Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica
Materia di Tesi: Linguaggi di programmazione

Implementazione di un linguaggio di programmazione distribuito

Tesi di Laurea di:
Paolo Milazzo

Relatore:
Chiar.mo Prof. Cosimo Laneve

Correlatore:
Dott. Lucian Wischik

II Sessione
Anno Accademico 2002-2003

Contents

Prefazione

Questo documento è scritto completamente in inglese. Lo scopo di questa prefazione è di illustrare brevemente e in lingua italiana il lavoro svolto. Si tratta perciò di una introduzione che si propone di dare una visione panoramica dell'intero lavoro.

Negli ultimi anni, le reti di computer hanno rapidamente sostituito i computer stand–alone come ambiente per l'esecuzione di applicazioni software. Al giorno d'oggi, quasi tutte le applicazioni più diffuse utilizzano Internet per comunicare, per procurare informazioni o per auto–aggiornarsi. In futuro, la relazione tra il software e la rete diventerà ancora più forte: ci si attende, infatti, che la rete venga utilizzata per la comunicazione fra componenti software collaborativi, ovvero che le applicazioni diventino *distribuite*.

L'esempio principe di componenti software distribuiti sono, ad oggi, i *Web Service* [Con02]. I Web Service sono interfacce che descrivono operazioni accessibili tramite la rete. L'esecuzione di tali operazioni può essere richiesta dalle applicazioni tramite l'invio di messaggi XML standardizzati. In questo scenario, quindi, un'applicazione distribuita può essere vista come un'*orchestrazione* di operazioni messe a disposizione da Web Service.

A fronte di questa evoluzione delle applicazioni software, non corrisponde però un'evoluzione dei linguaggi di programmazione. I linguaggi attualmente più diffusi si basano in maggioranza sui paradigmi di programmazione imperativo o orientato agli oggetti. Tali paradigmi non sono fatti per descrivere esecuzioni parallele e distribuite di componenti software, e per questo motivo spesso diventa difficile descrivere correttamente il comportamento di un'applicazione o verificarne delle proprietà.

Concorrenza e distribuzione possono invece essere descritte e studiate utilizzando una delle tante varianti del π–calculus [MPW92]: un'algebra di processi che negli ultimi dieci anni è divenuta lo strumento più diffuso per ragionare formalmente su questioni di parallelismo e interoperabilità. Tramite il π–calculus si possono descrivere insiemi di processi eseguiti in parallelo che comunicano tra loro utilizzando message passing sincrono su canali. Un canale del π–calculus è un entità, identificata da un nome, che può essere creata dai processi, può essere utilizzata per inviare e ricevere messaggi e il cui nome può essere utilizzato come contenuto di una comunicazione.

Tra le innumerevoli varianti del π–calculus quella sicuramente più utilizzata dalla comunità di ricerca è il π–calculus *asincrono* [MS98]. In tale variante l'invio di messaggi su canali è effettuato in modo asincrono, quindi non bloccante: questa caratteristica meglio si presta a descrivere le comunicazioni sulla rete. Un'altra variante del π–calculus (che verrà utilizzata in questo lavoro) è invece il *localized linear forwarder calculus* [GLW03]. In tale algebra si usano comunicazioni asincrone, vi è la possibilità di redirezionare singoli messaggi da un canale ad un altro ed esiste una definizione implicita di *locazione* che può essere utilizzata per descrivere l'aspetto di distribuzione.

Lo scopo principale di questo lavoro è di definire *Hipi*: un linguaggio di programmazione distribuito basato sul π–calculus. Tramite questo linguaggio sarà possibile scrivere programmi distribuiti il cui comportamento potrà essere studiato attraverso i metodi normalmente utilizzati per il π–calculus. Si definirà inoltre un linguaggio intermedio, *XML Intermediate Language*, basato sul localized linear forwarder calculus, nel quale i programmi verranno compilati e il quale verrà interpretato da un insieme di macchine virtuali distribuite sulla rete. Il processo di compilazione verrà descritto formalmente e ne verrà dimostrata la correttezza. Inoltre si implementerà il compilatore seguendone la specifica formale e si fornirà un semplice simulatore di una rete di macchine virtuali distribuite che potrà essere utilizzato per interpretare il codice intermedio. In fine si analizzeranno alcuni aspetti particolari della progettazione e della programmazione di applicazioni distribuite in hipi: si descriveranno tecniche di implementazione e di design che consentono di migliorare l'efficienza delle applicazioni sotto diversi punti di vista.

Il percorso che verrà seguito nello svolgere questo lavoro andrà, in tre passi, dalla teoria all'implementazione: innanzitutto si creeranno le fondamenta teoriche del compilatore utilizzando un nucleo di hipi; in seguito si presenteranno le sintassi complete di hipi e XIL e si descriverà l'implementazione del compilatore; in fine si illustreranno strumenti, design pattern ed esempi che illustreranno come progettare e scrivere programmi in hipi in maniera efficiente.

Fondamenta teoriche Si definisce un nucleo di hipi (chiamato *hipi core language*) che sostanzialmente contiene i termini del π–calculus con variabili tipate anziché semplici nomi di canali, con un concetto di locazione (il che rende il linguaggio distribuito) e con un costrutto per la migrazione di processi tra locazioni. Si definisce, inoltre, un linguaggio intermedio (detto *bytecode*) basato sul localized linear forwarder calculus ed esteso con un termine per la migrazione di processi. Si definiscono le sintassi operazionali e denotazionali di entrambi i linguaggi ed un encoding $[\![\,\cdot\,]\!]$ che traduce programmi da hipi core language a bytecode. Tale encoding rappresenta il processo di compilazione e per garantirne la correttezza si dimostra che il comportamento del programma sorgente (rispetto alla semantica dell'hipi core language) è equivalente al comportamento del codice generato (rispetto alla semantica del bytecode).

Il risultato conclusivo che si dimostra è che dati due programmi nell'hipi core language P e Q, si ha:

$$P \dot{\approx} Q \iff [\![P]\!] \dot{\approx} [\![Q]\!]$$

dove $\dot{\approx}$ è la relazione che indica equivalenza di comportamento fra processi nello stesso linguaggio (più precisamente è la *barbed bisimulation* [MS92]).

Hipi, XIL e l'implementazione del compilatore Si definisce hipi come un'estensione dell'hipi core language comprendente, tra l'altro, costrutti di programmazione sequenziale come funzioni e cicli for. Si illustra un encoding $\{\![\,\cdot\,]\!\}$ che permette di tradurre programmi hipi in programmi nel linguaggio core. Si definisce l'XML Intermediate Language (XIL) come riscrittura, con sintassi XML, del bytecode utilizzato per il lavoro teorico. In seguito si descrive l'implementazione del compilatore da hipi a XIL: tale compilatore è basato sull'encoding $\{\![\,\cdot\,]\!\}$ appena definito e sull'encoding $[\![\,\cdot\,]\!]$ di cui si è provata la correttezza. Si descrive inoltre l'implementazione del simulatore utilizzato come interprete XIL: tale simulatore viene inizializzato utilizzando un documento XML che descrive le caratteristiche della rete nella quale si vuole effettuare la simulazione. In seguito è possibile caricare ed eseguire uno o più file XIL e monitorarne l'esecuzione.

Progettazione e programmazione in hipi Si illustra innanzitutto come le variabili mutabili e le strutture dati più comuni possano essere implementate in hipi. In seguito si affrontano problemi di progettazione definendo tre design pattern per hipi. Il primo design pattern presentato è "il Delegato", il quale descrive come minimizzare il numero di messaggi scambiati fra un produttore e un consumatore tramite la migrazione di un processo. Il secondo pattern, invece, è "l'Assistente", il quale descrive come realizzare un semplice algoritmo dinamico di bilanciamento di carico. In seguito si ha il ben noto design pattern "Proxy" [GHJG94], che descrive come progettare un surrogato locale per un servizio offerto remotamente. In conclusione si presenta un esempio completo di programma hipi: tale esempio è l'implementazione di un servizio on–line di traduzione Italiano/Inglese.

Chapter 1

Introduction

In the past few years, computer networks have replaced stand-alone computers as the environment where programs are executed. Nowadays, a lot of widely used applications use networks for communication, to retrieve information or to upgrade themselves. The future direction of programs is to be *distributed* on networks: this means that an application could be divided in several sub–parts that are executed in different computers of a network. These sub–parts can be used by different programs and this allows the sharing of the execution load. From the point of view of business, it allows to define new kinds of software licences that give access to a distributed application, instead of giving permission to run a local copy of it.

The main example of such new kinds of applications are *Web Services* [Con02]. A Web Service is *"an interface that describes a collection of operations that are network–accessible through standardized XML messaging"* [Kre01]. These operations can access other operations on different Web Services, so a Web Application can be seen as an *orchestration* of a set of Web Services. The use of XML as object of the communications between Web Services and the use of standard protocols like http [FGM+99] and SOAP [Con03] allow for the interaction of services written in different languages and executed in different environments.

This massive use of networks in programs has not so far been accompained by the evolution of programming languages. The most popular programming languages, also used to write distributed applications, are based on formalisms which represent sequential execution well, but which do not express clearly the effects of parallelism and distribution. This makes difficult to reason about properties of programs, and requires the use of system calls for communications and parallelism that usually are not programmer–friendly.

The π–calculus of Milner, Parrow and Walker [MPW92] is a widely studied formalism for describing and reasoning about concurrent systems. It is based on synchronous message passing over channels where the objects of communications are themselves channel names. The asynchronous π–calculus of Merro and Sangiorgi [MS98] is a well-known subcalculus of the π–calculus where communications are based on asynchronous message passing. Finally, the *localized linear forwarder calculus* of Gardner, Laneve and Wischik [GLW03] is an extension of the asynchronous π–calculus with a new term (the linear forwarder) that allows the redirection of output messages. In the localized linear forwarder calculus there is an implicit notion of location: it is used to specify where a channel resides and where a process is executed. In the localized linear forwarder calculus a process executing an input on a channel must be *co-located* with such a channel. Hence, input–capability (which is the ability to receive a channel name and subsequently receive messages on it) is not possible; but it can be encoded using linear forwarders.

All the calculi just described allow us to describe of the behavior of concurrent (and distributed) processes and allow us to study properties of distributed applications. A programming language based on one of them could be used to write distributed programs whose behavior and whose properties can be studied using the theories of the calculus. This motivated me to define and implement such a language.

1.1 Description of the Work

In this work I define *hipi*, a distributed programming language based on the π–calculus. It compiles into *XML Intermediate Language (XIL)* based on the localized linear forwarder calculus, which is executed by an interpreter. A hipi program specifies a computation composed of a set of parallel processes where interaction is achieved by message passing over channels. In hipi there is an implicit notion of location: it is intuitively defined as "the place where a channel resides" or "the place where a process is executed". Two channels x and y are *colocated* if they reside at the same location. In similar ways two processes can be colocated and a process can be colocated with a channel. The presence of locations makes the language *distributed*.

The equivalent of locations in the real world depends on how the language and the runtime environment are implemented. For instance, a location can be a processor with its private memory in a multi–processor system, or a workstation on a local area network, or an Application Server on Internet. In our implementation a location corresponds to a virtual machine that acts as a *channel server* and also as an interpreter for programs compiled into the XML Intermediate Language. Channels are represented by URIs [BLFM98] and a possible way to implement communications on them is by using the *http* protocol [FGM+99]. An example of such a runtime environment, but for the explicit fusion calculus [Wis01], is described in [Per03].

The work described in this thesis has three parts:

1. Define hipi and XIL, define their semantics and give an encoding from the former to the latter that is proved to be correct. I do this theoretical work on a subset of hipi called the *hipi core language* and then I give an encoding from hipi into hipi core.

2. Implement a compiler from hipi into XIL based on the encodings given in point 1. I also implement a network simulator that acts as a XIL interpreter. It is used to execute compiled programs.

3. Show how hipi might be used in practice by a programmer. I describe how to obtain some features that are not part of the language and I present some design patterns.

1.2 Related Works

In recent years a number of π–calculus variants have been introduced in order to study some aspect of distributed and mobile processes. They include:

- The π_1–calculus of Amadio [Ama00], for modelling distributed computations with mobility of processes, failures and routing of messages;

- The *distributed* π–calculus $D\pi$ of Hennessy and Riely [HR98], that extends the π–calculus with explicit locations and migrations;

- The Distributed Join Calculus of Fournet et al [FGL+96], intended as the basis for a mobile agent language;

- The Fusion Calculus of Parrow and Victor [PV98] and the Explicit Fusion Calculus of Gardner and Wischik [GW00], where communications are symmetric and channels can be *fused*.

In particular, the explicit fusion calculus was used as a foundation for the Fusion Machine of Gardner, Laneve and Wischik [GLW02], a forerunner of the localized linear forwarder calculus.

Some programming languages based on the π–calculus, or on one of its variants, were also implemented. Some of them include:

- Pict of Pierce and Turner [PT00], a strongly–typed concurrent programming language without mobility of processes;

- Nomadic Pict of Sewell, Wojciechowski and Pierce; [WS00], which has mobile agents and explicit locations

- The Join Calculus Language [Joi02] and Jocaml [CF99] both based on the Join Calculus.

Chapter 2

Core Language and Compiler Theory

The aim of this chapter is to define the theoretical foundations of our work. We first describe the *hipi core language* and the *bytecode*. Then we define their semantics and the encoding from the former to the latter. Finally, we prove the correctness of the encoding.

The hipi core language is a variant of the *synchronous* π–calculus [MPW92] which includes mobility of processes (through an explicit migration term) and some simple data types (integers, strings and sorted channels). The bytecode is a variant of the localized linear forwarder calculus [GLW03] with explicit migrations and data types (integers, strings and unsorted channels). In the following we do not describe formally the type checker or the properties of the type system. As a future work, we expect to replace this simple type system with XML data types as in XDuce [HP03].

2.1 Hipi Core Language Syntax

We now define the syntax of the hipi core language. We assume an infinite set of variable identifiers $\mathcal{X}$ ranged over by $u, w, x, y, z, \ldots$, an infinite set of type identifiers ranged over by $r, s, t, \ldots$ and an infinite set of identifiers for recursive definitions $\mathcal{R}$ ranged over by $D, E, \ldots$. We use $\widetilde{x}$ to denote the (possibly empty) sequence $x_1, \ldots, x_n$. Sometimes we also use the same notation $\widetilde{x}$ to denote the set $\{x_1, \ldots, x_n\}$.

A *program* in the hipi core language is a triple (Φ,Γ,P) where Φ is a mapping from type identifiers into type expressions, Γ is a mapping from recursive definition names into abstractions and P is a process. Φ allows us to use type identifiers as types for variables declared inside a program and it allows recursive types. Γ is a *function environment*: it allows the definition of recursive definition calls (as in π–calculus). Finally, P is the "main" process executed at run–time.

Definition 2.1 (Process) *Processes in the hipi core language are defined by the following grammar:*

$P ::= \mathbf{0}$	nil
$\mid\ x\ .\ \mathtt{send}\ (\ \widetilde{E}\ )\ ;\ P$	channel output
$\mid\ x\ .\ \mathtt{recv}\ (\ \widetilde{Tx}\ )\ ;\ P$	channel input
$\mid\ \mathtt{let}\ T\ x\ =\ E\ \mathtt{in}\ P$	new constant
$\mid\ \mathtt{new}\ T\ x\ \mathtt{in}\ P$	new channel

$\mid$ `if` (E) { P } { P }	if then else
$\mid$ D ($\widetilde{E}$) ; P	recdef call
$\mid$ `spawn` [@ x] { P } P	new process

Processes P include statements used for operations on channels, declaration of variables and control–flow commands. The term **0** represents the empty process and we usually omit it at the end of a sequence of statements. The `send` and the `recv` statements are the output and the input terms of the π–calculus with typed objects. The `let` defines a new immutable variable x assigning E to it (as a constant) and its scope is the following P. The statement `new` creates a new channel x and its scope is the following P. The `if` is as usual in programming languages. The recursive definition call is as usual in π–calculus [Mil99] but includes a continuation P that is executed in parallel. Finally, the `spawn` statement can be used to execute two processes in parallel.

The optional argument `@x` of the `spawn` statement can be used for the *explicit migration* of a process. Explicit migration are optimizations for communications: if the spawned process (that is the process that executes the first occurrence of P in the statement) executes a lot of communications on the channel x then it could be convenient for it to migrate to the location of x: this migration is achieved with the `@x` option.

Abstractions are as in π–calculus [Mil99] with typed variables instead of channel names. An abstraction $A = (\widetilde{Tx}).P$ can be instanciated using a concretion term $A\langle\widetilde{y}\rangle$. This concretion becomes $P\{\widetilde{y}/\widetilde{x}\}$.

Definition 2.2 (Types) *Types of the language are the followings:*

T ::= `int`	integer type
$\mid$ `string`	string type
$\mid$ `channel` $< \widetilde{T} >$	channel type
$\mid$ t	type identifier

Types are only `int`, `string` and `channel<...>`. The type system is completed with type identifiers: using them it is possible to define recursive types.

Definition 2.3 (Expressions) *Expressions and literal values are described by the following grammar:*

E ::= x	variable
$\mid$ *LiteralInt*	integer value
$\mid$ *LiteralString*	string value
$\mid$ $E + E \mid E - E$	sum and sub
$\mid$ $E * E \mid E \ / \ E \mid E \ \% \ E$	mul, div and mod
$\mid$ $E > E \mid E == E \mid E \ != E$	comparators
$\mid$ $E \ \&\& \ E \mid \ ! \ E$	boolean operators

Expressions E can be used to specify arguments of invocations of recursive definitions, and also the entities sent on channels. They are evaluated during execution (i.e. in the operational semantics). A *LiteralInt* is an element of $\mathbb{Z}$ and a *LiteralString* represents a string. Operators on expressions are all standard: combining them it is possible to obtain also `<`,`<=`,`>=`, the unary minus and the boolean choice $||$.

Definition 2.4 (Well–Formed Hipi Core Program) *A* well–formed *program in the hipi core language satisfies the following requirements:*

1. *All the operations on variables, functions and channels always respect their types;*
2. *All the free names in* main *are used with the same (inferred) channel type;*
3. *In recursive definition bodies, the free names are a subset of the parameters of the recursive definition.*

In what follows we consider only well–formed programs.

Examples We give two simple examples of programs in the hipi core language. The first example creates a new channel and reacts on it:

$$\begin{aligned}
\Phi &= \varnothing \\
\Gamma &= \varnothing \\
P &= \texttt{new channel<> x in} \\
&\ \texttt{spawn \{ x.send(); \}} \\
&\ \texttt{x.recv();}
\end{aligned}$$

The second example illustrates recursive type declarations and recursive definitions. It executes an infinite sequence of reactions between two processes, so simulating a ping pong match. The objects of the communications are the same channels used as subjects.

$$\Phi = \big\{(\texttt{pingpongT}, \texttt{channel<pingpongT>})\big\}$$

$$\begin{aligned}
\Gamma = \ &\big\{(\texttt{recursive_player}, (\texttt{pingpongT ping}).R)\big\} \\
&\text{where } R = \texttt{ping.recv(pingpongT pong);} \\
&\phantom{\text{where } R = }\ \texttt{pong.send(ping);} \\
&\phantom{\text{where } R = }\ \texttt{recursive_player(ping);}
\end{aligned}$$

$$\begin{aligned}
P = \ &\texttt{new pingpongT p1 in} \\
&\texttt{new pingpongT p2 in} \\
&\texttt{recursive_player(p1);} \\
&\texttt{recursive_player(p2);} \\
&\texttt{p1.send(p2);}
\end{aligned}$$

2.2 Hipi Core Language Semantics

In this section we describe the operational and the denotational semantics of the hipi core language.

Definition 2.5 (State) *A state of the system is a triple* (Φ, Γ, S) *where* Φ *is a mapping from type identifiers to type expressions,* Γ *is an environment and* S *is a set of parallel processes defined by the following grammar:*

$$S ::= \mathbf{0} \quad \big| \quad P \quad \big| \quad S \mid S \quad \big| \quad (x)S$$

where P *is as in Definition 2.1.*

In the following, when Φ and Γ are obvious, we write only S to identify a state. The sets of bound names and free names of a state S, denoted by $bn(S)$ and $fn(S)$, are as in the π–calculus with `new Tx`, `y.recv(x)` and (x) as binders for the name x. Let $R, S, T, \ldots$ range over states.

Now we define *structural congruence*.

Definition 2.6 (Structural Congruence) *Structural congruence on states $\equiv$ is the smallest equivalence relation satisfying the following and closed with respect to alpha–renaming and parallel composition:*

$$S \mid \mathbf{0} \equiv S \qquad S \mid T \equiv T \mid S \qquad S \mid (T \mid R) \equiv (S \mid T) \mid R$$

$$(x)\mathbf{0} \equiv \mathbf{0} \qquad (x)(y)S \equiv (y)(x)S$$

$$(x)(S \mid T) \equiv S \mid (x)T \quad \textit{if } x \notin \mathit{fn}(S)$$

We give a big step semantics for expression evaluation. This semantics is used to evaluate expressions inside some rules of the transition system. Note that variables are immutable and expressions have no side effects. We assume i, j, k ranging over *LiteralInt* and s over *LiteralString*.

Definition 2.7 (Semantics of Expressions) *The relation for expression evaluation $E \downarrow v$ is the smallest relation satisfying the following:*

$$i \downarrow i \qquad s \downarrow s \qquad x \downarrow x$$

$$\frac{E_1 \downarrow i \qquad E_2 \downarrow j \qquad i = j}{(E_1 == E_2) \downarrow 1} \qquad \frac{E_1 \downarrow i \qquad E_2 \downarrow j \qquad i \neq j}{(E_1 == E_2) \downarrow 0}$$

$$\frac{E_1 \downarrow i \qquad E_2 \downarrow j \qquad i > j}{(E_1 > E_2) \downarrow 1} \qquad \frac{E_1 \downarrow i \qquad E_2 \downarrow j \qquad i \leq j}{(E_1 > E_2) \downarrow 0}$$

$$\frac{E_1 \downarrow i \qquad E_2 \downarrow j \qquad k = i \times j}{(E_1 \&\& E_2) \downarrow k} \qquad \frac{E \downarrow 0}{!E \downarrow 1} \qquad \frac{E \downarrow i \qquad i \neq 0}{!E \downarrow 0}$$

$$\frac{E_1 \downarrow i \qquad E_2 \downarrow j \qquad k = i \; op \; j}{(E_1 \; op \; E_2) \downarrow k} \qquad op \in \{+, -, *, /, \%\}$$

It will transpire, in the operational semantics, that the evaluation $x \downarrow x$ will only happen when x is a channel name. For this reason we define the *values* of the operational semantics as literal integers, literal strings and channel names. All the expressions in the operational semantics are evaluated into values. Let v range over values.

Definition 2.8 (Reactions) *The reaction relation $(\Phi, \Gamma, S) \rightarrow (\Phi, \Gamma, S')$ between states is as follows. Φ and Γ never change; we assume them in the following rules. Reactions are closed with respect to structural congruence $\equiv$, restriction $(x)\_$, parallel composition $\_ \mid \_$ and alpha–renaming:*

$$\text{spawn}\{P_1\}P_2 \quad \rightarrow \quad P_1 \mid P_2 \qquad \text{(spawn)}$$

$$\text{spawn@}x\{P_1\}P_2 \quad \rightarrow \quad P_1 \mid P_2 \qquad \text{(migrate)}$$

$$\text{new}T x \text{ in } P \quad \rightarrow \quad (x)P \qquad \text{(new)}$$

$$\frac{E_1 \downarrow v_1 \; \cdots \; E_n \downarrow v_n}{x.\text{send}(\widetilde{E});P_1 \mid x.\text{recv}(T_1 y_1, \ldots, T_n y_n);P_2 \quad \rightarrow \quad P_1 \mid P_2\{\widetilde{v}/\widetilde{y}\}} \qquad \text{(react)}$$

$$\frac{E_1 \downarrow v_1 \ \cdots \ E_n \downarrow v_n}{D(\widetilde{E});P \ \ \rightarrow \ \ \Gamma(D)\langle\widetilde{v}\rangle \mid P} \qquad \text{(call)}$$

$$\frac{E \downarrow v}{\text{let } Tx{=}E \text{ in } P \ \ \rightarrow \ \ P\{^v/x\}} \qquad \text{(let)}$$

$$\frac{E \downarrow i \qquad i \neq 0}{\text{if}(E)\{P_1\}\{P_2\} \ \ \rightarrow \ \ P_1} \qquad \text{(if1)}$$

$$\frac{E \downarrow 0}{\text{if}(E)\{P_1\}\{P_2\} \ \ \rightarrow \ \ P_2} \qquad \text{(if2)}$$

We write $\Rightarrow$ for a sequence of zero or more reactions $\rightarrow^*$.

As discussed, the intention of the optional `@x` argument in `spawn` is to support location–based optimizations. But our current high–level semantics abstract away from location, and so the `@x` argument has no effect. It will have effect, however, in the low–level bytecode semantics (which do model locations).

The (react) transition is as usual in the synchronous π–calculus, but with expressions and typed variables. Recursive definitions calls are usually removed from the reactions using structural congruence, instead we define (call) in order to handle them. Other reactions are standard.

Now we define the *observation relation* and the related *weak barbed bisimulation*. Let α range over labels x and $\overline{x}$.

Definition 2.9 (Observation) *The observation relation $S \downarrow \alpha$ is the smallest relation satisfying the followings:*

$$\begin{aligned}
x.\text{send}(\widetilde{y});S &\downarrow \overline{x} \\
x.\text{recv}(\widetilde{y});S &\downarrow x \\
S \mid T \downarrow \alpha & \quad \textit{if } S \downarrow \alpha \textit{ or } T \downarrow \alpha \\
(x)S \downarrow \alpha & \quad \textit{if } S \downarrow \alpha \textit{ and } \alpha \neq x, \overline{x}
\end{aligned}$$

We write $\Downarrow$ for $\Rightarrow\downarrow$.

Definition 2.10 (Barbed Bisimulation) *A relation $\mathcal{R}$ is a weak barbed bisimulation if whenever $S_1 \ \mathcal{R} \ S_2$ then*

1. *$S_1 \downarrow \alpha$ implies $S_2 \Downarrow \alpha$*
2. *$S_1 \rightarrow S_1'$ implies that exists S_2' such that $S_2' \Rightarrow S_2'$ and $S_1' \ \mathcal{R} \ S_2'$*
3. *$S_2 \downarrow \alpha$ implies $S_1 \Downarrow \alpha$*
4. *$S_2 \rightarrow S_2'$ implies that exists S_1' such that $S_1 \Rightarrow S_1'$ and $S_2' \ \mathcal{R} \ S_1'$*

We define $\dot{\approx}$ to be the largest weak barbed bisimulation.

2.3 Bytecode Syntax

A hipi program will be compiled into bytecode. The bytecode is based on the localized linear forwarder calculus [GLW03] and it differs from the hipi core in the following points: it is *asynchronous*, it includes a *linear forwarder* term, it lacks the *let* operator and it merges the parallel composition and the recursive definition call into a single statement. We have chosen a unified operator for parallel compositions and for recursive definition calls because their behavior are very similar (both execute a parallel process). A linear forwarder allows the redirection of an output message from a channel to another one. It is used to encode input operations on remote channels as in [GLW03]

A program in bytecode is a set of *named threads*. A named thread is a pair (n, A) where n is the *thread names* and A is an abstraction. A thread name is a name with which one can start the execution of the thread. The first thread executed by a program is the one with distinguished name `main`. We require thread names to be unique, and so we define a bytecode program as a function Γ_{bc} from thread names into abstractions. We assume an infinite set of thread names $\mathcal{N}$ ranged over by `main`,n,m,. . ..

Definition 2.11 (Bytecode Syntax) *A bytecode program is a function Γ_{bc} from thread names n into abstractions $(\widetilde{x}).T$ where T is a list of statements given by:*

$$
\begin{array}{rll}
T & ::= \mathbf{0} & \text{nil} \\
 & \mid\ \nu x.T & \text{new channel} \\
 & \mid\ \overline{x}\widetilde{E} & \text{channel output} \\
 & \mid\ x(\widetilde{y}).T & \text{channel input} \\
 & \mid\ x \multimap y & \text{linear forwarder} \\
 & \mid\ \text{spawn}(n)\langle\widetilde{x}\rangle.T & \text{new process} \\
 & \mid\ @x.T & \text{migration} \\
 & \mid\ [E]T, T & \text{if then else}
\end{array}
$$

where E is as in Definiton 2.3 .

The input term $x(\widetilde{y}).T$ where T is the continuation, and the output term $\overline{x}\widetilde{y}$ are as in the asynchronous π–calculus. The statement $\nu x.T$ is used to create a new channel and the scope of the name x is the continuation T. The linear forwarder $x \multimap y$ is the same as in the linear forwarder calculus [GLW03]. The command spawn$(n)\langle\widetilde{x}\rangle.T$ executes the thread n using $\widetilde{x}$ as parameters of the concretion. The statement $@x.T$ executes T at the location of the channel x. The if-then-else and the nil terms are standard. We often omit $\mathbf{0}$ at the end of a bytecode fragment.

We now define well–formedness for bytecode programs. We recall that an input capability is the ability of receive a channel name and than accept inputs on it.

Definition 2.12 (Well–Formed Bytecode Program) *A well-formed bytecode program satisfies the following:*

1. *No input capabilities are present;*

2. *In threads, apart from* `main`*, the free names are a subset of the parameters of their abstractions;*

3. *The* `main` *thread is an abstraction with no parameters;*

4. *Expressions and channels operations are well–typed.*

The no-input-capability property is inherited by the localized linear forwarder calculus [GLW03]. It is extended to parameters of abstractions in point 2: this is done because names in the parameters can be instanced to either local or remote channel by concretions. In the following we consider only well–formed programs.

Examples Here we show how the examples of section 2.1 can be expressed in the bytecode language. The first example executes a reaction:

$$\begin{array}{rcl} \texttt{main} & : & \nu x.\mathrm{spawn}(\mathit{child})\langle x\rangle.x() \\ \texttt{child} & : & (x').\overline{x'} \end{array}$$

The second example, instead, simulates a ping pong match:

$$\begin{array}{rcl} \texttt{main} & : & \nu p_1.\nu p_2.\mathrm{spawn}(\mathit{recursive\_player})\langle p_1\rangle. \\ & & \mathrm{spawn}(\mathit{recursive\_player})\langle p_2\rangle.\overline{p_1}p_2 \\ \texttt{recursive\_player} & : & (p).\nu q.\mathrm{spawn}(n)\langle p,q\rangle.q(p').\overline{p'}p. \\ & & \mathrm{spawn}(\mathit{recursive\_player})\langle p\rangle \\ \texttt{n} & : & (p,q).p\multimap q \end{array}$$

In `recursive_player`, the input operation on p is executed through a linear forwarder in order to satisfy well–formedness.

2.4 Bytecode Semantics

We now define the state of the operational semantics as a set of *located machines* executing processes. Processes executed by a machine are its *body*. We consider *co–location* as an equivalence relation on channel names and we define locations as sets of co–located channels.

Definition 2.13 (State) *A state of the system is a pair (Γ_{bc}, M) where Γ_{bc} is a function from thread names into abstractions and M is a set of located machines defined by the following grammar:*

$$\begin{array}{rlll} M ::= & \mathbf{0} \;\big|\; \widetilde{u}\big[\,\widetilde{z} : B\,\big] \;\big|\; M, M & & \textit{set of machines} \\ B ::= & \mathbf{0} \;\big|\; T \;\big|\; B \mid B & & \textit{bodies} \end{array}$$

where T is as in Definition 2.11 . In the machine $\widetilde{u}\big[\,\widetilde{z} : B\,\big]$, all the names in $\widetilde{u}\widetilde{z}$ are said to be defined *by the machine. In a set of machines, no name may be multiply defined.*

A *machine* is an entity that represents a location. $\widetilde{u}\big[\,\widetilde{z} : B\,\big]$ represents a machine at location $\widetilde{u}\widetilde{z}$, where the *external* channel $\widetilde{u}$ existed before execution began, and the *internal* channels $\widetilde{z}$ were created at this location in the course of execution. The scope of $\widetilde{z}$ is the whole set of machines. The body B of the machine is the set of processes executed at this location. Different machines of a state execute in parallel at different locations. Let $M, N, \ldots$ range over machines. In the follows we often assume the Γ_{bc} and we omit it from states.

Definition 2.14 (External and Internal Channels) *Given a machine M the sets $ec(M)$ and $ic(M)$ representing the* external *and* internal *channels of M are given by:*

$$ec(\mathbf{0}) = \varnothing \qquad ec(\widetilde{u}\big[\,\widetilde{z} : B\,\big]) = \widetilde{u} \qquad ec(M_1, M_2) = ec(M_1) \cup ec(M_2)$$

$$ic(\mathbf{0}) = \varnothing \qquad ic(\widetilde{u}\big[\,\widetilde{z} : B\,\big]) = \widetilde{z} \qquad ic(M_1, M_2) = ic(M_1) \cup ic(M_2)$$

and the set $dc(M)$ containing the defined channels *of M is*

$$dc(M) = ec(M) \cup ic(M)$$

Bound names $bn(B)$ and free names $fn(B)$ of a body B are defined as in the π–calculus. We extend their definitions to machines: let $M = \widetilde{u}\big[\,\widetilde{v} : B\,\big], M'$ we define $bn(M)$ and $fn(M)$ as $bn(B) \cup bn(M')$ and $fn(B) \cup fn(M')$.

Definition 2.15 (Structural Congruence) *The structural congruence $\equiv$ for states of the system is the smallest equivalence relation satisfying the following and closed with respect to alpha renaming and to parallel composition of machines $M, \_$ and $\_ , M$:*

$$\widetilde{u}[\,\widetilde{z} : B \mid \mathbf{0}\,] \equiv \widetilde{u}[\,\widetilde{z} : B\,] \qquad \widetilde{u}[\,\widetilde{z} : B_1 \mid B_2\,] \equiv \widetilde{u}[\,\widetilde{z} : B_2 \mid B_1\,]$$

$$\widetilde{u}[\,\widetilde{z} : (B_1 \mid B_2) \mid B_3\,] \equiv \widetilde{u}[\,\widetilde{z} : B_1 \mid (B_2 \mid B_3)\,]$$

$$M, \mathbf{0} \equiv M \qquad M_1, M_2 \equiv M_2, M_1 \qquad (M_1, M_2), M_3 \equiv M_1, (M_2, M_3)$$

$$\widetilde{u}[\,\widetilde{z} : B\,], M \equiv x\widetilde{u}[\,\widetilde{z} : B\,], M \equiv \widetilde{u}[\,x\widetilde{z} : B\,], M \qquad \text{if } x \notin fn(B) \cup fn(M)$$

We now define the operational semantics. We recall expression evaluation $E \downarrow v$ given in Definition 2.7 .

Definition 2.16 (Reactions) *The reaction relation $(\Gamma_{bc}, M) \to (\Gamma_{bc}, M')$ between states is as follows. Since Γ_{bc} never change, we omit it. Reactions are closed with respect to structural congruence, parallel composition of bodies and alpha–renaming:*

$$\widetilde{u}[\,\widetilde{z} : \overline{x}\widetilde{E} \mid x \multimap y\,] \;\to\; \widetilde{u}[\,\widetilde{z} : \overline{y}\widetilde{E}\,] \quad \text{if } x \in \widetilde{u}\widetilde{z} \tag{react-fwd}$$

$$\widetilde{u}[\,\widetilde{z} : \overline{x}\widetilde{E}\,], \widetilde{w}[\,\widetilde{k} : \,] \;\to\; \widetilde{u}[\,\widetilde{z} : \,], \widetilde{w}[\,\widetilde{k} : \overline{x}\widetilde{E}\,] \quad \text{if } x \in \widetilde{w}\widetilde{k} \tag{move-out}$$

$$\widetilde{u}[\,\widetilde{z} : x \multimap y\,], \widetilde{w}[\,\widetilde{k} : \,] \;\to\; \widetilde{u}[\,\widetilde{z} : \,], \widetilde{w}[\,\widetilde{k} : x \multimap y\,] \quad \text{if } x \in \widetilde{w}\widetilde{k} \tag{move-fwd}$$

$$\widetilde{u}[\,\widetilde{z} : \mathrm{spawn}(n)\langle\widetilde{x}\rangle.T\,] \;\to\; \widetilde{u}[\,\widetilde{z} : \Gamma_{bc}(n)\langle\widetilde{x}\rangle \mid T\,] \tag{spawn}$$

$$\widetilde{u}[\,\widetilde{z} : @x.T\,], \widetilde{w}[\,\widetilde{k} : \,] \;\to\; \widetilde{u}[\,\widetilde{z} : \,], \widetilde{w}[\,\widetilde{k} : T\,] \quad \text{if } x \in \widetilde{w}\widetilde{k} \tag{migrate1}$$

$$\widetilde{u}[\,\widetilde{z} : @x.T\,] \;\to\; \widetilde{u}[\,\widetilde{z} : T\,] \quad \text{if } x \in \widetilde{u}\widetilde{z} \tag{migrate2}$$

$$\widetilde{u}[\,\widetilde{z} : \nu x.T\,] \;\to\; \widetilde{u}[\,x'\widetilde{z} : T\{^{x'}/_{x}\}\,] \quad \text{if } x \notin \widetilde{u}\widetilde{z} \tag{new}$$

$$\frac{E_1 \downarrow v_1 \cdots E_n \downarrow v_n}{\widetilde{u}[\,\widetilde{z} : \overline{x}\widetilde{E} \mid x(\widetilde{y}).T\,] \;\to\; \widetilde{u}[\,\widetilde{z} : T\{^{\widetilde{v}}/_{\widetilde{y}}\}\,]} \quad \text{if } x \in \widetilde{u}\widetilde{z} \tag{react}$$

$$\frac{E \downarrow i \qquad i \neq 0}{\widetilde{u}[\,\widetilde{z} : [E]T_1, T_2\,] \;\to\; \widetilde{u}[\,\widetilde{z} : T_1\,]} \tag{if1}$$

$$\frac{E \downarrow 0}{\widetilde{u}[\,\widetilde{z} : [E]T_1, T_2\,] \;\to\; \widetilde{u}[\,\widetilde{z} : T_2\,]} \tag{if2}$$

$$\frac{M_1 \;\to\; M_2}{M_1, M_3 \;\to\; M_2, M_3} \quad \text{if } dc(M_2) \cap dc(M_3) = \varnothing \tag{context}$$

The transition system has two reaction rules: between one input and one output on a channel (react) and between one output and a linear forwarder (react-fwd). Reactions can occur only at the location where the subject channel resides. Transitions (move-fwd) and (move-out) describe the mobility of linear forwarders and output terms: input terms don't move, so they must be co–located with their subject channel. In (new) a new internal channel is created using a fresh name; the side condition for (context) ensures that this fresh name does not clash with the rest of the machine.

Lemma 2.17 *Given a state M:*

1. *If $M \rightarrow M'$ then $ec(M) = ec(M')$;*
2. *If $M \rightarrow M'$ then $ic(M) \subseteq ic(M')$.*

Proof. Part 1 is straightforward because no rules of the operational semantics change the set of external channel of the state. Also part 2 is straightforward because the only rule that changes the number of internal channels is (new) that increases it by one. □

Now we define the *observation relation* and the related *weak barbed bisimulation.* We chose to observe input as well as output commands: this is unusual for an asynchronous calculus but it allows us to easily compare the behavior of an hipi core program with the behavior of a bytecode program.

Definition 2.18 (Observation for Bodies) *The observation relation for bodies $B \downarrow \alpha$ is the smallest relation satisfying the followings:*

$$\begin{aligned} \overline{x}\widetilde{E} &\downarrow \overline{x} \\ x(\widetilde{Ty}).T &\downarrow x \\ x \multimap y &\downarrow x \\ B_1 \mid B_2 &\downarrow \alpha \quad \text{if } B_1 \downarrow \alpha \text{ or } B_2 \downarrow \alpha \end{aligned}$$

Definition 2.19 (Observation for Machines) *The observation relation for machines $M \downarrow \alpha$ is the smallest relation satisfying the followings:*

$$\begin{aligned} \widetilde{u}\big[\,\widetilde{z} : B\,\big] \downarrow x &\quad \text{if } B \downarrow x \;\;,\;\; x \in \widetilde{u} \;\;,\;\; x \notin \widetilde{z} \\ \widetilde{u}\big[\,\widetilde{z} : B\,\big] \downarrow \overline{x} &\quad \text{if } B \downarrow \overline{x} \;\;,\;\; x \in \widetilde{u} \;\;,\;\; x \notin \widetilde{z} \\ M_1, M_2 \downarrow \alpha &\quad \text{if } M_1 \downarrow \alpha \text{ or } M_2 \downarrow \alpha \end{aligned}$$

We write $\Downarrow$ for $\Rightarrow\downarrow$.

Definition 2.20 (Barbed Bisimulation) *A relation $\mathcal{R}$ is a weak barbed bisimulation if whenever $M \,\mathcal{R}\, N$ then*

1. *$M \downarrow \alpha$ implies $N \Downarrow \alpha$*
2. *$M \rightarrow M'$ implies $\exists N'$ such that $N \Rightarrow N'$ and $M' \,\mathcal{R}\, N'$*
3. *$N \downarrow \alpha$ implies $M \Downarrow \alpha$*
4. *$N \rightarrow N'$ implies $\exists M'$ such that $M \Rightarrow M'$ and $M' \,\mathcal{R}\, N'$*

Let $\dot{\approx}$ be the largest weak barbed bisimulation. Through abuse of notation we use the same symbol $\dot{\approx}$ for machines as for hipi core states. It will be clear from the context which is intended.

2.5 Core Language Encoding

In this section we give the encoding from the hipi core language into bytecode and we prove it correct. The main encodings concern recursive definitions and communications (due to the asynchrony and the no–input–capability constraints of the bytecode).

As said in section 2.1, a program in the hipi core language is a triple (Φ,Γ,P). We assume a well–formed and type–checked program as input of the encoding. We assume a 1-1 mapping μ between recursive definition names and thread names: the name `main` cannot be an output of the mapping. We write n_D for $\mu(D)$.

Definition 2.21 (Encoding) *The translation $[\![\cdot]\!]$ from a hipi core program (Φ,Γ,P) into a bytecode program is as follows. It makes use of a subsidiary translation $[\![\cdot]\!]_{\widetilde{u}}$, detailed below, which translates individual processes.*

$[\![(\Phi,\Gamma,P)]\!]$ is the least bytecode program containing the following:

- *For $[\![P]\!]_{\varnothing}=(T_M,\Gamma_M)$, let $[\![(\Phi,\Gamma,P)]\!]$ contain $\Gamma_M\cup(\mathtt{main},().T_M)$*
- *For every $(D,(\widetilde{x}).P')$ in Γ, with $[\![P']\!]_{\varnothing}=(T',\Gamma')$, let $[\![(\Phi,\Gamma,P)]\!]$ contain $\Gamma'\cup(n_D,(\widetilde{x}).T')$*

The subsidiary translation $[\![\cdot]\!]_{\widetilde{u}}$ from a hipi process into a bytecode fragment (T,Γ) is as follows. It is parametrized on a set $\widetilde{u}$ of channel names which are known to be local to P.

$$[\![T'x{=}E \text{ in } P]\!]_{\widetilde{u}} = [\![P\{^{E}/x\}]\!]_{\widetilde{u}}$$

$$[\![\text{new}T'x \text{ in } P]\!]_{\widetilde{u}} = (\,\nu x.T\ ,\ \Gamma_{bc}\,) \qquad \textit{where} \quad [\![P]\!]_{x\widetilde{u}}=(T,\Gamma_{bc})$$

$$[\![\text{if}(E)\{P_1\}\{P_2\}]\!]_{\widetilde{u}} = (\,[E]T_1,T_2\ ,\ \Gamma_{bc1}\cup\Gamma_{bc2}\,) \qquad \textit{where} \quad [\![P_i]\!]_{\widetilde{u}}=(T_i,\Gamma_{bci})$$

$$[\![x.\text{send}(\widetilde{E});P]\!]_{\widetilde{u}} = (\,\nu y.\text{spawn}(n)\langle y\rangle.y().T\ ,\ \Gamma_{bc}[n\mapsto(y).\overline{x}\langle y,\widetilde{E}\rangle]\,)$$
$$\textit{where} \quad y\notin \mathit{fn}(P) \quad n \textit{ fresh} \quad [\![P]\!]_{\widetilde{u}}=(T,\Gamma_{bc})$$

$$[\![x.\text{recv}(\widetilde{Tv});P]\!]_{\widetilde{u}} = \begin{cases} (\,x(y,\widetilde{v}).\text{spawn}(n)\langle y\rangle.T\ ,\ \Gamma_{bc}[n\mapsto(y).\overline{y}]\,) & \textit{if } x\in\widetilde{u} \\ \quad \textit{where } n \textit{ fresh} \quad [\![P]\!]_{\widetilde{u}}=(T,\Gamma_{bc}) & \\ & \\ (\,\nu w.\text{spawn}(n)\langle xw\rangle.w(y,\widetilde{v}).\text{spawn}(m)\langle y\rangle.T\ , & \\ \Gamma_{bc}[n\mapsto(xw).x\multimap w\ ,\ m\mapsto(y).\overline{y}]\,) & \textit{otherwise} \\ \quad \textit{where } w\notin \mathit{fn}(P) \quad n{,}m \textit{ fresh} \quad [\![P]\!]_{\widetilde{u}}=(T,\Gamma_{bc}) & \end{cases}$$

$$[\![D(\widetilde{E});P]\!]_{\widetilde{u}} = (\,\text{spawn}(n_D)\langle\widetilde{E}\rangle.T\ ,\ \Gamma_{bc}\,) \qquad \textit{where} \quad [\![P]\!]_{\widetilde{u}}=(T,\Gamma_{bc})$$

$$[\![\text{spawn}\{P_1\}P_2]\!]_{\widetilde{u}} = (\,\text{spawn}(n)\langle \mathit{fn}(P_1)\rangle.T_2\ ,\ \Gamma_{bc1}\cup\Gamma_{bc2}\cup\{n\mapsto(\mathit{fn}(P_1)).T_1\}\,)$$
$$\textit{where } n \textit{ fresh} \quad [\![P_i]\!]_{\widetilde{u}}=(T_i,\Gamma_{bci})$$

$$[\![\text{spawn@}x\{P_1\}P_2]\!]_{\widetilde{u}} = (\,\text{spawn}(n)\langle \mathit{fn}(P_1)\rangle.T_2\ ,\ \Gamma_{bc1}\cup\Gamma_{bc2}\cup\{n\mapsto(\mathit{fn}(P_1)).@x.T_1\}\,)$$
$$\textit{where } n \textit{ fresh} \quad [\![P_1]\!]_{x}=(T_1,\Gamma_{bc1}) \quad [\![P_2]\!]_{\widetilde{u}}=(T_1,\Gamma_{bc2})$$

We remark that named threads in the bytecode come from two sources: (1) there is a named thread for each recursive definition, and also for "main"; (2) there is a named thread for every `spawn` statement. We remark that the encoding of `recv` using linear forwarders is standard from [GLW03]. The encoding of synchronous `send` and `recv` into asynchronous `send` and `recv` is standard from [SW01]. In the encoding $[\![\cdot]\!]_{\widetilde{u}}$ the set of names $\widetilde{u}$ are "certainly–local". These channels can be used as subject of input operations. The `new` command creates a certainly–local name. After a `spawn@x` command only `x` is certainly–local.

Correctness of the Encoding

We now describe some properties of the encoding that are the results of our theoretical work. Our aim is to prove that the encoding preserves behavior, i.e. we want to prove that a hipi core program and its translation into bytecode are behavioral equivalent. Finally we prove that for any pair of barbed bisimilar hipi core programs, they are encoded into a pair of barbed bisimilar bytecode programs. These results ensure the correctness of the encoding. Hence we can implement a compiler based on this encoding.

Now we define a function loc that will be used to check if an input operation in a bytecode process is executed at the correct location. This is a crucial property that must be verified by the encoding in order to ensure that all the input operations will be executed.

Definition 2.22 loc *is a partial function from bytecode bodies into sets of names defined by:*

$$
\begin{aligned}
\mathsf{loc}\ \mathbf{0} &= \varnothing \\
\mathsf{loc}\ \overline{x}\widetilde{y} &= \varnothing \\
\mathsf{loc}\ x \multimap y &= \varnothing \\
\mathsf{loc}\ \mathrm{spawn}(n)\langle\widetilde{x}\rangle.T &= \mathsf{loc}\ T \\
\mathsf{loc}\ x(\widetilde{y}).T &= \begin{cases} \mathsf{loc}\ T \cup \{x\} & \textit{if } \mathsf{loc}\ T \cap\ \widetilde{y} = \varnothing \\ \textit{undefined} & \textit{otherwise} \end{cases} \\
\mathsf{loc}\ @x.T &= \begin{cases} \varnothing & \textit{if } \mathsf{loc}\ T \subseteq \{x\} \\ \textit{undefined} & \textit{otherwise} \end{cases} \\
\mathsf{loc}\ \nu x.T &= \begin{cases} \mathsf{loc}\ T \setminus \{x\} & \textit{if } \mathsf{loc}\ T \textit{ is defined} \\ \textit{undefined} & \textit{otherwise} \end{cases} \\
\mathsf{loc}\ [E]T_1, T_2 &= \begin{cases} \mathsf{loc}\ T_1 \cup \mathsf{loc}\ T_2 & \textit{if } \mathsf{loc}\ T_1 \textit{ and } \mathsf{loc}\ T_2 \textit{ are defined} \\ \textit{undefined} & \textit{otherwise} \end{cases} \\
\mathsf{loc}\ B_1 \mid B_2 &= \begin{cases} \mathsf{loc}\ B_1 \cup \mathsf{loc}\ B_2 & \textit{if } \mathsf{loc}\ B_1 \textit{ and } \mathsf{loc}\ B_2 \textit{ are defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}
\end{aligned}
$$

The loc function maps a bytecode body into the set of names used for input and not guarded by a νx, a $w(\widetilde{x})$ or a $@w$ terms. The loc function is undefined if something wrong occurs in the body of P: in particular, in the case of an input capability and in the case of a migration to x followed by an input on a different channel.

The loc function allows us to define well–formedness of machines.

Definition 2.23 (Well–Formedness of Machines) *A bytecode machine*

$$M = \widetilde{u}\big[\,\widetilde{z} :\, B\,\big], M'$$

is well–formed *if the followings are satisfied:*

1. *$\mathsf{loc}\ B$ is defined and* $\mathsf{loc}\ B \subseteq \widetilde{u}\widetilde{z}$
2. *M' is a well–formed machine*

The empty machine $\mathbf{0}$ *is always well–formed.*

A *well–formed machine* is a machine where each input operation is executed at the location of its subject channel. We give some examples of machines and then we discuss their well–formedness. In each example we write B to refer to the body of the machine.

$$\widetilde{u}\big[\,\widetilde{z} :\, @w.\nu x.x(y)\,\big] \qquad \mathsf{loc}\ B = \varnothing \tag{i}$$

$$\widetilde{u}\big[\,\widetilde{z} :\, \nu x.@w.x(y)\,\big] \qquad \mathsf{loc}\ B = \begin{cases} \varnothing & \textit{if } w = x \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{ii}$$

$$\widetilde{u}\big[\,\widetilde{z} :\, \nu w.x(y)\,\big] \quad w \neq x \qquad \mathsf{loc}\ B = \{x\} \tag{iii}$$

In the examples illustrated above (i) is well–formed, (ii) is well–formed if $w = x$ and (iii) is well–formed if $x \in \widetilde{u}\widetilde{z}$.

We now give a proposition on states (Γ_{bc}, M) that is based on well–formedness of machines M. We assume Γ_{bc} and we represent states as machines. This allow us to use well–formedness of machines as a property for states without define it explicitly.

Proposition 2.24 *Well–Formedness of machines is preserved by reactions*

Proof sketch. We refer to the bytecode reactions of Definition 2.16. The only relevant reactions are (migrate1), (migrate2) and (new).

In (migrate1) we have:

$$\widetilde{u}\big[\,\widetilde{z} :\, @x.T\,\big], \widetilde{w}\big[\,\widetilde{k} :\ \big] \;\;\rightarrow\;\; \widetilde{u}\big[\,\widetilde{z} :\ \big], \widetilde{w}\big[\,\widetilde{k} :\, T\,\big] \quad \text{if } x \in \widetilde{w}\widetilde{k}$$

The well–formedness of the left hand side implies $\mathsf{loc}\ T \subseteq \{x\}$. Since $x \in \widetilde{w}\widetilde{z}$ we have that also the right hand is well–formed.

The case of (migrate2) is a particular case of (migrate2) and the proof is the same.

Finally, the (new) reaction is:

$$\widetilde{u}\big[\,\widetilde{z} :\, \nu x.T\,\big] \;\;\rightarrow\;\; \widetilde{u}\big[\,x'\widetilde{z} :\, T\{^{x'}/x\}\,\big] \quad \text{if } x \notin \widetilde{u}\widetilde{z}$$

If $\mathsf{loc}\ \nu x.T = \mathsf{loc}\ T$ (i.e. if no input operations on x are present in T) the right hand side is trivially well–formed. Otherwise, we have: $\mathsf{loc}\ \nu x.T = \mathsf{loc}\ T \setminus \{x\} \neq \mathsf{loc}\ T$ and $\mathsf{loc}\ T\{^{x'}/x\} = \mathsf{loc}\ T \cup x' \setminus x$ that guarantees the well–formedness of the right hand side. $\square$

The final goal of our theoretical work is to prove that the encoding $[\![\,\cdot\,]\!]$ preserves the behavior of the source program. The proof of this proposition requires a *loader*. A loader is an entity that inserts a bytecode program into the environment where it will be executed. A loader allows us to define how the execution of a bytecode program starts.

Definition 2.25 (Loader) *A* loader *L is a partial mapping from bytecode programs Γ_{bc} into well–formed bytecode states (Γ_{bc}, M) where M has the form:*

$$\widetilde{x}_1\big[\,\varnothing :\, P_1\,\big] \;\;,\;\; \ldots \;\;,\;\; \widetilde{x}_n\big[\,\varnothing :\, P_n\,\big]$$

such that

$$fn(\Gamma_{bc}(\mathtt{main})\langle\rangle) \subseteq \widetilde{x}_1 \ldots \widetilde{x}_n$$

and

$$\exists k \in 1 \ldots n \quad s.t. \quad P_k = \Gamma_{bc}(\mathtt{main})\langle\rangle$$
$$and \quad \forall i \neq k \;\; P_i = \mathbf{0}$$

the degenerate loader L_d *maps every program* Γ_{bc} *into* $(\Gamma_{bc}, fn(P)\big[\,\varnothing : \, P\,\big])$ *where* $P = \Gamma_{bc}(\mathtt{main})\langle\rangle$

The *degenerate loader* L_d represents the centralized execution of the program Γ_{bc}. A single location is created by this loader and the program is executed there.

We now justify the "degenerate" loader. We justify in the sense that, if *any* loader is defined for a bytecode program Γ_{bc}, then the degenerate loader will also be defined for Γ_{bc}.

Proposition 2.26 *Given a bytecode program* Γ_{bc}*, if a loader* L *is defined on* Γ_{bc} *then the degenerate loader* L_d *is also defined on* Γ_{bc}*.*

Proof. Without loss of generality, we assume that

$$L(\Gamma_{bc}) = \widetilde{x}_1\big[\,\varnothing : \, P\,\big] \;\; , \;\; \widetilde{x}_2\big[\,\varnothing : \, \mathbf{0}\,\big] \;\; , \;\; \ldots \;\; , \;\; \widetilde{x}_n\big[\,\varnothing : \, \mathbf{0}\,\big]$$

where $P = \Gamma_{bc}(\mathtt{main})\langle\rangle$ with $\mathsf{loc}\ P \subseteq \widetilde{x}_1$. Hence, $\mathsf{loc}\ P \subseteq \{\widetilde{x}_1, \ldots, \widetilde{x}_n\}$ and the degenerate loader L_d is defined:

$$L_d(\Gamma_{bc}) = \widetilde{x}_1, \ldots, \widetilde{x}_n\big[\,\varnothing : \, P\,\big]$$

and this concludes the proof. □

Now we give an example that shows that L_d defined doesn't implies that every possible loader is also defined. Consider the following bytecode program:

$$\Gamma_{bc} = \big\{\big(\,\mathtt{main}\ , \ ().x().y()\ \big)\big\}$$

If the degenerate loader is defined on Γ_{bc} we have:

$$L_d(\Gamma_{bc}) = xy\big[\,\varnothing : \, x().y()\,\big]$$

A possible different loader L is:

$$L(\Gamma_{bc}) = x\big[\,\varnothing : \, x().y()\,\big], y\big[\,\varnothing : \;\,\big]$$

But $L(\Gamma_{bc})$ is not well–formed because $\mathsf{loc}\ x().y()$ is $\{x, y\}$ and $\{x, y\} \not\subseteq \{x\}$.

Given a program Γ_{bc}, the degenerate loader is a good representative for the set of defined loaders of Γ_{bc}. Our aim, now, is to prove that all loaded states are equivalent in behavior to the degenerate loader.

Lemma 2.27

$$\widetilde{u}\widetilde{w}\big[\,\widetilde{z} : \, P\,\big], M \quad \dot{\approx} \quad \widetilde{u}\big[\,\widetilde{z} : \, P\,\big], \widetilde{w}\big[\,\varnothing : \;\,\big], M$$

if $\widetilde{u}\widetilde{w} \cap dc(M) = \varnothing$ *and where both sides are assumed to be well–formed.*

Proof. Consider the relation $\mathcal{S}$ on machines defined by:

$$\mathcal{S} = \big\{\big(\,M_1\ ,\ M_2\,\big) \;\;\big|\;\; M_1 = \widetilde{u}\widetilde{w}\big[\,\widetilde{z} : \, P\,\big], M \quad M_2 = \widetilde{u}\big[\,\widetilde{z} : \, P\,\big], \widetilde{w}\big[\,\varnothing : \;\,\big], M$$
$$\text{where } M_1, M_2 \text{ well–formed} \quad \forall \widetilde{u}\widetilde{w}\widetilde{z}PM\big\}$$

To prove that $\mathcal{S}$ is a bisimulation we must verify the followings:

1. $M_1 \downarrow \alpha$ implies $M_2 \Downarrow \alpha$
2. $M_1 \rightarrow M_1'$ implies $\exists M_2'$ such that $M_2 \Rightarrow M_2'$ and $M_1' \mathcal{S} M_2'$
3. $M_2 \downarrow \alpha$ implies $M_1 \Downarrow \alpha$
4. $M_2 \rightarrow M_2'$ implies $\exists M_1'$ such that $M_1 \Rightarrow M_1'$ and $M_1' \mathcal{S} M_2'$

Point (1) is due to the well–formedness of M_2, as follows. In the case of $\alpha = \overline{x}$ the proof is trivial. In the case of $\alpha = x$ the observed input operation on x must be colocated with x in M_2 in order to be observed also there. This is guaranteed by the well–formedness of M_2.

In point (2) the only problematic case is when the transition between M_1 and M_1' is (react). For instance, if we lacked the well–formedness requirement in $\mathcal{S}$, then $N_1 \mathcal{S} N_2$ where:

$$N_1 = xy[\varnothing : \overline{y} \mid y()] \quad \text{and} \quad N_2 = x[\varnothing : \overline{y} \mid y()], y[\varnothing : \;]$$

would be valid. But $N_1 \xrightarrow{(react)} N_1'$ and $N_2 \xrightarrow{(move-out)} N_2'$ with $(N_1', N_2') \notin \mathcal{S}$. The well–formedness requirement in $\mathcal{S}$ guarantees that if $M_1 \xrightarrow{(react)} M_1'$ then $M_2 \xrightarrow{(react)} M_2'$.

The proofs of points 3. and 4. are trivial inductions. □

This completes our justification for considering the degenerate loader L_d as a representative loader.

Now, in order to prove that $[\![\cdot]\!]$ preserves bisimilarity, we define an extended encoding $\{\![\cdot]\!\}_{\widetilde{u}}$ from hipi core states into bytecode states based on the subsidiary translation $[\![\cdot]\!]_{\widetilde{u}}$ (Definition 2.21). For the sake of simplicity and through abuse of notation we write $[\![P]\!]_{\widetilde{u}}$ for (`main`,T_M) where $[\![P]\!]_{\widetilde{u}} = (T_M, \Gamma_M)$. This work–around is necessary because the loader requires a complete bytecode program instead of a fragment (that is the output of the subsidiary encoding).

Definition 2.28 (Extended Encoding) *The* extended encoding $\{\![\cdot]\!\}_{\widetilde{u}}$ *maps hipi core states into bytecode states as follows. In* $\{\![\cdot]\!\}_{\widetilde{u}}$*, the names* $\widetilde{u}$ *are* certainly–local *, i.e. the context in which* $\{\![P]\!\}_{\widetilde{u}}$ *occurs will ensure that the term is executed co–located with* $\widetilde{u}$*:*

$$\{\![P]\!\}_{\widetilde{u}} \;=\; L_d([\![P]\!]_{\widetilde{u}})$$

$$\{\![(x)S]\!\}_{\widetilde{u}} \;=\; A \setminus x[Bx : T_{bc}] \qquad \text{where } A[B : T_{bc}] = \{\![S]\!\}_{\widetilde{u}}$$

$$\{\![S_1 \mid S_2]\!\}_{\widetilde{u}} \;=\; A_1A_2[B_1B_2 : T_{bc1} \mid T_{bc2}] \qquad \text{where } A_i[B_i : T_{bci}] = \{\![T_i]\!\}_{\widetilde{u}}$$

We remark that the extended encoding $\{\![\cdot]\!\}_{\widetilde{u}}$ is a partial mapping because L_d is also partial. For the sake of simplicity, in what follows we consider only hipi core programs P such that the encoding $\{\![P]\!\}_{\widetilde{u}}$ is defined.

The following lemma states a trivial property of the extended encoding that we will use in proofs. It states that substitution of names can be done either before or after the encoding, with the same result.

Lemma 2.29 *Given a hipi core state S, for every $\widetilde{u}$:*

$$\{\![S]\!\}_{\widetilde{u}}\{^{x'}/x\} = \{\![S\{^{x'}/x\}]\!\}_{\widetilde{u}}$$

We give also a lemma that states a property of linear forwarders. The proof of the lemma is trivial and partially will emerge inside the proof of Theorem 2.31.

Lemma 2.30 *If $\widetilde{x} \subseteq \widetilde{u}$*

$$\widetilde{u}[\,\varnothing : \llbracket P \rrbracket_{\varnothing} \mid T\,], M \quad \dot{\approx} \quad \widetilde{u}[\,\varnothing : \llbracket P \rrbracket_{\widetilde{x}} \mid T\,], M$$

We now give the theorem that guarantees the correctness of the encoding $\llbracket \cdot \rrbracket$. It is obtained through the correctness of the encoding $\{\!\llbracket \cdot \rrbracket\!\}_{\widetilde{u}}$: this is possible because a hipi core program is a particular case of hipi core state and in this case $\{\!\llbracket P \rrbracket\!\}_{\varnothing}$ represents the execution of $\llbracket P \rrbracket$.

This is the main theorem of our theoretical work: it ensures that the encoding preserves the behavior of the source hipi core program.

Theorem 2.31 (Correctness) *Given a hipi core state S, for every $\widetilde{u}$ the following are satisfied:*

1. *$S \downarrow \alpha$ implies $\{\!\llbracket S \rrbracket\!\}_{\widetilde{u}} \Downarrow \alpha$*
2. *$S \to S'$ implies $\exists M'$ such that $\{\!\llbracket S \rrbracket\!\}_{\widetilde{u}} \Rightarrow M'$ and $\{\!\llbracket S' \rrbracket\!\}_{\widetilde{u}} \equiv M'$*
3. *$\{\!\llbracket S \rrbracket\!\}_{\widetilde{u}} \downarrow \alpha$ implies $S \Downarrow \alpha$*
4. *$\{\!\llbracket S \rrbracket\!\}_{\widetilde{u}} \to M'$ implies $\exists S'$ such that $S \Rightarrow S'$ and $\{\!\llbracket S' \rrbracket\!\}_{\widetilde{u}} \dot{\approx} M'$*

Proof. Parts 1. and 3. are trivial inductions on the structure of S. We prove part 2. by induction on the derivation of $S \to S'$ as defined in Definition 2.16 . The most relevant cases are (react),(spawn),(migrate) and (call) all closed with respect to structural congruence, restriction and parallel composition.

In the (react) case we have:

$$\equiv (\widetilde{z})(P \mid x.\mathrm{send}(\widetilde{E});P_1 \mid x.\mathrm{recv}(\widetilde{Ty});P_2) \quad \to \equiv (\widetilde{z})(P \mid P_1 \mid P_2\{\widetilde{v}/\widetilde{y}\})$$

where $\widetilde{v}$ is the evaluation of $\widetilde{E}$. For the sake of clarity, in what follows we omit $\equiv$, $(\widetilde{x})$ and P. We consider first the case $x \in \widetilde{u}$:

$$\begin{aligned}
& \{\!\llbracket x.\mathrm{send}(\widetilde{E});P_1 \mid x.\mathrm{recv}(\widetilde{Ty});P_2 \rrbracket\!\}_{\widetilde{u}} \\
= \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,\varnothing : \llbracket x.\mathrm{send}(\widetilde{E});P_1 \mid x.\mathrm{recv}(\widetilde{Ty});P_2 \rrbracket_{\widetilde{u}}\,] \\
= \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,\varnothing : \nu z.\mathrm{spawn}(n)\langle z\rangle.z().\llbracket P_1 \rrbracket_{\widetilde{u}} \mid x(z,\widetilde{y}).\mathrm{spawn}(m)\langle z\rangle.\llbracket P_2 \rrbracket_{\widetilde{u}}\,] \\
\to \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,z : \mathrm{spawn}(n)\langle z\rangle.z().\llbracket P_1 \rrbracket_{\widetilde{u}} \mid x(z,\widetilde{y}).\mathrm{spawn}(m)\langle z\rangle.\llbracket P_2 \rrbracket_{\widetilde{u}}\,] \\
\to \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,z : \overline{x}z,\widetilde{E} \mid z().\llbracket P_1 \rrbracket_{\widetilde{u}} \mid x(z,\widetilde{y}).\mathrm{spawn}(m)\langle z\rangle.\llbracket P_2 \rrbracket_{\widetilde{u}}\,] \\
\to \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,z : z().\llbracket P_1 \rrbracket_{\widetilde{u}} \mid (\mathrm{spawn}(m)\langle z\rangle.\llbracket P_2 \rrbracket_{\widetilde{u}})\{\widetilde{v}/\widetilde{y}\}\,] \\
\to \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,z : z().\llbracket P_1 \rrbracket_{\widetilde{u}} \mid (\overline{z} \mid \llbracket P_2 \rrbracket_{\widetilde{u}})\{\widetilde{v}/\widetilde{y}\}\,] \\
\to \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,z : \llbracket P_1 \rrbracket_{\widetilde{u}} \mid \llbracket P_2 \rrbracket_{\widetilde{u}}\{\widetilde{v}/\widetilde{y}\}\,] \\
\equiv \;& \mathit{fn}(P_1 \mid P_2)[\,\varnothing : \llbracket P_1 \rrbracket_{\widetilde{u}} \mid \llbracket P_2 \rrbracket_{\widetilde{u}}\{\widetilde{v}/\widetilde{y}\}\,] \\
= \;& \{\!\llbracket P_1 \mid P_2\{\widetilde{v}/\widetilde{y}\} \rrbracket\!\}_{\widetilde{u}}
\end{aligned}$$

In the last two lines structural equivalence $\equiv$ is used to remove $\cup x$ and z from the machine channels and Lemma 2.29 to obtain the final result.

In the case $x \notin \widetilde{u}$ we have:

$$
\begin{aligned}
& \{\!\![x.\mathrm{send}(\widetilde{E});P_1 \mid x.\mathrm{recv}(\widetilde{Ty});P_2]\!\!\}_{\widetilde{u}} \\
= \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,\varnothing : [\![x.\mathrm{send}(\widetilde{E});P_1 \mid x.\mathrm{recv}(\widetilde{Ty});P_2]\!]_{\widetilde{u}}\,] \\
= \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,\varnothing : \nu z.\mathrm{spawn}(n)\langle z\rangle.z().[\![P_1]\!]_{\widetilde{u}} \\
& \qquad\qquad \mid \nu w.\mathrm{spawn}(o)\langle xw\rangle.w(z,\widetilde{y}).\mathrm{spawn}(m)\langle z\rangle.[\![P_2]\!]_{\widetilde{u}}\,] \\
\rightarrow^4 \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,zw : \overline{x}z\widetilde{E} \mid z().[\![P_1]\!]_{\widetilde{u}} \\
& \qquad\qquad \mid x \multimap w \mid w(z,\widetilde{y}).\mathrm{spawn}(m)\langle z\rangle.[\![P_2]\!]_{\widetilde{u}}\,] \\
\rightarrow \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,zw : \overline{w}z\widetilde{E} \mid z().[\![P_1]\!]_{\widetilde{u}} \mid w(z,\widetilde{y}).\mathrm{spawn}(m)\langle z\rangle.[\![P_2]\!]_{\widetilde{u}}\,] \\
\rightarrow \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,zw : z().[\![P_1]\!]_{\widetilde{u}} \mid (\mathrm{spawn}(m)\langle z\rangle.[\![P_2]\!]_{\widetilde{u}})\{\widetilde{v}/\widetilde{y}\}\,] \\
\rightarrow \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,zw : z().[\![P_1]\!]_{\widetilde{u}} \mid (\overline{z} \mid [\![P_2]\!]_{\widetilde{u}})\{\widetilde{v}/\widetilde{y}\}\,] \\
\rightarrow \;& \mathit{fn}(P_1 \mid P_2) \cup x[\,zw : [\![P_1]\!]_{\widetilde{u}} \mid [\![P_2]\!]_{\widetilde{u}}\{\widetilde{v}/\widetilde{y}\}\,] \\
\equiv \;& \mathit{fn}(P_1 \mid P_2)[\,\varnothing : [\![P_1]\!]_{\widetilde{u}} \mid [\![P_2]\!]_{\widetilde{u}}\{\widetilde{v}/\widetilde{y}\}\,] \\
= \;& \{\!\![P_1 \mid P_2\{\widetilde{v}/\widetilde{y}\}]\!\!\}_{\widetilde{u}}
\end{aligned}
$$

It is important to note that in the two cases just described ($x \in \widetilde{u}$ and $x \notin \widetilde{u}$) the sequences of reactions are different, but the general behavior is the same. This equivalence partially proves also Lemma 2.30.

The cases of (spawn) and (migrate) are identical apart from a $@x$, but $@x$ has not influence on the behavior of a single location machine. We consider only the former:

$$\equiv (\widetilde{z})(P \mid \mathrm{spawn}\{P_1\}P_2) \;\; \rightarrow \; \equiv (\widetilde{z})(P \mid P_1 \mid P_2)$$

Applying the encoding we have $\Gamma_{bc}(n) = (\mathit{fn}(P_1)).[\![P_1]\!]_{\widetilde{u}}$ with the following behavior. For the sake of clarity, we omit $\equiv$, $(\widetilde{z})$ and P:

$$
\begin{aligned}
& \{\!\![\mathrm{spawn}\{P_1\}P_2]\!\!\}_{\widetilde{u}} \\
= \;& \mathit{fn}(P_1 \mid P_2)[\,\varnothing : \mathrm{spawn}(n)\langle \mathit{fn}(P_1)\rangle.[\![P_2]\!]_{\widetilde{u}}\,] \\
\rightarrow \;& \mathit{fn}(P_1 \mid P_2)[\,\varnothing : [\![P_1]\!]_{\widetilde{u}} \mid [\![P_2]\!]_{\widetilde{u}}\,] \\
= \;& \{\!\![P_1 \mid P_2]\!\!\}_{\widetilde{u}}
\end{aligned}
$$

The last case is (call). We assume $\Gamma(D) = (\widetilde{Tx}).P_1$ The rule in the hipi core semantics is:

$$\equiv (\widetilde{z})(P \mid D(\widetilde{E});P_2) \;\; \rightarrow \; \equiv (\widetilde{z})(P \mid \Gamma(D)\langle\widetilde{v}\rangle \mid P_2)$$

Applying the encoding we have $\Gamma_{bc}(n_D) = (\widetilde{x}).[\![P_1]\!]_{\varnothing}$ and we obtain the following behavior. As usual, we omit $\equiv$, $(\widetilde{z})$ and P:

$$
\begin{aligned}
& & \{\!\![D(\widetilde{E});P_2]\!\!\}_{\widetilde{u}} \\
& = \;& \mathit{fn}(P_2)[\,\varnothing : [\![D(\widetilde{E});P_2]\!]_{\widetilde{u}}\,] \\
& = \;& \mathit{fn}(P_2)[\,\varnothing : \mathrm{spawn}(n_D)\langle\widetilde{E}\rangle.[\![P_2]\!]_{\widetilde{u}}\,] \\
& \rightarrow \;& \mathit{fn}(P_2)[\,\varnothing : [\![P_1]\!]_{\varnothing}\{\widetilde{E}/\widetilde{x}\} \mid [\![P_2]\!]_{\widetilde{u}}\,] \\
(1) \qquad & = \;& \mathit{fn}(P_2)[\,\varnothing : [\![P_1\{\widetilde{E}/\widetilde{x}\}]\!]_{\varnothing} \mid [\![P_2]\!]_{\widetilde{u}}\,] \\
(2) \qquad & \dot{\approx} \;& \mathit{fn}(P_2)[\,\varnothing : [\![P_1\{\widetilde{E}/\widetilde{x}\}]\!]_{\widetilde{u}} \mid [\![P_2]\!]_{\widetilde{u}}\,] \\
(3) \qquad & = \;& \{\!\![P_1\{\widetilde{E}/\widetilde{x}\} \mid P_2]\!\!\}_{\widetilde{u}}
\end{aligned}
$$

The last three lines are due to: (1) Lemma 2.29, (2) Lemma 2.30 and the well–formedness of hipi core programs (Definition 2.4) that ensures that the free names of $P_1\{\widetilde{E}/\widetilde{x}\}$ are a subset of the free names of P_2, (3) the definition of $\{\![\,\cdot\,]\!\}_{\widetilde{u}}$.

We now prove part 4. by induction on the derivation of $\{\![S]\!\}_{\widetilde{u}} \to M'$ (Definition 2.16). The only possible reactions for $\{\![S]\!\}_{\widetilde{u}}$ are (spawn), (new), (if1) and (if2). In the (spawn) case, the reaction of $\{\![S]\!\}_{\widetilde{u}}$ then S must have one of the following forms:

$$D(\widetilde{E}); P \qquad \text{spawn}\{P_1\}P_2 \qquad \text{spawn@}x\{P_1\}P_2$$

In all the three cases $S \to S'$ such that $\{\![S']\!\}_{\widetilde{u}} \dot{\approx} M'$ trivially.

In the case of (new) then S must have one of the following forms:

$$\text{new}Tx \text{ in } P \qquad x.\text{send}(\widetilde{E});P \qquad x.\text{recv}(\widetilde{Ty});P$$

In all these three cases $S' = S$ such that $\{\![S]\!\}_{\widetilde{u}} \dot{\approx} M'$ trivially.

Finally, in both the cases of (if1) and (if2) S must have the form:

$$\text{if}(E)\{P_1\}\{P_2\}$$

that executes a reaction to S' where either $S' = P_1$ and $\{\![S']\!\}_{\widetilde{u}} = M'$ or $S' = P_2$ or $\{\![S']\!\}_{\widetilde{u}} = M'$ in accordance to the evaluation of E. □

Now, we give our final result in the following proposition.

Proposition 2.32 *Given the hipi core states S_1 and S_2 then*

$$S_1 \dot{\approx} S_2 \textit{ if and only if } \{\![S_1]\!\}_{\widetilde{u}} \dot{\approx} \{\![S_2]\!\}_{\widetilde{u}}$$

for any subscript $\widetilde{u}$

Proof. ($\Rightarrow$) Consider the relation $\mathcal{S}$ on machines defined by:

$$\mathcal{S} = \{\ (M_1, M_2)\ \mid\ M_1 \dot{\approx} \{\![S_1]\!\}_{\widetilde{u}}\ ,\ M_2 \dot{\approx} \{\![S_2]\!\}_{\widetilde{u}}\ ,\ S_1 \dot{\approx} S_2 \quad \text{for all } \widetilde{u}\}$$

To prove that $\mathcal{S}$ is a bisimulation we must verify the followings:

1. $M_1 \downarrow \alpha$ implies $M_2 \Downarrow \alpha$
2. $M_1 \to M_1'$ implies $\exists M_2'$ such that $M_2 \Rightarrow M_2'$ and $M_1'\ \mathcal{S}\ M_2'$
3. $M_2 \downarrow \alpha$ implies $M_1 \Downarrow \alpha$
4. $M_2 \to M_2'$ implies $\exists M_1'$ such that $M_1 \Rightarrow M_1'$ and $M_1'\ \mathcal{S}\ M_2'$

The proof of parts 1. and 3. is trivial due to Theorem 2.31 (Correctness) and because $S_1 \dot{\approx} S_2$, $M_1 \dot{\approx} \{\![S_1]\!\}_{\widetilde{u}}$ and $M_2 \dot{\approx} \{\![S_2]\!\}_{\widetilde{u}}$.

The following commutative diagram establishes part (2):

$$\begin{array}{cclll}
M_1 & \longrightarrow & M_1' & & \\
\dot{\approx} & & \dot{\approx} & & \text{by definitions of } \mathcal{S} \text{ and } \dot{\approx} \\
\{\![S_1]\!\}_{\widetilde{u}} & \Longrightarrow & B_1' & & \\
 & & & & \text{by step (4) of Theorem 2.31} \\
S_1 & \Longrightarrow & S_1' & s.t.\ B_1' \dot{\approx} \{\![S_1']\!\}_{\widetilde{u}} & \\
\dot{\approx} & & \dot{\approx} & & \text{by definitions of } \mathcal{S} \text{ and } \dot{\approx} \\
S_2 & \Longrightarrow & S_2' & & \\
 & & & & \text{by step (2) of Theorem 2.31} \\
\{\![S_2]\!\}_{\widetilde{u}} & \Longrightarrow & B_2' & s.t.\ B_2' \equiv \{\![S_2']\!\}_{\widetilde{u}} & \\
\dot{\approx} & & \dot{\approx} & & \text{by definitions of } \mathcal{S} \text{ and } \dot{\approx} \\
M_2 & \Longrightarrow & M_2' & &
\end{array}$$

The proof of part 4 is the same as that of part 2 because $\mathcal{S}$ is symmetric. We conclude that if $S_1 \stackrel{\cdot}{\approx} S_2$ then $\{\![S_1]\!\}_{\widetilde{u}}\ \mathcal{S}\ \{\![S_2]\!\}_{\widetilde{u}}$ by definition. Hence, $\{\![S_1]\!\}_{\widetilde{u}} \stackrel{\cdot}{\approx} \{\![S_2]\!\}_{\widetilde{u}}$

($\Leftarrow$) Consider the relation $\mathcal{S}$ on hipi core states defined by:

$$\mathcal{S} = \{\ (S_1, S_2) \quad | \quad \{\![S_1]\!\}_{\widetilde{u}} \stackrel{\cdot}{\approx} \{\![S_2]\!\}_{\widetilde{u}} \quad \text{for all } \widetilde{u}\}$$

To prove that $\mathcal{S}$ is a bisimulation we must verify the followings:

1. $\{\![S_1]\!\}_{\widetilde{u}} \downarrow \alpha$ implies $\{\![S_2]\!\}_{\widetilde{u}} \Downarrow \alpha$
2. $\{\![S_1]\!\}_{\widetilde{u}} \rightarrow M_1'$ implies $\exists M_2'$ such that $\{\![S_2]\!\}_{\widetilde{u}} \Rightarrow M_2'$ and $M_1'\ \mathcal{S}\ M_2'$
3. $\{\![S_2]\!\}_{\widetilde{u}} \downarrow \alpha$ implies $\{\![S_1]\!\}_{\widetilde{u}} \Downarrow \alpha$
4. $\{\![S_2]\!\}_{\widetilde{u}} \rightarrow M_2'$ implies $\exists M_1'$ such that $\{\![S_1]\!\}_{\widetilde{u}} \Rightarrow M_1'$ and $M_1'\ \mathcal{S}\ M_2'$

The proofs of parts (1) and (3) are trivial. We prove part (2) by giving the following diagram.

$$\begin{array}{ccccl}
S_1 & \longrightarrow & S_1' & & \\
 & & & & \text{by step (2) of Theorem 2.31} \\
\{\![S_1]\!\}_{\widetilde{u}} & \Longrightarrow & M_1' & \equiv \{\![S_1']\!\} & \\
\stackrel{\cdot}{\approx} & & \stackrel{\cdot}{\approx} & & \text{by definitions of } \mathcal{S} \text{ and } \stackrel{\cdot}{\approx} \\
\{\![S_2]\!\}_{\widetilde{u}} & \Longrightarrow & M_2' & & \\
 & & & & \text{by step (2) of Theorem 2.31} \\
\{\![S_2]\!\}_{\widetilde{u}} & \Longrightarrow & S_2' & s.t.\ M_2' \equiv \{\![S_2']\!\}_{\widetilde{u}} &
\end{array}$$

Part 4 is due to the symmetry of $\mathcal{S}$. We conclude that if $\{\![S_1]\!\}_{\widetilde{u}} \stackrel{\cdot}{\approx} \{\![S_2]\!\}_{\widetilde{u}}$ then $S_1\ \mathcal{S}\ S_2$. Hence, $S_1 \stackrel{\cdot}{\approx} S_2$. $\square$

Chapter 3

High Level Languages and Hipi Compiler

The aim of this chapter is to describe *hipi*, the *XML Intermediate Language* (XIL) and the implementation of the *hipi Compiler* (the “hipi core” and the “bytecode” of the previous chapter were idealizations/simplifications of hipi and XIL used to make proof clearer). We also describe the implementation of a *Network Simulator* that can be used to execute compiled programs.

3.1 Hipi: Language Definition

Hipi is a programmer–friendly extension of the hipi core language described in section 2.1. The main differences with the core language that hipi has: a different program structure, C–like functions instead of recursive definitions, for–loops, URIs and `new` as expressions for binding of channel names. These differences are all syntactic sugar.

A hipi program contains a set of *schedules*. A schedule is the equivalent of a full program in the core language: it has a “main” and some local functions. Schedules are named, and their names are used by the compiler to choose filenames for its output. *Functions* can be either local to a schedule, or global: the former are placed inside a schedule and their scope is that schedule, the latter are placed outside any schedule and their scope is the whole program. A *URI* is used to refer to a global (i.e. well–known and pre–existing) channel.

In this section we describe hipi. The semantics of hipi may be understood either informally through the descriptions in this section, or formally through syntactic sugar translation into the hipi core language (following section).

Structure of a Program

We start with some programming examples. As in every reference manual of any programming language, the first example is a program that prints the *Hello World* string on the standard output.

```
schedule HelloWorld {
    main {
        // global channel constant for console interaction
        channel<string> console = "ch://string.console";
        // prints Hello World
        console.send("Hello World");
    }
```

```
}
```

The example partially illustrates the structure of a hipi program: there is a *schedule*, namely `HelloWorld`, that is a sort of container. Inside the schedule there is a *main* that is a block containing the statements executed at run–time. Finally, inside the main there are two statements: the first declares that the constant name "console" refers to a global channel and the second sends a string on it: the global channel is identified using a URI and it is provided by the run–time environment. The HelloWorld program contains also two lines of comment: in hipi they are as in C++ where `//` is used for single lines and `/*` $\cdots$ `*/` for multiple lines.

The Complete Language Grammar

An hipi program is made of four parts: (1) a set of type declarations, (2) imports of external source files, (3) function declarations and (4) schedules. Statements are defined for communications, control–flow commands, function calls and the definition of new variables. Expressions are the same as the core language extended with `new channel` and URI that are used in substitution of the `new` statement and to avoid free names.

$$\begin{aligned}
\mathit{Program} &::= \mathit{TypeDecl}* \;\; \mathit{FileImport}* \;\; \mathit{Function}* \;\; \mathit{Schedule}*
\end{aligned}$$

$$\begin{aligned}
\mathit{TypeDecl} &::= \texttt{typedef}\ \mathit{typeid} = \mathit{Type}\ ; \\
\mathit{Type} &::= \texttt{channel} < \big[\ \mathit{TypeList}\ \big] > \\
&\quad \mid \texttt{int} \mid \texttt{string} \mid \mathit{typeid} \\
\mathit{TypeList} &::= \mathit{Type}\ \big(\ ,\ \mathit{Type}\ \big)*
\end{aligned}$$

$$\begin{aligned}
\mathit{FileImport} &::= \texttt{import}\ \mathit{filename}\ ;
\end{aligned}$$

$$\begin{aligned}
\mathit{Function} &::= \big(\ \mathit{Type} \mid \texttt{void}\ \big)\ \mathit{funName}\ (\ \mathit{ParamList}\ )\ \{\ \mathit{Statement}\ \} \\
\mathit{ParamList} &::= \mathit{Type}\ \mathit{var}\ \big(\ ,\ \mathit{Type}\ \mathit{var}\ \big)*
\end{aligned}$$

$$\begin{aligned}
\mathit{Schedule} &::= \texttt{schedule}\ \mathit{schedname}\ \big[\ \texttt{colocatedwith}\ \mathit{URIList}\ \big] \\
&\qquad \{\ \big(\ \mathit{Declaration} \mid \mathit{Function}\ \big)* \\
&\qquad \texttt{main}\ \{\ \mathit{Statement}\ \} \\
&\qquad \big(\ \mathit{Declaration} \mid \mathit{Function}\ \big)*\ \} \\
\mathit{URIList} &::= \mathit{uri}\ \big(\ ,\ \mathit{uri}\ \big)* \\
\mathit{Declaration} &::= \mathit{Type}\ \mathit{var} = \mathit{Expression}\ ;
\end{aligned}$$

$$\begin{aligned}
\mathit{Statement} &::= \mathit{Type}\ \mathit{var} = \mathit{Expression}\ ; \\
&\quad \mid \texttt{spawn}\ \{\ \mathit{Statement}\ \} \\
&\quad \mid \texttt{spawn@}\ \mathit{var}\ \{\ \mathit{Statement}\ \} \\
&\quad \mid \mathit{var}\ .\ \texttt{send}\ (\ \big[\ \mathit{ExpressionList}\ \big]\ )\ ; \\
&\quad \mid \mathit{var}\ .\ \texttt{asend}\ (\ \big[\ \mathit{ExpressionList}\ \big]\ )\ ; \\
&\quad \mid \mathit{var}\ .\ \texttt{recv}\ (\ \mathit{ParamList}\ )\ ; \\
&\quad \mid \mathit{funName}\ (\ \big[\ \mathit{ExpressionList}\ \big]\ ); \\
&\quad \mid \texttt{return}\ \big[\ \mathit{Expression}\ \big]\ ; \\
&\quad \mid \texttt{if}\ (\ \mathit{Expression}\ )\ \mathit{Statement}\ \big[\ \texttt{else}\ \mathit{Statement}\ \big]
\end{aligned}$$

$$
\begin{array}{rl}
 & \mid \ \texttt{for} \ var = Expression \ \texttt{to} \ Expression \\
 & \quad \big[\ \texttt{by} \ Expression \ \big] \ Statement \\
 & \mid \ Statement \ \ Statement \\
 & \mid \ \{ \ Statement \ \}
\end{array}
$$

$$
\begin{array}{rl}
Expression & ::= LiteralInt \mid LiteralString \mid var \\
 & \mid \ Expression \ \ op \ \ Expression \\
 & \mid \ - \ Expression \mid \ ! \ Expression \\
 & \mid \ funName \ (\ \big[\ ExpressionList \ \big] \ ) \\
 & \mid \ uri \mid \texttt{new} \ Type \\
 & \mid \ (\ Expression \ ) \\
op & ::= + \mid - \mid * \mid / \mid \% \mid \&\& \mid || \mid \\
 & \mid \ < \mid > \mid <= \mid >= \mid == \mid != \\
ExpressionList & ::= Expression \ \big(\ , \ Expression \ \big)*
\end{array}
$$

Types and Type Declarations

Three types are defined in hipi: *integers*, *strings* and *channels*. Integers and strings are standard. Channel types are identified by the keyword `channel<...>` where a list of types representing the objects of communications is written between angles. A variable with type `channel<string>`, for instance, can be used for communications using only strings as objects. Channel types are the extensions of *sorts* in π–calculus [Mil99] with integers, strings and channels as possible objects of communications.

The grammar rule describing *types* (or *type expressions*) is:

$$
\begin{array}{rl}
Type & ::= \texttt{channel} \ < \ \big[\ TypeList \ \big] \ > \\
 & \quad \mid \ \texttt{int} \ \mid \ \texttt{string} \ \mid \ typeid
\end{array}
$$

where *TypeList* is a (non empty) comma–separated list of *Type* elements and *typeid* is a type identifier. A type identifier is a variable that can be instanced with a type expression. This is obtained through a *type declaration* with the following syntax:

$$
TypeDecl \ ::= \texttt{typedef} \ \ typeid \ = \ Type \ \ ;
$$

As discussed above, type declarations are placed at the beginning of a program and they allow recursion. The simplest example of recursive declaration is:

```
typedef recT = channel<recT>;
```

that is the type of a channel that can be used as the object of a communication on itself.

In a type declaration the right hand side can be a type identifier (i.e. aliases are admitted) but the definition of self–referent or mutually–referent identifiers is forbidden. Examples of wrong declarations are:

```
typedef wrongT = wrongT;
typedef wrongA = wrongB;
typedef wrongB = wrongA;
```

We use *structural equivalence* for types, i.e. types are considered the same if a the fixed point of the sequences their expansions are equivalent. We give an example:

```
typedef alphaT = channel<alphaT>;
typedef betaT = channel<channel<alphaT>;
```

This kind of equivalence relation for types allows to assign values of type `alphaT` to variables of type `betaT` and vice–versa. Structural equivalence was chosen instead of name equivalence (where types with different names are always considered different) because as a future work we want to add XML data types from XDuce [HP03], which themselves use structural equivalence.

Import Declarations

An import declaration is used to include the source code contained in a specified file. The syntax of an import declaration is the following:

$$\textit{FileImport} \ ::= \texttt{import} \ \textit{filename} \ ;$$

The import of a file copies the source code contained inside it into the current program: type definitions, functions and schedules of the imported file are written in the correct place. The names of the imported code must not clash with the names of the current program. If the imported file contains other import declarations, they are executed only if referred to different files (i.e. if they are not the main file or another already imported file).

Schedules

The syntax of schedules is given by the following rules:

$$\begin{aligned}
\textit{Schedule} \ ::=& \ \texttt{schedule} \ \textit{schedname} \ \big[\ \texttt{colocatedwith} \ \textit{URIList} \ \big] \\
& \{ \ \big(\ \textit{Declaration} \ \mid \ \textit{Function} \ \big)* \\
& \texttt{main} \ \{ \ \textit{Statement} \ \} \\
& \big(\ \textit{Declaration} \ \mid \ \textit{Function} \ \big)* \ \} \\
\textit{Declaration} \ ::=& \ \textit{Type} \ \ \textit{var} \ = \ \textit{Expression} \ ;
\end{aligned}$$

where *URIList* is a comma–separated list of URIs, *Function* is a local function and *Statement* is an instruction. A *Declaration* is used to declare an immutable variable that can be used in the whole schedule (i.e. in `main` and inside function bodies). The expression assigned to the declared variable is evaluated once, before main starts. It is also possible to define a constraint on the location where the schedule will be executed: this is achieved by using the `colocatedwith` option. This option is followed by a list of URIs that are references to channels. The use of the `colocatedwith` option guarantees that the schedule will be executed at the location where the listed channels reside (and implicitly requires that all these channels be colocated).

The main motivation for schedules comes from future works. Our expectation is to define shared channels that can be used by multiple schedules to coordinate their work but that are invisible to the rest of the world.

Functions

Functions can be defined either outside or inside schedules. In the first case their scope is the whole program; in the second it is the schedule where they are contained.

The syntax of a function definition is similar to C++:

$$\begin{aligned}
\textit{Function} \ ::=& \ \big(\ \textit{Type} \ \big| \ \texttt{void} \ \big) \ \textit{funName} \ (\ \textit{ParamList} \ ) \\
& \{ \ \textit{Statement} \ \} \\
\textit{ParamList} \ ::=& \ \textit{Type} \ \ \textit{var} \ \big(\ , \ \textit{Type} \ \ \textit{var} \ \big)*
\end{aligned}$$

As usual the scope of the variables declared in *ParamList* is the body of the function.

The distinction between global and local scope requires a discipline on function names: the same name can't be used for two different local functions of the same schedule or for two different global functions. If a global function and a local one exist with the same name, invocations inside the schedule refer to the local one.

Inside a function there can be also `return` statements: as usual, the returned value must be of the same type of the function, or it must be omitted if the type of the function is `void`. Return statements cannot be used inside the main block.

Statements

Some statements of hipi are inherited from the core language, albeit with a slightly different syntax: these are immutable variable declarations, spawns, communications and selections. Others, as function calls and for–loops, are new. Recursive definitions and the `new` statement of the core language are not inherited.

Immutable Variable Declarations The syntax of an immutable variable declaration is:

$$\mathit{Type}\ \ \mathit{var}\ =\ \mathit{Expression}\ \ ;$$

It differs from the variable declaration of the hipi core language by the absence of the `let` keyword.

The scope of the variable is the block that contains it. The value of the expression is assigned to the new variable and it cannot be replaced by subsequent statements. The type of the expression must be structurally equivalent to *Type*. This kind of declaration substitutes also the `new` statement of the core language because here a `new channel` expression is defined.

Parallel Processes (and Migrations) The syntax of Spawn statements is:

$$\texttt{spawn}\ \Big[\ @\ \ \mathit{var}\ \Big]\ \{\ \mathit{Statement}\ \}$$

and the behavior is that the contained *Statement* starts a parallel execution. If the option `@x` is used, the *Statement* is executed at the location of channel x.

Communications on Channels The syntax of output and input on channels is given by the following rules:

$$\mathit{var}\ .\ \texttt{send}\ \ (\ \Big[\ \mathit{ExpressionList}\ \Big]\ )\ ;$$
$$\mathit{var}\ .\ \texttt{asend}\ \ (\ \Big[\ \mathit{ExpressionList}\ \Big]\ )\ ;$$
$$\mathit{var}\ .\ \texttt{recv}\ \ (\ \mathit{ParamList}\ )\ ;$$

where:

$$\mathit{ExpressionList}\ ::= \mathit{Expression}\ \Big(\ ,\ \mathit{Expression}\ \Big)*$$

and *ParamList* is as defined above.

Communications are as in the core language with the addition of an *asynchronous send* statement. `send` and `recv` are used for synchronous message passing over channels: the scope of the variables in *ParamList* is the whole continuation.

The `asend`, instead, is asynchronous: when it is used for a communication the execution of the program continues immediately, without waiting for the reaction with a `recv`. The `asend` statement is the same as a `send` statement inside a spawned block (i.e. `x.asend();` is equivalent to `spawn{x.send();}`. We added this statement because we found it to be used in hipi programs. However we didn't add an `arecv` statement because it was not so frequently needed and because its implementation could lead to ambiguities. An `arecv` can be encoded to a `recv` placed inside a spawn block (for instance `spawn{x.recv();}`) or as a `recv` inside a migration term (for instance `spawn@x{x.recv();}`). Since the sequence and the kind of the messages exchanged between locations in the two cases are different, we preferred to leave them to the programmer.

Function Calls and Returns The function call statement can be used with `void` functions or when the return value of the function is not used in the rest of the program. Return statements, can be used only inside function bodies. Their syntax is given by:

$$\textit{funName} \;\; (\;\; \Big[\;\; \textit{ExpressionList} \;\; \Big] \;\;);$$
$$\texttt{return} \;\; \Big[\;\; \textit{Expression} \;\; \Big] \;\; ;$$

where *ExpressionList* is as defined above.

Function calls are replacements for recursive definition calls of the core language. The difference between them is that the statements after a function call are executed only when the called function terminates, but the statements after a recursive definition are executed immediately.

We experienced, in programming in hipi, that both functions and recursive definition can be useful for a programmer. We preferred to insert functions in hipi instead of recursive definitions because a function can behave as a recursive definition by adding a spawn statement to the caller.

Control–Flow Commands In hipi two control–flow statements are defined: *selection* and *for–loop*. Their syntaxes are:

$$\texttt{if} \;\; (\;\; \textit{Expression} \;\;) \;\; \textit{Statement}_{\,1} \;\; \Big[\;\; \texttt{else} \;\; \textit{Statement}_{\,2} \;\; \Big]$$
$$\texttt{for} \;\; \textit{var} \; = \; \textit{Expression}_{\,1} \;\; \texttt{to} \;\; \textit{Expression}_{\,2} \;\; \Big[\;\; \texttt{by} \;\; \textit{Expression}_{\,3} \;\; \Big] \;\; \textit{Statement}$$

Selection (i.e. the `if` statement) is as usual. The for–loop is as in Pascal: a read–only integer variable, whose scope is *Statement*, is used as the index of the iteration. It ranges between the values of $\textit{Expression}_1$ and $\textit{Expression}_2$. At each iteration it is increased by the value of $\textit{Expression}_3$ if it specified, otherwise it is increased by 1. The three expressions are evaluated once before the loop is entered and never during the cycle. The iteration terminates when the index becomes equal or greater to the value of $\textit{Expression}_2$

Blocks Blocks of code can be written to limit the scope of a variable declaration or to group together statements for an `if` or a for–loop. The syntax rule of a block is the following:

$$\textit{Statement} \;\; ::= \;\; \{ \;\; \textit{Statement} \;\; \}$$

Expressions

The simplest expressions in hipi are *literals* (integer and string values), *variables* and *function calls*. Then there are *URIs*, used to refer to pre–existing channels, and the `new channel` expression, which has the side effect of creating a new channel. Expressions are enriched with unary and binary integer operators, and comparison operators.

In hipi, expressions are described by the following grammar:

$$\begin{aligned} \mathit{Expression} ::= &\ \mathit{LiteralInt} \mid \mathit{LiteralString} \mid \mathit{var} \\ \mid &\ \mathit{Expression}\ \ \mathit{op}\ \ \mathit{Expression} \\ \mid &\ -\ \mathit{Expression} \mid\ !\ \mathit{Expression} \\ \mid &\ \mathit{funName}\ \left(\ \big[\ \mathit{ExpressionList}\ \big]\ \right) \\ \mid &\ \mathit{uri} \mid \texttt{new}\ \ \mathit{Type} \\ \mid &\ (\ \ \mathit{Expression}\ \ ) \\ \mathit{op} ::= &\ + \mid - \mid * \mid / \mid \% \mid \&\& \mid || \mid \\ \mid &\ < \mid > \mid <= \mid >= \mid == \mid != \end{aligned}$$

Comparisons are allowed only between elements whose types are structurally congruent; the other operators can be used only between integer expression. Precedence rules are as usual: unary operators come first, then there are `*`,`/` and `%`, then `+` and `-`, then comparators and finally `&&` and `||`. Parenthesis can be used to specify different precedences as usual.

The *Type* used in the `new` expression must be a channel type. The `new` expression creates a new channel and can be used to give it a name (by assigning the expression to a variable), to pass it to functions or to send it over channels. Since the type used after `new` must be a channel type, we often call this expression `new channel`.

Examples

Here we illustrate the examples of sections 2.1 and 2.3 in hipi. The first one executes a reaction:

```
schedule Reaction {
    main {
        new channel<> x;
        spawn { x.send(); }
        x.recv();
    }
}
```

The second one simulates a ping pong match between two parallel processes:

```
typedef pingpongT = channel<pingpongT>;

schedule PingPong {
    void recursive_player(pingpongT ping) {
        ping.recv(pingpongT pong);
        pong.send(ping);
        spawn { recursive_player(ping); }
    }

    main {
        // creates the channels used by the players
        pingpongT p1 = new pingpongT;
        pingpongT p2 = new pingpongT;
        // starts the execution of both players in parallel
        spawn { recursive_player(p1); }
        spawn { recursive_player(p2); }
        // starts the match
        p1.send(p2);
```

```
    }
}
```

3.2 Translating Hipi into Hipi core

An hipi program can be translated into a set of programs in the hipi core language. We recall that a program in the hipi core language is a triple (Φ,Γ,P) where Φ is a mapping from type identifiers into type expressions, Γ is a mapping from recursive definition names into abstractions and P is a process whose syntax is described in section 2.1.

We now give a formal definition of the translation from hipi into hipi core. For the sake of simplicity we define a two step translation: the first step translates for–loops into functions and the second step translates the result into a set of hipi core programs. For the sake of the simplicity we assume no import declarations. Let $global(H)$ be the set of global functions of a hipi program H and $local(Sdl)$ be the set of local functions of a schedule Sdl.

Definition 3.1 (For–Loops Encoding) *Given a hipi program H, the correspondent For–Loops Free Program H_n is obtained by replacing every for loop*

$$\texttt{for}\ x = E_1\ \texttt{to}\ E_2\ \texttt{by}\ E_3\ S$$

with:

$$\texttt{int}\ x_1 = E_1;\ \texttt{int}\ x_2 = E_2;\ \texttt{int}\ x_3 = E_3;\ f(fn(S), x_1, x_2, x_3)\ ;$$

where f is the following global function:

$$\begin{array}{l} \texttt{void}\ f\,(\widetilde{Tv}, \texttt{int}\ a, \texttt{int}\ b, \texttt{int}\ c)\{ \\ \quad \texttt{if(}\ a < b\ \texttt{)}\{\ S\ f(fn(S), a + c, b, c);\ \} \\ \} \end{array}$$

where $\widetilde{Tv}$ is a typed representation of $fn(S)$.

Before defining the second step of the encoding, we introduce a derived composition operator $\cdot$ for statements in the hipi core language. We use parenthesis to specify precedences.

Definition 3.2 *Composition $\cdot$ is a binary operator for statements in the hipi core language defined by:*

$$\begin{array}{rcl} x.\mathrm{send}(\widetilde{E});P_1 \cdot P_2 & = & x.\mathrm{send}(\widetilde{E});(P_1 \cdot P_2) \\ x.\mathrm{recv}(\widetilde{Tx});P_1 \cdot P_2 & = & x.\mathrm{recv}(\widetilde{Tx});(P_1 \cdot P_2) \\ \mathrm{let}\ x{=}\widetilde{E}\ \mathrm{in}\ P_1 \cdot P_2 & = & \mathrm{let}\ x{=}\widetilde{E}\ \mathrm{in}\ (P_1 \cdot P_2) \\ \mathrm{new}Tx\ \mathrm{in}\ P_1 \cdot P_2 & = & \mathrm{new}Tx\ \mathrm{in}\ (P_1 \cdot P_2) \\ \mathrm{if}(E)\{P_1\}\{P_2\} \cdot P_3 & = & \mathrm{if}(E)\{P_1 \cdot P_3\}\{P_2 \cdot P_3\} \\ D(\widetilde{E});P_1 \cdot P_2 & = & D(\widetilde{E});(P_1 \cdot P_2) \\ \mathrm{spawn}\{P_1\}P_2 \cdot P_3 & = & \mathrm{spawn}\{P_1\}(P_2 \cdot P_3) \\ \mathrm{spawn@}x\{P_1\}P_2 \cdot P_3 & = & \mathrm{spawn@}x\{P_1\}(P_2 \cdot P_3) \\ \mathbf{0} \cdot P & = & P \end{array}$$

The composition operator is used to add continuations to statements. Note that the continuation after an `if` statement is copied into both branches of the statement. Also it shows that the scope of a variable is the whole continuation.

Now we define the second (and main) step of the translation from hipi into hipi core. The translation uses an encoding $\{\!\![\cdot]\!\!\}$ that takes the type definitions, the global functions and one schedule of an hipi program and generates one hipi core program (Φ, Γ, P). The encoding is repeated once for each schedule of the source program. The final result is a set of hipi core programs whose elements are as many programs as there were schedules.

Type declarations of the source hipi programs are used by the translation to define the type function Φ in the obvious way. Given a schedule its local functions and the global functions of the hipi program are used to define Γ. If the name of a global function clashes with the name of a local one, such a global function is ignored. Finally P is given by the encoding of the main block of the schedule.

Definition 3.3 (Translation) *Given a schedule Sdl of a hipi program H without for–loops, its translation* $\{\!\![H]\!\!\}$ *into the hipi core language is the triple* (Φ, Γ, P) *that follows. It makes use of a subsidiary translation* $\{\!\![\cdot]\!\!\}$ *form hipi statements into hipi core statements, defined below.*

$$\Phi = \left\{ \, (\, t \, , \, t' \,) \;\middle|\; \texttt{typedef t} = \texttt{t}'; \in H \, \right\}$$

$$\Gamma = \Big\{ \big(\, f, \; (\widetilde{Tx}, \texttt{channel}\langle \texttt{T}'' \rangle \; r \,).S' \,\big) \;\Big|\; \texttt{T}'\texttt{f}(\widetilde{\texttt{Tx}})\{\texttt{S}\} \in \mathit{local}(Sdl) \cup \mathit{global}(H) \setminus \mathit{local}(Sdl) \Big\}$$

$$P = \{\!\![M]\!\!\}$$

where:

r not free in S'

M is the `main` *of Sdl*

$$T'' = \begin{cases} T' & \text{if } T' \neq \texttt{void} \\ \varnothing & \text{otherwise} \end{cases}$$

$$S' = \begin{cases} \{\!\![S\{^{\texttt{r.send(E);}}/_{\texttt{return E;}}\}]\!\!\} & \text{if } T' \neq \texttt{null} \\ \{\!\![S\{^{\texttt{r.send();}}/_{\texttt{return;}}\}]\!\!\} & \text{otherwise} \end{cases}$$

The encoding $\{\!\![\cdot]\!\!\}$ *maps hipi statements into statements in the core language:*

$$\{\!\![x.\mathrm{send}(\widetilde{E});]\!\!\} = A \cdot x.\mathrm{send}(B); \qquad \text{where } (\![E]\!) = (A, B)$$

$$\{\!\![x.\mathrm{asend}(\widetilde{E});]\!\!\} = A \cdot \mathrm{spawn}\{x.\mathrm{send}(B);\} \qquad \text{where } (\![E]\!) = (A, B)$$

$$\{\!\![x.\mathrm{recv}(\widetilde{Ty});]\!\!\} = x.\mathrm{recv}(\widetilde{Ty});$$

$$\{\!\![\mathrm{spawn}\{S\}]\!\!\} = \mathrm{spawn}\{\{\!\![S]\!\!\}\}$$

$$\{\!\![\mathrm{spawn@}x\{S\}]\!\!\} = \mathrm{spawn@}x\{\{\!\![S]\!\!\}\}$$

$$
\begin{aligned}
\{\!\![\, f(\widetilde{E});]\!\!\} &= \text{new } \mathit{channel}\langle T'\rangle\ r \text{ in } \cdot A \cdot f(B,r);r.\text{recv}(T'y); \\
&\qquad\qquad\qquad\qquad\qquad \textit{where } r,y \textit{ fresh} \quad (\![E]\!) = (A,B) \\
\{\!\![\{S\}]\!\!\} &= \{\!\![S]\!\!\} \\
\{\!\![Tx{=}E \text{ in }]\!\!\} &= A \cdot \text{let } Tx{=}B \text{ in } \varnothing \qquad\qquad \textit{where } (\![E]\!) = (A,B) \\
\{\!\![\text{if}(E)\{S_1\}\{S_2\}]\!\!\} &= A \cdot \text{if}(B)\{S_1'\}\{S_2'\} \qquad\qquad \textit{where } (\![E]\!) = (A,B)
\end{aligned}
$$

The encoding $(\![\cdot]\!)$ *is the following mapping from sequences of hipi expressions into pairs* $(S,\widetilde{E})$ *where* S *is a statement and* $\widetilde{E}$ *a sequence of expressions in the hipi core language.*

$$
\begin{aligned}
(\![\widetilde{E}]\!) &= \big(\, A_1 \cdots A_n \,,\ B_1 \cdots B_n \,\big) && \textit{where } (\![E_i]\!) = (A_i,B_i) \\
(\![\, \textit{uri}\,]\!) &= \big(\, \mathbf{0} \,,\ x_{uri} \,\big) \\
(\![\texttt{new}T]\!) &= \big(\, \text{new } Tx \text{ in } \,,\ x \,\big) && x \textit{ fresh} \\
(\![\, f(\widetilde{E});]\!) &= \big(\, \text{new } \mathit{channel}\langle T'\rangle\ r \text{ in } \cdot A\cdot f(B,r);r(T'y) \,,\ y \,\big) \\
& && \textit{where } x \textit{ fresh} \ \ (\![E]\!) = (A,B) \\
(\![E_1 \ \mathit{op}\ E_2]\!) &= \big(\, A_1 \cdot A_2 \,,\ B_1 \ \mathit{op}\ B_2 \,\big) && \textit{where } (\![E_i]\!) = (A_i,B_i) \\
(\![!E]\!) &= \big(\, A \,,\ !B \,\big) && \textit{where } (\![E]\!) = (A,B) \\
(\![-E]\!) &= \big(\, A \,,\ -B \,\big) && \textit{where } (\![E]\!) = (A,B) \\
(\![i]\!) &= \big(\, \mathbf{0} \,,\ i \,\big) && \textit{where } i \in \textit{ LiteralInt} \\
(\![s]\!) &= \big(\, \mathbf{0} \,,\ s \,\big) && \textit{where } s \in \textit{ LiteralString} \\
(\![x]\!) &= \big(\, \mathbf{0} \,,\ x \,\big) \\
(\![(E)]\!) &= \big(\, A \,,\ B \,\big) && \textit{where } (\![E]\!) = (A,B)
\end{aligned}
$$

where op ranges over $\{+,-,*,/,\%,\&\&,||,<,>,<=,>=,==,!=\}$.

The main non–straightforward rules of the encoding $\{\!\![\ \cdot\ ]\!\!\}$ are related to the translation of function calls. The *synchrony* of a function call is obtained by using a communication on a new channel r to transmit the return value. The function call is translated into a recursive definition call followed by an input on r. Return statements are translated into outputs on r (as shown in the definition of Γ).

We remark that `arecv` is encoded into a `recv` statement contained in a spawned block, and that to both branches of the `if` statement the whole continuation is appended.

The encoding $(\![\cdot]\!)$ has the feature of removing side effects from the `new` and the function call expressions. We remark that expressions in hipi have side effect but those in the core language do not: $(\![\cdot]\!)$ substitutes side effects with statements. We now give an example that shows the translation of a function call expression:

```
x.send(fact(10));
```

it is translated into the following sequence of hipi core statements (without side effects):

```
new channel<int> r in
fact(10);
r.recv(int y);
x.send(y);
```

3.3 XIL: XML Intermediate Language

The XML Intermediate Language (XIL) is an XML representation of the bytecode language described in section 2.3. A XIL file represents a schedule (i.e. a loadable and executable entity) and it is an XML document made of a sequence of named threads. One of these threads is named “main” and it is the starting point of the execution.

The only non–syntactic difference between XIL and the bytecode language is that the the former uses a stack to store variable values, while the latter uses substitutions in the operational semantics. A stack is a standard way to implement substitutions: in XIL its elements can be strings, integers and URIs (used to represent channels). We now give the complete syntax of XIL, and then we explain each part of it. The XML–schema document that can validate XIL files is shown in appendix A.

$$
\begin{array}{rcl}
\mathit{XILProg} & ::= & <\texttt{schedule}\ \texttt{name} = \text{“}\ \mathit{schedName}\ '' > \\
 & & \quad \mathit{Thread}* \\
 & & </\texttt{schedule}> \\
\mathit{Thread} & ::= & <\texttt{thread}\ \texttt{name} = \text{“}\ \mathit{threadName}\ '' > \\
 & & \quad <\texttt{stacksize}\ \texttt{init} = \text{“}\ n\ ''/> \\
 & & \quad \mathit{Statement}* \\
 & & </\texttt{thread}> \\
 & & \\
\mathit{Statement} & ::= & <\texttt{send}>\ \mathit{LoadExp}\ \ \mathit{Expression}*\ \ </\texttt{send}> \\
 & | & <\texttt{recv}>\ \mathit{LoadExp}\ \ \mathit{StoreExp}*\ \ </\texttt{recv}> \\
 & | & <\texttt{fwd}>\ \mathit{LoadExp}\ \ \mathit{LoadExp}\ \ </\texttt{fwd}> \\
 & | & <\texttt{spawn}\ \texttt{thread} = \text{“}\ \mathit{threadName}\ '' \\
 & & \qquad \left[\ \texttt{dest} = \text{“}\ n\ |\ \mathit{uri}\ ''\ \right]\ > \\
 & & \quad \mathit{Expression}* \\
 & & </\texttt{spawn}> \\
 & | & <\texttt{store}\ \texttt{idx} = \text{“}\ n\ '' >\ \mathit{Expression}\ \ </\texttt{store}> \\
 & | & <\texttt{newch}\ \texttt{idx} = \text{“}\ n\ ''/> \\
 & | & <\texttt{if}>\ \mathit{Expression} \\
 & & \quad <\texttt{then}>\ \mathit{Statement}*\ \ </\texttt{then}> \\
 & & \quad <\texttt{else}>\ \mathit{Statement}*\ \ </\texttt{else}> \\
 & & </\texttt{if}> \\
 & | & <\texttt{terminate}/> \\
 & & \\
\mathit{LoadExp} & ::= & <\texttt{load}\ \texttt{src} = \text{“}\ n\ ''/> \\
 & | & <\texttt{load}\ \texttt{src} = \text{“}\ \mathit{uri}\ ''/>
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{StoreExp} & ::= & <\texttt{store idx} = \text{``}\ n\ \text{''}/> \\
\\
\mathit{Expression} & ::= & \mathit{LoadExp} \\
 & | & <\texttt{int}>\ \mathit{LiteralInt}\ </\texttt{int}> \\
 & | & <\texttt{string}>\ \mathit{LiteralString}\ </\texttt{string}> \\
 & | & <\texttt{op type} = \text{``}\ \mathit{ExpOp}\ \text{''}> \\
 & & \quad \mathit{Expression}\ \left[\ \mathit{Expression}\ \right] \\
 & & </\texttt{op}> \\
\mathit{ExpOp} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{mod} \mid \texttt{and} \mid \texttt{or} \\
 & | & \texttt{eq} \mid \texttt{neq} \mid \texttt{lt} \mid \texttt{gt} \mid \texttt{le} \mid \texttt{ge} \mid \texttt{umin} \mid \texttt{not}
\end{array}
$$

In XIL *LiteralInt* and *LiteralString* are integers and strings as usual. The number of children of the `op` tag depends by the arity of the operator as described in table 3.1

Op.Type	**N. of Children**	**Description**
`add`	2	sum
`sub`	2	subtraction
`mul`	2	multiplication
`div`	2	division
`mod`	2	modulus
`and`	2	boolean and
`or`	2	boolean or
`eq`	2	equal to
`neq`	2	not equal to
`lt`	2	lower than
`gt`	2	greater than
`le`	2	lower or equal than
`ge`	2	greater or equal than
`umin`	1	unary minus
`not`	1	boolean not

Table 3.1: Values of the type attribute of `<op>` and their descriptions

Each `<thread>` element of a XIL program has `<stacksize>` as a required first child indicating (as the value of its required attribute `init`) the initial size of the stack that will be associated to it at run–time. The `main` thread must have initial stack size equal to zero. The initial stack size corresponds to the number of parameters of an abstraction in the bytecode language. An example of `<thread>` is:

```
<thread name = "foo">
    <stacksize init="2"/>
    <store idx = "2">
        <op type="sum">
            <load idx="0"/>
            <load idx="1"/>
        </op>
    </store>
</thread>
```

Statements are based on the terms of the bytecode language of section 2.3. <send> and <recv> are used for communication over channels. Their first child represent the subject of the communication. The remaining children of the <send> tag are the object of the communication; the remaining of the <recv>, on the other hand, are used to describe at which positions of the stacks the received elements must be stored. We now give examples of <send> and <recv>, the first shows the output of 5 and ch://channel.x over the channel at position 10 of the stack:

```
<send>
    <load src="10"/>
    <int>5</int>
    <load src="ch://channel.x"/>
</send>
```

and the second shows the input of two elements (stored at positions 3 and 4 of the stack) over the channel at position 10 of the stack:

```
<recv>
    <load src="10"/>
    <store idx="3"/>
    <store idx="4"/>
</recv>
```

<fwd> is a linear forwarder: it must have exactly two children that are the old and the new destination of the forwarded message. An example is:

```
<fwd>
    <load src="3"/>
    <load src="4"/>
</fwd>
```

<spawn> starts the execution of the specified named thread: the optional dest attribute is used to execute the new thread at a different location. The children of the <spawn> tag are parameters passed to the new thread in order to build its stack. An example of <spawn> (that executes the foo thread defined above) is:

```
<spawn thread = "foo">
    <int>10</int>
    <load src="12"/>
</spawn>
```

<store> puts the value obtained by its child into a specific position of the stack (given by the required idx attribute). The <newch> statement creates a new channel and stores its URI at the specified position of the stack (as using a <store> with a <newch/> expression inside it). The <if> is as usual and <terminate> is used to terminate the execution of the current thread.

The <load> expression is defined for loading values from stack and for referencing to global (pre–existing) channels: in the first case the required src attribute is an index of the stack, in the second case it is a URI. <int> and <string> are used to represent literals and <op> is used for unary and binary operators.

We explain briefly how the stack is used at run–time. At the start the main thread is executed and a new (empty) stack is associated to it. This is the reason why the init value of the <stacksize> child of the main thread is required to be 0. During execution, every time a

<spawn> is executed, a new stack for the spawned thread is created. Such a new data structure contains the element represented by the n children of <spawn> at the first n positions. The <stacksize> tag of the spawned thread must have `init` value equal to n.

Every time a new variable is bound its value is stored into the stack and every time that variable is used its value is loaded from the stack: this is how substitutions in the bytecode operational semantics (described in section 2.4) are implemented using a stack. The <store> statement is used to put values into a location of the stack (previous values are overwritten): this allows space optimizations, obtained by reusing of stack positions.

We now rewrite here the examples of section 2.3 using the XML notation just described. The first example is the reaction:

```
<schedule name="Example1">
    <thread name="main">
        <stacksize init="0"/>
        <newch idx="0">
        <spawn thread="child">
            <load src="0"/>
        </spawn>
        <recv>
            <load src="0"/>
        </recv>
    </thread>
    <thread name="child">
        <stacksize init="1"/>
        <send>
            <load src="0"/>
        </send>
    </thread>
</schedule>
```

The second example simulates a ping–pong match:

```
<schedule name="Example2">
    <thread name="main">
        <stacksize init="0"/>
        <newch idx="0"/>
        <newch idx="1"/>
        <spawn thread="recursive_player">
            <load src="0"/>
        </spawn>
        <spawn thread="recursive_player">
            <load src="1"/>
        </spawn>
        <send>
            <load src="0"/>
            <load src="1"/>
        </send>
    </thread>
    <thread name="recursive_player">
        <stacksize init="1"/>
        <newch idx="1"/>
        <spawn thread="n">
            <load src="0"/>
            <load src="1"/>
        </spawn>
```

```
        <recv>
            <load src="1"/>
            <store idx="2"/>
        </recv>
        <send>
            <load src="2"/>
            <load src="0"/>
        </send>
        <spawn thread="recursive_player">
            <load src="0"/>
        </spawn>
    </thread>
    <thread name="n">
        <stacksize init="2"/>
        <fwd>
            <load src="0"/>
            <load src="1"/>
        </fwd>
    </thread>
</schedule>
```

3.4 The Hipi Compiler

In chapter 2 we presented the hipi core language and the bytecode and we defined an encoding that can be used to translate a program in the core language into a piece of *behaviorally equivalent* bytecode. In the first sections of this chapter we illustrated hipi as an extension of the hipi core and we illustrated how to translate an hipi program into a set of core programs.

Combining the two encodings and using the XML notation of XIL for the bytecode, we obtain *the* way to translate an hipi program into a set of XIL files. This hipi-to-XIL encoding is what we implemented in the hipi compiler.

The compiler operates in phases, each of which transforms the source program from one representation to another. The 4 phases of the hipi compiler are:

1. lexical and syntax analysis
2. semantic analysis
3. encoding
4. XIL code generation

Phase 1 was implemented using the parser generator JavaCC: a tool that creates a set of Java classes that can be used to detect if a source file matches a given grammar. The output of this phase is the Abstract Syntax Tree representing the input source file.

The semantic analysis (phase 2) checks the well–formedness of the source program and gathers the information for the subsequent phases. It uses the AST to identify the operators and operands of expressions and statements and uses a symbol table to store temporary information. Some information useful in next phases are stored into nodes of the AST.

The encoding phase (numbered 3) takes the checked AST, enriched by the semantic analysis, and applies to it the encoding formally defined in previous chapter and sections. The output of this phase is a tree structure representing the XIL document that will be returned by the compiler.

Finally, the code generation phase makes some optimizations to the tree structure returned by the encoding and generates the XIL output document. We now give more details about each phase.

Lexical and Syntax Analysis Both lexical and syntax analysis were implemented using a parser generator. The tool used was the *Java Compiler Compiler (JavaCC)* that is one of the most popular parser generators for Java. The main features of JavaCC are: (1) generation of top–down (recursive descent) parsers as opposed to bottom–up parsers generated by YACC-like tools; (2) tree building preprocessor (via the included JJTree tool); (3) syntactic and semantic lookahead specification. By default, JavaCC generates an LL(1) parser, but it is possible to change the value of the lookahead for specific portions of the grammar. It is possible also to use a special lookahead that checks if an input fragment is equal to a specific sequence of terminals or satisfies a specific non–terminal rule of the grammar.

In our implementation, the grammar specified by the input file of JavaCC is equivalent to the one of section 3.1 but without left recursions (because they are avoided in LL(n) grammars). JJTree is used to create the Abstract Syntax Tree with the nodes listed in table 3.2: for each node of the tree a Java class is created.

Node	**Description**
`ASTHiPiProgram`	The root of the Abstract Syntax Tree. Childnodes are type declarations, imports, functions and schedules.
`ASTTypeDeclaration`	Contains the type identifier and the type expression of a type declaration. No childnodes.
`ASTImportDeclaration`	Contains the name of an imported file. No childnodes.
`ASTFunction`	Contains the type and the description of the formal parameters. Childnodes are the body of the function.
`ASTSchedule`	Contains the name of the schedule and a set of URIs representing the co–located channels. Childnodes are functions, local declarations and the main block.
`ASTScheduleMain`	No information contained. Childnodes are statements.
`ASTLocalDeclStatement`	Contains the name and the type of the new variable or channel. The childnode is the assigned expression.
`ASTFunctionCall`	Contains the name of the called function. Childnodes are the expressions used as arguments.
`ASTSend`	Contains the name of the subject channel. Childnodes are the expressions used as object.
`ASTRecv`	Contains the name of the subject channel and the description of the received names with their types
`ASTSpawnStatement`	Optionally contains a destination channel (if it is a migration). Childnodes are statements of the new parallel process.
`ASTBlock`	Childnodes are statements.
`ASTReturnStatement`	The childnode is the returned expression (if any).
`ASTSelectionStatement`	The first childnode is an integer expression, then there are two child blocks representing the two branches of the selection.
`ASTNewChannel`	Represents the `new channel` expression.
`ASTExpIdentifier`	Represents a variable inside an expression. Contains the name of the variable.
`ASTExpInteger`	Represents a literal integer. Contains an integer value,
`ASTExpString`	Represents a literal string. Contains a string value,
`ASTExpOp`	Contains the rapresentation of an operator. Childnodes are the expressions used as operands.

Table 3.2: Nodes of the Abstract Syntax Tree

Semantic Analysis This phase is implemented as a depth–first visit of the Abstract Syntax Tree. During the visit the well–formedness of the tree (that reflects the well–formedness of the source program) is checked. Variables scopes are checked in this phase: a variable must be defined before it is used inside expressions or as subject of a communication. No pairs of variables sharing the same name can be defined in the same block. During the visit of the AST a *depth counter* is used: at the root node it is set to 0 and its value is changed during the visit. Using this counter together with a symbol table it is possible to check the scopes and types of the variables.

The symbol table is a dynamic data structure used to store temporary information about variables. The symbol table is an hash table that contains name, type and depth of each variable declared in the program. It can be accessed using an hash function on the name of the variable. A graphical representation of the symbol table is given in figure 3.1.

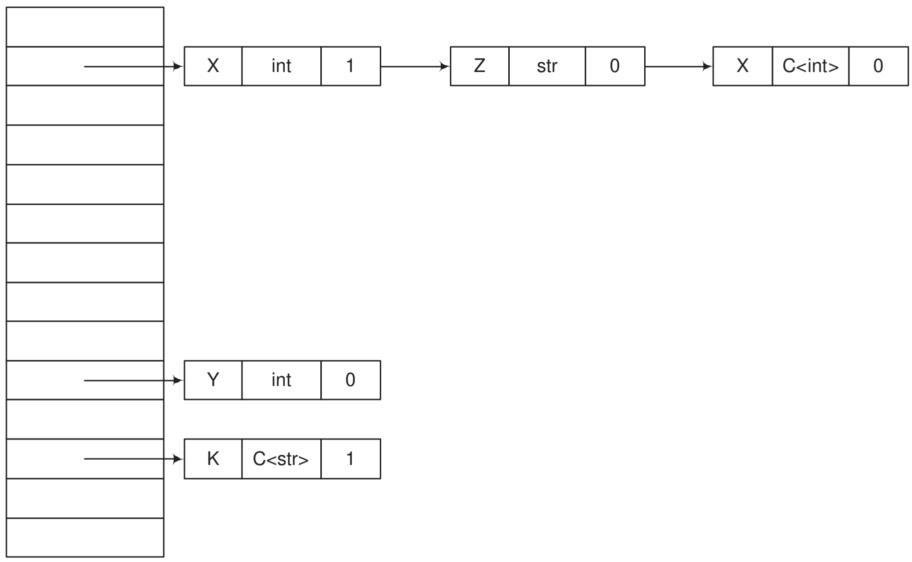

Figure 3.1: A graphical representation of the symbol table

The figure shows a symbol table containing five variables. Z and the two occurrences of X are an example of names whose hash values are the same: they form a list. We give a description of how variables are checked during the visit. Let us denote elements of the symbol table as a set of triples (id,type,depth):

1. the depth counter is initialized to 0;

2. the visit starts;

3. every time a new variable `x` is declared: if (`x`,t,d) is present into the symbol table and d is equal to the value of the depth counter, then an error message is returned. Otherwise (`x`,t,c) is inserted, where t is the type of `x` and c is the value of the counter;

4. every time a variable is used inside an expression or as a subject of a communication it is found inside the symbol table: if it is not present an error message is returned;

5. every time a block or a branch of a selection or a spawn block is entered the depth counter is incremented by one;

6. every time a block or a branch of a selection or a spawn block is exited the depth counter is decremented by one and all the elements of the symbol table with depth equal to the old value are removed.

Encoding In this phase the AST is transformed in order to obtain a data structure that represents the output XIL program. The encoding was described in the previous sections; we now give only an example of how it is applied to the tree structure.

We recall that in hipi a `send` statement is defined and it is used for synchronous communications. In the bytecode, instead, output operations are asynchronous, so an acknowledgment channel is used to encode the hipi into bytecode. Figure 3.2 shows the AST fragment representing the synchronous `send` of hipi on the left, and its encoding with a new channel for the acknowledgment on the right.

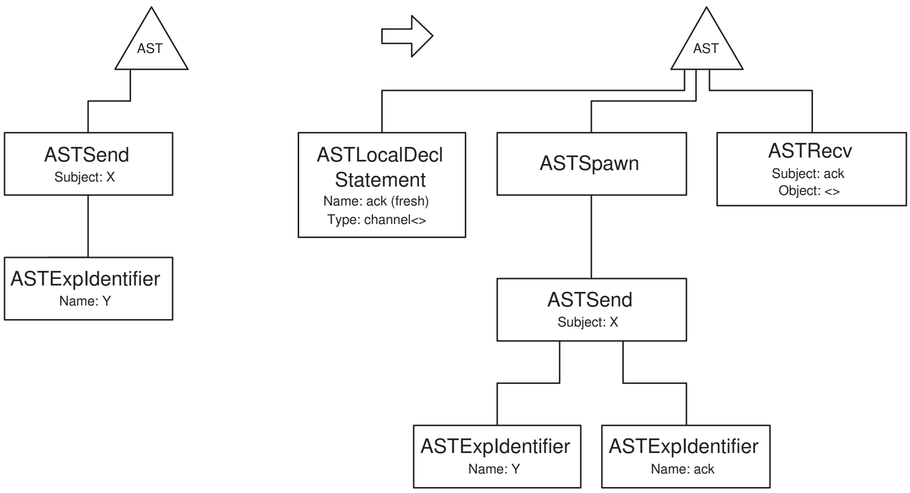

Figure 3.2: The encoding of the `send` statement as a transformation of the AST

XIL code generation The tree structure generated by the encoding is used to generate the XIL document that is the output of the encoding. In this phase all the subtree of the ASTSpawn nodes and of the ASTFunction nodes are removed from the AST and used to generate the sequence of named thread of the XIL document. All the other nodes are directly translated into one or more XIL tag in the obvious manner. Some space optimizations are made in this phase: for instance, some threads generated by the previous encoding are repeated (like the thread that sends the acknowledgment after an output), so they are substituted by a single thread executed once for each output.

3.5 The Network Simulator

Together with the hipi compiler we implemented also a *network simulator* that can be used to execute XIL programs generated by the compiler. This tool simulates a network of *virtual machines*. In our implementation a virtual machine is the equivalent of a location: it can host channels and it can execute XIL programs.

The simulator takes the filename of a "geometry" file describing the layout of the simulated network as a command line argument, and uses this geometry for its initialization. An example geometry file is the following:

```
<network>
    <vm name="Bologna">
        <channel uri="ch://virtualmachine.bo.it/one"/>
        <channel uri="ch://virtualmachine.bo.it/two"/>
    </vm>

    <vm name="NewYork">
        <channel uri="ch://virtualmachine.ny.com/one"/>
        <channel uri="ch://virtualmachine.ny.com/two"/>
    </vm>
    <vm name="Paris"/>
</network>
```

The network described by the example document is made of three virtual machines, namely *Bologna*, *NewYork* and *Paris*. Two channels are hosted at both Bologna and NewYork, no channels are hosted by Paris. The four declared channels are the only pre–existing channels: they are the equivalent of the *external channels* described in section 2.4 and their URIs can be used to communicate on them.

After initialization the simulator starts a command shell that allow the user to load XIL programs and simulate their execution. The first screen of the console illustrates the available commands:

```
RNG seed = 1000
Simulator initialized...
Verbose mode is off
Starting console...
Commands:
  load <VM_name> <filename>   - Load a schedule on a VM
  play                        - Execute forever
  step <num>                  - Execute <num> steps
  s                           - The same as: step 1
  view                        - Print simulator state
  help                        - This message
  verb                        - Switch verbose mode on|off
  quit                        - Exit the simulator

ATT: Console is case sensitive

SIM>
```

Figure 3.3 shows a graphical representation of the simulator.

The simulator handles a set of virtual machines created as specified in the geometry document, and a set of DOM trees that are the representations of all the loaded schedules. A virtual machine contains a set of channel managers that are used to handle communication over channels. Each virtual machine is started with one channel manager for each channel it hosts as declared in the geometry file. New channel managers can be created at run–time. A virtual machine contains also a set of threads: each thread has its own stack and a reference to a node of a DOM tree (this is the equivalent of a program counter). Threads of each virtual machine are executed using a round robin scheduler and blocked threads can be enqueued on the input or output queue of a channel manager during execution.

Each time a new schedule is loaded on a virtual machine, the related XIL document is transformed into a DOM tree and a new thread is started in the virtual machine. The stack of this new thread is initialized empty and the DOM tree pointer is initialized to the node that identifies the `main` thread of the new schedule. A simulation step is executed on a randomly chosen virtual machine. In this machine the step is executed in accordance with the round robin policy of its scheduler. When a thread communicates on a channel it is enqueued on the channel manager that handles the channel. When a channel manager has at least one enqueued input thread and one enqueued output thread, a reaction step is possible.

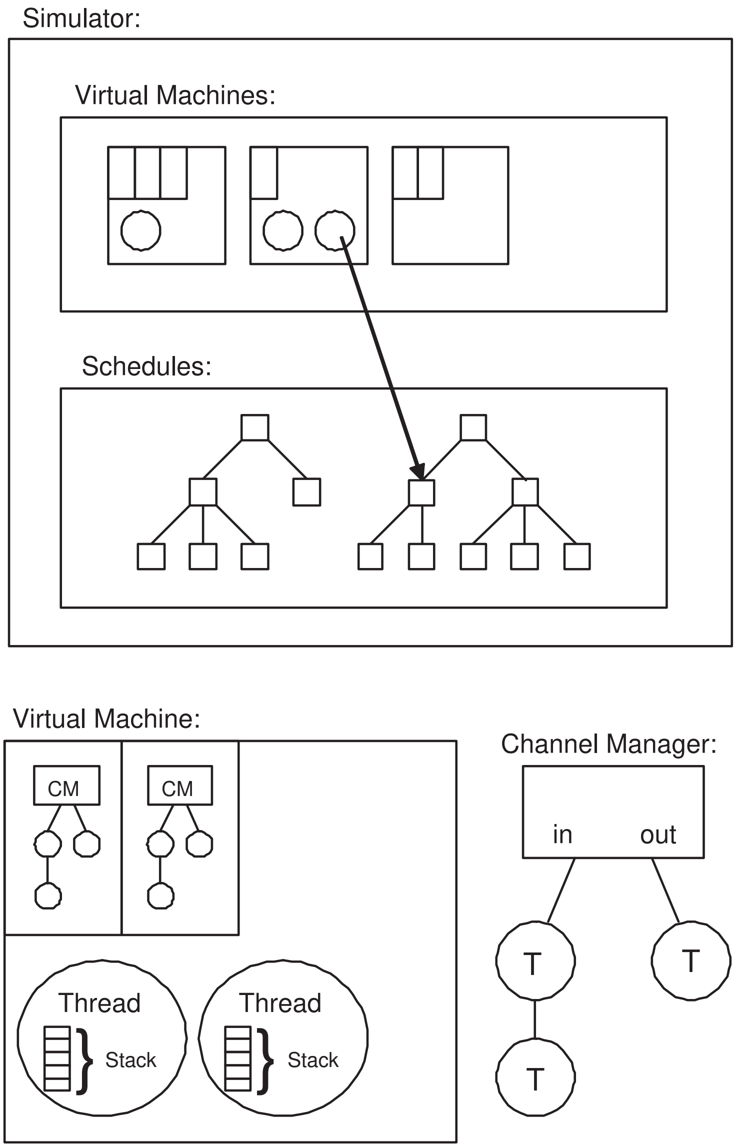

Figure 3.3: A graphical representation of the Network Simulator

Three special URIs are defined for console input and output: `ch://string.console` `ch://int.console` and `ch://channel.console`. These carry strings, integers and channels (of any type). Values sent on them are immediately printed on the console and input operations on them are immediately translated into console inputs. URIs are used to represent channel values on the console.

The state of the simulator can be printed on the console. The state contains information about the channels currently hosted by the virtual machines and about the state of the running and enqueued processes. Here is an example:

```
SIM> view
Bologna:
   Channels - 2
      ch://virtualmachine.bo.it/one
         INPUT QUEUE: 0
         OUTPUT QUEUE:0
      ch://virtualmachine.bo.it/two
         INPUT QUEUE: 1
         OUTPUT QUEUE:0
   Active threads - 0
NewYork:
   Channels - 3
      ch://virtualmachine.ny.it/one
         INPUT QUEUE: 3
         OUTPUT QUEUE:0
      ch://virtualmachine.ny.it/two
         INPUT QUEUE: 0
         OUTPUT QUEUE:1
   Active threads - 2
Paris:
   Channels - 1
      ch://paris/localchannels/idx_1
         INPUT QUEUE: 0
         OUTPUT QUEUE:0
   Active threads - 4
```

Finally we describe some implementation details of the network simulator. It is implemented in Java and it requires a JVM version 1.3 or later. The geometry XML document used during initialization is processed using a SAX parser. All the XIL documents are transformed into DOM documents. The library for XML handling used to parse and elaborate documents is the widely–used `xerces` for Java of the Apache Foundation [Fou]. The random number generator used at each step of the execution is the `Random` class of the Java Standard API: the seed of the rng can be set as a command line argument.

Chapter 4

Thinking in Hipi

Channels, processes and locations are the new features that hipi offers to programmers. The aim of this chapter is to explore how to use them effectively. We show also how some elements often used in imperative languages, but missing in hipi, can be substituted. We begin by describing some idioms, then we show how mutable variables and data structures can be simulated using processes. Then we describe some design patterns for hipi and finally we give a complete programming example.

4.1 Idioms

Here we describe some idioms that are frequently used in hipi programs.

Reply and Return Channels Network communication is often based on request/reply protocols. Requests are made of two parts: (1) a message describing the kind and the data of the request, (2) the address of the sender and some information describing how to reply. Examples are emails and the socket API [Ste93]. In hipi a request/reply communication can be obtained by adding a reply channel to the request, as in the following example:

```
// CLIENT PROCESS: send a request and waits for reply
channel<string> replyCh = new channel<string>;
server.send(request,replyCh);
replyCh.recv(string reply);

// SERVER PROCESS: receive a request and sends reply
server.recv(requestT req, channel<string> reply);
/* handle request */
reply.send(handle(req));
```

In a similar way a channel can be used for additional return values in functions:

```
// function definition
string foo(channel<string> reply) {
    reply.asend("World");
    return "Hello";
}

// function call
channel<string> replyCh = new channel<string>;
string s1 = foo(replyCh);
replyCh.recv(string s2);
```

Note the use of asynchronous communication to send the additional return value. Otherwise results a deadlock.

Mutex Channel A `channel<>` can be used guarantee mutual exclusion in the access to a resource. At the beginning it must be created and initialized by asynchronously sending on it. Then every access to the resource must be preceded by a `recv` and followed by an `asend`. We give an example:

```
// creation and initialization
channel<> mutex = new channel<>;
mutex.asend();

// resource access
mutex.recv();
use_resource();
mutex.asend();
```

A mutex channel is similar to a binary semaphore [Ste93].

Parallel Recursion Tail recursion in a void function should be done by putting the recursive call inside a spawn block. We give an example:

```
void tailRecursive() {
    /*
     * do something
     */
    spawn { tailRecursive(); }
}
```

The reason for this is efficiency of our current implementation. A function call is translated (by the encodings of previous chapters) into a parallel thread that executes the body of the function. The thread of the caller waits until the end of the new thread and then continues its execution. If the function call has no continuation (as in the case of tail recursion) the caller thread can immediately terminate. This is obtained by putting the call inside a spawn statement.

Output Consumer When a message is expected on a channel but it is not used in the continuation, it can be consumed without waste of time by putting the `recv` command into a spawn block:

```
// makes a request and ignores reply
channel<string> replyCh = new channel<string>;
server.send(request,replyCh);
spawn{ replyCh.recv(string ignoreMe); }
```

The use of output consumers avoids "dead" messages on channels.

4.2 Mutable Variables as Processes

Sometimes mutable variables can be useful in hipi programs: for instance, they allow the creation of counters. Milner used channels as pointers to store the value of the mutable variable [Mil99]. He used also a process running a choice to handle operations on such a channel.

The implementation of Milner's encoding in hipi is not straightforward. This is due to the absence in our language of the choice operator of the π–calculus [MPW92]. Instead, we suggest a simple protocol for simulating mutable variables. It uses channels as pointer and leaves the handling part to the programmer. The protocol is as follows:

1. define a new channel that carries values of the type of the expected mutable variable;
2. initialize the variable by asynchronously sending the initial value on the new channel;
3. every time the variable must be accessed or modified: (a) receive the old value on the channel, (b) send asynchronously the new value (or the old one if not changed) on the channel to allow other accesses.

This protocol guarantees the persistence of the variable because every access is followed by an asynchronous send that restores the value. The mutual exclusion on the access of the variable is also guaranteed: this is due to the fact that there is always almost one process sending the value on the channel (this can be easily seen in the protocol). We now give a little programming example of counter in hipi:

```
channel<int> counter = new channel<int>; // 1. define a new channel
counter.asend(0);                        // 2. send initial value
for i=0 to 10 {
    x.recv(string msg);
    if (msg=="countme") {
        counter.recv(int oldval);   // 3a. get old value
        counter.asend(oldval+1);    // 3b. set new value
    }
}
```

The use of polyadic channels allows us also to define simple data structures: for instance, it is possible to define a mutable variable whose type is a pair `(int,string)` by using a channel of type `channel<int,string>`.

4.3 Data Structures as Processes

The type system of hipi lacks any kind of data structure. The reason for this is that as a future work we want to define a new type system based on XML data types. Here we want to explain how, in its absence, data structures as vectors, sets, lists, etc... can be implemented in hipi.

The implementation is inspired by the design given by Milner for the π–calculus in [Mil99]. Milner's uses a chain of processes linked by channels: each element of the chain stores one value of the data structure and a reference to the rest of the chain. The last element of the chain is a special process called *nil*. The element of the chain can be heterogeneous but they are all accessed in the same way (i.e. their communication interface is the same).

Our implementation is a simplification of Milner's design. We consider only lists of homogeneous elements terminated by a "nil" channel. This allows us to use fewer channels and makes operations on the list easier to implement. We give an example of how a list of strings can be implemented in hipi using this design: the main operations on the list are represented by functions.

```
// type of a list of strings
typedef listT = channel<listT,string>

schedule listExample {

    // the terminator of the list
    listT nil = new listT;
```

```
    // list constructor
    listT new_list() { return nil; }

    // adds an element at the head of the list
    // and returns the updated list
    listT add_first(listT l, string s) {
        listT l2 = new listT;
        l2.asend(l,s);
        return l2;
    }

    // adds an element at the tail of the list
    // and returns the updated list
    listT add_last(listT l, string s) {
        if (l==nil) return add_first(l,s);
        l.recv(listT l2, string s2);
        listT l3=add_last(l2,s);
        l.asend(l3,s2);
        return l;
    }

    // removes the element at the head of the list
    // returns the updated list and sends the removed
    // element on channel c
    listT remove_first(listT l, channel<string> c) {
        if (l==nil) {
            c.asend("");
            return l;
        }
        else {
            l.recv(listT l2, string s);
            c.asend(s);
            return l2;
        }
    }

    // removes the element at the tail of the list
    // returns the updated list and sends the removed
    // element on channel c
    listT remove_last(listT l, channel<string> c) {
        if (l==nil) {
            c.asend("");
            return l;
        }
        l.recv(listT l2, string s);
        if (l2==nil) {
            c.asend(s);
            return nil;
        }
        listT l3=remove_last(l2,c);
        l.asend(l3,s);
        return l;
    }
}
```

Now we write an example of `main` that executes some operations on a list:

```
main {
    // creates a new list
    listT mylist = new_list();

    // adds some elements
    listT mylist2 = add_first(mylist,"red");
    listT mylist3 = add_last(mylist2,"white");

    // removes the elements
    channel<string> c = new channel<string>;
    listT mylist4 = remove_first(mylist3, c);
    c.recv(string s1); // receives "red"
    listT mylist5 = remove_first(mylist4, c);
    c.recv(string s2); // receives "white"
}
```

There is an awkwardness in what we have seen: every time an operation on the list is done a new variable is defined (namely `mylist2`, `mylist3`, ...). The new variable is necessary because all the functions implementing the operations return the (possibly) updated list. Furthermore, since all the functions return only the updated list, we need to use a communication on a channel in order to receive the removed element. In the shown example, both the `remove_first` and the `remove_last` functions use the channel `c` to return the removed element.

A better implementation can be done using the protocol of section 4.2 for mutable variables. This allows us to avoid the need of updated lists as return values. The use of mutable variables complicates a bit the implementation of the functions, but simplifies their usage. The previously shown example can be rewritten as follows:

```
// type of an element of the list
typedef elemT = channel<elemT,string>;
// type of a list
typedef listT = channel<elemT>;

schedule listExample {
    // the terminator of the list
    elemT nil = new elemT;

    // list constructor
    listT new_list() {
        listT l = new listT;
        l.asend(nil);
        return l;
    }

    // adds an element at the head of the list
    void add_first(listT l, string s) {
        l.recv(elemT e);
        elemT e2 = new elemT;
        e2.asend(e,s);
        l.asend(e2);
    }

    // adds an element at the tail of the list
    void add_last(listT l, string s) {
        l.recv(elemT e);
        if (e==nil) {
            elemT e2 = new elemT;
```

```
            e2.asend(e,s);
            l.asend(e2);
        }
        else {
            e.recv(elemT e2, string s2);
            listT l2 = new listT;
            l2.asend(e2);
            add_last(l2,s);
            l2.recv(elemT e3);
            e.asend(e3,s2);
            l.asend(e);
        }
    }
    // removes the element at the head of the list
    string remove_first(listT l) {
        l.recv(elemT e)
        if (e==nil) {
            l.asend(e);
            return "";
        }
        else {
            e.recv(elemT e2, string s);
            l.asend(e2);
            return s;
        }
    }

    // removes the element at the tail of the list
    string remove_last(listT l) {
        l.recv(elemT e);
        if (e==nil) {
            l.asend(e);
            return "";
        }
        e.recv(elemT e2, string s);
        if (e2==nil) {
            l.asend(e2);
            return s;
        }
        listT l2 = new listT;
        l2.asend(e2);
        string s2 = remove_last(l2);
        l2.recv(elemT e3);
        l.asend(e3);
        return s2;
    }
}
```

And now, the `main` becomes more friendly:

```
main {
    // creates a new list
    listT mylist = new_list();
    // adds some elements
    add_first(mylist,"red");
    add_last(mylist,"white");
    // removes the elements
```

```
    string s1 = remove_first(mylist); // "red"
    string s2 = remove_first(mylist); // "white"
}
```

The implementation of data structures like sets, stacks and vectors based on lists is standard. We suggest also an alternative implementation for vectors with constant time access operations based on channel types. For instance, a channel typed `channel< string, string, string, string, string>` can be used to store a vector of five strings. This allows the access of the whole array with a single `recv` instruction, but this requires pre–determined array sizes and it is very unpractical for large vectors.

4.4 Design Pattern: The Delegate (n–ary Forwarder)

Overview Assume a process that continuously produces elements and sends them on a channel x. A Delegate is a process used to obtain a lot of elements while reducing the number of messages exchanged.

Example A process continuously produces electronic tickets and sends each one as a message on a channel x. A remote process requires a lots of tickets: since they are sent one–by–one on x the number of messages exchanged becomes very high. We want to design a ticket consumer that uses few messages.

Motivations `recv` operations on remote channels are executed using linear forwarders (as in the encoding in section 2.5). For each `recv` on a remote channel executed two messages are exchanged: the first one (from the receiver to the channel owner) is the linear forwarder and the second one (in the reverse direction) is a redirected output. The use of linear forwarders can be avoided by migrating the receiver process to the location of the remote channel. In this pattern a new process is started at the location of the remote channel and it is used as an n–ary forwarder.

Pattern Description The producer is executed at the location of x, the consumer is executed elsewhere. Channel x is used by the producer to send out tickets to consumers. No requests are necessary to obtain a ticket: they can be directly received on x. The consumer creates a new channel y and starts the execution of a delegate process at the location of x. The delegate process receives as many tickets as needed by the consumer, sends them to y and then terminates its execution.

Pattern Model The geometry of the network requested by the pattern is the following:

Location x type of x = `channel<ticketT>`

Location _ no channels are needed here

where `ticketT` is assumed to be the type of a ticket. The actors involved in the pattern are:

Producer Executed at x. Continuously produces tickets and sends them on x.

Consumer Not colocated with x. Creates the channel y and starts the execution of the Delegate at the location of x. Finally receives the tickets on y.

Delegate Executed at y. Receives tickes on x and sends them on y.

The exchange of messages between the actors is shown on figure 4.1.

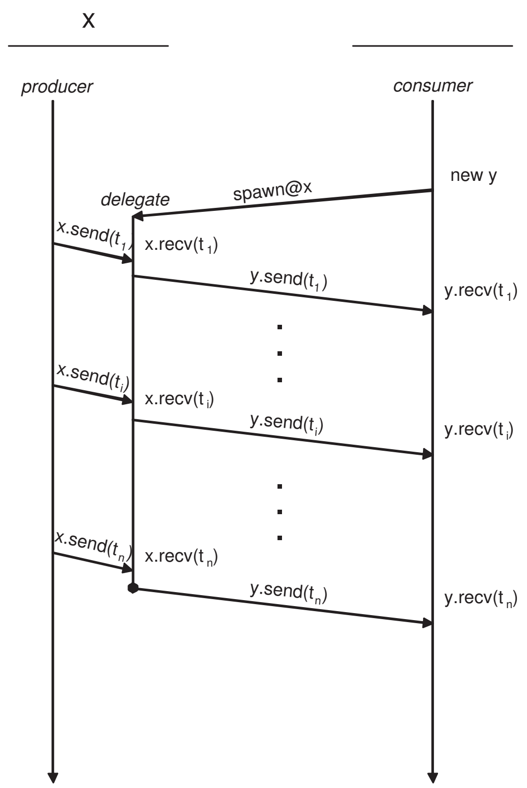

Figure 4.1: A message–based model of the delegate pattern

Implementation We implement Producer and Consumer as two different schedules. We assume x to be identified by `ch://channel.x`. We assume the colocation of the Producer with x (using the `colocatedwith` option on the schedule). The source code of the Producer, assuming `ticketT` to be the type of a ticket and `new_ticket()` to be a ticket constructor, is:

```
schedule Producer
colocatedwith ch://channel.x {

    // declaration of channel x
    channel<ticketT> x = ch://channel.x;

    // starts the production of tickets
    main {
        produce();
    }

    // produces tickets and sends them on x
    void produce() {
        x.send(new_ticket());
        spawn{ produce(); }
    }
}
schedule Consumer {

    // starts a 100-ary forwarder
    main {
        channel<ticketT> x = ch://channel.x;

        // the Delegate
        spawn@x{
            for i=1 to 100 {
                x.recv(ticketT t);
                y.send(t);
            }
        }
    }
}
```

Observations This design pattern shows how migration can be used to optimize communication. The Delegate is executed at the location of x: this implies the exchange of one big message for its migration, and one message for each ticket. Messages exchanged between the Producer and the Delegate are intra–location, so we don't count them. As an alternative, the use of `recv` on the remote channel x eliminates the initial big message but requires two messages for each query (due to linear forwarders).

The pattern can be slightly changed by adding ticket requests: in this case the Delegate must not only forward elements from x to y, it must also request all them one–by–one before forwarding.

4.5 Design Pattern: The Assistant (Dynamic Forwarder)

Overview A service–provider receives requests on a channel and handles them. When it is overloaded with requests, it asks an assistant server for help.

Example A search engine waits for user requests on a channel x. A request is a pair (`string`,`channel<string>`) where the first element is the keyword searched by the user and the second one is the channel for the engine's answer. The search requires time and resources: we want to design a server that handles user requests and asks for help to an assistant application when it is overloaded. We want to design the assistant application too.

Motivations As in the Delegate design pattern, our aim is to describe how the number of messages exchanged for remote `recv` operations can be reduced. As in the Delegate pattern with its n–ary forwarder, here we describe how a *dynamic* forwarder can be implemented. Dynamic means that it can be started and terminated when necessary.

Pattern Description The server process is executed at the location of x. Channels y and h and the assistant process are all at a different location. Channel x is used to receive user queries: when the load factor of the server process becomes greater than a specified threshold, an *help* message is sent to the assistant process on channel h. The assistant starts the execution of a new process at the location of x: this new process redirects messages from x to y. User queries redirected to y are handled by the assistant. When the load factor of the server decreases under a lower threshold, an *enough* message is sent to the assistant.

Pattern Model The geometry of the network requested by the pattern is made of two locations:

Location x type of x = `channel<string, channel<string>>`

Location yh type of h = `channel<string>`
type of y = `channel<string, channel<string>>`

The actors involved in the pattern are:

Server Executed at x. Receives user requests on x, executes the searches and replies to the users using the channels contained into the requests. Sends a *help* or *enough* messages on h when needed.

Assistant Executed at h. Waits for *help* messages on h: when one is received, creates a new channel z with type `channel<>`. It starts Forwarder at x and waits for user requests on y. When receives *enough* on h, sends a message on z and stops receiving user requests.

Forwarder Started at x by the Assistant. Receives user requests on z and sends them on y. Concurrently waits for a message on the remote channel z: when the message is received, terminates its execution.

The exchange of messages between the actors is shown in Figure 4.2.

Implementation We implement Server and Assistant as two different schedules. We assume channels x, y and h to be identified by: `ch://channel.x`, `ch://channel.y` and `ch://channel.h`. We require the colocation of Server with x and the colocation of Assistant with h and y (using the `colocatedwith` option on the schedules). The source code for Server is:

```
schedule Server
colocatedwith ch://channel.x {

    // channel declarations
    channel<string,channel<string>> x = ch://channel.x;
    channel<string> h = ch://channel.h;

    // mutable record for the state of the server
    // the first element is the load factor
    // the second element is 1 if the assistant
    // is helping the server, 0 otherwise
    channel<int,int> state = new channel<int,int>;
```

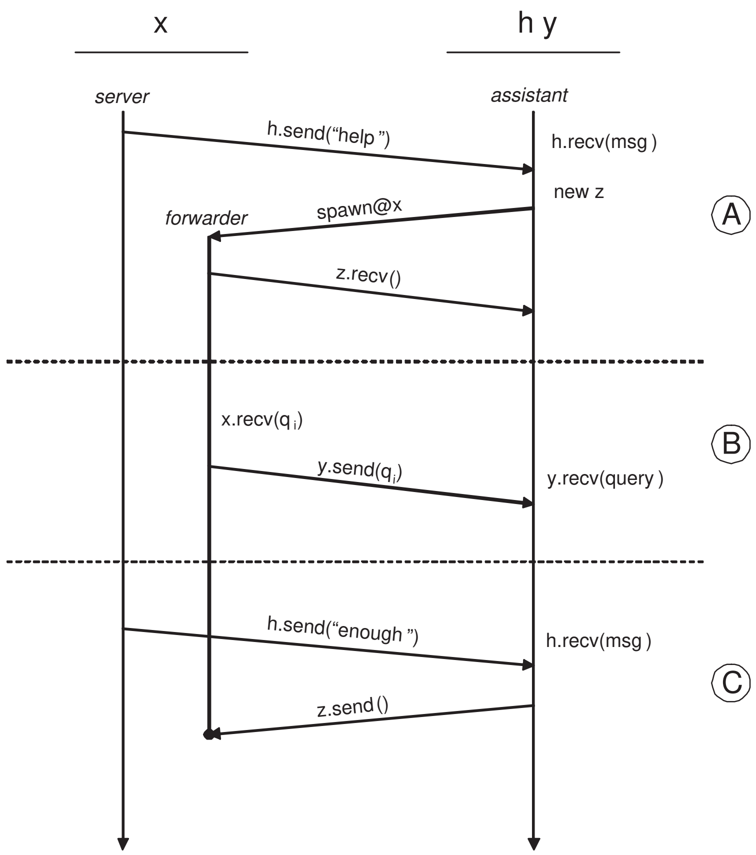

Figure 4.2: Messages used in the assistant pattern: (A) is the help request, (B) is the redirection of a user request, (C) is the end of the help

```
    // thresholds
    int lower_threshold = 100;
    int upper_threshold = 150;

    // initializes the state and starts the service
    main {
        state.asend(0,0);
        receive_queries();
    }

    // handles user requests
    void receive_queries() {
        // receive a new request and checks the state
        x.recv(string msg, channel<string> replyCh);
        state.recv(int served, int isHelped);

        // if help is needed sends a message to the assistant
        if ((served >= upper_threshold) && (!isHelped)) {
            h.send("help");
            state.asend(served+1, 1);
        }
        // otherwise only updates the state
        else {
            state.asend(served+1, isHelped);
        }

        // handle the request and waits a new one
        spawn { receive_queries(); }
        string result = handle(msg);
        replyCh.send(result);

        // if the load factor decreases stops the help
        state.recv(int served2, int isHelped2);
        if ((served2 <= lower_threshold) && (isHelped2)) {
            h.send("enough");
            state.asend(served2-1,0);
        }

        // otherwise updates the state
        else { state.asend(served2-1, isHelped2); }
    }

    // handles a user request and return the result
    string handle(string msg) {
        /*
         * not implemented
         */
    }
}
```

This is the Assistant schedule. It contains also the code of the Forwarder process:

```
schedule Assistant
colocatedwith ch://channel.y ch://channel.h {

    // channel declarations
```

```
channel<string,channel<string>> y = ch://channel.y;
channel<string,channel<string>> x = ch://channel.x;
channel<string> h = ch://channel.h;

// starts the help-message handler and the assistant
main {
    spawn { help(); }
    assist_queries();
}

// handles help and enough messages
void help() {
    // receives a message and checks the state
    h.recv(string msg);
    state.recv(int isHelped);

    // if help is needed
    if ((msg=="help") && (!isHelped)) {
        state.asend(1);
        channel<> z = new channel<>;

        // starts the Forwarder process at x
        spawn@x {
            channel<int> terminate = new channel<int>;
            terminate.asend(0);
            // handles the termination signal
            spawn {
                z.recv();
                terminate.recv(int oldval);
                terminate.send(1);
            }

            // in parallel redirects messages
            forwarder(terminate);
        }

        // handles a new message
        spawn { help(); }
    }

    // if help is enough
    else if ((msg=="enough") && (isHelped)) {

        // sends the termination signal
        z.send();
        state.asend(0);

        // handles a new message
        spawn{ help(); }
    }
}

// redirects messages from x to y
void forwarder(channel<int> terminate) {

    // looks for termination requests
```

```
        terminate.recv(int term);

        // if it must not terminate
        if (!term) {
            // receives a user request
            x.recv(string q, channel<string> r);

            // updates the mutable variable
            terminate.asend(term);
            spawn {forwarder(terminate);}

            // redirects the request
            y.send(q,r);
        }
    }

    // handles user request received on y
    void assist_queries() {

        // receives the (redirected) request
        y.recv(string msg, channel<string> replyCh);
        spawn { assist_queries(); }

        // handles the request
        string result = handle(msg);
        replyCh.send(result);

    }

    // handles a user request and returns the result
    string handle(string msg) {
        /*
         * not implemented
         */
    }
}
```

Observations This design pattern shows how a load–balancing protocol can be easily implemented in hipi. The design was influenced by considering the number of messages exchanged between locations.

The Forwarder process is executed at the location of x: this implies the exchange of one big message at the begin of the help (the migration of the Forwarder process) and one message for each user request. The possible alternative is to let the Assistant use directly `recv` on the remote channel x: this eliminates the initial big message but implies two messages for each query (due to the encoding in bytecode, these are linear forwarder and an output on a fresh channel).

The Forwarder starts a parallel process that waits until a termination signal is sent on z. Then the parallel process terminates the execution of the Forwarder by setting 1 on the mutable variable `terminate`. We recall that mutable variables are always used without race conditions due to the protocol of section 4.2, so their use for inter–process communications is safe.

4.6 Design Pattern: Proxy

Overview Specifies how to create a local surrogate of a service provided on a remote channel. The local surrogate can avoid the need for some accesses to the remote resource.

Example A text editor is used to read a file stored on a remote location. The remote file system provides a channel that can be used to ask for the whole text or for a specific line. For the sake of the simplicity we assume that the editor is used only for reading the file — modifications are not allowed. A request to the remote file system requires time: we want to design a process that prevents remote accesses when two subsequent requests for the same line of text are made.

Motivations Proxies are widely used on networks: we show how they can be designed in hipi.

Pattern Description This is the equivalent of the Proxy design pattern for object oriented programming (described in [GHJG94]), but in a distributed environment. The file system service containing the file to access is executed at the location of x and it receives requests for this file over this channel. A Proxy process is executed at the location of the text editor. It receives requests on a local channel y: if it can satisfy a request it sends back the result to the editor, otherwise it access the remote service. The requests received by the Proxy are exactly the same accepted by the file system service. The Proxy stores the last line received by the service: if the same line is requested again, no remote access is needed.

Pattern Model The geometry of the network requested by the pattern is the following:

Location x type of x = `channel<string,int,channel<string>`

Location y type of y = `channel<string,int,channel<string>`

The actors involved in the pattern are:

Editor Executed at y. Makes requests for the whole text or for a single line of a remote file accordingly to user necessities. Requests are sent to the Proxy on channel y.

FileSystem Executed at x. Receives requests on x and replies with the result.

Proxy Executed at y. Receives requests on y: if necessary forwards them to the FileSystem , otherwise (if they are repeated requests for the same line) it replies to them with the cached result.

Requests are triples (s,i,c) where s is `"text"` or `"line"` to represent a request for the whole text or a single line; i is the line number if s is `"line"` or it is ignored otherwise; c is the channel used for the reply.

The exchange of messages between the actors is shown in Figure 4.3

Implementation We implement Editor, Proxy and FileSystem as three different schedules. We assume x and y to be identified by `ch://channel.x` and `ch://channel.y`. We require the FileSystem to be colocated with x and of the Editor and the Proxy with y (using the colocatedwith option). The source code of the Editor is:

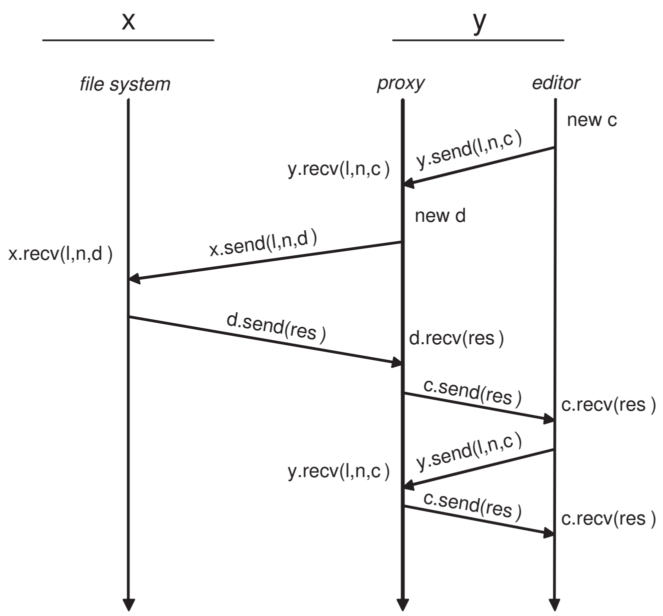

Figure 4.3: Messages used in the proxy pattern: here two subsequent requests for the same line are made, the second one is handled locally

```
schedule Editor
colocatedwith ch://channel.y {

    // simulates some user requests
    main {
        // declaration of channel y
        channel<string,int,channel<string>> y = ch://channel.y;

        // simulates the requests
        channel<string> c = new channel<string>;
        y.send("text",0,c);
        c.recv(string text);
        y.send("line",10,c);
        c.recv(string line10);
        y.send("line",35,c);
        c.recv(string line35);
        y.send("line",35,c);
        c.recv(string line35bis);
    }
}
```

The source code of the Proxy is:

```
schedule Proxy
colocatedwith ch://channel.y {

    // mutable variables used to store the last line
    channel<int> lastLineNum = new channel<int>;
    channel<string> lastLine = new channel<string>;
```

```
    // initializes lastLine and lastLineNum and starts the proxy
    main {
        lastLineNum.asend(-1);
        lastLine.asend("");
        proxy();
    }

    // handles user requests and forwards them to
    // the remote file system if necessary
    void proxy() {

        // receives a request
        y.recv(string s, int i, channel<string> c);

        // if the full texts is requested forward to
        // the file system
        if (s=="text")
            x.asend(s,i,c);
        // otherwise
        else {
            lastLineNum.recv(int linenum);
            lastLine.recv(string line);

            // if it is a repeated line request satisfies it
            if (linenum==i) {
                c.asend(line);
                lastLine.asend(line);
                lastLineNum.asend(linenum);
            }

            // otherwise accesses the remote file system,
            // stores a copy of the reply and forwards the
            // reply to the user
            else {
                channel<string> d = new channel<string>;
                x.send(s,i,d);
                d.recv(string newline);
                c.asend(newline);
                lastLine.asend(newline);
                lastLineNum.asend(i);
            }
        spawn{ proxy(); }
    }
}
```

Finally, the source code of the FileSystem, where the file access functions `get_text()` and `get_line()` are assumed, is:

```
schedule FileSystem
colocatedwith ch://channel.x {

    // declares channel x
    channel<string,int,channel<string>> x = ch://channel.x;

    // starts the service
    main { request_handler(); }
```

```
    // receives requests
    void request_handler() {
        x.recv(string s, int i, channel<string> c);
        spawn { request_handler(); }
        if (s=="text")
            c.send(get_text());
        else
            c.send(get_line(i));
    }
}
```

Observations This design pattern shows how a communications between locations can be reduced by storing data in a cache. Cache proxies are well–known and widely–used entities on networks.

4.7 Example: Concurrent Dictionaries

This program is a complete example of a distributed Italian-English dictionary. It is made of two schedules: the first one — OnLineDict — represents a dictionary server: when it is executed it loads its database of words and waits for requests on a global channel (3 global channels are provided). The second schedule — DictClient — asks the user for a word in Italian or English, then sends a request on all the three global channels. The first answer received is shown to the user; the other two are discarded.

```
typedef DictChannel = channel<string,string,channel<string,string>>;
import "hipi_lib/SVector.hp";

// Server schedule
schedule OnlineDict {

    // these are the global channels where requests are received
    DictChannel netDict1 = "http://italian_to_english.com/dict";
    DictChannel netDict2 = "http://the_translator.com/ita_eng";
    DictChannel netDict3 = "http://freetranslations.org/ita_eng";

    // channels for console interaction
    channel<string> sconsole = "sim://string.console";
    channel<DictChannel> cconsole = "sim://channel.console";

    // inserts a pair of words into dictionaries
    void insertDict(SVector italian, string ita,
                    SVector english, string eng) {
        int size = SVector_capacity(italian);
        SVector_add(italian,size,ita);
        SVector_add(english,size,eng);
    }

    // translates a word if present on the dictionary
    // return an error message otherwise
    string translate(SVector source, string word, SVector dest) {
        int idx = SVector_indexOf(source, word);
        SVectorInfo info = SVector_get(dest, idx);
        string retVal = SVectorInfo_elem(info);
```

```
    if (retVal!="")
        return retVal;
    else
        return "Word not found";
}

// accepts a request on a channel
// (terminates if word is __TERMINATE__)
void acceptReq(string serviceID, DictChannel dictChan,
                SVector italian, SVector english) {
    dictChan.recv(string lang, string word,
                    channel<string,string> reply);
    if (word!="__TERMINATE__") {
        spawn { acceptReq(serviceID, dictChan, italian, english); }

        // selects the language
        if (lang=="italian") {
            string trans = translate(italian, word, english);
            reply.send(trans,serviceID);
        }
        else if (lang=="english") {
                string trans = translate(english, word, italian);
                reply.send(trans,serviceID);
        }
        else
            reply.send("language not supported",serviceID);
    }
}

main {

    // schedule initialization
    sconsole.send("Insert a nick name for this dictonary");
    sconsole.recv(string nick);
    sconsole.send("Insert the URI where requests will be received:");
    sconsole.send("   http://italian_to_english.com/dict");
    sconsole.send("   http://the_translator.com/ita_eng");
    sconsole.send("   http://freetranslations.org/ita_eng");
    cconsole.recv(DictChannel dictChan);

    spawn@dictChan {
        // italian and english words (they are created here in
        // order to colocate them with dictChan)
        SVector italian = new_SVector();
        SVector english = new_SVector();

        // dictionaries initialization
        insertDict(italian, "rosso", english, "red");
        insertDict(italian, "verde", english, "green");
        insertDict(italian, "nero", english, "black");
        insertDict(italian, "giallo", english, "yellow");
        insertDict(italian, "rosa", english, "pink");

        // service start
        acceptReq(nick,dictChan,italian,english);
    }
```

```
    }
}

// Client schedule
schedule DictClient{

    // channel for console interaction
    channel<string> sconsole = "sim://string.console";

    void makeReq() {
        // start query
        sconsole.send("Insert source language:");
        sconsole.recv(string srcLang);
        sconsole.send("Insert word:");
        sconsole.recv(string word);

        // channel for return values
        channel<string,string> values = new channel<string,string>;

        // parallel search
        spawn { netDict1.send(srcLang, word, values); }
        spawn { netDict2.send(srcLang, word, values); }
        spawn { netDict3.send(srcLang, word, values); }

        // return the first result received
        values.recv(string trans, string id);

        // discard other results received
        spawn {
            values.recv(string id2, string trans2);
            values.recv(string id3, string trans3);
        }
        sconsole.send("The translation is");
        sconsole.send(trans);
        sconsole.send("The answer was received by");
        sconsole.send(id);

        // ask if a new search is needed
        sconsole.send("Would you like to continue (y/n) ?");
        sconsole.recv(string ans);
        if ((ans=="y")||(ans=="Y")||(ans=="yes")
            ||(ans=="YES")||(ans=="Yes"))
            spawn { makeReq(); } // start new search
    }

    main {
        makeReq();
    }
}
```

Chapter 5

Conclusions

The π–calculus is a widely used process algebra for reasoning about the behavior and the properties of parallel processes. The use of the π–calculus with an implicit definition of location allows us to reason about distributed processes. Finally, the definition of a programming language based on such a located calculus, allows us to write distributed programs whose properties can be discussed using the theories of the calculus.

Hipi has been introduced here as a programming language based on of the π–calculus. Our work concerned: (1) the definition of the hipi core language, (2) the definition of a bytecode language that is proper for a network implementation, (3) the definition and the implementation of the compiler from hipi into bytecode and (4) the discussion on how to efficiently use hipi for network aware programming.

The implementation of the hipi compiler followed the formal definition we gave of it. This allowed us to prove its correctness. We implemented also a network simulator that was used to execute compiled programs.

With hipi we experimented distributed programming. We studied which idioms are common using a π–like programming language and we have seen that network entities like n-ary and dynamic forwarders can be easily implemented. Hipi lacks a complete type system, the choice operator of the π–calculus and some features that can be useful in distributed programming. We leave all these as future works.

5.1 Future Works

What follows describes the main extension that hipi requires in order to become a complete distributed programming language.

Data Types The actual type system of hipi is too simple: it should be replaced with a type system comprehensive of structured data types. Our expectation is to integrate the XML data types of XDuce [HP03] in hipi. This integration is the aim of XRel [Bis03], that is a tool for XML processing inspired by XDuce. It will be merged with hipi and this will allow us to use data structures in programs and to send and receive XML documents over channels.

Choice Operator Hipi lacks the choice operator of the π–calculus. This operator should be added to the language after having studied well its impact on the run–time environment and the possible problems that could arise from its use in distributed programs.

Shared Channels A feature that we expect to add in the future is shared channel definitions. A shared channel is a channel that can be used from multiple schedules to coordinate their work, but that is invisible to the rest of the word. Shared channels allow programs to become modular. This could be a useful feature addresses to orchestration of distributed application.

5.2 Ringraziamenti

Ringrazio innanzitutto il prof.Cosimo Laneve per avermi motivato a scegliere questo argomento di tesi e avermi dato fiducia. Ringrazio tantissimo Lucian Wischik per aver seguito il mio lavoro passo dopo passo, correggendomi negli errori e motivandomi quando necessario. Lo ringrazio anche per tutto quello che ho imparato da lui in questi mesi e per la disponibilità che ha sempre dimostrato nei miei confronti. Infine ringrazio Fabrizio Bisi, Manuel Mazzara e Samuele Carpineti, con i quali ho avuto un sacco di interessanti conversazioni che si sono rivelate molto utili per il mio lavoro.

Appendix A

XML-Schema of XIL

Here we illustrate the XML-Schema of the XML Intermediate Language described in section 3.3. This schema can be used to validate bytecode files generated by the hipi compiler.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="schedule">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="unbounded" minOccurs="0"
                name="thread" type="ThreadType"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:complexType>
        <xsd:key name="threadName">
            <xsd:selector xpath="thread"/>
            <xsd:field xpath="@name"/>
        </xsd:key>
        <xsd:keyref name="threadRef" refer="threadName">
            <xsd:selector xpath="statements/spawn"/>
            <xsd:field xpath="@thread"/>
        </xsd:keyref>
    </xsd:element>

    <xsd:complexType name="ThreadType">
        <xsd:sequence>
            <xsd:element name="stacksize" type="StackSizeType"/>
            <xsd:group maxOccurs="unbounded" minOccurs="0"
            ref="statements"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:ID" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="StackSizeType">
        <xsd:attribute name="init" use="required"
        type="xsd:nonNegativeInteger"/>
    </xsd:complexType>

    <xsd:group name="statements">
        <xsd:choice>
            <xsd:element name="send" type="SendStmType"/>
            <xsd:element name="recv" type="RecvStmType"/>
```

```
        <xsd:element name="fwd" type="FwdStmType"/>
        <xsd:element name="store" type="StoreStmType"/>
        <xsd:element name="spawn" type="SpawnStmType"/>
        <xsd:element name="newch" type="NewChStmType"/>
        <xsd:element name="if" type="IfStmType"/>
        <xsd:element name="terminate">
            <xsd:complexType/>
        </xsd:element>
    </xsd:choice>
</xsd:group>

<xsd:complexType name="SendStmType">
    <xsd:sequence>
        <xsd:element name="load" type="LoadExpType"/>
        <xsd:group minOccurs="0" maxOccurs="unbounded"
        ref="expressions"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="RecvStmType">
    <xsd:sequence>
        <xsd:element name="load" type="LoadExpType"/>
        <xsd:element name="store" minOccurs="0"
        maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:attribute name="idx"
                type="xsd:nonNegativeInteger" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="FwdStmType">
    <xsd:sequence>
        <xsd:element name="load" minOccurs="2" maxOccurs="2"
        type="LoadExpType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="SpawnStmType">
<xsd:sequence>
    <xsd:group minOccurs="0" maxOccurs="unbounded"
    ref="expressions"/>
<xsd:sequence>
    <xsd:attribute name="thread" type="xsd:IDREF"
    use="required"/>
    <xsd:attribute name="dest" type="srcType"/>
</xsd:complexType>

<xsd:complexType name="StoreStmType">
    <xsd:sequence>
        <xsd:group ref="expressions"/>
    </xsd:sequence>
    <xsd:attribute name="idx" type="xsd:nonNegativeInteger"
    use="required"/>
</xsd:complexType>
```

```
<xsd:complexType name="NewChStmType">
    <xsd:attribute name="idx" type="xsd:nonNegativeInteger"
    use="required"/>
</xsd:complexType>

<xsd:complexType name="IfStmType">
    <xsd:sequence>
        <xsd:group ref="booleanExp"/>
        <xsd:element name="then">
            <xsd:complexType>
                <xsd:group minOccurs="0" maxOccurs="unbounded"
                ref="statements"/>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="else">
            <xsd:complexType>
                <xsd:group minOccurs="0" maxOccurs="unbounded"
                ref="statements"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:group name="expressions">
    <xsd:choice>
        <xsd:element name="load" type="LoadExpType"/>
        <xsd:element name="int" type="xsd:integer"/>
        <xsd:element name="string" type="xsd:string"/>
        <xsd:element name="newch">
            <xsd:complexType/>
        </xsd:element>
        <xsd:element name="op" type="OpType"/>
    </xsd:choice>
</xsd:group>

<xsd:group name="booleanExp">
    <xsd:choice>
        <xsd:element name="load" type="LoadExpType"/>
        <xsd:element name="int" type="xsd:integer"/>
        <xsd:element name="op" type="OpType"/>
    </xsd:choice>
</xsd:group>

<xsd:complexType name="LoadExpType">
    <xsd:attribute name="src" type="srcType"/>
</xsd:complexType>

<xsd:simpleType name="srcType">
    <xsd:union memberTypes="xsd:nonNegativeInteger xsd:anyURI"/>
</xsd:simpleType>

<xsd:complexType name="OpType">
    <xsd:sequence>
        <xsd:group maxOccurs="2" minOccurs="1"
        ref="expressions"/>
```

```
        </xsd:sequence>
        <xsd:attribute name="type" use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="add"/>
                    <xsd:enumeration value="sub"/>
                    <xsd:enumeration value="mul"/>
                    <xsd:enumeration value="div"/>
                    <xsd:enumeration value="mod"/>
                    <xsd:enumeration value="and"/>
                    <xsd:enumeration value="or"/>
                    <xsd:enumeration value="eq"/>
                    <xsd:enumeration value="neq"/>
                    <xsd:enumeration value="lt"/>
                    <xsd:enumeration value="gt"/>
                    <xsd:enumeration value="le"/>
                    <xsd:enumeration value="ge"/>
                    <xsd:enumeration value="umin"/>
                    <xsd:enumeration value="not"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>

</xsd:schema>
```

Bibliography

[Ama00] Roberto M. Amadio. On modelling mobility. *Theoretical Computer Science*, 240(1):147–176, June 2000.

[Bis03] Fabrizio Bisi. XRel: XML Processing Through Regular Expressions, December 2003. Masters Thesis Project — http://www.cs.unibo.it/∼bisi/thesis.html.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, August 1998. ftp://ftp.math.utah.edu/pub/rfc/rfc2396.txt, ftp://ftp.internic.net/rfc/rfc2396.txt.

[CF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, October 1999.

[Con02] W3C World Wide Web Consortium. Web services activity web site, 2002. http://www.w3.org/2002/ws/.

[Con03] W3C World Wide Web Consortium. SOAP Version 1.2 Part 0: Primer, June 2003. http://www.w3.org/TR/soap12-part0/.

[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.

[FGM+99] Robert Fielding, Jim Gettys, Jeff Mogul, Henrik Frystyk, L. Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999. http://www.ietf.org/rfc/rfc2616.txt.

[Fou] The Apache Software Foundation. Xerces: Xml parser for java. http://xml.apache.org.

[GHJG94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (GoF). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, October 1994. ISBN: 0-201-63361-2.

[GLW02] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. The fusion machine (extended abstract). In L. Brim, P. Jancar, and M. Kretinsky, editors, *CONCUR 2002*, volume 2421 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2002.

[GLW03] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. Linear forwarders. In R. Amadio and D. Lugiez, editors, *CONCUR 2003*, volume 2761 of *Lecture Notes in Computer Science*, pages 415–430. Springer-Verlag, 2003.

[GW00] Philippa Gardner and Lucian Wischik. Explicit fusions. In Mogens Nielsen and Branislav Rovan, editors, *MFCS 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 373–382. Springer-Verlag, 2000. Full version to appear in TCS.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[HR98] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Proceedings of HLCL '98: High-Level Concurrent Languages*, number 16.3 in Electronic Notes in Theoretical Computer Science, pages 3–17. Elsevier, 1998.

[Joi02] The join calculus language, 2002. Implementations available from: http://pauillac.inria.fr/join/unix/eng.htm.

[Kre01] Heather Kreger. Web services conceptual architecture (WSCA 1.0). *IBM*, May 2001.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–40, 41–77, 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623, pages 685–695, 1992.

[MS98] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Lecture Notes in Computer Science*, 1443:856–872, 1998.

[Per03] Enrico Persiani. Progettazione e implementazione di una macchina per fusioni distribuite. Masters thesis in computer science, University of Bologna, 2003.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[PV98] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*, pages 176–185. IEEE, Computer Society Press, July 1998.

[Ste93] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1993.

[SW01] Davide Sangiorgi and David Walker. *The π–calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001. ISBN: 0-521-78177-9.

[Wis01] Lucian Wischik. *Explicit Fusions: Theory and Implementation*. PhD thesis, Computer Laboratory, University of Cambridge, 2001.

[WS00] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency. The Computer Society's Systems Magazine*, 8(2):42–52, April-June 2000.