitare l'applicabilità di una regola a situazioni che verifichino il vincolo. Le transizioni esprimono il passo di esecuzione di un 'pezzo' di sintassi(solitamente astratta) S a partire dallo stato s e si presentano nella forma:

- $< S, s > \Rightarrow < S', s' >$: nel caso in cui la computazione non conduca ad una configurazione terminale e rimangano ulteriori passi di esecuzione.
- $< S, s > \Rightarrow < s' >$: nel caso in cui il frammento sintattico abbia ultimato la propria evoluzione e l'unica informazione che rimane da tracciare sia lo stato.

L'esecuzione di un frammento sintattico S a partire dallo stato s è rappresentata da una sequenza di derivazioni:

- $finita$: $\gamma_0, \gamma_1, \ldots, \gamma_k$ dove γ_0 è $< S, s >$, per ogni i esiste una istanza di una regola di transizione per cui $\gamma_i \Rightarrow \gamma_{i+1}$ e γ_k è una configurazione terminale o una configurazione non utilizzabile come sorgente di alcuna istanza delle regole di transizione;
- $infinita$: $\gamma_0, \gamma_1, \ldots$ dove γ_0 è $< S, s >$, per ogni i esiste una istanza di una regola di transizione per cui $\gamma_i \Rightarrow \gamma_{i+1}$ con $i \geq 0$.

Definito cosa si indende per *sequenza di derivazione* possiamo pensare di costruire un funzione semantica (parziale) generale S_{SOS}, che parametrizzata secondo una determinata configurazione, agisca come una mappa da stato di partenza a stato di arrivo, dove il primo è lo stato associato alla configurazione iniziale fornita come parametro, mentre il secondo è lo stato in cui termina una sequenza di derivazione terminante. La funzione può essere così formalizzata:

$$S_{SOS} : Configurazione \ \rightarrow \ (Stato \rightarrow Stato)$$

e definita:

$$S_{SOS}[\gamma]s = \left\{ \begin{array}{rl} s' & if \ < S, s > \Rightarrow^* s' \\ indefinito & altrimenti \end{array} \right\}$$

dove $\Rightarrow^*$ è la chiusura transitiva e riflessiva di $\Rightarrow$, cioè la più piccola relazione che contiene $\Rightarrow$ e gode delle proprietà riflessiva e transitiva. Viene naturale a questo punto chiedersi quando due configurazioni(quindi costrutti di programmazione perchè di tali stiamo parlando) si possano definire semanticamente equivalenti. Notiamo come una situazione di equivalenza semantica di due costrutti di programmazione S_1 e S_2 sia verificata nel caso in cui la computazione porti per entrambi nel medesimo stato finale s' oppure risulti per entambi indefinita. Nel primo caso non è necessaria l'uguaglianza di traccia delle due configurazioni, ma è sufficente che esistano due costanti k_1 e k_2 per cui $< S_1, s > \overset{k_1}{\Rightarrow} s'$ e $< S_2, s > \overset{k_1}{\Rightarrow} s'$. Formalizzando la nozione otterremo:

$$S_{SOS}[S_1]s \equiv S_{SOS}[S_2]s \left\{ \begin{array}{r} \exists_{k_1,k_2,s'}(< S_1, s > \overset{k_1}{\Rightarrow} s' \wedge < S_2, s > \overset{k_2}{\Rightarrow} s') \\ \bigvee \\ S_{SOS}[S_1]s = indefinito \wedge S_{SOS}[S_2]s = indefinito \end{array} \right\}$$

1.4.2 Ambienti

Estendendo i costrutti dei linguaggi di programmazione con blocchi contenenti dichiarazioni di variabili, procedure e assegnamenti ci troviamo di fronte alla necessità di introdurre concetti semantici importanti come quelli di ambiente e store. Prendendo in considerazione un semplice linguaggio imperativo dotato di espressioni aritmetiche, un costrutto `if-then-else` e un ciclo indefinito `while` come quello specificato in '*Semantics with application*' di H. e F. Nielson e arricchiamolo inizialmente di un blocco per la definizione di variabili:

$$\texttt{begin } D_v \ S \texttt{ end}$$

$$D_v \texttt{ ::= var } x\texttt{=}a\texttt{;}D_v \texttt{ | } \epsilon$$

Dove a spazia all'interno dell'insieme N dei naturali, S all'interno della categoria sintattica degli statement e ϵ rappresenta la dichiarazione vuota. L'idea è che variabili dichiarate fra i due estremi `begin` e `end` di un blocco siano locali ad esso e non visibili esternamente. Indicando con $\rightarrow_D$ una speciale transizione che esplica il cambiamento dello stato in seguito a dichiarazioni di variabili, con DV(D_v) l'insieme delle variabili dichiarate in D_v e con $s'[X \mapsto s]$ lo stato identico ad s' ad eccezione delle variabili appartenenti ad X, mappate come in s, forniamo una semantica ai nuovi costrutti:

$$\frac{< D_v, s > \rightarrow_D s', < S, s' > \rightarrow s''}{< \texttt{begin } D_v \ S \texttt{ end}, s > \rightarrow s''[DV(D_v) \mapsto s]}$$

$$\frac{< D_v, s[x \mapsto A[a]s]] > \to_D s'}{< \texttt{var}\ x := a;\ D_v, s > \to s'}$$

$$< \epsilon, s > \to s$$

Estendiamo ulteriormente il nostro linguaggio con la dichiarazione di procedure. Modificando la grammatica inseriamo

$$D_P \ ::= \ \texttt{proc}\ p\ \texttt{is}\ S; D_P | \epsilon$$

p è una meta-variabile che spazia all'interno dell'insieme **Pname** dei nomi di procedura.
Nasce a questo punto la necessità di definire la semantica in tre maniere diverse, che differiscono nella scelta delle regole di scope per variabili e procedure:

- scope dinamico per variabili e procedure,
- scope dinamico per variabili e scope statico per procedure,
- scope statico per variabili e procedure.

Scope dinamico per variabili e procedure

L'idea generale è che per eseguire la chiamata di procedura p, mediante lo statement p, dobbiamo eseguirne il corpo. Per questo motivo dobbiamo tenere traccia dell'associazione fra nomi di procedura e corpi di procedura. Per facilitare ciò introduciamo la nozione di *ambiente*. Dato un nome di procedura, l'*ambiente* env_P fornisce il relstivo corpo. env_P si configura quindi come una funzione

$$\mathbf{Env}_P = \mathbf{Pname} \longrightarrow \mathbf{Stm}$$

Il passo successivo consiste nell'includere l'ambiente all'interno delle regole, parametrizzandole

$$env_P \vdash < S, s > \to s'$$

La presenza dell'ambiente garantisce che possiamo sempre accedervi e richiedere i corpi delle procedure una volta conosciuto il nome. La regola di chiamata a procedura viene stilizzata

$$\frac{env_P \vdash < S, s > \to s'}{env_P \vdash < \texttt{call}\ p, s > \to s'} \text{ con } env_p\ p = S$$

Le dichiarazioni di procedure seguono quelle delle variabili, prima dell'esecuzione degli statement. Ogni dichiarazione di procedura aggiunge all'ambiente un nuovo legame fra nome e corpo.

Scope statico per procedure

Allo scopo di legare il corpo di una procedura con il proprio nome e l'ambiente esistente al momento della dichiarazione, estendiamo la connotazione di quest'ultimo

$$\mathbf{Env}_P = \mathbf{Pname} \longrightarrow \mathbf{Stm\ x\ Env}_P$$

La definizione ricorsivo non comporta problemi in quanto l'ambiente viene istanziato inizialmente vuoto e accresciuto man mano con le dichiarazioni di procedura. In caso di scope statico abbiamo la necessità, in caso di chiamata di procedura ricorsiva, di aggiornare l'ambiente durante la chiamata, per assicurarci che l'ambiente contenga anche il legame fra il corpo e il nome della procedura corrente:

$$\frac{env'_P[p \mapsto (S, env'_P)] \vdash < S, s > \to s'}{env_P \vdash < \texttt{call}\ p, s > \to s'} \text{ con } env_p\ p = (S, env_P)$$

Scope statico per variabili

Per ottenere scope statico per variabili oltreche per procedure, sostituiamo lo stato presente all'interno delle configurazioni con due mappature: un *ambiente di variabili* che associa un locazione ad ogni identificatore di variabile, e uno *store* che lega ad ogni locazione un valore. Definiamo formalmente un elemento env_v come

$$\mathbf{Env}_V = \mathbf{Var} \longrightarrow \mathbf{Loc}$$

dove **Loc** è un insieme di locazioni(per semplicità considerabili come interi).Uno store *sto* è invece un elemento dell'insieme

$$\mathbf{Store} = \mathbf{Loc} \cup \{\mathbf{next}\} \longrightarrow \mathbf{Loc}$$

dove next punta alla prima locazione di memoria libera. Piuttosto che avere una singola mappatura identificatore-valore come in precedenza, abbiamo scomposto lo stato in due parti, ricomponibili attraverso $s = sto \circ env_V$. Possiamo in questo modo mantenere in memoria diverse istanze della stessa variabile, dichiarata in blocchi differenti, associando ad ognuna una locazione diversa e immagazzinando l'informazione insieme all'ambiente delle procedure, per risolvere i riferimenti non locali. Quest'ultimo viene definitivamente modificato in

$env_V, env_P \vdash < S, sto > \rightarrow sto'$

mentre la struttura dell'ambiente procedurale sfocia in

$$\mathbf{Env}_P = \mathbf{PName} \longrightarrow \mathbf{Stm\ x\ Env}_V\ \mathbf{x\ Env}_P$$

Nel caso in cui la mole di informazione legate al contesto semantico di cui si vuole tenere traccia risulti molto vasta, possiamo scomporre l'ambiente in sotto-componenti a loro volta ricorsivamente scomponibili. Il linguaggio XQuery di interrogazione di dati XML è un chiaro esempio di questo utilizzo modulare dell'ambiente.

In XQuery l'ambiente contiene tutte le informazioni che posso influenzare la valutazione delle espressioni ed è organizzato in due macro moduli: ambiente statico (*statEnv*) e ambiente dinamico (*dynEnv*).*statEnv* raggruppa i componenti che sono disponibili durante l'analisi statica, durante la quale possono essere anche modificati. *statEnv* è presente anche durante l'analisi dinamica.

Alcune delle parte più importanti che compongono *statEnv* sono:

- **statEnv.namespace**: un ambiente di mappatura per i nomi in-scope. Prende in input un prefisso e fornisce l'URI collegato;
- **statEnv.typeDefn**: un ambiente di mappatura fra nomi di tipo e rispettiva definizione;
- **statEnv.elemDecl**: un ambiente di mappatura fra nomi di elemento e rispettiva definizione;
- **statEnv.attrDecl**: un ambiente di mappatura fra nomi di attributo e rispettiva definizione;
- **statEnv.varType**: un ambiente di mappatura fra nomi in-scope di variabile e rispettivo tipo.
- **statEnv.funcType**: un ambiente di mappatura fra nomi in-scope di funzione e rispettivo tipo.

  ⋮

Alcune delle parti fondamentali dell'ambiente dinamico sono:

- **statEnv.namespace**: un ambiente di mappatura per i nomi in-scope. Prende in input un prefisso e fornisce l'URI collegato;
- **dynEnv.funcDefn**: un ambiente di mappatura fra nomi di funzione(e parametri) in-scope e rispettivo corpo;

- **dybEnv.varValue**: un ambiente di mappatura fra nomi di variabile e valore corrente.
  ⋮

1.5 Framework Derivati

Nel corso degli anni numerosi studiosi, prendendo spunto dal lavoro seminale di Plotkin, hanno elaborato framework alternativi per esprimere la semantica operazionale. Le differenze con SOS non riguardano in questi casi solamente le restrizioni imposte alla sintassi delle regole, ma bensì una diversa concezione del contenuto stesso delle regole.

1.5.1 Semantica Naturale

Definita anche 'SOS big step'(in contrapposizione con la SOS tradizionale small step) la semantica naturale è un framework di semantica operazionale introdotto da G. Kahn nel 1987. Si differenzia principalmente dall'originale Plotkiniana per la mancanza di esplicitazione dei singoli passi di computazione intermedia, passando direttamente da configurazione iniziale a terminale e guardando al processo con cui vengono ottenuti i risultati in maniera più globale.

Mentre in SOS le transizioni sono nella forma $< S, s > \Rightarrow < S', s' >$,nel caso in cui il passo di computazione sfoci in una configurazione intermedia, oppure $< S, s > \Rightarrow s'$, in caso di passo di computazione terminale, all'interno del framework della semantica operazionale solo il secondo schema è contemplato. Ogni evoluzione dei sintagmi durante la computazione rimane dunque nascosta, per concentrarsi unicamente sulla relazione fra *statoiniziale* e *statofinale*. Le regole assumono l'aspetto seguente:

$$\frac{< S_1, s_1 > \rightarrow s'_1, \ldots, < S_n, s_n > \rightarrow s'_n}{< S, s > \rightarrow s'} \text{ if } \ldots$$

1.5.2 SOS Modulare

La SOS Modulare(**MSOS**) è una forma di SOS che assicura un alto grado di modularità: le regole di transizione sono completamente indipendenti dalla presenza o meno di altri costrutti del linguaggio di cui si da la descrizione. In caso di ampliamento o cambiamento del linguaggio descritto le regole di semantica non necessitano mai di una riformulazione.

Per esplicare in maniera sufficientemente comprensibile l'idea fondazionale del framework MSOS introduciamo brevemente la definizione di categoria:

Definizione 1.5.1. (Categoria) *Una categoria consiste in:*

- *Un insieme di oggetti O,*
- *un insieme di morfismi(mappa fra oggetti dell'insieme O) A.*

Sono definite le funzioni 'source' e 'target' da A a O, una funzione parziale da A^2 ad A per la composizione di morfismi, e una funzione da O ad A per esplicitare il morfismo identità per ogni oggetto.

L'idea basilare della MSOS è di incorporare tutte le entità semantiche come componenti delle *label*. MSOS introduce un'innovativa forma di LTS denominati **Arrow LTS**, in cui le *label* sono frecce(morfismi) di una categoria, che operano sugli oggetti della medesima. Vediamo di seguito un esempio di regola che riporta la proprietà di componibilità delle label.

$$\frac{\gamma \stackrel{\alpha_1}{\rightarrow}^{+} \gamma_1 \qquad \gamma_1 \stackrel{\alpha_2}{\rightarrow}^{+} \gamma_2 \qquad \alpha = \alpha_1 ; \alpha_2}{\gamma \stackrel{\alpha}{\rightarrow}^{+} \gamma_2}$$

Differentemente dalla SOS tradizionale Plotkiniana, in cui le configurazioni coinvolgono sintassi(astratta), valori computati e costituenti semantici(ambiente e store), nelle regole di MSOS solo la sintassi ed i valori computati fanno parte delle configurazioni, mentre tutti i componenti semantici sono localizzati nelle label, conferendo in questa maniera la proprietà di modularità al framework. In caso di aggiunta di strutture come ambienti e store al linguaggio le regole risulterebbero intatte e solo la categoria delle label verrebbe alterata.

Possiamo ora passare ad una formalizzazione, esponendo il concetto di 'Arrow-Labelled Transition Sistem':

Definizione 1.5.2. (Arrow-Labelled Transition Sistem)*(ALTS) Un ALTS è una quadrupla* (Γ, A, $\rightarrow$,T) *dove A è una categoria con* $|A|$ *insieme degli oggetti e Morph(A) insieme dei morfismi (o frecce),* Γ *è l'insieme delle configurazioni, T quello delle configurazioni terminali e* $\rightarrow$ *l'insieme delle regole di transizione.*

Per ogni freccia $\alpha \in Morph(A)$ posso determina l'oggetto 'source' con $pre(\alpha)$ e il 'target' con $post(\alpha)$;ad ogni oggetto $o \in |A|$ corrisponde una freccia identità indicata con $id(o)$. La composizione di frecce $\alpha 1; \alpha 2$ è definita solamente nel caso in cui $post(\alpha 1) = pre(\alpha 2)$.

Poniamo ora l'attenzione sull'insieme A. A solitamente non è un'unica categoria ma una struttura modulare, divisa in componenti a cui accedere tramite un indice ognuno dei quali è a sua volta una categoria. Con la scrittura $\alpha.i$ indichiamo l'i-esima categoria di A. Indagando la natura di questi componenti, possiamo classificarli in tre gruppi:

- $read-only$: in cui gli oggetti posso essere ispezionati ma non modificati dai morfismi durante la transizione;
- $read - write$ in cui gli oggetti possono essere sia ispezionati che modificati dai morfismi durante la transizione;
- $write - only$ in cui glli oggetti non sono ispezionabili, ma risultano come output della transizione.

Indicando con $\alpha.i$ l'oggetto della i-esima componente in ingresso alla transizione e $\alpha.i'$ il medesimo oggetto in uscita, notiamo che per i componenti $read - only$ $\alpha.i'$ è indefinito, per i $read - write$ sono definiti sia $\alpha.i$ che $\alpha.i'$ mentre per i $write - only$ solo $\alpha.i'$ è definito.

Possiamo agilmente passare da un ALTS ad un tradizionale LTS riportando le informazioni sullo stato all'interno delle configurazioni. Gli oggetti di componenti $read - only$ verranno lasciati immutati nella configurazione di ingresso e uscita, quelli $read - write$ verranno riportati secondo il loro cambiamento, mentre gli oggetti $write - only$ vengono incorportati all'interno dell'insieme Act delle azioni.

Riportiamo a titolo di esempio un paio di regole in MSOS riguardanti i costrutti di assegnamento e dichiarazione. Il costrutto $(x \mapsto l)$ indica un ambiente composto dalla sola associazione fra l'identificatore x e la locazione di memoria l, mentre con spazio sopra le costanti nel regno dei naturali o booleani.

$$\frac{\{\rho, \sigma = \sigma_0, \sigma' = \sigma_0, \ldots\}\ \rho(x) = l}{x := con \stackrel{\{\rho, \sigma=\sigma_0, \sigma'=\sigma_0[l \mapsto con], \ldots\}}{\longrightarrow} \mathbf{nil}}$$

$$\frac{\{\rho, \sigma = \sigma_0, \sigma' = \sigma_0, \ldots\}\ l \notin dom(\sigma)}{\mathbf{var}\ x := con \stackrel{\{\rho, \sigma=\sigma_0, \sigma'=\sigma_0[l \mapsto con], \ldots\}}{\longrightarrow} (x \mapsto l)}$$

1.6 Utilizzo SOS

1.6.1 Domini di ricerca

Dopo una breve introduzione sull'argomento SOS passiamo ad una indagine su quali siano gli ambiti di maggior utilizzo. Come precedentemente accennato la semantica operazionale strutturale nasce dalla necessità di fornire un formalismo matematicamente rigoroso per esprimere il significato dei linguaggi di programmazione. Il lavoro di G. Plotkin *'A structural approach to operational semantics'* si pone proprio come primo obiettivo quello di fare chiarezza all'interno del mondo della semantica operazionale, fino a quel momento popolato da formalismi individuali e solo parzialmente formali, proponendo i sistemi transizionali come base formale per esprimere i passi di computazione di un programma sottoposto all'azione dell'interprete di una macchina astratta.

A differenza dell'approccio semantico denotazionale che pone l'attenzione verso il prodotto della computazione, il risultato raggiunto, quindi la funzione calcolata dal programma, la SOS si focalizza sulla formalizzazione di come il programma giunge a quel risultato, individuando in suoi passi intermedi, mostrando le progressive mutazioni del frammento sintattico protagonista dell'esecuzione. I linguaggi di programmazione costituiscono quindi il campo originario di utilizzo della semantica operazionale strutturale. Risulta comunque indispensabile sottolineare come l'opera di Plotkin non costituisca l'imposizione di uno strumento statico ed immutabile, ma piuttosto l'indicazione di una strada da percorrere per allontanarsi dalle tante scontatezze solitamente assunte al momento della definizione della specifica semantica di un linguaggio e avvicinarsi ad un'approccio metodologico che sia in grado di consolidare formalmente il significato dei costrutti di programmazione, ponendo un freno ai meccanismi di ambiguità e cattivo utilizzo altrimenti inevitabili.

Con il tempo l'ambito di utilizzo della SOS si è notevolmente allargato. Nella versione tradizionale della SOS plotkiniana le transizioni sono sprovviste di informazioni ausiliarie sulle frecce, esprimendo unicamente l'avanzamento da una configurazione sintattica ad un'altra. In fase di definizione del linguaggio CCS, il suo creatore Robin Milner cercando un formalismo in grado di esprimere la sintassi del linguaggio per calcolo di processi nascituro, trova un semplice stratagemma per declinare i sistemi transizionali alla sua esigenza: inserire sopra le frecce delle etichette che indichino le informazioni che la transizione ha necessità di esprimere. In questo modo si allarga

notevolmente l'ambito di utilizzo della semantica operazionale strutturale, venendo col tempo a costituire la modalità usuale per esprimere i meccanismi di funzionamento anche per quanto riguarda i sistemi concorrenti.

Con il suo utilizzo nel campo dei sistemi concorrenti, oltre alle informazioni sulle frecce, è stata introdotta la possibilità di avere una relazione di transizione non deterministica, compagna ideale di concorrenza e parallelismo. La necessità di esprimere una sequenza di esecuzione chiara e soprattutto unica, in un programma sequenziale, impone invece l'impossibilità di far transire una configurazione sintattica in diverse configurazioni di destinazione. Appurato che la semantica operazionale di un linguaggio di programmazione decreta indirettamente il modo di operare dell'interprete, un implementazione di quest'ultimo risulterebbe impossibile venendo meno il vincolo di determinismo.

Successivamente l'interesse del mondo accademico si è concentrato sulla SOS, tanto da far nascere un nuova branca di ricerca riguardante i formati delle regole e lo studio ravvicinato delle loro caratteristiche. La semantica operazionale strutturale passa quindi da linguaggio con cui esprimere il singnificato di programmi a vero e proprio oggetto di studio, dando vita alla Meta-Teoria SOS. Un formato di regola non viene più considerato solamente come l'insieme delle sue feature sintattiche, ma bensì come il legame esistente fra le restrizioni sintattiche imposte in esso e le proprietà della semantica operazionale esprimibile con regole di un determinato formato. Data una semantica operazionale, ci si focalizza in particolar modo a osservare quando due sistemi possano dirsi equivalenti in materia di comportamento o quando un sistema possa venir considerato restrizione stretta dell'altro.

All'interno della letteratura inerente la SOS i formati di regola proposti e su cui sono stati provati diversi meta-teoremi sono una miriade. Il passaggio da un formato all'altro si è reso necessario, a seconda dei casi, sia per ragioni di rilassamento dei vincoli sintattici imposti, al fine di riuscire ad esprimere la semantica di linguaggi comprendenti costrutti sempre più complessi, sia per scopi puramente meta-teorici, ragionando sulla conservatività o meno di certi risultati ottenuti su un determinato formato in seguito ad estensione o restrizione di esso. I formati utilizzati più di frequente all'interno di articoli sono descritti in dettaglio in un paragrafo dedicato.

A fine di sintesi riassumiamo nella Tabella 1.1 gli ambiti di utilizzo della semantica operazionale strutturale.

Semantica Operazionale Strutturale	
Ambito	Funzione
Linguaggi di programmazione	Fornire in modo formale significato ai costrutti sintattici del linguaggio in un'ottica orientata alla computazione passo-passo deterministica
Linguaggi di calcoli di processi	Esprimere formalmente il comportamento del sistema concorrente ricorrendo anche al non determinismo e comunicando l'azione intrapresa mediante etichette
Meta-Teoria	Oggetto di studio, in modo particolare interessano le relazioni che intercorrono fra formato di regola e teoremi di congruenza

Tabella 1.1: Ambiti di utilizzo SOS

1.6.2 Esempi

Passiamo ora in rassegna analitica alcuni esempi di utilizzo reale della SOS all'interno di specifiche di linguaggi di programmazione, calcolo di processi e articoli meta-teorici, cercando di mettere in risalto quali caratteristiche sintattiche sono presenti, in relazione alla varietà esposta nel paragrafo precedente.

Jinja java-like language

Cominciamo con l'analisi della semantica operazionale strutturale di un linguaggio di alto livello simile a Java, dotato di tipi, denominato Jinja, presentato nell'articolo di Tobias Nipkow 'Jinja: Towards a comprehensive formal semantics for a Java-like language (2003)'. Partiamo dall'espressione dei termini della singnature. Nonostante la maggior parte dei manuali di riferimento tenda a fornire formalmente almeno la parte sintattica, solitamente sotto forma di grammatica BNF o simil BNF, la specifica sintattica di Jinja è definita a parole. I termini che compongono il linguaggio e che andranno a costituire la configurazioni su cui opereranno le transizioni, sono spiegati in diversi trafiletti dell'articolo che riportiamo di seguito in alcune figure. In relazione alla varietà di esposizione dei termini della signature, ci troviamo evidentemente di fronte ad un caso informale, in cui la descrizione è lasciata

al linguaggio naturale. La sintassi dei costrutti del linguaggio è esplicata dalla Figura 1.2.

The following expressions are supported by Jinja: creation of new objects (*New C*), casting (*Cast C e*), values (*Val v*), variable access (*Var V*), binary operations (e_1 «*bop*» e_2 where *bop* is one of *Add* or *Eq*), variable assignment ($V{:=}e$), field access ($\{D\}e{\cdot}F$)[1], field assignment ($\{D\}e_1{\cdot}F{:=}e_2$), method call ($e{\cdot}M(es)$), block with locally declared variable ($\{V{:}T;\ e\}$), sequential composition ($e_1;\ e_2$), conditional (*If* (e) e_1 *Else* e_2), loop (*While* (e) e'), exception throwing (*Throw* e) and catching (*Try* e_1 *Catch*$(C\ V)$ e_2). Note that there is no return statement because everything is an expression and returns a value. To ease notation we introduce some abbreviations:

$$\begin{array}{lcl} \mathit{addr}\ a & \equiv & \mathit{Val}(\mathit{Addr}\ a) \\ \mathit{null} & \equiv & \mathit{Val}\ \mathit{Null} \\ \mathit{true} & \equiv & \mathit{Val}(\mathit{Bool}\ \mathit{True}) \\ \mathit{false} & \equiv & \mathit{Val}(\mathit{Bool}\ \mathit{False}) \end{array}$$

Figura 1.2: Sintassi Jinja-1

Le configurazione delle transizioni di Jinja prevedono l'utilizzo di predicati oltre che di termini sintattici. I predicati sono specificati in un contesto P che identifica il programma corrente. I predicati utilizzabili in Jinja sono inseribili all'interno delle premesse delle regole di transizioni e rappresentano condizioni di applicabilità della regola stessa. L'elenco dei predicati ammessi ed il relativo significato è esposto in Figura 1.3.

***is-class* P C** means class C is defined in P.

$P \vdash D \preceq_C C$ means D is a **subclass** of C. It is transitive and reflexive.

$P \vdash C$ ***sees-method*** $M{:}Ts \rightarrow T = (pns, body)$ ***in*** D means that in P from class C a method M is visible in class D (taking overriding into account) with argument types Ts (a type list), result type T, formal parameter list *pns*, and body *body*.

$P \vdash C$ ***sees-field*** $F{:}T$ ***in*** D means that in P from class C a field F of type T is visible in class D.

$P \vdash C$ ***has-field*** $F{:}T$ ***in*** D means that in P a (not necessarily proper) superclass D of C has a field F of type T.

Figura 1.3: Predicati ammissibili come premesse all'interno delle regole di transizione in Jinja

All'interno delle regole di transizione della semantica operazionale di Jinja lo stato non è presentato a livello di contesto ma incluso nelle configurazio-

ni. Si tratta di uno stato modulare, composto da uno *heap* e uno *store*, due mappe di natura differenti come spiegato in Figura 1.4. Accedo alle sottocomponenti *heap* e *store* di s attraverso le funzioni d'ambiente hp e lcl. All'interno delle premesse viene fatto largo uso di funzioni semantiche,

4.1 State

The **state** during expression evaluation is a pair of a **heap** and a **store**. A store is a map from variable names to values. A heap is map from addresses to objects. An object is a pair of a class name and a field table, and a **field table** is a map from pairs (F, D) (where D is the class where F is declared) to values. It is essential to include D because an object may have multiple fields of the same name. The variable convention is that h is a heap, l is a store (the *l*ocal variables), and s a state. The projection functions hp and lcl are synonyms for fst and snd.

Figura 1.4: Descrizione dello stato presente nelle configurazioni

a cui vengono affidati diversi compiti, dalla classica mappatura tra mondo sintattico e aritmetico (funzione *binop*) alla richiesta di un indirizzo di memoria libero (funzione 'new-Addr') all'interno dello heap o dello store. La descrizione delle funzioni semantiche è lasciata completamente al linguaggio naturale, come testimonia la Figura 1.5, addirittura relegando la spiegazione del funzionamento di talune di esse all'intuito del lettore.

fields are set to their default values: function *new-Addr* returns a "new" address, i.e. *new-Addr* h = *Some* a implies h a = *None*; relation *has-fields* computes the list *FDTs* of all field declarations in and above C, where each field F of type T declared in class D is represented as a triple $((F,D),T)$; and *init-vars FDTs* maps each pair (F,D) to the default value of type T — the definition of the default value is irrelevant for our purposes, it suffices to know that it is *Some* rather than *None*.

Binary operations are evaluated from left to right. Function *binop* takes a binary operation and two values and applies the operation to the values — its precise definition is not important here.

Figura 1.5: Funzioni semantiche utilizzate nelle transizioni

Passiamo ora ad analizzare la conformazione sintattica delle regole di transizione della specifica semantica operazionale di Jinja. Possiamo trovare uno stralcio delle regole valutative dei costrutti fondamentali in Figura 1.6. Notiamo immediatamente la singolarità di presentazione delle premesse: non vi è nessuna strutturazione grafica del tipo numeratore/denominatore fra le

premesse e la conclusione delle regole. Entrambe vengono scritte in linea, legando fra di loro le premesse in AND con il simbolo ';' e separando le premesse dalla conclusione con il simbolo '$\Longrightarrow$' che sostituisce la linea di frazione, mentre il simbolo di transizione adoperato è '$\rightarrow$' senza la presenza di alcuna etichettatura.

New:

$[\![new\text{-}Addr\ h = Some\ a;\ P \vdash C\ has\text{-}fields\ FDTs;$
$h' = h(a \mapsto (C,\ init\text{-}vars\ FDTs))]\!]$
$\Longrightarrow P \vdash \langle New\ C,(h,\ l)\rangle \rightarrow \langle addr\ a,(h',\ l)\rangle$

$new\text{-}Addr\ h = None \Longrightarrow P \vdash \langle New\ C,(h,\ l)\rangle \rightarrow \langle throw\ OutOfMemory,(h,\ l)\rangle$

Cast:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle Cast\ C\ e,s\rangle \rightarrow \langle Cast\ C\ e',s'\rangle$

$P \vdash \langle Cast\ C\ null,s\rangle \rightarrow \langle null,s\rangle$

$[\![hp\ s\ a = Some\ (D,\ fs);\ P \vdash D \preceq_C C]\!] \Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),s\rangle \rightarrow \langle addr\ a,s\rangle$

$[\![hp\ s\ a = Some\ (D,\ fs);\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ (addr\ a),s\rangle \rightarrow \langle throw\ ClassCast,s\rangle$

$P \vdash \langle Cast\ C\ (Throw\ e),s\rangle \rightarrow \langle Throw\ e,s\rangle$

Binary operation:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e \ll bop \gg e_2,s\rangle \rightarrow \langle e' \ll bop \gg e_2,s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle Val\ v_1 \ll bop \gg e,s\rangle \rightarrow \langle Val\ v_1 \ll bop \gg e',s'\rangle$

$v = binop\ bop\ v_1\ v_2 \Longrightarrow P \vdash \langle Val\ v_1 \ll bop \gg Val\ v_2,s\rangle \rightarrow \langle Val\ v,s\rangle$

Figura 1.6: Esempi di regole di transizione in Jinja

Le configurazioni, come anticipato, alloggiano al loro interno sia i costrutti sintattici che lo stato. Quest'ultimo, essendo modulare, viene presentato come un elenco delle sue sottocomponenti h e l che identificano rispettivamente lo *heap* e lo *store* oppure semplicemente con s qualora risulti superfluo specificarne i moduli separatamente. A fini riepilogativi raccogliamo tutte le informazioni riguardanti la varietà sintattica presente nelle regole di transizione di Jinja nella Tabella 1.2.

π-calcolo

Come esempio di formalizzazione mediante SOS della semantica operazionale di un linguaggio di calcolo di processi, conserideriamo il π-calcolo, di Robin

Semantica Operazionale strutturale di Jinja		
FEATURE	PRESENZA	FORMA
Signature	SI	Linguaggio Naturale
Funzioni Semantiche	SI	Linguaggio Naturale
Configurazioni	SI	Sintassi + Stato
Ambienti	SI	Ambiente Modulare Notazione: • Composta: (h, l) • Singola: s
Label	NO	
Premesse Negative	NO	
Premesse infinite	NO	
Lookahead	SI	
Predicati	SI	Tra le premesse
Notazione Configurazione	SI	$< S, s >$
Notazione Ambienti	SI	A livello configurazione a destra del termine
Notazione Freccia	SI	$\rightarrow$

Tabella 1.2: Riepilogo feature sintattiche presenti nelle regole di transizione di Jinja

Milner, esposto dettagliatamente dallo stesso autore in ‘A Calculus of mobile processes (1989)’. Come con il caso precedente partiamo analizzando il modo in cui sono formalizzati i termini della signature. I termini della signature che verrano inseriti nelle transizioni, non sono in questo caso altro che i processi coinvolti all’interno del sistema concorrenziale che si va a descrivere. Robin Milner dimostra una grande attenzione verso la formalizzazione di ogni aspetto del suo calcolo a cominciare dalla sintassi dei termini espressa in forma simil BNF come mostrato in Figura 1.7.

All’interno della specifica semantica dei linguaggi di process calculi, difformemente a ciò che avviene per i linguaggi di programmazione, assume grande importanza l’insieme delle label, che identificano il tipo di azione che il processo compie nell’esecuzione di una transizione. Le label non solo

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \\
 & | & \overline{x}y\,.\,P \\
 & | & x(y)\,.\,P \\
 & | & \tau\,.\,P \\
 & | & (x)P \\
 & | & [x=y]P \\
 & | & P \mid Q \\
 & | & P+Q \\
 & | & A(y_1,\ldots,y_n)
\end{array}
$$

Figura 1.7: Grammatica stile BNF dei termini del calcolo

sono determinanti nel conferimento di significato all'intero calcolo, ma risultano assolutamente obbligatorie. Ogni transizione assumerà quindi la forma $P \xrightarrow{\alpha} Q$. Anche un transizione senza alcuna informazione di input o output viene comunque etichetta, dalla specifica label che rappresenta la transizione silente. Come chiarito in Figura 1.8, in cui si elencano le tipologie di label, nel π-calcolo le label non sono semplici costanti, ma bensì termini aperti composti da una coppia di variabili. Unica eccezione la transizione silente etichettata con τ.

α	Kind	Free/Bound	Polarity	fn(α)	bn(α)
τ	Silent	f	0	$\emptyset$	$\emptyset$
$\overline{x}y$	Free Output	f	$-$	$\{x,y\}$	$\emptyset$
$x(y)$	Input	b	$+$	$\{x\}$	$\{y\}$
$\overline{x}(y)$	Bound Output	b	$-$	$\{x\}$	$\{y\}$

Table 1: The actions.

Figura 1.8: Le label ammissibili nelle transizioni del π-calcolo

Non avendo a che fare con domini differenti come i naturali ed i booleani, come avviene nella maggior parte dei linguaggi di programmazione, non ho necessità di funzioni semantiche che mi traducano frammenti sintattici in diversi regni di significato. Allo stesso modo non ho una netta separazione, all'interno delle configurazioni portate a transire, fra sintassi e stato, in quanto entrambi sono presenti nella conformazione attuale del processo.

Passiamo ora in rassegna l'insieme delle regole di transizione del π-calcolo, mostrate in Figura 1.9.

$$\text{TAU-ACT}: \quad \frac{-}{\tau.P \xrightarrow{\tau} P} \qquad \text{OUTPUT-ACT}: \quad \frac{-}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$$

$$\text{INPUT-ACT}: \quad \frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}} \quad w \notin \mathrm{fn}((z)P)$$

$$\text{SUM}: \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{MATCH}: \quad \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$$

$$\text{IDE}: \quad \frac{P\{\widetilde{y}/\widetilde{x}\} \xrightarrow{\alpha} P'}{A(\widetilde{y}) \xrightarrow{\alpha} P'} \quad A(\widetilde{x}) \stackrel{\mathrm{def}}{=} P$$

$$\text{PAR}: \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$$

$$\text{COM}: \quad \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad \text{CLOSE}: \quad \frac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

$$\text{RES}: \quad \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'} \quad y \notin \mathrm{n}(\alpha) \qquad \text{OPEN}: \quad \frac{P \xrightarrow{\overline{x}y} P'}{(y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin \mathrm{fn}((y)P') \end{array}$$

Table 2: Rules of Action. Rules involving the binary operators $+$ and $|$ additionally have symmetric forms.

Figura 1.9: Le regole di transizione del π-calcolo

Notiamo immediatamente la grande pulizia delle regole, ognuna dotata di un nome di riferimento. Tutti i termini presenti all'interno delle transizioni appartengono alla signature o all'insieme delle label, con l'unica eccezione del costrutto $P\{y/x\}$ utilizzato per introdurre la nozione di sostituzione.

Non sono presenti predicati inseriti in posizione di premessa o sui termini, ma solo come condizioni di regola, solitamente coinvolgendo operazioni di tipo insiemistico sui nomi liberi di un processo oppure equazioni di definizione nella forma$A(\widetilde{y}) \stackrel{def}{=} P$. Le configurazioni dei processi, come si può facilmente notare, sono entità atomiche composte da termini che, per quanto complessi, sono singoli aggregati e non elenchi come accade nelle configurazioni dei linguaggi di programmazione che includono al loro interno le informazioni di stato. Le configurazioni non sono racchiuse fra parentesi angolari o altro tipo di delimitatori. Evidenziamo inoltre la mancanza di possibilità di lookeahead nelle premesse delle regole. Riassumiamo tutte le caratteristiche sintattiche delle regole SOS del π-calcolo nella Tabella 1.3.

Semantica Operazionale strutturale del π-calcolo		
FEATURE	PRESENZA	FORMA
Signature	SI	Simil BNF
Funzioni Semantiche	NO	
Configurazioni	SI	Sintassi dei processi e sostituzione
Ambienti	NO	
Label	SI	Termini aperti
Premesse Negative	NO	
Premesse infinite	NO	
Lookahead	NO	
Predicati	SI	a livello di regola
Notazione Configurazione	NO	
Notazione Ambienti	NO	
Notazione Freccia	SI	$\rightarrow$

Tabella 1.3: Riepilogo feature sintattiche presenti nelle regole di transizione del π-calcolo

Progetto HATS: il lunguaggio core ABS

Il linguaggio core ABS, presente all'interno del progetto HATS, nella sua specifica più recente, si pone come base di una sua versione full, che sia in grado di indirizzare software product lines. Cruciale per la sua funzione di appoggio sono la possibilità di supportare concorrenza e distribuzione, e la fornitura di un modello di comunicazione flessibile. Inoltre, come fondazione

di un metodo formale, deve avere una semantica definita formalmente. Per rispettare questo ultimo requisito si ricorre alla SOS.

Cominciamo con l'osservare la grammatica del linguaggio, atta a costruire i termini delle transizioni. La grammatica è data in stile simil BNF, focalizzandosi sulla sintassi astratta e trascurando quella concreta, superflua ai fini di una formalizzazione semantica. La Figura 1.10 riporta il sunto della sintassi astratta del linguaggio.

P	::=	$\overline{Dd}\ \overline{F}\ \overline{In}\ \overline{Cl}\ [B]$	program
In	::=	interface I [extends I^+] $\{\overline{M_s}\}$	interface declaration
Cl	::=	class $C(\overline{T f})$ [implements I^+] $\{\ \overline{T f}\ [B]\ \overline{M}\ \}$	class definition
M	::=	$M_s\ B$	method definition
M_s	::=	$T\ m(\overline{T x})$	method signature
B	::=	$\{\ \overline{T x}\ s\ \}$	blocks
T	::=	$I \mid D \mid$ Fut$(T) \mid$ Void $\mid$ Bool $\mid$ Guard	types
v	::=	$x \mid$ this.f	state variables
e	::=	$e_p \mid e_e$	expressions
e_p	::=	$v \mid e_f \mid$ null $\mid e_p = e_p$	pure expressions
e_e	::=	new [cog] $C(\overline{e_p}) \mid e_p!m(\overline{e_p}) \mid e_p.m(\overline{e_p}) \mid e_p$.get	expressions with side effects
s	::=	$v := e \mid$ await $g \mid$ skip $\mid$ suspend $\mid e$	statements
		$\mid$ if (e_p) s else $s \mid$ while (e_p) $s \mid s;s$	
g	::=	$v? \mid g \wedge g \mid e_f$	guards
Dd	::=	data D $\{\overline{Co(\overline{T})}\}$	data type declaration
F	::=	def T $fn(\overline{T z}) = e_f$	function declaration
t	::=		term
		z	logical variables
	$\mid$	$Co(\overline{e_p})$	constructor expressions
	$\mid$	(e_p, e_p)	pair constructor
p	::=	$z \mid Co(\overline{p}) \mid (p,p)$	pattern
e_f	::=	t	functional expressions
	$\mid$	let $z{:}T = e_p$ in e_f	local value definition
	$\mid$	$fn(\overline{e_p})$	function application
	$\mid$	case e_p of $\overline{b}$	case expression
b	::=	$p \rhd e_f$	branch

Figure 3.1: ABS abstract syntax

Figura 1.10: Sintassi Astratta del core di ABS

La particolarità del linguaggio consta nella sua modularità. Oltre al lin-

guaggio definito con la grammatica di Figura 1.11 il core di ABS ha una parte funzionale per le espressioni e una parte object-oriented adibita alla descrizione della concorrenza presente all'interno del sistema. Partiamo con l'analisi della parte funzionale del linguaggio. I termini coinvolti nel formare espressioni sono formalizzati nella grammatica di figura X.X.

v	$::=$	$x \mid t$	values
t	$::=$		term
		$Co(\bar{t})$	constructor term
	$\mid$	(t,t)	pair constructor
p	$::=$	$x \mid Co(\bar{p}) \mid (p,p)$	pattern
e	$::=$		expressions
		v	values
	$\mid$	$\mathsf{let}\, x{:}T = v\, \mathsf{in}\, e$	value definition
	$\mid$	$\mathsf{let}\, x{:}T = \lambda x{:}T.e_f\, \mathsf{in}\, e$	function definition
	$\mid$	$\mathsf{let\ data}\, D = Co(T)\ldots Co(T)\, \mathsf{in}\, e$	data type definition
	$\mid$	$\mathsf{case}\ v\ \mathsf{of}\ \bar{b}$	case
	$\mid$	$e\ e$	function application
b	$::=$	$p \triangleright e$	branch

Table 4.1: Abstract syntax of the expression language

Figura 1.11: Sintassi astratta della parte funzionale di ABS

Il linguaggio funzionale di ABS è tipato e dotato di costruttori di termini, costruttori di coppia, e definizione di tipi di dato in un contesto valutativo di un espressione. Il contesto, come avviene sempre nei linguaggi funzionali venendo meno il concetto di valori in memoria e store, rappresenta il fondamento su cui basare la computazione. Un contesto Γ raccoglie infatti tutte le informazioni di ambiente necessarie alla valutazione dell'espressione. Avviciniamoci ora alla semantica del linguaggio mostrata in Figura 1.12.

Come in altri casi precedentemente posti ad analisi, le regole sono denominate univocamente. Le prime tre sono assiomi in quanto non presentano premesse, mentre le restanti sono nella forma premesse/conseguenza. Evidenziamo come in ogni regola sia presente il contesto Γ posto a sinistra dei termini all'interno delle configurazioni, che costituisce l'ambiente di valutazione. Il contesto può anche non essere il semplice Γ, ma essere arricchito di definizioni di tipo e costruttori di termini, che vanno a collocarsi in una lista al suo seguito. Difformemente alla prassi classica nei linguaggi di program-

$$\Gamma \vdash \text{let} x{:}T = (\text{let} x' : T' = e'_1 \text{ in } e'_2) \text{ in } e \rightarrow \Gamma \vdash \text{let} x' : T' = e'_1 \text{ in } (\text{let} x : T = e'_2 \text{ in } e) \qquad \text{R-SEQ}$$

$$\Gamma \vdash \text{let} x{:}T = v \text{in } e \rightarrow \Gamma \vdash e[v/x] \qquad \text{R-LET}$$

$$\Gamma \vdash \text{let } f{:}T' = \lambda(x{:}T).e_1 \text{ in} e_2 \rightarrow \Gamma, f : T' = \lambda x{:}T.e_1 \vdash e_2 \qquad \text{R-FDEC}$$

$$\frac{\Gamma' = \Gamma, D{:}*, \ldots Co_i{:}T_i \rightarrow n, \ldots}{\Gamma \vdash \text{let data } D = Co_1(T_1) \ldots Co_n(T_n) \text{ in } e \rightarrow \Gamma' \vdash e} \text{ R-DDEC}$$

$$\frac{}{\Gamma \vdash \text{let } x{:}T = (\text{if } v = v \text{ in } e_1 \text{ else} e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let } x{:}T = e_1 \text{ in } e} \text{ R-COND}_1$$

$$\frac{v_1 \neq v_2}{\Gamma \vdash \text{let } x{:}T = (\text{if } v_1 = v_2 \text{ in } e_1 \text{ else} e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let } x{:}T = e_1 \text{ in } e} \text{ R-COND}_2$$

$$\frac{\Gamma = \Gamma_1, f : T = \lambda(y{:}T).e_1, \Gamma_2}{\Gamma \vdash \text{let} x{:}T = f(e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let} x{:}T = e_1[y/e_2] \text{ in } e} \text{ R-APP}$$

Figura 1.12: Regole transizionali della parte funzionale del core di ABS

mazione, le configurazioni non presentano alcun delimitatore. Analogamente ad altri casi, anche nella parte funzionale di ABS viene fatto uso del costrutto extra-sintattico al linguaggio $[y/x]$ per indicare una sostituzione. Tale costrutto, giustamente non incluso all'interno della grammatica, non è accessibile ai programmatori, ma necessario in sede di formalizzazione semantica per dare un significato ai binding del costrutto `let-in`.

Dopo esserci soffermati sulla parte funzionale del core di ABS poniamo l'attenzione sulla sua parte object-oriented per descrivere concorrenza. La sintassi è anche in questo caso fornita con una grammatica simil BNF e viene mostrata in Figura 1.13.

Oltre ai termini utilizzabili a livello di programma, le configurazione comprendono al proprio interno anche costrutti semantici ausiliari indispensabili a delineare la completa semantica operazionale del linguaggio. Questi termini sono identificati dalla categoria sintattica s della grammatica, posta in fondo.

Le regole transizionali sono invece elencate in Figura 1.14 e Figura 1.15.

Da un primo colpo d'occhio possiamo immediatamente notare la stravaganza di notazione utilizzata per la freccia di transizione '$\rightsquigarrow$', stante a sottolineare il carattere di riduzione delle regole. A differenza delle configurazioni della parte funzionale del linguaggio, in questo caso le configurazioni dispongono al proprio interno di parentesi angolari '< ... >', ma da un attenta analisi ci possiamo facilmente accorgere che non isolano la configurazione

$$
\begin{array}{rcll}
P & ::= & o[b, C, \sigma] & \text{object } o \\
 & | & n\langle b, o, s, \sigma\rangle & \text{task } n \\
 & | & b[l] & \text{lock } b \text{ for a concurrent object group} \\
 & | & P \parallel P & \text{composition .}
\end{array}
$$

$$
s \quad ::= \quad \ldots \mid \mathsf{grab}\,(b) \mid \mathsf{release}(b) \mid \mathsf{let}\, y{:}T = s\, \mathsf{in}\, s
$$

Figura 1.13: Sintassi dei termini del linguaggio object-oriented di core ABS presenti nelle regole di transizione

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = v\, \mathsf{in}\, s\rangle \leadsto n\langle b,o,\sigma,s[v/z]\rangle \qquad \text{R-RED}$$

$$n\langle b,o,\sigma,\mathsf{let}\, z_2{:}T_2 = (\mathsf{let}\, z_1{:}T_1 = s_1\, \mathsf{in}\, s)\, \mathsf{in}\, s'\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z_1{:}T_1 = s_1\, \mathsf{in}\, (\mathsf{let}\, z_2{:}T_2 = s\, \mathsf{in}\, s')\rangle \qquad \text{R-SEQ}$$

$$n\langle b,o,\sigma,\mathsf{skip};s\rangle \leadsto n\langle b,o,\sigma,s\rangle \qquad \text{R-SKIP}$$

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = x\, \mathsf{in}\, s\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z{:}T = \sigma(x)\, \mathsf{in}\, s\rangle \qquad \text{R-PURE}$$

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = (\mathsf{if\ true\ then}\, s_1\, \mathsf{else}\, s_2)\, \mathsf{in}\, s\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z{:}T = s_1\, \mathsf{in}\, s\rangle \qquad \text{R-COND}_1$$

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = (\mathsf{if\ false\ then}\, s_1\, \mathsf{else}\, s_2)\, \mathsf{in}\, s\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z{:}T = s_2\, \mathsf{in}\, s\rangle \qquad \text{R-COND}_2$$

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\, \mathsf{in}\, s\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z{:}T = s_1\, \mathsf{in}\, s\rangle \qquad \text{R-COND}_3$$

$$\frac{v_1 \neq v_2}{n\langle b,o,\sigma,\mathsf{let}\, z{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\, \mathsf{in}\, s\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z{:}T = s_2\, \mathsf{in}\, s\rangle} \ \text{R-COND}_4$$

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = (\mathsf{while\ true\ do}\, s_1)\, \mathsf{in}\, s_2\rangle \leadsto n\langle b,o,\sigma,\mathsf{let}\, z' : T = s_1\, \mathsf{in}(\mathsf{let}\, z{:}T = (\mathsf{while\ true\ do}\, s_1)\, \mathsf{in}\, s_2)\rangle \qquad \text{R-WHILE}_1$$

$$n\langle b,o,\sigma,\mathsf{let}\, z{:}T = (\mathsf{while\ false\ do}\, s_1)\, \mathsf{in}\, s_2\rangle \leadsto n\langle b,o,\sigma,s_2\rangle \qquad \text{R-WHILE}_2$$

$$n\langle b,o,\sigma,x := v;s\rangle \leadsto n\langle b,o,\sigma[x \mapsto v],s\rangle \qquad \text{R-ASSIGN}$$

$$o[b,C,\sigma] \parallel n\langle b,o,\sigma',\mathsf{let}\, z : T = o.f\, \mathsf{in}\, s\rangle \leadsto o[b,C,\sigma] \parallel n\langle b,o,\sigma',\mathsf{let}\, z : T = \sigma(f)\, \mathsf{in}\, s\rangle \qquad \text{R-FLOOKUP}$$

$$o[b,C,\sigma] \parallel n\langle b,o,\sigma',o.f := v;s\rangle \leadsto o[b,C,\sigma[f \mapsto v]] \parallel n\langle b,o,\sigma',s\rangle \qquad \text{R-FUPDATE}$$

Figura 1.14: Regole di transizione per la parte object-oriented di ABS(1)

ma sono invece parte della sintassi dei termini. Come nella maggior parte dei linguaggi adibiti al calcolo concorrenziale, anche in questo caso lo stato o l'ambiente non sono esplicitamente esplicitamente indicati ma si trovano a livello di termine. Si nota l'uso di qualche predicato inserito fra le premesse, come nel caso della regola R-COND4, che fa utilizzo di operatori di confronto. Altri predicati fanno uso di costrutti per testare l'essere *fresh* di un nome, o la natura effettiva di un termine. L'uso dei costrutti ausiliari si fa massiccio nel secondo blocco di regole. I costrutti $grab(b)$ e $release(b)$ sono usati come primitive di sincronizzazione dei processi, al fine di gestire i lock del sistema.

$$\frac{n' \textit{ fresh} \quad \textit{body}(m,C) = s(\overline{x}) \quad s_{task} = (\mathsf{let}\, z'{:}T = s[o'/\mathsf{this}][\overline{v}/\overline{x}]\,\mathsf{in}\, z')}{o'[b,C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = o'.m(v)\,\mathsf{in}\, s_2\rangle \rightsquigarrow o'[b,C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = n'.\mathsf{get}\,\mathsf{in}\, s_2\rangle \parallel n'\langle b,o',\sigma_{init}, s_{task}\rangle} \ \text{R-SCALL}$$

$$\frac{n' \textit{ fresh} \quad \textit{body}(m,C) = s(\overline{x}) \quad s_{task} = (\mathsf{let}\, z{:}T = \textit{grabs}(b); s[o'/\mathsf{this}][\overline{v}/\overline{x}]\,\mathsf{in}\,\mathsf{release}(b); z)}{o'[b',C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = o'!m(v)\,\mathsf{in}\, s_2\rangle \rightsquigarrow o'[b',C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = n'\,\mathsf{in}\, s_2\rangle \parallel n'\langle b,o',\sigma, s_{task}\rangle} \ \text{R-ACALL}$$

$$\frac{o' \text{ and } n' \text{ fresh} \quad B = \{\overline{T\,f}\; s'\} \text{ is initializer block of } C \quad s_{task} = (s'[o'/\mathsf{this}]; o')}{n\langle b,o,\sigma,\mathsf{let}\, z{:}T = \mathsf{new}\ C(\overline{v})\,\mathsf{in}\, s\rangle \rightsquigarrow n'\langle b,o',\sigma'_{init}, s_{task}\rangle \parallel o'[b,C,\sigma_{init}[\overline{f}' \mapsto \overline{v}]] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = n'.\mathsf{get}\,\mathsf{in}\, s\rangle} \ \text{R-NEWO}$$

$$\frac{b', o', \text{and } n' \text{ fresh} \quad B = \{\overline{T\,f}\; s'\} \text{ is initializer block of } C \quad s_{task} = (s'[o'/\mathsf{this}]; \mathsf{release}(b'))}{n\langle b,o,\sigma,\mathsf{let}\, z{:}T = \mathsf{new}\ C(\overline{v})\,\mathsf{in}\, s\rangle \rightsquigarrow b'[\top] \parallel n'\langle b',o',\sigma'_{init}, s_{task}\rangle \parallel o'[b',C,\sigma_{init}] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = o'\,\mathsf{in}\, s\rangle} \ \text{R-NEWOG}$$

$$b[\bot] \parallel n\langle b,o,\sigma,\mathsf{grab}\ (b); s\rangle \rightarrow b[\top] \parallel n\langle b,o,\sigma,s\rangle \quad \text{R-GRAB}$$

$$b[\top] \parallel n\langle b,o,\sigma,\mathsf{release}(b); s\rangle \rightarrow b[\bot] \parallel n\langle b,o,\sigma,s\rangle \quad \text{R-RELEASE}$$

$$n\langle b,o,\sigma,\mathsf{suspend}; s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b); \mathsf{grab}\ (b); s\rangle \quad \text{R-SUSPEND}$$

$$n_1\langle b',o',\sigma',\mathsf{let}\, x : T = n_1.\mathsf{get}\,\mathsf{in}\, t\rangle \parallel n_2\langle b,o,\sigma,v\rangle \rightsquigarrow n_1\langle b',o',\sigma',\mathsf{let}\, x : T = v\,\mathsf{in}\, t\rangle \parallel n_2\langle b,o,\sigma,v\rangle \quad \text{R-GET}$$

$$\frac{\textit{futnames}(g) = n_1 \dots n_k \text{ with } k \geq 1}{n_1\langle b_1,o_1,\sigma_1,v_1\rangle \parallel \dots \parallel n_k\langle b_k,o_k,\sigma_k,v_k\rangle \parallel n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b); \mathsf{await}\ (g)[\overline{\mathsf{true}}/\overline{n?}]; s\rangle} \ \text{R-AWAIT}^?_1$$

$$\frac{n' \in \textit{futnames}(g) \quad s' \neq v}{n'\langle b',o',\sigma',s'\rangle \parallel n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b)\ \mathsf{await}\ (g); s\rangle} \ \text{R-AWAIT}^?_2$$

$$\frac{\textit{futnames}(g) = \emptyset \quad [\![g]\!]_\sigma = \textit{true}}{n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,s\rangle} \ \text{R-AWAIT}_1 \qquad \frac{\textit{futnames}(g) = \emptyset \quad [\![g]\!]_\sigma = \textit{false}}{n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b); \mathsf{await}\ (g); s\rangle} \ \text{R-AWAIT}_2$$

Figura 1.15: Regole di transizione per la parte object-oriented di ABS(2)

Il costrutto *suspend* di cui si fa utilizzo in alcune premesse non è altro che zucchero sintattico per un *grab* seguito a ruota da un *release*. Questa parte del linguaggio ABS possiede un'eccezione rispetto ai consueti linguaggi di sistemi concorrenti, per quanto riguarda le label. Infatti anche questo modulo del linguaggio non prevede l'etichettatura delle label. La motivazione si trova guardando con occhio più attento la capacità espressiva del linguaggio. Infatti i termini disponibili sono in grado solamente di regolare un sistema di sincronizzazione basato su lock, senza la possibilità di comunicazione diretta fra processi.

Come atto conclusivo di questa indagine su alcuni esempi di utilizzo della SOS, riepiloghiamo anche le caratteristiche della varietà linguistica nel linguaggio core ABS in Tabella 1.4.

1.6.3 Peso delle feature

Cerchiamo ora di trarre le somme di tutto il lavoro di ricerca fatto fino a questo punto al fine di riuscire ad esprimere un giudizio sulla rilevanza effet-

Semantica Operazionale strutturale di core ABS		
FEATURE	PRESENZA	FORMA
Signature	SI	Simil BNF
Funzioni Semantiche	SI	linguaggio naturale
Configurazioni	SI	Sintassi dei termini e sostituzione
Ambienti	SI	contesto di transizione $\Gamma \vdash$
Label	NO	
Premesse Negative	NO	
Premesse infinite	NO	
Lookahead	NO	
Predicati	SI	tra le premesse
Notazione Configurazione	NO	
Notazione Ambienti	SI	a sinistra della configurazione
Notazione Freccia	SI	$\rightsquigarrow$

Tabella 1.4: Riepilogo feature sintattiche presenti nelle regole di transizione di core ABS

tiva della varietà linguistica all'interno della famiglia SOS. Ciò che si evince da decine di articoli scientifici e manuali letti sull'argomento è che in realtà, nonostante ci troviamo davanti ad uno spazio di variabilità notevole, le varianti che trovano impiego più di frequente sono un sottoinsieme decisamente piccolo rispetto alle disponibilità relegando certe variazioni ad esercizi di stravaganza sintattica di certi autori.

Partiamo dalla struttura dei termini. Come espresso dalla definizione formale di sistema di specifiche transizionali, i termini protagonisti delle transizioni, sono forniti da una *signature*, una specie di loro 'fabbrica' che ammette variabili, costanti e funzioni n-arie con cui costruire ricorsivamente i termini. A seconda dei casi la signature può essere esterna o interna. Interna quando il linguaggio di specifica semantica in uso presenta un meccanismo di costruzione dei termini, esterna altrimenti. Nel primo caso è il linguaggio stesso che si preoccupa della sintassi dei termini, adottando una grammatica stile BNF o un a definizione più vicina agli abstract data type, quando al suo interno è consentito l'appoggio a tipi di termini di base, da cui derivare quelli personalizzati. Nel secondo caso i termini mi vengono invece forniti da una

specifica sintattica esterna, che può essere simile a quella interna a livello di metodi di definizione, ma può comprendere anche una descrizione informale tramite prosa in linguaggio naturale, con tutte le ambiguità del caso. La 'signature' può comprendere al suo interno sorte differenti. Nel qual caso, ogni sorta dovrebbe specificare un meccanismo per la sintesi dei suoi termini. Le signature che comprende n-sorte differenti sono comunque rare in letteratura, riguardando solamente qualche formalismo per il calcolo di processi. La maggior parte dei linguaggi di programmazione e di calcolo di processi fanno utilizzo di grammatica simil BNF, esterna o interna, mentre la definizione in abstract data type è più comune nei tool che forniscono un'implementazione della SOS. Sottolineiamo come, in tutti gli esempi visti in letteratura, le differenti definizioni dei termini compaiano sempre mutuamente esclusive l'una rispetto all'altra, senza mostrare forme di ibridazione alcuna.

Spostandoci verso i costrutti extra-sintattici che vengono spesso inclusi nelle configurazioni, non possiamo citare l'importanza delle funzioni semantiche. Il loro utilizzo spazia dalla mappatura fra regni diversi, in modo particolare tra sintattico e numerico, alla fornitura di primitive necessarie (o più efficaci e veloci) al fine di esprimere determinati concetti, da cui dipende la completezza semantica del sistema. Le funzioni semantiche più comuni sono utilizzate nei linguaggi di programmazione per tradurre un termine sintattico che rappresenta un numero naturale, un valore booleano,ecc... nel corrispondente regno di appartenenza, o nel calcolo dei processi per fornire primitive di sincronizzazione in ausilio alle transizioni. Ammiriamo negli esempi in letteratura la possibilità di avere un invocazione esplicita, tramite una vera e propria invicazione di funzione, o implicita, data dal contesto. Entrambe le forme trovano un riscontro equivalente in termini di frequenza di apparizione, e sono da considerarsi perfettamente equivalenti, con il notevole vantaggio per la prima di non dare adito ad alcuna ambiguità. Anche in questo caso la scelta solitamente è esclusiva per una o l'altra forma cercando di evitare incoerenza. Altri costrutti extra-sintattici spesso presenti sono la sostituzione (aperta o chiusa) di termini, con cui è esprimibile la semantica del binding e le richieste di aggiornamento o consultazione degli ambienti.

Gli ambienti, rappresentano fondamentalmente mappe fra oggetti denotabili, come identificatori di variabile, locazioni di memoria, nomi di funzione, ecc..., e valori, la cui natura dipende dal contesto in cui il linguaggio opera. La struttura può essere semplice, nel caso in cui la mappa sia fra un singolo insieme di oggetti denotabili e valori, oppure modulare qualora ci sia necessità di specificare un numero maggiore di mappe. La disponibilità di un ambiente parametrizza inevitabilmente le regole, costituendo un compo-

nente fondamentale per contestualizzare il significato. Esempio lampante di ambiente modulare è la Modular SOS. In questo caso la scelta fra le due tipologie di ambiente non è esclusiva, in quanto una è inclusa nell'altra.

Caratteristica basilare dei sistemi transizionali è la presenza di label. Gli LTS prendono il nome proprio dalla possibilità di venire etichettati. Come abbiamo già ribadito numerose volte, i linguaggi di programmazione generalmente non prevedono la possibilità di etichettare le transizioni, in quanto non vi è alcuna necessità di trasmettere informazione in merito alla transizione stessa. Nei lingaugggi di calcolo di processi questa possibilità non solo è contemplata, ma costituisce il bastione centrale della semantica del calcolo, che spesso poggia sullo scambio di nomi e altre entità fra i processi. La scelta è quindi semantica, in un caso identifica una semplice trasformazione sintattica e di stato, nell'altro porta informazione in qualche forma. Le label possono essere composte da costanti o termini aperti, in numero singolo o in lista. La SOS modulare di P. Mosses incorpora nelle label le informazioni di ambiente per una maggiore flessibilità del formalismo. Sono un caso particolare di termini aperti a lista.

Passiamo ora ad analizzare il contenuto delle premesse delle regole. Come da definizione di LTS e TSS le premesse sono costituite da transizioni vere e proprie o da predicati. Per quanto riguarda il primo tipo prende importanza la nozione di configurazione, il termine sorgente o destinazione contenente sintassi e informazione di stato. La configurazione è sempre presente all'interno delle regole di transizione dei linguaggi di programmazione che, per quanto poveri, prevedono solitamente anche un ambiente di cui tenere traccia. Nei linguaggi di calcolo di processi la nozione di configurazione si contrae con l'aspetto meramente sintattico, non prevedendo una separazione con lo stato. Le transizioni possono essere anche negative. Questa evenienza ha avuto grande attenzione dal mondo accademico per quanto riguarda l'aspetto meta-teorico della sua inclusione, ma poca applicazione al di fuori di esso. Uguale sorte per la possibilità di esprimere un numero infinito di transizioni, che richiede un numero infinito di azioni compibili, caratteristica riservata a pochi esemplari di linguaggi impiegati nei sistemi concorrenti. I predicati sono un altro tipo di transizioni presentabili nell'insieme delle premesse di una regola. I predicati esprimono solitamente una condizione di applicabilità di una regola. e coinvolgono operatori dei domini più disparati, dal confronto di valori ai free-test di una variabile, dal test di tipo all'appartenenza ad un insieme. Questo tipo di predicati può comparire indifferentemente nella zona adibita alle premesse che a lato della regola interessata, costituendo semplicemente un'alternativa di presentazione. Diverso significato assumono invece

i predicati a livello di termine, utilizzati per esprimere la terminazione di un processo e caratteristiche simili. Sono utilizzati abbondantemente all'interno del calcolo di processi, di norma raccolti in un insieme apposito in modo da poterli identificare.

L'ultima sezione di diversificazione sintattica riguarda l'aspetto notazionale. Le regole sono nella quasi totalità dei casi espresse secondo lo schema frazionale premesse/conclusione o con lo schema in linea nel caso di assiomi. Solo pochi testi presentano una notazione alternativa, spesso poco chiara e di interesse decisamente trascurabile. Anche la freccia transizionale rappresenta un standard di fatto. La notazione più comune in questo versante è la '$\rightarrow$' per la SOS small-step e la $\Rightarrow$ per la SOS big-step (o semantica naaturale). In alcuni testi che trattano entrambe le tipologie di semantica operazionale, possiamo incorrere in uno scambio. Di simbologie differenti se ne trovano solamente rare tracce, anche in questo caso poco interessanti. I termini, nei linguaggi di programmazione, sono generalmente costituiti da una parte sintattica ed una di stato, separate da virgole e comprese all'interno di parentesi angolari. Nei linguaggi di concorrenza questa forma di isolamento dei termini non viene di norma adottata, ed esistono anche esempi in numero più che trascurabile di specifiche semantiche di linguaggi di programmazione che seguono la sstessa tendenza, spesso in concomitanza ad una notazione di contesto. La presenza del contesto è proprio una delle caratteristiche su cui esiste la maggior variabilità. Come contesto intendiamo tutta una serie di informazioni atte a fornirci la capacità di valutare i frammenti sintattici nella loro interezza. La scelta si pone proprio al momento di dover decretare se mantenere queste informazioni in un costrutto esterno alla configurazione, appunto il contesto, oppure internamente, caso in cui si parla di ambiente e sue componenti. Nel primo il contesto si posiziona ordinariamente a sinistra della configurazione, separato da essa dal simbolo $\vdash$, nel secondo caso compare separato dalla parte sintattica del termine con virgole, in forma di lista. Entrambe le notazioni trovano un numero cospicuo di esempi in letteratura. Come già discusso poco sopra la SOS modulare di P. Mosses sposta la lista di ambienti all'interno della label. Riguardando esclusivamente l'aspetto di presentazione, tutta questa varietà non ha rilevanza ai fini della espressività semantica del formalismo.

A fine di chiarezza riportiamo in Figura 1.16 uno schema riassuntivo della varietà linguistica presente all'interno della famiglia SOS. Il simbolo '□' identifica una caratteristica opzionale.

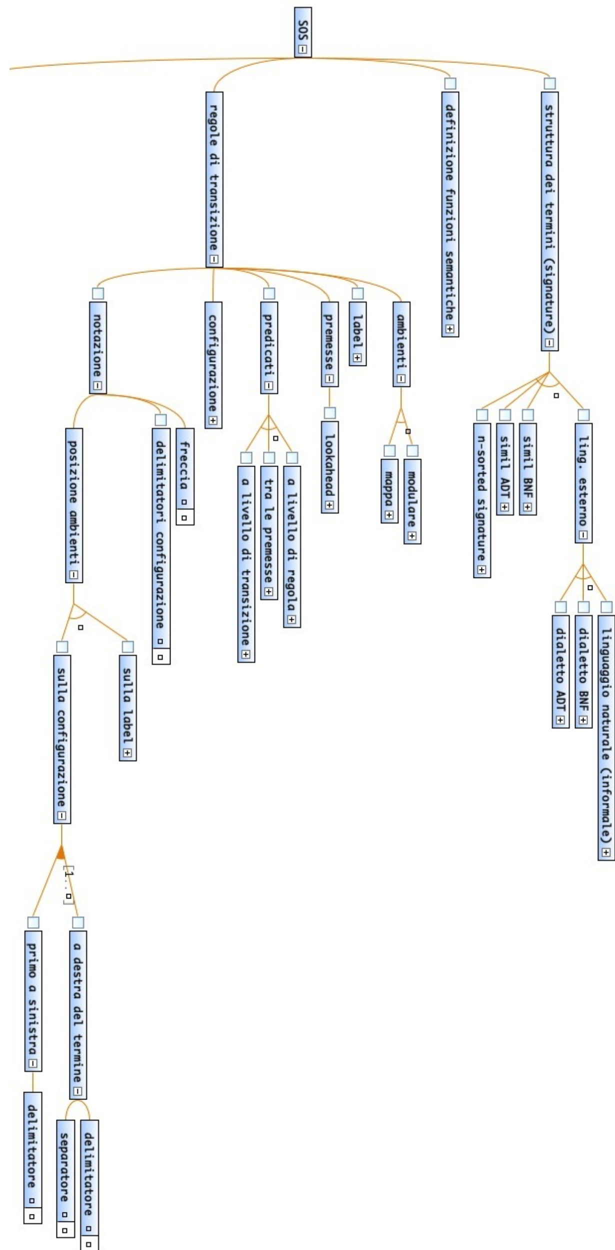

Figura 1.16: Riassunto della varietà sintattica dei linguaggi della famiglia SOS

Capitolo 2

Tool

2.1 Panoramica sui tool esistenti

Passiamo ora in rassegna una serie di strumenti software creati recentemente per permettere una agevole formulazione di sintassi operazionale per linguaggi di programmazione. Premettiamo che il funzionamento dei suddetti strumenti è lungi da raggiungere una buona affidabilità, ma risulta comunque istruttivo vedere all'opera tali implementazioni.

2.2 Il Sistema Centaur

2.2.1 La storia

Il sistema Centaur, sviluppato nella seconda metà degli anni '80 del secolo scorso, presso l'unità di ricerca dell'istituto INRIA situato all'interno del parco tecnologico di Sophia-Antipolis nel sud della Francia, ha li'importanza storica di essere il primo tentativo di implementazione e utilizzo pratico di specifiche di semantica operazionale. Centaur si presenta come un generico ambiente interattivo, esenzialmente scritto in LISP (o meglio Le-Lisp), attraverso il quale, fornite specifiche sintattiche e semantiche di un particolare linguaggio di programmazione, costruire un'ambiente di sviluppo per tale linguaggio. L'ambiente risultante include un editor, un debugger, un'interprete e altri tool, ognuno dei quali dotato di interfaccia grafica verso l'utilizzatore. I tre bastioni fondamentali per l'implementazione di Centaur sono:un componente di database, che provvede ad assicurare persistenza, un motore logico, costituito da una variazione sul tema del Prolog (Mu Prolog) e un interfaccia grafica intuitiva, poggiante sulle capacità grafiche delle librerie Le-Lisp a loro volta costruite sopra il sistema X-Windows,che equipaggia il sistema di un

facile accesso verso le sue funzioni. Il suo schema architetturale è suddiviso in tre componenti, ognuna delle quali adibita ad una particolare funzione:

- *un kernel* per la rappresentazione e la manipolazione di oggetti strutturati;
- *un livello di specifica* per il supporto degli aspetti semantici e sintattici dei linguaggi;
- *un interfaccia utente* per la gestione di tutte le comunicazioni interattive fra il sistema ed i suoi utenti.

Il *kernel* di Centaur gioca un ruolo fondamentale all'interno dell'architettura, in quanto destinato ad essere utilizzato, direttamente o indirettamente, da tutti gli altri componenti. L'obiettivo che si prefigge è fornire un supporto alla manipolazione simbolica di documenti strutturati. Con manipolazione simbolica si intende la capacità di saper esaminare analiticamente la struttura o fornire un qualche tipo di valutazione, del documento in fase di editing. Allo scopo il kernel si avvale dell'utilizzo di due componenti:

- una macchina astratta, denominata *Virtual Tree Processor (VTP)*, impiegata nella gestione degli aspetti sintattici;
- una macchina logica che cattura gli aspetti semantici, come la valutazione. A questo livello la macchina logica è un interprete Prolog.

Le due componenti sono totalmente cooperanti adottando primitive di *controllo* attraverso le quali richiamarsi a vicenda e primitive di *trasfomazione* con le quali operare la conversione fra termini del VTP in termini Prolog e viceversa.

Il *livello di specifica* ha il compito di raccogliere le specifiche per la sintassi astratta, sintassi concreta e semantica del linguaggio di programmazione di cui si intende generare lo specifico ambiente di sviluppo, ognuna delle quali scritta in un appropriato formalismo. La *specifica sintattica* contiene tutte le informazioni necessarie alla costruzione di un editor orientato alla struttura. In seguito a compilazione si ottengono uno scanner, un parser, una pretty-printer ed una tabella di sintassi astratta, utilizzati allo scopo di fornire per il passaggio ambidirezionale fra una rappresentazione testuale ed una strutturata e per effettuare un controllo di validità sulle operazioni di editing. Le specifiche di sintassi astratta e concreta sono scritte nel linguaggio METAL. Una specifica in METAL è una collezione di regole grammaticali che fanno uso di annotazioni, mediante le quali discrivere le modalità di sintesi dell'albero di sintassi astratta. Il modo in cui mostrare testualmente gli alberi

sintatti, mansione della pretty-printer, è specificato con il formalismo PPML, una serie di regole di unparsing associate a specifici pattern degli alberi.

Per derivare sistematicamente un ambiente interattivo per la programmazione è tuttavia necessario andare oltre i meri aspetti sintattici, introducendo le specifiche semantiche indispensabili al proposito di equipaggiare l'ambiente di agenti di semantica statica, come controlli di tipo o interazioni dipendenti dal contesto, oppure di semantica dinamica, come un interprete o un debugger. La semantica dei linguaggi di programmazione può essere specificata facendo uso di diversi formalismo, tra cui il più indicato è la semantica naturale di G. Kahn, in quanto con il sistema è fornita un'implementazione di quest'ultima, il linguaggio TYPOL. Le specifiche formalizzate in TYPOL vengono poi compilate, in forma eseguibile, in clausole Prolog.

Le funzioni che il sistema Centaur offre sono accessibili attraverso un'*interfaccia grafica*, costituita da comodi menù per accedere a varie opzioni, bottoni per il controllo dell'esecuzione, pseudo-terminali per la visualizzazione dell'output, ecc... All'interno dell'interfaccia sono presenti differenti viste con le quali l'utente tiene monitorata contemporaneamente la struttura ad albero del documento che sta editando e la visualizzazione testuale del medesimo. L'utente di norma si trova nella condizione di dover manipolare diversi formalismi (METAL, PPML, TYPOL) per ognuno dei quali è fornito una specifica vista con ambiente personalizzato. E' inoltre implementato un controllo di consistenza fra i contenuti delle viste che dipendono reciprocamente.

2.2.2 Il linguaggio METAL

Addentriamoci all'interno del linguaggio METAL per cercare di capire come possa costituire un buon formalismo per descrivere la sintassi astratta e concreta di una linguaggio. La definizione di una sintassi astratta di un linguaggio consiste in nodi chiamati **operatori** e tipi di nodo chiamati **phyla**. Gli operatori sono i terminali e non terminali del linguaggio e sono definiti dal tipo e dal numero dei loro discendenti. A titolo di esempio consideriamo un ipotetico linguaggio elementare `EXP`, di espressioni matematiche. La grammatica per questo linguaggio potrebbe essere formata da un unico plylum `EXP` che raccoglie tutte le modalità possibili di formazione di una valida espressione aritmetica. Nel linguaggio è compresa inoltre la possibilità di effettuare un assegnamento. La sua composizione potrebbe essere:

```
EXP ::= plus minus prod uminus assign variable integer
```

Gli operatori `plus`, `minus`, `prod` e `uminus` sono operatori binari e unari posizionati come nodi interni dell'albero di sintassi astratta aventi come nodi

discendenti altri elementi del phylum EXP. Fornita una definizione di sintassi astratta, il compilatore METAL genera un rappresentazione persistente in forma tabellare. Ogni volta che cerchiamo di costruire un albero di sintassi astratta in un dato formalismo, sia a mano che tramite parsing, le tabelle sono lette dalla memoria e utilizzate come base per la sintesi dell'albero.

Se da una parte la sintassi astratta specifica un insieme di alberi di sintassi astratta legali, la sintassi concreta serve come base di un parser che traduca un documento testuale in un albero di sintassi astratta. Le specifiche di sintassi concreta includono token e keyword del linguaggio, permettendo al parser la costruzione di un albero di sintassi astratta in modo non ambiguo. In METAL la specifica concreta è distinta da quella astratta, in quanto ad una stessa specifica astratta possono corrispondere differenti sintassi concrete. Le specifiche di sintassi concreta di METAL sono compilate in programmi Lex e Yacc, a loro volta compilati al fine di generare un parser.

Alla luce dell'introduzione fatta, la sintassi astratta del linguaggio EXP, scritta in METAL risulterebbe:

```
definition of Exp is
   abstract syntax
      exp_s -> EXP + ...;
      assign -> VAR EXP;
      plus -> EXP EXP;
      minus -> EXP EXP;
      prod -> EXP EXP;
      uminus -> EXP;
      variable -> implemented as IDENTIFIER;
      integer -> implemented as INTEGER;

EXP_S ::= exp_s;
EXP ::= plus minus prod uminus assign variable integer
VAR ::= variable;
INTEGER ::= integer; end definition
```

L'identificatore che segue `definition of` è il nome del linguaggio.

L'operatore `exp_s` è l'unico operatore del phylum radice `EXP_S` e denota una lista contenente almeno un costrutto di tipo `EXP`. Gli operatori `variable` e `integer` sono atomici e poggiano sui phylum built-in `INTEGER` e `IDENTIFIER`. L'operatore `variable` appare sia sul lato destro del phylum `VAR` che su quello del phylum `EXP`, utilizzato nel primo caso come restrittore del

dominio dei valori denotabili in un assegnamento, nel secondo come espressione a se stante. Sarà compito della sintassi concreta(che qui non trattiamo) sostituire agli operatori i consueti simboli aritmetici `+, -, *, :=`. La sintassi astratta definisce frasi valide nel linguaggio che rappresentiamo con alberi di sintassi astratta.

2.2.3 Il linguaggio TYPOL

Per andare oltre il regno sintattico e avere la possibilità di valutare veramente un albero di sintassi astratta, abbiamo bisogno di descrivere la semantica di `Exp` con un insieme di regole semantiche scritte nel linguaggio Tipol che, come accennato nel primo paragrafo del capitolo corrente, altro non è che un'implementazione della semantica naturale. In particolare Typol è implementato in Prolog, con il quale condivide una semantica simile, ma è stato scelto rispetto a quest'ultimo con le seguenti motivazioni:

- TYPOL offre una modalità generale di espressione di regole semantiche, indipendentemente dal linguaggio in cui è compilato (Prolog);
- TYPOL offre meccanismi che permettono all'utente di tenere traccia dei programmi durante la valutazione;
- TYPOL offre controlli automatici di tipo sulle variabili istanziate, basati sulla definizione del linguaggio di cui si intende esprimere la semantica.

Il linguaggio TYPOL ruota attorno a provare la validità di determinate proposizioni in un certo contesto di valutazione. La sintassi ha la forma

$$\texttt{context} \vdash \texttt{expression : value}$$

dove ':' esprime la validità della proposizione. In questo caso la semantica espressa dal costrutto verrebbe espressa con il linguaggio naturale nel seguente modo:'Nel contesto `context` l'espressione `expression` hs il valore `value`'. La nozione di contesto comprende non solo gli aspetti di ambiente, ma tutta l'informazione che può condizionare la valutazione. All'inizio di ogni definizione scritta in TYPOL, vengono elencate le regole di valutazione generali che verranno utilizzate in tutto il programma. Il costrutto *judgement* è adibito a tal compito e nel caso del linguaggio elementare `EXP` può comprendere le seguenti dichiarazioni iniziali:

```
judgement |- EXP : INTEGER
judgement |- EXP_S : INTEGER
```

Queste indicazioni segnalano la necessità di valutare una qualunque espressione o serie di espressioni nel risultato finale di un numero intero. La necessità nasce dal fatto che siamo in ambito di semantica naturale, quindi i passi intermedi di computazione rimangono nascosti.

A seguito di queste indicazioni generali, troviamo le regole di transizione, che possono presentare come di consueto la forma di assiomi o regole con premesse e conseguenza. La sintesi delle regole si avvale naturalmente di costrutti esterni di tipo built-in, che implementano le classiche funzioni semantiche aritmetiche e di ambiente. A titolo chiarificatore riportiamo in Figura 2.1 le regole di valutazione utilizzate all'interno del linguaggio Exp per i costrutti aritmetici elementari:

```
|- integer N : integer N ;

|- EXP1 : integer x1  &
|- EXP2 : integer x2  &
add(x1, x2, x)
----------------
|- plus(EXP1, EXP2) : integer x ;

|- EXP1 : integer x1  &
|- EXP2 : integer x2  &
sub(x1, x2, x)
----------------
|- minus(EXP1, EXP2) : integer x ;

|- EXP1 : integer x1  &
|- EXP2 : integer x2  &
mult(x1, x2, x)
----------------
|- prod(EXP1, EXP2) : integer x ;

|- EXP  : integer x1  &
neg(x1, x)
----------------
|- uminus(EXP) : integer x ;

|- EXP1 : v
----------------
|- exp_s[EXP1] : v ;

|- EXP1 : v1  &
|- EXPS : v
----------------
|- exp_s[EXP1.EXPS] : v ;
```

Figura 2.1: Regole di transizione per gli operatori aritmetici in TYPOL

I simboli '[...]' delineano liste, mentre l'operatore '.' concatena elementi. Le ultime due regole dell'elenco interpretano la sequenza base di espressioni,

composta da una sola di esse, e la ricorsione sulle sequenze con un numero maggiore di espressioni. L'utilizzo di costrutti esterni avviene a livello delle premesse della regole.

In caso di utilizzo di ambienti nel linguaggio, è preferibile modificare la sintassi astratta introducendo le voci:

```
env -> PAIR * ... ;
pair -> VAR INTEGER ;

ENV ::= env ;
PAIR ::= pair ;
```

con cui indico di costruire l'ambiente come una lista arbitraria di coppie variabile-intero. Le regole di transizione aggiuntive, atte a completare la definizione semantica sono riportate in Figura 2.2

```
env |- EXP : v, env' &
UPDATE(env' |- variable V, v : env'')
----------------
env |- assign(variable V, EXP) : v, env'' ;

GETVALUE(env |- variable V : v, env')
----------------
env |- variable V : v, env' ;
```

Figura 2.2: Regole semantiche per l'ambiente

Le funzioni `UPDATE` e `GETVALUE` sono funzioni ausiliarie per l'ambiente utilizzate per l'aggiornamento e l'ottenimento di un valore collegato ad un identificatore.

2.2.4 Riepilogo

Inseriamo in tabella Tabella 2.1, a fini riepilogativi, un sunto delle componenti sintattiche presenti nel sistema Centaur, in relazione a quanto esposto nella sezione riguardante la varietà sintattica, nel Capitolo 1 del presente eleborato.

Semantica Operazionale Strutturale nel sistema Centaur		
FEATURE	PRESENZA	FORMA
Signature	SI	Linguaggio interno METAL simil BNF
Funzioni Semantiche	SI	Definizione built-in e definizione personalizzata
Configurazioni	SI	Sintassi + Stato
Ambienti	SI	Ambiente Modulare o Semplice (dipende dalla definizione)
Label	NO	
Premesse Negative	NO	
Premesse infinite	NO	
Lookahead	SI	
Predicati	SI	Tra le premesse
Notazione Configurazione	NO	
Notazione Ambienti	SI	A livello configurazione a sinistra del termine
Notazione Freccia	SI	:

Tabella 2.1: Riepilogo feature sintattiche presenti nel sistema Centaur

2.3 Rml e tools

2.3.1 Il linguaggio rml

Rml è un meta-linguaggio sviluppato originariamente da Mikael Petterson, che si pone l'obiettivo di consentire la generazione automatica di interpreti o compilatori per un determinato linguaggio a partire dalla sua specifica di semantica operazionale (in questo caso viene utilizzato il formalismo di semantica naturale, che differisce dalla SOS di Plotkin nel considerare come passi di computazione transizioni che portano direttamente ad uno stato, senza esplicitare i passi intermedi).

Il compilatore rml si pone idealmente in sequenza ad un altra serie di generatori, come Lex per l'analisi lessicale o Yacc per l'analisi sintattica (processo

di parsing), e presi in input una descrizione della semantica naturale del linguaggio per cui dovrà generare il compilatore, scritta in RML, e un albero di sintassi astratta elaborato in seguito ad analisi sintattica di un determinato programma scritto in quel linguaggio , produce la forma intermedia del programma (ottimizzabile) dalla facile traduzione in linguaggio macchina. Allo stesso modo è possibile costruire un interprete.

2.3.2 Sintassi RML

La sintassi di RML di articola in due macro categorie:la prima riguarda la definizione delle categorie della sintassi astratta del linguaggio, la seconda la sintesi delle regole semantiche. Formalizzare la sintassi astratta del linguaggio è operazione fondamentale per la successiva stesura delle regole di semantica operazionale.

Come semplice esempio ci baseremo su un linguaggio elementare che contempla esclusivamente espressioni aritmetiche. La descrizione della sintassi astratta avviene in modo analogo a quella dell'algebra lineare e agli abstract data-type, poggiando su moduli che descrivono sia le forme assumibili dalle diverse categorie sintattiche, sia le regole di valutazione. La definizione di categorie sintattiche segue lo schema seguente:

```
datatype Exp = INTconst of int
             | ADDop of Exp * Exp
             | SUBop of Exp * Exp
             | MULop of Exp * Exp
             | DIVop of Exp * Exp
             | NEGop of Exp
```

La dicitura datatype sta a indicare che la categoria sintattica può assumere svariate forme (in questo caso può essere una espressione composta da addizioni, sottrazioni, ecc...) ognuna delle quali espressa (anche ricorsivamente) come prodotto cartesiano di altre categorie sintattiche. RML comprende diverse categorie sintattiche di base quali *int, bool, real* o tipi parametrici quali *list* e *vector* insieme ad una serie di funzioni che manipolano questi tipi primitivi come int_add, int_sub o $list_head$. Qualora la categoria sintattica preveda una sola forma avremo il costrutto type seguito dal prodotto cartesiano dei

tipi che lo compongono.

In seguito alla dichiarazione delle categorie sintattiche astratte definiamo le regole di valutazione tramite il formalismo della semantica operazionale declinato secondo RML. Il costrutto relation definisce entità che nei capitoli precedenti abbiamo chiamato funzioni semantiche, che fanno da tramite fra il dominio sintattico (astratto) e quello semantico e prevede due tipi di sottovoci: axiom e rule. Il costrutto axiom identifica regole senza premesse e solitamente si utilizza per mappare i costrutti elementari come booleani e interi. Per le forme non elementari usiamo invece il costrutto rule in cui sono presenti premesse (anche negative) e conclusione, con la possibilità di aggiuingere condizioni di regola.

$$\textbf{relation}\ eval : Exp \Rightarrow int =$$

$$\textbf{axiom}\ eval(\texttt{INTconst}(ival)) \Rightarrow ival$$

$$\textbf{rule}\ \frac{eval(e1) \Rightarrow v1 \quad eval(e2) \Rightarrow v2 \quad int_add(v1, v2) \Rightarrow v3}{eval(\texttt{ADDop}(e1, e2)) \Rightarrow v3}$$

$$\textbf{rule}\ \frac{eval(e1) \Rightarrow v1 \quad eval(e2) \Rightarrow v2 \quad int_sub(v1, v2) \Rightarrow v3}{eval(\texttt{SUBop}(e1, e2)) \Rightarrow v3}$$

Nel caso il nostro linguaggio venga arricchito con la possibilità di fare uso di variabili, abbiamo la necessità di appoggiarci ad una struttura di ambiente (*Env*), che contenga le coppie (Identificatore, Valore), tramite cui poter accedere o aggiornare il valore associato ad un determinato identificatore.

A questo scopo costruiamo una funzione semantica *lookup* che permetta di interrogare l'ambiente e ritornare una coppia valore-ambiente. La definizione dell'ambiente si dipende dalla struttura dati scelta per rappresentarlo; in questo caso faremo uso della struttura lista (primitiva in RML) e dei suoi operatori di concatenazione(::) e separazione della lista restante dalla sua testa(rest).

In prima istanza viene cercato un pattern matching fra l'identificatore parametro di *lookup* e quello presente in testa alla lista. Se questo fallisce si richiama ricorsivamente *lookup* sul resto della lista. Qualora anche questo

tentativo non andasse a buon fine l'identificatore viene aggiunto in testa alla lista con il valore di default 0. Il pattern _ fa match con qualunque costrutto sintattico. Le regole di *lookup* fanno uso di premesse negative.

$$\textbf{relation } lookup : (Env, Ident) \Rightarrow (Env, Value) =$$

$$\textbf{rule } \frac{id = id2}{lookup((id2, value) :: rest, id) \Rightarrow ((id2, value) :: rest, value)}$$

$$\textbf{rule } \frac{\textbf{not } id = id2 \quad lookup(rest, id) \Rightarrow (rest, value)}{lookup((id2, _) :: rest, id) \Rightarrow ((id2, _) :: rest, value)}$$

$$\textbf{rule } \frac{\textbf{not } lookup(env, id) \quad (id, 0) :: env \Rightarrow env'}{lookup(env, id) \Rightarrow (env', 0)}$$

2.3.3 Structural Operational Semantics Development Tooling - Eclipse Plugin

Andiamo ora a presentare un ambiente di sviluppo integrato chiamato *Structural Operational Semantics Development Tooling* (SOSDT) progettato da Adrian Pop e Peter Fritzon. SOSDT è basato sul systema RML e fornisce una semplice ed intuitiva interfaccia grafica verso di esso vivendo all'interno dell'IDE Eclipse come plug-in.

SOSDT consta di tre principlai elementi: l'editor RML, il browser RML e i componenti di debug di RML. All'interno dell'ambiente di SOSDT l'utente è in grado di gestire e creare progetti e file RML attraverso un procedimento guidato. L'editor RML provvede evidenziazione della sintassi, auto indentazione, suggerimenti al completamento del codice, informazioni sui tipi e sugli errori. Il browser RML fornisce una facile navigazione all'interno dei file e si poggia alle funzionalità del parser RML per organizzare le informazioni per ciascun file sui tipi, valori, relazioni, regole e mostrarle ad albero. Il componente di debug comunica via socket con il framework di debug di RML al fine di equipaggiare SOSDT di funzionalità quali breakpoint, ispezione passo-passo, valori di variabili, ecc... (Figura 2.3).

Tutti i componenti di SOSDT fanno utilizzo dei prospetti di visualizzazione di cui dispone l'ambiente integrato Eclipse, popolati con informazioni

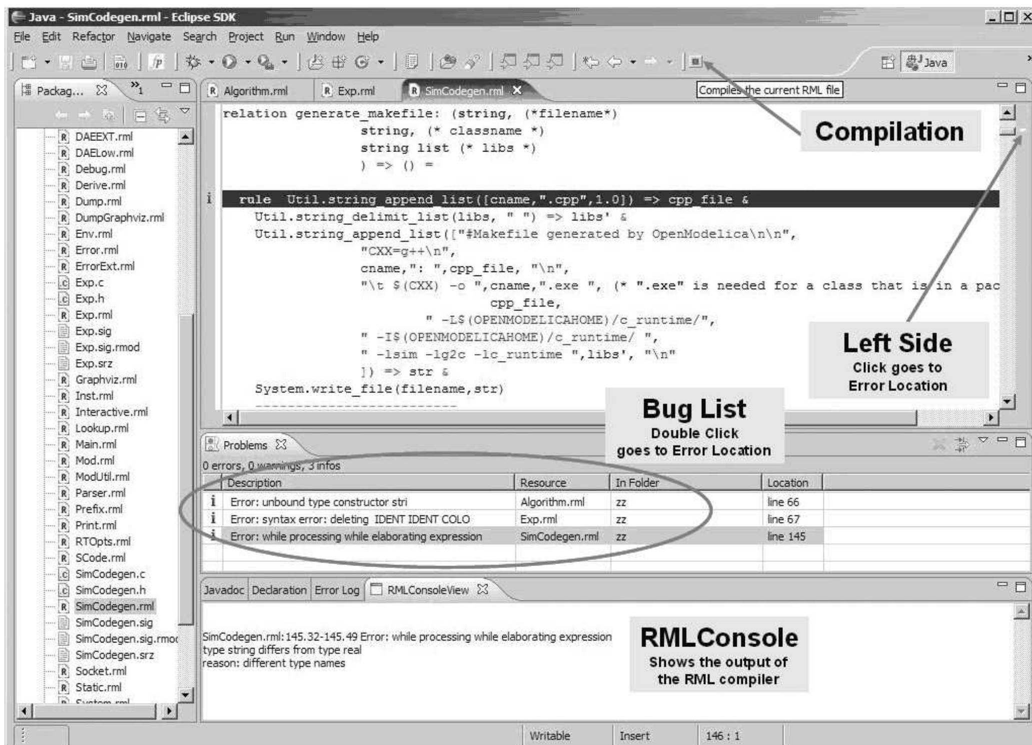

Figura 2.3: Uno screenshot del tool SOSDT

provenienti dal parser e dal compilatore RML. Ad ogni salvataggio il parser legge il file e aggiorna le informazioni visualizzate dal browser RML. Il compilatore, sempre al momento del salvataggio, esegue un controllo ed invia lo stato degli errori alla finestra Problems View (Figura 2.3).

2.3.4 Riepilogo

Inseriamo in tabella Tabella 2.2, a fini riepilogativi, un sunto delle componenti sintattiche presenti nel sistema RML, in relazione a quanto esposto nella sezione riguardante la varietà sintattica, nel Capitolo 1 del presente eleborato.

2.4 Maude e tools

2.4.1 Il sistema Maude

Il sistema Maude è un'implementazione della logica di riscrittura sviluppata allo SRI International. Il linguaggio utilizzato è in grado di formalizzare

Semantica Operazionale Strutturale nel sistema RML		
FEATURE	PRESENZA	FORMA
Signature	SI	Linguaggio interno simil Abstract Data Type
Funzioni Semantiche	SI	Definizione built-in e definizione personalizzata
Configurazioni	SI	Sintassi + Stato
Ambienti	SI	Ambiente Modulare o Semplice (dipende dalla definizione)
Label	NO	
Premesse Negative	SI	uso di **bf** davanti a transizioni
Premesse infinite	NO	
Lookahead	SI	
Predicati	SI	Tra le premesse
Notazione Configurazione	SI	(S, S)
Notazione Ambienti	SI	A livello configurazione a destra del termine
Notazione Freccia	SI	$\Rightarrow$

Tabella 2.2: Riepilogo feature sintattiche presenti nel sistema RML

specifiche e scrivere programmi di logica di riscrittura.

La logica di ricrittura è una logica utilizzata nella computazione concorrente e possiede buone proprietà di framework per la semantica operazionale di linguaggi e modelli concorrenti. Maude fa uso sistematico di riflessione, caratteristica che lo rende particolarmente estensibile e performante, permettendo operazioni di composizione di moduli e applicazioni di metaprogrammazione e metalinguaggi.

2.4.2 Sintassi Maude

Le funzionalità di specifica di Maude sono articolate in moduli, i quali consistono in insiemi di termini, equazioni e regole di riscrittura. I moduli che

comprendono regole equazionali sono chiamati moduli funzionali e sono dichiarati con la sintassi `fmod`, mentre quelli che prevedono al loro interno regole di reiscrittura sono moduli di sistema introdotti con `fmod`. I termini sono costruiti tramite funzioni, che prendono in input 0 o più argomenti di una certa sorta e danno in output un termine di una specifica sorta. Le funzioni con arietà 0 sono considerate constanti, i termini elementari del linguaggio. La sintassi ricorda quella dei moduli della semantica algebrica. Riscriviamo a proposito il modulo sui Booleani introdotto nella Sezione 1.3.6 utilizzando la sintassi di Maude:

```
fmod BOOL is
sort Bool .
op true :  -> Bool [ctor] .
op false :  -> Bool [ctor] .
op _and_ :  Bool Bool -> Bool [ctor] .
op not_ :  Bool -> Bool [ctor] .
op _or_ :  Bool Bool -> Bool .
op _impl_ :  Bool Bool -> Bool .
op _xor_ :  Bool Bool -> Bool .
vars B B1 B2 :  Bool .
eq true and B : B .
eq false and false :  false .
eq false and true :  false .
eq not true :  false .
eq not false :  true .
eq B1 or B2 :  not ((not B1) and (not B2)) .
eq B1 impl B2 :  (not B1) or B2 .
eq B1 xor B2 :  (B1 or B2) and (not (B1 and B2)) .
endfm
```

In esecuzione una semplice espressione booleana della forma (((not false) and true) or (true xor (not false))) viene valutata:

```
Maude> reduce in BOOL : (((not false) and true) or (true
xor (not false))) .
reduce in BOOL : true and not false or true xor not false
.
rewrites:  8 in 0ms cpu (0ms real) (2000000 rewrites/second)
result Bool:  true
```

Poniamo ora l'attenzione sui moduli che comprendono regole di riscrittura. Come brevemente accennato in precedenza la sintassi di introduzione di tali moduli è `mod MOD-NAME is ... endm`. La definizione dei moduli di sistema differisce da quella dei moduli funzionali per via della sostituzione delle equazioni (precedute da `eq : ...`) con le regole di riscrittura formalizzate con la sintassi `rl [rule-name] : t => t'` dove t e t' sono termini del linguaggio definito. La parola chiave `[ctor]` sottolinea la presenza di un costruttore mentre `[assoc comm]` attribuisce le proprietà associativa e commutativa all'operazione in questione. Come esempio prendiamo un semplice ciclo delle sigarette. Il sistema in questione verrà modellato nel seguente modo:

```
mod CIGARETTES is
sort State .
op c :  -> State [ctor] .  *** cigarette
op b :  -> State [ctor] .  *** butt
op __ :  State State -> State [ctor assoc comm] .
rl [smoke] :  c => b .
rl[makenew] :  b b b b => c .
endm
```

La riscrittura della stringa c c c c c c c c c c c c c c c c c darà luogo al seguente risultato:

```
Maude> rewrite in CIGARETTES : c c c c c c c c c c c c
c c c c c .
rewrite in CIGARETTES : c c c c c c c c c c c c c c c c
c .
rewrites:  27 in 0ms cpu (0ms real) (1687500 rewrites/second)
result State:  b b
```

Utilizzando Maude in modalità avanzata (*Full Maude*) possiamo utilizzare anche moduli Object-Oriented in cui gli oggetti sono istanze di specifiche classi. I moduli Object-Oriented sono specificati con `mod MODULE-NAME is ... endom`. Le classi sono dichiarate in Maude con un nome e una serie di attributi che la caratterizzano. Gli attributi sono dichiarati con nome e sorta e sono tra loro separati da virgole. Vediamo un esempio:

```
class TABLE | occupied :  Bool , chairs :  Nat .
```

Le variabili che denotano idenficatori di oggetti sono di sorta `Oid`. Un istanza della classe `TABLE` risulterebbe:

```
< A : TABLE | occupied :  O , chairs :  N >
```

Dove `A` è di tipo `Oid`, mentre le variabili`O` e `N` sono rispettivamente di tipo `Bool` e `Nat`.

Le classi possono avere sottoclassi definibili mediante la sintassi

```
subclass OutDoorTABLE < TABLE .
```

Ogni attributo della classe viene ereditato dalle sue sottoclassi. A differenza del Maude classico in cui la parte portante delle specifiche era fornito da equazioni o da regole di riscrittura, in Full Maude le operation sono definite da messaggi dichiarati con la parola chiave `msg` seguita dal nome del messaggio e dal pattern ricorrente : `| Oid* -> Msg`.

Le regole di riscrittura vengono definite analizzando gli attributi dell'oggetto (o degli oggetti) forniti come parametro del messaggio e modificandoli in uscita. Una regola di riscrittura si pone quindi come un trasformatore di uno o più oggetti coinvolti all'interno di un messaggio. Forniamo di seguito un semplice esempio chiarificatore, di un modulo `RESTAURANT` in cui gli oggetti della classe `TABLE` vengono trasformati dalle regole di riscrittura dopo essere stati forniti in ingresso ai messaggi. Il costrutto `protecting` consente di includere moduli in modalità read-only.

```
(omod RESTAURANT is
protecting NAT .
class TABLE | occupied :  Bool , chairs :  Nat .
class OutDoorTABLE | next2heater :  Bool .
subclass OutDoorTABLE < TABLE .
msgs sit@ heaterswitch :  Oid -> Msg .
msgs _borrowchairfrom_ changetables :  Oid Oid -> Msg .

vars A B : Oid .  vars M N : Nat .

rl [sit] :  sit@(A) < A : TABLE | occupied :  false >
=> < A : TABLE | occupied :  true >

rl [switchon] :  heaterswitch(A) < A : TABLE | Next2heater
:  false >
=> < A : TABLE | Next2heater :  true > .

rl [switchoff] :  heaterswitch(A) < A : TABLE | next2heater
:  true >
=> < A : TABLE | next2heater :  false > .
```

```
rl [move] : changetables(A, B) < A : TABLE | occupied
: true >
< B : TABLE | occupied : false >
=> <A:TABLE|occupied:false>
< B : TABLE | occupied : true > .

rl [takechair] : (A borrowchairfrom B) < A : TABLE | chairs
: M >
< B : TABLE | chairs : s N >
=> <A:TABLE|chairs:sM> < B : TABLE | chairs : N > .
endom)
```

2.4.3 Maude MSOS Tool

Maude MSOS Tool(MMT) è un ambiente di programmazione per la specifica di regole MSOS, implementato come estensione di Maude ed operante come un compilatore da specifiche MSOS a logica di riscrittura. L'opera di mappatura fra i due framework avviene partendo dalla considerazione che all'interno delle specifiche MSOS le label descrivono cambiamenti da uno stato sorgente ad uno stato destinazione, e quindi possono essere riportate all'interno delle configurazioni della transizione.

Una transizione $P \xrightarrow{u} P'$ è modellata come un passo di riscrittura . $< P, u^{pre} > \to < P', u^{post} >$ dove u^{pre} e u^{post} sono *record* descriventi lo stato prima e dopo la transizione. L'uso dell'operatore '.' è d'obbligo per emulare il passo singolo, evitando riscritture transitive che snaturerebbero la traduzione da MSOS.

I campi dei *record* di stato possono essere specificati sequenzialmente all'interno di parentesi _ usando un'operazione commutativa ed associativa '_,_'. I campi vengono copiati dagli attributi degli stati, all'interno della label originaria: ad esempio lo store può essere inserito specificandolo come 'σ : _'.

I *record* u^{pre} e u^{post} sono costruiti dalla label u con le seguenti modalità:

- **Campi read-only:** sono aggiunti identici sia a u^{pre} che a u^{post} in quanto non modificati;
- **Campi read-write:** sono aggiunti differenziati nei due record. $u.i$ è aggiunto a u^{pre}, mentre $u.i'$ è aggiunto a u^{post};
- **Campi write-only:** presenti solo nella forma $u.i'$ sono tradotti con una nuova variabile L in u^{pre} e con Lv in u^{post};

- **Campi trascurabili:** rimpiazzati da una generica variabile W.

I moduli rappresentano il cuore dell'organizzazione delle specifiche MSOS in MMT. Sono specificati con la sintassi

```
msos MODULE-NAME is ...  sosm
```

All'interno della definizione del modulo trovano posto le inclusioni di altri moduli con cui esiste una relazione di dipendenza, introdotti da

```
see MODULE-REQUIRED-NAME .
```

Segue la definizione della sintassi astratta del linguaggio di cui intendiamo fornire semantica, che al pari di Maude consiste fondamentalmente in una definizione di abstract data-type. In prima istanza prendono posto le dichiarazioni delle categorie sintattiche con la sintassi

```
CATEGORY .
```

che vengono poi specificate in costrutti simil-BNF nella conformazione

```
CATEGORY ::= ...
```

dove il simbolo'`::=`' sta a significare inclusione di sottinsiemi nel caso in cui il membro destro sia a sua volta una categoria sintattica o un tipo built-in, oppure è un costruttore nel caso risulti come sequenza di terminali e categorie sintattiche (ad esempio `Exp ::= if Exp then Exp else Exp .`). Quest'ultimo tipo di costrutto può prensentare attributi, mediante i quali esplicitare proprietà e ordine di precedenza (con un intero in cui i valori minori identificano priorità maggiori) allo scopo di disambiguare la costruzione dell'albero sintattico e costruire facilmente espressioni aritmetiche.

Insieme alle categorie sintattiche trovano spazio anche le dichiarazioni di elementi semantici come l'ambiente e lo store, indicati con la specifica

```
SEMANTIC-COMPONENT = ...
```

in cui il lato destro è solitamente un tipo built-in parametrico come *Map* o *List*.

Ogni modulo MMT contiene anche le dichiarazioni delle label nel numero massimo di una. Una label consiste in una sequenza di dichiarazioni di campi nella forma `index : CATEGORY`. Gli indici (identificatori minuscoli) possono essere nella forma *index′* a seconda se il componente sia read-only, read-write o write-only. La label assume in definitiva la forma:

`Label = { env : Env, st : Store, st' : Store, ... }`

Le transizioni specificate in MMT assumono differenti connotazioni. Le transizioni incondizionate stabiliscono come possibili relazioni fra termini semantica big-step('`==>`') e small-step('`-->`'). Entrambe le relazioni sono ternarie e sono composte dal frammento sintattico da 'matchare seguito dalla categoria sintattica a cui appartiene, da una label racchiusa fra parentesi graffe e da un frammento sintattico risultante. All'interno delle transizioni è consentito l'uso di variabili con apici o pedici(ad esempio `Exp1` o `Exp'` spaziano nella categoria sintattica `Exp`).

Le transizioni condizionali presentano quattro tipi diversi di condizioni: condizioni di uguaglianza, predicati, istanziazione di variabili e transizioni condizionali vere e proprie. Le condizioni di uguaglianza asseriscono l'uguaglianza di due termini, come '`first (Pids') = Int`'. Un singolo termine '`P`' è utilizzato come predicato per abbreviare condizioni di uguaglianza '`P = true`'. Nuove meta-variabili libere sono istanziate con la formula '`CATEGORY := ...`', come in '`Value := lookup(Id, Env)`'. Infine una transizione condizionale ha la medesima sintassi di una transizione incondizionata, con l'eccezione che il membro sinistro che si cerca di 'matchare', deve anche soddisfare una serie di premesse necessarie per la realizzazione della transizione.

Prendiamo come esempio un construtto condizionale della sintassi astratta di un linguaggio a cui vogliamo assegnare la classica semantica dell'`if-then-else`. Il termine '`(cond Exp Exp1 Exp2) : Exp`' è di tipo `Exp` e la meta-varibile `Exp` successiva a `cond` deve essere valutata come espressione booleana per decidere quale ramo fra `Exp1` o `Exp2` prendere. La semantica in stile small-step è specificata dalle regole seguenti:

```
              Exp -{...}-> Exp'
    -- -------------------------------------------
  (cond Exp Exp1 Exp2) :  Exp -{...}-> (cond Exp' Exp1 Exp2)
```

```
(cond tt Exp1 Exp2) :  Exp --> Exp1
 (cond ff Exp1 Exp2) :  Exp --> Exp2
```

L'interfaccia utente di MMT è identica a quella classica di Maude, con i moduli caricabili da prompt o da un file di specifiche (scelta decisamente migliore) con il comando `load` e le richieste di riduzione o riscrittura lanciate con `rewrite` o `reduce`. L'unico comando nuovo introdotto da MMT è `set flag` che permette di impostare la modalità di esecuzione passo-passo.

Diamo ora un semplice esempio di un linguaggio funzionale con ambiente, riprendendo tutte le regole sintattiche esplicate in precedenza.

```
msos SIMPLE-LANGUAGE is
***Definizione categorie sintattiche
Id .
Exp .
Exp ::= Int | Id .  ***Simbolo di inclusione insiemistica
Exp ::= let Id = Int in Exp end | Exp sum Exp .  ***Simbolo
di costruzione
Env = (Id, Int)Map .
Label = { env :  Env, ...  } .
***Regole di transizione


            Exp1 -{...}-> Exp1'
 -- --------------------------------------------
  (Exp1 sum Exp2) :  Exp -{...}-> (Exp1' sum Exp2)

            Exp2 -{...}-> Exp2'
 -- --------------------------------------------
   (Int sum Exp2) :  Exp -{...}-> (Int sum Exp2')


            Int3 := Int1 + Int2
 -- --------------------------------------------
```

```
(Int1 sum Int2) :  Exp --> Int3
```

Per la definizione della semantica di `let-in-end` ci avvaliamo delle funzioni built-in del tipo *Map* `_->_` che costruisce un binding fra il primo e il secondo parametro, `_/_` che va a sovrascrivere i binding del primo parametro nel secondo e `lookup(_,_)` che ottiene il valore legato ad un identificatore fornito come primo parametro da una mappa fornita come secondo.

```
Env' := (Id |-> Int) / Env, Exp -{env = Env', ...}->
                Exp'
-- ----------------------------------------------------
(let Id = Int in Exp end) :  Exp -{env = Env, ...}->
        (let Id = Int in Exp' end)


(let Id = Int in Int' end) :  Exp --> Int' .


         Int := lookup (Id, Env)
-- ----------------------------------------------------
     Id :  Exp -{env = Env , ...}-> Int .


sosm
```

Proviamo ora un'istanza di un programma che fa uso delle nostre specifiche, completandole con gli dentificatori in gioco:

```
(msos TEST is
see SIMPLE-LANGUAGE .

Id ::= x | y .
sosm)
```

Lanciamo ora il comando `rewrite` per eseguire un semplice programma:

```
(rewrite < (let x = 10 in
let y = 10 in
```

```
x sum y
end
end), {env = void} ) .

rewrite in TEST : < ...  >
result Conf :
< 20, {env = void}

Bye
```

2.4.4 Riepilogo

Inseriamo in tabella Tabella 2.3, a fini riepilogativi, un sunto delle componenti sintattiche presenti nel tool MSOS, in relazione a quanto esposto nella sezione riguardante la varietà sintattica, nel Capitolo 1 del presente eleborato.

Semantica Operazionale Strutturale nel tool MSOS		
FEATURE	PRESENZA	FORMA
Signature	SI	Linguaggio interno simil Abstract Data Type
Funzioni Semantiche	SI	Definizione built-in e definizione personalizzata
Configurazioni	SI	Sintassi
Ambienti	SI	Ambiente Modulare
Label	SI	Informazioni di stato
Premesse Negative	NO	
Premesse infinite	NO	
Lookahead	SI	
Predicati	SI	Tra le premesse
Notazione Configurazione	NO	
Notazione Ambienti	SI	A livello di label
Notazione Freccia	SI	`-{...}->`

Tabella 2.3: Riepilogo feature sintattiche presenti nel tool MSOS

Bibliografia

[AFV99] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In *Handbook of Process Algebra*, pages 197–292. Elsevier, 1999.

[BCIK87] Patrick Borras, Dominique Clement, Janet Incerpi, and Gilles Kahn. Centaur : the system. INRIA Research Report research report number 777, INRIA, December 1987.

[BG87] Jos C. M. Baeten and Rob J. van Glabbeek. Merge and termination in process algebra. In *Proceedings of the Seventh Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 153–172, London, UK, 1987. Springer-Verlag.

[BG96] Roland Bol and Jan Friso Groote. The meaning of negative premises in transition system specifications. *J. ACM*, 43:863–914, September 1996.

[BHMM03] Christiano de O. Braga, E. Hermann Haeusler, José Meseguer, and Peter D. Mosses. Mapping modular sos to rewriting logic. In *Proceedings of the 12th international conference on Logic based program synthesis and transformation*, LOPSTR'02, pages 262–277, Berlin, Heidelberg, 2003. Springer-Verlag.

[CB05] Fabricio Chalub and Christiano Braga. An implementation of modular sos in maude. In *Master's thesis, Universidade Federal Fluminense*, 2005.

[cRM07] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics (extended abstract). *Electron. Notes Theor. Comput. Sci.*, 192:125–141, October 2007.

[FV98] Wan Fokkink and Chris Verhoef. A conservative look at operational semantics with variable binding. *INFORMATION AND COMPUTATION*, 146:146–24, 1998.

[Gal03] Vashti Galpin. A format for semantic equivalence comparison. *Theor. Comput. Sci.*, 309:65–109, December 2003.

[Gla95] R. J. Glabbeek. The meaning of negative premises in transition system specifications ii. Technical report, Stanford, CA, USA, 1995.

[GMR06] Jan Friso Groote, Mohammad Reza Mousavi, and Michel A. Reniers. A hierarchy of sos rule formats. *Electron. Notes Theor. Comput. Sci.*, 156:3–25, May 2006.

[Gro90] Jan Friso Groote. Transition system specifications with negative premises (extended abstract). In *Proceedings of the Theories of Concurrency: Unification and Extension*, CONCUR '90, pages 332–341, London, UK, 1990. Springer-Verlag.

[HSBJS10] Reiner Hahlne, Martin Steffen, Einar Broch-Johnsen, and Jan Schafer. Hats: Deliverable d1.1a report on the core abs language and methodology: Part a. HATS 7th framework programme deliverable D1.1A rev 2.1, UIO, April 2010.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, pages 22–39, London, UK, 1987. Springer-Verlag.

[Mcc62] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[MG06] Simone Martini and Maurizio Gabbrielli. *Linguaggi di programmazione - Principi e paradigmi*. McGraw-Hill, Italia, 1st edition, 2006.

[MGR05] MohammadReza Mousavi, Murdoch J. Gabbay, and Michel A. Reniers. *SOS for higher order processes*, pages 308–322. Springer-Verlag, London, UK, 2005.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil99] Robin Milner. *Communicating and mobile systems: the &pgr;-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[Mos99] Peter D. Mosses. Foundations of modular sos. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, MFCS '99, pages 70–80, London, UK, 1999. Springer-Verlag.

[Mos03] Peter D. Mosses. Modular sos: Differences from sos, 2003.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195 – 228, 2004. Structural Operational Semantics.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100:1–40, September 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100:41–77, September 1992.

[MR05] Mohammadreza Mousavi and Michel Reniers. Congruence for structural congruences. In *In Proceedings of the 8th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2005, volume 3441 of LNCS*, pages 47–62. Springer, 2005.

[MRG05] Mohammad Reza Mousavi, Michel A. Reniers, and Jan Friso Groote. Notions of bisimulation and congruence formats for sos with data. *Inf. Comput.*, 200:107–147, July 2005.

[NM03] Tobias Nipkow and Technische Universität München. Jinja: Towards a comprehensive formal semantics for a java-like language. In *In Proceedings of the Marktoberdorf Summer School. NATO Science Series.* Press, 2003.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[NS94] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: theory and application. *Inf. Comput.*, 114:131–178, October 1994.

[PF07] Adrian Pop and Peter Fritzson. An eclipse-based integrated environment for developing executable structural operational semantics specifications. *Electron. Notes Theor. Comput. Sci.*, 175:71–75, May 2007.

[Plo81] G. D. Plotkin. A structural approach to operational semantics, 1981.

[RJ92] Laurence Rideau and Ian Jacobs. A centaur tutorial. INRIA Technical Report technical report number 141, INRIA, August 1992.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Tho95] Bent Thomsen. A theory of higher order communicating systems. *Inf. Comput.*, 116:38–57, January 1995.