Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Scienze dell'Informazione

Un linguaggio distribuito con agenti mobili: analisi progettuale e implementazione

Tesi di Laurea

di

Luca Padovani

Relatore

Chiar.mo Prof.

Andrea Asperti

Parole chiave: agenti, mobilità, distribuzione, concorrenza, join-calculus.

Sessione III

Anno Accademico 1996-97

Introduzione

Circa a metà degli anni '60 si osservò che un complesso linguaggio di programmazione poteva essere studiato in termini di un piccolo linguaggio "nucleo" che fosse caratteristico per un certo stile di programmazione. L'intento era quello di "capire" i linguaggi studiando il modo in cui costrutti complessi potessero essere interpretati nei termini del linguaggio "nucleo".

Il linguaggio "nucleo" (o *calcolo*) più diffuso, anche per l'avvento della cosiddetta "semantica denotazionale" [Sto77], era quello detto λ-calculus [Bar91], un formalismo introdotto da Alonzo Church circa 30 anni prima, in cui tutto si riduce alla definizione ed all'applicazione di funzioni. Il λ-calculus, oltre a poter essere visto come un semplice linguaggio in cui tradurre costrutti di programmazione complessi, gode di una elegante e profonda teoria matematica (esso nasce con l'intento di catturare matematicamente la nozione di "calcolabilità effettiva"); questa natura formale/matematica, unita alla sua semplicità ed eleganza concettuale, è stata, ed è tuttora, il suo punto di forza.

Ma da allora lo scenario dell'"informazione automatica" è radicalmente cambiato: non più linguaggi di programmazione a basso livello, ma astrazioni e oggetti, non più computazioni sequenziali, ma concorrenti, non più sistemi centralizzati, ma distribuiti. Il mondo dei calcolatori è stato investito da più "rivoluzioni" dall'effetto devastante, non ultima la cosiddetta "connettività globale" che, attraverso la rete Internet, realizza una sorta di "arena" in cui tutti i calcolatori, grandi e piccoli, di qualsiasi architettura, di qualsiasi marca, si scambiano informazioni e interagiscono tra loro.

Il software, ed il linguaggi di programmazione in particolare, hanno cominciato ad assumere un ruolo sempre più predominante a causa della complessità delle nuove problematiche e del bisogno di astrarre le caratteristiche delle singole piattaforme. Ai ricercatori nell'area dei calcoli fondazionali parve subito chiaro che il "nucleo" per i nuovi linguaggi di programmazione *non poteva*

essere una semplice estensione del λ-calculus, troppo legato al concetto di computazione sequenziale. Malgrado questa semplice intuizione, non era affatto facile immaginare un corrispondente "distribuito" del λ-calculus, tant'è che numerosi sono stati i calcoli proposti e ancora oggi nessuno si è imposto come modello di riferimento. Addirittura, sono ancora oggetto di discussione le aree che questi calcoli devono essere in grado di "catturare".

Uno tra i calcoli proposti, il join-calculus [FGL$^+$96], risulta essere particolarmente attraente per diversi motivi: intanto per la semplicità e l'immediatezza del modello da cui deriva (il modello chimico, o CHAM [FG96]), in secondo luogo per il fatto che, a dispetto della sua semplicità, presenta (a modo suo, indubbiamente) sfaccettature ad alto livello, tra cui uno stile ad oggetti.

Il problema del join-calculus, comune anche agli altri calcoli per la concorrenza, è che ancora manca di riscontri tangibili ed esaurienti. Da una parte abbiamo la concretezza della tecnologia Java [F$^+$97] che, tra accordi e conflitti, dilaga in tutti gli angoli della rete, mentre dall'altra abbiamo l'astrazione e la semplicità dei calcoli, ancora non del tutto esplorati. Mancano strumenti per verificare l'utilità dei calcoli, mancano studi sui problemi implementativi, mancano riscontri con le reali necessità degli utenti. È in questa ottica che abbiamo realizzato il nostro prototipo, nella speranza di capire alcuni dei problemi legati all'implementazione e cercando di apportare un seppur infinitesimale contributo al progetto globale di realizzare *la* macchina astratta per la concorrenza ed i sistemi distribuiti.

Il linguaggio che abbiamo utilizzato è già stato implementato pochi mesi fa da un'équipe di ricercatori francesi nell'ambito del progetto CONFER. Ci siamo appoggiati ad un linguaggio esistente sostanzialmente perché l'abbiamo ritenuto adeguato e perché progettare un linguaggio *ex novo* è un compito certamente non facile e che richiede tempo. Comunque, non abbiamo mancato di modificare o fare aggiunte al linguaggio al fine di renderlo più adatto ad una implementazione da portare a termine in pochi mesi e per sperimentare costrutti "insoliti". In particolare, il sistema dei tipi è stato abbondantemente trascurato, con la riserva di renderlo finalmente completo ed espressivo in un'eventuale sviluppo futuro.

Seguono una breve organizzazione dei capitoli principali e gli immancabili ringraziamenti.

Capitolo 1. Il linguaggio Viene descritto il linguaggio implementato attra-

verso un breve *tutorial* che ne mette in evidenza gli aspetti fondamentali.

Capitolo 2. La macchina virtuale Viene presentata la macchina virtuale progettata e realizzata per l'implementazione. Ogni costrutto del linguaggio viene "tradotto" in opportune sequenze di istruzioni.

Capitolo 3. Implementazione della macchina virtuale Se la macchina virtuale è "astratta" per definizione, la sua implementazione si basa invece su un complesso programma scritto in C descritto in questo capitolo.

Capitolo 4. Descrizione formale del linguaggio Il linguaggio, introdotto in modo informale nel primo capitolo, viene qui descritto minuziosamente e in modo rigoroso.

Capitolo 5. Il compilatore Il capitolo propone una veloce panoramica sulla struttura del compilatore, evidenziando gli aspetti meno comuni nel progetto di compilatori per gli usuali linguaggi di programmazione.

Capitolo 6. Conclusioni Vengono presentati alcuni confronti con l'implementazione "francese" ed anche un esempio più complesso di programma scritto nel linguaggio implementato, nonché possibili sviluppi futuri.

Infine, nelle appendici vengono presentate una breve introduzione al join-calculus ed una piccola guida di riferimento alla libreria POSIX thread usata nell'implementazione.

Ringraziamenti

Questa tesi è il risultato dell'aiuto e della collaborazione di un numero incalcolabile di persone, prime fra tutte i miei genitori, che mi hanno permesso di intraprendere la strada dello studio all'università.

Devo un ringraziamento particolare ad Andrea Asperti e a tutte le persone che hanno contribuito alla mia formazione di studente. Farei un torto se non ringraziassi anche tutti gli amici, tra cui desidero ricordare (in ordine alfabetico) Alberto, Mauro, Roberto e Stefania.

Indice

Introduzione **i**

Indice **v**

Elenco delle figure **ix**

Elenco delle tabelle **xi**

1 Introduzione al linguaggio **1**

1.1 Fondamenti 2

1.1.1 Processi ed espressioni 2

1.1.2 Definizione di nomi 4

1.2 Canali 5

1.2.1 Canali asincroni 5

1.2.2 Canali sincroni 7

1.2.3 Definizioni ricorsive 9

1.2.4 Primitive 10

1.3 Join pattern 10

1.4 Codifica delle strutture di controllo 12

1.4.1 Codifica del costrutto *join let* 12

1.4.2 Cicli asincroni 13

1.4.3 Cicli sincroni 14

1.4.4 Mutua esclusione 14

1.4.5 Barriere 15

1.5 Codifica di semplici strutture dati 16

1.5.1 La mutua esclusione rivista 16

1.5.2 Canali bidirezionali 17

1.5.3 Celle 18
1.5.4 Pile 20
1.6 Programmazione distribuita 22
1.6.1 Le locazioni 22
1.6.2 Il nameserver 23
1.6.3 Migrazione 25
1.6.4 Esecuzione del codice *à la Java* 27

2 La macchina virtuale **29**
2.1 Fondamenti 29
2.2 I thread della macchina virtuale 30
2.2.1 Struttura di un thread 31
2.2.2 Corrispondenza tra processi e thread 33
2.2.3 L'ambiente 34
2.2.4 La generazione a due fasi 35
2.3 Join Pattern 37
2.3.1 Generazione a due fasi dei pattern 38
2.3.2 Codifica di una *join definition* 39
2.3.3 Comunicazione 40
2.3.4 Porte sincrone 42
2.3.5 Il costrutto *join let* 43
2.3.6 Le primitive 45
2.4 Locazioni 46
2.4.1 Lo heap 46
2.4.2 La tabella delle porte 48
2.4.3 Creazione di una nuova locazione 48
2.4.4 Migrazione 53
2.5 Eccezioni 54

3 Il supporto runtime **57**
3.1 Architettura generale 57
3.2 Implementazione della macchina virtuale 59
3.2.1 Il ciclo di esecuzione principale 59
3.2.2 Thread utente: descrittori statici e dinamici 60
3.2.3 Porte e join pattern 62
3.2.4 Struttura dei messaggi e comunicazione 66

3.2.5 Le istruzioni bloccanti 67
3.2.6 Le primitive 68
3.2.7 Locazioni 69
3.2.8 Comunicazione inter-locazione 72
3.2.9 Migrazione 77
3.3 Comunicazione a basso livello 80
3.3.1 La tabella delle connessioni 80
3.3.2 La tabella di instradamento 82
3.3.3 Messaggi ritardati 83
3.4 Primitive e nameserver 85
3.4.1 Implementazione delle primitive 85
3.4.2 Il nameserver 86
3.5 Invocazione del supporto runtime 88

4 Descrizione formale del linguaggio 91
4.1 Convenzioni lessicali 91
4.1.1 Identificatori 91
4.1.2 Costanti 92
4.1.3 Operatori 93
4.1.4 Commenti 94
4.2 Grammatica del linguaggio 94
4.2.1 Corpo principale 94
4.2.2 Processi 94
4.2.3 Espressioni 97
4.2.4 Dichiarazioni 99
4.2.5 Espressioni di tipo 103

5 Il compilatore 105
5.1 Architettura generale del compilatore 105
5.2 Analisi lessicale e sintattica 106
5.3 Semantica statica 107
5.3.1 Controllo dello scope 108
5.3.2 Controllo dei tipi 111
5.3.3 Controllo delle continuazioni 112
5.4 Generazione del codice intermedio 113
5.4.1 Caratteristiche del codice intermedio 113

5.4.2 La pila dei thread . . . 114
5.4.3 Compilazione delle locazioni . . . 119
5.5 Ottimizzazioni . . . 122
5.5.1 Thread non bloccanti . . . 122
5.5.2 Salti condizionati . . . 124
5.5.3 Ottimizzazioni minori . . . 124
5.6 Generazione del codice oggetto . . . 125
5.7 Invocazione del compilatore . . . 126

6 Conclusioni **129**
6.1 Differenze sintattiche . . . 129
6.2 Prestazioni . . . 130
6.3 Un sistema di bilanciamento del carico . . . 132
6.4 Estensioni future . . . 133

A Introduzione al join-calculus **135**
A.1 La reflexive CHAM . . . 135
A.2 Descrizione del modello . . . 136
A.3 Regole di riduzione . . . 138
A.4 La cella . . . 139
A.5 Estensione: continuazione implicta . . . 139
A.6 Computazioni distribuite . . . 140
A.6.1 La distributed reflexive CHAM . . . 140
A.6.2 L'albero delle locazioni . . . 141
A.6.3 Esempio: Remote Procedure Call . . . 142
A.7 Locazioni in movimento . . . 142
A.7.1 Migrazione . . . 142
A.7.2 Esempio: esecuzione del codice *à la Java* . . . 143
A.7.3 Esempio: computazione remota . . . 143

B La libreria Pthread **145**
B.1 Thread POSIX . . . 145
B.2 Gestione dei thread . . . 146
B.3 Sincronizzazione . . . 147

Bibliografia **150**

Elenco delle figure

2.1 Esecuzione di un thread e strutture associate 55

3.1 Architettura del supporto runtime 58
3.2 Struttura di un messaggio interno 66
3.3 Possibili implementazioni della tabella delle risorse 79
3.4 Formato del messaggio di migrazione 80
3.5 Conflitto tra connessioni contemporanee 82
3.6 Instradamento dei messaggi e algoritmo di routing 84

5.1 Fasi del compilatore 105
5.2 Rappresentazione della tabella dei simboli 110
5.3 Generazione dei thread con diversa associatività dell'operatore | 118
5.4 Pool delle costanti e pila delle locazioni 121

Elenco delle tabelle

2.1 Istruzioni della macchina virtuale 31

4.1 Sequenze di escape riconosciute dal compilatore 93

6.1 Tempi di esecuzione per il calcolo di `fibonacci(27)` 131
6.2 Tempi di esecuzione per l'algoritmo QuickSort nel caso pessimo 131

A.1 Sintassi del join-calculus 137
A.2 Variabili definite, ricevute e libere 138

Capitolo 1

Introduzione al linguaggio

Il linguaggio che abbiamo implementato (derivato in gran parte da quello descritto in [FM97]) è un linguaggio *concorrente*, *distribuito* e con *agenti mobili*. Vediamo cosa significano, presi singolarmente, questi termini.

Un linguaggio concorrente è un linguaggio che mette a disposizione del programmatore delle entità, chiamate in genere processi o *thread*, che possono evolvere in maniera indipendente, ma possono anche cooperare e/o competere tra loro. La programmazione concorrente, contrapposta alla programmazione sequenziale dove c'è un'unica *trama* di esecuzione, può facilitare lo sviluppo di applicazioni specifiche, ed è particolarmente indicata per programmi *event driven* o applicazioni che interagiscono con una rete. Tra i linguaggi concorrenti ricordiamo Ada, Modula-3 o il più recente Java [F$^+$97].

Il termine distribuito è sempre associato ad un ambiente di lavoro dove sono coinvolti più calcolatori connessi tra loro attraverso una rete. In questo ambito i programmi in esecuzione sui vari calcolatori possono scambiare messaggi l'uno con l'altro attraverso il supporto di comunicazione. Un linguaggio distribuito offre direttamente al programmatore questi meccanismi di comunicazione, o più in generale può presentare un'*astrazione* della comunicazione che assolve l'utente da tutte le noiose problematiche tipiche della programmazione in rete. È il caso di precisare che il C, ad esempio, non è un linguaggio distribuito in senso stretto, anche se è possibile scrivere applicazioni distribuite in C usando apposite librerie. In C i meccanismi di comunicazione non sono intrinseci del linguaggio, sono semplicemente forniti dall'esterno e presentati con un'interfaccia compatibile con il linguaggio. Java fa un passo avanti

presentando un'interfaccia ad alto livello per l'interazione tra programmi su calcolatori differenti, ma l'impressione globale è comunque quella di un *add-on* che arricchisce la libreria del linguaggio, ma non ne fa parte direttamente.

Il concetto di "agente mobile" sta assumendo un significato preciso solo negli ultimi anni ed indica un'entità, simile ad un processo, che ha la caratteristica di potersi spostare lungo le linee di comunicazione e di poter essere eseguita su calcolatori differenti. Una definizione precisa probabilmente non esiste e quella che abbiamo dato qui deve essere attentamente valutata. In particolare si è detto che un agente mobile è simile ad un processo, e non ad un programma, attenendosi alla terminologia tipica dei sistemi operativi dove per processo si intende un programma in esecuzione. In effetti è ormai pratica usuale trasferire programmi da un calcolatore all'altro attraverso Internet, ma ciò non elegge automaticamente questi programmi al rango di agenti mobili. Un agente può essere visto come un processo e tutte le risorse allocate per quel processo: quando l'agente si sposta, il processo e le sue risorse si spostano assieme. In Java vengono definiti i cosiddetti *applet* che si avvicinano al concetto di agente mobile, tuttavia non catturano appieno le proprietà che abbiamo elencato a causa di alcune restrizioni. In particolare, il "ciclo di vita" di un *applet* è di solito costituito da tre fasi specifiche e ordinate (spostamento, esecuzione e terminazione) che l'*applet* non può alterare.

In questo capitolo introdurremo i vari aspetti del linguaggio attraverso una serie di esempi, cisacuno eventualmente accompagnato dai risultati della computazione di seguito al simbolo $\Rightarrow$.

1.1 Fondamenti

1.1.1 Processi ed espressioni

La distinzione concettuale che assumeremo nel seguito tra processi ed espressioni sta nel fatto che una espressione produce un risultato, un processo no. Di conseguenza, queste entità differiscono anche nel tipo di composizione che è possibile tra esse.

Due espressioni possono essere composte attraverso l'operatore di composizione sequenziale ; (punto e virgola). In questo modo si specifica senza

ambiguità l'ordine di valutazione delle due espressioni che, stando alla prassi comune, è da sinistra verso destra.

Due processi possono essere composti attraverso l'operatore di composizione parallela `|` (barra verticale, pronunciata *par*). Due processi concorrenti avanzano in maniera indipendente, non è prevedibile e nemmeno sensato dire chi verrà eseguito prima e chi dopo.

Una espressione ed un processo possono venir combinati (in quest'ordine) attraverso l'operatore di composizione sequenziale per dare vita ad un altro processo.

Nel linguaggio l'esecuzione di una espressione è indicata dalla parola chiave `do`:

```
do print(1); print(2)
;;
⇒ 12
```

`print` è il nome di una *primitiva* che stampa a video l'intero indicato come argomento. A questo proposito si veda la sezione 4.2.4 per l'elenco delle primitive supportate. Nell'esempio la stampa del numero 1 avviene sempre prima della stampa del numero 2.

L'esecuzione di uno o più processi è invece indicata dalla parola chiave `spawn`:

```
spawn print(1); | print(2);
;;
```

Ci sono tre cose importanti da notare qui:

- `print(x);` è un processo composto dall'espressione `print(x)` e dal processo vuoto "".
- L'operatore di composizione parallela opera esclusivamente tra processi.
- L'ordine di esecuzione dei due processi è del tutto arbitrario. Il fatto che numerose esecuzioni dello stesso codice portino sempre allo stesso ordine di stampa dei numeri non autorizza a fare alcuna assunzione su come evolvono i processi.

1.1.2 Definizione di nomi

In ogni linguaggio di programmazione è possibile associare un nome al risultato di una computazione, in particolare alla valutazione di una espressione. Nel nostro linguaggio questa associazione è ottenuta attraverso il costrutto `let`:

```
let x: int = 1 + 2
;;

do print(x)
;;
```

Si noti in particolare come la dichiarazione del nome `x` sia accompagnata da un tipo (`int`, cioè un numero intero): nel linguaggio ogni nome deve avere un tipo associato e definito esplicitamente dal programmatore. In contrapposizione, altri linguaggi inferiscono il tipo di un nome dall'utilizzo del nome stesso (*type inference*, [ASU85]). In effetti la differenza più evidente tra il nostro linguaggio e quello da cui abbiamo preso ispirazione è la gestione dei tipi, che nel nostro caso è esplicita.

Più nomi possono essere definiti nello stesso costrutto `let`:

```
let one: int = 1,
    two: int = one + one
;;
```

La valutazione delle varie espressioni è sequenziale, dall'alto al basso. Il costrutto `join let`, una variante del semplice `let`, consente la valutazione di più espressioni "in parallelo":

```
join let one: int = 1,
         two: int = 1 + 1
;;

do print(one); print(two)
;;
```

L'idea è che le espressioni `1` e `1 + 1` vengono valutate da due processi distinti che possono evolvere in modo indipendente. Ciò che viene garantito è che le

istruzioni presenti dopo il costrutto vengono eseguite solo quando tutti i processi creati con `join let` hanno terminato di valutare le relative espressioni. In questo senso si può dire che tali processi si sincronizzano implicitamente al termine del costrutto.

Esempi più significativi di utilizzo di questi costrutti saranno descritti dopo aver introdotto i *canali*.

1.2 Canali

Nel linguaggio l'entità primitiva principale è il canale, che ha lo scopo di trasportare messaggi. A volte ci si riferirà ad un canale come ad una porta, in questo senso i due termini possono essere considerati intercambiabili.

1.2.1 Canali asincroni

La definizione di un canale è sempre accompagnata dalla definizione di un processo da eseguire quando viene spedito un messaggio sul canale stesso. Vediamo un esempio:

```
def     echo: int->
where   echo(x) = { print(x); }
;;
```

Notare come il costrutto sia diviso in due parti. La prima parte, quella che segue `def`, è in effetti una *dichiarazione* (seguendo la distinzione che fa il C tra dichiarazioni e definizioni [KR88]). In sostanza si dichiarano nome e tipo di una o più porte, dette *porte definite*. Dall'esempio si vede che il tipo `int->` indica un canale (che poi classificheremo *asincrono*) che accetta come messaggi dei numeri interi. La seconda parte, quella che segue `where`, è la *definizione* vera e propria: consente di dare un nome al messaggio inviato sul canale (nell'esempio è `x`, che prende il nome di *variabile ricevuta*), e associa al canale un processo, racchiuso tra parentesi graffe.

A questo punto è possibile inviare un messaggio sul canale `echo` appena definito scrivendo:

```
spawn echo(1) | echo(2)
;;
```

$\Rightarrow 21$

Da ciò capiamo anche che l'azione di inviare un messaggio su un canale è a tutti gli effetti un processo e non un'espressione (infatti abbiamo usato il costrutto `spawn` e non `do`).

La seconda osservazione importante da fare è che, nonostante il processo associato al canale sia unico, ogni volta che un messaggio viene inviato sul canale una nuova attivazione, o *istanza*, del processo è creata e istanze attive contemporaneamente non interferiscono tra loro.

I canali che possono essere definiti sono, nella terminologia del π-calculus [Mil91], *poliadici*, in quanto trasportano messaggi che possono contenere più elementi di natura diversa. È quindi ammissibile scrivere:

```
def     echo3: <int, int, int> ->
where   echo3(a, b, c) = {
            print(a);
            print(b);
            print(c);
        }
;;
```

Il tipo `<int, int, int>` indica una *tupla* composta da tre interi. I tipi `<int>->` e `int->` sono equivalenti, anche se occorre precisare che una tupla non può apparire all'interno di un'altra tupla.

I canali, così come i numeri interi, i valori booleani, i caratteri e le stringhe di caratteri, sono entità di prima classe, intendendo con ciò che possono essere inviati a loro volta come valori nei messaggi.

A questo punto anche il programmatore meno esperto avrà notato la somiglianza tra la definizione di canali e la definizione di procedure (o funzioni) così comuni negli usuali linguaggi di programmazione, tant'è che spesso useremo il termine "invocare" piuttosto che "inviare un messaggio". In pratica sia i canali che le procedure hanno un nome, una lista di argomenti (o parametri) e c'è un *corpo*, nel nostro caso un processo, che viene eseguito ad ogni "invocazione".

A differenza di procedure e funzioni, tuttavia, l'invio di un messaggio su un canale non arresta l'esecuzione del "chiamante", mentre l'invocazione di una procedura (o funzione) provoca la sospensione del chiamante fino a quando la procedura *ritorna*. In più, procedure e funzioni possono comunicare dei valori nel momento in cui ritornano, mentre i processi semplicemente terminano.

Se si volessero specificare due canali e si volesse fare in modo che l'invocazione del primo seguisse quella del secondo si potrebbe fare così:

```
def     before: <void ->> ->,
        after: void ->
where   before(c) = {
            print_string("Before.");
            c()
        }, after() = {
            print_string("After.\n");
        }
;;

spawn   before(after)
;;
```

⇒ Before.After.

In questo esempio sono stati introdotti alcuni nuovi elementi. Innanzi tutto si è visto come dichiarare e invocare un canale che accetta messaggi vuoti attraverso la parola chiave `void` (un messaggio vuoto sarà spesso chiamato *segnale*). In secondo luogo è stata introdotta una nuova primitiva, `print_string`, che stampa la stringa passata come argomento sul terminale.

Questo modo di procedere, per quanto funzionale, appare macchinoso e sicuramente poco "elegante". Ecco allora che il linguaggio offre un secondo tipo di canale, detto *sincrono* in contrapposizione con il primo tipo, detto *asincrono*, che consente di nascondere i dettagli relativi alle *continuazioni*, dettagli che nell'esempio sono chiaramente visibili con il passaggio del canale `after` al momento dell'invocazione di `before`.

1.2.2 Canali sincroni

Il meccanismo di continuazione implicita aumenta di fatto le somiglianze tra canali e funzioni. Riscrivendo l'esempio precedente trasformando `before` in un canale sincrono si ottiene:

```
def     before: void -> void,
        after: void ->
where   before() = {
```

```
            print_string("Before.");
            reply
        }, after() = {
            print_string("After.\n");
        }
;;

spawn before(); after()
;;
```

Sono tre le cose importanti da notare:

- Il tipo di un canale sincrono che accetta messaggi vuoti (segnali) e ritorna messaggi vuoti è `void -> void`. Più in generale, un canale sincrono ha un tipo t_1 `->` t_2 dove t_1 e t_2 indicano rispettivamente il tipo dei messaggi accettati dal canale ed il tipo dei messaggi restituiti.
- `reply` è una parola chiave del linguaggio che rappresenta un processo speciale usato per ritornare. In un certo senso ha lo stesso effetto di `c()` nell'esempio precedente.
- L'invio di un messaggio ad un canale sincrono è visto come una espressione e non come un processo.

A questo punto si possono definire delle vere e proprie funzioni che effettuano calcoli sugli argomenti e restituiscono un risultato:

```
def     succ: int -> int
where   succ(x) = { reply x + 1 }
;;

do print(succ(1))
;;
```

$\Rightarrow 2$

L'istruzione `reply` è più generale di una semplice `return` in C [KR88], e consiste di fatto nell'invio di un messaggio che può essere costituito da un numero qualsiasi di elementi. La funzione (a questo punto i canali sincroni possono essere chiamati anche così) che scambia due interi può essere scritta nel modo seguente:

```
def     swap: <int, int> -> <int, int>
where   swap(a, b) = { reply b, a }
;;

let c, d: int = swap(1, 2)
;;

do print(c); print(d)
;;
```
$\Rightarrow 21$

L'esempio con i canali `before` e `after` entrambi in forma asincrona ci permette però di riflettere sull'operatore di composizione sequenziale. In particolare, ai fini dell'espressività del linguaggio, tale operatore è totalmente superfluo, tant'è che nel calcolo che sta alla base del linguaggio è assente [FG96, FGL+96, Mil93]. La composizione parallela non è semplicemente una caratteristica *aggiunta*, essa è la vera e unica forma di composizione nel linguaggio: la composizione sequenziale è solo un caso particolare di composizione parallela.

1.2.3 Definizioni ricorsive

Un'aspetto fondamentale delle definizioni è che i nomi delle porte definite sono visibili anche all'interno del processo associato. Ciò permette di scrivere definizioni *ricorsive*:

```
def     fact: int -> int
where   fact(x) = {
            if x <= 1 then { reply 1 }
            else { reply fact(x - 1) * x }
        }
;;
```

Nell'esempio viene anche introdotto il costrutto di esecuzione condizionale *if-then-else*. Tale costrutto è presente in due forme a causa della distinzione tra espressioni e processi. Affinché la grammatica del linguaggio non risulti ambigua la forma per i processi necessita obbligatoriamente delle parentesi graffe per racchiudere i due rami.

La definizione di processi mutuamente ricorsivi è consentita dal fatto che le porte vengono dichiarate *tutte* prima della definizione vera e propria:

```
def     Succ, Copy: int -> int
where   Copy(x) = {
                if x = 0 then { reply 0 }
                else { reply Succ(x - 1) }
        }, Succ(x) = {
                reply 1 + Copy(x)
        }
;;

do print(Succ(123))
;;
⇒ 124
```

1.2.4 Primitive

Le primitive possono essere considerate come particolari porte (in genere sincrone) che attivano processi interni al supporto runtime, e che pertanto necessitano della sola dichiarazione attraverso la parola chiave `external`:

```
external print: int -> void,
         print_string: string -> void,
         print_char: char -> void
;;
```

La dichiarazione di una primitiva non rende la primitiva stessa automaticamente utilizzabile, bisogna che essa sia stata definita anche all'interno del supporto runtime. Tuttavia è solo al momento dell'invocazione di una primitiva che il supporto runtime ne verifica l'esistenza. Un elenco delle primitive definite nell'implementazione sarà presentato nella sezione 4.2.4.

1.3 Join pattern

Anche se gli strumenti visti fino ad ora possono sembrare più che sufficienti nell'ambito della programmazione sequenziale, essi non lo sono affatto in quello della programmazione concorrente.

Avere più processi in esecuzione "parallela" può essere una cosa utile, ma presto o tardi arriva il momento in cui questi processi devono sincronizzarsi tra loro. Abbiamo mostrato in precedenza come nel costrutto `join let` sia presente una forma primitiva di sincronizzazione, ma ciò non è certamente sufficiente per coprire tutte le possibili interazioni e soprattutto il programmatore non ha molti modi per intervenire.

Nel linguaggio, e anche nel calcolo [FG96] dal quale deriva, è di fondamentale importanza il concetto di *pattern*, definito nella sua forma più astratta come come un insieme di canali. Quando su ogni canale di un pattern è presente almeno un messaggio allora il pattern si dice attivo. L'attivazione del pattern causa la *consumazione* di un messaggio da ognuno dei canali che lo costituiscono e l'esecuzione del processo associato.

In tutti gli esempi visti fino ad ora sono sempre stati definiti pattern con un unico canale. L'invio di un messaggio su un canale provocava l'immediata attivazione del pattern e di conseguenza l'esecuzione del processo associato. In pratica abbiamo usato un caso molto particolare di sincronizzazione.

La definizione di pattern complessi si integra ed estende naturalmente le semplici definizioni viste. Nel complesso, il costrutto per la definizione di porte e pattern prende il nome di *join-definition* o, più brevemente, *join-def*. Nell'esempio che segue viene stampato un messaggio a video solo quando il pattern si attiva:

```
def     msg: string ->,
        arg: int ->
where   msg(s) | arg(n) = {
            print_string(s);
            print(n);
        }
;;

spawn arg(6) | msg("The argument is: ")
;;
```

⇒ The argument is: 6

Da notare che le variabili ricevute `s` ed `n` sono entrambe visibili all'interno del processo associato. Inoltre, l'ordine con cui i messaggi arrivano ai canali di un pattern non ha importanza. Nell'esempio viene inviato un messaggio ad `arg` in

parallelo con il messaggio inviato a `msg`, e i nomi dei canali sono stati invertiti rispetto all'ordine di apparizione nel pattern per evidenziare questo fatto, ma si ricorda che non ha senso parlare di "ordine" di esecuzione di due processi concorrenti.

Un canale sincrono, come abbiamo visto in precedenza, *deve* avere nel processo associato al pattern una `reply` corrispondente. Nel caso di pattern con un unico canale sincrono ciò non crea ambiguità, ma quando i canali sincroni sono più d'uno è necessario usare una forma "estesa" di `reply`.

L'esempio che segue illustra l'utilizzo di questa forma estesa:

```
def     swap1, swap2: int->int
where   swap1(arg1) | swap2(arg2) = {
            reply arg1 to swap2 | reply arg2 to swap1
        }
;;

join let a1:int = swap1(1),
         a2:int = swap2(2)
;;

do print(a1); print(a2)
;;
⇒ 21
```

In pratica, con `to` si rende esplicito il canale su cui si invia il messaggio di risposta.

1.4 Codifica delle strutture di controllo

1.4.1 Codifica del costrutto *join let*

Il costrutto `join let` può essere facilmente codificato usando classica una *join-definition*. In realtà esso è proprio una abbreviazione per una forma particolare di *join-def* usata ad esempio per valutare gli argomenti di una funzione in parallelo. Nei normali linguaggi di programmazione che seguono lo stile *call-by-value* gli argomenti di una funzione vengono valutati in un certo ordine dipendente dal linguaggio o dalla implementazione, ma nell'ottica della

programmazione concorrente ciò rappresenta una inutile restrizione che può portare a delle inefficienze.

Se si ha una funzione `f` che accetta due argomenti la cui valutazione può essere fatta in parallelo si potrebbe scrivere così:

```
do      def     c1: t1->,
                c2: t2->
        where   c1(l1) | c2(l2) = { f(l1, l2) }
        in      c1(expr1) | c2(expr2)
;;
```

Con il costrutto `join let` il programma risulta molto più naturale ed il compilatore può trarne vantaggio per produrre un codice più efficiente evitando una proliferazione di *join-def* che probabilmente verrebbero usate una sola volta:

```
do      join let l1: t1 = expr1,
                 l2: t2 = expr2
        in       f(l1, l2)
;;
```

1.4.2 Cicli asincroni

Il linguaggio presentato, anche se completo dal punto di vista computazionale, manca di alcuni costrutti comuni, come ad esempio l'iterazione o la mutua esclusione. È utile e anche istruttivo vedere come possono essere codificati questi costrutti usando gli strumenti messi a disposizione. Cominciamo dal costrutto più comune, l'iterazione o ciclo:

```
def     loop: <void->, int> ->
where   loop(a, x) = {
            if x > 0 then { a() | loop(a, x - 1) }
        }
;;

def     echo_star: void->
where   echo_star() = { print_char('*'); }
;;
```

```
spawn   loop(echo_star, 5)
;;
⇒ *****
```

In questo caso il ciclo è "asincrono", nel senso che il codice da iterare è eseguito tante volte quanto richiesto senza preoccuparsi dell'ordine di esecuzione. Quando l'ordine è importante, il canale `a` che compare in `loop` deve essere sincrono, e l'operatore `|` deve essere sostituito con `;`.

1.4.3 Cicli sincroni

Anche con l'ultima modifica il ciclo rimane comunque asincrono in quanto l'esecuzione procede dopo l'invio di un messaggio sul canale `loop`. L'implementazione seguente realizza invece un ciclo sincrono:

```
def     loop: <int->void, int> -> void
where   loop(a, x) = {
            if x > 0 then { a(x); loop(a, x - 1); reply }
            else { reply }
        }
;;

def     print_int: int -> void
where   print_int(x) = { print(x); print_char(' '); reply }
;;

do      loop(print_int, 5)
;;
⇒ 5 4 3 2 1
```

Da notare che in questo caso sono necessari entrambi i rami del costrutto *if-then-else* in quanto una restrizione del linguaggio impone che le continuazioni inviate nei due rami siano uguali. Nel capitolo 4 verranno spiegate in dettaglio le motivazioni di questi vincoli.

1.4.4 Mutua esclusione

Un uso accorto delle *join-definition* consente di controllare l'avanzamento dei processi in modo semplice ed elegante. Un tipico esempio di competizione tra

processi è l'accesso a dati comuni. In genere, per mantenere la consistenza, si vuole che l'accesso a questi dati sia effettuato in mutua esclusione racchiudendo le sezioni critiche del programma all'interno di un apposito protocollo:

```
def     lock, unlock: void -> void,
        mutex: void ->
where   mutex() | lock() = { reply to lock },
        unlock() = { mutex() | reply to unlock }
;;

spawn   mutex()
;;
```

Con queste definizioni è sufficiente racchiudere le sezioni critiche all'interno di `lock();...unlock();` per garantire un accesso sequenziale ai dati. Se un processo invoca la `lock` quando un altro processo è all'interno della sezione critica, il primo processo si blocca.

Naturalmente il tutto funziona supponendo che tutti i processi si comportino "correttamente", ovvero rispettino sempre il protocollo di mutua esclusione che prevede prima l'invocazione di `lock` seguita da una `unlock` senza ulteriori `lock` tra le due. Inoltre, da come è strutturato l'esempio, si nota che il canale `mutex`, pur non essendo di interesse per i processi, è comunque visibile ed un suo uso non accorto o malizioso può compromettere il corretto funzionamento del programma. Si vedrà in una sezione seguente come risolvere in parte questo problema adottando una metodologia *object-oriented*.

1.4.5 Barriere

Diverse volte è stato fatto notare che più processi concorrenti avanzano in modo indipendente e non è possibile fare assunzioni sul loro stato di avanzamento in un determinato momento. Può accadere comunque che ci sia bisogno di sincronizzare due o più processi in un punto specifico della computazione.

Nell'esempio che segue due processi sono in competizione per la stampa a video dei due caratteri `'a'` e `'b'`, e si vuole assicurare che questi siano stampati in un ordine qualsiasi, ma comunque all'interno di due parentesi:

```
def     join1, join2: void -> void
```

```
where   join1() | join2() = {
            reply to join1 | reply to join2
        }
;;

spawn { print_char('('); join1();
        print_char('a'); join1();
        print_char(')'); } |
      { join2(); print_char('b'); join2(); }
;;
```

$\Rightarrow$ (ba)

1.5 Codifica di semplici strutture dati

Le *join-definition* possono essere usate anche per codificare strutture dati in uno stile *object-oriented*, dove i dettagli di realizzazione rimangono nascosti mentre dall'esterno è possibile agire solo attraverso i *metodi* che vengono resi disponibili.

1.5.1 La mutua esclusione rivista

L'esempio visto in precedenza sulla mutua esclusione aveva evidenziato in parte la pericolosità nel rendere pubblici i nomi di canali di servizio che non sono di competenza dell'utilizzatore di un oggetto. Riscriviamo allora l'esempio facendo in modo però di "nascondere" il canale `mutex`:

```
def     mklock: void -> <void->void, void->void>
where   mklock() = {
            def     lock, unlock: void->void,
                    mutex: void->
            where   lock() | mutex() = { reply to lock },
                    unlock() = {
                        mutex() | reply to unlock
                    }
            in      mutex() | reply lock, unlock
        }
;;
```

```
let     l, u: void->void = mklock()
;;
```

In pratica i canali definiti nella *join-def* interna non sono visibili se non vengono esplicitamente comunicati. Notare come il "corpo" della `mklock` sia diviso in tre sezioni ben distinte:

- La parte che segue `def` è a tutti gli effetti una dichiarazione dei canali che si useranno in seguito: questa parte ricorda il concetto di *interfaccia*, ovvero l'elenco dei metodi ed il loro tipo definiti in un oggetto.
- Nella definizione vi è l'implementazione vera e propria dell'oggetto, ovvero la specifica dei suoi metodi. È interessante notare come diventi molto più sfumata la distizione tra dati e procedure, così netta invece nei normali linguaggi a oggetti imperativi.
- La parte che segue `in` è l'inizializzazione dell'oggetto e la comunicazione dei metodi pubblici.

1.5.2 Canali bidirezionali

Una caratteristica del join-calculus [FG96], il modello dal quale è stato derivato il nostro linguaggio, è che tutti i canali definiti sono monodirezionali: i messaggi viaggiano sempre e solo in una direzione. Non deve fuorviare la nozione di canale sincrono, dove apparentemente i messaggi possono tornare oltre che andare, infatti, proprio come è stato fatto vedere, un canale sincrono è realizzato con una coppia di canali asincroni.

Ci proponiamo allora di realizzare un oggetto che ricorda i canali bidirezionali del π-calculus [Mil91]:

```
def     new_pi_channel: void -> <int->, void->int>
where   new_pi_channel() = {
            def   send: int->,
                  receive: void->int
            where send(x) | receive() = {
                      reply x to receive
                  }
```

```
            in    reply send, receive
        }
;;
```

A titolo di esempio, possiamo codificare il processo in notazione π-calculus

$$(\nu c)(\nu d)\ (\ c![1] \mid c![2] \mid c?x.d![x+x] \mid d?y.print(y)\ )$$

ottenendo:

```
spawn   let sc: int->, rc: void->int = new_pi_channel(),
            sd: int->, rd: void->int = new_pi_channel()
        in  sc(1) | sc(2) |
            { let x:int = rc() in sd(x + x) } |
            { let y:int = rd() in print(y); }
;;
```

$\Rightarrow 4$

1.5.3 Celle

Essendo il linguaggio di ispirazione funzionale non esiste un metodo esplicito per dichiarare una variabile e poi operare degli assegnamenti su di essa. Tuttavia, con l'ausilio delle *join-def*, è possibile codificare un oggetto che ha uno stato interno mutevole, e un insieme di metodi per agire su di esso.

Il classico esempio che accompagna ogni tutorial sul join-calculus è quello della cella [FG96, FGL+96]:

```
def     mkcell: int -> <void->int, int->void>
where   mkcell(v0) = {
            def   get: void->int,
                  set: int->void,
                  v: int->
            where get() | v(x) = { v(x) | reply x to get },
                  set(x) | v(y) = { v(x) | reply to set }
            in    v(v0) | reply get, set
        }
;;
```

Tra l'altro, il fatto di codificare queste strutture in tale maniera risolve automaticamente il problema dell'accesso contemporaneo da parte di più processi. L'esempio seguente, che codifica un contatore, renderà più chiaro il perché:

```
def     mkcounter: void -> <void->int, void->void>
where   mkcounter() = {
                def     get: void->int,
                        inc: void->void,
                        Counter: int->
                where   get() | Counter(n) = {
                            Counter(n) | reply n to get
                        },
                        inc() | Counter(n) = {
                            Counter(n + 1) | reply to set
                        }
                in      Counter(0) | reply get, inc
        }
;;
```

Può essere utile mostrare una possibile implementazione in C di questo oggetto, supponendo che siano ammissibili invocazioni "contemporanee" dei metodi, ad esempio in una applicazione composta da più thread:[1]

```
int Counter = 0;

int get() {
        return Counter;
}

void inc() {
        Counter = Counter + 1;
        return;
}
```

[1]Il termine contemporaneo può risultare fuorviante. È chiaro infatti che in un sistema monoprocessore non può esserci esecuzione contemporanea. In questo contesto si intende una situazione in cui l'invocazione di un metodo *interrompe* una invocazione avvenuta in precedenza che non è ancora terminata.

Supponiamo che ci siano due invocazioni distinte, ma "sovrapposte" del metodo `inc`. In ogni caso una invocazione parte prima dell'altra: supponiamo però che *dopo* aver calcolato `Counter + 1`, ma *prima* che sia effettuato l'assegnamento, la prima invocazione subisca il prerilascio del processore e vada in esecuzione la seconda invocazione che riesce a terminare senza ulteriori prerilasci.

Se prima delle due invocazioni il valore di `Counter` è `n`, al termine di entrambe esso vale `n + 1` e non `n + 2` come ci saremmo aspettati: le due invocazioni contemporanee hanno interferito tra loro producendo un risultato inconsistente.

Questo comportamento è scongiurato nel nostro linguaggio dal meccanismo di *pattern matching*. Quando si invoca il metodo `inc` e si attiva il secondo pattern della *join-def*, il messaggio sul canale `Counter` viene *consumato* e quindi ogni ulteriore invocazione di `inc` o di `get` è interdetta fino a quando un nuovo messaggio non viene inviato sul canale, cosa che accade dopo aver calcolato il nuovo valore per il contatore (`Counter(n + 1)`).

1.5.4 Pile

Il nostro linguaggio ha delle severe restrizioni dal punto di vista dei tipi e non ammette ad esempio la definizione di tipi ricorsivi quali possono essere liste, pile o alberi. Vediamo allora come codificare una pila nel linguaggio usando il solito meccanismo delle *join-def*:

```
def   mkstack: void -> <void->bool, int->void, void->int>
where mkstack() = {
        def   empty: void->bool,
              push: int->void,
              pop: void->int,
              null: void->,
              some: <int,void->bool,int->void,void->int>->
        where empty() | null() = {
                  null() | reply true to empty
              }, empty() | some(x, e, psh, pp) = {
                  some(x, e, psh, pp) | reply false to empty
              }, push(x) | null() = {
                  let e: void->bool, psh: int->void,
```

```
                    pp: void->int = mkstack()
                in  some(x, e, psh, pp) | reply to push
            }, push(x) | some(y, e, psh, pp) = {
                psh(y); some(x, e, psh, pp) |
                reply to push
            }, pop() | some(y, e, psh, pp) = {
                if e() then { null() | reply y to pop }
                else {
                    some(pp(), e, psh, pp) |
                    reply y to pop
                }
            }
      in    null() | reply empty, push, pop
    }
;;

let empty:void->bool, push:int->void, pop:void->int = mkstack()
;;

do push(1); push(2); push(3);
   print(pop()); print(pop()); print(pop())
;;
⇒ 321
```

La pila ha due possibili stati interni: la pila vuota è indicata da un messaggio sul canale `null`. Una pila con almeno un elemento è indicata da un messaggio sul canale `some` che contiene l'elemento stesso e i tre metodi per agire sulla pila sottostante.

È interessante notare come non ci sia un pattern che gestisca il caso in cui si tenta di invocare una `pop()` quando la pila è vuota. Normalmente in questo caso viene generata un'eccezione, mentre nell'esempio l'invocazione di `pop()` semplicemente blocca il chiamante finché un altro processo non inserisce un elemento sulla pila.

1.6 Programmazione distribuita

Come abbiamo visto, il meccanismo primitivo che sta alla base del linguaggio è la comunicazione. Di conseguenza la sintassi del linguaggio consente di gestire in modo totalmente trasparente la comunicazione "locale", cioè tra processi della stessa macchina, e la comunicazione "remota", cioè tra processi appartenenti a macchine distinte.

In effetti il linguaggio introduce un concetto più astratto di quello di "macchina", che è quello di *locazione*. Ogni processo appartiene ad una certa locazione e ogni locazione è situata su una certa macchina. Le locazioni sono organizzate in modo gerarchico e, all'interno di una stessa macchina, l'insieme delle locazioni forma un albero.

Inoltre, le locazioni possono essere create dinamicamente e possono *migrare*, ovvero diventare figlie di una locazione diversa. Affronteremo la migrazione in una sezione successiva, mentre ora vedremo come sono strutturate e come vengono create le locazioni.

1.6.1 Le locazioni

Abbiamo detto che ogni processo appartiene ad una determinata locazione, ma negli esempi visti finora non abbiamo mai dichiarato una locazione che contenesse i processi creati. In effetti il supporto runtime crea una locazione "invisibile" che contiene tutti i processi di un programma.

Una nuova locazione è introdotta dalla parola chiave `location`. La locazione appena creata diviene figlia della locazione in cui viene fatta la dichiarazione:

```
location new_loc
        succ: int -> int
where   new_loc {
            with succ(x) = { reply x + 1 }
            init print_string("'new_loc' created.\n");
        }
;;

let s1: int = succ(1)
;;
```

Nell'esempio si definisce una nuova locazione `new_loc` con al suo interno il pattern `succ` che realizza la funzione successore. A differenza delle definizioni canoniche, l'elenco dei canali viene specificato dopo il nome della locazione, mentre i pattern vengono definiti solo dopo la parola chiave `with`.

La parola chiave `init` indica invece il processo dedicato all'inizializzazione della locazione. Sia la parte `with` che la parte `init` sono opzionali e possono essere omesse.

Come si vede dall'esempio, la funzione `succ` viene invocata in modo del tutto analogo a quanto fatto nelle comunicazioni intra-locazione.

Più locazioni possono essere definite nello stesso costrutto:

```
location a
        a1, a2, a3: void->
and      b
        b1, b2, b3: int->
and      c
        c1, c2, c3: bool->
where   a {
            with a1() | a2() | a3() = { ... b1(5) ... }
            ...
        }, b {
            with    b1() | b2() = { ... a3() | c2(false) ... },
                    b2() | b3() = { ... }
        }, c {
            ...
        }
;;
```

È importante notare che lo *scope* dei canali definiti in una locazione copre tutte le locazioni definite nello stesso costrutto e le istruzioni che seguono nel programma, tuttavia un canale definito in una locazione non può comparire in un pattern di una locazione diversa. Si veda il capitolo 4 per una descrizione dettagliata della sintassi e delle restrizioni sui *join pattern*.

1.6.2 Il nameserver

Il costrutto `with` all'interno di una locazione permette di definire dei canali il cui nome è visibile anche dall'esterno della locazione stessa. Questo meccani-

smo è efficace quando le locazioni che devono comunicare sono tutte definite in un unico file sorgente e quindi è possibile referenziare i canali per nome. Tuttavia esistono situazioni in cui il codice è organizzato in più file, oppure si deve interagire con un programma scritto da terzi e del quale non è disponibile il sorgente. In questi casi non è possibile usare un canale remoto in quanto nel nostro codice sorgente tale canale non è stato effettivamente dichiarato!

Il supporto runtime mette allora a disposizione due primitive che consentono di registrare in una tabella un valore associato ad un nome simbolico e successivamente reperire il valore registrato attraverso lo stesso nome.

In questo modo è sufficiente conoscere i nomi simbolici delle entità registrate per poterle reperire. Nell'esempio che segue si definisce in un file la funzione successore e la si registra attraverso il nameserver. In un altro file si accede al nameserver per conoscere la funzione.

```
def   succ: int->int
where succ(x) = {
          reply x + 1
      }
;;

do register(succ, "Succ")
;;
```

```
let s:int->int = lookup("Succ")
;;

do print(s(1))
;;
⇒ 2
```

Un processo che invoca la primitiva `lookup` su un nome non presente nel nameserver si blocca finché il nome non viene registrato attraverso la primitiva `register`.

Attualmente le primitive `register` e `lookup` non effettuano alcun controllo sul tipo delle entità utilizzate. Ciò significa che è possibile registrare un valore con un tipo e leggerlo con tipo diverso. Nell'esempio che segue si usa il nameserver per convertire un carattere nel codice ASCII corrispondente:

```
do  register('a', "a char");
    let a_ascii: int = lookup("a char") in
        print(a_ascii)
;;
⇒ 97
```

Nonostante questo metodo possa funzionare in certi casi (come dimostra

l'esempio), *il suo utilizzo è decisamente sconsigliato* e può portare a risultati imprevedibili.

1.6.3 Migrazione

Per *migrazione* si intende lo spostamento di una locazione e tutte le sotto-locazioni definite al suo interno in un'altra locazione. La migrazione viene effettuata nel momento in cui si invoca la primitiva `go` con argomento il nome di una locazione:

```
location dest
and      source
where   dest {
        }, source {
            init print_string("Here!\n");
                 go(dest);
                 print_string("There!");
        }
;;
```

Vediamo un caso in cui si rende utile la migrazione. Supponiamo che una certa locazione metta a disposizione un servizio, ad esempio una funzione matematica, e che registri questo servizio attraverso il nameserver:

```
location server_location
        service: int->int
where   server_location {
            with service(x) = { reply x }
            init register(service, "Service");
                 register(server_location, "Server Location");
                 print_string("Service registered.\n");
        }
;;
```

Un ipotetico cliente che si trova in una locazione su una macchina distinta ha due modi per usufruire del servizio:

1. Usare `lookup` per conoscere la funzione che offre il servizio e invocare la funzione da remoto.

2. Spostarsi nella locazione dove risiede il servizio, o perlomeno in una locazione nella stessa macchina, effettuare la computazione e poi comunicare il risultato.

Il vantaggio della prima soluzione è una più semplice organizzazione del programma in quanto non sono coinvolte migrazioni:

```
let service: int->int = lookup("Service")
;;

def     do_work: int->int
where   do_work(n) = {
            if n = 0 then { reply 0 }
            else { reply service(n) + do_work(n - 1) }
        }
;;
```

Tuttavia, se nel complesso la computazione richiede l'accesso al servizio numerose volte, si generano altrettanti messaggi in rete con un corrispondente utilizzo della banda disponibile e soprattutto un notevole calo delle prestazioni.

In quest'ultima ipotesi diventa allettante la seconda alternativa: si prepara un *agente* che migra nella locazione dove viene offerto il servizio, l'agente effettua la computazione localmente limitando l'utilizzo di banda (che si riduce alla sola migrazione e al messaggio di risposta) e usufruendo delle prestazioni offerte dalla comunicazione locale.[2] A computazione terminata l'agente può o tornare indietro o semplicemente comunicare il risultato:

```
let service_location: location = lookup("Service Location"),
    service: int->int = lookup("Service")
;;

def     do_work: int->int
where   do_work(n) = {
            location agent
                do_work_aux: int->int,
                go_away: void->void
```

[2]In realtà i messaggi scambiati sulla rete nella soluzione proposta di seguito sono quattro, per esercizio individuare gli altri due.

```
            where agent {
                with do_work_aux(n) = {
                    if n = 0 then { reply 0 }
                    else { reply service(n) + do_work_aux(n-1) }
                }, go_away() = {
                    go(service_location);
                    reply
                }
            } in go_away();
                 reply do_work_aux(n)
        }
;;
```

1.6.4 Esecuzione del codice *à la Java*

Nell'ultimo esempio visto un agente si spostava dal cliente al servente dove portava a termine la computazione. Il caso opposto è quello oggi più comune di un agente che dal servente migra al cliente quando quest'ultimo si collega o richiede un particolare servizio. È il tipico esempio di un applet Java che migra all'interno del browser quando ci si collega ad un certo indirizzo.

Nel nostro linguaggio il server si scriverebbe così:

```
def     download_applet: location -> (void->void)
where   download_applet(l) = {
            location applet
                    go_to_client, do_work: void->void
            where applet {
                    with    go_to_client() = { go(l); reply },
                            do_work() = { ... }
            } in    go_to_client();
                    reply do_work
        }
;;

do register(download_applet, "Applet Service")
;;
```

La chiamata a `download_applet` termina a migrazione effettuata e ritorna la porta da invocare per eseguire l'applet.

Capitolo 2

La macchina virtuale

Questo capitolo descrive la macchina virtuale (detta anche macchina astratta) che abbiamo realizzato per l'implementazione del linguaggio introdotto nel capitolo precedente.

In particolare si vedranno le sequenze di istruzioni che corrispondono ai vari costrutti del linguaggio, verranno motivate le scelte fatte per particolari istruzioni, saranno messe in evidenza le strutture "astratte" che compongono la macchina virtuale stessa.

Ciò che non verrà descritto qui è invece *come* la macchina virtuale sia stata implementata, cosa che sarà oggetto principale del prossimo capitolo.

2.1 Fondamenti

Tutti i termini che abbiamo introdotto in precedenza (processo, locazione, pattern, canale, ecc.) hanno un significato ed una rappresentazione corrispondente nella macchina virtuale. In altre parole, così come esistono processi nel linguaggio, esistono processi nella macchina virtuale che li implementano. Questo fatto non implica però che ci sia sempre una corrispondenza uno-a-uno tra processi del linguaggio e processi della macchina astratta. Allo stesso modo, mentre nel linguaggio un canale è rappresentato da un nome simbolico, nella macchina astratta esso sarà visto come un'entità numerica.

Consapevoli di queste ed altre differenze ci accingiamo a descrivere in dettaglio la struttura e le istruzioni della macchina virtuale.

Tutte le istruzioni agiscono in dipendenza di zero o più tra argomenti ed

operandi. Gli argomenti sono quantità statiche decise dal compilatore e rappresentate a fianco delle stesse istruzioni, gli operandi invece sono usati per comunicare dati tra le istruzioni attraverso lo *stack*. Non esistono di fatto *registri* utilizzabili direttamente, esistono invece dei registri "invisibili" che caratterizzano lo stato della macchina in un determinato istante:

Registro del processo corrente: Abbiamo detto che anche nella macchina virtuale esiste il concetto di processo. La macchina deve poter distinguere processi diversi attraverso un identificatore e questo registro contiene l'identificatore del processo che è attualmente in esecuzione. Inoltre, questo registro ha la caratteristica di essere *globale*, essendo unico in tutta la macchina.

Puntatore allo stack (*Stack Pointer*): Ogni processo ha uno stack ed un registro che indica la prima posizione libera nello stack stesso. Questo registro, al contrario del precedente, è *locale*, ovvero ne esiste una copia distinta per ogni processo.

Contatore di programma (*Program Counter*): Questo registro contiene l'indirizzo della successiva istruzione da eseguire nel processo corrente. Anche questo registro è locale.

Le istruzioni della macchina virtuale, elencate nella tabella 2.1, si dividono in varie categorie in base al tipo di operazione svolta. La maggior parte di queste istruzioni verranno descritte nel corso del capitolo, mentre per una descrizione dettagliata di tutte si rimanda al codice sorgente.

2.2 I thread della macchina virtuale

È arrivato il momento di essere più precisi riguardo all'implementazione dei processi nella macchina virtuale. Nell'introduzione si è già accennato al fatto che non esiste una perfetta corrispondenza tra processi nel linguaggio e processi nella macchina astratta. Da ora in poi, per evitare confusione, chiameremo *thread* le unità di esecuzione della macchina virtuale.

Ogni thread ha uno *stato* interno memorizzato in una struttura detta DTD (Dynamic Thread Descriptor). Il DTD contiene il valore dei registri locali (cioè lo Stack Pointer ed il Program Counter) ed altre informazioni di competenza

Classe	Istruzioni
Aritmetica	`ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `INC`, `DEC`.
Stack	`PUSH`, `LOAD`, `STORE`, `DUP`.
Controllo del flusso	`JUMP`, `JLT`, `JGT`, `JEQ`, `JNE`, `JLE`, `JGE`.
Entità fondamentali	`THREAD`, `LOCATION`, `PATTERN`, `EXTERN`, `PORT`.
Comunicazione	`GET-CONT`, `SEND`, `RECV`, `RECVFROM`.
Migrazione	`GO`.

Tabella 2.1: Istruzioni della macchina virtuale

del modulo runtime. L'insieme di queste informazioni inquadra il thread in uno *stato di avanzamento* che può essere uno dei seguenti:

Esecuzione: il thread ha la possibilità di usare il processore virtuale per eseguire le proprie istruzioni. Si dice che un thread in esecuzione "avanza".

Pronto: il thread è pronto per l'avanzamento, ma il processore virtuale è impegnato ad eseguire le istruzioni di un altro thread. I thread pronti sono raccolti in una coda (detta "coda dei pronti") dalla quale vengono estratti con disciplina FIFO quando il processore si rende disponibile.

Bloccato: il thread è bloccato in attesa di un evento (nella fattispecie la ricezione di un messaggio) e non è in grado di avanzare né può considerarsi pronto.

2.2.1 Struttura di un thread

Oltre al DTD, ogni thread ha un segmento codice dove sono memorizzate in sequenza le istruzioni che lo compongono. Ogni istruzione è eventualmente accompagnata dai suoi argomenti e risiede ad un determinato indirizzo. Il Program Counter non è altro che un puntatore all'interno del segmento codice alla successiva istruzione da eseguire.

L'altra struttura fondamentale del thread è lo stack. Lo stack ha una dimensione illimitata[1] ed è usato come contenitore temporaneo per tutti i calcoli intermedi.

[1]Il termine è da intendersi come "senza limiti imposti a priori", anche se, per come è

Infine, ogni thread ha un'*area dati* (eventualmente di dimensione nulla) dove vengono memorizzati i valori dei nomi definiti all'interno del thread stesso. In altre parole, tutte le volte che si definisce un nuovo nome, ad esempio nel processo

```
let x: int = 2 in ...
```

la macchina astratta memorizza il risultato della valutazione dell'espressione a un certo offset nell'area dati del thread. In pratica tutti i nomi definiti in un thread hanno una posizione nell'area dati. Le istruzioni `LOAD` e `STORE` consentono di trasferire i valori dall'area dati allo stack e viceversa.

Ora abbiamo abbastanza elementi per esaminare un primo esempio:

```
let one: int = 1,
    two: int = one + one
;;
```

Il codice ottenuto dalla compilazione risulta:

```
0028:    PUSH  1 (1)
0030:    STORE one (8,int)
003c:    LOAD  one (8,int)
0048:    LOAD  one (8,int)
0054:    ADD
0058:    STORE two (12,int)
0064:    EXIT
```

I numeri (in base sedici) che compaiono a sinistra delle istruzioni rappresentano l'indirizzo dell'istruzione stessa. Il Program Counter, durante l'esecuzione, assumerà in sequenza questi valori.

L'istruzione `PUSH` spinge l'intero che ha come argomento sullo stack, e la `STORE` che segue provvede a memorizzare questo valore all'offset 8 dell'area dati, corrispondente al nome `uno`.

Segue la valutazione di "`one + one`" e la conseguente memorizzazione del risultato in `two`. Infine la `EXIT` fa terminare il processo.

strutturata la macchina virtuale, è possibile calcolare la dimensione massima che lo stack di un thread potrà raggiungere.

2.2.2 Corrispondenza tra processi e thread

Mentre i processi formano una categoria sintattica nella grammatica del linguaggio, i thread sono le entità usate per implementarli nella macchina virtuale. Processi e thread non possono essere in corrispondenza biunivoca, infatti mentre tra processi sussiste una relazione di inclusione, tra thread sussiste una relazione di parentela. Consideriamo il processo:

$$P \ ::= \ \texttt{if}\ E\ \texttt{then}\ \{\ P_1\ \}\ \texttt{else}\ \{\ P_2\ \}$$

Chiaramente il processo P include i processi P_1 e P_2. Se pensiamo a un thread come alla sequenza di istruzioni che implementano un determinato processo, potremmo dire che il thread per P *genera* i thread per P_1 e P_2, e quindi ne è padre. In altre parole, il thread per P avrebbe la forma:

1. valuta l'espressione E;
2. se E è vera genera il thread per P_1;
3. se E è falsa genera il thread per P_2;
4. termina.

Da questa discussione capiamo che, al limite, quella che si può avere tra processi e thread è una corrispondenza tra le relazioni di inclusione e parentela, tuttavia mentre i processi sono entità astratte particolarmente comode per la descrizione del linguaggio attraverso una grammatica, i thread rappresentano entità "concrete" (seppur nell'ambito della macchina virtuale) e, intuitivamente, il loro numero deve essere limitato il più possibile per ovvie ragioni di efficienza. L'importante è che la riduzione del numero dei thread, qualora applicata, non violi la semantica dei costrutti.

Riprendendo l'esempio di prima, notiamo innanzi tutto che il thread per P termina dopo la generazione di P_1 o P_2, e inoltre la semantica del costrutto suggerisce che P_1 e P_2 non sono mai in esecuzione contemporaneamente. In conclusione, il codice per P può evitare del tutto la generazione di nuovi thread, eseguendo un "salto" al codice per P_1 o per P_2 dipendente dalla valutazione di E.

Ripetendo il ragionamento per tutti i costrutti del linguaggio si possono trovare altri casi in cui è possibile un'ottimizzazione. Terminiamo questa sezione esaminando il costrutto di composizione parallela:

$$P \ ::= \ P_1 \mid P_2$$

L'implementazione banale del thread per P sarebbe:

1. genera il thread per P_1;
2. genera il thread per P_2;
3. termina.

Non ci vuole molto a capire che se P, anziché generare il thread per P_2, ne esegue direttamente le istruzioni, si risparmia una generazione mantenendo il codice corretto. In maniera speculare si può decidere di generare P_2 ed eseguire P_1.

Queste ottimizzazioni alterano ovviamente la corrispondenza tra la relazione di inclusione e la relazione di parentela, ma non in modo drastico: quello che possiamo dire in generale è che la relazione di inclusione è un raffinamento della relazione di parentela.

2.2.3 L'ambiente

Abbiamo visto che ogni nome definito in un thread ha una posizione nella rispettiva area dati e che ogni thread ha un'area dati separata da quella di altri thread. Tuttavia ci sono casi in cui un nome definito in un thread viene usato all'interno di un thread diverso, in questo caso si dice che il nome è esterno al thread in cui è usato. Si definisce *ambiente* di un thread l'insieme di tutti i suoi nomi esterni.

Siccome le aree dati sono disgiunte, un thread non può referenziare direttamente un nome esterno con l'istruzione `LOAD`. Allora, l'ambiente viene "passato" al thread al momento della generazione ed è situato nella stessa area dati. Così l'area dati viene spezzata in due sottoaree: la prima contiene l'ambiente del thread, mentre la seconda è l'area riservata ai nomi definiti nel thread stesso.

In questo modo un thread può far riferimento ai nomi esterni esattamente come se fossero locali, e cioè attraverso l'istruzione `LOAD`.[2]

2.2.4 La generazione a due fasi

La precedente discussione sull'ambiente fa sorgere un problema: nel momento in cui viene generato un nuovo thread occorre passargli esplicitamente l'ambiente. Ci chiediamo allora quali siano le occasioni in cui un nuovo thread viene creato, in modo da avere un'idea generale delle istruzioni di cui possiamo aver bisogno. Le occasioni sono sostanzialmente tre:

1. Esecuzione dell'istruzione `spawn`.
2. Composizione parallela di più processi.
3. Attivazione di un pattern.

Anche se i pattern non sono ancora stati discussi nell'ambito della macchina virtuale, sappiamo comunque abbastanza su di loro per evidenziare una differenza fondamentale tra i casi 1 e 2 ed il caso 3.

Nei primi due la generazione di un nuovo thread è immediata, o se vogliamo sincrona, nel senso che il thread viene generato nel momento in cui si esegue il codice del costrutto. Nel terzo caso invece la generazione è asincrona, in quanto un nuovo thread verrà generato solo nel momento in cui il pattern sarà attivo, e questo potrà capitare subito, dopo un po', o non capitare affatto.

La creazione ed il passaggio dell'ambiente nei primi due casi è banale: entrambe le fasi possono essere effettuate nel momento stesso in cui il thread è generato. Ma nell'ultimo caso le cose non sono così semplici: mentre la creazione dell'ambiente può (e deve) essere effettuata subito, il suo passaggio deve avvenire solo al momento della generazione vera e propria.

Ci sono almeno due motivi per cui la creazione dell'ambiente non può essere ritardata assieme alla generazione: in primo luogo, gli elementi che costituiscono l'ambiente appartengono all'area dati di processi che, al momento dell'attivazione del pattern, possono essere terminati (e la terminazione causa la deallocazione delle aree dati). In secondo luogo, ammesso di aggirare questo

[2]Non essendo presente una istruzione esplicita di assegnamento non sarà mai possibile avere una `STORE` su un nome esterno, ciò scongiura situazioni di inconsistenza dei dati.

primo problema, la soluzione non sarebbe efficiente visto che l'ambiente creato sarebbe sempre identico.

Se vediamo il caso 3 come una generalizzazione dei casi 1 e 2, possiamo immaginarci una istruzione il cui compito consiste unicamente nel creare un ambiente, e più istruzioni che si occupano della sola generazione dei thread (immediata o ritardata) usando l'ambiente creato.

Ecco perché si parla di generazione a due fasi: per qualsiasi thread si fa distinzione tra la fase di creazione dell'ambiente, che può ricordare la creazione di una *chiusura* (vedere ad esempio la CAM [CCM85, MS86]), e la fase di generazione, in cui la chiusura viene usata per passare l'ambiente al nuovo thread.

Dopo questa introduzione verbale, vediamo finalmente un esempio di generazione di thread in cui c'è un passaggio dell'ambiente:

```
let one: int = 1
;;

spawn let two: int = one + one in
;;
```

Il processo che segue la parola chiave `spawn` definisce il nome `two`, ma usa a sua volta il nome `one` che è esterno. Il codice risulta:

```
0028:    PUSH  1 (1)
0030:    STORE one (8,int)
003c:    LOAD  one (8,int)
0048:    THREAD  __00000000  (8,1,int)
0060:    SPAWN
0064:    EXIT

0068:      __00000000:
0068:        LOAD  one (0,int)
0074:        LOAD  one (0,int)
0080:        ADD
0084:        STORE two (4,int)
0090:        EXIT
```

Le istruzioni agli indirizzi `0028` e `0030` calcolano l'espressione `1` e memorizzano il risultato. Le istruzioni agli indirizzi `003c` e `0048` servono invece per creare l'ambiente del thread da generare.

L'istruzione `THREAD` ha come argomenti l'indirizzo della prima istruzione del thread, rappresentata simbolicamente dall'etichetta `__00000000`, la dimensione dell'area dati del thread da creare (nell'esempio è 8 byte, 4 per l'intero `one` che fa parte dell'ambiente, e 4 per l'intero `two`, definito invece all'interno del thread). Seguono il numero ed il tipo degli elementi che compongono l'ambiente.

L'istruzione `THREAD` prepara così un thread per l'esecuzione, ma non lo genera ancora. Piuttosto, lascia sullo stack un identificatore dell'STD (Static Thread Descriptor) del thread. L'STD, contrapposto al DTD che abbiamo incontrato in precedenza, è un descrittore che contiene le informazioni di base per un thread da generare, e quindi l'ambiente, l'indirizzo della prima istruzione e, anche se nell'esempio non è mostrato, la dimensione complessiva del codice del thread.

L'istruzione `SPAWN`, che segue immediatamente, legge i dati nell'STD identificato dal valore in cima allo stack e genera il thread corrispondente effettuando il passaggio dell'ambiente.

Il thread figlio, quello che segue l'etichetta `__00000000`, fa riferimento a `one` all'offset 0, mentre riferisce `two` all'offset 4, come ci aspettavamo visto che, è stato detto, l'ambiente viene inserito all'inizio dell'area dati. Nella figura 2.1 viene mostrata un'"istantanea" della computazione quando il thread figlio sta per eseguire l'istruzione `ADD`.

Nel seguito si incontreranno altre varianti dell'istruzione `SPAWN`. Quello che ci interessa evidenziare qui è che l'aver spezzato la fase di generazione di un thread in due passi ci consente di usare un'unica istruzione, la `THREAD`, che va bene per tutte le situazioni, sia le `SPAWN` dirette sia, come vedremo nella prossima sezione, per la generazione dei pattern.

2.3 Join Pattern

In questa sezione vedremo come vengono create e come sono viste le porte dalla macchina virtuale. Vedremo inoltre come definire dei pattern di porte e come associare ad ognuno di essi un thread.

2.3.1 Generazione a due fasi dei pattern

Il passaggio esplicito dell'ambiente crea nuove difficoltà anche al di fuori della semplice generazione dei thread. I nomi coinvolti in una *join-definition* sono classificabili in due categorie:

- Le *porte definite* sono quelle che compaiono nei pattern e che sono state esplicitamente dichiarate dopo la parola chiave `def`.
- Le *variabili ricevute* rappresentano gli elementi nei messaggi ricevuti sulle porte che compongono ogni pattern.

Lo scope delle variabili ricevute è limitato al solo processo associato al pattern, e quindi è logico pensare che tali variabili siano allocate nell'area dati del processo stesso. Lo scope delle porte definite, invece, si estende anche al di fuori dei processi associati a pattern, per cui queste porte vengono allocate nell'area dati del thread padre, quello che genera l'intero costrutto. Di conseguenza, i nomi delle porte definite vengono visti come esterni nell'ambito dei processi associati per cui, in caso di necessità, devono essere passati nell'ambiente.

Ciò significa che essi devono essere già noti quando viene creato l'STD di un thread da associare a un pattern e l'operazione di creazione del pattern dovrà avvenire dopo, necessitando proprio dell'STD.

Come se non bastasse sono ammesse definizioni mutuamente ricorsive, quindi è possibile che nell'ambiente di un thread associato a un certo pattern ci sia una porta che compare solo in un pattern definito successivamente.

Da questo ragionamento risulta chiaro che le fasi di creazione delle porte e di definizione dei pattern devono essere distinte, con la prima che precede la seconda. Consideriamo una parte della cella, quella che crea i pattern per i "metodi" `get` e `set`:

```
def get: void->int,
    set: int->void,
    v: int->
where get() | v(x) = { v(x) | reply x to get },
    set(x) | v(y) = { v(x) | reply to set }
in ...
```

Il codice prodotto risulta:

```
0064:    PORT  3
006c:    STORE v (16,a-port)
0078:    STORE set (12,s-port)
0084:    STORE get (8,s-port)
0090:    LOAD  v (16,a-port)
009c:    THREAD  __00000001  (12,1,a-port)
00b4:    LOAD  v (16,a-port)
00c0:    THREAD  __00000003  (16,1,a-port)
00d8:    PATTERN [1,2],[0,2]
```

La prima istruzione, `PORT`, crea tre nuove porte e spinge gli identificatori di queste porte sullo stack. Le tre istruzioni successive memorizzano nell'area dati del thread questi identificatori. Non è un caso che queste `STORE` appaiano in ordine inverso rispetto all'ordine di dichiarazione delle porte, ciò è dovuto al fatto che gli identificatori vengono inseriti su uno stack, e quindi l'identificatore in cima è relativo all'ultima porta creata.

La seconda parte del codice provvede a creare i descrittori STD dei thread associati ad ogni pattern. Essendo due i pattern è giusto che ci siano due thread da creare. Da notare che i thread per entrambi i pattern usano al loro interno il nome `v` che deve essere passato nell'ambiente. Notare inoltre che, sebbene nel codice sorgente si faccia riferimento anche alle porte `get` e `set`, questi nomi non vengono passati nell'ambiente. In effetti l'istruzione `reply` invia un messaggio non *alle* porte indicate, ma alla porta di continuazione implicita ricevuta *da* tali porte. In ogni modo, una discussione più approfondita sulle porte sincrone sarà data nella sezione 2.3.4.

Le istruzioni `THREAD` lasciano sullo stack un identificatore per l'STD creato e quindi al momento dell'esecuzione dell'istruzione `PATTERN` sullo stack sono presenti due identificatori di STD. L'istruzione `PATTERN` elenca i pattern e le porte che compaiono in ciascuno di essi ed associa ad ogni pattern l'STD il cui identificatore viene prelevato dallo stack.

2.3.2 Codifica di una *join definition*

Le istruzioni `PORT` e `PATTERN` di una stessa *join-definition* fanno riferimento allo stesso insieme di porte, anche se in modi diversi.

Siccome le porte sono entità che vengono create dinamicamente (cioè a tempo di esecuzione), il compilatore non può "indovinare" i loro identificatori

che invece saranno decisi dall'istruzione `PORT`. Allo stesso tempo l'istruzione `PATTERN` deve poter fare riferimento in modo semplice ed efficiente alle porte create.

Abbiamo visto che nell'istruzione `PATTERN` le porte vengono identificate con numeri sequenziali a partire da 0. In effetti il numero i nell'istruzione `PATTERN` sta ad indicare l'i-esima porta creata nell'ultima istruzione `PORT`. Tra l'altro, una rappresentazione conveniente per una *join-definition* è una matrice di bit (ciò è possibile in quanto i pattern hanno la caratteristica di essere *lineari*, vedi a pagina 137). In quest'ottica i numeri che compaiono in `PATTERN` possono essere considerati direttamente come l'indice del bit che corrisponde a una certa porta.

Come esempio, la matrice della *join-definition* per la cella sarebbe:

$$\begin{array}{r} \\ \text{pattern 1} \\ \text{pattern 2} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{get} & \texttt{set} & \texttt{v} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}$$

2.3.3 Comunicazione

Ora che abbiamo visto come vengono creati i pattern, possiamo finalmente analizzare le istruzioni per la comunicazione dei messaggi. Queste istruzioni sono state pensate in modo da assomigliare il più possibile alle normali funzioni di comunicazione disponibili nell'ambito della programmazione distribuita.

L'esempio della cella introduce già il concetto di porta sincrona che vogliamo esaminare più in dettaglio nella prossima sezione. Consideriamo invece l'esempio con le porte `before` e `after`, dove per semplicità è stata omessa la stampa dei messaggi a video:

```
def     before: <void->> ->,
        after: void->
where   before(c) = { c() },
        after() = { }
;;

spawn before(after)
;;
```

Il codice compilato risulta:

```
0028:    PORT  2
0030:    STORE after (12,a-port)
003c:    STORE before (8,a-port)
0048:    THREAD  __00000000  (4,0)
005c:    THREAD  __00000001  (0,0)
0070:    PATTERN [1],[0]
0088:    LOAD  before (8,a-port)
0094:    LOAD  after (12,a-port)
00a0:    THREAD  __00000002  (8,2,a-port,a-port)
00bc:    SPAWN
00c0:    EXIT

00c4:    __00000002:
00c4:      LOAD  before (0,a-port)
00d0:      LOAD  after (4,a-port)
00dc:      SEND  (1,a-port)
00e8:      EXIT

00ec:    __00000001:
00ec:      EXIT

00f0:    __00000000:
00f0:      LOAD  c (0,a-port)
00fc:      SEND  (0)
0104:      EXIT
```

In generale l'istruzione `SEND` ha come argomenti il numero ed il tipo degli elementi che deve spedire in un messaggio. La porta a cui inviare il messaggio e gli elementi stessi sono invece operandi reperiti dallo stack. All'indirizzo `00dc` viene inviata la porta `after` come messaggio sul canale `before`, mentre all'indirizzo `00fc` viene inviato un segnale sul canale `c` (che corrisponde alla porta `after`).

L'elenco di tipi nella `SEND`, già incontrato nell'istruzione `THREAD`, è chiamato *signature* di un messaggio. Dal punto di vista della macchina virtuale può sembrare che esso non abbia una reale utilità, ma dal punto di vista dell'implementazione la *signature* è fondamentale. Ne parleremo più approfonditamente quando descriveremo il sistema runtime.

Notiamo infine che di fatto l'ambiente è trattato come un qualsiasi mes-

saggio dotato di *signature*: il suo cosiddetto "passaggio" è in effetti una comunicazione implicita gestita dal modulo runtime.

2.3.4 Porte sincrone

Le porte sincrone non fanno direttamente parte del calcolo [FG96] da cui è stato derivato il linguaggio, ma la loro importanza è tale da giustificare un certo sforzo per implementarle. Per questo la macchina astratta mette direttamente a disposizione alcune istruzioni per utilizzare in modo semplice ed efficiente le porte sincrone.

Nel calcolo, come nell'esempio precedente, per assicurare l'ordine voluto di esecuzione occorreva passare le porte di continuazione in maniera esplicita. Nel report sul join-calculus distribuito [FGL+96] viene mostrato l'esempio di un servizio di stampa di numeri interi. Per stampare una serie di numeri in ordine si usa il seguente processo:

$$\textbf{def } \mathit{print}\langle x, \kappa\rangle \ \triangleright \ \ldots \kappa\langle\rangle \ldots \ \textbf{in def } \kappa\langle\rangle \ \triangleright \ \mathit{print}\langle 4, \kappa'\rangle \ \textbf{in } \mathit{print}\langle 3, \kappa\rangle$$

Dal punto di vista del calcolo non c'è nulla di sbagliato, ma pensando ad una implementazione si ha l'impressione che sia necessario usare una nuova definizione di κ ogni volta che serve una continuazione. Ciò costituirebbe un inutile spreco in quanto la continuazione verrebbe usata una sola volta.

Invece, nella macchina astratta, ogni thread ha un'unica porta di continuazione che viene usata ad ogni invocazione di una porta sincrona. Il fatto che la porta di continuazione sia unica non crea problemi perché, per definizione, se un thread esegue una "send" sincrona non può fare altro finché non ritorna il messaggio di risposta.

Rivediamo l'esempio precedente con `before` porta sincrona:

```
def     before: void->void,
        after: void->
where   before() = { reply },
        after() = { }
;;

spawn before(); after()
;;
```

Se si esclude il fatto che `before` ha un tipo sincrono, le uniche differenze rispetto al codice di prima stanno nei thread etichettati `__00000002` e `__00000000`:

```
00c4:    __00000002:
00c4:      LOAD  before (4,s-port)
00d0:      GET-CONT
00d4:      SEND  (1,a-port)
00e0:      RECV
00e4:      LOAD  after (0,a-port)
00f0:      SEND  (0)
00f8:      EXIT

0100:    __00000000:
0100:      LOAD  @before (0,a-port)
010c:      SEND  (0)
0114:      EXIT
```

Come prima, durante la `SEND` alla porta `before` (indirizzo `00d4`) si invia nel messaggio una porta di continuazione che è esattamente la porta di continuazione del thread. L'identificatore di tale porta viene inserito sullo stack tramite la nuova istruzione `GET-CONT`.

Il corpo della `before` (indirizzo `0100`) è fondamentalmente identico al precedente, se non che il segnale di continuazione viene inviato ad una porta con nome `@before`. In pratica, per ogni porta sincrona che compone un pattern, il compilatore introduce un parametro invisibile, la porta di continuazione, che prima passavamo esplicitamente, e chiama questo parametro con lo stesso nome della porta sincrona preceduto dal carattere `@`.

L'istruzione `RECV` (indirizzo `00e0`) blocca il thread che la esegue finché non arriva un messaggio sulla porta di continuazione del thread stesso. Il messaggio ricevuto (nell'esempio è un segnale, ma in generale può essere un messaggio qualsiasi) viene spinto sullo stack, con il primo elemento più in profondità e l'ultimo elemento in cima.

2.3.5 Il costrutto *join let*

Come abbiamo visto, il costrutto `join let` consente di valutare più espressioni in parallelo e di assegnare i risultati delle valutazioni a nomi. Fino ad

ora abbiamo visto come creare i thread e come comunicare con le istruzioni `SEND` e `RECV`, ma tutto ciò non è ancora sufficiente per una implementazione "economica" del costrutto.

L'ideale sarebbe poter usare la porta di continuazione del thread che esegue il codice del costrutto per ricevere i risultati inviati dai singoli thread dedicati alle valutazioni. Purtroppo però, supponendo di aver già generato gli n thread dedicati al calcolo delle espressioni, non è sufficiente eseguire n `RECV`, perché non è noto l'ordine di terminazione degli stessi thread e quindi l'ordine di arrivo dei messaggi di risposta.

Quello che serve è una forma "selettiva" di `RECV`, che rimanga in attesa di un messaggio proveniente da una *specifica* sorgente. Così nasce l'istruzione `RECVFROM`, che blocca il thread che la esegue in attesa di un messaggio da uno specifico thread. Il thread da cui si attende il messaggio è identificato dal suo DTD.

A questo punto serve un sistema per conoscere l'identificatore di DTD di un thread creato, informazione che la `SPAWN` già vista non rende disponibile. Si introduce allora una leggera variante della `SPAWN`, detta `SPAWN-ID`, che lascia sullo stack del thread padre l'identificatore richiesto. Vediamo un semplice esempio di codice per una `join let`:

```
join let one: int = 1,
         two: int = 1 + 1
;;

0028:    THREAD  __00000000  (0,0)
003c:    SPAWN-ID
0040:    THREAD  __00000001  (0,0)
0054:    SPAWN-ID
0058:    RECVFROM
005c:    STORE two (8,int)
0068:    RECVFROM
006c:    STORE one (12,int)
0078:    EXIT

007c:    __00000001:
007c:      PUSH  1 (1)
0084:      PUSH  1 (1)
```

```
008c:       ADD
0090:       SEND  (1,int)
009c:       EXIT

00a0:     __00000000:
00a0:       PUSH  1 (1)
00a8:       SEND  (1,int)
00b4:       EXIT
```

Come al solito bisogna tenere presente che gli identificatori di DTD vengono inseriti in uno stack, per cui le `RECVFROM` e le relative `STORE` vengono eseguite in ordine inverso rispetto all'ordine di generazione dei thread.

Occorre invece soffermarsi sui thread `__00000001` e `__00000000`, i quali terminano dopo aver inviato il risultato della valutazione. Ma a quale porta sono destinati questi messaggi? Nel codice non c'è traccia di istruzioni `GET-CONT`. In effetti è ancora l'istruzione `SPAWN-ID` che spinge sullo stack del thread appena creato l'identificatore della porta di continuazione del padre.

A questo punto ci si potrebbe domandare perché per il costrutto `join let` non è stato fatto lo stesso tipo di ottimizzazione visto per la composizione parallela di più processi. In altre parole, se bisogna valutare n espressioni in parallelo, una la può valutare il thread padre, mentre le rimanenti $n - 1$ possono essere a carico di thread figli.

Il problema sorge se l'espressione che il thread padre si incarica di valutare (dopo aver generato gli altri thread) contiene al suo interno la chiamata ad un canale sincrono. In questo caso, come si è visto, il codice contiene una istruzione `RECV` che blocca il thread in attesa di un messaggio. Ma così viene a mancare quell'ipotesi per cui quando un thread invoca un canale sincrono, la sua porta di continuazione non è "in uso" da nessun altro thread. Siccome la `RECV` è "anonima", ovvero blocca il thread in attesa di un messaggio da chiunque, potrebbe generarsi confusione tra i messaggi provenienti dalla chiamata alla porta sincrona ed i messaggi dei thread figli che nel frattempo hanno terminato la valutazione.

2.3.6 Le primitive

Nella breve introduzione al linguaggio abbiamo visto come dichiarare primitive attraverso la parola chiave `external`. Abbiamo anche visto che l'invocazione

di queste primitive avviene esattamente come se fossero porte, ed infatti il codice generato in questi casi è esattamente lo stesso visto per le normali porte definite.

Tuttavia ogni porta, per poter essere utilizzata, deve essere definita in una qualche *join-definition*, oppure deve essere la porta di continuazione di un thread. Le "porte di primitiva" non ricadono in nessuna di queste due categorie e quindi è richiesto un metodo per avere il loro identificatore attraverso un nome simbolico.

L'istruzione `EXTERN` fa esattamente questo: essa prende come argomento una sequenza di caratteri[3] e spinge sullo stack l'identificatore di una porta che servirà ad invocare la primitiva.

In tutti gli esempi visti sinora il codice non cominciava mai all'indirizzo 0 in quanto è stata sempre omessa la dichiarazione di due primitive implicite che il compilatore inserisce in ogni programma, nella sezione 2.4.1 vedremo esempi completi di codice.

2.4 Locazioni

Dal punto di vista della macchina virtuale una locazione è una collezione di thread che condividono uno *heap* ed una tabella delle porte. Per questo si può pensare alla locazione come ad una generalizzazione di un programma sequenziale.

2.4.1 Lo heap

Lo heap è un'area di dimensione illimitata condivisa da tutti i thread di una locazione. L'accesso allo heap avviene attraverso dei "puntatori virtuali", ovvero dei numeri interi che rappresentano una certa posizione nello heap. La prima locazione dello heap ha indirizzo 0.[4]

In generale l'utilizzo dello heap è di competenza del modulo runtime in quanto la macchina virtuale non mette a disposizione istruzioni per la copia

[3] In questo momento non possiamo essere più precisi perché non abbiamo ancora visto come vengono gestite le stringhe, tutto diverrà più chiaro nella sezione 2.4.1.

[4] Quindi, a differenza dei puntatori in C, 0 è un indirizzo valido per lo heap, mentre l'equivalente di NULL è -1.

o lo spostamento di dati in quest'area, tuttavia alcune istruzioni assumono implicitamente che sia presente una struttura di questo tipo.

L'aspetto più evidente riguarda la gestione delle stringhe: il compilatore tratta una stringa come se fosse un semplice puntatore all'interno dello heap e il pool delle stringhe costanti di una locazione è posizionato proprio all'inizio dello stesso heap.

Nell'esempio che segue vengono "assegnate" due stringhe ai nomi `a` e `b`:

```
let a:string = "Hello,",
    b:string = "world!"
;;
```

Il codice compilato risulta:

```
0000:    EXTERN  0 (0)
0008:    EXTERN  12 (c)
0010:    STORE lookup (4,s-port)
001c:    STORE register (0,s-port)
0028:    PUSH  20 (14)
0030:    STORE a (8,string)
003c:    PUSH  28 (1c)
0044:    STORE b (12,string)
0050:    EXIT

0000: 72 65 67 69 ... 6c 6f 6f 6b  register....look
0010: 75 70 00 00 ... 77 6f 72 6c  up..Hello,..worl
0020: 64 21 00 00                  d!..
```

Finalmente possiamo notare la dichiarazione implicita per le primitive `lookup` e `register`. Gli argomenti delle istruzioni `EXTERN` sono proprio puntatori nello heap (mostrato di seguito al codice). Notiamo infine che i valori memorizzati in `a` e in `b` non sono le stringhe vere e proprie, ma puntatori alle stringhe.

Di fatto non ci sono limitazioni, dal punto di vista della macchina virtuale, su cosa possa essere memorizzato nello heap. Ad esempio, il supporto runtime può decidere di allocare qui le aree dati dei thread generati.

Un uso più direttamente correlato con la macchina virtuale riguarda invece la comunicazione. Le istruzioni `SEND` funzionano allo stesso modo sia per porte

locali (cioè definite all'interno della locazione) sia per porte remote (cioè definite in altre locazioni). Abbiamo visto che le stringhe sono in realtà trattate come puntatori all'interno dello heap, ma cosa accade se si invia una stringa in un messaggio? Se la porta destinazione appartiene alla stessa locazione allora il puntatore trasmesso mantiene un significato consistente, in quanto i thread della stessa locazione condividono tutti un unico heap. Ma se la porta di destinazione appartiene ad una locazione differente il puntatore presente nel messaggio non avrà alcun significato una volta giunto a destinazione.

Una soluzione può essere quella di memorizzare nel messaggio l'intera stringa, e non il suo puntatore. Una volta che il messaggio è arrivato a destinazione la stringa viene copiata nello heap locale dove "guadagna" un puntatore valido, che sarà quello usato per accedervi.

Nel capitolo 3 verranno prese in esame più in dettaglio tutte le problematiche della comunicazione e in particolare questi "messaggi intelligenti" creati attraverso il processo di *marshalling*.

2.4.2 La tabella delle porte

Non è difficile immaginare come ogni porta creata dalla macchina astratta abbia uno "stato" da mantenere in un qualche descrittore. Ad esempio, è logico supporre che ci sia una coda dei messaggi inviati alla porta, ma non ancora consumati. Anche questo descrittore avrà un'occupazione di memoria nell'ambito della locazione di appartenenza e, proprio come lo heap o i thread, esso dovrà spostarsi ad ogni migrazione della locazione. Visto che le porte vengono create dinamicamente i loro descrittori potrebbero essere allocati nello heap, ma per ragioni di efficienza dell'implementazione risulta più conveniente avere una struttura separata (ma altrettando dinamica) in cui memorizzare i descrittori delle porte definite. Tale struttura prende il nome di *tabella delle porte*.

Un identificatore di porta è semplicemente un indice per questa tabella.

2.4.3 Creazione di una nuova locazione

Ora che abbiamo introdotto le principali componenti di una locazione passiamo ad esaminare le istruzioni per crearne una nuova. Come prima considerazione notiamo che anche per le locazioni sono ammesse definizioni mutuamente ricor-

sive, ovvero in cui le porte definite inizialmente in una locazione sono visibili in tutte le altre locazioni della stessa definizione.

Dunque, anche in questo caso è stato scelto di operare una generazione a due fasi in cui prima si creano tutte le locazioni e tutte le porte definite al loro interno, e solo in un secondo momento si creano i pattern e si eseguono i processi di inizializzazione.

Nell'esempio che segue si crea una locazione con al suo interno un'unica porta:

```
location new_loc
        succ: int -> int
where   new_loc {
            with succ(x) = { reply x + 1 }
        }
;;
```

Il codice compilato risulta:

```
0028:    LOCATION  1,20,0
0038:    STORE new_loc (8,location)
0044:    STORE succ (12,s-port)
0050:    LOAD  new_loc (8,location)
005c:    THREAD  __00000001  (4,0)
0070:    SPAWN-LOC
0074:    RECV
0078:    SEND  (0)
0080:    EXIT

0084:    __00000001:
0084:      STORE succ (0,s-port)
0090:      THREAD  __00000000  (8,0)
00a4:      PATTERN [0]
00b4:      SEND  (0)
00bc:      RECV
00c0:      EXIT

00c4:      __00000000:
00c4:        LOAD  @succ (0,a-port)
```

```
00d0:        LOAD  x (4,int)
00dc:        PUSH  1 (1)
00e4:        ADD
00e8:        SEND  (1,int)
00f4:        EXIT
```

L'istruzione `LOCATION` provvede a creare una nuova locazione "vuota" (cioè priva di processi in esecuzione), ma ha tre argomenti che specificano, nell'ordine, il numero di porte definite inizialmente nella locazione (nel nostro caso una, corrispondente a `succ`) e di seguito l'indirizzo e la dimensione in byte nello heap del padre del pool di costanti della locazione figlia.[5]

La `LOCATION` spinge sullo stack del thread padre tanti identificatori di porta quanti ne sono stati definiti inizialmente ed infine spinge un identificatore per la locazione appena creata. Successivamente il padre memorizza nella propria area dati il valore di questi identificatori (nel solito ordine inverso).

A questo punto occorre generare il processo di inizializzazione della locazione (quello che nel linguaggio viene eventualmente specificato di seguito alla parola chiave `init`). Per fare questo si crea un thread nel modo usuale, ma per eseguirlo si usa l'istruzione `SPAWN-LOC` che genera il thread nell'ambito della locazione voluta (l'istruzione `SPAWN-LOC` prende due parametri dallo stack, l'identificatore di una locazione e l'identificatore di STD di un thread).

Il compito della `SPAWN-LOC` non si esaurisce qui: essa spinge sullo stack del thread padre l'identificatore della porta di continuazione del thread figlio, mentre sullo stack del thread figlio spinge prima l'identificatore della porta di continuazione del padre e successivamente gli identificatori delle porte definite inizialmente nella locazione, proprio come aveva fatto in precedenza la `LOCATION` sullo stack del padre.

Il perché di quest'ultima operazione è presto spiegato: la `LOCATION` ha solo creato le porte nella locazione, ma non i pattern (notare che il thread di inizializzazione non ha alcuna istruzione `PORT`, vedi indirizzo `0084`). La possibilità di avere definizioni mutuamente ricorsive ha richiesto un meccanismo, come questo, che consente la creazione delle porte (e in particolare la conoscenza dei rispettivi identificatori) *prima* della definizione effettiva delle locazioni. In

[5] Il compilatore individua sempre il potenziale indirizzo di inizio del pool, anche se poi risulta avere dimensione nulla. Nell'esempio il pool della nuova locazione comincerebbe all'indirizzo `20`.

una *join-definition* "classica" gli identificatori nello stack del thread vengono inseriti dall'istruzione `PORT`, che qui manca e il cui compito è così assolto dalla `SPAWN-LOC`.

La motivazione dello "scambio" di porte di continuazione tra il thread padre ed il figlio è qualcosa di più sottile e strettamente legato all'implementazione. Sospendiamo il discorso per un attimo e notiamo che padre e figlio eseguono le istruzioni `SEND` e `RECV` in successione (ma in ordini diversi) a un certo punto della loro esecuzione. In altri termini, padre e figlio si sono *sincronizzati*.

Nell'esempio la sincronizzazione sembra effettivamente inutile, ma le cose cambiano quando più locazioni vengono definite nello stesso costrutto:[6]

```
location a
    square: int->int
and b
    cubic: int->int
where
    b {
        with cubic(x) = { reply square(x) * x }
    }, a {
        with square(x) = { reply x * x }
    }
;;
```

In pratica si creano due locazioni (`a` e `b`) che definiscono inizialmente le porte `square` e `cubic` dove il processo associato alla seconda porta fa uso della prima. Il fatto che le definizioni siano invertite rispetto alle dichiarazioni serve solo per evidenziare il problema, ma non si pensi che una definizione "ordinata" lo risolva. Il codice compilato risulta:

```
0028:    LOCATION  1,20,0
0038:    STORE b (16,location)
0044:    STORE cubic (20,s-port)
0050:    LOCATION  1,20,0
0060:    STORE a (8,location)
006c:    STORE square (12,s-port)
0078:    LOAD  b (16,location)
```

[6]Purtroppo l'esempio è un po' complesso, ma in questo caso non si può fare altrimenti.

```
0084:    LOAD  square (12,s-port)
0090:    THREAD  __00000001  (8,1,s-port)
00a8:    SPAWN-LOC
00ac:    RECV
00b0:    LOAD  a (8,location)
00bc:    THREAD  __00000003  (4,0)
00d0:    SPAWN-LOC
00d4:    RECV
00d8:    SEND  (0)
00e0:    SEND  (0)
00e8:    EXIT

00ec:      __00000003:
00ec:        STORE square (0,s-port)
00f8:        THREAD  __00000002  (8,0)
010c:        PATTERN [0]
011c:        SEND  (0)
0124:        RECV
0128:        EXIT

0164:      __00000001:
0164:        STORE cubic (4,s-port)
0170:        LOAD  square (0,s-port)
017c:        THREAD  __00000000  (12,1,s-port)
0194:        PATTERN [0]
01a4:        SEND  (0)
01ac:        RECV
01b0:        EXIT
```

Le istruzioni agli indirizzi `0028-006c` creano le due locazioni e registrano il valore degli identificatori nell'area dati del padre. Segue la generazione del thread di inizializzazione per la locazione `b`. Notare che in `b` si usa il nome `square` che essendo esterno va passato nell'ambiente. Il padre conosce l'identificatore di `square` proprio grazie al meccanismo di generazione a due fasi.

Ora viene la parte interessante: all'indirizzo `00ac` il padre esegue una `RECV`, portandosi quindi nello stato di "bloccato", e l'unico thread che può risvegliarlo è quello che ha appena creato. Infatti, all'indirizzo `01a4`, il thread figlio esegue una `SEND` sulla porta di continuazione del padre, sbloccandolo, e subito dopo

si blocca a sua volta su una `RECV`. Notare che la `SEND` è stata eseguita *dopo* la creazione dei pattern.

A questo punto il padre è di nuovo in condizione di proseguire, genera quindi il thread di inizializzazione per la locazione `a` e di nuovo si sincronizza con esso. Questa volta però esegue anche due `SEND` che di fatto sbloccano entrambi i thread figli.

Tutto questo lavoro è volto ad un unico scopo: evitare che un thread utilizzi una porta quando quest'ultima non è ancora stata "inizializzata" per mezzo dell'istruzione `PATTERN`. Il fatto che una porta venga creata (per mezzo di `PORT` o di `LOCATION`) non autorizza a pensare che il sistema runtime sia pronto ad accettare messaggi su di essa, in quanto non è ancora noto in quanti e quali pattern essa compare. Con queste sincronizzazioni, invece, si dà il "via libera" ai thread solo quando si è sicuri che tutte le porte definite sono state "inizializzate" per mezzo di una `PATTERN`.

2.4.4 Migrazione

L'ultimo aspetto che riguarda le locazioni dal punto di vista della macchina astratta è la migrazione, che viene effettuata attraverso l'istruzione `go` tradotta dal compilatore nella primitiva omonima:

```
location here
where    here {
             (* circular migration! *)
             init go(here);
         }
;;
```

L'identificatore della locazione destinataria viene prelevato dallo stack del thread che esegue l'istruzione:

```
0000:    EXTERN  0 (0)
0008:    EXTERN  12 (c)
0010:    STORE lookup (4,s-port)
001c:    STORE register (0,s-port)
0028:    LOCATION  0,20,0
0038:    STORE here (8,location)
0044:    LOAD  here (8,location)
```

```
0050:    LOAD  here (8,location)
005c:    THREAD  __00000000  (4,1,location)
0074:    SPAWN-LOC
0078:    RECV
007c:    SEND  (0)
0084:    EXIT

0088:      __00000000:
0088:        SEND  (0)
0090:        RECV
0094:        LOAD  here (0,location)
00a0:        GO
00a4:        EXIT
```

2.5 Eccezioni

Durante l'esecuzione possono verificarsi delle condizioni di errore dette *eccezioni*. Un tipico esempio di eccezione è la divisione per zero. Nel nostro linguaggio anche la migrazione di una locazione ad una locazione figlia della locazione stessa è considerata una eccezione.

Il thread che causa una eccezione viene terminato.

Capitolo 3

Il supporto runtime

In questo capitolo analizzeremo in dettaglio l'architettura del supporto runtime che implementa la macchina virtuale.

3.1 Architettura generale

Il supporto runtime è un'applicazione *multithreaded* realizzata usando la libreria di thread POSIX [RR96] messa a disposizione dal sistema operativo Solaris (una brevissima descrizione delle funzioni principali viene data nell'appendice B).

Il termine "thread" ora diviene potenziale fonte di confusione in quanto è relativo a due tipi di entità simili e contemporaneamente molto differenti. Nel corso del capitolo, quando non risulterà chiaro dal contesto, si farà riferimento a *thread utente* come ai thread così come sono stati definiti nel capitolo 2, ovvero le unità elementari di esecuzione della macchina virtuale. I *thread di sistema* sono invece i thread POSIX che implementano le varie parti del supporto runtime.

Inizialmente avevamo pensato di associare un thread di sistema ad ogni thread utente, ma poi l'idea è stata abbandonata per evitare di appesantire l'implementazione che invece trae vantaggio dal fatto che la granularità di scheduling della macchina virtuale è la singola istruzione. In altre parole, un cambiamento di thread può avvenire solo al confine tra due istruzioni, ma non all'interno delle stesse. Se avessimo usato i thread di sistema per implementare i thread utente non avremmo avuto alcun modo efficiente per controllare lo

scheduling, anzi quest'ultimo avrebbe potuto interrompere l'esecuzione delle istruzioni, rendendo necessario il ricorso alla mutua esclusione per proteggere le numerosissime sezioni critiche.

Nella figura 3.1 possiamo osservare una visione semplificata del supporto runtime.

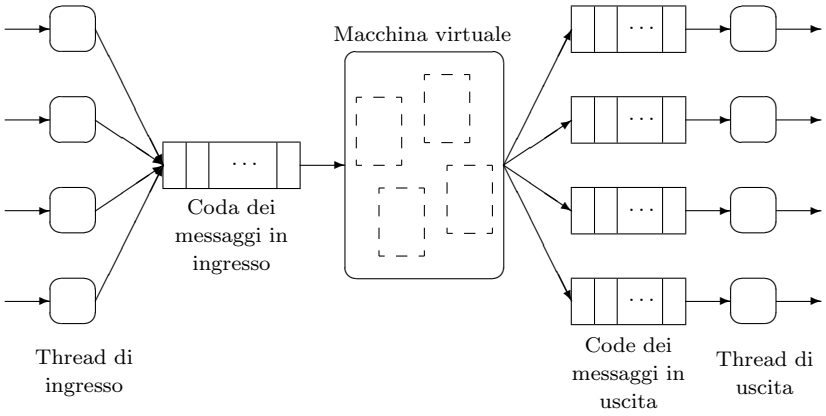

Figura 3.1: Architettura del supporto runtime

Sulla sinistra sono presenti, in numero variabile, i thread di ingresso che si occupano della ricezione dei messaggi provenienti da altri runtime. Questi messaggi vengono inseriti in una coda unica ed estratti con disciplina FIFO.

Il thread principale è quello che implementa la macchina virtuale e si occupa anche dell'instradamento dei messaggi estratti dalla coda di ingresso. Al suo interno sono rappresentati (con linee tratteggiate) alcuni thread utente il cui scheduling viene "simulato" opportunamente.

I messaggi generati dai thread della macchina virtuale e destinati ad altri runtime vengono inseriti nelle code relative alle corrette destinazioni. In particolare, per ogni connessione con un runtime, esiste una coda dei messaggi di uscita ed un thread di sistema che si occupa della spedizione di questi messaggi.

3.2 Implementazione della macchina virtuale

3.2.1 Il ciclo di esecuzione principale

Il cuore della macchina virtuale è composto da una semplice funzione, `run`, implementata nel file `scheduler.c`. Questa funzione realizza il ciclo di esecuzione principale che interpreta le istruzioni dei processi.

Di seguito vediamo una versione semplificata della funzione `run` in cui le righe sono state numerate:

```
 1: PUBLIC void run() {
 2:    char *pc;
 3:
 4:    while (!kill_runtime) {
 5:       running = wait_event();
 6:
 7:       if (running != NULL) {
 8:          do
 9:             pc = exec_opcode[fetch_opcode(pc)]();
10:          while (pc != NULL);
11:       }
12:
13:       if (in_queue_size() > 0) process_incoming_messages();
14:    }
15: }
```

Il ciclo che inizia alla riga 4 è il ciclo di esecuzione principale che, durante l'esecuzione, non viene mai abbandonato fino alla terminazione del supporto runtime. Il ciclo ripete una sequenza di tre azioni, la prima delle quali consiste nell'invocazione della funzione `wait_event` che blocca il runtime fino a quando non si verifica almeno una delle seguenti condizioni:

1. La coda dei processi pronti contiene almeno un elemento. In questo caso il DTD del primo processo nella coda viene restituito e assegnato alla variabile globale `running`, che rappresenta il registro del processo corrente.

2. La coda dei messaggi in ingresso contiene almeno un elemento. In questo

caso l'invocazione di `in_queue_size` alla linea 13 restituirà un valore positivo.

3. Il runtime è terminato. Questa condizione si verifica essenzialmente quando non ci sono più né processi pronti, né processi bloccati.

Quando la chiamata a `wait_event` ritorna un DTD valido in `running`, allora va in esecuzione il ciclo alle linee 8–10, che è il ciclo che si occupa dell'interpretazione delle istruzioni. `exec_opcode` è un array di puntatori alle funzioni che implementano le istruzioni della macchina virtuale. Questo array viene indicizzato tramite il codice operativo dell'istruzione stessa, che viene letto dal segmento codice tramite la funzione `fetch_opcode`.

Il ciclo interno di esecuzione si arresta solo quando il *program counter* `pc` diviene `NULL`, il che accade quando il processo `running` termina oppure si blocca su un'istruzione `RECV` o `RECVFROM`.

Non ci sembra il caso di descrivere singolarmente la struttura delle funzioni che implementano le istruzioni della macchina virtuale; la maggior parte di queste sono molto semplici ed intuitive. Piuttosto descriveremo le strutture dati che tali funzioni utilizzano e modificano, strutture che riflettono in gran parte quelle descritte in modo più o meno astratto nel corso del capitolo precedente.

3.2.2 Thread utente: descrittori statici e dinamici

Un descrittore STD viene creato dall'istruzione `THREAD` e costituisce il primo passo per la generazione di un thread utente. La sua struttura è la seguente:

```
typedef struct static_thread_descriptor {
        unsigned pc;
        unsigned code_size, data_size, env_size, env_real_size;
        char     *env;
} STD;
```

Il campo `pc` rappresenta l'indirizzo della prima istruzione che compone il thread, cioè il valore iniziale assunto dall'omonimo campo nel DTD. `code_size` e `data_size` rappresentano la dimensione del codice e dell'area dati (non comprensiva dell'eventuale ambiente). `env` è un puntatore al messaggio che contiene l'ambiente da passare al thread al momento della sua generazione. I

campi `env_size` e `env_true_size`, sebbene ridondanti in quanto disponibili indirettamente attraverso lo stesso ambiente `env`, vengono duplicati per maggior efficienza e rappresentano rispettivamente la dimensione dell'ambiente nell'area dati e la dimensione complessiva del messaggio `env`, la cui struttura sarà esaminata più in generale quando parleremo delle istruzioni per la comunicazione.

Il processo di generazione di un thread, avviato da una istruzione della classe `SPAWN` oppure dall'attivazione di un pattern, ha come risultato la creazione di un DTD la cui struttura è la seguente:

```
typedef struct dynamic_thread_descriptor {
        VPTR      data_ptr;
        unsigned  stack_size, stack_top;
        unsigned  pc;
        VM_PORTID cont;
        VM_PORTID id;
        unsigned  event_mask;
        VM_PORTID wait_dtd;
        VM_PORTID first_created;
        int       new_loc;
        VM_WORD   *stack;
        struct location_entry *loc;
        struct dynamic_thread_descriptor *next;
} DTD;
```

Il campo `data_ptr` è un puntatore all'area dati del thread. Notare che questo puntatore ha tipo `VPTR`, in quanto è un puntatore "virtuale" che fa riferimento allo heap della locazione di appartenenza del thread. Quindi, come avevamo anticipato, le aree dati vengono allocate dinamicamente in questa zona. `id` è l'identificatore dello stesso DTD, esattamente la quantità lasciata sullo stack dall'istruzione `SPAWN-ID`. I campi `stack`, `stack_size` e `stack_top` inquadrano la situazione dello stack, ovvero la sua locazione in memoria, la dimensione attuale, e la prima posizione libera.

`pc` è un puntatore all'interno del segmento codice del thread alla successiva istruzione da eseguire. `pc` e `stack_top` sono quindi i due registri che abbiamo definito "locali" nel capitolo 2.

`cont` rappresenta l'identificatore della porta di continuazione del thread, ovvero la quantità lasciata sullo stack dall'istruzione `GET-CONT`. Di conseguen-

za il suo tipo, `VM_PORTID`, rappresenta un indice nella tabella delle porte della locazione di appartenenza. Anche il campo `id` già descritto ha come tipo `VM_PORTID`, pur non essendo propriamente un indice di porta. In effetti, nell'ambito del supporto runtime, la tabella delle porte assume un ruolo più generale che consente di associare un *identificatore* a diverse strutture fondamentali, come i DTD. Si potrebbe obiettare che il puntatore fisico ad una struttura rappresenta già di per sé un identificatore univoco e sicuramente maneggevole. Il problema è che un puntatore fisico ha un significato preciso solo all'interno dello stesso supporto runtime, mentre un identificatore nella tabella delle porte conserva un significato su ogni supporto runtime in cui migra la locazione. Il ruolo generalizzato della tabella delle porte sarà descritto in dettaglio nella sezione 3.2.9.

Il campo `loc` è un puntatore al descrittore della locazione di appartenenza, mentre `next` è un puntatore usato dallo scheduler per collegare tra loro i thread nella coda dei pronti. In effetti, gli elementi di questa coda sono proprio dei DTD.

3.2.3 Porte e join pattern

Ogni porta creata per mezzo dell'istruzione `PORT` (o `LOCATION`, ma il principio è identico), ha un descrittore relativo nella tabella delle porte. Tuttavia, raramente accade che una porta ricopra un ruolo isolato, mentre più spesso appartiene ad un pattern composto da diverse porte che si "influenzano" reciprocamente. Ciò suggerisce che tutte le porte di uno stesso pattern, o meglio di una stessa *join-def*, debbano far riferimento ad una struttura comune detta *descrittore di pattern*. Questo descrittore, essendo di dimensioni relativamente grandi, non può essere memorizzato come elemento nella tabella delle porte, ma viene referenziato attraverso un opportuno identificatore.

In conclusione, ogni porta che compare in una *join-definition* ha un'entrata distinta nella tabella delle porte, ma che a sua volta fa riferimento ad un unico descrittore di pattern.

Porte convenzionali

La struttura semplificata di un elemento nella tabella delle porte è la seguente:

```
typedef struct port_entry {
```

```
        PORT_CLASS class;    /* P_CONV */
        union {
                ...
                struct {
                        VM_PORTID pattern;
                        int id;
                } conv;
                ...
        } u;
} PORT;
```

`class` distingue la classe dei vari descrittori che possono comparire nella tabella delle porte. Le porte che compaiono in una *join-def* sono dette "convenzionali". Come si nota, il descrittore contiene l'identificatore del descrittore di pattern condiviso con le altre porte della stessa *join-definition* e l'identificatore della porta relativo alla definizione stessa (esattamente lo stesso numero che compare nell'istruzione `PATTERN`).

L'attenzione si sposta allora alla struttura di un descrittore di pattern:

```
typedef struct pattern_descriptor {
        int             n_patt;
        int             n;
        int             matrix[MAX_PATTERNS][MAX_PORTS/INT_BITS];
        int             cur_pattern[MAX_PORTS/INT_BITS];
        VMESSAGE_QUEUE *queue[MAX_PORTS];
        VM_PORTID       thread[MAX_PATTERNS];
        VPTR            sequence[MAX_PATTERNS];
} PATTERN;
```

Nell'implementazione abbiamo preferito mantenere le cose semplici ed abbiamo imposto una dimensione massima per le *join-definition*, sia in termini di numero di porte che di numero di pattern. Il sistema di pattern matching, comunque, non dipende da queste restrizioni e può essere agevolmente adattato per casi più generali.

I campi `n_patt` ed `n` rappresentano dunque il numero effettivo di pattern e porte che compongono la *join-definition*, `matrix` è la nota matrice di bit che rappresenta i pattern, mentre `cur_pattern` è una stringa di bit che rappresenta

le porte sulle quali è presente almeno un messaggio. Questa rappresentazione è particolarmente comoda dal punto di vista implementativo in quando consente un controllo molto efficiente per l'attivazione dei pattern, usando esclusivamente gli operatori logici *and* e *xor*.

`queue` è un'array di code di messaggi, una per ogni porta che compare nella definizione, mentre `thread` è un array (se vogliamo ortogonale al precedente) che contiene, per ogni pattern, l'STD del thread da generare quando lo stesso pattern si attiva.

Il campo `sequence` riguarda un aspetto puramente implementativo che può essere aggirato con una diversa organizzazione del compilatore. La sua funzione è quella di contenere dei puntatori alle sequenze di porte che compongono ogni pattern. La matrice di bit vista prima fornisce già questa informazione, ma non dice in quale *ordine* le porte compaiano nel pattern. L'ordine, nel nostro caso, è importante perché il compilatore alloca lo spazio per le variabili ricevute nell'ordine di apparizione delle porte relative. Un esempio servirà a chiarire il concetto.

Supponiamo che un pattern sia composto da tre porte e che ciascuna accetti numeri interi come messaggi. Supponiamo anche che l'ordine di apparizione delle porte nel pattern non ne rispecchi la sequenzialità degli indici:

```
def     a, b, c: int ->
where   c(x) | b(y) | a(z) = { ... }
;;
```

Il codice risultante sarebbe:

```
0028:    PORT  3
0030:    STORE c (16,a-port)
003c:    STORE b (12,a-port)
0048:    STORE a (8,a-port)
0054:    THREAD  __00000000  (12,0)
0068:    PATTERN [2,1,0]
```

Nell'ambito della definizione la porta `a` ha indice 0, la porta `b` ha indice 1 ed infine la porta `c` ha indice 2. Tuttavia, siccome la porta `c` compare per prima nel pattern, il compilatore alloca lo spazio per `x` per primo, in altre parole, visto che non c'è ambiente, `x` ha offset 0 nell'area dati, `y` ha offset 4

(supponendo di lavorare su una macchina a 32 bit) e `z` ha offset 8. Ora, quando il pattern si attiva significa che in ogni coda delle tre porte è presente almeno un messaggio. Il supporto runtime deve estrarre un messaggio da ogni coda e registrarlo nell'area dati. Per conoscere l'offset a cui memorizzare ciascun messaggio deve sapere con quale ordine il compilatore ha allocato lo spazio per le variabili ricevute, cosa possibile se il supporto runtime può accedere alla sequenza `[2,1,0]` dell'istruzione `PATTERN`. In conclusione, `sequence` è proprio un array di puntatori alle sequenze che compaiono nelle istruzioni `PATTERN`.

Porte veloci

Il supporto runtime opera una distinzione tra i pattern composti da un'unica porta ed i pattern composti da più porte. Nel primo caso, infatti, l'attivazione del pattern è contemporanea all'invio di un messaggio sull'unica porta costituente e l'implementazione può essere migliorata ottenendo un sostanziale incremento delle prestazioni.

Una porta viene definita "veloce" se compare sola in un pattern. In questo caso il descrittore creato nella tabella delle porte ha il seguente formato:

```
typedef struct port_entry {
        PORT_CLASS class;    /* P_FAST */
        union {
                ...
                struct {
                        VM_PORTID std_id;
                        int id;
                } fast;
                ...
        } u;
} PORT;
```

Il campo `id` non ha alcun utilizzo pratico, è presente per uniformità con le porte convenzionali, mentre `std_id` è l'identificatore di STD del thread da creare quando si invia un messaggio sulla porta. Notare in particolare che non serve più una coda di messaggi e tutto il meccanismo di pattern matching risulta enormemente semplificato. Questa ottimizzazione è stata pensata anche a fronte del particolare significato che hanno le porte veloci: come abbiamo visto,

i pattern con un'unica porta realizzano qualcosa di molto simile a procedure e funzioni comuni in altri linguaggi, ed è logico aspettarsi che questo tipo di pattern sia in qualche modo più numeroso degli altri.

3.2.4 Struttura dei messaggi e comunicazione

Tutti i messaggi scambiati con l'istruzione `SEND` (e lo stesso ambiente, che è un messaggio a sua volta) hanno una struttura generale comune rappresentata nella figura 3.2.

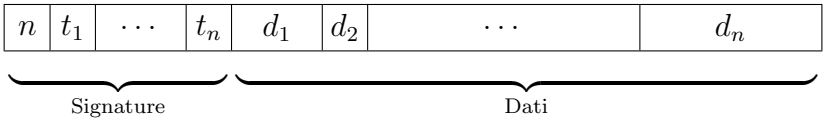

Figura 3.2: Struttura di un messaggio interno

In testa al messaggio è memorizzata la *signature* che indica il numero di elementi (n) ed il loro tipo ($t_1, \ldots, t_n$), seguita dai dati veri e propri ($d_1, \ldots, d_n$). La figura suggerisce che, mentre i campi che contengono i tipi (t_i) hanno tutti uguale dimensione, i campi che contengono i dati hanno dimensioni variabili e dipendenti dal tipo relativo.

Una coda di messaggi ha un descrittore con la forma seguente:

```
typedef struct vqueue {
        VMESSAGE_HEADER *first, *last;
        int n;
} VMESSAGE_QUEUE;
```

`n` è il numero di messaggi presenti nella coda, `first` e `last` sono puntatori al primo ed all'ultimo elemento. Ogni elemento nella coda è un header che contiene un puntatore al messaggio vero e proprio:

```
typedef struct a_message_header {
        struct a_message_header *next;
        VM_PORTID sender_id;
        unsigned  size;
        char      *msg;
} VMESSAGE_HEADER;
```

Tra le altre cose, notiamo che sono presenti un campo che contiene la dimensione del messaggio (`size`) ed un identificatore del mittente del messaggio stesso (`sender_id`), informazione che si rivela cruciale per l'implementazione dell'istruzione `RECVFROM`.

La comunicazione attraverso l'istruzione `SEND` prevede in generale la creazione di un messaggio come quello mostrato nella figura 3.2 in cui la *signature* è già disponibile come argomento dell'istruzione, mentre i dati vengono reperiti dallo stack. Nel caso di comunicazioni intra-locazione poi, le cose possono essere ulteriormente semplificate in quanto i dati da comunicare non hanno bisogno di alcuna trasformazione (*marshalling*).

Abbiamo già accennato al fatto che certi dati hanno un significato preciso solo nell'ambito della locazione in cui risiedono. Ad esempio, l'identificatore di una porta è in realtà un indice nella tabella delle porte della locazione, per cui avrà significato fintantoché rimarrà all'interno della locazione stessa. Analogo ragionamento può essere fatto per i puntatori all'interno dello heap.

Di conseguenza, in particolari situazioni in cui il marshalling non è richiesto, il supporto runtime non crea effettivamente dei messaggi, ma provvede semplicemente a spostare le informazioni inalterate prelevandole direttamente dallo stack del thread che esegue la `SEND`.

3.2.5 Le istruzioni bloccanti

Abbiamo visto che la macchina virtuale ha due istruzioni, `RECV` e `RECVFROM`, dal comportamento bloccante. Entrambe bloccano il thread che esegue l'istruzione in attesa che arrivi un messaggio sulla porta di continuazione del thread stesso. Abbiamo già visto come vengono trattate le porte create per mezzo di `PORT` o `LOCATION`. La porta di continuazione di un thread viene invece creata al momento della generazione. Il campo `cont` nel DTD del thread identifica la porta di continuazione che, nella tabella delle porte, ha il seguente descrittore:

```
typedef struct port_entry {
        PORT_CLASS class;    /* P_CONT */
        union {
                ...
                struct {
                        VM_PORTID dtd_id;
                        VMESSAGE_QUEUE *queue;
```

```
                } cont;
                ...
        } u;
} PORT;
```

Il campo `queue` è ovviamente un puntatore alla coda dei messaggi inviati sulla porta, mentre `dtd_id` è l'identificatore del DTD del thread a cui appartiene la porta di continuazione.

Quando un thread esegue l'istruzione `RECV`, il supporto runtime verifica se la coda `queue` è vuota. In questo caso il thread viene bloccato rimuovendo il suo DTD dalla coda dei pronti. Per notificare il blocco di un thread, il supporto runtime setta il campo `event_mask` nel suo DTD (vedi pagina 61) con il flag `WAIT_RECV`. Questo campo serve per distinguere il tipo di evento che farà risvegliare il thread.

L'istruzione `RECVFROM` si comporta in modo analogo, ma in questo caso il thread che si blocca attende un messaggio da una specifica sorgente. Il supporto runtime distingue questa situazione dalla precedente settando il campo `event_mask` con il flag `WAIT_RECVFROM` e inoltre memorizzando nel campo `wait_dtd` (sempre nel DTD) l'identificatore del thread da cui si attende un messaggio.

3.2.6 Le primitive

L'istruzione `EXTERN` prende come argomento un puntatore al nome della primitiva per la quale deve creare un riferimento ed alloca un descrittore nella tabella delle porte che ha la forma:

```
typedef struct port_entry {
        PORT_CLASS class;    /* P_EXTERN */
        union {
                ...
                struct {
                        VPTR name;
                        PRIMITIVE f;
                } extrn;
                ...
        } u;
} PORT;
```

Ogni primitiva è implementata nel supporto runtime attraverso una funzione scritta in C. Il campo `name` è il suddetto puntatore al nome simbolico della primitiva, mentre `f` è una piccola cache per migliorare le prestazioni: quando si invia per la prima volta un messaggio ad una porta di primitiva, il supporto runtime ricerca la funzione che la implementa in una apposita tabella (che descriveremo nella sezione 3.4) e registra il puntatore alla funzione trovata in `f`, in modo che agli accessi successivi non sia più necessario effettuare la ricerca.

Quando una locazione migra, tutte le entrate `f` presenti nella tabella delle porte vengono invalidate, forzando la ricerca alle invocazioni successive. Questo sistema, che realizza un binding dinamico, fa in modo che ovunque la locazione risieda, i riferimenti alle primitive vengano sempre risolti con le primitive locali.

3.2.7 Locazioni

La locazione è l'unità minima in movimento e in quanto tale necessita di un sistema particolare di identificazione. Quando si invia un messaggio ad una porta remota (cioè residente in una locazione distinta da quella da cui parte il messaggio), il supporto runtime deve prima di tutto individuare quale host ospita la locazione in cui è definita la porta, quindi deve indirizzare il messaggio verso l'host in questione. Ci penserà il supporto runtime remoto a portare a termine la consegna.

La cosa importante da evidenziare è che le locazioni, potendosi spostare e dovendo essere referenziate da un punto qualsiasi della rete, necessitano di un sistema di identificazione *globale*. In altre parole, è necessario che non esistano due locazioni con lo stesso identificatore.

Una situazione analoga la si può trovare in Internet, in cui è richiesto che ogni macchina abbia un indirizzo IP distinto [Ste90]. In Internet gli indirizzi vengono "emanati" da un organismo centrale e quindi, salvo errori o malizie, sono potenzialmente scongiurate collisioni di indirizzi. È ovvio che questo sistema non può essere applicato anche per le locazioni, che sono entità ben più dinamiche degli host in Internet (in effetti gli host non hanno alcuna parvenza di dinamicità, il problema dei sistemi mobili è tuttora in fase di analisi), però è possibile sfruttare gli indirizzi Internet per creare un ulteriore livello di indirizzamento a noi utile.

In pratica ogni host può ospitare un unico supporto runtime, questo fa

in modo che esista una corrispondenza uno a uno tra indirizzi IP e supporti runtime.[1]

Possiamo immaginare che il supporto mantenga un contatore sequenziale che incrementa ogni volta che "nasce" una nuova locazione, identificandola con la coppia costituita dall'indirizzo IP dell'host e dal valore del contatore al momento della nascita.

C'è solo un dettaglio che potrebbe rivelarsi fatale: se un runtime termina per qualche condizione anomala (un problema nel software oppure perché l'host che lo ospita "cade") il valore del contatore va perso, a meno che non venga appositamente memorizzato su un supporto di massa *fault tolerant* (un cosiddetto *stable storage*). Per rendere le cose più semplici, abbiamo scelto di inserire nell'identificatore di locazione il PID del processo che esegue il supporto runtime, in modo da differenziare successive esecuzioni del supporto sulla stessa macchina. Pur non essendo perfetto, riteniamo che il sistema sia più che soddisfacente. In conclusione, un identificatore di locazione è una tripla con la seguente struttura:

```
typedef struct location_id {
        struct in_addr  host;
        pid_t           process;
        int             counter;
} LOCATION_ID;
```

Se questo identificatore risolve i problemi di unicità, è causa invece di alcuni "disagi" dal punto di vista della macchina virtuale. Soprattutto non è facile da maneggiare viste le sue dimensioni (tra l'altro è più grande di un descrittore per la tabella delle porte), senza contare che sotto un certo aspetto è ancora incompleto. Infatti, nonostante identifichi univocamente una locazione, non dice ancora nulla su *dove* la locazione risieda in un determinato momento.

Il luogo più consono per la registrazione dell'identificatore è il descrittore associato ad ogni locazione, che ha la forma seguente:

```
typedef struct location_entry {
        struct location_entry   *mother, *daughter, *sister;
        LOCATION_ID     id;
```

[1] In realtà un host può avere più di un indirizzo IP, l'importante è che ad ogni indirizzo IP corrisponda *al più* un supporto runtime.

```
        unsigned        code_size, pool_size, heap_size;
        char            *code_ptr;
        char            *heap_ptr;
        unsigned        n_ports;
        unsigned        port_table_size;
        PORT            *port;
        VM_PORTID       freer;
        VPTR            freep;
} LOCATION;
```

Questo descrittore è a tutti gli effetti un elemento dell'albero delle locazioni all'interno di ogni runtime. I puntatori `mother`, `daughter` e `sister` sono rispettivamente i puntatori alla locazione madre, alla prima figlia, e alla lista delle locazioni sorelle. Gli altri campi servono per individuare lo stato della tabella delle porte, dello heap e naturalmente l'identificatore `id` della locazione.

Il problema del posizionamento delle locazioni sui vari runtime viene risolto mantenendo una tabella di instradamento del tutto analoga alle tabelle costruite con gli algoritmi di *routing* per il protocollo IP. Ogni runtime ha una tabella le cui entrate hanno la struttura seguente:

```
typedef struct routing_table_entry {
        struct routing_table_entry *next;
        struct location_entry *loc;
        LOCATION_ID    lid;
        struct in_addr addr;
} ROUTING_ENTRY;
```

La tabella ha un'organizzazione hash per renderne più efficiente l'accesso (il puntatore `next` è il puntatore per i conflitti). `loc` è un puntatore al descrittore della locazione se quest'ultima è locale al supporto runtime e vale `NULL` in caso contrario, `lid` è il consueto identificatore di locazione mentre `addr` rappresenta l'indirizzo IP dell'host sul quale il supporto runtime "suppone" che si trovi la locazione: a causa della natura intrinsecamente distribuita di tutto il sistema, questa informazione non può essere sempre precisa e aggiornata.

Dal punto di vista dei thread utente, un identificatore di locazione è ancora una volta un indice per un'entrata particolare della tabella delle porte che ha la struttura seguente:

```
typedef struct port_entry {
        PORT_CLASS class;    /* P_LOC */
        union {
                ...
                struct {
                        ROUTING_ENTRY *loc;
                } loc;
                ...
        } u;
} PORT;
```

Questo livello di indirezione ulteriore consente di individuare una locazione con un semplice indice (la cui dimensione è pari a quella degli altri indici e valori manipolati dalla macchina virtuale) e inoltre consente di mantenere tutte le informazioni sulle locazioni (in particolare l'indirizzo dell'host che le ospita) in un'unica struttura condivisa, la *tabella di instradamento*. In questo modo, quando il runtime viene a conoscenza di nuove informazioni sulla posizione di una locazione, deve aggiornare unicamente questa tabella, e tutti i thread che riferiscono la locazione usufruiscono in modo trasparente delle informazioni aggiornate.

3.2.8 Comunicazione inter-locazione

Uno degli aspetti critici che caratterizzano la coesistenza di più locazioni è la comunicazione dei messaggi. Per le comunicazioni locali (anche dette intra-locazione) il lavoro del supporto runtime è davvero minimo e si limita alla creazione di semplici messaggi oppure al trasferimento diretto dei dati. È stato fatto più volte notare invece come la comunicazione di messaggi tra locazioni differenti (comunicazione remota o anche inter-locazione) presenti delle difficoltà nuove che vanno gestite in modo opportuno.

Marshalling

Alcuni tipi di valori presenti nei messaggi hanno un significato preciso solo nell'ambito della locazione di appartenenza, mentre al di fuori di questa risultano incompleti oppure totalmente privi di senso. Cerchiamo allora di riassumere in una tabella i tipi di dato "delicati" da questo punto di vista:

Stringhe: le stringhe sono viste come puntatori nello heap della locazione, di conseguenza questi puntatori hanno significato solo all'interno della locazione stessa.

Porte: Il supporto runtime tratta le porte come indici per la tabella delle porte e locazioni diverse hanno tabelle diverse, per cui questi indici hanno un significato completo solo localmente.

Locazioni: Anche le locazioni vengono trattate come indici per la tabella delle porte, valgono quindi le stesse considerazioni fatte per le porte.

Gli altri tipi di dato (interi, booleani, caratteri) hanno invece un significato assoluto che non è legato ad altre strutture della macchina astratta e quindi possono essere comunicati in modo trasparente.

Dunque, il supporto runtime deve prestare particolare attenzione quando si cerca di comunicare dati "delicati" nei messaggi. Innanzi tutto è richiesto un modo per conoscere il tipo degli elementi che compongono un messaggio e questa informazione è disponibile grazie alla *signature* che compare in ogni istruzione di comunicazione (ovvero la `SEND`, ma anche la `THREAD`). La codifica che abbiamo scelto per la signature è esplicita e per certi versi dispendiosa in termini di spazio (vedi figura 3.2), ma ancora una volta abbiamo preferito la semplicità realizzativa all'accuratezza di una implementazione "seria".

Un'alternativa più accorta potrebbe prevedere la codifica del tipo di un dato all'interno del dato stesso in modo da rendere gli elementi di un messaggio "autodescrittivi". Ad esempio, ogni dato potrebbe "sacrificare" uno o più bit della sua rappresentazione binaria per indicarne il tipo. In genere il tipo più esigente è l'intero, in quanto è vantaggioso avere un intervallo disponibile il più ampio possibile, e quindi per gli interi si potrebbe sacrificare un solo bit, diciamo il più significativo. Ogni parola con il bit più significativo azzerato verrebbe interpretata come numero intero. Quando il medesimo bit viene trovato a 1, invece, si controllerebbe il bit successivo, usando così una specie di codice *expanding* comune nei codici operativi di molti microprocessori.[2] I dati con meno esigenze di spazio sarebbero ovviamente i booleani (che necessitano di un solo bit per la codifica).

Qualunque sia la codifica adottata, una volta noto il tipo di un elemento in un messaggio, il supporto runtime deve prendere adeguati provvedimenti

[2]Ovviamente 0 e 1 possono essere invertiti a discrezione di chi progetta il sistema.

affinché la comunicazione sia in qualche modo completa trasformando i dati in una rappresentazione "assoluta". A volte questo processo di trasformazione viene chiamato *marshalling* ed è comune negli ambienti che ammettono la chiamata di procedure remote quando le procedure stesse agiscono su dati dinamici o di grandi dimensioni.

Il processo di marshalling è ovviamente diverso in base al particolare tipo di dato da comunicare. Per le stringhe è ammissibile la soluzione proposta in precedenza che prevede il trasferimento della stringa vera e propria e non del suo puntatore. Nell'implementazione la stringa viene copiata nel messaggio preceduta da un intero che ne indica la lunghezza. Per le porte il problema è più sottile: una porta ha significato proprio perché è definita nel contesto di una locazione, non avrebbe senso tentare di comunicarne il descrittore, o il pattern, o il processo associato. Quello che si vuole far arrivare al destinatario del messaggio è l'indicazione di *dove* la porta sia definita, in modo che i messaggi spediti su quella porta possano arrivare fino alla locazione di definizione.

In realtà a questo punto abbiamo già tutti gli elementi che ci servono, bisogna solo comporli in modo opportuno. Intanto, visto che ogni locazione ha un identificatore unico, la coppia formata dall'identificatore della locazione di appartenenza e dall'indice della porta in questione rappresenta un identificatore assoluto (unico in tutta la rete) della stessa porta. Rimane un unico particolare: il supporto runtime di destinazione potrebbe non sapere qual è l'host che ospita la locazione con l'identificatore che stiamo inviando, è quindi opportuno inserire nel messaggio anche questa informazione, reperibile dalla tabella di instradamento.

Per quanto riguarda la comunicazione di indici di locazioni, ci troviamo in un sottocaso della comunicazione delle porte, è sufficiente copiare nel messaggio identificatore e host ospitante della locazione.

Unmarshalling

Una volta compresi i meccanismi di creazione dei messaggi, è il caso di vedere cosa accade una volta che i messaggi arrivano a destinazione. Innanzi tutto osserviamo che tutti i meccanismi di ricezione dei messaggi devono funzionare ugualmente bene sia per messaggi che provengono dall'esterno, sia per messaggi locali. Ma, come abbiamo appena visto, i messaggi remoti possono aver subito il processo di marshalling per cui è richiesto un sistema per sapere se

un elemento di un messaggio è stato trasformato oppure no. Ancora una volta viene in aiuto la signature che accompagna ogni messaggio. Nella signature c'è un campo per ogni elemento che ne indica il tipo. Se un elemento subisce il processo di marshalling, allora il corrispondente campo nella signature viene modificato dal supporto runtime settando un particolare bit. Il runtime a destinazione controlla i campi nella signature uno ad uno e, quando trova il bit che indica il marshalling settato, "sa" che l'elemento nel messaggio ha subito la trasformazione e va trattato opportunamente.

Resta appunto da vedere il trattamento dei dati trasformati da parte del supporto runtime. Per le stringhe il problema si risolve facilmente allocando in modo opportuno lo heap locale: l'indirizzo dell'area allocata servirà per accedere alla stringa (o meglio alla sua copia). Da notare che per le stringhe la comunicazione remota è a tutti gli effetti una duplicazione, non avviene di fatto alcuna condivisione dei dati.

Tutte le porte, in quanto tali, sono viste dal supporto runtime come indici per la tabella delle porte, quindi è logico aspettarsi che anche le porte ricevute da una locazione remota siano "tradotte" in un indice per la tabella locale. Il problema si sposta su cosa debba contenere il descrittore nella tabella alla posizione indicata da questo indice. Ovviamente non può trattarsi né di una porta convenzionale né di una porta veloce, questi descrittori sono validi solo per porte locali. Il descrittore per una porta remota dovrebbe fare da tramite in modo che i messaggi spediti su di essa vengano instradati verso la locazione di definizione. In pratica si definisce un alias che nella tabella delle porte ha la seguente struttura:

```
typedef struct port_entry {
        PORT_CLASS class;    /* P_REMOTE */
        union {
                ...
                struct {
                        ROUTING_ENTRY *loc;
                        VM_PORTID pid;
                } remote;
                ...
        } u;
} PORT;
```

La comunicazione dell'ambiente

Il passaggio dell'ambiente, anche se si tratta sempre di comunicazione, necessita di alcuni accorgimenti particolari. Nella maggior parte dei casi l'istruzione `THREAD` assume che la comunicazione dell'ambiente sia locale, ovvero che il thread generato faccia sempre parte della locazione in cui viene eseguita la `THREAD` stessa. Questa assunzione non è valida quando si genera una nuova locazione. In quest'ultimo caso, infatti, l'istruzione `SPAWN-LOC` che segue la `THREAD` genera un thread nell'ambito di una locazione differente, e quindi con uno heap e una tabella delle porte diverse dalla locazione di origine. La `THREAD`, che opera sempre prima della `SPAWN-LOC`, ha quindi bisogno di un sistema per sapere quando l'ambiente deve essere creato con il procedimento di marshalling o quando invece il marshalling non è richiesto.

Osservando il codice prodotto dalla compilazione delle locazioni (ad esempio a pagina 51) si può notare che le istruzioni `THREAD` sono sempre precedute da un certo numero di `LOCATION`; per essere precisi, ad ogni `LOCATION` corrispondono una istruzione `THREAD` e una `SPAWN-LOC`. Il supporto runtime sfrutta questo fatto incrementando il campo `new_loc` nel DTD del thread che crea le locazioni ad ogni istruzione `LOCATION`. Al momento di eseguire la `THREAD`, il supporto controlla il campo `new_loc`: se questo è maggiore di zero allora è in corso la generazione di una nuova locazione e la creazione dell'ambiente viene fatta usando il procedimento di marshalling. Contemporaneamente, il campo `new_loc` viene decrementato.

C'è un altro aspetto che riguarda il passaggio dell'ambiente durante la creazione delle locazioni. Vedremo nei prossimi capitoli che il compilatore proibisce il passaggio esplicito delle primitive nei messaggi, ciò nondimeno le primitive, che sono sempre dichiarate globalmente, risultano comunque visibili all'interno delle locazioni e vengono passate nell'ambiente, cioè attraverso un messaggio. Il problema è che il meccanismo di marshalling non riesce a distinguere le normali porte dalle primitive: entrambe le entità hanno nella signature un generico tipo "porta", di conseguenza le primitive subiscono il marshalling e nella nuova locazione divengono alias, andando contro al sistema di binding dinamico delle primitive adottato per l'implementazione. In questo caso non si può fare altro che un "aggiustamento" *ad hoc* che provveda a riconvertire gli alias relativi a primitive in descrittori opportuni a completamento dell'istruzione `SPAWN-LOC`.

3.2.9 Migrazione

La migrazione degli agenti è forse l'aspetto più "singolare" del supporto runtime: quando una locazione si sposta, non solo il codice (che ne rappresenta la parte statica), ma anche tutte le risorse allocate nell'ambito della stessa locazione devono migrare. A questo punto è il caso di essere più precisi sul significato del termine "risorsa", in modo da avere un'idea più completa nel seguito.

La tabella delle risorse

Le risorse di una locazione comprendono: il segmento codice, lo heap, la tabella delle porte ed infine qualsiasi descrittore allocato nell'ambito della locazione, ovviamente non compreso nello heap o nella tabella delle porte.

Il vantaggio di allocare la memoria nello heap è subito evidente: lo heap è un'area contigua di memoria, quando lo heap si sposta, tutte le strutture allocate al suo interno si spostano di conseguenza. Tuttavia, nell'implementazione attuale, non sempre è conveniente allocare descrittori nello heap. La ragione principale consiste nel fatto che lo heap, potendo crescere senza limiti, è soggetto a operazioni di "riallocazione" per mezzo della funzione di libreria del C `realloc` e tale funzione non garantisce che, a seguito di una riallocazione, un'area di memoria continui a risiedere nella stessa posizione fisica. Questo particolare rende l'accesso alle strutture nello heap gravoso e soprattutto poco chiaro, in quanto ogni riferimento va specificato con puntatori virtuali e non si può beneficiare dell'aritmetica dei puntatori del C [KR88].

Per contro, l'allocazione delle strutture in modo non contiguo presenta un problema di "reperibilità": quando una locazione migra, occorre individuare *tutte* le strutture che la riguardano. Nell'implementazione questo problema è stato risolto eleggendo la tabella delle porte a una più generale *tabella delle risorse*. Ogni descrittore inerente ad una locazione è reperibile in modo indiretto attraverso un'entrata in questa tabella. Si spiega così il perché in molte strutture esaminate in precenza il tipo degli identificatori era spesso `VM_PORTID`: questi identificatori non sono altro che indici per la tabella delle risorse, proprio come lo sono le normali porte. Con questo accorgimento, basta che il supporto runtime scorra la tabella delle risorse di una locazione per conoscere tutte le strutture allocate al suo interno.

L'utilità della tabella delle risorse non si esaurisce qui: uno dei problemi più spinosi che riguardano il marshalling è la comunicazione di strutture dinamiche che contengono riferimenti ad altre strutture al loro interno. Ad esempio, quando si vuole comunicare una lista concatenata, tutti gli elementi della lista vanno inseriti nel messaggio e occorre un sistema per riferire i singoli elementi non più attraverso puntatori fisici che, tra macchine distinte, perdono significato, ma attraverso puntatori "logici". Occorre anche prestare attenzione quando una struttura è condivisa, intanto perché è conveniente inviarla in un'unica copia per ovvie ragioni di efficienza, e poi perché il suo stato di condivisione potrebbe essere fondamentale. Anche se nel linguaggio che abbiamo implementato non esiste la possibilità di creare siffatte strutture dinamiche (o almeno, non così direttamente), la migrazione di una locazione è a tutti gli effetti la comunicazione di un messaggio contenente strutture che si riferiscono l'una con l'altra. L'impiego della tabella delle risorse risolve automaticamente tutti i problemi elencati: introduce un sistema di "indirizzamento" logico per le strutture che è universale (cioè non dipendente dalla particolare macchina) e in più rende la comunicazione di strutture condivise estremamente semplice: l'unica copia della struttura inserita in un messaggio è quella reperibile direttamente attraverso la tabella delle risorse.

Concludiamo questa discussione con un esempio che dovrebbe rendere definitivamente chiara la situazione. In particolare, prendiamo come esempio la gestione delle *join-definition*. Abbiamo visto che tutte le porte definite fanno riferimento ad un unico descrittore di pattern condiviso.[3] Nella figura 3.3 sono mostrate due possibili implementazioni.

Nell'implementazione di sinistra le porte riferiscono direttamente il descrittore di pattern per mezzo di un puntatore fisico. Quando la locazione migra si deve evitare che questo descrittore venga copiato tre volte nel messaggio (ad esempio inserendo un apposito campo nel descrittore stesso che indichi se la struttura è già stata copiata una volta oppure no), ma non solo, serve un modo per "collegare" i descrittori di porta con il descrittore di pattern senza usare puntatori fisici. Invece, nell'implementazione a destra, che è anche quella che abbiamo adottato, tutti e tre i descrittori di porta riferiscono il descrittore di pattern per mezzo di un identificatore logico valido sia prima che dopo la migrazione. Inoltre, il riferimento fisico al descrittore di pattern è unico, di

[3]Escludiamo ovviamente le ottimizzazioni riguardanti pattern con un'unica porta.

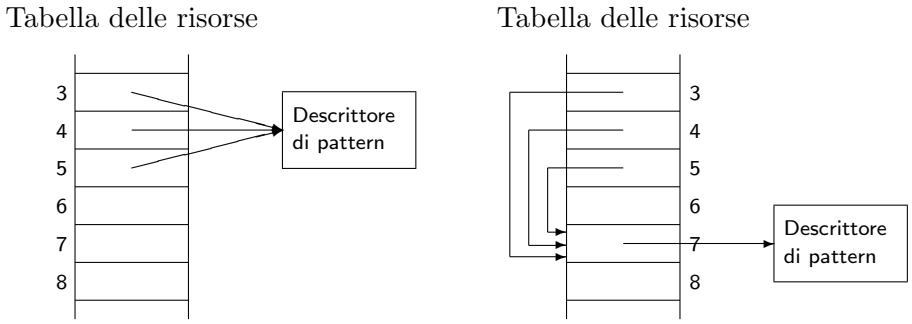

Figura 3.3: Possibili implementazioni della tabella delle risorse

conseguenza il supporto runtime, scorrendo la tabella, copierà il descrittore nel messaggio di migrazione una sola volta.

Il messaggio di migrazione

Il messaggio relativo ad una locazione che migra è fondamentalmente diverso dagli altri messaggi che abbiamo esaminato per ovvie ragioni. Senza entrare eccessivamente nei dettagli (che possono comunque essere studiati esaminando il codice sorgente), la caratteristica "ad alto livello" più rilevante è dovuta al fatto che, quando una locazione si muove, tutte le sotto-locazioni definite al suo interno si muovono con lei. Più in generale possiamo dire che la migrazione riguarda non una singola locazione, ma un sottoalbero di locazioni.

In pratica, il processo di "congelamento" (o serializzazione per usare la terminologia Java) della locazione che migra viene applicato ricorsivamente a tutte le locazioni figlie. L'accesso alle locazioni figlie è immediato, grazie ai puntatori all'interno del descrittore di locazione (vedi sezione 3.2.7), ma la struttura del sottoalbero che migra deve poter essere ricostruita a migrazione avvenuta, e ancora una volta i puntatori fisici non si prestano facilmente ad una operazione di questo tipo. Il messaggio di migrazione è così composto da tutte le locazioni serializzate, ognuna preceduta dall'identificatore univoco della relativa madre. La prima locazione del messaggio, quella che ha effettivamente dato inizio al processo di migrazione, ha come locazione madre la locazione destinazione, ovvero quella indicata come argomento nell'istruzione

`go`. Nella figura 3.4 è visibile una rappresentazione schematica del messaggio di migrazione.

Figura 3.4: Formato del messaggio di migrazione

3.3 Comunicazione a basso livello

Oltre alla realizzazione dell'interprete per la macchina virtuale, il runtime si occupa anche di fornire un supporto per la comunicazione a basso livello dei messaggi. Fino ad ora la comunicazione è sempre stata trattata dal punto di vista della macchina astratta, ma non abbiamo ancora visto come i messaggi vengano trasportati fisicamente da un runtime all'altro.

3.3.1 La tabella delle connessioni

Anche per l'implementazione della comunicazione abbiamo seguito le stesse linee che caratterizzano l'intero progetto, privilegiando la semplicità ad una realizzazione più efficiente, ma anche più complessa. Gli strumenti messi a disposizione dal sistema operativo sono i classici Berkeley socket ed i comuni protocolli UDP e TCP [Ste90].

Anche se il tipo di comunicazione richiesto dalla macchina astratta è chiaramente di natura *connection-less*, l'impiego di UDP è problematico perché la dimensione dei messaggi è limitata (e quindi potrebbe essere necessario un sistema di frammentazione e ricompattamento dei messaggi lunghi), ma soprattutto UDP non realizza un supporto di comunicazione affidabile (cioè con rilevamento degli errori e conseguente ritrasmissione dei pacchetti corrotti).

D'altro canto, il protocollo TCP non pone vincoli sulla quantità di informazione da trasportare e realizza una comunicazione affidabile, ma essendo di natura *connection-oriented* presenta l'handicap di richiedere fasi di apertura e chiusura delle connessioni che non possono certo considerarsi "leggere".

Scartata la possibilità di aprire e chiudere una connessione ogni volta che c'è un messaggio da trasmettere, la soluzione che abbiamo adottato prevede

allora l'utilizzo di connessioni "persistenti". In pratica una connessione viene aperta solo quando il runtime invia per la prima volta un messaggio ad un altro runtime. Una volta che la connessione è stata aperta, il runtime utilizza sempre la stessa per le successive comunicazioni. Questo sistema supplisce al comportamento non ideale di TCP, ma richiede una struttura ulteriore nel supporto runtime per individuare le connessioni aperte.

Tale struttura prende appunto il nome di *tabella delle connessioni* e realizza un insieme di associazioni tra indirizzi IP (e quindi tra supporti runtime) e connessioni TCP. Quando il runtime deve inviare un messaggio, per prima cosa controlla se esiste già una connessione con il runtime destinazione e in caso contrario provvede ad aprirla aggiornando la tabella delle connessioni.

Ogni elemento della tabella ha la struttura seguente:

```
typedef struct connection_table_entry {
        struct connection_table_entry *next;
        struct in_addr addr;
        MESSAGE_QUEUE  *queue;
        int            fd;
} CONNECTION_ENTRY;
```

dove `addr` è l'indirizzo IP dell'host connesso, `fd` è il descrittore di file relativo alla connessione ed infine `queue` è la coda dei messaggi in attesa di essere inviati lungo la connessione.

Per ogni connessione, e quindi per ogni elemento della tabella delle connessioni, ci sono due thread di sistema dedicati alla ricezione e alla spedizione dei messaggi (vedi la figura 3.1 a pagina 58). I thread di ingresso si occupano della ricezione dei messaggi e del loro inserimento nella coda dei messaggi in ingresso, che è unica. I thread di uscita estraggono uno per volta i messaggi dalle code di uscita relative e li inviano lungo la connessione.

Visto che TCP realizza delle connessioni bidirezionali, è sufficiente usare un'unica connessione sia per i messaggi in uscita che quelli in ingresso in un certo runtime, e di conseguenza nell'elemento della tabella delle connessioni visto in precedenza c'è un unico descrittore `fd`. Questo sistema, seppure comodo, ha un risvolto che va gestito in modo opportuno. In pratica una nuova connessione viene registrata nella tabella in due occasioni:

- Il supporto runtime apre attivamente una connessione per inviare uno o più messaggi.

- Il supporto runtime accetta passivamente una connessione dall'esterno, in particolare da un altro runtime che deve inviare messaggi.

Nella figura 3.5 viene mostrata una situazione in cui due runtime cercano di connettersi reciprocamente.

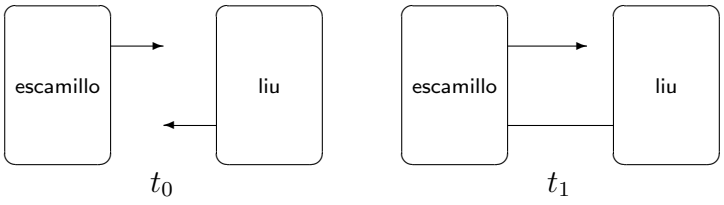

Figura 3.5: Conflitto tra connessioni contemporanee

Al tempo t_0 entrambi i runtime hanno avviato la procedura di connessione che però non è ancora stata completata. Al tempo t_1 una delle due connessioni è stabilita, ma a breve arriverà anche la seconda, al che i runtime dovranno decidere quale delle due mantenere e quale invece scartare. Nell'implementazione viene sempre privilegiata la connessione che "arriva prima".

3.3.2 La tabella di instradamento

Una struttura fondamentale, che peraltro abbiamo già incontrato, è la tabella di instradamento che realizza un insieme di associazioni tra locazioni e indirizzo del supporto runtime che le ospita.

Il supporto runtime usa questa tabella ogni volta che effettua una comunicazione inter-locazione, cercando l'entrata corrispondente alla locazione di destinazione. Se la locazione di destinazione è su un runtime diverso, allora il messaggio viene inserito nella coda opportuna dopo una ricerca nella tabella delle connessioni.

Visto che le locazioni sono entità estremamente dinamiche e possono spostarsi da un host all'altro ci si può aspettare che anche la tabella di instradamento sia altrettanto dinamica. In effetti il contenuto di questa tabella cambia al verificarsi di determinati eventi.

Il caso più immediato riguarda una locazione che migra. In questa situazione il supporto runtime modifica le entrate relative alla locazione e tutte le sue locazioni figlie (che si muovono anch'esse) per indicare il fatto che si trovano

su un runtime differente. Se in futuro il runtime dovesse ricevere un messaggio destinato ad una porta di una di queste locazioni, allora instraderebbe il messaggio verso l'host sul quale le locazioni sono migrate. Questo è quanto di meglio il runtime possa fare per portare i messaggi a destinazione, anche se non può sapere se nel frattempo le locazioni sono migrate altrove.

Però il runtime che esegue l'instradamento ha senza dubbio delle informazioni più aggiornate rispetto al runtime dal quale ha ricevuto il messaggio da instradare, per cui può informare a sua volta il runtime mittente con un messaggio di *routing* dandogli la presunta posizione delle locazioni. Questo meccanismo, se implementato da ogni runtime, può convergere in una situazione in cui le comunicazioni remote avvengono senza passare attraverso runtime intermedi. Abbiamo detto "può convergere" e non "converge" perché la velocità di convergenza dipende dalla frequenza delle migrazioni: se queste sono frequenti allora anche le tabelle di instradamento tenderanno a cambiare frequentemente.

Nella figura 3.6 sono rappresentate alcune istantanee di una computazione. Per ogni host è mostrata l'entrata nella tabella di instradamento relativa alla locazione l1, mentre le frecce indicano il flusso dei messaggi.

Al tempo t_0 la locazione l1, che risiede su escamillo, migra su pinkerton. Il runtime su liu è a conoscenza della posizione di l1, come si nota dalla sua tabella. Al tempo t_1 la locazione ha terminato lo spostamento e un messaggio m proveniente da liu raggiunge escamillo. Il runtime consulta la tabella di instradamento, nota che la locazione cercata è presente su pinkerton e quindi, al tempo t_2, instrada il messaggio verso la destinazione corretta, informando contemporaneamente liu della nuova posizione di l1. Infine, al tempo t_3, tutti i runtime raggiungono la convergenza delle tabelle. A questo punto, se l1 non effettuerà migrazioni ulteriori, tutte le comunicazioni remote avverranno attraverso un singolo passaggio (nella terminologia delle reti, un unico *hop*).

3.3.3 Messaggi ritardati

La possibilità di scambiare messaggi di instradamento tra runtime apre la strada a nuovi scenari potenzialmente pericolosi. Abbiamo visto nel capitolo 2 come il compilatore generi un codice particolare per evitare che le porte definite in una locazione vengano usate prima della loro "inizializzazione". In pratica, si è voluto evitare che il supporto runtime si trovasse in una situazione

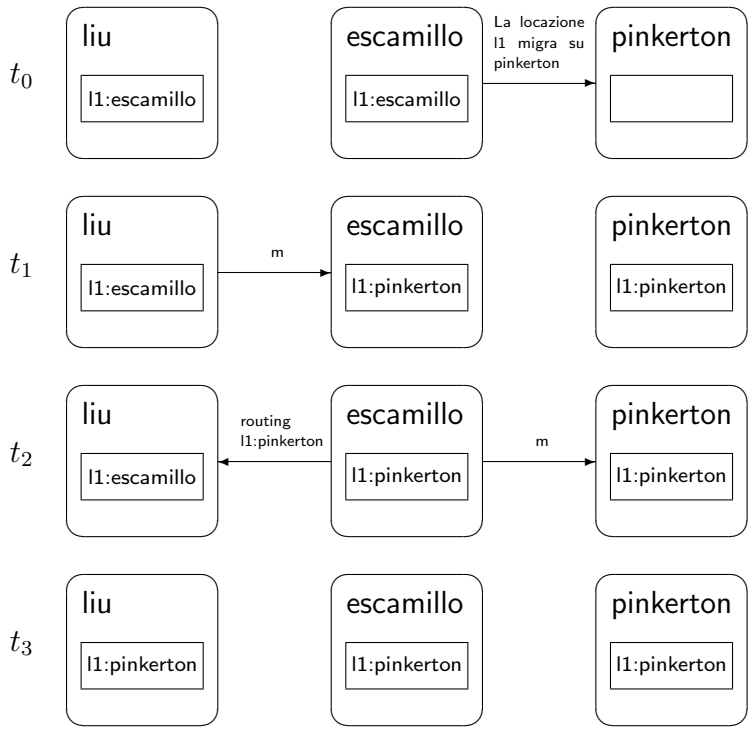

Figura 3.6: Instradamento dei messaggi e algoritmo di routing

in cui la porta di destinazione di un messaggio non aveva ancora un descrittore valido.

L'introduzione dei messaggi di instradamento causa un problema analogo, ma ad un livello più alto: cosa succede se un runtime riceve un messaggio per una locazione di cui non conosce la posizione, ovvero per la quale non esiste una entrata nella tabella di instradamento? È possibile una situazione simile?

Osserviamo innanzi tutto che evitando la generazione dei messaggi di instradamento il sistema complessivo continua a funzionare regolarmente e questo problema non si presenta: i messaggi per una determinata locazione seguono sempre il percorso che la locazione ha fatto nel corso degli spostamenti, in più, visto che il protocollo usato (TCP) mantiene l'ordine dei messaggi, e visto che tutte le code all'interno dei vari runtime usano una disciplina FIFO, i messaggi per una locazione "inseguono" sempre la locazione che migra senza possibilità di "sorpassarla". In conclusione, questo significa che se un runtime riceve un messaggio per una locazione, allora un'entrata relativa alla stessa

locazione è sicuramente presente nella tabella di instradamento.

Le cose cambiano se sono ammessi i messaggi di instradamento. In questo caso i normali messaggi possono seguire connessioni diverse da quelle attraversate dalle locazioni di riferimento, rendendo del tutto imprevedibile l'ordine dei messaggi in arrivo in un certo runtime. In queste ipotesi è possibile che un messaggio arrivi ed un runtime non sappia assolutamente cosa farne. Però c'è una considerazione fondamentale da fare: se un messaggio è arrivato su un certo host, significa che "qualcuno" ha saputo che la locazione di destinazione si sta spostando proprio su tale host. Il runtime deve solo "ritardare" il messaggio ricevuto finché non riceve la locazione interessata e a quel punto può effettuare la consegna.

Nell'implementazione esistono di fatto due code dei messaggi in ingresso: una è quella già descritta per i messaggi che hanno una destinazione nota, mentre la seconda, detta appunto *coda dei messaggi ritardati*, serve da contenitore per i messaggi con destinazione ignota, cioè destinati ad una locazione per la quale non esistono entrate nella tabella di instradamento. Per semplicità, e visto che comunque i messaggi ritardati non dovrebbero essere affatto frequenti, questa coda è unica e viene controllata al verificarsi di determinati eventi, come ad esempio l'arrivo di una locazione nel runtime.

3.4 Primitive e nameserver

3.4.1 Implementazione delle primitive

Le primitive messe a disposizione dal modulo runtime vengono implementate come normali funzioni C con zero o più argomenti (reperiti direttamente dallo stack del thread utente che le invoca) e che restituiscono un risultato (un segnale oppure, in genere, un messaggio con un unico elemento).

Le primitive obbediscono alle regole di binding dinamico, nel senso che una locazione fa sempre riferimento alle primitive del runtime sul quale si trova, e non a quelle sul runtime su cui è stata definita. Questo implica che gli eventuali effetti collaterali delle primitive, quali ad esempio la stampa di messaggi sul terminale, si manifestano di volta in volta sull'host in cui la primitiva viene invocata.

La primitive possono essere approssimativamente suddivise in due cate-

gorie: lente e veloci. In base alla particolare categoria il runtime esegue le primitive in modo differente. Infatti, essendo implementate come normali funzioni C, le primitive non obbediscono alle politiche di scheduling della macchina virtuale, e quindi non è previsto il prerilascio del processore durante l'esecuzione di una primitiva. Questo implica che durante l'esecuzione di una primitiva lenta, quale può essere la lettura di un carattere dal terminale, nessun altro thread nella macchina virtuale, seppure pronto, può avanzare. Questo comportamento, quantomeno fastidioso, non sarebbe stato possibile se ogni thread utente avesse avuto un thread di sistema a disposizione, ma abbandonata questa soluzione per le ragioni già presentate occorreva un sistema alternativo.

Così il supporto runtime discrimina le primitive veloci, cioè quelle che terminano senza attese eccessive, da quelle lente, cioè quelle che bloccano l'esecuzione per tempi prolungati (o anche indefiniti) ed esegue queste ultime nell'ambito di un nuovo thread di sistema separato, consentendo al thread che implementa la macchina virtuale di continuare ad eseguire i thread utente pronti.

Questa soluzione presenta un unico inconveniente: le primitive veloci, che vengono eseguite in modo sincrono, possono far affidamento sul fatto che, alla loro terminazione, il thread che deve ricevere il messaggio di risposta sia ancora presente nel runtime, perché nel frattempo nessun thread utente ha avuto l'opportunità di avanzare e di conseguenza causare una migrazione. Ma le primitive lente, che vengono eseguite in modo asincrono, ovviamente non possono fare la stessa assunzione. Nell'implementazione, il messaggio di risposta di una primitiva lenta viene inserito nella coda dei messaggi in ingresso al runtime, proprio come se fosse un messaggio proveniente da un altro host, lasciando al supporto runtime, e non alla primitiva, il compito di effettuarne la consegna, eventualmente instradandolo verso la locazione se quest'ultima è migrata.

Ogni primitiva, proprio come qualsiasi altro nome soggetto a binding dinamico, ha un'entrata corrispondente in una tabella del supporto runtime gestita da un'entità detta nameserver, che andiamo ora a descrivere.

3.4.2 Il nameserver

Compito del nameserver è quello di mantenere in una tabella un insieme di associazioni tra nomi simbolici e "valori". I nomi si dividono in due categorie

che sono quelle delle primitive, già descritte, e dei nomi registrati attraverso la primitiva `register`.

Ogni elemento di questa tabella ha la struttura seguente:

```
typedef struct nametable_entry {
        STRING    name;
        PRIMITIVE f;
        char      *message;
        unsigned  size;
        struct nametable_entry *next;
} NAMETABLE_ENTRY;
```

Il campo `name` è un puntatore fisico al nome simbolico, `f` è un puntatore alla funzione C che implementa la primitiva e vale `NULL` se il nome è stato registrato con `register`. I campi `message` e `size` caratterizzano il "valore" del nome sotto forma di messaggio (che ha subito il marshalling in previsione di una probabile comunicazione inter-locazione).

L'unica difficoltà che riguarda l'implementazione del nameserver è dovuta al comportamento bloccante della primitiva `lookup`. In altre parole, un processo si blocca se invoca la `lookup` su un nome che non è stato (ancora) registrato nella tabella del nameserver. La difficoltà deriva dal fatto che in questo tipo di comunicazione ad un estremo c'è un processo normale, mentre all'altro estremo c'è il runtime che gestisce il nameserver e non un altro processo.

In pratica, il supporto runtime tratta una `lookup` su un nome non presente nel nameserver alla pari di un messaggio per una destinazione sconosciuta, proprio come accade a volte per i messaggi destinati a locazioni "in ritardo". In questo modo le richieste al nameserver che non possono essere soddisfatte si accumulano nella coda dei messaggi ritardati. Naturalmente ora il runtime deve controllare questa coda non solo quando avviene una migrazione (in entrata), ma anche tutte le volte che viene invocata una `register`.

La soluzione adottata, sebbene migliorabile dal punto di vista dell'efficienza, risponde però a quei canoni di semplicità che ci eravamo imposti e in ogni caso la riteniamo soddisfacente in quanto i casi in cui interviene sono rari e comunque destinati a risolversi in breve tempo.

3.5 Invocazione del supporto runtime

L'invocazione del supporto runtime segue l'usuale sintassi dei comandi UNIX:

```
jcvm [ option ... ] file ...
```

I file specificati sono file oggetto risultanti dal processo di compilazione, mentre le opzioni riconosciute sono le seguenti:

`-[1,2,3]` Serve per specificare l'indirizzo di loopback. È utile per eseguire più supporti runtime sulla stessa macchina, assegnando indirizzi IP diversi a runtime diversi. A seconda del numero specificato, l'indirizzo di loopback è assunto pari a 127.0.0.$[1, 2, 3]$.

`-C` Abilita la *console*. La console permette di collegarsi ad un runtime persistente in esecuzione attraverso il comando `telnet` (specificando la porta 6667). La console permette di eseguire semplici interrogazioni per visualizzare la tabella di instradamento, lo stato del nameserver, l'albero delle locazioni, ecc.

`-H name` Specifica il nome (o eventualmente l'indirizzo) dell'host che ospita il runtime, nel caso il runtime stesso non riuscisse a determinarlo automaticamente.

`-S name` Specifica il nome (o eventualmente l'indirizzo) dell'host che ospita il nameserver.

`-c` Esegue il runtime in modo "client". In questa modalità il runtime non può accettare connessioni dall'esterno, anche se può aprirne a sua volta. È utile per poter eseguire più runtime sulla stessa macchina.

`-p` Esegue il runtime in modalità "persistente". Di norma un runtime termina quando non vi sono più processi in esecuzione, mentre con questa opzione lo si costringe a rimanere attivo fino alla terminazione forzata.

`-r` Abilita la comunicazione dei messaggi di instradamento, come descritto nella sezione 3.3.2.

`-s` Esegue il runtime attivando il nameserver.

`-v` Visualizza numerose informazioni durante l'esecuzione del supporto runtime. È utile in fase di debugging per osservare i messaggi scambiati.

`-z` Abilita la compressione di messaggi "lunghi" (di dimensione maggiore o uguale a 1024 byte) tramite `gzip`.

Capitolo 4

Descrizione formale del linguaggio

In questo capitolo descriveremo la sintassi concreta del linguaggio, le restrizioni sintattiche e semantiche, le regole di *scope* degli identificatori e le regole di controllo dei tipi.

La descrizione del linguaggio comincerà dalla struttura dei *token* per procedere poi in modo *top-down* alla presentazione dei costrutti.

4.1 Convenzioni lessicali

4.1.1 Identificatori

Un identificatore è una sequenza di lunghezza arbitraria di lettere, cifre e caratteri '_' che comincia con una lettera:

$$\begin{array}{lcl} \textit{letter} & ::= & \texttt{a..z} \mid \texttt{A..Z} \\ \textit{ident} & ::= & \textit{letter} \{ \textit{letter} \mid \texttt{0..9} \mid \texttt{_} \} \end{array}$$

Parole chiave

I seguenti identificatori sono parole chiave del linguaggio e non possono essere ridefiniti: `and`, `bool`, `char`, `def`, `do`, `else`, `external`, `if`, `in`, `init`, `int`, `join`, `let`, `location`, `mod`, `not`, `or`, `reply`, `spawn`, `string`, `then`, `to`, `void`, `where`, `with`.

Primitive

Le uniche primitive definite dal compilatore sono: `go`, `lookup`, `register`. Le altre primitive supportate dal modulo runtime possono essere dichiarate usando la clausola `external` (vedere sezione 4.2.4).

4.1.2 Costanti

Le costanti si suddividono in costanti intere, booleane, carattere e stringa.

Interi

I numeri interi sono normalmente specificati in base dieci. Alternativamente è possibile specificare costanti in base due, otto o sedici usando rispettivamente i prefissi `0b`, `0o` o `0x`.

int-const ::= { `0..9` }$^+$
| (`0b` | `0B`){ `0..1` }$^+$
| (`0o` | `0O`){ `0..7` }$^+$
| (`0x` | `0X`){ `0..9` | `A..F` | `a..f` }$^+$

Valori booleani

bool-const ::= `true` | `false`

Caratteri

Una costante carattere è un carattere regolare oppure una sequenza di *escape* tra singoli apici. Le sequenze di *escape* riconosciute sono riassunte nella tabella 4.1.

character ::= *regular-char*
| \ (\ | " | `n` | `r` | `t` | `b`)
| *ddd*

char-const ::= ' *character* '

Sequenza	Carattere
\\	*backslash* \
\"	doppi apici "
\n	nuova linea (LF)
\r	ritorno a capo (CR)
\t	tabulazione (TAB)
\b	*backspace* (BS)
ddd	carattere con codice ASCII *ddd*

Tabella 4.1: Sequenze di escape riconosciute dal compilatore

Stringhe

Una costante stringa è una sequenza di zero o più caratteri racchiusi tra doppi apici.

string-const ::= `"` { *character* } `"`

4.1.3 Operatori

Gli operatori si suddividono in aritmetici, booleani, relazionali, binari e unari. Tutti gli operatori hanno la semantica ovvia e precedenza e associatività usuali.

Operatori binari (o infissi)

infix-op ::= `+` | `-` | `*` | `/` | `mod` | `and` | `or`
rel-op ::= `<` | `>` | `=` | `<=` | `>=` | `<>`

Gli operatori `and` e `or` operano esclusivamente sui booleani, gli operatori relazionali sono *overloaded*, ovvero operano indifferentemente su interi, booleani e caratteri.

Operatori unari (o prefissi)

prefix-op ::= `-` | `not`

L'operatore `not` opera esclusivamente su booleani.

4.1.4 Commenti

I commenti sono racchiusi tra le sequenze `(*` e `*)`. Non sono ammessi commenti annidati.

4.2 Grammatica del linguaggio

4.2.1 Corpo principale

Il corpo principale di un programma (modulo) è una sequenza di processi, espressioni e dichiarazioni globali. Le espressioni e le dichiarazioni sono eseguite in modo sincrono, mentre i processi sono eseguiti in modo asincrono:

module ::= *top-expr-list*

top-expr-list ::=
::= *top-expr-list top-expr*

top-expr ::= *decl* `;;`
| `do` *expr* `;;`
| `spawn` *process* `;;`

Lo *scope* dei nomi di una dichiarazione globale comprende la dichiarazione stessa (vedere la sezione 4.2.4) ed il resto del programma. È ammessa la ridefinizione di nomi.

4.2.2 Processi

process ::= *ident*(*expr-list*)
| `reply` *expr-list*
| `reply` *expr-list* `to` *ident*
|
| *process* `|` *process*
| *expr* `;` *process*
| `if` *expr* `then` { *process* }
| `if` *expr* `then` { *process* } `else` { *process* }
| *decl* `in` *process*
| { *process* }

Messaggio asincrono

L'invio di un messaggio su un canale asincrono è un processo. Il messaggio è composto da una lista di espressioni, il tipo complessivo della lista di espressioni deve essere uguale al tipo dei messaggi accettati dal canale.

Continuazione

L'istruzione `reply` consente di inviare un messaggio di continuazione a una porta sincrona. Se ci sono ambiguità, ad esempio se il processo è associato ad un pattern con più porte sincrone, la porta può essere indicata esplicitamente dopo la parola chiave `to`.

Per motivi di efficienza esistono severe restrizioni sull'uso dell'istruzione `reply`. Il compilatore deve verificare che ogni porta sincrona definita in un pattern riceva prima o poi un messaggio di continuazione attraverso `reply`. Dato un pattern p ed il processo associato P, sia $\mathcal{S}(p)$ l'insieme delle porte sincrone che compaiono nel pattern e $\mathcal{R}(P)$ l'insieme delle porte a cui è stata ceduta la continuazione nel processo P, allora deve valere $\mathcal{S}(p) = \mathcal{R}(P)$.

La `reply` a una particolare porta sincrona può essere eseguita solo nel processo associato al pattern dove compare la porta oppure in un processo in composizione parallela con esso. In seguito verranno mostrate le relazioni che l'insieme $\mathcal{R}(P)$ deve soddisfare nei casi più interessanti.

Processo nullo

Il processo nullo (o vuoto) non esegue alcuna computazione, semplicemente termina. Può essere utile quando si uniscono in composizione parallela più espressioni, come ad esempio `print(1); | print(2);`.

Composizione parallela

La composizione parallela di due processi è a sua volta un processo. Notare tuttavia che all'atto pratico non esiste alcuna gerarchia di processi.

Se in un processo compare una `reply` ad una determinata porta, allora nell'altro processo non può comparire una `reply` alla stessa porta. Formalmente, detto $P ::= P_1 \mid P_2$, allora deve valere $\mathcal{R}(P_1) \cap \mathcal{R}(P_2) = \emptyset$, in questo caso $\mathcal{R}(P) = \mathcal{R}(P_1) \cup \mathcal{R}(P_2)$.

Composizione sequenziale

Una espressione ed un processo possono essere combinati, in quest'ordine, per formare un nuovo processo. L'espressione deve avere tipo `void`.

Esecuzione condizionale

L'esecuzione condizionale di uno o due processi è a sua volta un processo. La forma ridotta senza ramo `else` è equivalente alla forma completa con il ramo `else` che esegue un processo vuoto (sebbene il compilatore possa decidere di non generare codice per il ramo alternativo). I processi devono obbligatoriamente comparire tra parentesi graffe. L'espressione valutata deve avere tipo booleano.

Se in un ramo si esegue una `reply` ad una porta, allora anche nell'altro ramo deve comparire una `reply` alla stessa porta. Ciò implica che nella forma ridotta senza ramo `else` non possono esserci `reply` all'interno del processo. Formalmente, detto $P ::= \texttt{if}\ E\ \texttt{then}\ \{\ P_t\ \}$ deve essere $\mathcal{R}(P_t) = \emptyset$ e vale $\mathcal{R}(P) = \emptyset$, mentre in $P ::= \texttt{if}\ E\ \texttt{then}\ \{\ P_t\ \}\ \texttt{else}\ \{\ P_e\ \}$ deve essere $\mathcal{R}(P_t) = \mathcal{R}(P_e)$ e vale $\mathcal{R}(P) = \mathcal{R}(P_t) = \mathcal{R}(P_e)$.

Dichiarazioni locali

Lo scope dei nomi di una dichiarazione locale comprende il solo processo che segue la parola chiave `in`. Vedere la sezione 4.2.4 per i costrutti ammissibili in una dichiarazione locale.

Raggruppamento

Il raggruppamento tra parentesi graffe di uno o più processi può essere utile per modificare la normale precedenza degli operatori di composizione. L'operatore di composizione sequenziale ha precedenza maggiore rispetto all'operatore di composizione parallela, così si può usare la forma `E ; { P1 | P2 }` per indicare l'esecuzione di E seguita dall'esecuzione parallela di P1 e P2, diversamente da `E ; P1 | P2` che invece indica l'esecuzione parallela del processo E ; P1 con il processo P2.

4.2.3 Espressioni

expr ::= *int-const*
| *char-const*
| *string-const*
| *bool-const*
| *ident*
| *ident*(*expr-list*)
| `go`(*expr*)
| *expr infix-op expr*
| *prefix-op expr*
| *expr rel-op expr*
| *expr* `;` *expr*
| `spawn` { *process* }
| `if` *expr* `then` *expr*
| `if` *expr* `then` *expr* `else` *expr*
| *decl* `in` *expr*
| (*expr*)

expr-ne-list ::= *expr*
| *expr* `,` *expr-ne-list*

expr-list ::=
| *expr-ne-list*

Costanti e nomi

Una costante costituisce una espressione il cui tipo è il tipo della costante stessa. Un identificatore costituisce una espressione con tipo il tipo associato all'identificatore stesso. L'identificatore non può essere una parola chiave o una primitiva. In altre parole, nonostante assomiglino notevolmente alle normali porte, le primitive non possono essere inviate come valori nei messaggi.

Messaggio sincrono

L'invio di un messaggio ad una porta sincrona è una espressione con tipo il tipo dei messaggi restituiti dalla porta sincrona. Il tipo complessivo della lista

di espressioni che costituisce il messaggio inviato deve coincidere con il tipo dei messaggi accettati dalla porta sincrona.

Operatori binari e unari

Gli operatori sono già stati descritti nella sezione 4.1.3. Per gli operatori binari (anche quelli *overloaded*) i tipi delle due espressioni devono essere uguali.

Il tipo dell'espressione risultante dipende dal particolare operatore. Gli operatori aritmetici producono un risultato intero, gli operatori relazionali e booleani producono un risultato booleano.

Le espressioni booleane sono cortocircuitate, ovvero una espressione booleana può non venire completamente valutata nel momento in cui è noto il suo valore. Ad esempio, `true or 1 / 0 = 0` non genera un errore runtime.[1]

Composizione sequenziale

Due espressioni possono essere composte con `;` per generare una nuova espressione. In $E_1; E_2$ il tipo di E_1 deve essere `void`, mentre l'espressione risultante ha come tipo il tipo di E_2.

Generazione di processi

La generazione di un nuovo processo tramite l'istruzione `spawn` è una espressione con tipo `void`.

Esecuzione condizionale

Per le espressioni esiste l'analogo costrutto di esecuzione condizionale visto per i processi. In questo caso però non sono necessarie le parentesi graffe per racchiudere le espressioni.

Il tipo della prima espressione, quella usata per il test, deve essere booleano, mentre il tipo delle espressioni nei due rami deve coincidere. Se il costrutto è usato nella forma ridotta (senza ramo `else`), il tipo dell'espressione nel ramo `then` deve essere `void`.

[1] Semmai può generare un errore a tempo di compilazione se il compilatore precalcola le sotto-espressioni costanti.

Notare che per raggruppare più espressioni in un ramo del costrutto bisogna usare le parentesi tonde, ad esempio si scriverà:

```
if ... then ( print("1"); 1 ) else ( print("2"); 2 )
```

Dichiarazioni locali

Valgono le stesse considerazioni fatte per le dichiarazioni locali nei processi.

Raggruppamento

Il raggruppamento delle espressioni mediante parentesi tonde può essere usato per modificare le normali precedenze e associatività tra gli operatori e per racchiudere più espressioni in un ramo di esecuzione condizionale.

4.2.4 Dichiarazioni

Le dichiarazioni introducono nuovi nomi oppure ridefiniscono nomi già esistenti. Non è possibile dichiarare più volte un nome nello stesso ambiente.

local-decl ::= `let` *expr-decl*
| `join let` *expr-decl*
| `def` *process-decl*
| `location` *agent-decl*

decl ::= *local-decl*
| `external` *t-name-ne-list*

Come si nota dalla grammatica, la dichiarazione delle primitive può essere fatta solo a livello globale.

Liste di nomi

Una lista di nomi è una lista di identificatori separati da virgole.

name ::= *ident*
name-ne-list ::= *name* { `,` *name* }
name-list ::=
| *name-ne-list*

Una lista di *t*-nomi è una lista di nomi, ognuno con un tipo associato. È possibile associare a più nomi uno stesso tipo usando una notazione abbreviata.

t-name ::= *name-ne-list* `:` *type-ord*
t-name-ne-list ::= *t-name* { `,` *t-name* }
t-name-list ::=
| *t-name-ne-list*

Valutazione di espressioni

expr-decl ::= *t-name-ne-list* `=` *expr*
| *expr-decl* `,` *expr-decl*

Il risultato della valutazione dell'espressione è assegnato ai nomi che compaiono a sinistra del simbolo `=`. In generale il tipo dell'espressione è una tupla che deve coincidere con la tupla ottenuta concatenando i tipi dei nomi che compongono la lista.

In un costrutto `let` le espressioni da valutare possono contenere anche nomi definiti in precedenza nello stesso costrutto. Ciò non è vero per il costrutto `join let`, in cui non esiste un ordine prefissato di valutazione.

Join definition

Una *join definition* comincia con l'elenco delle porte in essa definite e segue, dopo la parola chiave `where`, con l'elenco dei pattern, ognuno dei quali ha un processo associato:

process-decl ::= *t-name-ne-list* `where` *join-pattern-ne-list*
join-pattern-ne-list ::= *join-pattern* `=` { *process* }
| *join-pattern-ne-list* `,` *join-pattern-ne-list*

Ogni pattern è composto da una o più porte tra quelle definite, ognuna con i nomi dei campi che costituiscono il messaggio:

join-pattern ::= *pattern*
| *pattern* `|` *pattern*
pattern ::= *ident* `(` *name-list* `)`

I pattern sono costituiti da una o più porte tra quelle definite.

Locazioni

agent-decl	::=	*agent-decl-header-list* `where` *agent-decl-list*
agent-decl-header-list	::=	*ident t-name-list*
	|	*agent-decl-header-list* `and` *agent-decl-header-list*
agent-decl-list	::=	*ident* [`@` *expr*] {
		[`with` *join-pattern-ne-list*] [`init` *process*] }
	|	*agent-decl-list* `,` *agent-decl-list*

La definizione di una lista di nuove locazioni comincia con un'intestazione in cui si elencano i nomi delle locazioni da creare e le porte definite inizialmente in ciascuna di esse. Questa lista è terminata dalla parola chiave `where`. Tutti i nomi delle locazioni dichiarate e i nomi delle porte dichiarate all'interno di ciascuna locazione sono visibili all'interno delle locazioni stesse, tuttavia nei pattern definiti inizialmente in una locazione non possono comparire porte definite inizialmente in una locazione diversa.

Segue la definizione delle locazioni: ogni locazione ha un nome, eventualmente seguito dal carattere `@` e da una locazione che indica la migrazione iniziale. La sintassi

```
name@l { init P }
```

è semplicemente una abbreviazione per

```
name { init go(l); P }
```

Il corpo della locazione è racchiuso tra parentesi graffe ed è costituito dai pattern definiti inizialmente e dal processo di inizializzazione. Sia i pattern che il processo di inizializzazione sono opzionali.

Primitive

Le primitive vengono dichiarate con la parola chiave `external` seguita da una lista di nomi, ognuno con il tipo associato. Dichiarare una primitiva con un tipo diverso da quello reale può portare a risultati imprevedibili. Le primitive fornite nella libreria standard sono riassunte di seguito, divise per categoria (l'elenco è stato tratto dal file `stdlib.ji`).

Input/Output sul terminale:

```
read: void -> int              (* legge un intero              *)
read_char: void -> char        (* legge un carattere            *)
print: int -> void             (* stampa un intero a video      *)
print_string: string -> void   (* stampa una stringa a video    *)
print_char: char -> void       (* stampa un carattere a video   *)
nl: void -> void               (* ritorno a capo                *)
print_location: location -> void (* stampa id di una loc.       *)
```

Indirizzamento di host e locazioni:

```
host_loc: string -> location   (* id. locazione di un host      *)
cur_loc: void -> location      (* id. locazione corrente        *)
hostname: void -> string       (* nome dell'host corrente       *)
```

Gestione delle stringhe:

```
strupdate: <string, int, char> -> string (* aggiorna           *)
strcpy: string -> string                 (* crea una copia     *)
strcat: <string, string> -> string       (* concatena          *)
strlen: string -> int                    (* lunghezza          *)
strcmp: <string, string> -> int          (* confronta          *)
```

Varie:

```
atoi: string -> int            (* converte stringa in intero    *)
rand: int -> int               (* numero casuale in range       *)
srand: void -> void            (* init algoritmo numeri casuali*)
abs: int -> int                (* valore assoluto               *)
sleep: int -> void             (* sleep di sistema              *)
usleep: int -> void            (* sleep al microsecondo         *)
```

4.2.5 Espressioni di tipo

type	::=	*type-ord*
	|	`void`
	|	`<` *type-tuple* `>`
type-ord	::=	`int`
	|	`bool`
	|	`char`
	|	`string`
	|	`location`
	|	(*type-ord*)
	|	*type* `->`
	|	*type* `->` *type*
type-tuple	::=	*type-ord*
	|	*type-ord* `,` *type-tuple*

Un tipo generico può essere un tipo ordinario oppure oppure una tupla. Per come è strutturata la grammatica sono vietati tipi con tuple annidate. Un tipo ordinario può essere un tipo base oppure il tipo di una porta (sincrona o asincrona). Una tupla di tipi è una lista di tipi separati da `,` e racchiusi tra parentesi angolate. La tupla vuota è indicata con `void`.

Capitolo 5

Il compilatore

In questo capitolo ci proponiamo di mostrare una panoramica della struttura interna del compilatore seguendo la sua naturale scomposizione in fasi, introducendo man mano la strutture usate per la compilazione di un programma.

Per non appesantire eccessivamente l'esposizione ometteremo descrizioni dettagliate delle strutture, o almeno dei campi meno rilevanti, rimandando al codice sorgente per ulteriori dettagli.

5.1 Architettura generale del compilatore

Il compilatore che abbiamo realizzato segue la classica divisione in *fasi* in cui ogni fase si occupa di un aspetto ben definito della traduzione [ASU85]. Nella figura 5.1 è rappresentata schematicamente la suddivisione in fasi del compilatore, con le frecce che indicano il "flusso di riconoscimento". Ogni fase interagisce con diverse strutture globali tra cui assume particolare importanza la tabella dei simboli.

La scomposizione in fasi è un mezzo comodo per capire e soprattutto per

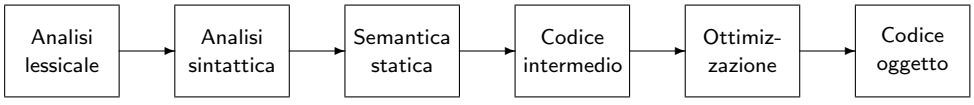

Figura 5.1: Fasi del compilatore

descrivere il funzionamento del compilatore, tuttavia una implementazione basata rigidamente su questa impostazione risulterebbe notevolmente inefficiente ed è stato preferito limitare il numero di "passate" a due soltanto. La prima passata, più consistente, racchiude tutte le fasi di analisi, la generazione del codice intermedio e parte della fase di ottimizzazione. Questo implica che nel codice sorgente del compilatore le funzioni che implementano le varie fasi sono intercalate tra loro. L'ultima passata serve unicamente per "compattare" il codice intermedio nel codice oggetto dopo aver operato alcune ottimizzazioni minori.

Visto che il linguaggio ha tra le sue caratteristiche principali quella di poter definire agenti mobili, ci siamo subito orientati verso un linguaggio oggetto di tipo *bytecode*. In altre parole, il risultato del processo di compilazione è un file contenente una codifica binaria delle istruzioni introdotte nel capitolo 2.

Il compilatore è organizzato in diversi moduli. In genere ogni modulo ha un file di implementazione (con estensione `.c`) ed un file di interfaccia (con estensione `.h`). In quest'ultimo sono elencate tutte le funzioni pubbliche del modulo. I file `const.h` e `types.h` contengono rispettivamente le definizioni di costanti o macro globali e le definizioni di tutte le strutture dati usate dal compilatore.

5.2 Analisi lessicale e sintattica

Le prime due fasi del processo di compilazione sono state agevolmente implementate grazie agli strumenti Lex e Yacc [ASU85] che consentono di generare automaticamente i riconoscitori per le grammatiche dei simboli e del linguaggio vero e proprio. Nei file `jlexan.l` e `jparser.y` sono presenti le specifiche Lex e Yacc delle grammatiche.

Già nella fase di analisi lessicale fa la sua apparizione la tabella dei simboli, in cui vengono memorizzate le parole chiave del linguaggio (vedi 4.1.1) e tutti gli identificatori definiti in un programma. La struttura della tabella dei simboli è quella classica con una tabella hash per la memorizzazione dei bucket ed una struttura ortogonale, detta *pila degli ambienti*, che contiene le informazioni sugli identificatori.

La struttura di un bucket è la seguente:

```
typedef struct st_bucket {
```

```
        STRING                  id;
        int                     token;
        struct binding_entry    *curr_binding;
        struct st_bucket        *next;
} STBUCKET;
```

dove `id` è un puntatore al nome simbolico dell'identificatore, mentre `token` è un codice che distingue le parole chiave dagli identificatori comuni. `next` è il puntatore al bucket di un altro identificatore con lo stesso codice hash. Esamineremo più in dettaglio la struttura `binding_entry` nella sezione 5.3.1.

In linea di principio il risultato dell'analisi sintattica di un programma è l'albero di sintassi astratta del programma stesso. In pratica ciò si traduce nell'esecuzione in un certo ordine delle azioni semantiche associate alle produzioni della grammatica del linguaggio. L'ordine esatto di esecuzione di queste azioni è quello imposto dall'analisi *bottom up* effettuata dal riconoscitore generato da Yacc. All'interno delle azioni semantiche possono essere trovate le chiamate alle funzioni che implementano le prime fasi della compilazione. In genere l'ordine di chiamata di queste funzioni rispecchia l'ordinamento delle fasi del compilatore.

5.3 Semantica statica

Scopo principale di questa fase è quello di effettuare il maggior numero di controlli per assicurarsi che il programma sia "semanticamente" corretto. Mentre la fase di analisi sintattica effettua implicitamente un controllo sulla correttezza della sintassi, la fase di analisi semantica verifica che le frasi (sintatticamente corrette) riconosciute dal parser abbiano anche una valenza semantica.

In questa fase ricadono principalmente tre tipi di controlli:

1. controllo dello scope;
2. controllo dei tipi;
3. controlli specifici del linguaggio, in particolare sull'utilizzo dell'istruzione `reply`.

5.3.1 Controllo dello scope

Il controllo dello scope deve verificare che ogni identificatore usato in un programma sia stato dichiarato in qualche punto precedente nel programma stesso ed obbedisca alle regole di visibilità delle dichiarazioni. In particolare, gli identificatori dichiarati globalmente sono visibili ovunque (salvo il caso in cui vengono nascosti da una successiva dichiarazione dello stesso nome), mentre gli identificatori dichiarati localmente sono visibili solo nella parte di codice che segue la parola chiave `in`.

Liste di nomi

Per convenienza, il compilatore non dichiara mai gli identificatori nel punto esatto in cui essi compaiono nel programma. Piuttosto raccoglie gli identificatori ed il loro tipo in una lista, detta lista di t-nomi, e li dichiara tutti in una volta al momento opportuno.

Le liste di t-nomi sono gestite dal modulo `tnamelist.c`. Ogni elemento di una lista ha la seguente struttura:

```
typedef struct t_name_list {
        STBUCKET                *st_bucket;
        TYPE                    *type;
        struct t_name_list      *next;
} T_NAME_LIST;
```

Il campo `st_bucket` individua il bucket nella tabella dei simboli del nome e `type` è un puntatore al suo tipo. Notare che il bucket di un identificatore è sempre presente nella tabella dei simboli, anche se l'identificatore non è ancora stato dichiarato, ma in quest'ultimo caso il campo `curr_binding` nel bucket vale `NULL`.

La pila degli ambienti

Per mantenere le informazioni corrette sulla visibilità degli identificatori, il compilatore gestisce una *pila degli ambienti*. Ad ogni dichiarazione viene aggiunto un nuovo elemento sulla pila che collega tutti gli identificatori nello stesso ambiente, mentre quando si esce dal campo di visibilità della dichiarazione (in caso di dichiarazione locale), l'ambiente in cima alla pila e tutti gli identificatori che contiene vengono rimossi.

Nella figura 5.2 è rappresentata una possibile situazione della tabella dei simboli in cui l'identificatore `x` è stato dichiarato due volte, prima con tipo `string` e successivamente con tipo `int`.

Ogni *binding entry* è una struttura contenente le informazioni di un nome. Da notare che due identificatori con lo stesso nome condividono lo stesso bucket, ma non condividono la stessa binding entry. Una binding entry ha la struttura seguente:

```
typedef struct binding_entry {
        int                     nesting_depth;
        STBUCKET                *st_bucket;
        struct binding_entry    *prev_id_binding,
                                *next;
        CLASS                   class;
        TYPE                    *type;
        int                     location_id;
        int                     thread_id;
        int                     offset;
        BOOL                    used;
        int                     port_id;
} BINDING_ENTRY;
```

Il tipo `CLASS` è definito come:

```
typedef enum { NO_CLASS, KEYWORD, NAME, DEF_PORT,
               EXT_PORT } CLASS;
```

Il campo `type` è un puntatore al tipo del nome, i campi `location_id` e `thread_id` contengono, rispettivamente, gli identificatori della locazione e del thread in cui il nome è stato definito e servono per distinguere i riferimenti a nomi locali dai riferimenti a nomi esterni.

Il campo `offset` indica l'offset del nome nell'area dati del thread in cui è definito, mentre `used` vale `TRUE` se l'identificatore, oltre ad essere stato dichiarato, è stato effettivamente utilizzato. Tipicamente questo campo viene impiegato per verificare che tutte le porte dichiarate in una *join-definition* compaiano in almeno un pattern della stessa definizione. Il campo `port_id` rappresenta invece l'indice di porta usato nell'istruzione `PATTERN` e sarà esaminato più avanti.

5.3.2 Controllo dei tipi

Il controllo dei tipi verifica che gli identificatori vengano usati in maniera consistente con il tipo ad essi associato. Ad esempio, se un identificatore è stato dichiarato con tipo `bool`, esso non potrà essere usato come operando per un'operazione aritmetica. Per verificare se due tipi sono uguali il compilatore ne controlla l'equivalenza *strutturale*.

Rappresentazione interna dei tipi

La rappresentazione dei tipi si avvale di strutture dinamiche ad albero. Ogni nodo dell'albero ha la seguente struttura:

```
typedef struct t_type {
        int             counter;
        TYPE_CONSTR     constr;
        struct t_type   *left,
                        *right;
} TYPE;
```

Il campo `counter` è usato per l'algoritmo di garbage collection e serve per deallocare il nodo solo quando non è più in uso. Il campo `constr` indica il *costruttore* di tipo e può assumere uno dei seguenti valori:

```
typedef enum {  T_ERR, T_INT, T_BOOL, T_CHAR, T_LOCATION,
                T_STRING, T_ASYNC_PORT, T_SYNC_PORT,
                T_TUPLE } TYPE_CONSTR;
```

Il costruttore `T_ERR` è usato per scopi interni al compilatore e rappresenta un tipo particolare equivalente a qualsiasi altro tipo. Quando il compilatore incontra un errore di tipo non sospende la compilazione, ma associa all'espressione errata il tipo `T_ERR`. In questo modo la compilazione può procedere senza generare una cascata di errori di tipo, ma con la possibilità di scoprirne altri.

In base al costruttore particolare possono assumere significato nessuno, uno soltanto o entrambi i campi `left` e `right`. Per i tipi base (`int`, `bool`, `char`, `location`, `string`) questi campi non sono utilizzati, per il tipo "porta asincrona" ha significato il solo campo `left`, per i tipi "porta sincrona" e "tupla" hanno significato entrambi.

Polimorfismo limitato per le primitive di accesso al nameserver

Una delle caratteristiche del tipo `T_ERR` è quella di essere equivalente ad ogni altro tipo. Questo fatto viene sfruttato dal compilatore per assegnare un tipo alle primitive `lookup` e `register` che sono polimorfe. Infatti la `register` accetta come argomenti una stringa ed valore di tipo qualsiasi, mentre la `lookup` accetta una stringa e restituisce un valore di tipo qualsiasi.

Siccome il linguaggio non permette di definire porte con tipi polimorfi, le primitive `register` e `lookup` vengono dichiarate implicitamente dal compilatore. In teoria le dichiarazioni per queste primitive sarebbero:

```
external register: <string, any> -> void,
         lookup: string -> any
;;
```

dove il tipo `any` viene realizzato internamente con `T_ERR`. Ecco perché queste primitive vengono dichiarate implicitamente e non sono elencate nel file `stdlib.ji`.

5.3.3 Controllo delle continuazioni

Il compilatore che abbiamo implementato effettua dei controlli per verificare che ogni porta sincrona che compare in un pattern riceva prima o poi il messaggio di continuazione nel processo associato.

Questo tipo di controllo è estremamente difficile da effettuare in generale a tempo di compilazione, per cui sono state imposte alcune restrizioni sull'uso dell'istruzione `reply`. In pratica il compilatore tiene traccia di tutte le porte a cui è stata inviata la continuazione per mezzo di una lista che viene sintetizzata durante il processo di riduzione. La lista è detta "lista di s-nomi" in quanto raccoglie unicamente un insieme di puntatori a bucket di identificatori (si veda il file `snamelist.c` per le funzioni di gestione di queste liste). La struttura di un elemento di tali liste è la seguente:

```
typedef struct s_name_list {
        STBUCKET                *st_bucket;
        struct s_name_list      *next;
} S_NAME_LIST;
```

In base alla particolare produzione della grammatica il compilatore effettua i dovuti controlli sulle liste e sintetizza la lista per il simbolo a sinistra della produzione (una descrizione formale di queste operazioni è fornita nella sezione 4.2.2).

Un'ulteriore difficoltà che concerne l'istruzione `reply` deriva dal fatto che essa può essere usata anche in forma "implicita", ovvero senza che sia specificata la porta a cui si invia il messaggio. Ciò può essere fatto senza ambiguità se nel pattern compare una sola porta sincrona, ma come fa il compilatore a sapere qual è la porta a cui inviare il messaggio se nella `reply` viene omessa? Questa informazione sarebbe agevolmente disponibile se Yacc permettesse di usare attributi ereditati. Osserviamo la produzione che associa un processo ad un pattern:

$$\textit{pattern-declaration} \;=\; \{\; \textit{process} \;\}$$

Il compilatore potrebbe estrarre dal pattern la lista di tutte le porte sincrone che compaiono in esso e passare questa lista come attributo ereditato in *process*, ma usando Yacc questa operazione non è evidentemente possibile. La soluzione di questo ed altri problemi analoghi richiede la presenza di un'altra struttura fondamentale che ci accingiamo a descrivere.

5.4 Generazione del codice intermedio

5.4.1 Caratteristiche del codice intermedio

Prima di produrre il codice oggetto, il compilatore introduce una forma di codice intermedio agevole da manipolare. In effetti il codice intermedio è una rappresentazione "a lista" del codice oggetto in cui le istruzioni, anziché comparire sequenzialmente in memoria, sono collegate tra loro per mezzo di puntatori. Questo accorgimento rende estremamente semplice appendere porzioni di codice in testa o in coda.

Ogni elemento di queste liste di istruzioni ha la struttura seguente:

```
typedef struct instruction {
        OPCODE          opcode;
        union {
                ...
        } u;
```

```
        struct instruction      *next;
} INSTRUCTION;
```

L'`opcode` è effettivamente il codice operativo dell'istruzione, esattamente del tipo riconosciuto dal supporto runtime, mentre la union `u` contiene numerose strutture che dipendono dalla particolare istruzione e che contengono gli argomenti delle istruzioni più altre informazioni ausiliarie del compilatore.

Oltre ai normali codici operativi, il campo `opcode` può assumere anche il valore `I_LABEL`, che *non* è valido per il supporto runtime, ma che nel compilatore sta ad indicare una etichetta. Infatti, nel codice intermedio, la destinazione dei salti viene specificata simbolicamente attraverso il nome di un'etichetta e non attraverso un offset. Sarà compito dell'ultima passata convertire questi riferimenti simbolici in offset comprensibili per la macchina virtuale.

Ogni lista di istruzioni ha un descrittore in cui si specificano i puntatori al primo ed all'ultimo elemento:

```
typedef struct instruction_list {
        INSTRUCTION     *first;
        INSTRUCTION     *last;
} INSTR_LIST;
```

5.4.2 La pila dei thread

In varie situazioni il compilatore ha necessità di accedere a informazioni che sono attinenti al thread nel quale sta generando istruzioni. Ad esempio, i nomi definiti in un thread occupano una posizione nell'area dati del thread stesso e il compilatore deve "allocare" quest'area man mano che i nomi vengono definiti. Oppure, come abbiamo visto prima, mentre si compila il codice per un processo può essere necessario conoscere la lista di porte a cui è lecito inviare una continuazione.

Dato che Yacc non consente di usare attributi ereditati, si usa una struttura globale, detta *pila dei thread*. Il compilatore inserisce e rimuove elementi sulla pila in corrispondenza dell'inizio e della fine della compilazione in un thread.

Ogni elemento della pila dei thread ha la seguente struttura:

```
typedef struct thread_stack {
        int                     nesting_depth;
```

```
        struct thread_stack     *prev;
        T_NAME_LIST             *e_cl;
        O_NAME_LIST             *fv_list;
        LOC_LIST                *loc_list;
        BOOL                    inherited_cl;
        int                     local_size;
        int                     env_size;
} THREAD_STACK;
```

Il campo `nesting_depth` rappresenta la profondità della pila e contemporaneamente l'identificatore del thread dal punto di vista del compilatore. Il campo `e_cl` (*expected continuation list*) è la lista delle porte a cui è lecito inviare la continuazione nell'ambito del thread, `local_size` è la dimensione dell'area dati del thread che viene aggiornata ad ogni nuova definizione. Altri campi di questa struttura saranno descritti in seguito.

Riferimenti a nomi esterni

Abbiamo visto che nella binding entry per un nome è presente un campo che contiene l'identificatore del thread nel quale il nome è definito. Confrontando questo identificatore con l'identificatore del thread in cima alla pila (cioè quello in cui il compilatore sta generando le istruzioni) si è in grado di distinguere i nomi locali dai nomi esterni. Abbiamo anche visto che i nomi esterni vengono passati esplicitamente al momento della creazione di un thread e vengono registrati in una zona particolare dell'area dati detta ambiente la quale, per motivi di efficienza, è comodo che sia allocata in modo *contiguo*.

Il compilatore assegna un offset ai nomi definiti all'interno di un thread "allocando" lo spazio nell'area dati, ovvero incrementanto il contenuto del campo `local_size` nel descrittore in cima alla pila dei thread. Se l'ambiente deve essere contiguo non si può utilizzare lo stesso sistema anche per i nomi esterni, perché questi vengono incontrati man mano durante la riduzione del processo. D'altro canto, ritardare la generazione di tutti gli offset fino alla riduzione completa del processo costituirebbe una notevole complicazione.

Allora, mentre i nomi definiti all'interno di un thread vengono allocati come descritto, i nomi esterni vengono allocati a offset negativi per mezzo del campo `env_size`. In pratica ogni nome occupa una parola nell'area dati (su una macchina a 32 bit una parola equivale a 4 byte), per cui il primo nome

locale definito in un thread avrà offset 0, il secondo offset 4, il terzo offset 8 e così via. Il primo nome esterno, invece, avrà offset -4, il secondo nome esterno offset -8 e via dicendo.

In questo modo, al termine della riduzione del processo, è sufficiente sommare la dimensione dell'ambiente (ancora una volta nel campo `env_size`) a tutti gli offset presenti nel codice per avere tutti i riferimenti che cominciano da 0.

I nomi esterni introducono però altre due complicazioni. In primo luogo, visto che occorre creare l'ambiente da passare a un thread, il compilatore deve conoscere tutti i nomi esterni riferiti in un processo. Alla stregua di quanto fatto per le porte sincrone e l'istruzione `reply`, si potrebbe pensare di mantenere un'attributo sintetizzato che contenga la lista di tutti i nomi esterni riferiti. Questa idea dà origine alla seconda complicazione: anche se in un processo si fa riferimento allo stesso nome esterno più volte, l'ideale sarebbe di passarlo una volta sola nell'ambiente. Però, usando solo attributi sintetizzati, non ci sarebbe modo di sapere se un nome esterno è già stato incontrato in precedenza (e quindi se ha già un'offset assegnato nell'ambiente del thread) oppure no.

Allora si potrebbe pensare di allocare una binding entry per un nome esterno nella tabella dei simboli quando il nome esterno è riferito per la prima volta. In questo modo, alle occorrenze successive, il compilatore userebbe sempre l'offset memorizzato nella binding entry. Tuttavia la pila degli ambienti e la pila dei thread non si evolvono in sincronia, di fatto potrebbe essere che un nome è esterno anche se lo si trova nell'ambiente corrente. Questa situazione rende impraticabile l'utilizzo della tabella dei simboli per i riferimenti esterni.

In conclusione, i nomi esterni vengono raccolti in una lista associata al thread corrente, e che quindi non viene sintetizzata con il normale meccanismo degli attributi di Yacc. Ogni volta che il compilatore incontra un riferimento ad un nome esterno controlla prima se il nome è già presente nella lista, cioè se ha già un offset nell'ambiente. Se il nome non è presente, allora viene allocato lo spazio nell'ambiente ed il nome viene inserito nella lista.

Esaminando l'elemento della pila dei thread si nota che la lista in questione (`fv_list`) ha tipo `O_NAME_LIST` (dipendente dal fatto che ogni elemento compare con un offset associato), la cui definizione è la seguente:

```
typedef struct o_name_list {
```

```
        BINDING_ENTRY           *be;
        int                     offset;
        struct o_name_list      *next;
} O_NAME_LIST;
```

Si noti infine che nella lista i nomi sono riferiti direttamente attraverso la loro binding entry e non attraverso il bucket.

Composizione parallela

Il costrutto di composizione parallela tra processi rappresenta il candidato ideale per vedere in che modo il compilatore "decide" la generazione di un nuovo thread. La produzione in questione è $P ::= P_1 \mid P_2$.

Per quanto abbiamo detto nel corso del capitolo 2, non è necessaria la presenza esplicita di un thread che generi sia P_1 che P_2: è sufficiente che P_1 si faccia carico della generazione di P_2 o viceversa. Avevamo anche detto che non c'era alcuna ragione per preferire la generazione di P_1 o quella di P_2, ma le cose vanno diversamente dal punto di vista del compilatore.

Il parser utilizzato è quello generato da Yacc che, com'è noto, è di tipo bottom up. In tale parser è possibile inserire delle azioni semantiche *tra* i simboli che compongono una produzione, azioni che vengono convertite internamente da Yacc tramite la tecnica dei *marker*. Tuttavia l'introduzione dei marker (che non sono altro che produzioni del tipo $M ::= \epsilon$) può provocare la generazione di conflitti. È questo il caso, ad esempio, se si inserisce un'azione semantica prima del processo P_1. In conclusione, se non si vuole stravolgere la grammatica, l'azione semantica più a sinistra che possiamo inserire nella produzione per la composizione parallela è posizionata *dopo* P_1. Ciò significa che non si ha la possibilità di inserire alcun elemento aggiuntivo sulla pila dei thread prima della riduzione di P_1. Questo inserimento possiamo farlo solo subito prima della riduzione di P_2. Ecco allora che tra le due alternative, il compilatore è obbligato a seguire quella che genera P_2 in un thread distinto.

È interessante osservare come diventi cruciale l'associatività che diamo all'operatore di composizione parallela per avere un codice efficiente. Nella figura 5.3 vengono mostrati i due tipi di generazione a seconda dell'associatività. Nel primo caso (associatività a destra) l'eventuale ambiente per P_3 dovrebbe passare attraverso P_2, mentre nel secondo caso è direttamente passato da P_1. Ne consegue che, nel primo caso, gli elementi nell'ambiente per P_3 che non fanno

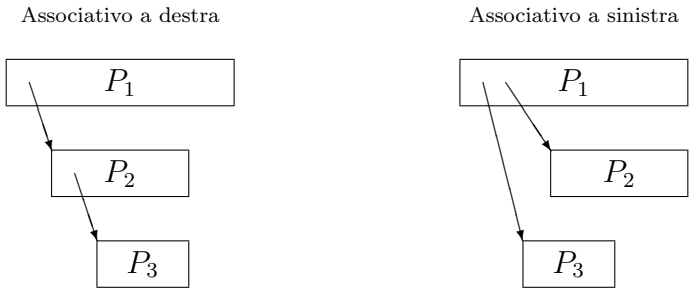

Figura 5.3: Generazione dei thread con diversa associatività dell'operatore |

parte anche dell'ambiente per P_2 vengono comunicati due volte anziché una soltanto.

La composizione parallela influenza anche il sistema di generazione delle `reply`. Come abbiamo detto, le `reply` obbediscono a diverse restrizioni, ma possono comparire nei processi in composizione parallela. Tuttavia l'uso della `reply` è legato alla presenza nel descrittore in cima alla pila dei thread della lista delle porte a cui è lecito inviare la continuazione, e quando due o più processi vengono legati attraverso la composizione parallela nuovi descrittori vengono inseriti sulla pila.

Allora, in questo caso particolare, il compilatore fa in modo che il nuovo descrittore che viene spinto in cima alla pila "erediti" la lista delle continuazioni dal descrittore sottostante. Questa operazione è eseguita semplicemente copiando il puntatore `e_cl` nel descrittore.

Visto che la copia interessa il solo puntatore, e non la lista vera e propria che di fatto è condivisa, occorre prestare attenzione quando un descrittore di thread viene rimosso dalla pila dei thread. Questa operazione di solito causa il rilascio di tutte le strutture presenti nel descrittore. Per la lista delle continuazioni il rilascio è evidentemente possibile solo se la lista stessa non è stata ereditata, perché in caso contrario essa sarà in utilizzo anche da parte di altri descrittori sottostanti.

Per evitare una deallocazione pericolosa il compilatore controlla il campo `inherited_cl`, sempre nel descrittore, che indica quando la lista è stata ereditata oppure è effettivamente la prima. Solo in quest'ultimo caso la lista viene deallocata dalla memoria.

Compilazione delle join definition

Nel capitolo 2 abbiamo visto che l'istruzione `PATTERN` identifica le porte attraverso un indice sequenziale che comincia da 0 e, cosa più importante, questi indici vengono calcolati dal compilatore.

In presenza di una *join-definition* il compilatore deve prima di tutto assegnare ad ogni porta un indice, poi deve raccogliere in una struttura le informazioni sui pattern che ha incontrato e finalmente è in grado di generare il codice per l'istruzione `PATTERN`.

L'assegnazione degli indici alle porte avviene al momento della dichiarazione delle porte stesse. Il campo `port_id` nella binding entry serve proprio a memorizzare questo indice.

Le informazioni sui pattern vengono raccole in una lista di pattern i cui elementi hanno la struttura seguente:

```
typedef struct pattern_list {
        int n;
        int count;
        int *port_id;
        struct pattern_list *next;
} PATTERN_LIST;
```

C'è un elemento della lista per ogni pattern. Il campo `n` indica quante sono le porte che compongono il pattern, gli indici delle porte componenti sono memorizzati nell'array `port_id` mentre `count` rappresenta il numero di pattern successivi nella lista (cioè la lunghezza della coda `next`) più uno. In pratica questo campo consente di conoscere il numero totale di pattern definiti senza dover scorrere l'intera lista.

5.4.3 Compilazione delle locazioni

Tutte le costanti stringa che compaiono in un programma sorgente vengono allocate nel pool delle costanti della locazione di appartenenza, quindi, per ogni locazione definita, occorre identificare lo stato di allocazione del pool, un po' come accade per l'allocazione dell'area dati di un thread. In effetti, anche per le locazioni esiste una struttura globale, detta *pila delle locazioni*, usata per memorizzare le informazioni rilevanti di ogni singola locazione.

Inoltre, è facile notare come per i vari pool di costanti esista una relazione di gerarchia analoga alla relazione che intercorre tra le locazioni, infatti il pool di una sublocazione è a sua volta un sottoinsieme del pool della locazione madre. Ora, visto che in un programma sorgente tutte le locazioni fanno parte della stessa gerarchia (la cui locazione radice è quella creata implicitamente dal supporto runtime), si ha che esiste un unico pool di costanti all'interno del quale sono localizzati, a offset diversi, i pool di tutte le sublocazioni definite.

Quando si comincia a generare il codice per una nuova locazione, il compilatore spinge sulla pila delle locazioni un nuovo descrittore della forma:

```
typedef struct location_stack {
        int     location_id;
        int     data_size;
        struct location_stack *prev;
} LOCATION_STACK;
```

Ogni volta che una nuova stringa viene allocata nel pool, il compilatore aggiorna il campo `data_size`. Per individuare l'offset del pool della locazione corrente relativo al pool della locazione madre è sufficiente accedere al medesimo campo, ma nel descrittore sottostante a quello in cima alla pila. Quando un descrittore di locazione viene rimosso dalla pila, la dimensione del pool viene sommata alla dimensione del pool nel descrittore sottostante.

Nella figura 5.4 è mostrata un'istantanea della compilazione in cui ci sono tre locazioni con un pool di dimensione 4, 16 e 12 rispettivamente. Notare come la dimensione del pool in ogni locazione individui l'offset del pool della locazione che sta sopra.

La pila delle locazioni è una struttura comoda durante la riduzione delle singole locazioni, ma la generazione del codice per il costrutto che definisce nuove locazioni è abbastanza complesso da richiedere un aiuto ulteriore. Infatti le istruzioni `LOCATION` devono comparire *prima* del codice che genera i thread di inizializzazione delle locazioni, e per compilare l'istruzione `LOCATION` occorre conoscere offset e dimensione del pool relativo. Queste informazioni erano disponibili nei descrittori di locazione che però, al momento della generazione delle `LOCATION`, sono già stati rimossi dalla pila delle locazioni. Ancora una volta si utilizza una lista che raccoglie gli elementi fondamentali di ogni locazione generata:

```
typedef struct loc_list {
        STBUCKET        *name;
        T_NAME_LIST     *dv_list;
        struct loc_list *next;
        unsigned        pool_offset;
        unsigned        pool_size;
} LOC_LIST;
```

Il campo `name` è un puntatore al bucket relativo al nome della locazione, `dv_list` è la lista delle porte definite inizialmente nella locazione, `pool_offset` e `pool_size` individuano rispettivamente l'offset e la dimensione del pool delle costanti all'interno del pool della locazione madre.

Una volta sintetizzata questa lista con il consueto meccanismo degli attributi del parser è possibile generare il codice per l'intero costrutto.

5.5 Ottimizzazioni

Il compilatore effettua alcune ottimizzazioni sul codice intermedio e sul codice oggetto al fine di migliorare le prestazioni a tempo di esecuzione ed evitare situazioni patologiche. Di seguito saranno descritte tre aree di ottimizzazione accompagnate da esempi di codice ottimizzato e non.

5.5.1 Thread non bloccanti

Un thread si definisce "non bloccante" se nel suo codice intermedio non compaiono istruzioni `RECV` o `RECVFROM`. Visto che queste sono le uniche istruzioni che causano il blocco di un thread, un thread non bloccante termina sempre. Dal punto di vista del linguaggio un processo si definisce "non bloccante" se nel suo codice non compaiono chiamate a porte sincrone. Non sempre c'è corrispondenza tra processi e thread non bloccanti, ad esempio la generazione di una locazione può far parte di un processo non bloccante, ciò nonostante il codice contiene sicuramente delle istruzioni `RECV` per la sincronizzazione. L'aspetto importante non è tanto la presenza o l'assenza di istruzioni bloccanti come la `RECV`, quanto il fatto che queste istruzioni, in certi casi, non bloccano indefinitamente i processi e non v'è spreco del processore virtuale in quanto al bloccarsi di un processo ce n'è subito uno pronto per andare in esecuzione.

Detto questo, è facile constatare come la composizione parallela di due o più processi non bloccanti sia causa di un inutile overhead dovuto alla generazione dei thread. Visto che comunque tutti i processi, prima o poi, devono usufruire del processore virtuale, visto che tutti terminano senza bloccarsi indefinitamente, visto che l'ordine di esecuzione di più processi paralleli è arbitrario, il compilatore può decidere di "fondere" tra loro questi processi producendo un codice corretto e contemporaneamente meno pesante.

Il vantaggio è tanto più evidente quanto più sono piccoli i processi in composizione parallela. Come esempio prendiamo il processo associato al pattern `v(x) | get()` nella cella:

```
0208:  __00000001:
0208:    LOAD  @get (4,a-port)
0214:    LOAD  x (8,int)
0220:    THREAD  __00000000 (8,2,a-port,int)
023c:    SPAWN
0240:    LOAD  v (0,a-port)
024c:    LOAD  x (8,int)
0258:    SEND  (1,int)
0264:    EXIT

0268:    __00000000:
0268:      LOAD  @get (0,a-port)
0274:      LOAD  x (4,int)
0280:      SEND  (1,int)
028c:      EXIT
```

```
0190:  __00000000:
0190:    LOAD  @get (4,a-port)
019c:    LOAD  x (8,int)
01a8:    SEND  (1,int)
01b4:    LOAD  v (0,a-port)
01c0:    LOAD  x (8,int)
01cc:    SEND  (1,int)
01d8:    EXIT
```

A sinistra vediamo il codice non ottimizzato in cui il thread `__00000001` prima crea un nuovo thread che dovrà inviare la continuazione alla porta `get` e successivamente invia immutato il messaggio `x` sul canale `v`. Quando si genera un nuovo thread, come in questo caso, occorre tenere conto anche dell'overhead dovuto al passaggio dell'ambiente.

A destra vediamo invece il codice ottimizzato, che appartiene ad un'unico thread. Non c'è alcuna generazione e di conseguenza nessun passaggio dell'ambiente.

Per sapere se un processo è bloccante oppure no il compilatore usa semplicemente un attributo booleano sintetizzato dal parser. Nel momento in cui il compilatore decide di ottimizzare un thread (che altrimenti sarebbe generato separatamente) deve modificare tutti i riferimenti ai nomi presenti nel codice del thread stesso.

In particolare, il compilatore appende all'area dati del thread padre l'area

dati del thread figlio senza tenere conto dell'eventuale ambiente. Per ogni riferimento x incontrato nel codice del figlio il compilatore esegue le seguenti modifiche:

- Se x è l'offset di un nome locale al thread figlio, allora il nuovo riferimento diviene $S + x$, dove S è la dimensione dell'area dati del padre prima dell'unione con quella del figlio.
- Se x è l'offset di un nome esterno al thread figlio, cioè fa parte del suo ambiente, il riferimento viene sostituito con il riferimento y valido per il padre e relativo al medesimo nome.

5.5.2 Salti condizionati

La generazione del codice intermedio per il costrutto di esecuzione condizionale porta spesso a sequenze di istruzioni non molto "belle" da vedere. In fase di generazione del codice target alcune di queste situazioni patologiche vengono modificate per ottenere un codice più lineare e leggeremente più breve.

Nell'esempio che segue viene mostrato parte del corpo della funzione fattoriale con e senza ottimizzazione (la versione non ottimizzata è a sinistra):

```
0074:  __00000002:
0074:    LOAD  x (8,int)
0080:    PUSH  0 (0)
0088:    JEQ   __00000000
0090:    JUMP  __00000001
0098:  __00000000:
0098:    LOAD  @fact (4,a-port)
00a4:    PUSH  1 (1)
00ac:    SEND  (1,int)
00b8:    EXIT
00bc:  __00000001:
```

```
006c:  __00000002:
006c:    LOAD  x (8,int)
0078:    PUSH  0 (0)
0080:    JNE   __00000001
0088:  __00000000:
0088:    LOAD  @fact (4,a-port)
0094:    PUSH  1 (1)
009c:    SEND  (1,int)
00a8:    EXIT
00ac:  __00000001:
```

Come si nota il codice di condizione del salto è stato invertito (da `JEQ` a `JNE`) permettendo di risparmiare una `JUMP`.

5.5.3 Ottimizzazioni minori

Sempre in fase di generazione del codice target vengono eseguite altre ottimizzazioni minori come la sostituzione di somme e sottrazioni di 1 con istruzioni

dedicate (`INC` e `DEC`), oppure un miglioramento nella gestione dello stack congiunta con l'accesso all'area dati. Si rimanda al codice sorgente (in particolare al file `packcode.c`) per i dettagli.

5.6 Generazione del codice oggetto

Il risultato finale della fase di generazione del codice intermedio è una rappresentazione a lista del codice. Questa rappresentazione è "imprecisa" per due particolari:

1. L'istruzione `THREAD` fa riferimento al codice del thread attraverso un puntatore *fisico* alla lista di istruzioni relativa.

2. Tutti i riferimenti a etichette (quindi quelli presenti nelle istruzioni di salto) sono in forma simbolica e non in forma di offset.

La fase di generazione del codice oggetto deve quindi tradurre tutti i riferimenti simbolici in riferimenti "fisici" per la macchina astratta (cioè sotto forma di offset all'interno del codice oggetto) e creare un unica sequenza di istruzioni concatenando opportunamente le liste di istruzioni incontrate ad ogni `THREAD`, tenendo conto che il codice di ogni locazione (cioè il codice risultante di tutti i thread definiti all'interno di ogni locazione) *deve* essere allocato in un'area contigua di memoria. Quest'ultima restrizione si rende necessaria per semplificare lo spostamento del codice durante una migrazione.

La traduzione delle etichette simboliche in riferimenti fisici avviene attraverso una tabella delle etichette ed il noto meccanismo di backpatching: ogni volta che si riferice una etichetta non registrata in tabella (tipicamente in un'istruzione di salto) si inserisce l'indirizzo dell'istruzione che fa il riferimento in una lista associata all'etichetta. Nel momento in cui si incontra l'etichetta vera e propria, tutte le istruzioni che compaiono nella lista relativa vengono "aggiustate" di conseguenza.

Diversa è la compattazione del codice dei thread. Esaminando la struttura del codice intermedio si nota come esso formi a tutti gli effetti un albero in cui ogni nodo è costituito dalla lista delle istruzioni che compongono un thread, mentre ogni istruzione `THREAD` ha un puntatore ad un nodo figlio.

Per compattare il codice è sufficiente una visita dell'albero: per ogni nodo visitato si compatta il codice presente nella lista di istruzioni e si spingono

su uno stack tutti i riferimenti ai nodi figli incontrati in corrispondenza delle istruzioni `THREAD`. Successivamente si estraggono tali riferimenti e si ricomincia il processo ricorsivamente.

Notare che questa semplice visita consente di mantenere contigui tutti i thread appartenenti ad una stessa locazione, proprio come era richiesto.

L'ultima operazione che il compilatore esegue è il salvataggio su un file binario del codice così compattato, seguito dal pool delle costanti. L'header di questo file binario ha la struttura seguente:

```
typedef struct object_file_header {
        char magic[34];
        char ver_hi, ver_lo;
        int  pool_size;
        int  code_size;
        int  data_size;
        int  word_alignment;
} OBJECT_FILE_HEADER;
```

Il campo `magic` è la stringa di identificazione del file, mentre `ver_hi` e `ver_lo` rappresentano la versione del compilatore che ha generato il codice. Seguono tre campi che indicano nell'ordine la dimensione del pool delle costanti, del codice e dell'area dati del primo thread. In pratica queste sono le informazioni che il supporto runtime userà per generare la locazione implicita di esecuzione. A posteriori è interessante notare come questi valori siano similari a quelli usati dalle istruzioni `LOCATION` e `THREAD` durante la creazione di locazioni ordinarie. Per come è stato compattato, il codice viene sempre eseguito a partire dall'indirizzo `0000`.

5.7 Invocazione del compilatore

L'invocazione del compilatore segue l'usuale sintassi dei comandi UNIX:

```
jcc [ option ... ] file
```

Il file specificato è il file sorgente da compilare (in genere con suffisso `.j`). Se non viene specificata l'opzione `-o` il file oggetto ha lo stesso nome del file sorgente, ma con suffisso `.jo`. Le opzioni riconosciute sono le seguenti:

`-S` Genera, oltre al file oggetto, un file con il codice in formato leggibile. Indispensabile per esaminare la generazione del codice, soprattutto in fase di debugging del compilatore.

`-c` Evita la generazione del codice oggetto, ma non quella eventuale del codice in formato leggibile.

`-f file` Specifica il nome del file destinato a contenere il codice in forma leggibile. Se non specificato, viene assunto il nome del file sorgente con suffisso `.js`.

`-n` Disabilita tutte le ottimizzazioni.

`-o name` Specifica il nome del file oggetto.

`-v` Visualizza alcune informazioni aggiuntive durante la compilazione, e soprattutto arricchisce le informazioni fornite con il codice in forma leggibile.

Capitolo 6

Conclusioni

In questo capitolo confronteremo il linguaggio implementato con quello da cui abbiamo preso ispirazione [FM97], sia dal punto di vista della sintassi che dal punto di vista delle prestazioni, cercando di capire dove e come sono possibili miglioramenti ed estensioni.

Presenteremo anche un esempio più complesso rispetto a quelli visti fino ad ora che rappresenta il modello semplificato e a scala ridotta di un sistema in cui può essere utile la migrazione del codice.

6.1 Differenze sintattiche

Se si escludono piccoli particolari, la differenza più significativa tra il nostro linguaggio e quello di Fournet e Maranget riguarda la gestione dei tipi, che nel nostro caso è esplicita. In effetti la scelta ha penalizzato una parte fondamentale di qualsiasi linguaggio ad alto livello, ma il tempo disponibile non avrebbe consentito un adeguato sviluppo di questo aspetto.

Il sistema di tipi del linguaggio risulta quindi molto debole, ma nonostante le differenze è molto semplice passare dalla nostra sintassi a quella di Fournet e Maranget. Nel nostro linguaggio la funzione di Fibonacci si scriverebbe così:

```
def   fibo: int -> int
where fibo(n) = {
          if n <= 1 then { reply 1 }
          else { reply fibo(n - 1) + fibo (n - 2) }
      }
```

```
;;
```

Nella maggior parte dei casi è sufficiente eliminare la *dichiarazione* che segue `def` e sostituire la parola chiave `where` con `let`:

```
let fibo(n) = {
        if n <= 1 then { reply 1 }
        else { reply fibo(n - 1) + fibo (n - 2) }
    }
;;
```

Se per le normali definizioni l'introduzione dei tipi espliciti è risultata in differenze poco significative, essa ha invece influito più pesantemente sulla sintassi per la generazione di nuove locazioni. Come prima, però, è sufficiente eliminare la parte dichiarativa e apportare lievi modifiche alla definizione per ottenere la sintassi di Fournet e Maranget [FM97].

6.2 Prestazioni

Anche se lo scopo della tesi non prevedeva la realizzazione di un sistema ad alte prestazioni, abbiamo comunque ceduto alla tentazione di mettere a confronto le due implementazioni. La nostra curiosità è stata anche alimentata dal fatto che i linguaggi usati per le implementazioni sono completamente differenti (da una parte un linguaggio funzionale, Objective Caml, dall'altra il più comune C) e che a priori non abbiamo assolutamente studiato l'implementazione esistente per mancanza di documentazione.

Nella tabella 6.1 sono riassunti i tempi di calcolo in secondi per la funzione di Fibonacci nella sua forma "classica" (che impedisce ottimizzazioni come la *tail recursion*) con argomento 27. Per ogni combinazione sono specificati i tempi della nostra implementazione (a sinistra) e quella di Fournet e Maranget (a destra).[1]

Le ottimizzazioni sono quelle effettuate dal compilatore sul codice oggetto, ma nel nostro caso il supporto runtime effettua un'altra importante ottimizzazione che è quella delle porte veloci e che non può essere disabilitata se

[1] Tutte le prove sono state eseguite su un PC ad architettura Intel con sistema operativo Linux 2.0.31 e una Sun SPARC Station Classic con sistema operativo SunOS 5.5.1 (Solaris).

Ottimizzazione	PC	Sun
Sì	21/77	148/404
No	21/78	161/403

Tabella 6.1: Tempi di esecuzione per il calcolo di `fibonacci(27)`

Ottimizzazione	PC	Sun
Sì	28/25	261/118
No	46/48	422/262

Tabella 6.2: Tempi di esecuzione per l'algoritmo QuickSort nel caso pessimo

non ricompilando il runtime. Eliminate le porte veloci, sul PC il tempo di esecuzione passa da 21 a 28 secondi.

In ogni modo, per valutare l'efficienza del meccanismo di *pattern matching* è necessario considerare un esempio in cui sono presenti più pattern e più porte. Abbiamo verificato le prestazioni del QuickSort su una lista di 100 elementi già ordinati (caso pessimo) e i risultati sono raccolti nella tabella 6.2.

In questo caso è molto più evidente l'influenza delle ottimizzazioni, in particolare quelle che riguardano i processi non bloccanti che appaiono numerosi nella codifica delle liste (si veda il file `qsort.j`). Risulta comunque chiaro che l'approccio di Fournet e Maranget per il *pattern matching* (si tratta di una mezza via tra la nostra tecnica delle stringhe di bit ed un automa a stati finiti) è efficace per *join-definition* generali.

Nell'ambito della programmazione distribuita diventa più difficile effettuare prove "cronometrate" in quanto l'implementazione di Fournet e Maranget ha dei vincoli di persistenza. In altre parole, il loro supporto runtime non termina spontaneamente alla fine della computazione, ma resta in esecuzione "per sempre" o, più realisticamente, fino alla terminazione forzata.

In ogni caso possiamo dire che da questo punto di vista la nostra implementazione è certamente più dispendiosa: la comunicazione tra runtime su macchine differenti è notevolmente più lenta della comunicazione locale, di conseguenza anche piccoli overhead nei messaggi diventano significativi (ci riferiamo in particolare alla *signature* dei messaggi).

A questo proposito non abbiamo una documentazione ufficiale che ripor-

ti le tecniche adottate nell'implementazione di Fournet e Maranget, però il manuale utente [FM97] lascia emergere alcuni dettagli sibillini: ad esempio, su una macchina a 32 bit, il range ammissibile di valori per i numeri interi è pari a $-2^{30} \ldots 2^{30} - 1$. In altre parole, un bit nella rappresentazione binaria viene "sacrificato" per scopi non documentati e probabilmente legati all'implementazione. Restrizioni analoghe sono presenti anche per altri tipi di dato, e tutto ciò fa presumere che la loro implementazione adotti una specie di codifica *expanding*, tipo quella che abbiamo accennato a pagina 73.

Infine, per quanto riguarda la nostra implementazione, ci sono diversi parametri che possono influenzare le prestazioni e anche la comunicazione dei messaggi. In particolare, tutte le strutture dinamiche, come lo heap, la tabella delle porte, lo stack, vengono allocate dinamicamente a "blocchi" di dimensione fissata.[2] Ad esempio lo heap viene allocato in blocchi di 1024 parole (equivalenti a 4096 byte in una macchina a 32 bit). Ciò significa che, anche se nello heap sono allocati pochissimi dati, il messaggio di migrazione contiene lo heap nella sua totalità.

Abbiamo implementato un sistema di compressione dei messaggi "lunghi" (tipicamente quelli per la migrazione) e abbiamo notato, dai rapporti di compressione, che effettivamente c'è una ridondanza notevole che inevitabilmente si ripercuote sui tempi di trasferimento.

6.3 Un sistema di bilanciamento del carico

Uno degli programmi più complessi che abbiamo codificato nel nostro linguaggio riguarda un modello semplificato di bilanciamento del carico. Il modello rappresenta un sistema di calcolatori connessi tra loro attraverso una rete in cui un calcolatore particolare, chiamato *master*, funge da riferimento ogni volta che si vuole iniziare una nuova computazione (*job*). Gli altri calcolatori, detti *slave*, notificano al master la propria disponibilità a "sobbarcarsi" parte del carico di lavoro in modo che il master possa distribuire i vari job al fine di ottenere un carico medio omogeneo.

L'aspetto interessante è che sia il numero dei job che la popolazione degli slave sono variabili, cosicché il sistema è soggetto a continui assestamen-

[2] Molti di questi parametri possono essere modificati nel file `const.h` relativo al supporto runtime.

ti. I job sono ovviamente rappresentati dal locazioni che, potendo migrare, trasferiscono il carico di lavoro sui vari calcolatori.

L'implementazione ovvia del sistema vedrebbe il master che mantiene un database contenente le informazioni principali sugli slave che si sono dichiarati disponibili e, in base all'arrivo di nuovi job, decide l'assegnazione di questi ultimi. Tuttavia, a causa della difficoltà di rappresentare complesse strutture dati quali un database, siamo stati costretti ad adottare una tecnica diversa.

Il master è ovviamente a conoscenza del numero totale di job e del numero di slave disponibili, per cui è in grado di stabilire il numero medio di job che ogni slave deve eseguire. Gli slave notificano *periodicamente* al master la propria disponibilità, e solo nel momento in cui avviene la notifica master e slave si scambiano informazioni per confrontare il numero effettivo di job sullo slave ed il numero medio di job previsto dal master. In base allo sbilanciamento di queste due quantità il master decide eventualmente la migrazione dei job in un verso o nell'altro.[3]

6.4 Estensioni future

Sono numerose le possibili aree di intervento nella nostra implementazione. In alcuni casi si tratta di vere e proprie estensioni per rendere il linguaggio più ricco e maneggevole, in altri casi si tratta invece di migliorare l'implementazione sul fronte delle prestazioni oppure di eliminare le semplificazioni che abbiamo adottato. Nella lista che segue vengono proposte alcune modifiche:

- Revisione completa del sistema dei tipi. Il sistema attuale è alquanto semplice e poco espressivo. L'idea è di realizzare un linguaggio *object oriented* consistente con i concetti di concorrenza, distribuzione, mobilità. Naturalmente le modifiche non riguarderebbero solo il compilatore, ma anche la macchina virtuale e di conseguenza il supporto runtime andrebbero rivisti, magari raffinando una tecnica di *garbage collection* distribuito.

[3]In questa versione, a dispetto del nome, il master è relativamente "magnanimo", in quanto se la media dei job non è un numero intero, si incarica "personalmente" di eseguire i job eccedenti.

- Ottimizzazioni nella macchina virtuale. Ad esempio si potrebbe adottare una tecnica di condivisione degli ambienti in modo da evitare il "passaggio" esplicito. Una ulteriore ottimizzazione potrebbe prevedere il precalcolo (a tempo di compilazione) della dimensione massima dello stack, in modo da evitare dispendiose allocazioni dinamiche.

- Ottimizzazioni nel pattern matching. Come abbiamo visto è possibile migliorare le prestazioni con una diversa organizzazione del sistema di matching dei pattern. Resta da individuare la strada da seguire per ottenere il miglioramento, perché la realizzazione di un sistema tipo quello impiegato da Fournet e Maranget potrebbe richiedere numerosi interventi alla struttura della macchina virtuale e del compilatore.

- Miglioramenti nell'analisi del codice sorgente, ad esempio attraverso l'introduzione di nuove passate, in modo da rendere agevoli ulteriori ottimizzazioni come la *tail recursion*.

- Supporto per la rilevazione dei guasti (il join-calculus distribuito prevede già un concetto primitivo di *failure*).

- Studio degli aspetti di sicurezza, assai rilevanti in qualsiasi sistema di natura distribuita. La sicurezza riguarda fondamentalmente due aree: sicurezza del codice, intesa a evitare che particolari istruzioni possano danneggiare più o meno involontariamente gli utenti di una macchina su cui transitano agenti, e sicurezza dei domini amministrativi, che riguarda la protezione di aree private della rete.

Appendice A

Introduzione al join-calculus

Il join-calculus nasce a partire dalla CHAM (Chemical Abstract Machine) e si propone come un modello formale di concorrenza consistente con i concetti di mobilità e distribuzione.

Uno degli aspetti interessanti di questo calcolo è che già in esso si intravedono, seppur a basso livello, concetti legati alla programmazione funzionale e object-oriented, cose che rendono il calcolo, ed il linguaggio da esso derivato, estremamente intuitivo ed espressivo.

A.1 La reflexive CHAM

Il modello opera su "soluzioni" della forma $\mathcal{R} \vdash \mathcal{M}$, dove $\mathcal{R}$ ed $\mathcal{M}$ sono rispettivamente multiinsiemi di reazioni e molecole.

Le molecole sono composte di *atomi* uniti tra loro attraverso l'operatore di join, "|". Una reazione ha la forma $J \triangleright P$: nel momento in cui nella soluzione compare una molecola con la stessa forma di J, la reazione consuma quella molecola e ne produce una nuova di forma P.

Come semplice esempio, supponiamo di avere tre atomi distinti, indicati con a, b e c, supponiamo inoltre che una reazione plausibile sia quella che consuma due atomi a e un atomo b per produrre la molecola composta dal solo atomo c. In questo caso la reazione avrebbe la forma $D = a \mid a \mid b \triangleright c$. Vediamo un possibile stato della CHAM in presenza di alcuni atomi a, b e c:

$$a \mid a \mid b \triangleright c \vdash a, a, a, b, c$$

Le molecole nella soluzione possono combinarsi o disgregarsi attraverso due processi reversibili e contrari detti *raffreddamento* e *riscaldamento*. Vediamo come potrebbe evolvere la CHAM vista sopra:

$$\begin{aligned} & D \vdash a, a, a, b, c \\ \rightleftharpoons \quad & D \vdash a \mid a, a, b, c \\ \rightleftharpoons \quad & D \vdash a \mid a \mid b, a, c \end{aligned}$$

Fino ad ora non è avvenuta alcuna reazione, semplicemente la soluzione si è trasformata in un'altra soluzione *equivalente*. Vedremo in seguito qual è il significato di questa equivalenza. A questo punto è comparsa una molecola che combacia con il *pattern* della reazione D; la reazione può quindi avvenire:

$$\begin{aligned} & D \vdash a \mid a \mid b, a, c \\ \longrightarrow \quad & D \vdash a, a, c \end{aligned}$$

Il join-calculus non è costruito direttamente sulla CHAM presentata, bensì sulla sua versione "riflessiva", in cui le reazioni possono essere create dinamicamente. Ciò è ottenuto attraverso una molecola di forma particolare, **def** D **in** P, il cui comportamento può essere schematizzato così:

$$\begin{aligned} & \mathcal{R} \vdash \textbf{def } D \textbf{ in } P \\ \rightleftharpoons \quad & \mathcal{R}, D \vdash P \end{aligned}$$

Da notare che la creazione della regola avviene per equivalenza, e non per reazione.

Manca ancora un anello di congiunzione tra la reflexive CHAM ed il join-calculus: si tratta dell'interpretazione da dare alle molecole nell'ambito della computazione. Come è facile intuire, le molecole non sono altro che processi in esecuzione "parallela", l'equivalenza di cui si parlava non è altro che la relazione di equivalenza strutturale tra processi e i join-pattern sono il modo con cui i processi si sincronizzano e scambiano informazioni.

A.2 Descrizione del modello

Presentiamo nella tabella A.1 la sintassi precisa per i processi, i join-pattern e le definizioni.

$$
\begin{array}{lcl}
P & \stackrel{\text{def}}{=} & x\langle\tilde{v}\rangle \\
 & & \textbf{def } D \textbf{ in } P \\
 & & P \mid P
\end{array}
\qquad
\begin{array}{lcl}
J & \stackrel{\text{def}}{=} & x\langle\tilde{v}\rangle \\
 & & J \mid J
\end{array}
\qquad
\begin{array}{lcl}
D & \stackrel{\text{def}}{=} & J \triangleright P \\
 & & D \wedge D
\end{array}
$$

Tabella A.1: Sintassi del join-calculus

I nomi, come è consuetudine nei calcoli per la concorrenza, costituiscono le entità primitive fondamentali e nel seguito saranno sempre scritti in lettere minuscole, come $x \in \mathcal{N}$, mentre $\tilde{x}$ denoterà una tupla di nomi $x_1, x_2, \ldots, x_n$.

Un processo P può essere l'emissione asincrona di un messaggio poliadico $x\langle\tilde{v}\rangle$, la definizione di una nuova reazione o la composizione parallela di più processi. Una definizione è composta da diverse definizioni elementari $J \triangleright P$ che fanno corrispondere un pattern J ad un "processo con guardia" (*guarded process*) P. Le definizioni elementari sono connesse tra loro tramite l'operatore $\wedge$.

L'unico *binder* è il join-pattern J e lo scope dei nomi che vi compaiono dipende dalla loro posizione nei messaggi. In un generico pattern $x\langle\tilde{v}\rangle \mid y\langle\tilde{w}\rangle \mid \ldots$ i nomi x e y sono detti nomi di porte *definite*, mentre i nomi che compaiono nelle tuple $\tilde{v}$ e $\tilde{w}$ sono detti nomi di variabili *ricevute*. Nella definizione **def** $J \triangleright P$ **in** Q i nomi delle porte definite sono legati sia in P che in Q, mentre le variabili ricevute sono legate solo in P.

Esiste una restrizione sintattica per i join-pattern: un nome non può comparire più volte nello stesso pattern J. Questa restrizione esclude pattern del tipo $x\langle v\rangle \mid y\langle v\rangle$ oppure $x\langle v\rangle \mid x\langle w\rangle$, evitando che per verificare l'attivazione di un pattern siano necessari dei confronti tra i valori ricevuti.

La tabella A.2 mostra come sono calcolate le variabili definite, ricevute e libere. Un nome viene definito *fresh* (nuovo) in un processo o in una soluzione se non è libero. Nel seguito si useranno le sostituzioni σ e $\{x/y\}$ per evitare collisioni tra nomi.

$$
\begin{array}{lcl}
rv(x\langle\tilde{v}\rangle) & \stackrel{\text{def}}{=} & \{u \in \tilde{v}\} \\
rv(J \mid J') & \stackrel{\text{def}}{=} & rv(J) \cup rv(J') \\
\\
dv(x\langle\tilde{v}\rangle) & \stackrel{\text{def}}{=} & \{x\} \\
dv(J \mid J') & \stackrel{\text{def}}{=} & dv(J) \cup dv(J') \\
\\
dv(J \triangleright P) & \stackrel{\text{def}}{=} & dv(J) \\
dv(D \wedge D') & \stackrel{\text{def}}{=} & dv(D) \cup dv(D') \\
\\
fv(J \triangleright P) & \stackrel{\text{def}}{=} & dv(J) \cup (fv(P) - rv(J)) \\
fv(D \wedge D') & \stackrel{\text{def}}{=} & fv(D) \cup fv(D') \\
\\
fv(x\langle\tilde{v}\rangle) & \stackrel{\text{def}}{=} & \{x\} \cup \{x \in \tilde{v}\} \\
fv(\textbf{def } D \textbf{ in } P) & \stackrel{\text{def}}{=} & (fv(P) \cup fv(D)) - dv(D) \\
fv(P \mid P') & \stackrel{\text{def}}{=} & fv(P) \cup fv(P')
\end{array}
$$

Tabella A.2: Variabili definite, ricevute e libere

A.3 Regole di riduzione

Vengono ora presentate le regole di riduzione, dette anche regole di reazione nel contesto particolare della CHAM, insieme con le regole che stabiliscono l'equivalenza strutturale tra processi. Per semplicità, solo le molecole coinvolte nelle riduzioni sono mostrate:

$$
\begin{array}{lrcl}
(\textbf{str-join}) & \vdash P \mid Q & \rightleftharpoons & \vdash P, Q \\
(\textbf{str-and}) & D \wedge E \vdash & \rightleftharpoons & D, E \vdash \\
(\textbf{str-def}) & \vdash \textbf{def } D \textbf{ in } P & \rightleftharpoons & D\sigma_{\text{dv}} \vdash P\sigma_{\text{dv}} \\
\\
(\textbf{red}) & J \triangleright P \vdash J\sigma_{\text{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\text{rv}}
\end{array}
$$

- (str-def) σ_{dv} è una sostituzione che istanzia le variabili $\text{dv}(D)$ a nomi distinti e *fresh*: $Dom(\sigma_{\text{dv}}) \cap (fv(\mathcal{R} \vdash \mathcal{M})) = \emptyset$;
- (red) σ_{rv} sostituisce i nomi trasmessi alle variabili ricevute $rv(J)$.

A.4 La cella

Di seguito è mostrata una possibile codifica della cella, in cui il "valore" della cella stessa è mantenuto in un messaggio e il meccanismo di interazione segue uno stile object-oriented:

$$\textbf{def}\ \mathit{mkcell}\langle v_0, \kappa_0\rangle\ \triangleright \left(\begin{array}{rl} \textbf{def} & \mathit{get}\langle\kappa\rangle \mid s\langle v\rangle\ \triangleright\ \kappa\langle v\rangle \mid s\langle v\rangle \\ \wedge & \mathit{set}\langle u, \kappa\rangle \mid s\langle v\rangle\ \triangleright\ \kappa\langle\rangle \mid s\langle u\rangle \\ \textbf{in} & s\langle v_0\rangle \mid \kappa_0\langle \mathit{get}, \mathit{set}\rangle \end{array}\right)$$

Ad ogni "invocazione" di *mkcell* viene creata ed inizializzata una nuova cella e i nomi *get* e *set* vengono restituiti al chiamante attraverso il canale di ritorno κ_0. In questo modo l'accesso alla cella è consentito esclusivamente per mezzo dei metodi restituiti, in quanto lo stato s è coperto da scoping lessicale.

A.5 Estensione: continuazione implicta

Già nell'esempio della cella ci si rende conto di come ogni risultato debba essere ritornato attraverso un messaggio esplicito. Ciò non deve sorprendere, visto che il join-calculus nasce proprio come modello in cui l'unica forma di comunicazione è asincrona, tuttavia esistono molteplici situazioni in cui è utile poter descrivere una esecuzione *sequenziale*. Supponiamo ad esempio di dover inviare lungo un canale, diciamo *print*, una sequenza ordinata di valori.[1] La soluzione "classica" sarebbe la seguente:

$$\textbf{def}\ \mathit{print}\langle x, \kappa\rangle\ \triangleright\ \ldots\kappa\langle\rangle\ldots\ \textbf{in def}\ \kappa\langle\rangle\ \triangleright\ \mathit{print}\langle 4, \kappa'\rangle\ \textbf{in}\ \mathit{print}\langle 3, \kappa\rangle$$

Una volta appurato che il controllo sequenziale è ammissibile si può arricchire la sintassi del calcolo con apposito zucchero sintattico e scrivere:

$$\textbf{def}\ \mathrm{print}(x)\ \triangleright\ \ldots\ \textbf{reply to}\ \mathrm{print}\ldots\textbf{in}\ \mathrm{print}(3); \mathrm{print}(4)$$

I nomi sincroni verranno scritti in tondo per ricordare che portano un canale implicito di continuazione. Formalmente essi vengono tradotti nel modo seguente (i κ_x sono nomi *fresh*):

[1] Anche se i valori numerici non sono compresi nella sintassi del calcolo è comodo usarli per rendere l'esempio meno astratto.

$$\begin{array}{rcll} \mathrm{x}(\tilde{v}) & \stackrel{\text{def}}{=} & x\langle \tilde{v}, \kappa_x \rangle & \text{(nei join pattern J)} \\ \textbf{reply } \tilde{V} \textbf{ to x} & \stackrel{\text{def}}{=} & \kappa_x \langle \tilde{V} \rangle & \text{(nei processi P con guardia)} \end{array}$$

Altri costrutti sintattici, quali **let**, istruzioni in sequenza, chiamate annidate, non verrano descritti qui, ma saranno comunque usati nel seguito (vedere [FGL+96] per i dettagli).

A.6 Computazioni distribuite

Il modello analizzato fino ad ora può dare l'impressione che tutta la computazione avvenga in un unico "sito". Vediamo allora come estendere la reflexive CHAM per modellare sistemi distribuiti introducendo il concetto di locazione.

A.6.1 La distributed reflexive CHAM

Una distributed reflexive CHAM (brevemente DRCHAM) è un multiinsieme di CHAM. Lo stato globale del sistema è rappresentato da diverse soluzioni $\mathcal{R}_i \vdash \mathcal{P}_i$ separate dal simbolo $\|$. Soluzioni distinte possono interagire attraverso la nuova riduzione:

$$(\textbf{comm}) \quad \vdash x\langle \tilde{v} \rangle \parallel J \triangleright P \vdash \longrightarrow \vdash \parallel J \triangleright P \vdash x\langle \tilde{v} \rangle \quad (x \in dv(J))$$

In pratica, un messaggio emesso su una porta definita in una CHAM distinta viene trasportato fino al sito di definizione. Successivamente questo messaggio può interagire localmente attraverso la regola **red**. Questa forma di comunicazione "globale" è dunque realizzata in due passi e riflette ciò che accade nelle implementazioni reali, in cui il trasporto ed il trattamento dei messaggi sono operazioni distinte. Come modello di computazione distribuita, la DRCHAM nasconde i dettagli di routing dei messaggi, ma non quelli di sincronizzazione.

Da ora in avanti si considereremo solo DRCHAM *ben formate*, in cui ogni nome è definito al più in una soluzione. Questa condizione assicura un trasporto dei messaggi deterministico e point-to-point.

A.6.2 L'albero delle locazioni

Una locazione verrà vista in join-calculus sia come definizione sintattica che come soluzione chimica locale in esecuzione. Il nome di una locazione servirà a correlare i due concetti.

Si useranno le lettere $a, b, \ldots \in \mathcal{L}$ per i nomi di locazione e $\varphi, \psi, \ldots \in \mathcal{L}^*$ per stringhe finite di nomi di locazione, ε per la stringa nulla. Una locazione in esecuzione è indicata come una soluzione "etichettata" $\mathcal{R} \vdash_\varphi \mathcal{P}$. Si dice che $\vdash_\varphi$ è una sub-locazione di $\vdash_\psi$ quando ψ è un prefisso di φ. Nel seguito considereremo una DRCHAM come un albero di soluzioni etichettate.

I nomi di locazione sono considerati entità di prima classe e possono essere creati, inviati e ricevuti nei messaggi. La categoria sintattica delle definizioni si arricchisce di un ulteriore costrutto per introdurre nuove locazioni:

$$D \stackrel{\text{def}}{=} \ldots \mid a[D : P]$$

In fase di "riscaldamento" questo costrutto ha l'effetto di creare una sotto-locazione della locazione corrente che contiene la definizione:

$$(\textbf{str-loc}) \quad a[D : P] \vdash_\varphi \rightleftharpoons \vdash_\varphi \parallel \{D\} \vdash_{\varphi a} \{P\} \quad (a \textit{ frozen})$$

La condizione a lato significa che questa regola di equivalenza è applicabile solo quando non sono presenti soluzioni della forma $\vdash_{\varphi a \psi}$ con $\psi \neq \varepsilon$. In pratica si asserisce che D contiene l'intero sotto-albero delle sub-locazioni di a. Questa restrizione avrà implicazioni importanti in seguito, in quanto farà in modo che una locazione in movimento porti con sé tutte le sub-locazioni contenute in essa.

Come era stato detto in precedenza, si assume di lavorare sempre con DRCHAM ben formate, ovvero in cui le regole di reazione che definiscono un certo nome appartengono alla stessa locazione. Per mantenere questo invariante si assume che, in una definizione $D \wedge D'$, $dv(D) \cap dvD'$ contenga solo nomi di porte che non sono definiti in una sub-locazione di D o D', in pratica sono escluse situazioni del tipo:

$$\textbf{def}\ x\langle\tilde{v}\rangle \triangleright \ldots \wedge a[\textbf{def}\ x\langle\tilde{u}\rangle \triangleright \ldots]$$

A.6.3 Esempio: Remote Procedure Call

Vediamo un semplice esempio che ricorda il paradigma di chiamata di procedure remote. Un server mette a disposizione un servizio attraverso una procedura f, il client invoca il servizio attraverso una funzione *stub rpc*:

$$\begin{array}{ll} & \mathrm{f}(y) \ \triangleright\ \textbf{reply}\ \mathrm{computation}(y)\ \textbf{to}\ \mathrm{f} \ \vdash_s \\ \| & \vdash_c\ \textbf{def}\ \mathrm{rpc}(g,x)\ \triangleright\ \textbf{reply}\ \mathrm{g}(x)\ \textbf{to}\ \mathrm{rpc}\ \textbf{in}\ \ldots \mathrm{rpc}(\mathrm{f},3)\ldots \end{array}$$

Una soluzione più sofisticata modella anche la possibilità che scatti un *timeout* durante la chiamata:

$$\begin{array}{l} \textbf{def}\ \mathrm{rpc}(\mathrm{f},x,\mathit{error})\ \triangleright \\ \quad \begin{array}{ll} \textbf{def} & \mathit{incall}\langle\rangle \mid \mathit{done}\langle r\rangle\ \triangleright\ \textbf{reply}\ r\ \textbf{to}\ \mathrm{rpc} \\ \wedge & \mathit{incall}\langle\rangle \mid \mathit{timeout}\langle\rangle\ \triangleright\ \mathit{error}\langle\rangle \\ \textbf{in} & \mathit{incall}\langle\rangle \mid \mathit{done}\langle \mathrm{f}(x)\rangle \mid \mathit{start_timer}\langle \mathit{timeout}, 5\rangle \end{array} \\ \textbf{in}\ \ldots \mathrm{rpc}(\mathrm{f},3,\mathit{error_handler})\ldots \end{array}$$

Il messaggio *incall* garantisce la mutua esclusione tra il messaggio di terminazione normale della procedura remota ed il messaggio di timeout proveniente dal timer.

A.7 Locazioni in movimento

A.7.1 Migrazione

Ora che abbiamo definito le locazioni, e implicitamente una nozione di unità di movimento, non è difficile immaginare una primitiva che consenta alle locazioni di *migrare*. Di conseguenza, si aggiunge a quelle esistenti una nuova regola di riduzione:[2]

$$P \stackrel{\mathrm{def}}{=} \ldots \mid \mathit{go}\langle b,\kappa\rangle$$

$$(\textbf{move}) \qquad a[D : P|\mathit{go}\langle b,\kappa\rangle] \ \vdash_\varphi \ \| \ \vdash_{\psi b} \longrightarrow \vdash_\varphi \ \| \ a[D : P|\kappa\langle\rangle] \ \vdash_{\psi b}$$

[2]La sintassi completa del join-calculus esteso (che include il processo nullo **0** e la definizione nulla **T**), le regole di scoping dei nomi, le regole di equivalenza e riduzione, possono essere trovate in [FGL+96].

Intuitivamente, la locazione a si muove dalla soluzione $\vdash_{\phi a}$ alla nuova soluzione $\vdash_{\psi ba}$. La continuzione $\kappa\langle\rangle$ può essere utile per attivare una computazione nel momento in cui lo spostamento è terminato. La condizione a lato della (**str-loc**) assicura che quando a si muove, tutte le sue sub-locazioni si muovono con lei.

Occorre tenere presente che, sebbene *go* venga usata con la stessa sintassi usata per i canali, essa è una *primitiva* e non può essere inviata come valore nei messaggi.

A.7.2 Esempio: esecuzione del codice *à la Java*

In Java gli applet vengono scaricati sul client a richiesta. Questo tipo di interazione può essere modellato in join-calculus così:

$$\begin{array}{rl} & \mathrm{load_applet}(a) \ \triangleright\ \mathbf{def}\ b[\mathrm{applet}(y) \ \triangleright\ \mathbf{reply}\ \ldots\ \mathbf{to}\ \mathrm{applet} \\ & \qquad\qquad : \mathrm{go}(a); \mathbf{reply}\ \mathrm{applet}\ \mathbf{to}\ \mathrm{load_applet}]\ \mathbf{in}\ \mathbf{0}\ \vdash_s \\ \| & \vdash_c\ \mathbf{let}\ \mathrm{f} = \mathrm{load_applet}(c)\ \mathbf{in}\ \ldots \mathrm{f}(3)\ldots \end{array}$$

Una volta che la richiesta perviene al server, quest'ultimo crea un'istanza dell'applet che, per mezzo della primitiva "go", si trasferisce nella locazione da cui è partita la chiamata, portando alla soluzione seguente:

$$\begin{array}{rl} & \mathrm{load_applet}(a) \ \triangleright\ \mathbf{def}\ b[\mathrm{applet}(y) \ \triangleright\ \mathbf{reply}\ \ldots\ \mathbf{to}\ \mathrm{applet} \\ & \qquad\qquad : \mathrm{go}(a); \mathbf{reply}\ \mathrm{applet}\ \mathbf{to}\ \mathrm{load_applet}]\ \mathbf{in}\ \mathbf{0}\ \vdash_s \\ \| & b'[\mathrm{applet}(y) \ \triangleright\ \mathbf{reply}\ \ldots\ \mathbf{to}\ \mathrm{applet}]\ \vdash_c\ \ldots \mathrm{applet}(3)\ldots \end{array}$$

A.7.3 Esempio: computazione remota

L'esempio speculare al precedente è l'invio al server del codice che porta a termine la computazione e successivamente torna al client per comunicare il risultato:

$$\mathbf{def}\ \mathrm{f}(x,s)\ \triangleright\ a[\mathrm{go}(s); \mathbf{reply}\ \ldots\ \mathbf{to}\ \mathrm{f} : \mathbf{0}]\ \mathbf{in}\ \ldots \mathrm{f}(3, \mathit{server})\ldots$$

L'aspetto importante è che, se l'agente alloca dei dati sul server, questi dati vengono trasferiti insieme all'agente quando quest'ultimo fa ritorno al client.

Appendice B

La libreria Pthread

In questa appendice descriveremo brevemente le funzioni di libreria relative alla gestione dei thread secondo lo standard POSIX 1003.4a [RR96].

B.1 Thread POSIX

Lo standard POSIX ha definito una libreria di sistema per la gestione di applicazioni multithreaded in ambiente UNIX. Com'è consuetudine per questi tipi di standard, POSIX ha definito unicamente una interfaccia lasciando gli aspetti implementativi a discrezione dei programmatori di sistema.

L'aspetto forse più rilevante tra quelli non decisi dallo standard riguarda lo "scope" di un thread. I due modelli tradizionali classificano i thread in *thread a livello utente* e *thread a livello kernel*.

I thread a livello utente in genere vengono eseguiti a insaputa del sistema operativo. In pratica un normale processo UNIX viene scomposto in più thread, ma questa scomposizione è trasparente al kernel che di fatto tratta il processo come un'unica entità. La politica di scheduling del kernel riguarda esclusivamente i processi.

I thread a livello kernel invece costituiscono la vera unità di scheduling del sistema operativo e in questo si avvicinano molto di più ai tradizionali processi, con la differenza che i thread di una stessa applicazione condividono le risorse allocate per il processo.

Entrambi i modelli hanno caratteristiche interessanti e penalizzanti. Ad esempio, essendo i thread a livello kernel gestiti dallo scheduler del sistema

operativo, essi comportano normalmente un maggior impiego della CPU per il passaggio da un thread all'altro, anche se inferiore al tempo per il passaggio tra processi diversi. Tuttavia, il fatto che il sistema operativo sia a conoscenza dei singoli thread può essere sfruttato in sistemi multiprocessore. In questi casi il kernel può decidere di allocare i thread appartenenti ad una stessa applicazione su processori distinti, raggiungendo un miglior grado di parallelismo.

Esiste di fatto un ulteriore modello, detto modello ibrido, che consiste nella combinazione dei due modelli appena descritti. In pratica il kernel suddivide un'applicazione in più entità soggette alle politiche di scheduling, ma ogni entità, a sua volta, può essere ulteriormente scomposta in thread a livello utente.

B.2 Gestione dei thread

Riportiamo di seguito una breve descrizione delle principali funzioni della libreria POSIX thread usate nell'implementazione del supporto runtime.

```
int pthread_create (pthread_t *th,
                    pthread_attr_t *attr,
                    pthread_func_t func,
                    any_t arg);
```

Questa funzione causa la creazione di un nuovo thread il cui identificatore viene memorizzato in `th`. Il corpo del thread è costituito dalla funzione `func` che viene invocata passando come unico argomento `arg`. Il parametro `attr` può essere usato per associare alcuni attributi ad un thread. Ad esempio, un uso tipico consiste nell'assegnare ai thread diverse priorità di esecuzione.

```
void pthread_exit (any_t status);
```

Termina il thread che invoca la funzione con il codice di terminazione `status`. Se il thread era stato marcato in precedenza con lo stato "detachable" tutte le risorse per il thread vengono deallocate.

```
int pthread_join (pthread_t th,
                  any_t *status);
```

Blocca il thread che invoca la funzione in attesa della terminazione del thread `th`. Il codice di terminazione del thread viene passato in `status`. Questa funzione causa anche il rilascio di tutte le risorse del thread che termina.

```
int pthread_detach (pthread_t *th);
```

Marca il thread `th` con l'attributo "detachable".

B.3 Sincronizzazione

I meccanismi di base definiti dallo standard per la sincronizzazione dei thread comprendono i cosiddetti "mutex" e le variabili di condizione. I primi sono adatti a proteggere brevi sezioni critiche e sono di fatto comparabili con le comuni primitive *lock* e *unlock* nell'ambito dei sistemi operativi concorrenti, mentre le variabili di condizione sono utili per sincronizzazioni più complesse.

Le funzioni per la creazione e la cancellazione dei "mutex" comprendono:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Le funzioni seguenti, invece, si occupano della sincronizzazione vera e propria:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

`pthread_mutex_lock` causa il blocco del thread che la esegue se il mutex non è disponibile, `pthread_mutex_trylock` è la versione non bloccante della precedente, mentre `pthread_mutex_unlock` serve per rendere disponibile un mutex. Un mutex appena creato è sempre disponibile.

Le funzioni per la gestione delle variabili di condizione ricalcano lo stesso schema visto per i mutex:

```
int pthread_cond_init (pthread_cond_t *cond,
                       pthread_condattr_t *attr);
int pthread_cond_destroy (pthread_cond_t *cond);
```

`pthread_cond_init` crea una nuova variabile di condizione. Le variabili di condizione sono sempre associate ad un mutex che protegge i dati condivisi che fanno parte del predicato da verificare.

```
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex);
```

Se il predicato di controllo risulta essere falso si invoca la funzione `pthread_cond_wait` che sospende il thread e contemporaneamente rilascia la mutua esclusione.

```
int pthread_cond_signal (pthread_cond_t *cond);
```

Quando un thread modifica le variabili che fanno parte del predicato, notifica gli altri thread in attesa sulla condizione con la chiamata alla funzione `pthread_cond_signal`. È conveniente invocare la funzione solo all'uscita della sezione critica per minimizzare il numero di prerilasci.

Bibliografia

[ASU85] A. V. Aho, Ravi Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[Bar91] H. P. Barendregt. *The Lambda Calculus*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, revised edition, 1991.

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 50–64. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[F+97] David Flanagan et al. *Java in a Nutshell*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, deluxe edition, 1997.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, 21–24 January 1996.

[FGL+96] C. Fournet, G. Gonthier, JJ. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag, Berlin.

[FM97] C. Fournet and L. Maranget. *The Join-Calculus Language release 1.02 - Documentation and User's Manual*, 1997.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Progamming Language*. Prentice Hall, Englewood Cliffs, 2 edition, 1988.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[Mil93] Robin Milner. Elements of interaction. *CACM: Communications of the ACM*, 36, 1993.

[MS86] M. Mauny and A. Suárez. Implementing functional languages in the categorical abstract machine. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 266–278. ACM, August 1986.

[RR96] K. Robbins and S. Robbins. *Practical Unix Programming*. Prentice-Hall, 1996.

[Ste90] Richard Stevens. *UNIX Network Programming*. Software Series. Prentice Hall PTR, 1990.

[Ste92] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, USA, 1992.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, Massachusetts, 1977.