ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in Informatica

Kairos
un linguaggio reversibile e concorrente

Relatore:
Chiar.mo Prof.
Ivan Lanese

Presentata da:
Nicolò Giuliani

Sessione [sessione] [anno]
Anno Accademico [20xx/20xx]

[dedica]

Abstract

Kairos è un linguaggio di programmazione *reversibile* e *concorrente*, ispirato a Janus[YG07] ed esteso per esprimere problemi concorrenti. Ogni programma Kairos può essere eseguito sia in avanti sia all'indietro: l'inverso di qualunque computazione è sempre ben definito e calcolabile. Il linguaggio offre parallelismo esplicito, comunicazione sincrona su canali tipizzati e transazioni con compensazione.

Questa tesi ne dà la semantica operazionale strutturale in stile small-step, definita come estensione della semantica di Janus di Lami, Lanese e Stefani[LLS24], e l'operatore di inversione che realizza l'esecuzione all'indietro. Sui canali si adotta una disciplina di sessione binaria sorvegliata durante l'esecuzione, che ammette la delega, cioè il passaggio di un canale da un titolare a un altro.

Indice

Elenco delle tabelle

Capitolo 0

Introduzione

Un linguaggio di programmazione è **reversibile** quando ogni sua computazione può essere ripercorsa all'indietro fino a ritrovare esattamente lo stato di partenza. Non si tratta di registrare una traccia da cui ricostruire il passato: in un linguaggio reversibile l'informazione necessaria a tornare indietro è già contenuta nello stato corrente, perché ogni costrutto è progettato per non distruggerla. L'assegnamento distruttivo $x := e$, che cancella il valore precedente di x, non appartiene a un linguaggio di questo tipo; al suo posto compaiono aggiornamenti invertibili come $x \mathrel{+}= e$, il cui inverso $x \mathrel{-}= e$ è immediato.

L'interesse per questo modo di calcolare ha tre radici indipendenti. La prima è fisica: il principio di Landauer lega la cancellazione di un bit a una dissipazione minima di energia, e un calcolo che non cancella può in linea di principio evitarla. La seconda è pragmatica: molti problemi si presentano naturalmente in coppie di trasformazioni inverse (compressione e decompressione, codifica e decodifica, serializzazione e parsing) e un linguaggio reversibile permette di scrivere *un solo* programma invece di due, eliminando per costruzione la possibilità che le due metà divergano. La terza riguarda l'affidabilità: poter tornare indietro è il meccanismo di base del *recovery*, e infatti il *rollback* di una transazione non è altro che esecuzione all'indietro fino a un punto sicuro.

È proprio la terza radice a rendere interessante il caso **concorrente**, ed è qui che si colloca questo lavoro. In un sistema sequenziale l'esecuzione all'indietro è una nozione semplice, perché c'è un solo passato da ripercorrere. In presenza di più thread il passato non è più lineare: due rami che procedono in parallelo possono aver interagito, e disfare un passo di uno può richiedere di disfare prima i passi dell'altro che ne dipendono. Il capostipite dei linguaggi reversibili, **Janus**[YG07], è puramente sequenziale; la reversibilità in contesto concorrente è stata studiata soprattutto su calcoli di processi[LMSS11;LLM+13], che sono modelli e non linguaggi di programmazione. Il territorio in mezzo (un linguaggio imperativo strutturato, con la sintassi e i costrutti che un programmatore si aspetta, che sia insieme reversibile e concorrente) è poco popolato: i tentativi esistono, ma finora hanno definito i costrutti concorrenti attraverso la loro traduzione, senza una semantica formale su cui poggiare le proprietà che ci si aspetta da un linguaggio reversibile.

Kairos sta in quel territorio. Il nome è il greco *καιρός* (*kairós*): non il tempo che scorre, che è *χρόνος* (*chrónos*), ma il momento opportuno, l'istante in cui una cosa va

fatta. La distinzione è pertinente a un linguaggio reversibile, dove il tempo non è una freccia ma una coordinata percorribile nei due versi, e ciò che conta di un passo non è *quando* avviene ma che sia quello giusto: l'inversa di un programma non ne riavvolge la storia, ne esegue un altro che riporta allo stato di partenza.

Al nucleo di Janus (assegnamenti invertibili, condizionali a doppia guardia, cicli `from ... until`, procedure invocabili con `call` e `uncall`, interi e array) Kairos aggiunge il parallelismo esplicito `par ... and ... rap`, la comunicazione sincrona su canali tipizzati `ssend`/`srecv`, uno stack come struttura dati primitiva e le transazioni `try ... rollback ... yrt`. La scelta di progetto che tiene insieme il tutto è la *disgiunzione*: rami paralleli operano su porzioni disgiunte dello store e comunicano solo per canali. Da questa singola ipotesi discende la confluenza degli interleaving che non toccano canali (§2.5), e quindi il fatto che il non-determinismo residuo non sia osservabile sullo store finale: la concorrenza di Kairos serve a esprimere il parallelismo dei problemi, non a introdurre corse critiche.

Il contributo di questo documento è la semantica formale del linguaggio. Diamo una semantica operazionale strutturale in stile small-step, definita come **estensione** della semantica di Janus di Lami, Lanese e Stefani [LLS24]: ereditiamo da quella base la forma dei giudizi, i contesti di valutazione, la configurazione terminale e il trattamento degli errori, e aggiungiamo le regole per i costrutti che Kairos introduce. Definiamo poi l'operatore di inversione $\mathcal{I}(\cdot)$, sintattico e calcolabile in tempo lineare, e mostriamo in che senso realizza l'esecuzione all'indietro.

0.1 Notazione e convenzioni

I giudizi hanno la forma $\langle \sigma, c \rangle \to \langle \sigma', c' \rangle$, con lo store a sinistra come nella semantica di base, non il turnstile big-step $\sigma \vdash e \Downarrow v$ usuale nella letteratura su Janus. Non è una variante stilistica: interleaving dei rami e sincronizzazione sui canali (Sezione 2.5) si esprimono molto più naturalmente passo per passo, perché un big-step collassa l'intera esecuzione di un comando in un unico giudizio e i punti in cui intercalare l'altro ramo restano impliciti. È la stessa motivazione dichiarata in [LLS24], che adotta lo small-step per Janus proprio in vista di una futura estensione concorrente.

Una convenzione tipografica riguarda ciò che sta *fuori* dalla relazione di transizione. Le regole d'errore, che producono $\bot$, portano il nome in rosso. I vincoli di buona formazione sono distinti secondo il momento in cui li si verifica: in verde quelli controllati sul testo del programma, prima di eseguirlo; in arancio quelli sorvegliati durante l'esecuzione, perché riguardano stato che il testo del programma non determina.

0.2 Struttura della tesi

Il Capitolo 1 richiama Janus, la semantica small-step su cui ci basiamo e i lavori sulla reversibilità concorrente. Il Capitolo 2 è il nucleo tecnico: sintassi astratta, sistema di transizione, il confine fra ciò che è ereditato dalla base e ciò che Kairos aggiunge, e le regole di riduzione dagli assegnamenti fino a `par` e alle transazioni, con gli errori a runtime. Il Capitolo 3 definisce l'operatore di inversione, ne dimostra le proprietà e riassume in un quadro unico tutte le regole. Seguono il Capitolo 4 sull'uso del

linguaggio, il Capitolo 5 sull'implementazione, il Capitolo 6 sul confronto con i lavori esistenti e il Capitolo 7 con le conclusioni.

Capitolo 1

Background

Janus e la programmazione reversibile

Janus è stato definito nel 1982 da Lutz e Derby come esercizio didattico e riportato all'attenzione della comunità da Yokoyama e Glück[YG07], che ne hanno dato la prima semantica formale e un self-interpreter invertibile. È un linguaggio imperativo strutturato in cui ogni costrutto è invertibile per costruzione, secondo due idee ricorrenti.

La prima è l'**aggiornamento invertibile**. Al posto dell'assegnamento distruttivo Janus ha solo aggiornamenti della forma $x \odot= e$ con $\odot \in \{+, -, \hat{}\}$, dove x non compare in e: il vecchio valore di x è recuperabile dal nuovo applicando l'operazione inversa, e la condizione sull'occorrenza di x garantisce che il valore di e sia lo stesso prima e dopo il passo.

La seconda è la **doppia guardia**. Un condizionale irreversibile `if` e `then` c_1 `else` c_2 non è invertibile perché, arrivati alla fine, non si sa più quale ramo sia stato preso. Janus richiede allora una seconda espressione, l'*asserzione* in coda:

$$\texttt{if } e_1 \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi } e_2,$$

con l'obbligo che e_2 valga vero all'uscita se e solo se si è preso il ramo `then`. All'indietro è e_2 a decidere il ramo ed e_1 a fare da asserzione: il ruolo delle due espressioni si scambia. Lo stesso schema governa il ciclo

$$\texttt{from } e_1 \texttt{ do } c_1 \texttt{ loop } c_2 \texttt{ until } e_2,$$

dove e_1 deve valere solo alla prima iterazione ed e_2 solo all'ultima, così che la condizione di ingresso all'indietro sia e_2 e quella di uscita e_1. I due corpi sono eseguiti secondo la traccia $c_1\,[\,c_2\;c_1\,]^*$: c_1 porta avanti il lavoro dell'iterazione, c_2 fa avanzare il contatore fra un'iterazione e la successiva. Kairos adotta la stessa forma; il ciclo a un corpo solo ne è il caso particolare $c_2 = \texttt{skip}$.

Su queste basi Janus è *r-Turing completo*: calcola esattamente le funzioni iniettive computabili, senza generare spazzatura. Le sue procedure si invocano con `call` in avanti e `uncall` all'indietro, e l'inverso di un programma si ottiene per trasformazione puramente sintattica.

Una semantica small-step per Janus

Le presentazioni classiche di Janus, a partire da[YG07], usano una semantica *big-step*: un giudizio $\sigma \vdash c \Downarrow \sigma'$ mette in relazione lo store prima e dopo l'esecuzione *completa* di c. È una scelta comoda per un linguaggio sequenziale, ma collassa in un unico passo l'intera computazione, e con essa i punti in cui un'altra esecuzione potrebbe intercalarsi.

Lami, Lanese e Stefani[LLS24] hanno dato una semantica **small-step** di Janus, con giudizi della forma $\langle \sigma, c \rangle \to \langle \sigma', c' \rangle$ fra configurazioni, contesti di valutazione per espressioni e comandi, configurazione terminale $\langle \sigma, \texttt{skip} \rangle$ ed errori segnalati da un esito distinto $\bot$. Uno degli scopi dichiarati di quel lavoro è proprio predisporre il terreno per estensioni concorrenti, sulle quali le proprietà di interesse si definiscono più agevolmente in forma small-step. Kairos parte da lì: adottarne le convenzioni significa che le scelte di presentazione sono già state fatte e argomentate a monte, e che le regole nuove sono solo quelle dei costrutti nuovi. La §2.3 elenca esattamente il confine fra le due cose.

Una semantica alternativa per Janus, in stile *reversible*, è quella di Lanese e Vidal[LV26], in cui le regole in avanti e quelle all'indietro sono date insieme sullo stesso sistema di transizione anziché derivare le seconde da un inverter sintattico.

Reversibilità e concorrenza

Fuori dai linguaggi imperativi, la reversibilità in presenza di concorrenza è stata studiata a fondo sui *calcoli di processi*. Il punto delicato è che in un sistema concorrente non esiste un unico ordine di esecuzione da rovesciare: la nozione corretta è la *reversibilità causalmente consistente*, per cui un passo può essere disfatto solo dopo che sono stati disfatti tutti i passi che da esso dipendono causalmente. In quest'ottica la reversibilità va anche *controllata*, perché tornare indietro senza limiti non è utile a nulla: in $\rho\pi$ e in croll-π[LMSS11;LLM+13] il ritorno all'indietro è governato da primitive esplicite, e il meccanismo che ne risulta (si torna indietro fino a un punto scelto e da lì si prosegue diversamente) è esattamente quello delle transazioni con compensazione. Le transazioni `try ... rollback ... yrt` di Kairos (§2.5) sono la versione a linguaggio di programmazione della stessa idea.

Kairos adotta però una strada più restrittiva, e per questo più semplice: rami paralleli lavorano su porzioni *disgiunte* dello store e interagiscono soltanto per comunicazione sincrona. Non c'è memoria condivisa, quindi non c'è dipendenza causale da tracciare fra passi che non comunicano, e la consistenza causale si riduce alla confluenza dimostrata in §2.5. Il prezzo è che alcuni schemi di programmazione a memoria condivisa non sono esprimibili; il guadagno è che nessun programma Kairos ben tipato può contenere una corsa critica.

Fra i calcoli di processo e i linguaggi di programmazione esiste comunque un precedente diretto: Janus è già stato esteso con un costrutto di composizione parallela e uno scambio di messaggi, e l'estensione è stata realizzata con un compilatore verso C++[Sam19]. Quel lavoro definisce i costrutti attraverso il codice in cui vengono tradotti, non attraverso regole di riduzione, e mantiene uno stato condiviso fra i rami; è quindi il punto di partenza più vicino a quello di questa tesi, e il Capitolo 6 lo riprende per un confronto puntuale.

Canali e tipi di sessione

La comunicazione di Kairos avviene su canali *sincroni* e *tipizzati*, con il vincolo che ogni canale abbia in ogni istante al più due titolari, una *sessione binaria*. È una restrizione che viene dalla teoria dei **tipi di sessione**[HVK98;HYC08], dove il tipo di un canale non descrive solo i dati scambiati ma la *sequenza* degli scambi, e i tipi dei due estremi devono essere duali. In Kairos il vincolo di sessione binaria serve a garantire che il partner di ogni comunicazione sia determinato: senza di esso un `ssend` potrebbe sincronizzarsi con più `srecv` diversi, e la scelta sarebbe un non-determinismo osservabile sullo store, in contrasto con la confluenza.

Il vincolo ha due forme, e la scelta fra le due non è di dettaglio. Nella forma *statica* un canale appartiene agli stessi due rami per tutta la sua vita, e questo si legge dal testo del programma. Nella forma *dinamica* un canale può stare prima fra due rami e poi fra altri due, purché il primo titolare smetta di usarlo prima che il nuovo cominci: è ciò che nella teoria dei tipi di sessione si chiama **delega**, il meccanismo con cui, in stile π-calcolo, un processo passa a un terzo il canale su cui sta parlando. La seconda è strettamente più espressiva, e Kairos adotta quella: il vincolo è sorvegliato osservando l'esecuzione (§2.5), non leggendo il testo del programma, perché il testo non basta a seguire un canale che cambia titolare.

Va detto dove sta il confine di questo lavoro. Il vincolo di sessione binaria e l'ipotesi di disgiunzione sono qui *assunzioni* sotto cui vale la confluenza, non un sistema di tipi: la semantica delle §§2.4–2.5 non impedisce di scrivere un programma con una corsa critica, si limita a non garantire nulla su di esso. La separazione è deliberata e segue la pratica corrente, in cui il calcolo si definisce senza preoccuparsi delle corse e un sistema di tipi sovrapposto le esclude; i tipi di sessione, binari o multiparty, sono precisamente una famiglia di sistemi di tipi di questo genere. In §2.5 diamo la disciplina binaria per i canali di Kairos, delega compresa, e i tipi di sessione che ne descrivono il protocollo: i tipi non si dichiarano ma si ricostruiscono dal programma, e i due estremi di ogni canale devono risultare duali. Restano fuori la scelta etichettata e i protocolli a più di due partecipanti, estensioni possibili che qui non perseguiamo.

Capitolo 2

Kairos: sintassi e semantica operazionale

2.1 Sintassi astratta

Insiemi di base: numeri interi $m \in \mathbb{Z}$, variabili $v \in \mathit{Var}$. Categorie derivate:

Espressioni aritmetiche $e \in \mathit{Exp}$**.**

$$e ::= m \mid v \mid a[e] \mid e \oplus e \qquad \oplus \in \{+, -, *, /, \%\}$$

$a[e]$ è la lettura della cella di indice e dell'array a. Gli operatori moltiplicativi compaiono solo dentro le espressioni, mai come assegnamento: $*$ e $/$ non hanno un'inversa che restituisca l'operando, mentre a destra di $+=$ non tolgono nulla, perché l'inversa dell'assegnamento è $-=$ con la *stessa* espressione, e questa dipende da variabili che il comando non tocca. $/$ e $\%$ sono la divisione intera e il resto; il divisore nullo non ha risultato e produce un errore:

$$\frac{}{\langle \sigma,\, m/0 \rangle \to_e \bot} \quad \text{(Div-Err)}$$

con la regola gemella per $\%$. È un fallimento come gli altri (§2.6), non un valore convenzionale: un valore convenzionale renderebbe l'operazione non iniettiva proprio dove serve che lo sia.

Condizioni $b \in \mathit{Cond}$**.**

$$b ::= e = e \mid e \neq e \mid e \geq e \mid e \leq e \mid e > e \mid e < e$$

Comandi $c \in \mathit{Com}$**.**

$c ::= \texttt{skip}$	(comando vuoto)
$\mid \ell \mathrel{+}= e \mid \ell \mathrel{-}= e \mid \ell \mathrel{\hat{}}= e \mid \ell \mathrel{<\!\!=\!\!>} \ell$	(assegnamenti reversibili)
$\mid \texttt{local}\ v = m \mid \texttt{delocal}\ v = m$	(scope)
$\mid \texttt{local}\ a[n] = m \mid \texttt{delocal}\ a[n] = m$	(scope, array)
$\mid c\ ;\ c$	(sequenza)
$\mid \texttt{if}\ b\ [\texttt{then}\ c]\ [\texttt{else}\ c]\ \texttt{fi}\ b'$	(condizionale)
$\mid \texttt{from}\ b_1\ [\texttt{do}\ c_1]$ $[\texttt{loop}\ c_2]\ \texttt{until}\ b_2$	(ciclo)
$\mid \texttt{call}\ f(\bar{v}) \mid \texttt{uncall}\ f(\bar{v})$	(chiamata / inversa)
$\mid \texttt{push}(v, s) \mid \texttt{pop}(v, s)$	(stack)
$\mid \texttt{ssend}(\langle \bar{v} \rangle, ch) \mid \texttt{srecv}(\langle \bar{v} \rangle, ch)$	(canali)
$\mid \texttt{par}\ c\ \texttt{and}\ c\ [\texttt{and}\ c]^*\ \texttt{rap}$	(parallelo)
$\mid \texttt{try}\ b\ c\ [\texttt{rollback}\ c]\ \texttt{yrt}\ b'$	(commit / rollback)

Il bersaglio di un assegnamento è un *luogo* $\ell ::= v \mid a[e]$: una variabile intera oppure una cella di array. Le due forme si comportano allo stesso modo — l'unica differenza è che la cella va individuata valutando l'indice prima di scrivere.

Un *programma* è una lista di procedure $\texttt{procedure}\ f(\bar{v})\ \{\, c \,\}$; $\bar{v}$ è la lista (eventualmente vuota) dei parametri. La sequenza è scritta con il punto e virgola esplicito, $c_1\,;\,c_2$, come nelle regole di §2.5. Il condizionale ha **due** guardie (b d'ingresso, b' d'uscita) e il ciclo pure (b_1 d'ingresso, b_2 d'uscita): è ciò che rende il linguaggio reversibile (§3). Le parentesi quadre segnano le clausole *facoltative*, come in Janus: un ramo assente vale `skip`, e nelle regole che seguono si scrivono per esteso tutte e due, senza perdita di generalità. Un programma ben formato soddisfa inoltre due vincoli: su `par`, ogni canale usato al suo interno è condiviso da esattamente due rami concorrenti (§2.5); su `try`…`rollback`…`yrt`, né il corpo c né il corpo di rollback r possono contenere `ssend`/`srecv`, né direttamente né tramite `call`/`uncall` (§2.5).

2.2 Sistema di transizione e store

Un sistema di transizione è una tripla $\langle \Gamma, \top, \to \rangle$ con Γ insieme degli stati, $\top \subseteq \Gamma$ stati terminali, $\to \subseteq \Gamma \times \Gamma$ relazione di transizione. $\to^*$ è la sua chiusura riflessiva e transitiva:

$$\frac{}{y \to^* y} \qquad \frac{y \to^* y' \quad y' \to y''}{y \to^* y''}$$

Per dare significato alle variabili introduciamo uno **store**. In Kairos le variabili sono di quattro tipi: `int` (interi), `int`$[n]$ (array: n interi, con n fissato alla dichiarazione), `stack` (sequenze di interi) e `channel` (canali: nomi di sincronizzazione tra rami paralleli). Array e stack sono entrambi sequenze di interi ma si usano in modi opposti: l'array ha lunghezza fissa e si legge per indice senza consumarlo, lo stack cresce e cala e si tocca solo in cima. Un canale **non** contiene valori: è un punto di rendez-vous, non un contenitore (§2.5), a differenza di `stack`, un canale allocato mantiene sempre la

stessa identità per tutta la sua vita, e $\sigma(ch)$ non compare mai nell'aggiornamento di una regola. Indicando con *Chan* l'insieme (opaco) delle identità di canale, lo store è

$$\sigma : \mathit{Var} \rightharpoonup \mathbb{Z} \cup \mathbb{Z}^n \cup \mathbb{Z}^* \cup \mathit{Chan},$$

parziale perché lo scope cambia con `local`/`delocal`. Per le sequenze: ε è la vuota, $n : \ell$ è il *cons* (la testa n è la cima dello stack). Scriviamo $\sigma[v \mapsto w]$ per l'aggiornamento e $\sigma \setminus v$ per la rimozione di v dal dominio. ε denota qui **solo** la sequenza vuota di uno `stack`: è un valore, non un comando. Il comando vuoto è un simbolo distinto, `skip` (§2.1, §2.5), le due nozioni non vanno confuse.

2.3 La semantica di base

Kairos non viene definito da zero. La sua semantica è data come **estensione** della semantica operazionale small-step di Janus di Lami, Lanese e Stefani [LLS24]: da lì ereditiamo la forma del giudizio, la configurazione terminale, il trattamento degli errori e l'apparato dei contesti di valutazione; a quella base aggiungiamo i costrutti che Janus non ha. Le scelte di progetto già compiute in quel lavoro non vengono quindi ridiscusse qui: dove questo documento tace, vale la convenzione della base. Nel seguito chiamiamo *la base* quella semantica, e segnaliamo esplicitamente ogni punto in cui ce ne discostiamo.

Giudizio. Le regole hanno la forma

$$\langle \sigma,\, c \rangle \to \langle \sigma',\, c' \rangle \qquad \text{oppure} \qquad \langle \sigma,\, c \rangle \to \bot,$$

dove la seconda denota un **errore a runtime** dovuto al fallimento di un'asserzione. Lo store sta a *sinistra* del comando, come nella base. La base usa un'unica relazione $\to$ su tutte le categorie sintattiche; qui manteniamo la decorazione $\to_e$ / $\to_b$ / $\to_c$ per distinguere il frammento delle espressioni, quello delle condizioni e quello dei comandi, ma si tratta della stessa relazione letta su categorie diverse.

Configurazione terminale. Nella base la computazione termina in $\langle \sigma,\, \texttt{skip} \rangle$, e per questo *non* esiste una regola per `skip`: citando [LLS24], *"there is no rule for* `skip`*, since* $\langle \sigma, \texttt{skip} \rangle$ *denotes the end of the computation"*. È la convenzione standard, e la adottiamo: ogni regola che porta a termine un comando conclude in $\langle \sigma,\, \texttt{skip} \rangle$.

Configurazione terminale. La convenzione è adottata *ovunque* in questo documento: ogni regola che porta a termine un comando conclude in $\langle \sigma,\, \texttt{skip} \rangle$, non nello store nudo. Non esiste quindi una regola per `skip`; il consumo di `skip` in sequenza è affidato a SeqS (§2.5), e la terminazione di un ramo parallelo a Par-L0/Par-R0 (§2.5).

Errori. L'esito d'errore è $\bot$, un elemento distinto da ogni configurazione. Un errore non viene propagato da una regola per ciascun costrutto, ma da un'unica regola contestuale CtxError (sotto): è la scelta della base, e sostituisce le regole di propagazione ad hoc.

Contesti di valutazione. Per dare una definizione compatta della semantica, la base solleva la riduzione di sotto-termini tramite *contesti di valutazione*: un contesto è un comando in cui un sotto-termine è sostituito da un buco $\bullet$. Si distinguono i contesti d'**espressione** $C_e[\bullet]$, il cui buco va riempito con un'espressione o una condizione, e i contesti di **comando** $C_s[\bullet]$, il cui buco va riempito con un comando; $C_e[e]$ e $C_s[c]$ denotano il comando ottenuto sostituendo $\bullet$ rispettivamente con e e con c. Adattati a Kairos, i contesti sono

$$
\begin{aligned}
C_e[\bullet] ::= {} & v \mathrel{+}= \bullet \mid v \mathrel{-}= \bullet \mid v \mathrel{\hat{}}= \bullet && \text{(§2.5)}\\
\mid {} & \texttt{if } \bullet \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi } b_2 && \text{(§2.5)}\\
\mid {} & \texttt{if } \mathit{tt} \texttt{ then skip else } c_2 \texttt{ fi } \bullet \mid \texttt{if } \mathit{ff} \texttt{ then } c_1 \texttt{ else skip fi } \bullet \\
\mid {} & ((\bullet, b_1),\, c_1,\, b_2,\, c_2) \mid (b_1,\, (\texttt{skip}, c_1),\, (\bullet, b_2),\, c_2) && \text{(§2.5)}\\
\mid {} & ((\bullet, b_1),\, (\texttt{skip}, c_1),\, b_2,\, (\texttt{skip}, c_2)) \\
C_s[\bullet] ::= {} & \bullet ;\, c && \text{(§2.5)}\\
\mid {} & \texttt{if } \mathit{tt} \texttt{ then } \bullet \texttt{ else } c_2 \texttt{ fi } b_2 \mid \texttt{if } \mathit{ff} \texttt{ then } c_1 \texttt{ else } \bullet \texttt{ fi } b_2 && \text{(§2.5)}\\
\mid {} & (b_1,\, (\bullet, c_1),\, (b_2, b_2),\, c_2) \mid ((b_1, b_1),\, (\texttt{skip}, c_1),\, b_2,\, (\bullet, c_2)) && \text{(§2.5)}\\
\mid {} & \texttt{par } \bullet \texttt{ and } c \texttt{ rap} \mid \texttt{par } c \texttt{ and } \bullet \texttt{ rap} && \text{(§2.5, propria di Kairos)}
\end{aligned}
$$

Le clausole con le quadruple appartengono al ciclo e sono spiegate in §2.5; quelle di `par` sono l'unica aggiunta di Kairos, e sono discusse in §2.5. Non c'è una clausola per `try`: le sue regole (§2.5) riducono il corpo con $\to_c^*$ in una premessa, non in posizione, quindi il corpo non è un buco di contesto. Le tre regole contestuali sono quelle della base:

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, C_e[e]\rangle \to_c \langle \sigma,\, C_e[e']\rangle}\ (\textsc{CtxExp}) \qquad \frac{\langle \sigma,\, c\rangle \to_c \langle \sigma',\, c'\rangle}{\langle \sigma,\, C_s[c]\rangle \to_c \langle \sigma',\, C_s[c']\rangle}\ (\textsc{CtxStmt})$$

$$\frac{\langle \sigma,\, c\rangle \to_c \bot}{\langle \sigma,\, C_s[c]\rangle \to_c \bot}\ (\textsc{CtxError})$$

CtxExp e CtxStmt sollevano un passo riuscito, CtxError solleva un errore. Le stesse regole valgono per una condizione al posto di un'espressione, sostituendo $\to_e$ con $\to_b$: nella base gli operatori relazionali sono operatori binari come gli altri e non esiste una categoria separata (§2.4). Applicandole ripetutamente si attraversa un annidamento di contesti di profondità qualunque.

Queste tre regole rendono superflue tutte le regole di congruenza e di propagazione dell'errore scritte una per costrutto: Seq1 (§2.5) è CtxStmt istanziata su $C_s[\bullet] = \bullet ;\, c$, Par-L/Par-R (§2.5) sono CtxStmt istanziata sulle due posizioni del `par`, e CtxError rende superflua ogni regola di propagazione dell'errore dedicata a un singolo costrutto (§2.6). Le istanze di contesto le manteniamo scritte per esteso dove aiutano la lettura, segnalando di volta in volta che sono istanze e non regole indipendenti.

Che cosa aggiunge Kairos

Quanto segue è il **contributo di questa tesi**: sono i costrutti e i vincoli che la base non ha e che vengono definiti qui per la prima volta sopra di essa. La base è esplicita sui propri limiti: *"In this paper, we do not consider parameters or local variables"*[LLS24]. Kairos estende dunque quella semantica su sette fronti:

1. **Variabili locali**: `local` / `delocal` (§2.5), assenti dalla base.

2. **Parametri di procedura**: `call` $f(\bar{v})$ / `uncall` $f(\bar{v})$ (§2.5); la base ha solo `call` *id* senza parametri, e passa i valori per effetto collaterale sullo store globale.
3. **Stack**: `push` / `pop` con la convenzione *zero-cleared* (§2.5); la base ha array $x[e]$, non stack.
4. **Canali**: `ssend` / `srecv` con **transizioni etichettate** (§2.5).
5. **Composizione parallela**: `par` ... `and` ... `rap`, con interleaving e sincronizzazione Par-Comm (§2.5).
6. **Transazioni**: `try` ... `rollback` ... `yrt` (§2.5).
7. **Sessione binaria** sui canali come proprietà dinamica, delega compresa (§2.5), con le proprietà che ne discendono: confluenza (§2.5) e reversibilità estesa al caso `par` (§3).

La base è sequenziale, ma adotta lo small-step *proprio in vista* di un'estensione concorrente: i punti 4–7 sono l'estensione che quella scelta rendeva possibile.

2.4 Espressioni e condizioni

Adottiamo **senza modifiche** la semantica small-step delle espressioni della base ([LLS24], Fig. 5): il giudizio $\langle \sigma, e\rangle \to_e \langle \sigma, e'\rangle$ non tocca mai lo store e riduce costanti (ConS), variabili (VarS) e operatori binari (Bop1–Bop3) con strategia interna sinistra. Come nella base, la sintassi di espressioni e condizioni è *estesa con i valori come foglie*, così che un termine parzialmente valutato resti un termine: è ciò che permette ai contesti $C_e[\bullet]$ (§2.3) di sollevarne la riduzione.

Le **condizioni** non sono una categoria separata nella base: gli operatori relazionali $(=, \neq, \geq, \leq, >, <)$ sono già fra i suoi operatori binari $\odot$, e non richiedono quindi regole proprie. Manteniamo per leggibilità la decorazione $\to_b$ per la riduzione di una condizione, i cui valori sono *tt* (vero) e *ff* (falso), dove la base usa i predicati $\mathsf{is\_true}?(v)$ e $\mathsf{is\_false}?(v)$ su un valore qualunque.

Poiché $\to_e$ e $\to_b$ sono deterministiche e non modificano lo store, si ha $\langle \sigma, e\rangle \to_e^* \langle \sigma, m\rangle$ con m unico e $\langle \sigma, b\rangle \to_b^* \langle \sigma, \beta\rangle$ con $\beta \in \{tt, ff\}$ unico; scriviamo $eval(e, \sigma) = m$ e $eval_b(b, \sigma) = \beta$. In Kairos `push` e `pop` restano **fuori** dalle espressioni (§2.5): le espressioni non hanno effetti collaterali, e la loro valutazione resta perciò esattamente quella della base.

2.5 Semantica dei comandi ($\to_c$)

$\Gamma_c = \{\langle \sigma, c\rangle\} \cup \{\bot\}$, terminali $\top_c = \{\langle \sigma, \texttt{skip}\rangle\} \cup \{\bot\}$: come nella semantica di base (§2.3), una configurazione *terminale* è $\langle \sigma, \texttt{skip}\rangle$: non esiste alcuna regola per `skip`, perché $\langle \sigma, \texttt{skip}\rangle$ *è* la fine della computazione.

Sequenza

`skip` è il comando vuoto e, da solo, è la configurazione terminale: non ha regole. Non va confuso con ε (§2.2), che è la sequenza vuota di uno `stack`, un *valore*, non un

comando. In sequenza, `skip` viene consumato dalla regola SeqS della base:

$$\frac{\langle \sigma,\, c_1\rangle \to_c \langle \sigma',\, c_1'\rangle}{\langle \sigma,\, c_1 ;\, c_2\rangle \to_c \langle \sigma',\, c_1' ;\, c_2\rangle}\ (\textsc{Seq1}) \qquad \frac{}{\langle \sigma,\, \texttt{skip} ;\, c_2\rangle \to_c \langle \sigma,\, c_2\rangle}\ (\textsc{SeqS})$$

$$\frac{\langle \sigma,\, c_1\rangle \xrightarrow{\alpha} \langle \sigma,\, c_1'\rangle}{\langle \sigma,\, c_1 ;\, c_2\rangle \xrightarrow{\alpha} \langle \sigma,\, c_1' ;\, c_2\rangle}\ (\textsc{Seq-Lbl})$$

Seq1 è CtxStmt (§2.3) istanziata sul contesto $C_s[\bullet] = \bullet\,;\, c$: la riportiamo per esteso solo per leggibilità. SeqS non è invece un'istanza di contesto: è la regola della base che consuma un `skip` già raggiunto e fa avanzare la sequenza al comando successivo (§2.3). Seq-Lbl solleva un passo **etichettato** (§2.5) di c_1 nella sequenza, per un'etichetta α generica: serve perché CtxStmt solleva solo passi $\to_c$ non etichettati, ed è ciò che permette a un `ssend`/`srecv` seguito da altro codice (l'idioma normale, non solo un canale da solo nel corpo di una procedura), di offrirsi comunque come partner in Par-Comm (§2.5). Non serve invece alcuna regola d'errore per la sequenza: un $\bot$ prodotto da c_1 risale grazie a CtxError sullo stesso contesto $\bullet\,;\, c$ (§2.6).

Assegnamenti reversibili

Vincolo (verificato staticamente): la variabile aggiornata non compare a destra ($v \notin e$), così l'operazione è iniettiva.

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, v \mathrel{+}= e\rangle \to_c \langle \sigma,\, v \mathrel{+}= e'\rangle}\ (\textsc{Add1})$$

$$\frac{}{\langle \sigma,\, v \mathrel{+}= m\rangle \to_c \langle \sigma[v \mapsto \sigma(v) + m],\, \texttt{skip}\rangle}\ (\textsc{Add2})$$

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, v \mathrel{-}= e\rangle \to_c \langle \sigma,\, v \mathrel{-}= e'\rangle}\ (\textsc{Sub1})$$

$$\frac{}{\langle \sigma,\, v \mathrel{-}= m\rangle \to_c \langle \sigma[v \mapsto \sigma(v) - m],\, \texttt{skip}\rangle}\ (\textsc{Sub2})$$

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, v \mathrel{\hat{}}= e\rangle \to_c \langle \sigma,\, v \mathrel{\hat{}}= e'\rangle}\ (\textsc{Xor1})$$

$$\frac{}{\langle \sigma,\, v \mathrel{\hat{}}= m\rangle \to_c \langle \sigma[v \mapsto \sigma(v) \oplus m],\, \texttt{skip}\rangle}\ (\textsc{Xor2})$$

$$\frac{}{\langle \sigma,\, v_1 \mathrel{<\!\!=\!\!>} v_2\rangle \to_c \langle \sigma[v_1 \mapsto \sigma(v_2)][v_2 \mapsto \sigma(v_1)],\, \texttt{skip}\rangle}\ (\textsc{Swap})$$

Celle di array. Quando il bersaglio è $a[e_i]$ le regole sono le stesse, con un passo in più davanti: l'indice si riduce a un intero prima che si scriva. Scriviamo $\sigma(a)_k$ per la k-esima componente e $\sigma[a\langle k\rangle \mapsto m]$ per l'aggiornamento della sola cella k, cioè $\sigma[a \mapsto \sigma(a)[k \mapsto m]]$. Diamo le regole per $+\!=$; $-\!=$ e $\hat{}=$ sono identiche a meno dell'operatore.

$$\frac{\langle \sigma,\, e_i\rangle \to_e \langle \sigma,\, e_i'\rangle}{\langle \sigma,\, a[e_i] \mathrel{+}= e\rangle \to_c \langle \sigma,\, a[e_i'] \mathrel{+}= e\rangle}\ (\textsc{IdxL})$$

$$\frac{0 \le k < |\sigma(a)|}{\langle \sigma,\, a[k] \mathrel{+}= m\rangle \to_c \langle \sigma[a\langle k\rangle \mapsto \sigma(a)_k + m],\, \texttt{skip}\rangle}\ (\textsc{AddA})$$

$$\frac{k < 0 \ \vee\ k \ge |\sigma(a)|}{\langle \sigma,\, a[k] \mathrel{+}= m\rangle \to_c \bot}\ (\textsc{Idx-Err})$$

L'indice fuori dai limiti è un errore e non un accesso silenzioso: in un linguaggio reversibile scrivere fuori dal vettore romperebbe l'inversione senza lasciare traccia, e l'errore comparirebbe altrove.

Il vincolo statico si estende di conseguenza: in $a[e_i] \mathrel{+}= e$ il nome a non può comparire in e. È più forte del necessario — basterebbe che non fosse la *stessa* cella — ma gli indici non sono decidibili staticamente, ed è la stessa scelta di Janus. Lo scambio $a[e_1] \mathrel{<}\!\!\Rightarrow a[e_2]$ resta invece ammesso: è involutivo comunque siano gli indici, ed è quello che rende scrivibile un ordinamento sul posto.

Scope: `local` / `delocal`

`local` alloca v con valore m; `delocal` *verifica* che v valga ancora m e poi lo rimuove (l'asserzione rende l'operazione invertibile).

$$\frac{v \notin \mathrm{dom}(\sigma)}{\langle \sigma,\ \texttt{local}\ v = m \rangle \to_c \langle \sigma[v \mapsto m],\ \texttt{skip} \rangle} \quad (\textsc{Local})$$

$$\frac{\sigma(v) = m}{\langle \sigma,\ \texttt{delocal}\ v = m \rangle \to_c \langle \sigma \setminus v,\ \texttt{skip} \rangle} \quad (\textsc{Deloc})$$

$$\frac{v \in \mathrm{dom}(\sigma)}{\langle \sigma,\ \texttt{local}\ v = m \rangle \to_c \bot} \quad (\textsc{Local-Err})$$

$$\frac{v \notin \mathrm{dom}(\sigma) \ \lor \ \sigma(v) \neq m}{\langle \sigma,\ \texttt{delocal}\ v = m \rangle \to_c \bot} \quad (\textsc{Deloc-Err})$$

Per un array il valore dichiarato vale per *ogni* cella: aprirlo lo riempie di m, chiuderlo verifica che tutte lo valgano ancora. Una sola cella fuori posto è informazione non restituita, e la chiusura fallisce.

$$\frac{a \notin \mathrm{dom}(\sigma)}{\langle \sigma,\ \texttt{local}\ a[n] = m \rangle \to_c \langle \sigma[a \mapsto m^n],\ \texttt{skip} \rangle} \quad (\textsc{LocalA})$$

$$\frac{\sigma(a) = m^n}{\langle \sigma,\ \texttt{delocal}\ a[n] = m \rangle \to_c \langle \sigma \setminus a,\ \texttt{skip} \rangle} \quad (\textsc{DelocA})$$

dove m^n è la sequenza di n copie di m. I due casi d'errore sono quelli di Local-Err e Deloc-Err, con in più il disaccordo sulla lunghezza: chiudere come $a[n']$ un array aperto come $a[n]$, con $n' \neq n$, non è l'inversa dell'apertura. Un'unica sottigliezza propria di Deloc-Err: se $v \notin \mathrm{dom}(\sigma)$, $\sigma(v)$ non è definito e non si può confrontare con m: la premessa copre esplicitamente anche questo caso, non solo $\sigma(v) \neq m$.

Condizionale (doppia guardia)

La guardia d'uscita b' deve essere coerente col ramo: *vera* dopo `then`, *falsa* dopo `else`. Adottiamo le quattro regole della base ([LLS24], Fig. 8), che **non** usano alcun comando ausiliario `assert`: la guardia d'uscita si valuta *nel contesto*, in posizione, e il confronto fra il valore del test e quello dell'asserzione decide fra passo riuscito ed errore.

Le regole si applicano quindi a un condizionale già ridotto: CtxExp (§2.3) ha portato il test a un valore, CtxStmt ha ridotto a `skip` il ramo selezionato, e CtxExp ha infine

valutato l'asserzione. Le clausole di $C_e[\bullet]$ e $C_s[\bullet]$ per il condizionale sono esattamente le posizioni usate qui.

$$\frac{}{\langle \sigma,\ \texttt{if}\ tt\ \texttt{then skip else}\ c_e\ \texttt{fi}\ tt\rangle \to_c \langle \sigma,\ \texttt{skip}\rangle}\ (\textsc{IfTrueEnd})$$

$$\frac{}{\langle \sigma,\ \texttt{if}\ ff\ \texttt{then}\ c_t\ \texttt{else skip fi}\ ff\rangle \to_c \langle \sigma,\ \texttt{skip}\rangle}\ (\textsc{IfFalseEnd})$$

$$\frac{}{\langle \sigma,\ \texttt{if}\ tt\ \texttt{then skip else}\ c_e\ \texttt{fi}\ ff\rangle \to_c \bot}\ (\textsc{IfError1})$$

$$\frac{}{\langle \sigma,\ \texttt{if}\ ff\ \texttt{then}\ c_t\ \texttt{else skip fi}\ tt\rangle \to_c \bot}\ (\textsc{IfError2})$$

Le prime due regole danno il passo riuscito quando test e asserzione hanno lo *stesso* valore di verità (entrambi tt, caso IfTrueEnd; entrambi ff, caso IfFalseEnd); le altre due segnalano l'errore quando i due valori *differiscono*. L'asserzione d'uscita non richiede quindi alcun comando ausiliario da eseguire in coda al ramo: viene valutata in posizione di contesto, come il test d'ingresso.[1]

Ciclo `from ... do ... loop ... until`

Adottiamo il ciclo **a due corpi** della base, con le sue sette regole ([LLS24], Fig. 9): b_1 è l'asserzione d'ingresso (vera solo alla prima entrata), c_1 il *do*-corpo, c_2 il *loop*-corpo eseguito *fra* un'iterazione e la successiva, b_2 il test d'uscita. La traccia d'esecuzione è $c_1\,[c_2\,c_1]^*$: c_1 gira k volte e c_2 gira $k-1$ volte.

Come nella base, l'esecuzione usa un termine ausiliario a runtime, la **quadrupla** (X_1, X_2, X_3, X_4), dove ciascuna componente è o il termine originale *a riposo*, o una *coppia* (corrente, originale) che segna la componente come in corso di valutazione e ne conserva accanto l'originale, per poterlo riavviare all'iterazione seguente:

$$X_1 ::= b_1 \mid (\beta, b_1) \quad X_2 ::= c_1 \mid (c, c_1) \quad X_3 ::= b_2 \mid (\beta, b_2) \quad X_4 ::= c_2 \mid (c, c_2)$$

con β un reduct di una condizione. La distinzione fra forma nuda e forma a coppia è ciò che rende le regole **mutuamente esclusive**: p.es. LoopExp1T si applica quando X_2 e X_4 sono nudi (prima entrata), LoopError2 quando sono coppie con corrente `skip` (fine di un'iterazione). Le riduzioni interne alla quadrupla sono sollevate dai contesti $C_e[\bullet]$ e $C_s[\bullet]$ di §2.3, le cui clausole per il ciclo sono esattamente le posizioni usate qui.

$$\frac{}{\langle \sigma,\ \texttt{from}\ b_1\ \texttt{do}\ c_1\ \texttt{loop}\ c_2\ \texttt{until}\ b_2\rangle \to_c \langle \sigma,\ ((b_1, b_1),\ c_1,\ b_2,\ c_2)\rangle}\ (\textsc{LoopMainS})$$

$$\frac{}{\langle \sigma,\ ((tt, b_1),\ c_1,\ b_2,\ c_2)\rangle \to_c \langle \sigma,\ (b_1,\ (c_1, c_1),\ (b_2, b_2),\ c_2)\rangle}\ (\textsc{LoopExp1T})$$

$$\frac{}{\langle \sigma,\ (b_1,\ (\texttt{skip}, c_1),\ (tt, b_2),\ c_2)\rangle \to_c \langle \sigma,\ \texttt{skip}\rangle}\ (\textsc{LoopEndS})$$

$$\frac{}{\langle \sigma,\ (b_1,\ (\texttt{skip}, c_1),\ (ff, b_2),\ c_2)\rangle \to_c \langle \sigma,\ ((b_1, b_1),\ (\texttt{skip}, c_1),\ b_2,\ (c_2, c_2))\rangle}\ (\textsc{LoopExp2F})$$

[1] Nella Fig. 8 di[LLS24] la regola IfFalseEnd è stampata con premessa $\mathsf{is\_true?}(v_2)$, il che la renderebbe identica a IfError2. Si tratta evidentemente di un refuso: il testo che accompagna la figura dice esplicitamente che le prime due regole coprono il caso in cui test e asserzione sono *"either both true* (IfTrueEnd *case*) *or both false* (IfFalseEnd *case*)*"*. Qui riportiamo la versione corretta, con l'asserzione falsa.

$$\frac{}{\langle \sigma,\ ((\mathit{ff}, b_1),\ (\texttt{skip}, c_1),\ b_2,\ (\texttt{skip}, c_2))\rangle \to_c \langle \sigma,\ (b_1,\ (c_1, c_1),\ (b_2, b_2),\ c_2)\rangle} \quad \text{(LoopExp1F)}$$

$$\frac{}{\langle \sigma,\ ((\mathit{ff}, b_1),\ c_1,\ b_2,\ c_2)\rangle \to_c \bot} \quad \text{(LoopError1)}$$

$$\frac{}{\langle \sigma,\ ((\mathit{tt}, b_1),\ (\texttt{skip}, c_1),\ b_2,\ (\texttt{skip}, c_2))\rangle \to_c \bot} \quad \text{(LoopError2)}$$

Il flusso è dunque: LoopMainS entra nel ciclo e arma la valutazione di b_1; CtxExp la porta a un valore, e LoopExp1T (che richiede b_1 *vera*) avvia c_1 e arma b_2. L'esecuzione di c_1 avviene tramite CtxStmt; quando c_1 è ridotto a `skip`, CtxExp valuta b_2: se è vera LoopEndS termina il ciclo in $\langle \sigma,\ \texttt{skip}\rangle$, se è falsa LoopExp2F avvia il *loop*-corpo c_2 e riarma b_1. Terminato c_2, b_1 viene rivalutata e deve ora essere *falsa*: LoopExp1F riavvia l'iterazione. Le due regole d'errore coprono i due casi in cui l'asserzione b_1 ha il valore sbagliato: LoopError1 quando b_1 è falsa *all'ingresso*, LoopError2 quando b_1 ridiventa *vera* a metà ciclo. Sono le uniche due regole d'errore del ciclo: ogni altro fallimento nel corpo risale per CtxError.

Il ciclo di Janus. Kairos adotta *tal quale* il ciclo di Janus, nella forma e con le regole date dalla base[LLS24], a sua volta fedele all'originale di Yokoyama e Glück[YG07]. La forma a un solo corpo, `from` b_1 `loop` c `until` b_2, ne è il caso particolare $c_2 = \texttt{skip}$, cioè `from` b_1 `do` c `loop skip until` b_2; con il secondo corpo si recupera l'idioma standard di Janus (il lavoro nel *do*-corpo, l'avanzamento del contatore nel *loop*-corpo), che con un corpo solo non era esprimibile direttamente.

Chiamata e inversa

Sia `procedure` $f(\bar{p})\ \{\, c_f \,\}$. La chiamata sostituisce i parametri formali $\bar{p}$ con gli attuali $\bar{v}$ (lo $\to_c$ opera sulle celle del chiamante). `uncall` esegue il corpo *invertito* $\mathcal{I}(c_f)$ (operatore $\mathcal{I}(\cdot)$, §3).

$$\frac{}{\langle \sigma,\ \texttt{call}\ f(\bar{v})\rangle \to_c \langle \sigma,\ c_f[\bar{v}/\bar{p}]\rangle} \quad \text{(Call)}$$

$$\frac{}{\langle \sigma,\ \texttt{uncall}\ f(\bar{v})\rangle \to_c \langle \sigma,\ \mathcal{I}(c_f)[\bar{v}/\bar{p}]\rangle} \quad \text{(Uncall)}$$

Stack

$\texttt{push}(v, s)$ sposta il valore di v in cima a s e **azzera** v; $\texttt{pop}(v, s)$ fa il duale (estrae la cima e la mette in v), con la premessa *zero-cleared*: il bersaglio deve essere già 0.

$$\frac{\sigma(v) = n}{\langle \sigma,\ \texttt{push}(v, s)\rangle \to_c \langle \sigma[v \mapsto 0][s \mapsto n : \sigma(s)],\ \texttt{skip}\rangle} \quad \text{(Push)}$$

$$\frac{\sigma(v) = 0 \quad \sigma(s) = n : \ell}{\langle \sigma,\ \texttt{pop}(v, s)\rangle \to_c \langle \sigma[v \mapsto n][s \mapsto \ell],\ \texttt{skip}\rangle} \quad \text{(Pop)}$$

$$\frac{\sigma(v) = 0 \quad \sigma(s) = \varepsilon}{\langle \sigma,\ \texttt{pop}(v, s)\rangle \to_c \bot} \quad \text{(Pop-Err)}$$

$$\frac{\sigma(v) \neq 0}{\langle \sigma,\ \texttt{pop}(v, s)\rangle \to_c \bot} \quad \text{(Pop-Err2)}$$

Se $\sigma(v) \neq 0$, Pop-Err2 dà $\bot$.

Canali: `ssend` / `srecv`

Un canale *ch* **non** contiene valori: è un nome di sincronizzazione, un punto di rendez-vous tra i due rami che in quel momento ne sono titolari (vincolo di buona formazione, sotto). `ssend` e `srecv` non riducono in isolamento: ciascuno produce un passo **etichettato**, che si combina con quello del ramo partner nella regola Par-Comm (§2.5) in un unico passo $\to_c$ del `par` che li contiene, le etichette sono

$$\alpha ::= ch!\langle \bar{v}, \bar{n} \rangle \mid ch?\langle \bar{w} \rangle$$

($ch!\langle \bar{v}, \bar{n} \rangle$: offerta in invio, variabili $\bar{v}$ e valori correnti $\bar{n}$; $ch?\langle \bar{w} \rangle$: offerta in ricezione, variabili bersaglio $\bar{w}$). I passi etichettati non modificano da soli lo store, il movimento avviene tutto in Par-Comm:

$$\frac{\sigma(v_1) = n_1 \quad \cdots \quad \sigma(v_k) = n_k}{\langle \sigma,\ \texttt{ssend}(\langle \bar{v} \rangle, ch) \rangle \xrightarrow{ch!\langle \bar{v}, \bar{n} \rangle} \langle \sigma,\ \texttt{skip} \rangle} \quad \text{(Ssend-Lbl)}$$

$$\frac{\sigma(w_1) = 0 \quad \cdots \quad \sigma(w_k) = 0}{\langle \sigma,\ \texttt{srecv}(\langle \bar{w} \rangle, ch) \rangle \xrightarrow{ch?\langle \bar{w} \rangle} \langle \sigma,\ \texttt{skip} \rangle} \quad \text{(Srecv-Lbl)}$$

$$\frac{\exists i.\ \sigma(w_i) \neq 0}{\langle \sigma,\ \texttt{srecv}(\langle \bar{w} \rangle, ch) \rangle \to_c \bot} \quad \text{(Srecv-Err)}$$

Srecv-Err non è un passo etichettato: è una premessa su stato **locale** al ramo ricevente ($\sigma(\bar{w}) \neq \bar{0}$), quindi produce $\bot$ direttamente, senza attendere né dipendere da alcun partner, esattamente come Pop-Err2. Ssend-Lbl non ha un errore corrispondente: leggere $\sigma(\bar{v})$ non ha precondizioni, può assumere qualunque valore.

Il blocco è ora **emergente**, non il fallimento di una premessa condivisa: se nessun ramo concorrente offre l'etichetta duale, nessuna regola dell'intero sistema si applica a quella configurazione: non c'è una regola Ssend/Srecv "in isolamento" da cui una premessa possa mancare; semplicemente Par-Comm (§2.5) non trova un match. La sincronizzazione stessa *è* la premessa condivisa.

Perché non serve una coda. Un canale ha al più due titolari (vincolo di sessione binaria, sotto): in ogni istante c'è al più un `ssend` e un `srecv` in attesa sullo stesso canale, e Par-Comm li consuma insieme nello stesso passo. Non c'è mai più di un messaggio "in transito" da bufferizzare; se il canale è dentro un ciclo, questo si ripete a ogni iterazione, ma un invio/ricezione alla volta.

La ragione *decisiva*, però, non è il conteggio dei messaggi in transito: è l'**invertibilità**. Se un canale fosse una coda FIFO, `ssend` e `srecv` non potrebbero essere l'uno l'inverso dell'altro, perché agirebbero su **estremità opposte**: l'inverso di un accodamento in fondo è un'estrazione dal fondo, non la `srecv` che preleva dalla testa. Non è l'ordine in sé a rompere la simmetria: si veda `push`/`pop` (§2.5), che operano su una struttura ordinata e restano una coppia inversa esatta proprio perché toccano *entrambi* la cima. È l'asimmetria delle due estremità. Con un unico valore scambiato in rendez-vous il problema non si pone, e la dualità $\mathcal{I}(\texttt{ssend}) = \texttt{srecv}$ enunciata in §3 vale esattamente, per la stessa convenzione *zero-cleared* che regola `push`/`pop`. La rinuncia alla coda non è quindi una semplificazione di comodo, ma ciò che rende i due comandi una coppia inversa.

Due garanzie, e due momenti in cui verificarle. Su un canale servono due cose distinte: *chi* può usarlo, e *in che ordine* i due titolari lo usano. Nella teoria dei tipi di sessione le dà entrambe un unico sistema di tipi, statico. Qui le due sono separate, e non per comodità di esposizione: la prima riguarda una titolarità che *cambia durante l'esecuzione*, perché un canale si può passare, e nessuna analisi del testo può seguirla; la seconda è invece una proprietà del testo del programma, e si controlla prima di eseguirlo. Da qui i due colori: la titolarità è sorvegliata a runtime, il protocollo verificato in compilazione.

Sessione binaria, come proprietà dinamica. Un canale ha, in ogni istante, **al più due** titolari: è una *sessione binaria*. Non è richiesto un ruolo fisso (mittente o ricevente): i due titolari possono scambiarsi i ruoli nel tempo, perché con due soli titolari è sempre chiaro chi sia il partner di una data offerta $ch!/ch?$. È ciò che rende unico l'accoppiamento all'indietro (§3). Nella letteratura sui calcoli di processo è la condizione di linearità garantita dai *session type* binari [HVK98], di cui esiste la generalizzazione a più di due partecipanti (*multiparty session types* [HYC08]).

La titolarità è **stato d'esecuzione**, non una proprietà del testo del programma, e può cambiare durante l'esecuzione. Un ramo diventa titolare quando usa il canale o quando lo riceve come payload di un altro canale, e smette di esserlo quando lo spedisce a un terzo. Da questa sola regola discendono, senza casi particolari, i due modi in cui un canale si passa:

- *name passing*, nello stile del π-calcolo: un ramo crea un canale privato, lo spedisce e continua a usarlo per ricevere la risposta. Cede la titolarità spedendolo e la riprende al primo uso successivo, così i titolari finiscono per essere mittente e destinatario.
- *delega*: il canale è fra A e B, A lo passa a C e smette di usarlo; i titolari diventano B e C. Se A lo riusasse sarebbe un terzo titolare, e viene fermato. La condizione classica della delega, *chi cede smette di usarlo prima che il nuovo cominci*, non è qui un obbligo dichiarato a parte: è una conseguenza del contare al più due titolari alla volta.

La differenza rispetto alla forma *statica* del vincolo, in cui un canale appartiene agli stessi due rami per tutta la vita, non è di comodo. Leggendo il testo del programma si può contare quanti rami *nominano* un canale, ma non si può seguire un canale che cambia titolare, perché il nuovo titolare lo riceve sotto un altro nome, magari in un'altra procedura; l'analisi statica è costretta o a rifiutare la delega o a esentarla dal controllo. Osservando l'esecuzione la difficoltà non si presenta: l'identità di un canale è il canale stesso, non il nome della variabile che lo contiene.

Il vincolo non fa parte della relazione di transizione: è una condizione sull'esecuzione, la cui violazione produce $\bot$. Sono due i modi di violarla, e corrispondono ai due errori di §2.6: un terzo ramo che tocca un canale già impegnato, e lo *stallo*, cioè un ramo che attende un partner che non può più arrivare.

Tipi di sessione. Il vincolo appena dato dice *quanti* rami toccano un canale, non *come* lo usano: due rami che eseguono entrambi un `ssend` lo soddisfano, e restano

bloccati per sempre. Un **tipo di sessione** descrive la sequenza degli scambi che un ramo compie su un canale:

$$S ::= \texttt{end} \mid !k.S \mid ?k.S \mid \mu X.S \mid X$$

dove k è il numero di interi scambiati in un passo. La **dualità** scambia le due direzioni e lascia il resto:

$$\overline{\texttt{end}} = \texttt{end} \qquad \overline{!k.S} = ?k.\overline{S} \qquad \overline{?k.S} = !k.\overline{S} \qquad \overline{\mu X.S} = \mu X.\overline{S} \qquad \overline{X} = X$$

Il tipo di un ramo si ricostruisce dal programma: $!k$ per un `ssend` di k valori, $?k$ per un `srecv`, concatenazione per la sequenza, $\mu X.S.X$ per un ciclo il cui corpo ha tipo S, e il tipo del corpo per una chiamata di procedura, con il canale sostituito al parametro formale. Due rami usano bene un canale quando i loro tipi sono duali, e la dualità è **invariante per inversione**: invertire un `par` scambia `ssend` e `srecv` in entrambi i rami (§3), cioè dualizza entrambi i tipi, e due tipi duali restano duali.

Questo controllo non sostituisce quello dinamico e non lo duplica: riconosce in anticipo i conflitti già visibili nel testo, come due rami che offrono la stessa direzione o si scambiano quantità diverse di valori, che altrimenti si manifesterebbero solo come stallo a esecuzione avviata. Il frammento è minimo e va preso per quello che è: non c'è *scelta etichettata*, quindi i due rami di un `if` che comunichi devono compiere le stesse azioni sul canale, e non ci sono sessioni *multiparty*. Dove il frammento non basta a descrivere l'uso di un canale, il tipo resta indeterminato e non si segnala nulla: si riconoscono i soli conflitti dimostrabili, mai rifiutando un programma corretto per insufficienza della descrizione.

Composizione parallela: `par ... and ... rap`

I rami procedono per *interleaving* non deterministico (caso a due rami; con più `and` si generalizza). Un passo avanza il ramo sinistro o il destro:

$$\frac{\langle \sigma,\, c_1\rangle \to_c \langle \sigma',\, c_1'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma',\, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle} \quad \text{(Par-L)}$$

$$\frac{\langle \sigma,\, c_2\rangle \to_c \langle \sigma',\, c_2'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma',\, \texttt{par}\ c_1\ \texttt{and}\ c_2'\ \texttt{rap}\rangle} \quad \text{(Par-R)}$$

Quando un ramo termina (cioè è diventato `skip`), lo si elimina dal blocco:

$$\frac{}{\langle \sigma,\, \texttt{par skip and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma,\, c_2\rangle} \quad \text{(Par-L0)}$$

$$\frac{}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and skip rap}\rangle \to_c \langle \sigma,\, c_1\rangle} \quad \text{(Par-R0)}$$

(Quando resta un solo ramo, `par` c `rap` si comporta come c.) Par-L e Par-R sono CtxStmt (§2.3) istanziata sulle due clausole $C_s[\bullet] = \texttt{par}\ \bullet\ \texttt{and}\ c\ \texttt{rap}$ e $\texttt{par}\ c\ \texttt{and}\ \bullet\ \texttt{rap}$: le riportiamo per esteso perché è qui che si vede l'interleaving, ma non sono regole indipendenti. Par-L0/Par-R0 lo sono, perché riguardano la terminazione di un ramo. Se un ramo fallisce, l'intero blocco fallisce (un errore non è localizzabile a un solo ramo una volta che i rami condividono il blocco `par`), ma anche questo non richiede regole

proprie: il fallimento di un ramo risale all'intero blocco per CtxError sulle stesse due clausole (§2.6).

Par-L/Par-R (e le varianti -L0/-R0) sollevano un passo $\to_c$ **ordinario** (non etichettato) di un solo ramo. Un `ssend`/`srecv` non produce mai un passo $\to_c$ da solo (§2.5), quindi queste regole non lo sollevano mai direttamente; ma se il ramo che lo contiene è a sua volta un `par` annidato, l'offerta deve poter risalire fino al livello dove si trova il suo partner. Due regole gemelle fanno da tramite, sollevando un passo **etichettato** anziché ordinario:

$$\frac{\langle \sigma,\, c_1 \rangle \xrightarrow{\alpha} \langle \sigma,\, c_1' \rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap} \rangle \xrightarrow{\alpha} \langle \sigma,\, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap} \rangle} \quad \text{(Par-L-Lbl)}$$

$$\frac{\langle \sigma,\, c_2 \rangle \xrightarrow{\alpha} \langle \sigma,\, c_2' \rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap} \rangle \xrightarrow{\alpha} \langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2'\ \texttt{rap} \rangle} \quad \text{(Par-R-Lbl)}$$

Con queste, un'offerta di canale generata da Ssend-Lbl/ Srecv-Lbl in fondo a un `par` annidato risale, un livello di nesting alla volta, fino al primo livello in cui trova un ramo gemello con l'offerta duale sullo stesso canale, esattamente lo stile standard alla CCS. La sincronizzazione vera e propria resta un'unica regola, applicata al livello dove le due offerte sono gemelle:

$$\frac{\langle \sigma,\, c_1 \rangle \xrightarrow{ch!\langle \bar{v}, \bar{n} \rangle} \langle \sigma,\, c_1' \rangle \qquad \langle \sigma,\, c_2 \rangle \xrightarrow{ch?\langle \bar{w} \rangle} \langle \sigma,\, c_2' \rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap} \rangle \to_c \langle \sigma[\bar{v} \mapsto \bar{0}][\bar{w} \mapsto \bar{n}],\, \texttt{par}\ c_1'\ \texttt{and}\ c_2'\ \texttt{rap} \rangle} \quad \text{(Par-Comm)}$$

(simmetrica in c_1, c_2: si applica anche scambiando quale dei due offre $ch!$ e quale $ch?$; con più di due `and` si applica alla coppia di rami che offre le etichette duali). Il canale ch compare solo per far combaciare le due etichette: **non** compare nell'aggiornamento dello store, che tocca solo $\bar{v}$ (azzerate) e $\bar{w}$ (riempite con $\bar{n}$), è il passo congiunto descritto in §2.5. Par-Comm produce un passo $\to_c$ ordinario (non etichettato): una volta sincronizzato, il risultato non deve risalire oltre, è *questo* passo, non l'offerta grezza, che Par-L/ Par-R sollevano ai livelli ancora più esterni.

Par-L-Lbl/Par-R-Lbl non aprono un instradamento sbagliato. Sollevare un'etichetta attraverso un `par` annidato potrebbe in linea di principio farla incontrare un partner diverso da quello inteso, se sullo stesso canale ci fossero più di due offerte possibili in giro per l'albero dei `par`. Ma il vincolo di sessione binaria (§2.5), sorvegliato durante l'esecuzione, garantisce che in ogni istante siano **al più due** i rami (a qualunque profondità di nesting) titolari di un dato canale, e quindi al più due quelli che possono offrire un'etichetta su di esso: Par-Comm confronta le etichette per *nome di canale*, quindi se solo due rami possono mai offrirne una su ch, non c'è un terzo candidato con cui una Par-Lbl possa accoppiarla per errore, qualunque cammino di risalita la porti a incontrare l'altra offerta su ch, è per forza quella giusta. Il controllo statico (implementato in `parser.py`, non riportato qui) conta esattamente questo: le foglie concorrenti (i rami che restano dopo aver scomposto ricorsivamente ogni `par` annidato nei suoi stessi rami, non i soli rami immediati) che toccano ciascun canale, indipendentemente da quanto in profondità si trovino; è la stessa nozione di "ramo" che qui risale con Par-L-Lbl/Par-R-Lbl.

Con `par` la relazione $\to_c$ **non** è più deterministica: l'ordine d'interleaving non è fissato. Tuttavia i rami **non condividono memoria**: ciascuno opera su una porzione *disgiunta* dello store (variabili `int`/`stack` proprie). L'**unica** forma di comunicazione/sincronizzazione tra rami passa per i canali, tramite Par-Comm. Grazie a questa disgiunzione, i diversi interleaving che non toccano canali producono lo stesso store finale (confluenza).

Perché vale la confluenza. Siano $\langle \sigma, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma_1, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle$ (via Par-L) e $\langle \sigma, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma_2, \texttt{par}\ c_1\ \texttt{and}\ c_2'\ \texttt{rap}\rangle$ (via Par-R) i due passi abilitati in σ, nessuno dei due un `ssend`/`srecv` (nessun tocco a canali). Poiché c_1 legge e scrive solo le proprie variabili `int`/`stack` e c_2 solo le proprie, e questi due insiemi sono disgiunti, l'aggiornamento di σ compiuto dal passo di c_1 e quello compiuto dal passo di c_2 agiscono su porzioni disgiunte del dominio: sono *indipendenti* e commutano. Applicando prima l'uno e poi l'altro (in un ordine o nell'altro) si ottiene quindi lo stesso σ_{12}, con $\langle \sigma_1, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma_{12}, \texttt{par}\ c_1'\ \texttt{and}\ c_2'\ \texttt{rap}\rangle$ e simmetricamente da σ_2: un diamante di confluenza locale. Estendendo per induzione sulla lunghezza della derivazione (ogni permutazione di passi indipendenti è una sequenza di scambi adiacenti di questo tipo) si ottiene che due interleaving qualunque che non toccano canali portano allo stesso store finale. La clausola è necessaria: un passo Par-Comm coinvolge **entrambi** i rami contemporaneamente (§2.5), non un passo di solo c_1 sollevato da Par-L indipendentemente da un passo di solo c_2, non c'è quindi un passo-di-c_1 e un passo-di-c_2 separati da far commutare, ce n'è uno solo che li consuma entrambi in un colpo; non rientra nel caso trattato qui. (§3 tratta l'argomento, analogo ma non lo stesso, per l'accoppiamento forzato all'indietro su un canale.)

Commit / rollback: `try` ... `rollback` ... `yrt`

Esegue il corpo c in avanti, poi valuta la guardia di commit b' sullo store risultante: se b' è *vera* il corpo resta applicato (commit); se è *falsa* il corpo viene **annullato** eseguendo l'inverso $\mathcal{I}(c)$ (§3), che riporta esattamente allo store iniziale, e poi si esegue il corpo di rollback r. A differenza di `if ... fi`, le due guardie b (ingresso) e b' (uscita) non sono legate da alcun vincolo di coerenza: la decisione commit/rollback dipende solo da b' valutata a fine corpo. La guardia d'ingresso b ha quindi valore puramente documentale: non compare in alcuna premessa, né nello zucchero sintattico. Darle un ruolo semantico (una regola d'ingresso che la valuti, sul modello dell'asserzione del ciclo) è una possibile estensione, che qui non perseguiamo.

Nella terminologia standard il costrutto è una **transazione**: con corpo di rollback vuoto ($r = \texttt{skip}$) è una transazione *acida* (o si applica interamente o non lascia traccia), mentre con un corpo di rollback non vuoto è una transazione **con compensazione**, dove r è appunto l'azione compensativa eseguita dopo aver disfatto il tentativo. Il rollback in contesto concorrente è studiato in [LMSS11;LLM+13]. Diamo qui le regole in stile big-step.

$$\frac{\langle \sigma, c\rangle \to_c^* \langle \sigma', \texttt{skip}\rangle \quad \langle \sigma', b'\rangle \to_b \langle \sigma', tt\rangle}{\langle \sigma, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \langle \sigma', \texttt{skip}\rangle} \quad \text{(Try-Commit)}$$

$$\frac{\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \quad \langle \sigma',\, b'\rangle \to_b \langle \sigma',\, \mathit{ff}\rangle \quad \langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \langle \sigma,\, \texttt{skip}\rangle \quad \langle \sigma,\, r\rangle \to_c^* \langle \sigma'',\, \texttt{skip}\rangle}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \langle \sigma'',\, \texttt{skip}\rangle} \quad (\textsc{Try-Rollback})$$

$$\frac{\langle \sigma,\, c\rangle \to_c^* \bot}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \bot} \quad (\textsc{Try-Err})$$

La premessa $\langle \sigma', \mathcal{I}(c)\rangle \to_c^* \sigma$ è garantita dalla proprietà di reversibilità (§3): poiché $\langle \sigma,\, c\rangle \to_c^* \sigma'$, l'inverso del corpo riporta allo store σ *pre-corpo*. Senza clausola `rollback` si pone $r = \texttt{skip}$ (solo annullamento). È il **backtracking nativo**: il ripristino dello stato è l'inverso esatto del tentativo, senza copie/snapshot. Try-Err lascia che un errore del corpo esca dal `try` inalterato: un $\bot$ è un bug di programma, non un fallimento della guardia b', quindi non attiva il rollback.

Vincolo di buona formazione: niente canali nel corpo. Le premesse $\langle \sigma,\, c\rangle \to_c^* \sigma'$ e $\langle \sigma', \mathcal{I}(c)\rangle \to_c^* \sigma$ usano la relazione **non** etichettata: un `ssend`/`srecv` dentro c produce però solo un passo etichettato (§2.5), e l'unica regola che lo consuma, Par-Comm (§2.5), vive al livello di un `par`, sopra c, non dentro. Se c (o r) contiene un `ssend`/`srecv`, diretto o raggiunto tramite `call`/`uncall`, $\langle \sigma,\, c\rangle \to_c^* \sigma'$ non è derivabile: il `try` resta bloccato, non fallisce con $\bot$. Un programma ben formato evita quindi `ssend`/`srecv` nel corpo e nel rollback di ogni `try`, direttamente o tramite `call`/`uncall` (verificato staticamente, non nella relazione di transizione). Non è un limite che valga la pena aggirare: anche potendolo esprimere, disfare uno `srecv` già sincronizzato richiederebbe di far ripartire dal punto giusto anche il ramo mittente, che nel frattempo può aver già proseguito, un `try`/`rollback` è un costrutto a un solo ramo, e non ha modo di coordinare la cooperazione di un partner che non controlla. Non è una limitazione ad hoc: è la difficoltà nota del rollback in contesto concorrente, dove annullare un'azione già comunicata obbliga a propagare l'annullamento ai partner che l'hanno osservata[LMSS11;LLM+13].

Realizzazione (zucchero sintattico): il costrutto si riscrive con `if` e `call`/`uncall` su una procedura sintetica f_c con corpo c (parametri = variabili libere di c):

$$\texttt{call}\ f_c(\bar{v})\,;\ \texttt{if}\ b'\ \texttt{then skip else uncall}\ f_c(\bar{v})\,;\ r\ \texttt{fi}\ b'.$$

2.6 Errori a runtime

L'esito d'errore è $\bot$, ereditato dalla base insieme alla regola che lo solleva: $\Gamma'_c = \Gamma_c \cup \{\bot\}$, $\top'_c = \top_c \cup \{\bot\}$. La **propagazione** è interamente a carico della base: CtxError (§2.3) fa risalire un $\bot$ da qualunque posizione di contesto $C_s[\bullet]$: attraverso una sequenza, i rami di un condizionale, i corpi di un ciclo e i rami di un `par`. Nessun costrutto ha quindi bisogno di una propria regola di propagazione. Resta un'eccezione, Try-Err (§2.5): il corpo di un `try` non è un buco di contesto, perché le sue regole lo riducono con $\to_c^*$ in una premessa e devono conservarlo intatto per poterne eseguire l'inverso $\mathcal{I}(c)$ in caso di rollback. Lì una regola d'errore esplicita serve ancora.

Quel che resta di specifico a Kairos è **quali** costrutti possono fallire *localmente*. Sono i costrutti che Kairos aggiunge alla base (§2.3), e ciascuno ha una regola propria che produce $\bot$ invece di restare bloccato, perché nessun altro ramo potrebbe mai soddisfare

quel vincolo, trattandosi di stato **locale** al ramo (§2.5): Local-Err ($v \in \mathrm{dom}(\sigma)$) e Deloc-Err ($v \notin \mathrm{dom}(\sigma)$ oppure $\sigma(v) \neq m$) per lo scope (§2.5); Pop-Err (stack vuoto) e Pop-Err2 ($\sigma(v) \neq 0$) per lo stack (§2.5); Srecv-Err ($\sigma(\bar{w}) \neq \bar{0}$) per la ricezione su canale (§2.5). Gli errori dei costrutti *ereditati*, cioè l'asserzione incoerente nel condizionale (IfError1/IfError2) e nel ciclo (LoopError1/LoopError2), vengono invece dalla base e non sono discussi qui.

$\perp$ contro blocco. Questa distinzione è propria di Kairos, perché è la concorrenza a renderla possibile. Il blocco di sincronizzazione sui canali (§2.5) non è il fallimento di una premessa: è l'assenza di un ramo partner pronto con cui combinarsi in Par-Comm (§2.5): nessuna regola si applica finché il partner non è pronto, ma un ramo parallelo può ancora offrirsi in un momento successivo, quindi **blocco**, non $\perp$. Il criterio è dunque: se la condizione violata riguarda stato locale al ramo, nessuna evoluzione dell'ambiente potrà mai renderla vera e l'esito è $\perp$; se riguarda la disponibilità di un partner, l'esito è blocco. È per questo che Srecv-Err produce $\perp$ e non blocco: la sua premessa $\sigma(\bar{w}) \neq \bar{0}$ parla delle variabili bersaglio del ricevente, non del canale.

Violazioni della sessione binaria. Il vincolo di §2.5 è una condizione sull'esecuzione, e si viola in due modi, entrambi con esito $\perp$. Il primo è un **terzo titolare**: un ramo tocca un canale che ne ha già due. Non è blocco, perché nessuna evoluzione dell'ambiente può liberare un posto: chi cede un canale lo fa spedendolo, e l'ha già fatto o non lo farà. Il secondo è lo **stallo**: tutti i rami di un `par` sono fermi in attesa su canale. Il singolo blocco resta blocco, come detto sopra, perché un partner potrebbe ancora offrirsi; ma quando non resta alcun ramo che possa offrirsi, l'attesa non è più provvisoria e diventa $\perp$. È la differenza fra non avere *ancora* un partner e non poterne avere.

La funzione di esecuzione

$$\mathit{exec} : \mathit{Com} \times \mathit{Store} \rightharpoonup \mathit{Store} \cup \{\perp\}$$

resta perciò parziale (un ciclo può divergere, un canale può bloccarsi) ma segnala esplicitamente le violazioni di reversibilità.

Capitolo 3

Reversibilità

L'esecuzione all'indietro è data dall'operatore di *inversione* $\mathcal{I}(\cdot)$, definito per induzione sulla struttura del comando:

$$\begin{aligned}
\mathcal{I}(\texttt{skip}) &= \texttt{skip} & \mathcal{I}(c_1\,;\,c_2) &= \mathcal{I}(c_2)\,;\,\mathcal{I}(c_1) \quad \text{(ordine rovesciato)}\\
\mathcal{I}(v \mathrel{+}= e) &= v \mathrel{-}= e & \mathcal{I}(v \mathrel{-}= e) &= v \mathrel{+}= e\\
\mathcal{I}(v \mathrel{\hat{}}= e) &= v \mathrel{\hat{}}= e & \mathcal{I}(v_1 \mathrel{<\!\!=\!\!>} v_2) &= v_1 \mathrel{<\!\!=\!\!>} v_2\\
\mathcal{I}(\texttt{local}\ v = m) &= \texttt{delocal}\ v = m & \mathcal{I}(\texttt{delocal}\ v = m) &= \texttt{local}\ v = m\\
\mathcal{I}(\texttt{push}(v, s)) &= \texttt{pop}(v, s) & \mathcal{I}(\texttt{pop}(v, s)) &= \texttt{push}(v, s)\\
\mathcal{I}(\texttt{call}\ f(\bar{v})) &= \texttt{uncall}\ f(\bar{v}) & \mathcal{I}(\texttt{uncall}\ f(\bar{v})) &= \texttt{call}\ f(\bar{v})\\
\mathcal{I}(\texttt{ssend}(\langle\bar{v}\rangle, ch)) &= \texttt{srecv}(\langle\bar{v}\rangle, ch)\\
\mathcal{I}(\texttt{srecv}(\langle\bar{v}\rangle, ch)) &= \texttt{ssend}(\langle\bar{v}\rangle, ch)
\end{aligned}$$

$$\mathcal{I}(\texttt{if}\ b\ \texttt{then}\ c_t\ \texttt{else}\ c_e\ \texttt{fi}\ b') = \texttt{if}\ b'\ \texttt{then}\ \mathcal{I}(c_t)\ \texttt{else}\ \mathcal{I}(c_e)\ \texttt{fi}\ b$$

$$\mathcal{I}(\texttt{from}\ b_1\ \texttt{do}\ c_1\ \texttt{loop}\ c_2\ \texttt{until}\ b_2) = \texttt{from}\ b_2\ \texttt{do}\ \mathcal{I}(c_1)\ \texttt{loop}\ \mathcal{I}(c_2)\ \texttt{until}\ b_1$$

$$\mathcal{I}(\texttt{par}\ c_1\ \texttt{and}\ \ldots\ \texttt{and}\ c_n\ \texttt{rap}) = \texttt{par}\ \mathcal{I}(c_1)\ \texttt{and}\ \ldots\ \texttt{and}\ \mathcal{I}(c_n)\ \texttt{rap}$$

Le riscritture chiave sono lo **scambio delle guardie**: nel condizionale $b \leftrightarrow b'$, nel ciclo $b_1 \leftrightarrow b_2$. È ciò che permette all'esecuzione inversa di scegliere il ramo giusto / sapere quando fermarsi senza memorizzare la storia.

Il `par` si inverte ramo per ramo, senza riordinarli fra loro, e le due proprietà che il linguaggio già impone bastano a giustificarlo. I rami operano su porzioni disgiunte dello store (§2.5), quindi ciò che un ramo fa alle proprie variabili si inverte indipendentemente da cosa stia facendo un altro; e ogni canale ha due soli titolari (§2.5), quindi ogni appuntamento ha un unico partner possibile. Poiché $\mathcal{I}(\texttt{ssend}) = \texttt{srecv}$ e $\mathcal{I}(\texttt{srecv}) = \texttt{ssend}$, rovesciando l'ordine dei comandi dentro ciascun ramo si rovescia la successione degli appuntamenti, e i due estremi di ciascuno restano l'uno di fronte all'altro. La dimostrazione è più sotto, nel caso induttivo `par`.

L'argomento ha un'ipotesi, ed è che i titolari di ogni canale siano fissati dal testo del programma. Se un canale può viaggiare *dentro* un messaggio — $\texttt{ssend}(\langle x, ch_2\rangle, ch_1)$, la mobilità del π-calcolo — la titolarità di ch_2 non è più una proprietà del testo ma

il risultato di un appuntamento precedente su ch_1. Rovesciando l'ordine dei comandi, gli usi di ch_2 verrebbero prima dell'appuntamento che lo consegna, e i due estremi non si troverebbero più l'uno di fronte all'altro: l'esecuzione all'indietro si blocca. Su quei programmi $\mathcal{I}(\cdot)$ non è definita per il `par`, e l'inversione va ottenuta per altra via (§5.3). È una restrizione che si riconosce sul testo, e nel corpus di prova riguarda due programmi su ottantanove.

`ssend` e `srecv` sono, come ogni altra coppia della tabella, l'una l'inversa dell'altra, senza bisogno di comandi ausiliari. Questo vale grazie al vincolo di sessione binaria (§2.5): con al più due titolari del canale, la sincronizzazione Par-Comm (§2.5) ha sempre un unico partner possibile, in avanti come all'indietro, quindi scambiare `ssend` $\leftrightarrow$ `srecv` non può mai accoppiare i passi con il partner sbagliato. Il caso `par` più sotto sviluppa questo argomento.

Proprietà di reversibilità. Per ogni comando c ben formato e stati σ, σ' (senza $\bot$):

$$\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \qquad \Longleftrightarrow \qquad \langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \langle \sigma,\, \texttt{skip}\rangle.$$

Cioè $\mathcal{I}(c)$ calcola la relazione inversa di c sullo store. In particolare, escludendo `par`, $\to_c$ è deterministica e c denota una funzione parziale *iniettiva*; di conseguenza $\langle \sigma,\, c\,;\, \mathcal{I}(c)\rangle \to_c^* \sigma$ (identità sullo store).

Il determinismo non è un'osservazione a parte: comando per comando, le premesse delle sue regole (successo ed $\bot$) sono a due a due incompatibili e la loro unione copre ogni σ possibile, sono cioè una **partizione** dello spazio degli stati, quindi esattamente una regola si applica in ogni configurazione. Ad esempio per `pop`: $\sigma(v) \neq 0$ (Pop-Err2), $\sigma(v) = 0 \wedge \sigma(s) = \varepsilon$ (Pop-Err), $\sigma(v) = 0 \wedge \sigma(s) = n : \ell$ (Pop) coprono ogni σ esattamente una volta, senza sovrapposizioni. Fa eccezione solo la sincronizzazione di canale (§2.5): lì la partizione locale resta valida (Srecv-Err copre comunque esattamente il proprio caso), ma Par-Comm può temporaneamente non trovare un ramo partner pronto, lasciando nessuna regola applicabile: è il blocco di sincronizzazione, non una rottura del determinismo: nessuna configurazione ha mai *due* regole applicabili contemporaneamente.

La dimostrazione è per induzione strutturale su c (equivalentemente, sulla derivazione di $\langle \sigma,\, c\rangle \to_c^* \sigma'$): nei casi base gli assegnamenti si annullano per i vincoli $v \notin e$ (Add2$\leftrightarrow$Sub2) o sono involuzioni (`=:`, `<=>`); `push`/`pop` si annullano per la convenzione *zero-cleared*: Pop richiede in ingresso che il bersaglio sia già 0, e Push lo ristabilisce esattamente in uscita. `ssend`/`srecv` (via Par-Comm, §2.5) si annullano per la **stessa** convenzione, applicata alla coppia di rami invece che a un solo comando: la premessa $\sigma(\bar{w}) = \bar{0}$ sul lato ricevente ed effetto $\bar{w} \mapsto \bar{n}$ sono zero-cleared esattamente come Pop, e l'effetto $\bar{v} \mapsto \bar{0}$ sul lato mittente è zero-cleared esattamente come Push: è ciò che rende `ssend`; `srecv` l'identità anche sulle variabili coinvolte, non solo un'astrazione sul canale; nei casi induttivi la doppia guardia di `if`/`from` garantisce che l'esecuzione inversa ripercorra lo stesso ramo / le stesse iterazioni.

Il caso induttivo `par`: sia $\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c^n \sigma'$ una derivazione di lunghezza n. Ogni passo è o un passo Par-L/Par-L0 (risp. Par-R/Par-R0) che avanza *un solo* ramo di un passo ordinario locale (esattamente uno dei casi già trattati sopra) o un passo Par-Comm (§2.5) che avanza *entrambi* i rami insieme in una sincronizzazione

di canale. Per IH ogni passo locale del primo tipo è invertito esattamente da $\mathcal{I}(\cdot)$ applicato al comando ridotto in quel passo, indipendentemente da cosa contenga in quel momento l'altro ramo: le variabili `int`/`stack` sono disgiunte tra rami (§2.5). Per il secondo tipo, Par-Comm stessa mostra come si inverte: le sue premesse sono $\sigma(\bar{v}) = \bar{n}$ (lato mittente) e $\sigma(\bar{w}) = \bar{0}$ (lato ricevente), la conclusione è $\sigma[\bar{v} \mapsto \bar{0}][\bar{w} \mapsto \bar{n}]$ (dallo stato risultante, $\sigma(\bar{w}) = \bar{n} \neq \bar{0}$ e $\sigma(\bar{v}) = \bar{0}$ sono esattamente le premesse per rieseguire Par-Comm con mittente e ricevente scambiati, cioè con `ssend` $\leftrightarrow$ `srecv` scambiati in ciascun ramo), che riporta esattamente a σ.

Percorrendo la derivazione all'indietro passo per passo, nell'ordine esatto inverso a quello forward, e invertendo ad ogni passo ciò che lo ha compiuto (un ramo solo per Par-L/Par-R, entrambi per Par-Comm), si ricostruisce σ: questo è esattamente ciò che $\mathcal{I}($`par` c_1 `and` c_2 `rap`$) =$ `par` $\mathcal{I}(c_1)$ `and` $\mathcal{I}(c_2)$ `rap` realizza staticamente. Resta da giustificare perché l'esecuzione all'indietro trovi *sempre* questo percorso, e non un altro: non basta che un percorso corretto esista. Con più di due rami candidati su un canale un'esecuzione all'indietro potrebbe accoppiare i passi con il partner sbagliato, esattamente il tipo di errore che il vincolo di sessione binaria evita per costruzione. Con **esattamente due** rami sul canale (§2.5), e purché siano i due che il testo del programma indica — è qui che serve l'ipotesi sopra: un canale che viaggia dentro un messaggio i suoi titolari li acquisisce a tempo di esecuzione —, l'accoppiamento è **unico**, non frutto di un interleaving fortunato: a ogni istante Par-Comm può combinare solo l'offerta corrente del ramo mittente con quella del ramo ricevente, perché non ce n'è un terzo con cui confondersi: non c'è scelta di interleaving da sbagliare. Questo vale identicamente in avanti e all'indietro: invertire scambia `ssend` $\leftrightarrow$ `srecv` in ciascun ramo (sopra) e ne inverte l'ordine dei comandi ($\mathcal{I}(c_1\,;\,c_2) = \mathcal{I}(c_2)\,;\,\mathcal{I}(c_1)$), ma non introduce mai un terzo candidato partner. È il complemento all'indietro della confluenza dimostrata in §2.5 per gli interleaving forward che non toccano canali: lì la confluenza viene dalla *disgiunzione* (rami che non toccano canali operano su porzioni disgiunte dello store, i loro passi commutano); qui viene dalla **linearità** del canale (esattamente due estremità, §2.5), che rende l'unico partner possibile sempre univoco. In entrambi i casi non serve registrare la storia dell'esecuzione: la doppia guardia la evita nei casi sequenziali di `if`/`from` (sopra), la disgiunzione delle variabili la evita nel `par` forward, e la sessione binaria la evita nella sincronizzazione di canale, in avanti e all'indietro.

Il costrutto `try ... rollback` (§2.5) non è un comando reversibile primitivo: è una *struttura di controllo* che *usa* l'inversione (esegue $\mathcal{I}(c)$ per annullare il tentativo fallito). Essendo zucchero per `if` + `call`/`uncall`, il suo inverso eredita quello del condizionale con chiamate.

3.1 Quadro riassuntivo delle regole

Comando	**Effetto sullo store**	**Inverso**
$v \mathrel{+}= e$	$\sigma(v) \mathrel{+}= \mathit{eval}(e)$	$v \mathrel{-}= e$
$v \mathrel{-}= e$	$\sigma(v) \mathrel{-}= \mathit{eval}(e)$	$v \mathrel{+}= e$
$v \mathrel{\hat{}}= e$	$\sigma(v) \mathrel{\oplus}= \mathit{eval}(e)$	sé stesso
$v_1 \mathrel{<\!\!=\!\!>} v_2$	scambia v_1, v_2	sé stesso
`local` $v = m$	alloca $v = m$	`delocal` $v = m$
`delocal` $v = m$	verifica $v = m$, rimuove	`local` $v = m$
`if` b `...fi` b'	ramo per b, asserisce b'	`if` b' `...fi` b
`from` b_1 `do` c_1 `loop` c_2 `until` b_2	ciclo, traccia $c_1[c_2\, c_1]^*$	`from` b_2 `do` $\mathcal{I}(c_1)$ `loop` $\mathcal{I}(c_2)$ `until` b_1
`call` f	esegue c_f	`uncall` f
`push`(v, s)	cima s, azzera v	`pop`(v, s)
`pop`(v, s)	verifica $v = 0$, estrae cima in v	`push`(v, s)
`ssend`$(\langle \bar{v} \rangle, ch)$	(con partner pronto) azzera $\bar{v}$	`srecv`$(\langle \bar{v} \rangle, ch)$
`srecv`$(\langle \bar{w} \rangle, ch)$	(con partner pronto) verifica $\bar{w} = \bar{0}$, riceve	`ssend`$(\langle \bar{w} \rangle, ch)$
`par` c_1 `and` c_2 `rap`	interleaving di c_1, c_2 (non det.)	`par` $\mathcal{I}(c_1)$ `and` $\mathcal{I}(c_2)$ `rap`
`try` b c `rollback` r `yrt` b'	commit se b', altrimenti $\mathcal{I}(c)$ poi r	zucchero (§2.5)

Tabella 3.1: Le regole di riduzione di Kairos, raggruppate per costrutto.

Le righe `ssend`/`srecv` non riducono mai da sole: l'effetto indicato si realizza solo insieme, nella sincronizzazione congiunta Par-Comm (§2.5, §2.5).

Ogni fallimento di premessa produce $\bot$ (stato locale) o blocco (stato condiviso, canale):

Condizione di fallimento	Regola	Esito
$v \in \mathrm{dom}(\sigma)$ (`local`)	LOCAL-ERR	$\bot$
$v \notin \mathrm{dom}(\sigma)$ o $\sigma(v) \neq m$ (`delocal`)	DELOC-ERR	$\bot$
test vero, asserzione falsa (`if`)	IFERROR1	$\bot$
test falso, asserzione vera (`if`)	IFERROR2	$\bot$
$\sigma(v) = 0 \wedge \sigma(s) = \varepsilon$ (`pop`)	POP-ERR	$\bot$
$\sigma(v) \neq 0$ (`pop`)	POP-ERR2	$\bot$
$\sigma(\bar{w}) \neq \bar{0}$ (`srecv`)	SRECV-ERR	$\bot$
nessun ramo partner pronto (`ssend`/`srecv`)	–	blocco
b_1 falsa all'ingresso (`from`)	LOOPERROR1	$\bot$
b_1 vera a metà ciclo	LOOPERROR2	$\bot$
indice fuori dai limiti ($a[k]$)	IDX-ERR	$\bot$
divisore nullo (e/e', $e \,\%\, e'$)	DIV-ERR	$\bot$

Tabella 3.2: Condizioni di fallimento e loro esito: errore o blocco.

Array. L'inversione non distingue una variabile da una cella: il bersaglio è un luogo, e il luogo resta lo stesso.

$$\mathcal{I}(a[e_i] \mathrel{+}= e) = a[e_i] \mathrel{-}= e \qquad \mathcal{I}(a[e_1] \mathrel{<\!\!=\!\!>} a[e_2]) = a[e_1] \mathrel{<\!\!=\!\!>} a[e_2]$$

$$\mathcal{I}(\texttt{local}\ a[n] = m) = \texttt{delocal}\ a[n] = m \qquad \mathcal{I}(\texttt{delocal}\ a[n] = m) = \texttt{local}\ a[n] = m$$

L'indice e_i non viene invertito perché non è un comando: è un'espressione, valutata nello stesso store in entrambe le direzioni. Vale però la condizione che rende l'inversione corretta: fra il comando diretto e il suo inverso le variabili che compaiono in e_i non devono essere cambiate, ed è la stessa condizione, già richiesta per l'espressione a destra, che il vincolo statico di §2.5 garantisce.

3.2 `show` e output osservabile

Kairos dispone di un costrutto `show`(v) che stampa il valore di una variabile su un flusso esterno. **Non** fa parte del nucleo semantico della macchina reversibile e per questo è stato omesso dalle regole precedenti: serve come strumento di *osservazione esterna* (ispezione dei valori durante l'esecuzione). La transizione associata è semplicemente $\langle \sigma, \texttt{show}(v) \rangle \to_c \langle \sigma, \texttt{skip} \rangle$: lo store non viene toccato, quindi `show` è trasparente rispetto a tutte le proprietà enunciate nelle sezioni precedenti, che parlano solo di store.

Resta però la domanda su cosa succeda all'**output osservabile** quando si esegue una procedura all'indietro. La risposta non è che l'inverso di `show` sia l'identità: è che `show` è un'*involuzione*,

$$\mathcal{I}(\texttt{show}(v)) = \texttt{show}(v),$$

sintatticamente autoinversa come ^= e <=>, con la differenza che per quelle l'involuzione è anche semantica, mentre `show ; show` stampa due volte (§3). Un `uncall` non sopprime dunque le stampe: le riesegue.

Nel frammento sequenziale (senza `par`), dove $\to_c$ è deterministica (§3), l'inversione rovescia l'ordine dei comandi ($\mathcal{I}(c_1\,;\,c_2) = \mathcal{I}(c_2)\,;\,\mathcal{I}(c_1)$), quindi i `show` vengono incontrati in ordine opposto: se in avanti la procedura stampa $n_1, n_2, \ldots, n_k$, all'indietro stampa $n_k, \ldots, n_2, n_1$. È una *simmetria osservabile*, la traccia esterna che rende visibile la reversibilità.

Con `par` la simmetria è solo locale. L'inversione agisce ramo per ramo,

$$\mathcal{I}(\texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}) = \texttt{par}\ \mathcal{I}(c_1)\ \texttt{and}\ \mathcal{I}(c_2)\ \texttt{rap} \quad (\S3),$$

quindi la sequenza di stampe *di ciascun ramo* è rovesciata; ma l'ordine con cui le stampe dei rami diversi si intercalano non è determinato, perché l'interleaving non lo è (§2.5). Già `par show`(a) `and show`(b) `rap` può emettere a, b sia in avanti sia all'indietro. L'output globale non è quindi il rovesciamento esatto di quello forward: lo è solo la proiezione su ogni singolo ramo.

Resta vero, come detto sopra, che `show` non modifica lo store e non incide su nessuna delle proprietà enunciate, che parlano solo di store.

3.3 Conformità a Janus

Che Kairos contenga Janus è un'affermazione verificabile, e conviene verificarla su programmi che non abbiamo scritto noi. Il report originale del linguaggio [LD82] ne contiene tre: la radice intera, un ordinamento a bolle con permutazione e la fattorizzazione di un intero. Sono del 1982, e la loro ortografia non è quella di oggi — il commento è `;`, lo scambio `:`, la disuguaglianza `#`, il resto `\`, e le variabili sono globali invece che parametri — ma la struttura è la stessa, e tradurli è una sostituzione di simboli, non una riscrittura.

La radice intera gira in entrambe le direzioni. Consuma *num* producendo $rt = \lfloor\sqrt{num}\rfloor$; la `uncall` riporta *rt* a zero e *num* al valore iniziale. Da sola esercita buona parte di ciò che serve a un programma Janus: moltiplicazione, divisione e resto dentro le guardie (§2.1), cicli con la sola clausola `loop`, e una procedura ausiliaria invocata con `call` e `uncall` dentro lo stesso corpo.

L'ordinamento a bolle non è invertibile. È il caso interessante, perché il difetto sta nel listato del 1982 e non nella traduzione. Il suo condizionale è

```
if list[j] > list[j+1]
then list[j] <=> list[j+1]
     perm[j] <=> perm[j+1]
fi perm[j] > perm[j+1]
```

e la guardia d'uscita deve distinguere i due rami: vera dopo `then`, falsa dopo `else` (§2.5). Non lo fa. Con $list = [4, 2, 5, 1, 3]$ e *perm* identità, alla terza iterazione esterna si ha $list = [1, 2, 4, 3, 5]$ e $perm = [3, 1, 0, 4, 2]$: il confronto $list[3] > list[4]$ è $3 > 5$, falso, quindi si prende il ramo vuoto, ma $perm[3] > perm[4]$ è $4 > 2$, vero. Il condizionale non ha inverso in quel punto, perché *perm* registra da dove viene ogni elemento e non se lo scambio è appena avvenuto, e le due cose non coincidono. È il caso d'uso dell'asserzione: senza, il programma proseguirebbe e l'errore comparirebbe molto più

tardi, come un valore sbagliato alla chiusura di una variabile. Del listato resta perciò la sola procedura che costruisce la permutazione identità, che è un programma Janus valido e usa un array come parametro: la chiamata lo riempie con $0, \ldots, n-1$, la `uncall` lo riazzera. La fattorizzazione, il terzo programma, non è stata tradotta.

Ai due si aggiunge un programma scritto per l'occasione, che esercita tutte e sei le forme delle clausole facoltative di `if` e `from` (§2.1) con le asserzioni che devono reggere. I tre stanno nel corpus di verifica insieme agli altri, quindi la conformità non è un'osservazione fatta una volta ma un controllo che accompagna ogni modifica.

Capitolo 4

Programmare in Kairos: compressione lossless reversibile

4.1 Il punto di partenza

La compressione *lossless* è il banco di prova naturale per un linguaggio reversibile: comprimere e decomprimere sono per costruzione l'una l'inversa dell'altra, quindi un unico programma reversibile le realizza entrambe e non esiste il rischio che le due metà divergano.

Il lavoro da cui partiamo è quello di Lyngby et al.[LNGY24], che ha realizzato in Janus versioni reversibili di due algoritmi dello standard zip: la compressione a dizionario LZW e la trasformata di Burrows–Wheeler. Il loro obiettivo non era la sola reversibilità ma la proprietà più forte di essere *clean*, cioè non lasciare spazzatura: un algoritmo reversibile non può cancellare, quindi le strutture ausiliarie vanno azzerate invece che buttate. La tecnica con cui ci riescono è una coppia di procedure: la prima produce l'uscita insieme ai dati ausiliari, la seconda ricalcola gli stessi ausiliari partendo dall'uscita, e chiamare la prima disfacendo la seconda lascia solo l'uscita, senza storia globale.

I risultati a cui arrivano sono netti. Il loro LZW gira in $\Theta(n)$ come la migliore versione irreversibile, quindi è asintoticamente ottimo. La loro BWT, con l'ordinamento ingenuo delle rotazioni, costa $O(n^3)$. Concludono che le tecniche della programmazione reversibile reggono su problemi reali, e che restano miglioramenti *possibili e necessari*, indicando in particolare l'ordinamento delle rotazioni.

Il loro lavoro è però interamente **sequenziale**, e nel descrivere la BWT osservano che in pratica l'ingresso si divide in blocchi, ciascuno trasformato separatamente. È l'osservazione da cui partiamo: blocchi trasformati separatamente sono blocchi indipendenti, e blocchi indipendenti sono rami di un `par` che operano su porzioni disgiunte dello store, cioè l'ipotesi sotto cui vale la confluenza (Sezione 2.5). In Janus quella strada non si può percorrere, perché il linguaggio non ha concorrenza. In Kairos sì, ed è il contributo di questo capitolo.

4.2 Che cosa cambia scrivendolo in Kairos

Gli algoritmi di compressione lavorano per indice, e l'accesso per indice è diretto: `blocco[j]` legge la cella j senza consumarla e senza toccare le altre, come nel lavoro di riferimento. Il ciclo che confronta due rotazioni legge quindi il blocco al costo di una lettura, non di una passeggiata.

```
procedure car(int blocco[], int n, int off, int k, int c)
    local int j = 0
        j += off
        j += k
        if j >= n then
            j -= n
        fi j < off
        c += blocco[j]
        if j < off then
            j += n
        fi j >= n
        j -= k
        j -= off
    delocal int j = 0
```

Il carattere k-esimo della rotazione che parte da *off* sta in posizione $(\mathit{off} + k) \bmod n$. Il modulo non si scrive con l'operatore ma con un `if`, e la ragione è la reversibilità: j è una variabile locale che va riportata a zero, quindi la correzione $j \mathrel{-}= n$ deve poter essere disfatta, e per disfarla serve sapere se è avvenuta. La guardia d'uscita $j < \mathit{off}$ lo dice — dopo la correzione j è sceso sotto *off*, senza correzione no — e il secondo `if`, che è l'inverso del primo, la usa per rimettere j a posto.

4.3 Il confronto lessicografico

È il punto che il lavoro di riferimento indica come la difficoltà principale, e lo si tocca con mano. Confrontare due rotazioni significa trovare il primo carattere in cui differiscono, ma **non si può uscire in anticipo**: la guardia d'uscita di un `if` non distinguerebbe "sono entrato e non ho ancora trovato" da "non sono entrato".

La via che funziona usa un testimone. Finché le due rotazioni coincidono, il contatore `pref` vale esattamente k; dopo il corpo vale k oppure $k + 1$, in entrambi i casi $\geq k$, mentre chi non è entrato è rimasto sotto.

```
if pref == k then // siamo ancora nel prefisso comune
    if c1 > c2 then
        gt += 1
    fi gt == 1
fi pref >= k // guardia d'uscita valida
```

Una formulazione alternativa, accumulare il valore posizionale in base B della differenza fra le rotazioni, ordina anch'essa correttamente ed è pura accumulazione, ma richiede di moltiplicare per B a ogni passo: in Kairos azzerare il fattore vorrebbe una divisione, cioè sottrazioni ripetute il cui costo dipende dal valore e non dalla lunghezza, e su numeri dell'ordine di B^n non è praticabile.

4.4 I blocchi in parallelo

Ogni blocco è un ramo, con le proprie strutture; l'unico punto di contatto è il canale su cui il ramo consegna il risultato. Il payload di un `ssend` può essere uno stack intero, quindi la colonna trasformata viaggia in un colpo solo insieme al suo indice.

```
procedure lavoratore(stack blocco, stack aux, stack offs, stack scarto,
                     stack out, channel c, int n)
    local int idx = 0
        call bwt(blocco, aux, offs, scarto, out, n, idx)
        ssend(<idx, out>, c)
    delocal int idx = 0

par
    call lavoratore(b0, aux0, offs0, sc0, out0, c0, n)
and
    call lavoratore(b1, aux1, offs1, sc1, out1, c1, n)
and
    call raccoglitore(c0, c1, col0, col1, indici)
rap
```

Qui il blocco viaggia come `stack` e non come array, e la ragione è il canale: il payload di un `ssend` può essere un intero, uno stack o un canale, non un vettore. Trasferire un array richiederebbe di deciderne la sorte nel mittente — un rendez-vous consuma ciò che manda, e un array ha lunghezza fissa — ed è una scelta che non abbiamo compiuto. La versione concorrente resta perciò quella su stack.

4.5 Risultati

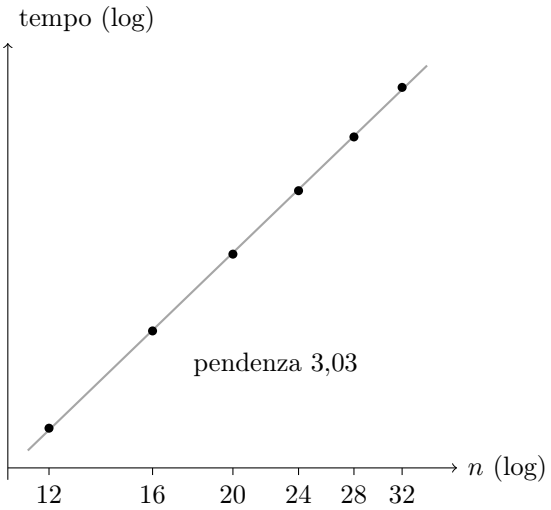
Correttezza. Sul vettore di prova del lavoro di riferimento, la stringa `banana` con alfabeto ridotto ($a = 1$, $b = 2$, $n = 3$):

blocco	`{2, 1, 3, 1, 3, 1}`	
rotazioni ordinate	`{5, 3, 1, 0, 4, 2}`	
ultima colonna	`{3, 3, 2, 1, 1, 1}`	cioè `nnbaaa`
indice	`3`	
ricostruzione da (colonna, indice)	`{1, 3, 1, 3, 1, 2}`	

Coincide con il loro esempio. La ricostruzione parte da *sola* colonna e indice, quindi è decompressione vera e non un semplice `uncall`: usa la trasformata inversa classica, che costruisce il vettore delle righe precedenti e lo percorre a partire dall'indice. Il cammino attraversa la matrice a partire dalla riga *idx*, quindi restituisce il blocco a meno di una rotazione: $\{1, 3, 1, 3, 1, 2\}$ è `banana` letta da capo dopo la `b`. Verificata anche su blocchi di dodici simboli e tre blocchi concorrenti.

Costo. Il tempo di esecuzione di un blocco, al variare della sua lunghezza n:

n	tempo (log)
12	0,500
16	1,725
20	2,692
24	3,491
28	4,168
32	4,792



Gli assi sono logaritmici, quindi l'allineamento dei punti dice che il costo è una legge di potenza e la pendenza ne dà l'esponente. La retta di regressione di $\log t$ su $\log n$ ha pendenza 3,03: il costo è n^3, lo stesso del lavoro di riferimento, e l'accesso per indice non aggiunge nulla all'ordine di grandezza. È quello che ci si aspetta dalla forma del programma: l'ordinamento a bolle compie $O(n^2)$ confronti fra rotazioni, e ciascun confronto ne scandisce $O(n)$ caratteri.

Concorrenza. Lo stesso calcolo in due varianti che differiscono in un solo punto, la procedura che orchestra i blocchi. Nella prima ogni blocco sta in un `par` a sé, insieme alla sola ricezione del proprio risultato: il lavoratore e il suo ricevente procedono insieme, ma i blocchi si susseguono, e non c'è parallelismo *fra* blocchi. Nella seconda tutti i lavoratori sono rami di un unico `par`, con il raccoglitore come ultimo ramo.

Sequenziale

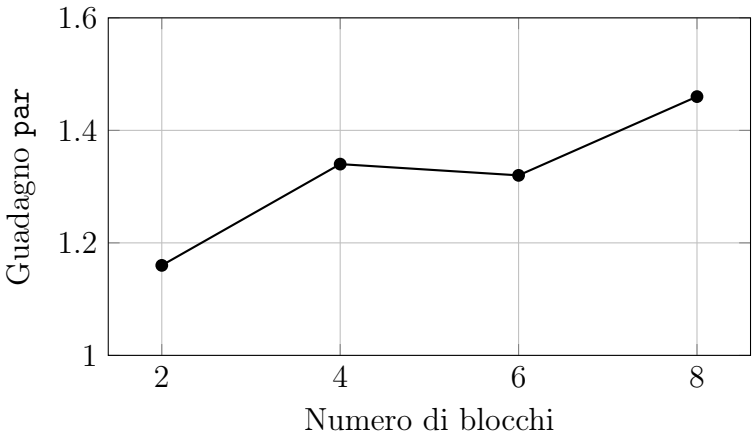
```
call bwt(b1, ..., offs1, sc1, out1, n,
     idx1)
call bwt(b2, ..., offs2, sc2, out2, n,
     idx2)
call raccoglitore(out1, out2, idx1,
    idx2, ...)
```

Parallela

```
par
    call lavoratore(b1, ..., out1, c1,
        n)
and
    call lavoratore(b2, ..., out2, c2,
        n)
and
    call raccoglitore(c1, c2, col1,
        col2, indici, n)
rap
```

Con $n = 8$, vediamo il guadagno della versione parallelizzata rispetto quella sequenziale:

blocchi	guadagno `par`
2	1,16×
4	1,34×
6	1,32×
8	1,46×



Notiamo come, aumentando il numero di blocchi, il guadagno della versione `par` tenda complessivamente ad aumentare, mostrando un maggiore sfruttamento del parallelismo disponibile.

4.6 Limiti

Tre, dichiarati. L'alfabeto delle prove è ridotto a tre simboli: nulla nel codice ne dipende, ma neppure lo esercita. I blocchi sono corti, imposti dal costo n^3 dell'ordinamento ingenuo delle rotazioni, che è il punto su cui il lavoro di riferimento stesso dichiara che i miglioramenti restano necessari. E la proprietà *clean* nel senso stretto, cioè azzerare anche l'ingresso e non solo le strutture ausiliarie, richiederebbe la coppia di procedure applicata all'intera trasformata: qui gli ausiliari si azzerano disfacendo le procedure che li hanno prodotti, l'ingresso resta.

Capitolo 5

Implementazione: interprete e DAP

5.1 La catena, dal sorgente all'esecuzione

L'implementazione è divisa in due metà, con un confine netto. La prima, in Python, porta il sorgente fino a un bytecode testuale; la seconda, in C, esegue quel bytecode nelle due direzioni.

	stadio	**prodotto**
Python	analisi lessicale	sequenza di token
	analisi sintattica	albero sintattico
	generazione	bytecode testuale
C	interprete	esecuzione, in avanti e all'indietro

Il confine sta nel bytecode, ed è un file di testo. La scelta ha un costo, che si vedrà in §5.3, e due ragioni. La prima è ispezionabilità: si può leggere ciò che la macchina esegue, e ogni riga porta il numero di riga del sorgente da cui proviene, che è quanto serve al debugger per fermarsi nel punto giusto. La seconda è che il testo basta a sé anche all'indietro: il corpo inverso di una procedura è a sua volta bytecode, emesso dallo stesso generatore e eseguito dallo stesso ciclo, e non serve una seconda macchina che lo interpreti.

5.2 Un esempio passo per passo

Si consideri una procedura che somma i primi n interi, con il ciclo a due corpi (Sezione 2.5):

```
procedure somma(int n, int s)
    local int i = 0
    from i == 0 do
        s += i
    loop
        i += 1
    until i == n
    delocal int i = n
```

Analisi lessicale. Il primo stadio riduce il testo a una sequenza di token, scartando spazi e commenti. Le parole chiave non sono espressioni regolari a sé: si riconosce un

identificatore e si guarda se compare in una tabella di riservate, così aggiungerne una costa una riga.

```
PROCEDURE ID(somma) LPAREN INT ID(n) COMMA INT ID(s) RPAREN
LOCAL INT ID(i) EQUALS NUMBER(0)
FROM ID(i) EQEQ NUMBER(0) DO
  ID(s) PLUSEQUALS ID(i)
LOOP
  ID(i) PLUSEQUALS NUMBER(1)
UNTIL ID(i) EQEQ ID(n)
DELOCAL INT ID(i) EQUALS ID(n)
```

Analisi sintattica. La grammatica è LALR(1), e ogni produzione costruisce direttamente il nodo corrispondente. Quella del ciclo mostra bene il rapporto fra le due cose:

```
statement : FROM condition DO opt_body LOOP opt_body UNTIL condition

p[0] = ('from', p[2], p[4], p[8], p.lineno(1), p.lineno(7), p[6])
// b1 c1 b2 c2
```

I due numeri di riga, quella del `from` e quella dell'`until`, servono al debugger per fermarsi sull'uno o sull'altro estremo del ciclo.

Albero sintattico. I nodi sono tuple, non oggetti, e ogni nodo porta in coda il numero di riga.

```
('procedure', 'somma', [('int','n'), ('int','s')],
 [('local', 'int', 'i', 0, 2),
  ('from', ('cond','==','i',0), // guardia d'ingresso
          [('assign','s','+=','i',4)], // corpo do
          ('cond','==','i','n'), // guardia d'uscita
          3, 7,
          [('assign','i','+=',1,6)]), // corpo loop
  ('delocal', 'int', 'i', 'n', 8)], 1)
```

Il secondo corpo sta in coda alla tupla, dopo i numeri di riga: le procedure che percorrono l'albero e leggono solo il primo corpo continuano a funzionare, e vanno estese soltanto dove il secondo conta davvero.

Bytecode. Ogni riga è un numero di riga sorgente, un opcode e i suoi argomenti.

```
@1 PROC somma
@1 PARAM int n
@1 PARAM int s
@2 LOCAL int i 0
@3 EVAL i == 0 // guardia d'ingresso
@3 ASSERTT // dev'essere vera qui (LoopError1)
@3 JMPF FROM_ERR_5
@3 JMP FROM_START_5 // entra saltando il secondo corpo
@3 LABEL FROM_BACK_5
@6 PUSHEQ i 1 // corpo loop
@3 EVAL i == 0
@3 ASSERTZ // e falsa qui (LoopError2)
@3 LABEL FROM_START_5
@4 PUSHEQ s i // corpo do
```

```
@7 EVAL i == n // guardia d'uscita
@7 JMPF FROM_BACK_5 // falsa: altra iterazione, passando dal loop
@7 LABEL FROM_END_5
@7 LABEL FROM_ERR_5
@8 DELOCAL int i n
@1 END_PROC somma
```

Il salto iniziale è ciò che realizza la traccia $c_1\,[c_2\,c_1]^*$: alla prima entrata il secondo corpo viene scavalcato, e ci si torna solo dal *back-edge*.

Le due asserzioni del ciclo (Sezione 2.5) — b_1 vera alla prima entrata, LoopError1, e falsa a ogni ri-entrata, LoopError2 — sono emesse come opcode dedicati, e abortiscono l'esecuzione quando cadono. Non sono diagnostica accessoria: è quell'invariante a rendere il ciclo invertibile per sola sintassi, perché senza di esso l'inversa `from` b_2 `do` $\mathcal{I}(c_1)$ `loop` $\mathcal{I}(c_2)$ `until` b_1 uscirebbe dopo una iterazione mentre l'originale ne compie n. L'implementazione di riferimento di Janus adotta la stessa scelta: `assertTrue` prima della prima iterazione e `assertFalse` dopo ogni corpo *loop* [YG07].

L'inverso di una procedura si ottiene in un modo solo, ed è quello della Sezione 2.5: si applica $\mathcal{I}(\cdot)$ all'albero sintattico e si emette un secondo corpo, dentro la stessa procedura, che l'esecuzione diretta scavalca con un salto e a cui `uncall` entra per etichetta. Da lì in poi `uncall` è una normale esecuzione in avanti: stesso ciclo, stesso frame, stessi parametri, e nessuna macchina separata che percorra il programma al contrario cercando di dedurne la struttura.

Il `par` non fa eccezione: il corpo inverso di una procedura che ne contiene uno è `par` $\mathcal{I}(c_1)$ `and` $\ldots$ `and` $\mathcal{I}(c_n)$ `rap`, cioè la regola del Capitolo 3 applicata alla lettera — rovesciando l'ordine dentro ogni ramo si rovescia la successione degli appuntamenti, e i due estremi di ciascuno restano l'uno di fronte all'altro. Vale a una condizione, ed è la stessa da cui viene l'unicità dell'accoppiamento: che i titolari di ogni canale siano fissati dal testo. Se un canale viaggia *dentro* un messaggio — `ssend`$(\langle x, ch_2\rangle, ch_1)$, la mobilità del π-calcolo — la titolarità di ch_2 nasce invece da un appuntamento precedente su ch_1, e rovesciando l'ordine dei comandi gli usi di ch_2 verrebbero prima dell'appuntamento che lo consegna. Non è un timore teorico: dei programmi di prova due lo fanno, e con il corpo inverso emesso comunque si fermano entrambi in stallo, ogni ramo in attesa su un canale di cui nessun altro è ancora titolare. Il compilatore riconosce quindi il caso, e lo riconosce sul programma intero perché un ramo può limitarsi a chiamare la procedura che trasmette; per quei due programmi su ottantanove non emette corpo inverso, e l'inversione resta affidata al motore dinamico, che accoppia gli appuntamenti mentre esegue invece di prevederli. Per tutti gli altri, `par` inclusi, l'inverso è compilato.

Sta dentro la stessa procedura, e non in una nuova, perché così ne condivide la tabella dei nomi: l'ordine in cui gli indici vengono assegnati, che il dump finale rende osservabile, resta quello dell'esecuzione diretta.

5.3 Strutture e ciclo di esecuzione

Strutture. Lo store σ della semantica è un `Frame`: un vettore di variabili, ciascuna con tipo, valore e, se è uno stack o un canale, la struttura associata. Una configurazione $\langle \sigma, c\rangle$ è la coppia frame corrente e puntatore alla riga di bytecode.

La tabella che traduce i nomi in indici non sta nel frame ma nella *procedura*, ed è condivisa da tutte le sue attivazioni. La distinzione conta: un indice denota allora lo stesso nome ovunque, mentre una tabella per attivazione lo legherebbe al momento in cui l'attivazione è nata, e lo stesso indice potrebbe indicare variabili diverse in due istanze della stessa procedura.

La traduzione nome-indice è una scansione lineare, con confronto sul primo carattere prima di quello completo. Il numero di nomi di una procedura è dell'ordine delle decine, e una ricerca ne tocca in media cinque, su poche linee di cache che restano calde fra un'istruzione e la successiva: a questa scala la scansione costa meno di un indice associativo, che andrebbe costruito e mantenuto e i cui accessi sono sparsi.

La risoluzione non viene però ripetuta. L'indice di un nome è una costante del programma — la tabella appartiene alla procedura ed è a sola aggiunta, quindi un indice assegnato non cambia — e viene perciò memorizzato sull'istruzione che lo usa: è la tecnica che gli interpreti chiamano *quickening* [Bru10;Sha21], e a differenza di un indice associativo non costruisce alcuna struttura. Vale un vincolo, e non è formale: risolvere un nome lo *inserisce*, e l'ordine di inserimento è osservabile nell'ordine del dump finale e nel conteggio delle celle. La cache si riempie dunque *dopo* la risoluzione, nel punto in cui l'interprete la compiva comunque, e accoglie solo i risultati positivi: un nome assente in questo istante può comparire più tardi per una dichiarazione locale in un'altra attivazione, e memorizzare un fallimento darebbe una risposta stantia. Un operando che comincia per cifra non è un nome — lo garantisce il lessico — e per esso non si esegue alcuna ricerca. Sul programma di compressione resta una ricerca di nome ogni quattro istruzioni e mezza eseguite: le altre trovano l'indice già scritto accanto a sé e non cercano affatto.

Nessuna struttura ha una dimensione fissata in anticipo: non la profondità di ricorsione, non il numero di rami di un `par`, non i parametri di una procedura, non i nomi che un frame può contenere né la lunghezza di ciascuno. Tutte crescono su richiesta, e l'unico limite che resta, sulla lunghezza di una riga di bytecode, è verificato e produce un errore esplicito invece di troncare. Un frame è così duecento byte: il vettore delle variabili e i riferimenti alla tabella dei nomi della sua procedura. Di frame ne esiste uno per procedura, non uno per attivazione: la ricorsione non ne duplica la struttura. Un nome aperto da `local` in un'attivazione annidata prende lo slot dell'omonimo dell'attivazione esterna e lo tiene da parte; il `delocal` che lo chiude rimette al suo posto quello esterno. È l'ombreggiamento della semantica (§2.5), letto direttamente nello store invece di essere imitato con una copia. Ciò che distingue un'attivazione dall'altra è allora solo quel poco che la macchina già salva sulla pila delle chiamate — i parametri legati e la pila dei nomi aperti — e per questo un secondo `local` sullo stesso nome nella *stessa* attivazione resta l'errore che la semantica prescrive, mentre fra attivazioni diverse è ombreggiamento.

Ciclo principale. Il testo del bytecode non cambia mai durante l'esecuzione: è una sola istanza condivisa da tutti i rami, e una variante di compilazione che la mappa in sola lettura lo verifica. Tutto ciò che se ne ricava è dunque una costante, e va calcolato una volta. L'interprete lavora perciò su un vettore di istruzioni *decodificate*, in cui l'indice di un'istruzione è il suo numero di riga meno uno: passare da un numero di riga a un'istruzione è dunque una sottrazione, non una scansione. La decodifica è pigra per procedura — una procedura mai chiamata non viene mai decodificata —

e le posizioni degli operandi sono scostamenti, non puntatori, perché i rami paralleli possono lavorare su copie del programma.

```
const Insn *in = &g_ir[ip]; /* indice = riga - 1 */
uint8_t op_tag = in->op; /* classificato una volta sola */
...
if (op_tag == INVOP_PUSHEQ) op_pusheq_ir(vm, cur_fi, in, ptr);
else if (op_tag == INVOP_MINEQ) op_mineq_ir (vm, cur_fi, in, ptr);
else if (op_tag == INVOP_XOREQ) op_xoreq_ir (vm, cur_fi, in, ptr);
```

La classificazione dal testo resta come ripiego, per le righe che l'array non copre; è uno `switch` sul primo carattere dell'opcode seguito dai soli confronti che quel carattere lascia possibili, quindi il suo costo non cresce col numero di opcode del linguaggio.

L'inversione non richiede alcuna storia dei dati, ed è il punto in cui la reversibilità del linguaggio si paga da sola: l'inverso di un comando è *un altro comando*, e $\mathcal{I}(\cdot)$ lo calcola dall'albero sintattico. Il corpo inverso è quindi bytecode come l'altro, prodotto una volta in compilazione, ed eseguito dallo stesso ciclo (§2.5). Nessuna struttura di controllo va ricostruita a tempo di esecuzione: dove comincia un ciclo e quale ramo è stato preso sono domande che il corpo inverso non pone, perché la risposta è già scritta nella sua forma. È l'invariante del ciclo a renderlo possibile, ed è la ragione per cui le asserzioni non sono diagnostica accessoria.

5.4 Il parallelismo

Un blocco `par` genera un thread di sistema per ramo. I rami non condividono memoria per costruzione (Sezione 2.5), quindi non serve alcun lock sullo store; l'unica primitiva di sincronizzazione è il canale, realizzato con un mutex e due code di attesa, una per i mittenti e una per i riceventi. Chi arriva per primo si accoda e si blocca, chi arriva per secondo trova il partner, lo sveglia, e lo scambio avviene: è il rendez-vous di Par-Comm.

Non condividere lo store non basta però a rendere i rami indipendenti: lo diventano solo se non condividono nulla *nemmeno nell'esecutore*. Quattro strutture stanno sul cammino di ogni ramo, e ciascuna è per thread o si tocca con operazioni atomiche: lo stato della tokenizzazione; il rilevatore di scritture concorrenti su una cella intera, che vive nella cella e non in un lock unico; l'indice dei nomi, che si consulta senza esclusione perché è a sola aggiunta e pubblica ogni voce solo quando è completa, e prende il lock soltanto per inserire; e la modalità di soppressione dell'output durante il replay in avanti, che è una proprietà dell'esecuzione di un singolo ramo.

Il criterio che le accomuna non è che tocchino i dati, visto che l'ultima non li tocca, ma che una copia sola condivisa fra i rami renderebbe il risultato dipendente dall'interleaving. Per lo store lo vieta il linguaggio; per le strutture di servizio dell'esecutore deve provvedere l'implementazione, e la confluenza vale sull'osservabile solo se provvede a tutte.

Del bytecode esiste una sola istanza, che tutti i rami leggono. È possibile perché il testo del programma porta soltanto istruzioni: ogni stato vive nei frame, quindi nessuno ha ragione di scrivere nel programma, e ciò che serve a delimitare una riga (dove comincia e quanto è lunga) sta nell'indice costruito dopo la normalizzazione. Chi deve tokenizzare

copia la riga in un buffer proprio, perché la tokenizzazione della libreria C sostituisce i separatori con terminatori, e quella copia è dell'ordine delle decine di byte invece che dell'intero programma.

La proprietà è verificata e non solo dichiarata: una modalità di collaudo mette il programma normalizzato in una regione di memoria protetta in sola lettura, e un'eventuale scrittura fa terminare il processo nel punto esatto in cui avviene. Sull'intero insieme dei programmi di prova non se ne verifica nessuna.

5.5 Il debugger DAP

L'interprete espone un'interfaccia di debug, e sopra di essa un'estensione per Visual Studio Code[1] parla il Debug Adapter Protocol, il protocollo con cui l'editor dialoga con un debugger qualunque. Si mettono punti di interruzione sul sorgente Kairos, si percorre l'esecuzione in avanti e all'indietro un passo per volta o fino al punto di interruzione successivo, si ispezionano le variabili mentre cambiano e si legge l'output del programma nella console di debug.

Il collegamento fra i due lati è ciò che il bytecode testuale rende immediato: ogni riga porta il numero della riga sorgente da cui proviene (§5.3), quindi un punto di interruzione posto sul sorgente diventa una condizione sulle righe eseguite, senza una tabella di corrispondenza da costruire e tenere allineata. Lo stato mostrato all'editor è lo stesso store dell'esecuzione, serializzato quando l'editor lo chiede: non una copia tenuta aggiornata in parallelo, ma il frame corrente letto in quel momento.

Il movimento all'indietro è ciò che distingue questo debugger da uno ordinario, e non è un'aggiunta all'interfaccia: un passo indietro esegue l'*inversa* dell'ultima istruzione, cioè la stessa direzione di esecuzione che il linguaggio possiede. L'inverso di un'operazione sui dati è un'altra operazione sui dati, e le istruzioni di controllo non toccano lo store, quindi all'indietro sono solo posizione. Un debugger a viaggio nel tempo in un linguaggio ordinario deve registrare e riprodurre; qui non c'è nulla da registrare, perché l'informazione per tornare indietro è già nello stato.

Un vincolo di realizzazione merita di essere detto, perché non è accidentale: il passo indietro avviene dentro il ciclo dell'interprete e non nello strato che parla il protocollo. Il puntatore all'istruzione corrente vive nel primo, e modificare lo store dal secondo lo lascerebbe fermo dov'era, con posizione e memoria che raccontano due storie diverse. Attraversare il confine di una chiamata è il caso che resta fuori: richiede di ricostruire il record di attivazione, che non è parte dello store ma della macchina che lo esegue.

[1] https://github.com/nicologiuliani6/kairos-vscode-debugger

Capitolo 6

Lavori correlati

Il lavoro più vicino per obiettivi è una precedente estensione concorrente di Janus [Sam19], che aggiunge al linguaggio un costrutto `fork ... and ... join` e uno scambio di messaggi, e li realizza con un compilatore verso C++ che usa i thread POSIX. Il confronto è utile proprio perché il problema di partenza è lo stesso, e ogni scelta diversa si lega a una proprietà diversa.

La prima differenza è di metodo: là i costrutti sono definiti dal codice in cui vengono tradotti, qui da regole di riduzione. Avere la semantica non è un ornamento: è ciò che permette di enunciare la confluenza e la reversibilità come proprietà del linguaggio, e non come comportamento osservato di una particolare traduzione.

Le altre due differenze riguardano lo stato e i canali, e sono legate fra loro. Quel lavoro mantiene deliberatamente uno *stato globale unico*, per non dover trattare allocazione e deallocazione nei rami concorrenti, e introduce le `struct` proprio per far arrivare dati ai thread. Kairos fa l'opposto e pretende che i rami tocchino porzioni disgiunte dello store: è una restrizione, ma è l'ipotesi da cui discende la confluenza degli interleaving (§2.5), che con memoria condivisa non sarebbe dimostrabile. Allo stesso modo, là i canali sono *porte*, cioè code di messaggi con una capacità massima; qui la comunicazione è un rendez-vous senza buffer. Anche questa è una rinuncia, ed è quella che rende `ssend` e `srecv` una coppia inversa esatta: su una coda i due agirebbero su estremità opposte, e l'inverso di un accodamento in fondo non sarebbe una ricezione dalla testa (§2.5).

Fra i linguaggi reversibili e concorrenti il più vicino per costruzione è invece **CRIL** [OY24], che però lavora a un livello di astrazione diverso: è un *linguaggio intermedio*, non un linguaggio in cui si scrive. Un programma CRIL è una collezione di *basic block*, ciascuno costituito da una singola istruzione a tre indirizzi con etichette per il flusso di controllo in avanti e all'indietro. CRIL estende RIL di Mogensen permettendo l'esecuzione concorrente di più chiamate di basic block in entrambe le direzioni, e la sua semantica operazionale soddisfa *causal safety* e *causal liveness* come criteri di correttezza per la reversibilità.

Le differenze rispetto a Kairos sono sistematiche, e riguardano il livello di astrazione prima ancora dei costrutti:

	CRIL [OY24]	**Kairos**
Livello	intermedio (basic block, 3 indirizzi)	alto, strutturato (stile Janus)
Concorrenza	chiamate concorrenti di blocchi	`par` ... `and` ... `rap` esplicito
Comunicazione	variabili condivise	canali sincroni, store *disgiunti*
Correttezza	causal safety / causal liveness	reversibilità (§3), confluenza (§2.5)

Tabella 6.1: Kairos e CRIL a confronto.

La scelta di comunicare esclusivamente per canali su porzioni disgiunte dello store, invece che per variabili condivise, non è incidentale: è precisamente l'ipotesi di disgiunzione da cui discende la confluenza degli interleaving che non toccano canali (§2.5). Un confronto puntuale fra i due criteri di correttezza (causal safety/liveness da un lato, la proprietà di reversibilità di §3 dall'altro) resta da svolgere.

Sulle **transazioni** il termine di paragone è invece la linea di lavori su $\rho\pi$ e croll-π [LMSS11;LLM+13], dove il ritorno all'indietro è governato da primitive esplicite e produce esattamente il comportamento di una transazione con compensazione. La differenza sta in dove si ferma la propagazione. In quei calcoli un rollback si propaga fra i processi: se un processo che deve essere disfatto ha nel frattempo comunicato con un altro, anche l'altro va disfatto, e il meccanismo che governa questa propagazione è buona parte della loro difficoltà. Kairos evita il problema per costruzione, vietando comunicazione e concorrenza dentro il corpo di un `try` ... `rollback` (§2.5): una transazione è locale a un ramo, e non c'è nulla da propagare. È una restrizione forte, ed è anche la ragione per cui le nostre transazioni si riducono a zucchero sintattico su costrutti già presenti; toglierla significherebbe affrontare la propagazione del rollback, che è il contributo proprio di quei lavori.

Capitolo 7

Conclusioni e lavori futuri

Abbiamo presentato Kairos, un linguaggio imperativo strutturato che è insieme reversibile e concorrente, e ne abbiamo dato la semantica operazionale strutturale in stile small-step, definita come estensione di quella di Janus di Lami, Lanese e Stefani. Ciò che Kairos aggiunge alla base sono i costrutti che servono a un linguaggio concorrente e a un linguaggio con dati: composizione parallela, comunicazione sincrona su canali tipizzati, transazioni con compensazione, stack, parametri di procedura e ambito locale. Abbiamo definito l'operatore di inversione, sintattico e calcolabile in tempo lineare, e mostrato che l'esecuzione all'indietro di un blocco parallelo è ben definita: la disgiunzione delle porzioni di store toccate dai rami dà la confluenza degli interleaving che non comunicano, e il vincolo di sessione binaria rende unico l'accoppiamento fra un invio e la ricezione che lo consuma. Quel vincolo è sorvegliato come proprietà dell'esecuzione e non del testo, il che consente la delega, cioè il passaggio di un canale da un titolare a un altro. Il linguaggio è realizzato da un interprete, e su di esso abbiamo condotto un caso di studio: la trasformata di Burrows–Wheeler reversibile, con i blocchi trasformati in parallelo, verificata sul vettore di prova del lavoro da cui parte e misurata.

Maturità pratica. Il divario più netto è fra ciò che la teoria dice possibile e ciò che servirebbe a un compressore vero, e i due algoritmi dello standard zip lo mostrano in modo opposto. La compressione a dizionario regge: nella versione reversibile di riferimento gira in $\Theta(n)$, come la migliore irreversibile, quindi la reversibilità non costa nulla in complessità. La trasformata no: costa $O(n^3)$, ed è l'esponente che misuriamo. Resta però $O(n^3)$, cioè l'ordinamento ingenuo delle rotazioni, che è il punto su cui il lavoro di riferimento stesso dichiara che i miglioramenti sono possibili e necessari: un ordinamento migliore è la direzione che indicheremmo per prima. Finché manca, i blocchi restano corti, e con blocchi corti la trasformata non ha il materiale su cui produrre l'effetto per cui esiste. Manca inoltre la proprietà *clean* nel senso stretto: le strutture ausiliarie si azzerano disfacendo le procedure che le hanno prodotte, ma l'ingresso sopravvive all'uscita, mentre un compressore deve consumarlo.

Il linguaggio. La disciplina di sessione è parziale per costruzione: mancando la scelta etichettata, i due rami di un `if` che comunichi devono compiere le stesse azioni sul canale, e dove il frammento non basta il tipo resta indeterminato. Aggiungere la scelta renderebbe totale ciò che oggi riconosce i soli conflitti dimostrabili; i protocolli

multiparty estenderebbero il tutto a canali con più di due partecipanti. Più in generale, l'assenza di corse critiche è qui un'ipotesi sotto cui valgono le proprietà, non un teorema: darle la forma di un sistema di tipi, separato dal calcolo, è il modo consueto di ottenerla e resta da fare. La comunicazione asincrona è un'altra estensione naturale, con l'avvertenza che non è un'aggiunta indolore: cambia ciò che si può dimostrare sull'inversione, perché un messaggio in transito è stato che il rendez-vous non produce.

L'implementazione. Sul calcolo concorrente la misura dice che il parallelismo si realizza, con l'ottantanove per cento del massimo che la macchina di prova rende disponibile, e indica nella frazione sequenziale del programma, cioè nel raccoglitore che riceve dai canali uno per volta, il candidato per la parte restante. Non dice però fin dove il guadagno cresca: servono una macchina con più nuclei e un raccoglitore ad albero, e sono due esperimenti, non due congetture.

Nella direzione opposta, quella in cui l'interprete aderisce al modello, ciò che la semantica descrive come una sola istanza è una sola istanza anche nell'esecutore. Il testo del programma è condiviso, perché porta solo istruzioni e nessuno stato; l'inversione lo è a sua volta, perché il corpo inverso è bytecode prodotto in compilazione e `uncall` non costruisce alcun contesto temporaneo; e lo store è una sola istanza anche lui. Un frame appartiene alla procedura, non all'attivazione: la ricorsione non ne duplica la struttura, e i nomi che `local` apre in un'attivazione ombreggiano gli omonimi di quelle esterne, come nella semantica. Ciò che resta separato è separato perché il linguaggio lo prescrive: i rami di un `par` operano su porzioni disgiunte dello store, e lì la separazione non è una copia di comodo.

Resta una sola eccezione, e non è una scorciatoia dell'esecutore ma l'ipotesi sotto cui $\mathcal{I}(\texttt{par})$ è definita: quando un canale viaggia dentro un messaggio i suoi titolari non sono più fissati dal testo, e il corpo inverso di un `par` non si calcola in compilazione. Sono due programmi su ottantanove, e per quei due l'inversione resta affidata al motore dinamico, che accoppia gli appuntamenti mentre esegue invece di prevederli.

Il confronto che manca. Il criterio di correttezza adottato qui è la reversibilità con la confluenza che ne discende; la letteratura sui linguaggi reversibili concorrenti usa invece *causal safety* e *causal liveness*. Le due formulazioni parlano dello stesso fenomeno da posizioni diverse, e metterle a confronto in modo puntuale, mostrando quale implichi quale e sotto quali ipotesi, chiarirebbe dove si colloca la scelta di far comunicare i rami soltanto per canali su porzioni disgiunte dello store.

Reversibilità e concorrenza si studiano di solito separate: la prima nei linguaggi di programmazione, la seconda nei calcoli di processo, con poche occasioni di incontro. Kairos è un tentativo di averle insieme in un linguaggio in cui si scrivono programmi, e il caso di studio è la verifica che l'accostamento produca qualcosa: un unico programma che comprime e decomprime, su blocchi trasformati in parallelo, è una cosa che nessuno dei due filoni ottiene da solo.

Riferimenti bibliografici

[Bru10] Stefan Brunthaler. Inline caching meets quickening. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 429–451. Springer, 2010.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems (ESOP 1998)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 273–284. ACM, 2008.

[LD82] Christopher Lutz and Howard Derby. JANUS: A time-reversible language. Technical report, California Institute of Technology, 1982. Descrizione del linguaggio e programmi di esempio, trasmessa a R. Landauer nel 1986.

[LLM+13] Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Concurrent flexible reversibility. In *Programming Languages and Systems (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pages 370–390. Springer, 2013.

[LLS24] Pietro Lami, Ivan Lanese, and Jean-Bernard Stefani. A small-step semantics for janus. In *Reversible Computation (RC 2024)*, volume 14680 of *Lecture Notes in Computer Science*, pages 105–123. Springer, 2024.

[LMSS11] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling reversibility in higher-order pi. In *CONCUR 2011 – Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[LNGY24] Thomas Lyngby, Rasmus R. Nylandsted, Robert Glück, and Tetsuo Yokoyama. Towards clean reversible lossless compression: A reversible programming experiment with zip. In *Reversible Computation (RC 2024)*, volume 14680 of *Lecture Notes in Computer Science*, pages 94–102. Springer, 2024.

[LV26] Ivan Lanese and Germán Vidal. A reversible semantics for janus. *CoRR*, abs/2602.16913, 2026.

[OY24] Shunya Oguchi and Shoji Yuen. Concurrent RSSA for CRIL: Flow analysis for a concurrent reversible programming language. In *Reversible Computa-*

tion (RC 2024), volume 14680 of *Lecture Notes in Computer Science*, pages 181–200. Springer, 2024.

[Sam19] Andrea Sammarchi. Il linguaggio janus concorrente: definizione, compilazione e sperimentazione. Tesi di laurea in informatica, Alma Mater Studiorum, Università di Bologna, 2019.

[Sha21] Mark Shannon. PEP 659 – specializing adaptive interpreter. Python Enhancement Proposal, 2021.

[YG07] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007)*, pages 144–153. ACM, 2007.

Ringraziamenti

[ringraziamenti]