ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

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

Importing Ownership Types into the Join Calculus

Tesi di Laurea in
Sistemi Operativi

Relatore:
Chiar.mo Prof.
Sangiorgi Davide

Co-Relatore:
Chiar.mo Prof.
Clarke Dave

Presentata da:
Patrignani Marco

Sessione I
Anno Accademico 2009-2010

Contents

List of Figures

Chapter 1

Introduction

Abstract

Gli ownership types sono un sistema di tipo [Pie02, Kob02, Car96] sviluppato nel mondo object-oriented per risolvere problemi come aliasing e object encapsulation.

Il join calculus è un modello formale per il calcolo di processi basato sulla nozione di mobilità.

Questa tesi presenta lo sviluppo di tre type systems per il join calculus. I primi due formalizzano l'idea di ownership types importata dal mondo object-oriented. Il terzo type system formalizza l'idea dei gruppi, un costrutto ideato per il π-calculus che trasportiamo nel join calculus. Per ogni type system vengono fornite regole di tipaggio e dimostrazione di subject reduction e no runtime errors.

Viene infine presentato un confronto tra i tre sistemi che ne dimostra l'equivalenza: i gruppi forniscono le stesse proprietà degli ownership types.

On one hand we have object-oriented programming, on the other we have process algebra . Despite first impressions, the two areas have a lot in common and they are semantically closer than one would expect.

Works in object-oriented programming led to the development of a particular type system called *ownership types* [CPN98, Cla01, CD02]. Such type systems have been used to manage aliasing, enforce object encapsulation and provide other useful properties to programs.

Work on distributed formal systems gave birth to the join calculus, an ML-flavoured calculus for mobile processes [FG96, FG00]. The join calculus has the same expressive power of the well known π-calculus but it also has an interesting and very powerful notion of locality. This notion made possible the creation of a concurrent programming language whose formal foundations rely on the join calculus: JOCaml [FFMS02].

In this thesis we import the notion of ownership types into the join calculus. We develop two approaches for this fusion, a channel-driven and a context-driven approach. We define typing rules for both systems and prove subject reduction and no runtime errors theorems. Then we import the notion of groups [CGG05] from the π-calculus to the join calculus developing a sound type system with strong security properties. This is the first attempt of such a translation and it provides a system to compare the other ones we created with.

Finally we show that the three systems are equivalent, groups provide the same benefits of ownership types.

Chapter 2

Ownership types

Abstract

In questo capitolo vengono presentati gli ownership types come modello per il mondo della programmazione orientata agli oggetti. Tale modello fornisce il linguaggio di una nozione esplicita di incapsulamento permettendo quindi una migliore gestione dell'aliasing e diversi altri benefici che spaziano dai settori dell' ottimizzazione del codice, alla garbage collection, all'interpretazione astratta. Dopo una panoramica sulle proprietà del sistema, si forniscono esempi che esplicano i benefici apportati dagli ownership types.

Ownership types are a type system developed for object-oriented programming languages. They provide a statically enforceable notion of object-level encapsulation. Ownership types have been developed as an alias protection mechanism in first place, but the further benefits they give have made them an interesting type system.

In object-oriented programming the main evaluation strategy is *call-by-reference*: a function receives a reference to its arguments, rather than a copy of its value. The benefits are multiple, we have greater time- and space-efficiency since arguments do not need to be copied, as well as the potential for greater communication between a function and its caller since the function can return information using its reference arguments. Of course not all that glitters is gold, as a matter of fact in addition to these gains some problems arise such as *aliasing*.

When two objects refer to another we have aliasing. An example can be seen in the code in Figure 2.1 Aliasing can be a serious threat for some aspects of system development, implementation and optimization. Both *debugging* and *maintaining* a piece of software require knowledge of which parts of the object graph are aliased and then accessed by the code we are observing. When *optimizing code*, compilers must check possible aliasing in the analyzed fragment in order to optimize without changing the program's semantics, and this can take a long time to be determined. *Garbage collection* can be improved when we know there is no alias in a certain block of code. Also *modular reasoning* [CD02] is possible when we have control of aliasing.

```java
Student std = new Student();
Student alias = new Student();
alias.setName("Wedge");
alias = std;
std.setName("Biggs");
System.out.println(alias.getName());
    // will display Biggs instead of Wedge.
```

Figure 2.1: An example of aliasing in a Java program.

Different techniques have been formulated through the years to solve the problem of aliasing, giving birth to ownership types as a stand-alone theory for alias management.[1]

2.1 State of the art

Ownership types were first presented in [CPN98] as enforcing the *owners as dominators* property over the program's object graph. This means that if an object l owns another object l', then any path from the root of the object graph to l' has to go through l, thus l dominates l'. This model was refined in [Cla01] and was used also in [Boy04].

Among the drawbacks of the ownership types system there is the inability to create iterators. Such a lack does not allow iterators for collections to access the internal representation of the collection object. To solve this problem different approaches have been taken. Another notion of ownership has been provided: *shallow ownership*, that loosens the constraints imposed by *deep ownership.* Nonetheless the lead authors preferred to address this problem by allowing local variables [CD02] or inner classes [BLS03] in order to have a less restrictive access while maintaining strong invariants on the system.

In [Wri06] ownership types were extended with owner-polymorphism and external uniqueness [CW03]. Finally in [Pot07] ownership types were combined with generics to achieve a more powerful type system.

A language called *Safe Java* [Boy04] makes use of ownership types to detect data races and deadlocks in object-oriented programming and allows safe persistent storage of objects [BLR02]. It also allows safe region-based memory management in real time computation [BSBR03], thanks to the ownership types system. These works benefit from the assumptions one can make both on the structure imposed by the owners as dominators property on the objects graph and on the references between objects. When locking an object l in a concurrent setting, with ownership one can be sure that the objects owned by l will not have any additional reference other than l's one.

Ownership types provide benefits that match perfectly with the requirements of *ef-*

[1]A somewhat dated list of such techniques can be found in [Cla01].

fects. *Effects* are annotation that make explicit the behavior of a certain piece of code thus allowing reasoning about what that code does. Both [CD02, Boy04] use the enforcements of ownership constraints to improve reasoning about programs and provide a language with both ownership and effects annotations.

2.2 Ownership types

We shall now present some more detailed insights of the ownership types system. The syntactical annotation of owners have varied through the papers. The notation used here comes from [Pot07, Boy04]. Owners are annotated as parameters between angle brackets "`<,>`", the first parameter is the object's owner while the others are used to carry ownership information. There we have keywords such as `world` and `This` to denote the owner being the root object and the owner being the current active object respectively. An example of such notation can be found in Figure 2.2.

```
class ANewClass<classOwner, paramOwner>{
    public Object<This> field;
        //an enclosed object.
        //The owner is the current active object
    public Object<world> shared;        //a shared object
    public Object<classOwner> sibling ;
        //an object whose owner is the same of
        //the current object's
}
```

Figure 2.2: Example of the notation we will use for ownership.

2.2.1 The owners as dominators model

To model the notion of *ownership* in a type system the following assumptions are always made: an object's owner is fixed for its lifetime, and every object has a single owner that is another object or the root of the system.

To generalize the model, the coupling between objects and owners is broken. *Contexts* represent the unit of ownership, they have been introduced to separate such ownership from objects. Each object has an *owner context* which abstractly represents its owner. Objects with the same context as owner are considered to have the same owner. Each object also has a *representation context* which is considered to be the owner of its representation.

Contexts form a partial order $(C, \prec:)$ that may be derived from a tree, a forest or a dag and it is created as follows. $C_0 = (\mathit{world}, \mathit{world} \prec: \mathit{world})$ identifies the presence of a root element called *world*. To define the introduction of a context c owned by another context o in the partial order $(C_i, \prec:_i)$ we write $C_{i+1} = (C_i \cup \{c\}, \prec:_i \cup\, c \prec: o)$.

The relation $\prec:$ is called *inside*. The aforementioned contexts associated to an object (*owner* and *representation*) are indicated with two functions: `owner` and `rep`. The containment invariant determines when an object can refer to another. It states:

$$\iota \rightarrow \iota' \Rightarrow \mathtt{rep}(\iota) \prec: \mathtt{owner}(\iota')$$

where $\iota \rightarrow \iota'$ means that ι *refers to* ι'. Another requirement is that $\mathtt{rep}(\iota) \prec: \mathtt{owner}(\iota)$ so that an object can access itself.

2.2.2 A diagrammatic notation

For a better understanding of the properties enforced by the ownership types system we now present a diagrammatic example that shows which references are allowed and which are not in a deep ownership setting. First of all we introduce a nest of objects. A square represents an object identity, it has a name on the inside. A rounded rectangle represents a context. An object's context `rep` is the one the objects stands on top of, while `owner` is the context immediately enclosing the object.

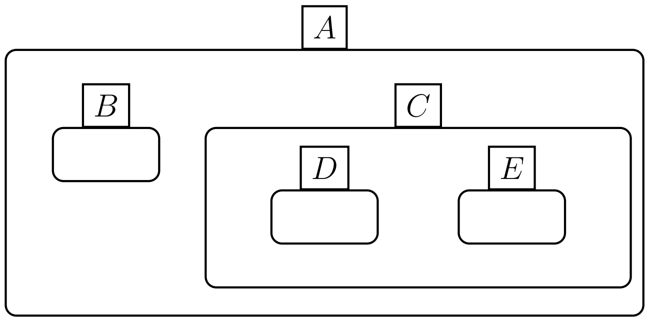

Figure 2.3: A nest of objects and contexts.

Figure 2.3 shows the system associated to the contexts set $\mathcal{C} = \{\mathit{world}, A, B, C, D, E\}$ and the inside relation given by $B, C \prec: A \prec: \mathit{world}$ and $D, E \prec: C$. The owner and representation contexts for each object are shown in Figure 2.4. Figure 2.5 points out which references are possible and which are not. A dashed line represents an invalid reference, a full line is a valid one. We do not show all the possible references but generally whether a reference is allowed to an object, it is also to a sibling of such object, unless it breaks the invariant.

Object	`owner`	`rep`
A	*world*	A
B	A	B
C	A	C
D	C	D
E	C	E

Figure 2.4: Contexts associated to the objects of Figure 2.3 and Figure 2.5

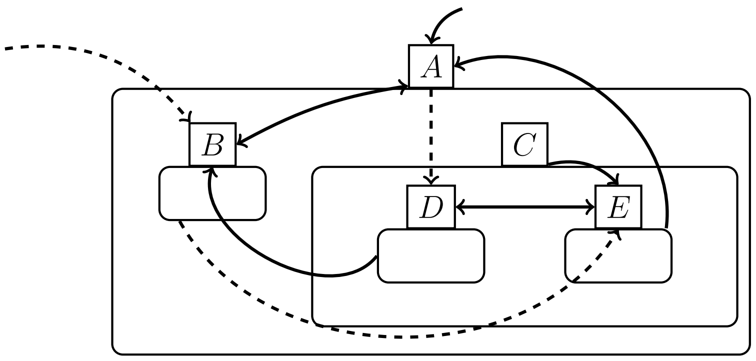

Figure 2.5: Valid and invalid references in a nest of objects and contexts.

2.3 Benefits of ownership typing

Having a clearer vision of the concepts of the ownership typing disciple, it should be easier to understand the benefits of such type system.

Among the benefits given by ownership types we reported the ability to reason about programs. Such reasoning in object-oriented programs involves reasoning about objects belonging to a class. This is possible with ownership types since this type system enforces full *object encapsulation*. An object o should own all the subobjects it depends on, that is all the objects o calls methods of that affect the invariant of o. Full encapsulation guarantees that outer objects must access o in order to interact with the subobjects o depends on. This prevents outer objects from violating the class' invariant, which is in fact handled within the class only.

Nevertheless when reasoning about a program, aliasing is a big threat. References to an object can be made in every part of the program, a statical analyzer must work hard to understand which references are aliased and which are not in large programs. Since a reference to an object can come only through its parent or from its descendant, aliasing checking requires much less effort with ownership types.

In addition to that, memory management can be improved by using ownership types.

Garbage collection can be sped up by the partitioning of the objects graph. Such a graph being a tree allows deallocating a whole subtree as soon as its root becomes unavailable. Since all references to an object pass through its owner, once the owner can be collected, so all of its owned objects can as well.

2.4 Examples

Let us now present an example of the type system at work. Next comes a code example that benefits from the use of ownership types. The gains are extremely high considering the very low syntactic overhead compared to an ownership-free system.

In this example [Pot07] we use ownership types to prevent *representation exposure.* The programmer defines the class invariant and then he (or his colleagues) will not be able to break it, either on purpose and accidentally.

Example 1 (No representation exposure)**.** *The code in Figure 2.6 shows two classes: a* `Point` *and a* `Rectangle`*. A* `Point` *consists of two integers representing the point's coordinates. A* `Rectangle` *consists of two private points representing the two extremes: the top left and bottom right points. The two extremes represent the class invariant, we want them to be protected from the outside, thus these two points declare* `This` *as owner. This means that a rectangle instantiation has its own private points, no outer reference to these objects are allowed. The method* `doIt()` *inside the rectangle class tries to expose the private fields via the* `exposeUpperLeft()` *method and fails when the receiver of the call is not explicitly* `this`*. We can see that an outer class willing to get the* `upperLeft` *point must call the* `getUpperLeft()` *method and it will receive a copy of the point. This gives the outer class the knowledge about such a point but does not allow any modification to the outer class' point to be reflected on the rectangle class invariant.*

```
class Point<owner>{
    Integer x; Integer y;
    Point(Integer x, Integer y){
        this.x=x; this.y=y;
    }
}

class Rectangle<Owner>{
    private Point<This> upperLeft;
    private Point<This> lowerRight;

    public Rectangle(Point<Owner> ul, Point<owner> lr){
        upperLeft= new Point<This>(ul.x, ul.y);
        lowerRight= new Point<This>(lr.x, lr.y);
        //assignment upperLeft=ul is illegal even in Java
        //due to incompatible type parameters This and owner
    }

    public void doIt(){
        Point<This> p;
        p=this.upperLeft;
        p=this.exposeUpperLeft();
        Rectangle<Owner> ro = this;
        p= ro.upperLeft;
            //wrong with ownership types (not in Java)
        p = ro.exposeUpperLeft();
            //wrong with ownership types (not in Java)
    }

    //the following method can be called only from the
    //inside  of the istance itself: this.exposeUpperLeft()
    public Point<This> esposeUpperLeft(){
        return upperLeft;
    }

    public Point<Owner> getUpperLeft(){
        return upperLeft;
        //wrong both in Java and ownership types
        return new Point<Owner> (upperLeft.x, upperLeft,y)
    }
}
```

Figure 2.6: Rectangle class using ownership types.

Chapter 3

The join calculus

Abstract

In questo capitolo viene fatta una breve panoramica sull'algebra dei processi per poi andare a trattare un calcolo specifico: il join calculus. Si forniscono quindi la sintassi e la semantica classica per tale linguaggio. Successivamente si presentano alcuni esempi che mostrano la potenza espressiva del join calculus ed i risultati più significativi per il calcolo in questione.

Process algebra (or process calculus) indicates a family of different approaches to formally modelling concurrent systems [Bae05].

Before going any further we ought to answer the question: what is a concurrent system?

Concurrent systems (or reactive systems) are systems whose parts:

- can accomplish their work in parallel; and
- can communicate with each other.

We are not necessarily talking about computer systems here, any model constituted of distinguished parts that work separately from each one other represent a concurrent system. But there is also another key aspect other than parallelism: communication. A part of the system may have the need to share its results to another one, thus creating a communication channel where information flows from a side to another.

An example of concurrent system is the human body, both in its macro and micro characterizations. A brain and a heart are two well distinguished organs whose works run in parallel. Blood exchange can be seen as a communication happening between the two ends, blood is pumped from the heart via the veins in order to keep the brain sprinkled.

A micro characterization of concurrent systems is a cell group. A single cell is a stand alone entity whose life is independent from the lives of its neighbors, and cells can communicate. Molecules can be passed via the membranes to activate the receiver's receptors and start the communication.

As we can see the whole world built around us is a reactive system made of smaller systems, the need for a better understanding of the world itself gave birth to the very subject.

The aim of process algebra is to provide high level descriptions of how different systems and their subparts interact and evolve. Generally the word *process* is used to define a component of a designed system.

The calculus provides tools for process definition and manipulation as well as instruments for process analysis and formal reasoning. As we already pointed out, the calculus has an explicit notion of communication since it is a key idea of the subject.

Process definition and manipulation are tools that allow us to point out what the real clue is: modelling behaviors. We describe certain aspects of a process' behavior, disregarding other aspects, so we are considering an abstraction or idealization of the 'real' behavior.

The word *algebra* denotes that we take an algebraic/axiomatic approach to talking about behavior. That is, the discipline uses the methods and techniques of universal algebra. This allows the calculus to have analysis and formal reasoning tools to fulfill its usefulness.

A process example could be a brain definition. We can define a brain as a process and model its behavior as: wait for blood, produce thoughts. We can then create a heart process that produces blood continuously. A combination of both heart and brain would result in a system whose two processes can communicate. This communication would transfer blood from the heart to the brain thus generating thoughts.

The first examples of such calculi are CCS (Calculus of Communicating Systems)[AILS07], CSP (Communicating Sequential Processes)[Hoa78] and ACP (Algebra of Communicating Processes)[BK89]. These languages have pioneered the area giving great results but also showing limitations in the expressiveness of the processes they could model.

Research on networks of processes whose processes are mobile and the configuration of communication links is dynamic gave birth to a language that is widely considered the heir and evolution of CCS: the π-calculus [Mil99, Mil92].

Many variants of π-calculus have been invented through the years, thus testifying the great versatility of the calculus. Amidst its most famous creations we see: the spi-calculus, a calculus for cryptographic protocols [AG99]; π-calculus for biomolecular systems [RPS+04]; and so forth. Each variation is a simple extension of the strong core of the π-calculus aimed to solve a difficulty in defining concurrent systems [PS96].

From a whole different family, yet retaining several common points with the π-calculus, comes the join calculus, an ML-flavoured language for modelling and analyzing reactive systems. The following section presents its formal semantics.

3.1 Join calculus

Join calculus [FG96, FG00] was developed to bridge the gap between calculi for concurrent processes and languages for programming distributed and mobile systems. A new model of concurrency was singled out in the chemical abstract machine (CHAM) firstly presented in [BB92]. A CHAM is a solution comprising a multiset of reaction patterns $\mathcal{R}$ and a multiset of molecules $\mathcal{M}$. Molecules are sorted, matched to a particular reaction pattern in order to trigger reactions and make the solution evolve. This is how computation happens in a CHAM.

Most process calculi rely on channels as an abstraction to the communication media for the exchange of data. Implementing such channels in both synchronous and asynchronous settings can be very difficult. The reflexive CHAM provides a model where we have a large number of sites where simple reactions can be triggered. As we can see this is an implementable distributed system with a language to model processes for it: the join calculus. We can therefore say that the join calculus is simply the syntactic description of such reflexive CHAM.

This language takes ideas from the π-calculus except that it combines several operators from π-calculus into a single receptor. This gives the language a locality property that will be discussed later on. The main theorem in [FG96] states that both join calculus and π-calculus have the same expressive power up to weak barbed congruence, this is achieved by exhibiting fully abstract encodings in each direction.

Next comes the syntax definition, then the semantics specified as a reflexive CHAM. We then introduce some examples for a better understanding of the calculus and conclude by discussing the most notable related work for the join calculus.

3.1.1 Syntax

Values in a CHAM are only names, as this is the case in the π-calculus. Let $\mathcal{N}$ be an infinite set of names. We use name variables in lowercase letters $x \in \mathcal{N}$ to denote its elements. The notation $\overrightarrow{y}$ indicates a tuple of name variables $y_1, y_2, \ldots, y_n$.

The following grammar defines processes, definitions and join patterns.

A process P is defined as follows:

$P = \emptyset$	null process;
$\mid\ P \mid P$	parallel composition;
$\mid\ x\langle\overrightarrow{y}\rangle$	emission of an asynchronous polyadic message. Channels $y_1, \ldots, y_n$ are sent on channel x;
$\mid$ **def** D **in** P	process definition.

A definition D is defined as follows:

$D = D \wedge D$	definitions conjunction;
$\mid\ J \triangleright P$	reaction pattern, if J is matched, start P.

A join pattern J is defined as follows:

$J = J \mid J$	synchronization pattern;
$\mid\ x\langle \overrightarrow{y} \rangle$	message definition.

A process definition **def** D **in** P binds names of D in P. This reflects the locality principle mentioned before. A channel defined in D can be named only within the scope of D and P. A definition D, even when it consists of a conjunction of definitions $D_1 \wedge D_2$, is to be treated like an atomic place for name definitions. This means that names defined in D_2 are not free in D_1 and vice versa. The only binding construct is the join pattern. A process P started by a reaction pattern J, such as in $J \triangleright P$, is generally called a *guarded process*. The formal parameters received in a join pattern are bound in the corresponding guarded process.

In the following sections there will be references to different kind of variables: *received (rv), defined (dv), free (fv)*, their definitions derive from the existing literature on the join calculus. They are all defined by structural induction in Figure 3.1. We define a fresh name with regards to a process or a solution to be a name that is not free in them.

$$rv(x\langle \overrightarrow{y} \rangle) = \{y_1\} \cup \ldots \cup \{y_n\} \qquad rv(J \mid J') = rv(J) \cup rv(J')$$

$$dv(x\langle \overrightarrow{y} \rangle) = \{x\} \qquad dv(J \mid J') = dv(J) \cup dv(J')$$

$$dv(J \triangleright P) = dv(J) \qquad dv(D \wedge D') = dv(D) \cup dv(D')$$

$$fv(J \triangleright P) = dv(J) \cup (fv(P) \setminus rv(J)) \qquad fv(D \wedge D') = fv(D) \cup fv(D')$$

$$fv(x\langle \overrightarrow{y} \rangle) = \{x\} \cup \{y_1\} \cup \ldots \cup \{y_n\}$$

$$fv(P \mid P') = fv(P) \cup fv(P') \qquad fv(\textbf{def } D \textbf{ in } P) = (fv(P) \cup fv(D)) \setminus dv(D)$$

Figure 3.1: Definition of received, defined and free variables in the join calculus.

3.1.2 Structural equivalence

The structural equivalence relation is axiomatized as the least equivalence relation satisfying the conditions defined in Figure 3.2.

$$\begin{aligned}
P &\equiv P' && \textit{if } P, P' \textit{are } \alpha\textit{-equivalent}\\
D &\equiv D' && \textit{if } D, D' \textit{are } \alpha\textit{-equivalent}\\
P|\emptyset &\equiv P &&\\
P|Q &\equiv P'|Q' && \textit{if } P \equiv P' \textit{and } Q \equiv Q'\\
D_1 \wedge D_2 &\equiv D_1' \wedge D_2' && \textit{if } D_1 \equiv D_1' \textit{and } D_2 \equiv D_2'\\
P|P' &\equiv P'|P &&\\
D \wedge D' &\equiv D' \wedge D &&\\
J|J' &\equiv J'|J &&\\
(P|Q)|R &\equiv P|(Q|R) &&\\
(D_1 \wedge D_2) \wedge D_3 &\equiv D_1 \wedge (D_2 \wedge D_3) &&\\
\mathbf{def}\ D\ \mathbf{in}\ \mathbf{def}\ D'\ \mathbf{in}\ P &\equiv \mathbf{def}\ D'\ \mathbf{in}\ \mathbf{def}\ D\ \mathbf{in}\ P && \textit{if } fv(D') \cap dv(D) = \emptyset\\
& && \wedge\ fv(D) \cap dv(D') = \emptyset\\
\mathbf{def}\ D \wedge D'\ \mathbf{in}\ P &\equiv \mathbf{def}\ D\ \mathbf{in}\ \mathbf{def}\ D'\ \mathbf{in}\ P && \textit{if } fv(D) \cap dv(D') = \emptyset
\end{aligned}$$

Figure 3.2: Structural congruence for the join calculus.

3.1.3 Semantics

The semantics is specified as a reflexive chemical abstract machine CHAM. The state of the computation is a chemical soup $\mathcal{E} \Vdash \mathcal{M}$ that consists of two sets: active definitions $\mathcal{E}$ and running processes $\mathcal{M}$.

There are two kind of rules for the soup to evolve:

structural rules (expressed by $\rightleftharpoons$) These rules are reversible and are used to rearrange terms. There are two kinds of structural rules: *heating* $\rightharpoonup$ and *cooling*: $\leftharpoondown$.

reduction rules (expressed by $\longrightarrow$) These rules represent the basic computational step. Each reduction rule consumes a process and replaces it with another.

The initial state of the computation will generally be $\emptyset \Vdash$ **def** D **in** P. The machine will start from this point and then unravel all the processes and definition until they are all separated in the soup. They will be recombined afterwards to trigger reaction patterns and then make the soup evolve.

Before explaining the φ function that performs the substitution, we need to introduce some additional notation.

With the notation y_j^x we represent a variable y that is the j-th parameter of another variable x.

The notation $|x|$ indicates the number of arguments one expects to be sent over a given channel x.

The substitution function $P[z/x]$

We now define the substitution function $P[z/x]$ which is analogous to the classical capture avoiding substitution function used in λ-calculus.

The expression $P[z/x]$ is read "P where all occurrences of x are substituted with z".

The function is defined in Figure 3.3 by structural induction on the term P. Note that the substituting variable z does not capture any name defined in a definition D. We also assume possibly implicit α-renaming for non free variables to avoid name clashes.

$$
\begin{array}{rclrcl}
\textbf{def } D \textbf{ in } P[z/x] & \equiv & \textbf{def } D[z/x] \textbf{ in } P[z/x] & (P \mid P')[z/x] & \equiv & P[z/x] \mid P'[z/x] \\
x\langle\overrightarrow{y}\rangle[z/x] & \equiv & z\langle\overrightarrow{y}\rangle & u\langle\overrightarrow{y}\rangle[z/x] & \equiv & u\langle\overrightarrow{y}\rangle \\
u\langle y_1, \ldots, x, \ldots, y_n\rangle[z/x] & \equiv & u\langle y_1, \ldots, z, \ldots, y_n\rangle & \emptyset[z/x] & \equiv & \emptyset \\
(D \wedge D')[z/x] & \equiv & D[z/x] \wedge D'[z/x] & (J \rhd P)[z/x] & \equiv & J \rhd P[z/x]
\end{array}
$$

Figure 3.3: Definition of the substitution function $P[z/x]$.

The function φ

φ is a function that substitutes the transmitted names for the distinct received variables. Such function takes three arguments:

- a process P;
- a parallel composition of messages $J' \equiv x_1\langle\overrightarrow{z_1}\rangle|\ldots|x_n\langle\overrightarrow{z_n}\rangle$;
- a join pattern $J \equiv x_1\langle\overrightarrow{y_1}\rangle|\ldots|x_n\langle\overrightarrow{y_n}\rangle$;

such that $dv(J)$=$dv(J')$ and returns a process P'.

P' is actually P where, for all $x \in dv(J)$, the j-th formal parameter of x has been substituted with the j-th actual parameter. All the formal parameters are found in $rv(J)$ while all the actual ones are taken from $rv(J')$.

Formally, since $J \equiv x_1\langle\overrightarrow{y_1}\rangle|\ldots|x_n\langle\overrightarrow{y_n}\rangle$ and $J' \equiv x_1\langle\overrightarrow{z_1}\rangle|\ldots|x_n\langle\overrightarrow{z_n}\rangle$, we can define:

$$P' = P[\overrightarrow{z_1}/\overrightarrow{y_1}, \ldots, \overrightarrow{z_n}/\overrightarrow{y_n}]$$

The substitution $[\overrightarrow{z}/\overrightarrow{y}]$ represents the ordered substitution $[z_1/y_1, \ldots, z_m/y_m]$ given $\overrightarrow{z} = z_1, \ldots, z_m$ and $\overrightarrow{y} = y_1, \ldots, y_m$.

The substitution function φ can be stated in an extended form as follows:

$$P' = P[z_1^{x_1}/y_1^{x_1}, \dots, z_{m_1}^{x_1}/y_{m_1}^{x_1}, \dots, z_1^{x_n}/y_1^{x_n}, \dots, z_{m_n}^{x_n}/y_{m_n}^{x_n}]$$

The rules for the CHAM are described in Figure 3.4. Here we mention only the elements of both multisets that participate in the rule. Such rules in fact apply to any matching subpart of the soup.

$$\begin{array}{rlcrll}
 & \Vdash P_1 | P_2 & \rightleftharpoons & & \Vdash P_1, P_2 & \text{S-PAR} \\
D \wedge D' & \Vdash & \rightleftharpoons & D, D' & \Vdash & \text{S-AND} \\
 & \Vdash \mathbf{def}\ D\ \mathbf{in}\ P & \rightleftharpoons & D & \Vdash P & \text{S-DEF} \\
 & & & & dv(D) \text{ are fresh.} & \\
J \rhd P & \Vdash J' & \longrightarrow & J \rhd P & \Vdash \varphi(P, J', J) & \text{R-BETA} \\
 & & & & dv(J) = dv(J') &
\end{array}$$

$$\frac{\begin{array}{c}\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons\!\!\!\!\!\!\longrightarrow \mathcal{D}_2 \Vdash \mathcal{P}_2 \\ (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset\end{array}}{\mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons\!\!\!\!\!\!\longrightarrow \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}_2, \mathcal{P}}\ \mathit{CTX}$$

$\rightleftharpoons\!\!\!\!\!\!\longrightarrow$ represents any computational step.

Figure 3.4: Chemical rules for the reflexive CHAM.

Rule S-AND and S-PAR from Figure 3.4 express that "|" and "$\wedge$" are commutative and associative. Rule S-DEF describes the heating of a molecule that defines new names. The side condition of such rule mimics the scope extrusion of the ν operator in π-calculus, and at the same time enforces a strict static scope for the definitions. Given a process **def** D **in** P, rule S-DEF enforces names defined in D to be unique for the soup and limits the binding of such names to the process P. Formally, by heating the **def** D **in** P molecule in a general soup as $\mathcal{E} \Vdash$ **def** D **in** $P, \mathcal{M}$ the following holds: $dv(D) \cap fv(\mathcal{E}, \mathcal{M}) = \emptyset$.

The basic computational step is provided with rule R-BETA. Such reduction consumes any molecule that matches a given pattern J, makes a fresh copy of the guarded process P, substitutes the received parameters in P with the actual sent names and releases such process (obtained via the φ function) in the solution as a new floating molecule.

The symbol $\rightleftharpoons\!\!\!\!\!\!\longrightarrow$ is an abstraction for any of the above reduction steps. Rule CTX states a general evolution rule for soups. Consider a chemical solution $\mathcal{D}_1 \Vdash \mathcal{P}_1$ that evolves in $\mathcal{D}_2 \Vdash \mathcal{P}_2$. We can add a set of definitions $\mathcal{D}$ and a set of processes $\mathcal{P}$ to the first soup if their names do not clash with those of the soup. Such addition does not affect the behavior of the solution.

Labeled semantics

In the following we will attach labels to the arrows representing computational steps in order to clarify what is happening in the soup.

Rule R-BETA will have a label pointing out the parallel composition of messages that triggers a specific rule. For example

$$\xrightarrow{x\langle y\rangle | a\langle\rangle}$$

means that messages $x\langle y\rangle$ and $a\langle\rangle$ are being sent.

General heating and cooling arrows will be labeled with the name of the rule that is being used. For example, a heating via S-DEF rule will be denoted by:

$$\xrightharpoondown{S-DEF}$$

For a better understanding of the calculus we introduce now some examples.

3.1.4 Examples

After the brief tour of the asynchronous core of the join calculus, it is time to see it at work. This section is aimed at showing the benefits of the calculus by presenting two examples, each one pointing out a particular gain. In the first example we formalize a secrecy example from [AG99].

Example 2 (Process communication example)**.** *Let us consider a system where a* Sender *wants to share a secret with a* Receiver*. The two can be formalized in the join calculus as:*

$$\begin{aligned}\text{Sender} = {} & \textbf{\textit{def}}\ secret\langle\rangle \triangleright send\langle secret\rangle \\ & \textbf{\textit{in}}\ \ secret\langle\rangle\end{aligned}$$

$$\begin{aligned}\text{Receiver} = {} & \textbf{\textit{def}}\ show\langle ch\rangle \triangleright recv\langle ch\rangle \\ & \wedge\quad chan\langle sec\rangle \triangleright \texttt{i have the secret} \\ & \textbf{\textit{in}}\ \ show\langle chan\rangle\end{aligned}$$

The Sender *is a process that knows a secret and communicates it by exporting it on the channel* send*. The* Receiver *is a process that wants to receive some input on channel* chan*, so he sends such channel on* recv*. Once something is transmitted on* chan *the process has the input in the local variable* s*. These two processes must be combined with an environment for them to communicate. There are two free variables in the above defined processes:* send *and* recv*. The environment in which* Sender *and* Receiver *will be placed in is meant to define them to bind such free occurrences.*

$$\begin{aligned}\text{Environment} = {} & \textbf{\textit{def}}\ send\langle s\rangle \mid recv\langle ch\rangle \triangleright ch\langle s\rangle \\ & \textbf{\textit{in}}\ \ \text{Sender} \mid \text{Receiver}\end{aligned}$$

This environment defines a communication rule. It waits for a channel to be sent over the send *channel and forwards it on the channel sent over the* recv *channel. The term* Environment *is a closed term, it has neither free variables nor holes where to place other processes. The only way to combine it with other processes is to place it in a parallel composition or after a definition D in the term **def** D **in** P. Whatever process we combine* Environment *with, it will not be able to steal the secret channel* secret*. This is the greatest benefit that derives from the locality principle which is typical join calculus.*

Let's place the three processes in a CHAM and analyze its evolution. Due to α-conversion in rule S-DEF, the channel names defined in the three processes are unique for these processes only. No outer channel with the same name as one in the process Environment *may synchronize with that channel, thus enforcing that channel names are local to their defining process.*

Unfortunately someone may add some malicious code in order to export these channels outside the created environment allowing a leakage of secrets. There is no control over such actions.

The second example shows how the computation happens in a CHAM.

Example 3 (Computation example)**.** *Let us now consider a one place buffer. Its implementation in our calculus is:*

$$\begin{aligned}\text{Onebuffer} = {} & \textbf{\textit{def}}\ put\langle x\rangle \mid empty\langle\rangle \triangleright full\langle x\rangle \\ & \wedge \quad get\langle ch\rangle \mid full\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \\ & \textbf{\textit{in}} \quad empty\langle\rangle \mid publ\langle put, get\rangle\end{aligned}$$

There are four defined channels: full, empty, get, put*. The usage is simple; when the state is* empty*, someone can put a channel in the buffer by sending it on the* put *channel. When something is stored in the buffer, it can be retrieved via the* get *channel. By sending a channel* ch *over* get*, a process will receive the contents of the buffer on* ch*. Channel* publ *is used to export both* put *and* get *outside the definition of* OneBuffer *so that external processes may use it.*

The following is a general environment where the buffer can be placed so that processes $P_1, \ldots, P_n$ can use it by sending a message on mb *containing the channel where they want to receive* put *and* get.

$$\begin{array}{lll} \mathrm{Activator} = \textbf{\textit{def}}\ mb\langle publ\rangle \triangleright \textbf{\textit{def}} & put\langle x\rangle \mid empty\langle\rangle \triangleright full\langle x\rangle & \\ \qquad\qquad\qquad\qquad\quad \wedge & get\langle ch\rangle \mid full\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle & \\ \qquad\qquad\qquad\qquad\quad \textbf{\textit{in}} & empty\langle\rangle \mid publ\langle put, get\rangle & \\ \qquad\quad \textbf{\textit{in}}\ \ P_1 \mid \ldots \mid P_n & & \end{array}$$

Note an interesting point. When a process P_i activates the buffer, it is added to the processes set. Let's assume P_i activates the buffer by sending a channel $\mathrm{channelP}_i$ *over* mb. *Then:*

$$Activator \Vdash P_1, \ldots, mb\langle channelP_i\rangle, \ldots, P_n \xrightarrow{mb\langle channelP_i\rangle} Activator \Vdash \left| \begin{matrix} P_1, \ldots, P_n, \\ OneBuffer_i \end{matrix} \right|$$

Now let's suppose another process P_j activates the buffer. Then:

$$Activator \Vdash \left| \begin{matrix} P_1, \ldots, mb\langle channelP_j\rangle, \ldots, P_n, \\ OneBuffer_i \end{matrix} \right| \xrightarrow{mb\langle channelP_j\rangle} Activator \Vdash \left| \begin{matrix} P_1, \ldots, P_n, \\ OneBuffer_i, OneBuffer_j \end{matrix} \right|$$

The newly created process $\mathrm{Onebuffer}_j$ *spawned by P_j has variables which are distinct from P_i's buffer's due to α-conversion in rule S-DEF.*

Let's apply rule S-DEF to unravel the process definition $\mathrm{OneBuffer}_i$. *The side condition of this rule will map all elements of dv($Onebuffer_i$) to fresh names, so there will be no conflict with dv($OneBuffer_j$).*

$$Activator \Vdash \left| \begin{matrix} P_1, \ldots, P_n, \\ OneBuffer_i, OneBuffer_j \end{matrix} \right| \xrightarrow{S-DEF}$$

$$\left| \begin{matrix} Activator, \\ put_i\langle x_i\rangle \mid empty_i\langle\rangle \triangleright full_i\langle x_i\rangle \\ \wedge\ get_i\langle ch_i\rangle \mid full_i\langle a_i\rangle \triangleright ch_i\langle a_i\rangle \mid empty_i\langle\rangle \end{matrix} \right| \Vdash \left| \begin{matrix} P_1, \ldots, P_n \\ empty_i\langle\rangle \mid publ_i\langle put_i, get_i\rangle \\ Onebuffer_j \end{matrix} \right|$$

If we apply rule S-DEF to unfold $\mathrm{Onebuffer}_j$ *we obtain a whole new set of variables that do not conflict with the already existing ones:*

$$\left| \begin{matrix} Activator, \\ put_i\langle x_i\rangle \mid empty_i\langle\rangle \triangleright full_i\langle x_i\rangle \\ \wedge\ get_i\langle ch_i\rangle \mid full_i\langle a_i\rangle \triangleright ch_i\langle a_i\rangle \mid empty_i\langle\rangle \end{matrix} \right| \Vdash \left| \begin{matrix} P_1, \ldots, P_n, \\ empty_i\langle\rangle \mid publ_i\langle put_i, get_i\rangle \\ Onebuffer_j \end{matrix} \right| \xrightarrow{S-DEF}$$

$$\left| \begin{matrix} Activator, \\ put_i\langle x_i\rangle \mid empty_i\langle\rangle \triangleright full_i\langle x_i\rangle \\ \wedge\ get_i\langle ch_i\rangle \mid full_i\langle a_i\rangle \triangleright ch_i\langle a_i\rangle \mid empty_i\langle\rangle, \\ put_j\langle x_j\rangle \mid empty_j\langle\rangle \triangleright full_j\langle x_j\rangle \\ \wedge\ get_j\langle ch_j\rangle \mid full_j\langle a_j\rangle \triangleright ch_j\langle a_j\rangle \mid empty_j\langle\rangle \end{matrix} \right| \Vdash \left| \begin{matrix} P_1, \ldots, P_n, \\ empty_i\langle\rangle \mid publ_i\langle put_i, get_i\rangle, \\ empty_j\langle\rangle \mid publ_j\langle put_j, get_j\rangle \end{matrix} \right|$$

Now processes from P_j will use the buffer variables labeled with a "j" and processes from P_i will use the ones labeled with an " i" and thus they will not interfere.

3.1.5 Related Work

We have shown the join calculus and its asynchronous core and we have given a couple of examples that should help the reader to understand how the calculus works and what its power is. We shall now present some of the most remarkable achievements related to the join calculus.

Surely one of the greatest achievements of the join calculus is the equivalence with π-calculus. As already stated, this equivalence is stated and proved in [FG96].

The join calculus was invented to provide a useful programming language for concurrent systems, this language is presented in [FFMS02]. The starting point was Objective Caml (OCaml), which is a compiled, general purpose, high level programming language that combines functional, imperative and object-oriented programming styles. By adding concurrency, distribution and mobility to such language we obtain JOCaml. A key concept in this language is the one of *location*, which is the basic unit of locality where processes are run. This model is adequate to represent a hierarchic network architecture: a distributed system. Processes can be defined in a syntax that stems out of the core join calculus. Obviously there will be all the needed additions, like primitives for printing and so on, which are required for a language to be useful. Additional information on the JOCaml language can be found in [FM98].

The asynchronous core of the join calculus has also been augmented with primitives for object handling in [FLMR00] following the objects-as-record paradigm. The result was a simple language of objects with asynchronous message passing. The addition of classes to join calculus enables the modular definition of synchronization. From a programming language point of view this strikes a good balance between flexibility and simplicity while not precluding type inference or efficient compilation of synchronization.

A programming language receives several benefits from static analyses of programs. Such concepts rely on type systems. By adapting the typing discipline developed for ML, a type system for the join calculus was published in [FLMR97]. This type system provides traditional parametric polymorphism strengthening the confidence in join calculus programs. The type systems developed in the next sections owe much to this work (and to [Sim10] as well), both in the definition of the typing rules and in the subject reduction proof.

Chapter 4

The channels-as-owners model (ChaO)

Abstract

In questo capitolo viene presentato un primo type system che importa l'idea di ownership types nel join calculus. Si forniscono le regole di sintassi, tipaggio e semantica di tale modello per poi attestarne la correttezza formale tramite le dimostrazioni di subject reduction e no runtime errors. Vengono poi presentati alcuni esempi di codice che evidenziano le proprietà del type system in questione.

This chapter outlines a type system for the join calculus. Here we want to import the idea of ownership types and their benefits into that particular model for concurrency. The join calculus already has a notion of locality, but this can be broken by exporting a channel out of the environment it was created in. Of course this behavior allows processes to communicate, so exporting should not be eliminated, it has to be controlled.

As we have seen in Chapter 2, ownership types provide a strong notion of encapsulation. We can import them into the join calculus to enforce such concept. Of course all the other properties enforced by ownership types hold. This system has a no representation exposure property, it allows a better memory management through improved garbage collection and it is better suitable for reasoning.

The analogy we follow here is the one that exists between a channel and an object. In classic ownership types systems there are objects which own other objects, so the idea is to let a channel own other channels.

Next comes the syntax definition, typing rules, semantics and theorems. Finally we show some examples of the type system at work.

4.1 Syntax

Before giving the syntax we establish a number of conventions.

Processes and definitions are defined modulo renaming of bound variables, substitution performs α-conversion to avoid captures. Furthermore, due to α-equivalence, we suppose all the name variables to be distinct. This lightens the rules of a heavy formal burden.

Let Σ denote a denumerable set of variables ranged over by: x, y, v, z, o. The name *world* is a system keyword, so it is not in Σ and channels are not allowed to have such name.

We define an *ownership scheme* l as a list of pairs $\langle y, l\rangle$, where y is a channel and l is another ownership scheme. An ownership scheme tracks a channel's parameters' owners and their ownership schemes.

$$l = \emptyset \mid l, \langle y, l\rangle$$

When accessing the i-th element of an ownership scheme, we will use the notation $l[i]$. To access the owner element of a pair we will use the function `own`, to access the scheme element of a pair we will use the function `sch`. For example, given an ownership scheme $l = \langle y, \langle k, \langle\emptyset\rangle\rangle\rangle, \langle j, \langle t, \langle\emptyset\rangle\rangle\rangle$, $\texttt{own}(l[1]) = y$, while $\texttt{sch}(l[2]) = \langle t, \langle\emptyset\rangle\rangle$. We assume that $\texttt{own}(\langle\emptyset\rangle) = \textit{world}$.

We define the environment Γ as a list of triples (x, y, l), where x is a channel name, y is an already defined channel name that is the owner of x, and l is an ownership scheme. Γ contains at least the triple $(\textit{world}, \textit{world}, \emptyset)$.

$$\Gamma = (\textit{world}, \textit{world}, \emptyset) \mid \Gamma, (x, y, l)$$

A process P is defined as follows:

$P = \emptyset$	null process
$\mid\ P \mid P$	parallel composition
$\mid\ x\langle\overrightarrow{y}\rangle$	send channels $y_{1\ldots n}$ on channel x
$\mid$ **def** D **in** P	process definition

A definition D is defined as follows:

$D = D \wedge D$	definition conjunction
$\mid\ J \triangleright P$	reaction pattern, if J is matched, start P

A join pattern J is defined as follows:

$J = J \mid J$	synchronization pattern
$\mid\ x_o\langle\overrightarrow{y}\rangle$	message definition.

The difference from the standard join calculus syntax is that the *message definition* syntax $x_o\langle\overrightarrow{y}\rangle$ has an owner parameter o. The syntax of such parameters is:

$o =$ rep	syntactic sugar for the defining channel itself (e.g. $x_{rep}\langle\rangle \equiv x_x\langle\rangle$)
$\mid$ world	a keyword that specifies the root of the channels tree
$\mid$ x	a channel name

In addition to variables of Figure 3.1, we have two new types of variable: *defined owner (do)* and *free owner (fo)*. Such variables are defined by structural induction in Figure 4.1. We report only the most interesting cases, the other ones can be obtained by classical structural induction.

$$do(x_o\langle\overrightarrow{y}\rangle) = \{o\} \qquad\qquad do(J \mid J') = do(J) \cup do(J')$$

$$fo(\textbf{def } D \textbf{ in } P) = (dv(D) \setminus do(D)) \cup (dv(P) \setminus do(P))$$

Figure 4.1: Definition of free and defined owners for ChaO.

The assumptions made for ownership types in Chapter refcapOT hold here as well. A channel must have an owner which is fixed throughout the object's lifetime. Every channel defines two contexts: *owner* which is the context it is defined in, and *representation* which is the context it defines. Given a channel x, the first context has the name of the channel that owns x, while the latter has name x. We will refer to these contexts with the following functions: `owner` and `rep`.

Contexts form a partial order $(C, \prec:)$ that is forest shaped. The relation $\prec:$ is called *inside*.

The following is the containment invariant enforced by the type system:

$$x\langle y\rangle \Rightarrow \texttt{rep}(x) \prec: \texttt{owner}(y)$$

A channel x may use another channel y, namely, y can be sent on x, if x is *inside* y's owner. This means that there is a path descending the channels tree leading from y's owner to x.

Of course $\texttt{rep}(x) \prec:\texttt{owner}(x)$.

4.1.1 Structural equivalence

Most of the rules for structural congruence mentioned in Figure 3.2 in Chapter 3 hold here as well, the only differences are shown in Figure 4.2. When swapping two definitions we must consider owners too and avoid clashes.

$$\textbf{def } D \wedge D' \textbf{ in } P \equiv \textbf{def } D \textbf{ in def } D' \textbf{ in } P \qquad \mathit{iff} fv(D) \cap dv(D') = \emptyset \wedge do(D) \cap dv(D) = \emptyset$$

Figure 4.2: Structural congruence for the ChaO system.

4.2 Typing judgments

Now we present the typing judgments.

$\Gamma \vdash \diamond$	Γ is a well-typed environment.
$\Gamma \vdash P$	P is a well-typed process in Γ.
$\Gamma \vdash D :: \Gamma'$	D is a well-typed definition in Γ and Γ' contains the bindings for $dv(D)$.
$\Gamma \vdash J :: \Gamma'$	J is a well-typed join pattern in Γ and Γ' contains the bindings for $dv(J)$.
$\Gamma \vdash x : o : l$	x has owner o in Γ and l is its ownership scheme.
$\Gamma \vdash o : l$	owner o and ownership scheme l are well-typed in Γ.
$\Gamma \vdash l$	l is a well formed ownership scheme in Γ.
$\Gamma \vdash x \prec: y$	x is *inside* y in Γ.

4.3 Typing rules

Figure 4.3, 4.4 points out the typing rules for the ChaO system.

Rule *Env-null* states that the environment consisting of the only triple $(\mathit{world}, \mathit{world}, \emptyset)$ is well formed. Rule *Env-build* states that we can add a triple (x, o, l) to a well formed environment Γ if x has not been already defined and if $o : l$ is a well formed type annotation. Rules for well-typed ownership schemes are two. The first one: *Scheme* states that an ownership scheme is well formed if it is composed of pairs of well constructed types. The second rule, *Scheme-null*, states that an empty scheme is well-typed as long as the type checking environment is well-typed. Rule *Type* states the correctness of type annotations. A type is well formed if the owner o is in the type checking environment and if the ownership scheme is well typed. Additionally we require the owner o to be *inside* all of the ownership scheme's owners. For a channel x to be well-typed with type $o : l$, rule *Chan* requires the environment to be well typed. In addition to that x must have not been introduced in the environment after the triple (x, o, l) that links the variable with the type annotation $o : l$. The axiom of the *inside* relation is reported in rule *Inside*. It states that an object is inside its owner. The relation *inside* is closed for transitivity as rule *Inside-trans* points out. The relation *inside* is closed for reflexivity also as we see in rule *Inside-relf*.

Good environments

$$\frac{}{(world, world, \emptyset) \vdash \diamond}\ \textit{Env-null} \qquad \frac{\Gamma \vdash \diamond \quad x \notin dom(\Gamma) \quad \Gamma \vdash o : l}{\Gamma, (x, o, l) \vdash \diamond}\ \textit{Env-build}$$

Good ownership schemes

$$\frac{\forall i(\Gamma \vdash o_i : l_i)}{\Gamma \vdash \langle o_1, l_1\rangle, \ldots, \langle o_n, l_n\rangle}\ \textit{Scheme} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \langle\emptyset\rangle}\ \textit{Scheme-null}$$

Good types

$$\frac{o \in dom(\Gamma) \qquad \Gamma \vdash l \qquad \forall i(\Gamma \vdash o \prec: \texttt{own}(l[i]))}{\Gamma \vdash o : l}\ \textit{Type}$$

Well-typed channels

$$\frac{\Gamma, (x, o, l), \Gamma' \vdash \diamond \qquad x \notin dom(\Gamma')}{\Gamma, (x, o, l), \Gamma' \vdash x : o : l}\ \textit{Chan}$$

Well-typed inside relation

$$\frac{\Gamma \vdash x : y : l}{\Gamma \vdash x \prec: y}\ \textit{Inside} \qquad \frac{\Gamma \vdash x \prec: y \qquad \Gamma \vdash y \prec: z}{\Gamma \vdash x \prec: z}\ \textit{Inside-trans} \qquad \frac{\Gamma \vdash \diamond \qquad x \in dom(\Gamma)}{\Gamma \vdash x \prec: x}\ \textit{Inside-refl}$$

Figure 4.3: Typing rules for ChaO system (part 1).

Rule *Null* states that an empty process is well-typed only if the environment is. In rule *Par* we find that for a parallel composition to be well-typed, both the involved processes are required to be well-typed. Rule *Pdef* states that for a **def** D **in** P to be well-typed we check the definition D in an environment augmented with the variables D defines and then we check the process P in the same environment. When sending a message rule *Msg* checks that all the involved variables have already been defined, and that all the owners and the schemes of the actual parameters coincide with the ones declared in l. Rule *And* points out that conjunction of definitions require both definitions to be well-typed. Also no variable may be defined twice. Rule *Run* describes the reaction rule. If a join pattern is well-typed as well as the process it starts, then we have a well-typed definition. Parallel composition of join patterns require both patterns to be checked, as rule *Join* states. We also impose no variable may be defined or received twice . Finally, rule *Cdef* explains channel definition. When defining a channel we require all the involved variables to be defined. The scheme of the defined channel is a composition of the owners and the

Well-typed processes rules

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \emptyset}\ \textit{Null} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P \mid P'}\ \textit{Par}$$

$$\frac{\Gamma, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \Gamma' \vdash P \qquad dom(\Gamma') = dv(D)}{\Gamma \vdash \textbf{def}\ D\ \textbf{in}\ P}\ \textit{Pdef}$$

$$\frac{\Gamma \vdash x : z : l \qquad \forall i(\Gamma \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \ldots, \langle z_n, l_n\rangle}{\Gamma \vdash x\langle \overrightarrow{y} \rangle}\ \textit{Msg}$$

Well-typed definitions rules

$$\frac{\Gamma \vdash D :: \Gamma' \qquad \Gamma \vdash D' :: \Gamma'' \qquad dv(D) \cap dv(D') = \emptyset}{\Gamma \vdash D \wedge D' :: \Gamma', \Gamma''}\ \textit{And} \qquad \frac{\Gamma \vdash J :: \Gamma' \qquad \Gamma \vdash P}{\Gamma \vdash J \rhd P :: \Gamma'}\ \textit{Run}$$

Well-typed join patterns rules

$$\frac{\Gamma \vdash J :: \Gamma' \qquad \Gamma \vdash J' :: \Gamma'' \qquad rv(J) \cap rv(J') = \emptyset \quad dv(J) \cap dv(J') = \emptyset}{\Gamma \vdash J \mid J' :: \Gamma', \Gamma''}\ \textit{Join}$$

$$\frac{\Gamma \vdash x : o : l \qquad \forall i(\Gamma \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \ldots, \langle z_n, l_n\rangle}{\Gamma \vdash x_o\langle \overrightarrow{y} \rangle :: (x, o, l), (y_1, z_1, l_1), \ldots, (y_n, z_n, l_n)}\ \textit{Cdef}$$

Figure 4.4: Typing rules for ChaO system (part 2).

schemes of its formal parameters.

4.4 Semantics

The semantics we show here is slightly different from the one presented in Chapter 3. This machine is called *reduced chemical abstract machine* (RCHAM) due to the loss of rule S-AND. RCHAM was presented in [FLMR97].

There is some notation we need to explain before introducing the semantics rules.

$\mathcal{J}$ is a parallel composition of messages. Its syntax is: $x\langle \overrightarrow{y} \rangle \mid \mathcal{J}|\mathcal{J}$.

From now on we use the notation $\mathcal{D}$ and $\mathcal{P}$ to denote sets of definitions and processes respectively.

For every chemical soup $\mathcal{D} \Vdash \mathcal{P}$ we require every name to be defined in exactly one definition of $\mathcal{D}$:

$$\forall D, D' \in \mathcal{D}, dv(D) \cap dv(D') = \emptyset$$

The rules for the RCHAM are described in Figure 4.5.

$$
\begin{array}{rlcrll}
 & \Vdash P_1|P_2 & \rightleftharpoons & & \Vdash P_1, P_2 & \text{S-PAR} \\
 & \Vdash \textbf{def } D \textbf{ in } P & \rightleftharpoons & D & \Vdash P & \text{S-DEF} \\
 & & & & \quad dv(D) \text{ are fresh.} & \\
D \wedge J \triangleright P \wedge D' & \Vdash \mathcal{J} & \longrightarrow & D \wedge J \triangleright P \wedge D' & \Vdash \varphi(P, \mathcal{J}, J) & \text{R-BETA} \\
 & & & & \quad dv(J){=}dv(\mathcal{J}). &
\end{array}
$$

Figure 4.5: Chemical rules for the RCHAM of ChaO.

The side condition $dv(J){=}dv(\mathcal{J})$ on rule R-BETA implies that the channel is well used, and with the correct number of parameters.

Every rule has to be interpreted as in Section 3.1.3. The modifications made to rule R-BETA express the associativity and commutativity of $\wedge$ previously pointed out by rule S-AND. The generic application of a rule as in Figure 3.4 in Chapter 3 holds here as well without variations. Recall such figure for any need.

The need of a restricted machine may not be clear now, we shall quote from [FLMR97] why we cannot use a standard CHAM.

Consider two definitions D_1, D_2 such that some names defined in D_1 occur free in D_2 but not the converse, we have that:

$$\textbf{def } D_1 \textbf{ in def } D_2 \textbf{ in } P \equiv \textbf{def } D_1 \wedge D_2 \textbf{ in } P$$

We would expect every typing property to be preserved but this is not the case here. The valid typing judgments for the names defined in D_1 and used in D_2 are not the same for both sides of the equivalence. This is why we eliminate rule S-AND and rewrite rule R-BETA in a generalized version that expresses anyway the commutativity and associativity of $\wedge$.

4.5 Additional typing judgments and rules

Typing of programs is extended to chemical solutions. To do so, we have an additional typing judgment:

$\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$	The chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is well-typed in Γ.

These are the additional typing rules for chemical solutions.

For a set of processes to be well-typed rule *P-elim* requires every single one to be.

Well-typed sets rules

$$\frac{\forall i(\Gamma \vdash P_i) \qquad \mathcal{P} = P_1, \ldots P_n}{\Gamma \vdash \mathcal{P}} \; \textit{P-elim}$$

$$\frac{\forall i(\Gamma, \Gamma' \vdash D_i :: \Gamma'_i) \qquad \mathcal{D} = D_1, \ldots, D_n \qquad \Gamma' = \Gamma'_1, \ldots \Gamma'_n \quad dv(D_1) \cap \ldots \cap dv(D_n) = \emptyset}{\Gamma \vdash \mathcal{D} :: \Gamma'} \; \textit{D-elim}$$

Well-typed soup rule

$$\frac{\Gamma, \Gamma' \vdash \mathcal{D} :: \Gamma' \qquad \Gamma, \Gamma' \vdash \mathcal{P} \qquad dom(\Gamma') = dv(\mathcal{D})}{\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}} \; \textit{Soup}$$

Figure 4.6: Additional typing rules for ChaO.

Rule *D-elim* states that, for a set of definitions to be well-typed, every single definition has to be so in an environment augmented with the variables defined by all the definitions. We also require no variable to be defined twice.

The last rule, *Soup*, concerns well-typed chemical solutions. For a soup to be well-typed we require all its definitions to be so in an environment augmented by all the defined variables of such definitions. We also require all the processes to be well-typed in the same augmented environment.

4.6 Properties

Before stating and proving theorems and lemmas, there is some notation to point out since proofs will use it.

- An ownership scheme l associated to a channel x can be referred to with the notation l^x.

- The length of an ownership scheme l is expressed with the notation $|l|$.

- The token $\mathcal{T}$ represent any of the following elements one can find in the right side of a typing judgment. $\mathcal{T} = \diamond \mid x : o : l \mid o : l \mid l \mid x \prec: y \mid P \mid D \mid J \mid \mathcal{D} \Vdash \mathcal{P}$. We use $\mathcal{T}$ as an abstraction over those terms to be able to state properties about them. Note that some of the terms are just syntactic tokens referring to multiple instantiations of such token, for example $D \equiv D \wedge D' \mid J \rhd P$.

- The j-th parameter y of a channel x will be indicated by y^x_j

Lemma 1 (Single variable substitution lemma)**.** *If* $\Gamma \vdash P$ *and* $\Gamma \vdash x : o : l$ *and* $\Gamma \vdash z : o : l$ *then* $\Gamma \vdash P[z/x]$*.*

Proof. The proof goes by structural induction on P. In this proof we use the definition of the substitution function in Figure 3.3 in Chapter 3.

Base case. There are two base cases: $P \equiv \emptyset$ and $P \equiv u\langle\overrightarrow{y}\rangle$.

$P \equiv \emptyset$: The only matching rule here is *Null*.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \emptyset}\ \textit{Null}$$

Since the substitution $[z/x]$ leaves $\emptyset$ unchanged, the proof holds.

$P \equiv u\langle\overrightarrow{y}\rangle$: Here we have two cases, in fact x can be u or it can be an argument of u i.e. $x = y_i$. For the sake of simplicity we will show a case where $\overrightarrow{y}$ consists of an element only and that element is x. The generalized case i.e. $\overrightarrow{y} = y_1, \dots y_{i-1}, x, y_{i+1}, \dots, y_n$ is as easy and straightforward but has some syntactical burden we avoid.

In both cases the only rule one can apply is *Msg* but it generates different hypotheses.

$x = u$: The application of rule *Msg* is:

$$\frac{\Gamma \vdash x : o : l \qquad \forall i(\Gamma \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \dots, \langle z_n, l_n\rangle}{\Gamma \vdash x\langle\overrightarrow{y}\rangle}\ \textit{Msg}$$

We can substitute the hypothesis $\Gamma \vdash x : o : l$ with $\Gamma \vdash z : o : l$, which is in the Lemma statement, to apply rule *Msg* again and prove that $P[z/x]$ holds. In this case $P[z/x] \equiv x\langle\overrightarrow{y}\rangle[z/x] \equiv z\langle\overrightarrow{y}\rangle$.

$$\frac{\Gamma \vdash z : o : l \qquad \forall i(\Gamma \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \dots, \langle z_n, l_n\rangle}{\Gamma \vdash z\langle\overrightarrow{y}\rangle}\ \textit{Msg}$$

$x = y$: The application of rule *Msg* is:

$$\frac{\Gamma \vdash u : y : l \qquad \Gamma \vdash x : o : l \qquad l = \langle o, l\rangle}{\Gamma \vdash u\langle\overrightarrow{x}\rangle}\ \textit{Msg}$$

We substitute the hypothesis $\Gamma \vdash x : o : l$ with the one in the Lemma statement: $\Gamma \vdash z : o : l$ since the owners and the ownership schemes of both x and z coincide. We can then apply rule *Msg* to prove that $P[z/x]$ holds. In this case $P[z/x] \equiv u\langle x\rangle[z/x] \equiv u\langle z\rangle$.

$$\frac{\Gamma \vdash u : y : l \qquad \Gamma \vdash z : o : l \qquad l = \langle o, l \rangle}{\Gamma \vdash u\langle \overrightarrow{z} \rangle}\ \mathit{Msg}$$

Inductive case. There are two inductive cases: $P \equiv \textbf{def}\ D\ \textbf{in}\ P$ and $P \equiv P \mid P'$.

$P \equiv \textbf{def}\ D\ \textbf{in}\ P'$: The only rule that matches here is *Pdef*.

$$\frac{\Gamma, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \Gamma' \vdash P'}{\Gamma \vdash \textbf{def}\ D\ \textbf{in}\ P'}\ \mathit{Pdef}$$

The inductive hypothesis gives:

- $\Gamma, \Gamma' \vdash D[z/x] :: \Gamma'$
- $\Gamma, \Gamma' \vdash P'[z/x]$

to which we can apply rule *Pdef* and prove that $P[z/x]$ holds.

$$\frac{\Gamma, \Gamma' \vdash D[z/x] :: \Gamma' \qquad \Gamma, \Gamma' \vdash P'[z/x]}{\Gamma \vdash (\textbf{def}\ D\ \textbf{in}\ P')[z/x]}\ \mathit{Pdef}$$

$P \equiv P' \mid P''$: The only rule that matches here is *Par*.

$$\frac{\Gamma \vdash P' \qquad \Gamma \vdash P''}{\Gamma \vdash P \mid P'}\ \mathit{Par}$$

From the inductive hypotheses:

- $\Gamma \vdash P'[z/x]$
- $\Gamma \vdash P''[z/x]$

we can apply rule *Par* and prove that $P[z/x]$ holds.

$$\frac{\Gamma \vdash P'[z/x] \qquad \Gamma \vdash P''[z/x]}{\Gamma \vdash (P \mid P')[z/x]}\ \mathit{Par}$$

$\square$

Lemma 2 (Substitution lemma). *Consider a process P, a join pattern $J \equiv x_1\langle \overrightarrow{y_1} \rangle | \ldots | x_n\langle \overrightarrow{y_n} \rangle$ and a sequence of messages $\mathcal{J} \equiv x_1\langle \overrightarrow{z_1} \rangle | \ldots | x_n\langle \overrightarrow{z_n} \rangle$ such that $dv(J) = dv(\mathcal{J})$.*

If for all $x \in dv(J)$, for all j ranging from 1 to n we have:

$$\Gamma \vdash y_j^x : o_j : l_j \ \textit{and}\ \Gamma \vdash z_j^x : o_j : l_j \ \textit{and}\ \Gamma \vdash P$$

then:

$$\Gamma \vdash \varphi(P, \mathcal{J}, J)$$

Proof. The definition of function φ can be found in Section 3.1.3.

The hypotheses tell us that, for all x and for all j, we have:

- $\Gamma \vdash y_j^x : o_j : l_j$;
- $\Gamma \vdash z_j^x : o_j : l_j$
- $\Gamma \vdash P$

which is all we need to apply Lemma 1 to substitute every formal parameter y_j^x with the actual parameter z_j^x in P.

What we obtain after every application of Lemma 1 is:

$$P[z_1^{x_1}/y_1^{x_1}], \dots, P[z_{m_1}^{x_1}/y_{m_1}^{x_1}], \dots, P[z_1^{x_n}/y_1^{x_n}], \dots, P[z_{m_n}^{x_n}/y_{m_n}^{x_n}]$$

which is equivalent to:

$$P[z_1^{x_1}/y_1^{x_1}], \dots, [z_{m_1}^{x_1}/y_{m_1}^{x_1}], \dots, [z_1^{x_n}/y_1^{x_n}], \dots, [z_{m_n}^{x_n}/y_{m_n}^{x_n}]$$

due to the associativity of the substitution function presented in Figure 3.3.

Since $\varphi(P, \mathcal{J}, J) \equiv P[z_1^{x_1}/y_1^{x_1}], \dots, [z_{m_1}^{x_1}/y_{m_1}^{x_1}], \dots, [z_1^{x_n}/y_1^{x_n}], \dots, [z_{m_n}^{x_n}/y_{m_n}^{x_n}]$ the theorem holds. $\square$

Lemma 3 (Useless variables). *Let v be a name that is not free nor defined in $\mathcal{T}$ or Γ, let $o : l$ be a valid typing annotation in Γ.*

$$\Gamma \vdash \mathcal{T} \Leftrightarrow \Gamma, (v, o, l) \vdash \mathcal{T}$$

Proof. The proof is split in two sub cases : $\Rightarrow$ and $\Leftarrow$ both of which are demonstrated by induction on the typing proof associated to $\mathcal{T}$ for all the possible instantiations of $\mathcal{T}$.

$\Rightarrow$ In this case we have the hypothesis $\Gamma \vdash \mathcal{T}$.

Base case. There is only one token we can assign to $\mathcal{T}$ which has a derivation tree of height one. Here we consider $\mathcal{T} \equiv \diamond$ in the case of $\Gamma \equiv (\mathit{world}, \mathit{world}, \emptyset)$. The hypothesis states:

$$\frac{}{(\mathit{world}, \mathit{world}, \emptyset) \vdash \diamond}\ \textit{Env-null}$$

Due to the hypotheses given in the lemma statement we can immediately apply rule *Env-build* and prove the thesis.

$$\frac{(\mathit{world}, \mathit{world}, \emptyset) \vdash \diamond \qquad v \notin \mathit{dom}((\mathit{world}, \mathit{world}, \emptyset)) \quad (\mathit{world}, \mathit{world}, \emptyset) \vdash o : l}{(\mathit{world}, \mathit{world}, \emptyset), (x, o, l) \vdash \diamond}\ \textit{Env-build}$$

Inductive case. The inductive cases cover all the remaining instantiation of $\mathcal{T}$. We avoid presenting all the cases since most of the proofs are the same except for the tokens involved. We provide the most useful cases, convinced that it is easy to see the correctness of the remaining ones from the stated proofs.

$\mathcal{T} \equiv \diamond$ **and** $\Gamma \not\equiv (world, world, \emptyset)$ We can apply rule *Env-build* and prove the thesis.

$$\frac{\Gamma \vdash \diamond \quad v \notin dom(\Gamma) \quad \Gamma \vdash o : l}{\Gamma, (v, o, l) \vdash \diamond}\;\textit{Env-build}$$

$\mathcal{T} \equiv x : z : l$ This case is analogous to both $o : l$ and l, the proof strategy is exactly the same, except of course the typing rule involved.

The hypothesis is $\Gamma \vdash x : z : l$. The only matching rule is *Chan*, its application gives us the following hypotheses.

$$\frac{\Gamma, (x, z, l), \Gamma' \vdash \diamond \qquad x \notin dom(\Gamma')}{\Gamma, (x, z, l), \Gamma' \vdash x : z : l}\;\textit{Chan}$$

The inductive hypothesis tells us that $\Gamma, (x, z, l), \Gamma', (v, o, l) \vdash \diamond$. We can apply rule *Chan* to conclude the thesis since we have all the required hypotheses.

$$\frac{\Gamma, (x, z, l), \Gamma', (v, o, l) \vdash \diamond \qquad x \notin dom(\Gamma')}{\Gamma, (x, z, l), \Gamma', (v, o, l) \vdash x : z : l}\;\textit{Chan}$$

$\mathcal{T} \equiv x \prec: y$ This case is analogous to both $x \prec: x$ and $x \prec: z$.

The hypothesis states that $\Gamma \vdash x \prec: y$. The only matching rule is *Inside*.

$$\frac{\Gamma \vdash x : y : l}{\Gamma \vdash x \prec: y}\;\textit{Inside}$$

We can apply rule *Inside* to the inductive hypothesis to prove the thesis as follows.

$$\frac{\Gamma, (v, o, l) \vdash x : y : l}{\Gamma \vdash x \prec: y}\;\textit{Inside}$$

$\mathcal{T} \equiv P|P$ This case shows a methodology that holds for the following cases of $\mathcal{T}$ as well: $\emptyset$, $P|P$, **def** D **in** P, $D \wedge D'$, $J \rhd P$, $J|J$. The only modifications needed involve the token used and the rule which can be applied to such token.

The hypothesis states that $\Gamma \vdash P|P$. The only matching rule is *Par*.

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P \mid P'}\;\textit{Par}$$

The inductive hypothesis gives us both $\Gamma, (v, o, l) \vdash P$ and $\Gamma, (v, o, l) \vdash P'$. Now we prove the thesis by applying rule *Par* to such inductive hypotheses.

$$\frac{\Gamma, (v, o, l) \vdash P \qquad \Gamma, (v, o, l) \vdash P'}{\Gamma, (v, o, l) \vdash P \mid P'}\ \textit{Par}$$

$\mathcal{T} \equiv x_z\langle\overrightarrow{y}\rangle$ This case is analogous to $x\langle\overrightarrow{y}\rangle$, the proof strategy is exactly the same except of course for the typing rule involved.

The hypothesis states that $\Gamma \vdash x_o\langle\overrightarrow{y}\rangle$. The only matching typing rule is *C-def.*

$$\frac{\Gamma \vdash x : z : l \qquad \forall i(\Gamma \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \ldots, \langle z_n, l_n\rangle}{\Gamma \vdash x_z\langle\overrightarrow{y}\rangle :: (x, z, l), (y_1, z_1, l_1), \ldots, (y_n, z_n, l_n)}\ \textit{Cdef}$$

The inductive hypothesis provide us $\Gamma, (v, o, l) \vdash x : z : l$ and $\forall i(\Gamma, (v, o, l) \vdash y_i : z_i : l_i)$. We have all the needed hypotheses to apply rule *Cdef* and conclude the thesis.

$$\frac{\Gamma, (v, o, l) \vdash x : z : l \qquad \forall i(\Gamma, (v, o, l) \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \ldots, \langle z_n, l_n\rangle}{\Gamma, (v, o, l) \vdash x_z\langle\overrightarrow{y}\rangle :: (x, z, l), (y_1, z_1, l_1), \ldots, (y_n, z_n, l_n)}\ \textit{Cdef}$$

$\Leftarrow$ The hypothesis here is $\Gamma, (v, o, l) \vdash \mathcal{T}$.

Base case. Here we have $\mathcal{T} \equiv \diamond$ and $\Gamma \equiv (\textit{world}, \textit{world}, \emptyset)$. The only matching rule is *Env-build.*

$$\frac{(\textit{world}, \textit{world}, \emptyset) \vdash \diamond \quad v \notin \textit{dom}((\textit{world}, \textit{world}, \emptyset)) \quad (\textit{world}, \textit{world}, \emptyset) \vdash o : l}{(\textit{world}, \textit{world}, \emptyset), (v, o, l) \vdash \diamond}\ \textit{Env-build}$$

As we can see the thesis is provided among the hypotheses of the rule. Since $(\textit{world}, \textit{world}, \emptyset) \vdash \diamond$ holds, the case is proven.

Inductive case. The inductive cases proofs follow the same pattern of the inductive cases of $\Rightarrow$, therefore we report only one sample of such proofs.

$\mathcal{T} \equiv P|P$ The hypothesis states that $\Gamma, (v, o, l) \vdash P|P$. We can only apply rule *Par* developing the following proof tree.

$$\frac{\Gamma, (v, o, l) \vdash P \qquad \Gamma, (v, o, l) \vdash P'}{\Gamma, (v, o, l) \vdash P \mid P'}\ \textit{Par}$$

The inductive hypothesis tells us that $\Gamma \vdash P$ and $\Gamma \vdash P'$. We can apply rule *Par* to such hypotheses and prove the thesis.

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P \mid P'}\ \textit{Par}$$

□

Theorem 1 (Subject reduction)**.** *One step chemical reductions preserve typings. If* $\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons \mathcal{D}' \Vdash \mathcal{P}'$*, then there exists* Γ' *such that* $\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}'$ *and* Γ *and* Γ' *are the same except for the names defined in* $\mathcal{D}$ *but not in* $\mathcal{D}'$ *and vice versa.*

Proof. The proof goes by induction on the number of applications of rule CTX in the derivation of the reduction. Note that in this proof we refer to subtrees with the notation Π. Sometimes a single derivation has been split in subproofs for the whole derivation tree would exceed the page.

Base case: Here we consider each reaction rule:

S-PAR: The reduction is: $\Vdash P_1 \mid P_2 \rightleftharpoons \;\Vdash P_1, P_2$.

Heating $\rightharpoonup$: In this case the hypothesis is $\Gamma \vdash \;\Vdash P_1 \mid P_2$.
The derivation tree is the following. First we can only apply *Soup* and then the only matching rule is *Par*:

$$\dfrac{\dfrac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2}\,\text{Par}}{\Gamma \vdash \;\Vdash P_1 \mid P_2}\,\text{Soup}$$

With the assumptions made in this derivation we can apply rule *P-elim* and then *Soup* to prove the thesis $\Gamma \vdash \;\Vdash P_1, P_2$.

$$\dfrac{\dfrac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1, P_2}\,\text{P-elim}}{\Gamma \vdash \;\Vdash P_1, P_2}\,\text{Soup}$$

Cooling $\leftharpoondown$: The other way round: first apply *Soup* and *P-elim*. Now, with those hypotheses, apply *Par* and *Soup* to prove the thesis.

S-DEF: The reduction is: $\Vdash \textbf{def } D \textbf{ in } P \rightleftharpoons D \Vdash P$.

Heating $\rightharpoonup$: In this case our hypothesis is $\Gamma \vdash \;\Vdash \textbf{def } D \textbf{ in } P$ with side condition that $dv(D)$ are mapped to fresh names so that $dv(D) \cap fv(\mathcal{E}, \mathcal{M}) = \emptyset$.
The derivation tree is the following. First we can only apply *Soup*, then the only matching rule is *Pdef*.

$$\dfrac{\dfrac{\Gamma, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \Gamma' \vdash P}{\Gamma \vdash \textbf{def } D \textbf{ in } P}\,\text{Pdef}}{\Gamma \vdash \;\Vdash \textbf{def } D \textbf{ in } P}\,\text{Soup}$$

Now we have all the premises to apply rule *Soup* and prove the thesis $\Gamma \vdash D \Vdash P$.

$$\dfrac{\Gamma, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \Gamma' \vdash P}{\Gamma \vdash D \Vdash P}\;\text{Soup}$$

The side condition holds since we have that $\Gamma, \Gamma' \vdash D :: \Gamma'$. This means there is no other variable whose name matches any channel name defined in D except $dv(D)$ themselves.

Cooling $\leftharpoondown$: The other way round: first apply *Soup*, then use the hypotheses to apply *Pdef* and *Soup*.

R-BETA: The reduction is: $D \wedge J \rhd P \wedge D' \Vdash \mathcal{J} \longrightarrow D \wedge J \rhd\ P \wedge D' \Vdash \varphi(P, \mathcal{J}, J)$ with the following side conditions: $\mathcal{J}$ is a parallel composition of messages. Its syntax is: $x\langle\overrightarrow{y}\rangle \mid \mathcal{J}|\mathcal{J}$. $dv(J)$=$dv(\mathcal{J})$.

We rewrite the left side of the soup as $J \rhd P \wedge D$ for the sake of simplicity.

The hypothesis in this case is $J \rhd P \wedge D \Vdash \mathcal{J}$. Π_2 is the derivation tree that states such hypothesis is well-typed.

$$\Pi_1 = \dfrac{\dfrac{\dfrac{\begin{array}{c}\Gamma, \Gamma', \Gamma_D \vdash x : o : l \\ \forall j(\Gamma, \Gamma', \Gamma_D \vdash y_j : o_j : l_j) \quad l = \langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle\end{array}}{\vdots}\;\textit{Msg}_i}{\Gamma, \Gamma', \Gamma_D \vdash x_1\langle\overrightarrow{y_1}\rangle|\dots|x_n\langle\overrightarrow{y_n}\rangle}\;\textit{Par}}{\Gamma, \Gamma', \Gamma_D \vdash \mathcal{J}}\;\mathcal{J} \equiv x_1\langle\overrightarrow{y_1}\rangle|\dots|x_n\langle\overrightarrow{y_n}\rangle$$

$$\Pi_2 = \dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \Gamma', \Gamma_D \vdash J :: \Gamma' \quad \Gamma, \Gamma', \Gamma_D \vdash P}{\Gamma, \Gamma', \Gamma_D \vdash J \rhd P :: \Gamma'}\;\textit{Run} \qquad \Gamma, \Gamma', \Gamma_D \vdash D :: \Gamma_D}{\Gamma, \Gamma', \Gamma_D \vdash J \rhd P \wedge D :: \Gamma', \Gamma_D}\;\textit{And} \qquad \Pi 1}{\Gamma \vdash J \rhd P \wedge D \Vdash \mathcal{J}}}{}\;\textit{Soup}$$

Thanks to the side condition that says $dv(J)$=$dv(\mathcal{J})$, and thanks to α-conversion when applying rule S-DEF, we can say that all the names in $dv(J)$ are not aliased in the soup.

Consider the top of Π_2.

Since the join pattern J is well-typed, we know that, all of its defined variables are. *For all* $x \in dv(J)$, x is well-typed and also its ownership scheme l is. Here we know that $l = \langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle$. This means that *for all* j ranging over x's parameters list the following holds:

– $\Gamma, \Gamma', \Gamma_D \vdash w_j^x : o_j : l_j$

Where the notation w_j^x represents the j-th formal parameter w of x.

Now consider the top of the tree Π_1.

As we can see, the application of rule Msg_i gives us an important assumption. *For all* $x \in dv(\mathcal{J})$, and *for all* j ranging over x's parameters' list, we have that:

– $\Gamma, \Gamma', \Gamma_D \vdash y_j^x : o_j : l_j$

This means that all the formal parameters of every channel $x \in dv(J)$ have the same owners and the same ownership scheme of the corresponding actual parameter.

By combining those two assumptions with this one from Π_2:

– $\Gamma, \Gamma', \Gamma_D \vdash P$.

We can apply Lemma 2 and obtain the following proposition:

– $\Gamma, \Gamma', \Gamma_D \vdash \varphi(P, \mathcal{J}, J)$.

Now we have all the hypotheses to apply *Soup* to prove the thesis.

$$\dfrac{\dfrac{\Gamma, \Gamma', \Gamma_D \vdash J \rhd P :: \Gamma' \quad \Gamma, \Gamma', \Gamma_D \vdash D :: \Gamma_D}{\Gamma, \Gamma', \Gamma_D \vdash J \rhd P \wedge D :: \Gamma', \Gamma_D}\,\textit{And} \qquad \dfrac{\dots}{\Gamma, \Gamma', \Gamma_D \vdash \varphi(P, \mathcal{J}, J)}}{\Gamma \vdash J \rhd P \wedge D \Vdash \varphi(P, \mathcal{J}, J)}\,\textit{Soup}$$

Inductive case: The inductive case is proven uniformly for the context rule $\rightleftharpoons$:

$$\dfrac{\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons \mathcal{D}_2 \Vdash \mathcal{P}_2 \qquad (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset}{\mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}_2, \mathcal{P}}\,\textit{CTX}$$

Figure 4.7: Generic context rule as reported from Figure 3.4 in Section 3.1.3.

The first hypothesis we have here is $\mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}_1, \mathcal{P}$. By applying rule *Soup* and then rules *D-elim* and *P-elim* to the subtrees we have:

$$\dfrac{\dfrac{\Gamma, \Gamma' \vdash \mathcal{D} :: \Gamma_D \quad \Gamma, \Gamma' \vdash \mathcal{D}_1 :: \Gamma_1 \quad \Gamma' = \Gamma_1, \Gamma_D}{\Gamma, \Gamma' \vdash \mathcal{D}, \mathcal{D}_1 :: \Gamma'}\,\textit{D-elim} \qquad \dfrac{\Gamma, \Gamma' \vdash \mathcal{P} \quad \Gamma, \Gamma' \vdash \mathcal{P}_1}{\Gamma, \Gamma' \vdash \mathcal{P}, \mathcal{P}_1}\,\textit{P-elim}}{\Gamma \vdash \mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}_1, \mathcal{P}}\,\textit{Soup}$$

The inductive hypothesis tells us that $\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons \mathcal{D}_2 \Vdash \mathcal{P}_2$. This means $\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ and there exists Γ'' such that $\Gamma'' \vdash \mathcal{D}_2 \Vdash \mathcal{P}_2$. These are the derivation trees for both sides of the IH.

$$\Pi_1 = \quad \dfrac{\Gamma, \Gamma_1 \vdash \mathcal{D}_1 :: \Gamma_1 \quad \Gamma, \Gamma_1 \vdash \mathcal{P}_1}{\Gamma \vdash \mathcal{D}_1 \Vdash \mathcal{P}_1}\,\textit{Soup}$$

$$\Pi_2 = \quad \dfrac{\Gamma'', \Gamma_2 \vdash \mathcal{D}_2 :: \Gamma_2 \quad \Gamma'', \Gamma_2 \vdash \mathcal{P}_2}{\Gamma'' \vdash \mathcal{D}_2 \Vdash \mathcal{P}_2}\,\textit{Soup}$$

We also know that Γ and Γ'' are the same except for the variables defined in $\mathcal{D}_1$ only and those defined in $\mathcal{D}_2$ only but not those defined in both.

Let this set of variables be denoted with X.

The second hypothesis from rule CTX: $(fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset$ tells us $dv(\mathcal{D})$ are disjoint from X, i.e. $dv(\mathcal{D}) \cap X = \emptyset$.

The variables that appear in $\mathcal{D}_1$ only and those that appear in $\mathcal{D}_2$ only are not used in $\mathcal{D}$ or $\mathcal{P}$. We can then apply Lemma 3 to remove all the variables of $\mathcal{D}_1$ only from $\mathcal{D} \Vdash \mathcal{P}$ and add all the ones from $\mathcal{D}_2$ and the following holds:

$$\dfrac{\Gamma'', \Gamma_D, \Gamma_2 \vdash \mathcal{D} :: \Gamma_D \qquad \Gamma'', \Gamma_D, \Gamma_2 \vdash \mathcal{P}}{\Gamma'', \Gamma_2 \vdash \mathcal{D} \Vdash \mathcal{P}}\ \textit{Soup}$$

Thanks to Lemma 3 again, we are allowed to add Γ_D to the environment that type-checks $\mathcal{D}_2 \Vdash \mathcal{P}_2$ since the variables defined in Γ_D do not interfere with the ones defined in Γ_2. Thus the following holds:

$$\dfrac{\Gamma'', \Gamma_D, \Gamma_2 \vdash \mathcal{D}_2 :: \Gamma_2 \qquad \Gamma'', \Gamma_D, \Gamma_2 \vdash \mathcal{P}_2}{\Gamma'', \Gamma_D \vdash \mathcal{D}_2 \Vdash \mathcal{P}_2}\ \textit{Soup}$$

Now we have all the assumptions to apply *D-elim* and *P-elim* and conclude the thesis by applying *Soup.*

$$\dfrac{\dfrac{\Gamma'', \Gamma^* \vdash \mathcal{D} :: \Gamma_D \qquad \Gamma'', \Gamma^* \vdash \mathcal{D}_2 :: \Gamma_2 \qquad \Gamma^* = \Gamma_D, \Gamma_2}{\Gamma'', \Gamma^* \vdash \mathcal{D}, \mathcal{D}_2 :: \Gamma^*}\ \textit{D-elim} \qquad \dfrac{\Gamma'', \Gamma^* \vdash \mathcal{P} \qquad \Gamma'', \Gamma^* \vdash \mathcal{P}_2}{\Gamma'', \Gamma^* \vdash \mathcal{P}, \mathcal{P}_2}\ \textit{D-elim}}{\Gamma'' \vdash \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}_2, \mathcal{P}}\ \textit{Soup}$$

And, as stated before, Γ and Γ'' are the same except for the variables defined in $\mathcal{D}_1$ only and those defined in $\mathcal{D}_2$ only but not those defined in both.

Having covered all the cases, the theorem holds. □

The following is the translation of the *owners as dominators* property for object-oriented programming to this system.

Property 1 (Owners as dominators)**.** *A channel y may be sent over a channel x only if x is* inside *y's owner.*

$$x\langle y\rangle \Rightarrow x \prec: \mathrm{owner}(y)$$

Proof. The well-typedness of the expression $x\langle y\rangle$ allows us to develop the following tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots \qquad \dfrac{\cdots \qquad \Gamma \vdash o \prec: \mathtt{own}(\langle i,h\rangle)}{\Gamma \vdash o : l}\,\textit{Type}}{\Gamma, (x,o,l) \vdash \diamond}\,\textit{Env-build}}{\dfrac{\vdots}{\Gamma' \vdash \diamond}} \qquad x \in dom(\Gamma')}{\Gamma' \vdash x : o : l}\,\textit{Chan} \qquad \Gamma' \vdash y : i : h \qquad l = \langle i,h\rangle}{\Gamma' \vdash x\langle y\rangle}\,\textit{Msg}$$

The hypothesis $\Gamma \vdash o \prec: \mathtt{own}(\langle i,h\rangle)$ tells us that $o \prec: i$ which is y's owner, thus proving the thesis. □

Definition 1 (Runtime errors)**.** *Consider a soup $\mathcal{D} \Vdash \mathcal{P}$. We say that a runtime error occurs if we find either these kind of messages in the processes set $\mathcal{P}$:*

- *a message $x\langle\overrightarrow{y}\rangle$ that is not defined in $\mathcal{D}$, i.e. no join pattern J in $\mathcal{D}$ has x in its defined variables;*
- *a message $x\langle\overrightarrow{y}\rangle$ that is defined in $\mathcal{D}$ but with different arity (i.e. the defined channel x wants four parameters while we call it with three);*
- *a message $x\langle\overrightarrow{y}\rangle$ where x is not* inside *some of its arguments' owner i.e. there exists y_i such that $x \not\prec: y_i$*

Theorem 2 (No runtime errors)**.** *In any way a well-typed soup can evolve, it's never going to generate a runtime error.*
If $\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}' \Vdash \mathcal{P}'$ then $\mathcal{P}$ does not contain runtime errors.

Proof. From Theorem 1 we know that typing is preserved by chemical rewriting. Thus, since $\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$, there exists Γ' such that $\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}'$.

Since $\mathcal{D}' \Vdash \mathcal{P}'$ is well-typed, we know that rule *Msg* applies to any message sent in $\mathcal{P}$.

Let us recall how such rule looks:

$$\dfrac{\Gamma \vdash x : z : l \qquad \forall i(\Gamma \vdash y_i : z_i : l_i) \qquad l = \langle z_1, l_1\rangle, \ldots, \langle z_n, l_n\rangle}{\Gamma \vdash x\langle\overrightarrow{y}\rangle}\,\textit{Msg}$$

The first two definitions of runtime error are ruled out by two hypothesis of the above shown rule.

- "A message $x\langle\overrightarrow{y}\rangle$ that is not defined in $\mathcal{D}$, i.e. no join pattern J in $\mathcal{D}$ has x in its defined variables" is matched by hypothesis $\Gamma \vdash x : z : l$, so no errors of this type may occur.
- "A message $x\langle\overrightarrow{y}\rangle$ that is defined in $\mathcal{D}$ but with different arity" is matched by the definition of the ownership scheme l. In Γ, l has an arity which is confirmed to be that of x's parameters list due to hypothesis $l = \langle z_1, l_1\rangle, \ldots, \langle z_n, l_n\rangle$ in rule *Msg.*

We now have only a definition of runtime error which is ruled out thanks to Property 1. Having covered all the cases the theorem holds. □

The following are two different characterizations of a secrecy property expressed by M. Abadi in [Aba99].

Definition 2 (Secrecy 1). *Suppose the process $P(x)$ has at most x as free variable. Then p preserves the secrecy of x if $P(M)$ and $P(N)$ are equivalent for all terms M and N without free variables.*

Definition 3 (Secrecy 2). *Suppose that S is a set of terms with no free variables, and P a process with no free variables. Suppose the free names of M are not bound in P or any process into which P evolves. Let R be the relation associated with P and S. Then P may reveal M from S if there exists P' and S' such that $R(P', S')$ and M belongs to the set of terms computable by S'. Otherwise P preserves the secrecy of M from S.*

We can reformulate Property 1 as a secrecy property. We can think at a secret as a representation of a channel. By having a no representation exposure property such as Property 1 we guarantee no channel can access the secret unless it is *inside* the secret's owner.

Property 2 (Secrecy). *By declaring a channel x as owned by a channel z, for all channels y such that $y \not\prec: z$, x is kept secret.*

4.7 Examples

4.7.1 Diagrammatic notation

In the following examples we will use a graphical notation to outline the shape of a channels graph. This helps understanding the system and how the processes behave and interact. Figure 4.8 is an example of a channel graph.

Consider channel x. Its *owner* context is o while its *rep* context is x itself. We can say that x can use every channel in the tree while z and o can use z and o only. No channel can use the *world* channel since it is a special keyword that is only used to represent the root of the channels tree.

This example we show is a porting of Example 2 from Chapter 3 into join calculus with ownership types.

Example 4 (Example of secrecy). *The processes defined in Example 2 are rewritten with contexts annotations. Figure 4.9 points out the channels graph related to this example.*

$$\begin{aligned} \text{Sender } = \ & \boldsymbol{def}\ secret_{send}\langle\rangle \triangleright send\langle secret\rangle \\ & \boldsymbol{in}\ \ secret\langle\rangle \end{aligned}$$

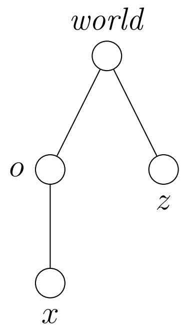

Figure 4.8: Visual example of the diagrammatic notation used to represent the channels graph.

$$
\begin{array}{lll}
\text{Receiver} = & \boldsymbol{def} & show_{send}\langle ch\rangle \triangleright recv\langle ch\rangle \\
& \wedge & chan_{send}\langle sec\rangle \triangleright \texttt{i have the secret} \\
& \boldsymbol{in} & show\langle chan\rangle
\end{array}
$$

$$
\begin{array}{lll}
\text{Environment} = & \boldsymbol{def} & send_{rep}\langle s\rangle \mid recv_{send}\langle ch\rangle \triangleright ch\langle s\rangle \\
& \boldsymbol{in} & \text{Sender} \mid \text{Receiver}
\end{array}
$$

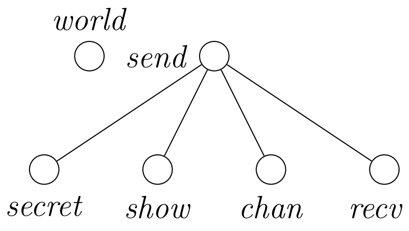

Figure 4.9: Channels graph for the processes of Example 4

Note that all the channels involved are defined as having send *as owner. This means that all the channels in the example cannot be exported outside* send*. A channel z, which is not in the tree rooted in* send*, sending one of the above mentioned variables among its parameters will constitute a bad typed expression.*

As we can see, the channels graph is a tree of depth one. In more complex programs the tree will grow deeper so it will define the system and its behavior in a more precise way.

The next example we show is a visual one that allows to see what a channel can refer to.

Example 5 (Visual example). *Let us consider the following process.*

$$\begin{aligned} \mathrm{P} = &\ \textbf{\textit{def}}\ x_{ctx}\langle\rangle \mid y_{ctx}\langle\rangle \mid o_x\langle\rangle \triangleright R \\ &\ \textbf{\textit{in}}\ \ Q \end{aligned}$$

Its channels graph is in Figure 4.10. Here we will highlight, for each channel, which channels it is allowed to access. A dashed line from y to o means that y cannot name o among its parameters. A straight line is a valid reference.

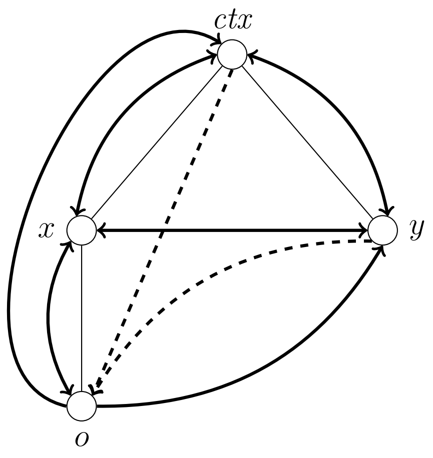

Figure 4.10: Graphical example representing accessibility in a channels graph.

Here we present another example that shows the benefits of the type system developed so far.

Example 6 (Example of no representation exposure). *Let us now consider a one place buffer similar to Example 3 presented in Chapter 3. We now give a bad typed implementation in our calculus:*

$$\begin{aligned} \text{Activator} = &\ \textbf{\textit{def}}\ mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \triangleright \textbf{\textit{def}}\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \triangleright full\langle x\rangle \\ &\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \\ &\qquad\qquad\qquad\qquad\qquad\quad \textbf{\textit{in}}\ \ empty\langle\rangle \mid publ\langle put, full\rangle \\ &\ \textbf{\textit{in}}\ \ \textbf{\textit{def}}\ pub_{mb}\langle p, g\rangle \triangleright P \\ &\qquad\ \ \textbf{\textit{in}}\ \ mb\langle pub\rangle \mid ctx\langle\rangle \end{aligned}$$

The one place buffer is the process started by the only join pattern in Activator. *The channels* empty *and* full *individualize the buffer's representation. To make this concept* *explicit they are declared as owned by channel* ctx*. Since* put *and* get *are used to access the buffer, they are made public by declaring* mb *as owner. As we can see,* mb *owns both* ctx *and* pub*. This makes* ctx *and* pub *siblings. The last thing to note is where bad typedness lies: the expression* $publ\langle put, full\rangle$ *.*

We can see that Property 1 does not hold here since full*'s owner cannot access* publ*'s one which is* mb*. We don't want other channels except those inside* ctx *to access* full *and* empty *since they represent the buffer's state.*

Let's now show an example reduction that the type system rules out. For the sake of simplicity we will refer to the buffer with the process name OneBuffer*.*

The starting point is: $\emptyset \Vdash Activator$*. By applying twice the heating transformation of rule S-DEF we obtain.*

$$\emptyset \Vdash Activator \xrightharpoonup{S-DEF} \xrightharpoonup{S-DEF} \begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \end{bmatrix} \Vdash mb\langle put\rangle \mid ctx\langle\rangle$$

Now we apply rule R-BETA, consume $mb\langle put\rangle \mid ctx\langle\rangle$ *and trigger OneBuffer.*

$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \end{bmatrix} \Vdash mb\langle put\rangle \mid ctx\langle\rangle \xrightarrow{mb\langle put\rangle \mid ctx\langle\rangle}$$
$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \end{bmatrix} \Vdash OneBuffer$$

We proceed by applying rule S-DEF to unravel the definition of OneBuffer.

$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \end{bmatrix} \Vdash OneBuffer \xrightharpoonup{S-DEF}$$
$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \rhd full\langle x\rangle \\ \wedge\ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \rhd ch\langle a\rangle \mid empty\langle\rangle \end{bmatrix} \Vdash empty\langle\rangle \mid pub\langle put, full\rangle$$

Application of heating of rule S-PAR will divide the processes $empty\langle\rangle \mid pub\langle put, full\rangle$ *bound in a parallel composition leading to the following soup:*

$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \rhd full\langle x\rangle \\ \wedge\ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \rhd ch\langle a\rangle \mid empty\langle\rangle \end{bmatrix} \Vdash empty\langle\rangle \mid pub\langle put, full\rangle \xrightharpoonup{S-PAR}$$
$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \rhd OneBuffer \\ pub_{mb}\langle p, g\rangle \rhd P \\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \rhd full\langle x\rangle \\ \wedge\ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \rhd ch\langle a\rangle \mid empty\langle\rangle \end{bmatrix} \Vdash empty\langle\rangle, pub\langle put, full\rangle$$

Now we would expect to be able to apply rule R-BETA and synchronize the join pattern $pub_{mb}\langle p,g\rangle \triangleright P$ *with the process* $pub\langle put, full\rangle$ *. Fortunately this is where the property of the type system come in help and forbids such reduction.*

Let's assume we could apply rule R-BETA to the soup. we would have the following reaction:

$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \triangleright OneBuffer \\ pub_{mb}\langle p,g\rangle \triangleright P \\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \triangleright full\langle x\rangle \\ \wedge\ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \end{bmatrix} \Vdash empty\langle\rangle, pub\langle put, full\rangle \xrightarrow{pub\langle put, full\rangle}$$

$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \triangleright OneBuffer \\ pub_{mb}\langle p,g\rangle \triangleright P \\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \triangleright full\langle x\rangle \\ \wedge\ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \end{bmatrix} \Vdash \begin{bmatrix} empty\langle\rangle, \\ \varphi(P, pub\langle put, full\rangle, pub_{mb}\langle p,g\rangle \triangleright P) \end{bmatrix}$$

By the definition of φ, $\varphi(P, pub\langle put, full\rangle, pub_{mb}\langle p,g\rangle \triangleright P) \equiv P[put/p][full/g]$.

Here we know that, since pub*'s owner is* mb, *all the formal parameters of* pub *must have an owner o such that* mb$\prec$:*o. In addition to that we know that the formal parameters and the actual ones have the same owners and ownership schemes. This would imply that* mb$\prec$:ctx, *since* ctx *is* full*'s owner.*

Of course this is not true since ctx *is owned my* mb *itself so the reduction cannot be consumed.*

We have proven that the following soup is bad typed and the unwanted behavior of representation exposure is eliminated.

$$\begin{bmatrix} mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \triangleright OneBuffer \\ pub_{mb}\langle p,g\rangle \triangleright P \\ put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \triangleright full\langle x\rangle \\ \wedge\ get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \end{bmatrix} \Vdash empty\langle\rangle, pub\langle put, full\rangle \overset{pub\langle put, full\rangle}{\not\longrightarrow}$$

The following example shows the tree shape a channels graph has when the size of the process augments.

Example 7 (Example of channels graph growth)**.** *We shall consider a well typed implementation of the one place buffer in Example 3 from Chapter 3.*

$$\begin{array}{lll} \text{Activator} = \textbf{\textit{def}}\ mb_{world}\langle publ\rangle \mid ctx_{mb}\langle\rangle \triangleright \textbf{\textit{def}} & & put_{mb}\langle x\rangle \mid empty_{ctx}\langle\rangle \triangleright full\langle x\rangle \\ & \wedge & get_{mb}\langle ch\rangle \mid full_{ctx}\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \\ & \textbf{\textit{in}} & empty\langle\rangle \mid publ\langle put, get\rangle \\ \quad\quad \textbf{\textit{in}}\ \ P_1 \mid \ldots \mid P_n & & \end{array}$$

Let's now define P_1 *as an environment that allows two processes to communicate as in Example 4. Of course there are some modifications to be made, here we show the* Sender, *the* Receiver *and the* Environment *we add instead of* P_1 *in the definition of* Activator. $P_1 \equiv Environment$.

$$\begin{array}{ll}\text{Environment} = & \boldsymbol{def}\ pub_{mb}\langle p, g\rangle \mid sender_{pub}\langle sec\rangle \mid receiver_{pub}\langle rec\rangle \triangleright sec\langle p\rangle \mid rec\langle g\rangle \\ & \boldsymbol{in}\ \ mb\langle pub\rangle \mid ctx\langle\rangle \mid Sender \mid Receiver\end{array}$$

$$\begin{array}{ll}\text{Sender} = & \boldsymbol{def}\ snd_{pub}\langle pt\rangle \mid secret_{mb}\langle\rangle \triangleright pt\langle secret\rangle \\ & \boldsymbol{in}\ \ sender\langle snd\rangle \mid secret\langle\rangle\end{array}$$

$$\begin{array}{ll}\text{Receiver} = & \boldsymbol{def}\ rcv_{pub}\langle gt\rangle \mid ch_{pub}\langle c\rangle \triangleright gt\langle c\rangle \\ & \wedge \quad chan_{mb}\langle s\rangle \triangleright \texttt{i have the secret} \\ & \boldsymbol{in}\ \ receiver\langle rcv\rangle \mid ch\langle chan\rangle\end{array}$$

The channels graph of this example is in Figure 4.11. We can see that not all channels are children of the first context, they are distributed along the tree.

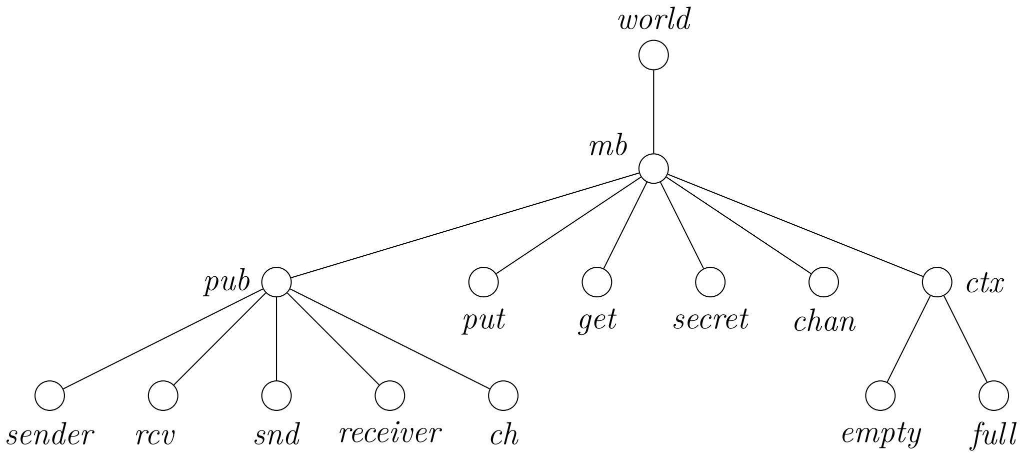

Figure 4.11: Channels graph of Example 7

The highest gain here is the structure imposed on the defined channels. The channels graph is not a simple tree of depth one but its height grows when we add more channels.

Chapter 5

The contexts-as-owners model (CtxO)

Abstract

In questo capitolo viene presentato un secondo type system che importa l'idea di ownership types nel join calculus. Si forniscono le regole di sintassi, tipaggio e semantica di tale modello per poi attestarne la correttezza formale tramite le dimostrazioni di subject reduction e no runtime errors. Vengono poi presentati alcuni esempi di codice che evidenziano le proprietà del type system in questione.

This chapter presents the second type system importing ownership types to the join calculus. The standard join calculus has not got the notion of context. Here we introduce this idea by adding a new context to each definition associated with the keyword **def**. The analogy we follow here is the one between classes and definitions. As classes are places where we define multiple objects, so are definitions. As classes own other classes and thus object at runtime, so definitions own other definitions. In order keep track of a definition we give it a name, called a context.

Every channel will be bound to a certain context and we won't let it escape such bounds, while it will be free to wander inside the area it has been defined in. The type system enforces this by forbidding that a channel defined in a context α be exported on a channel defined in context β if β is an ancestor of α. The gains are the same as provided by ownership types seen in Chapter 2. Such benefits do not differ from the ones provided by the type system introduced in Chapter 4.

Next comes the syntax definition, typing rules, semantics and theorems. Finally we show some examples of the type system at work.

5.1 Syntax

The conventions introduced in Section 4.1 hold here as well but we have a few more to introduce to deal with contexts.

Let Ω denote a denumerable set of contexts names ranged over by c, o, α, β. Ω and Σ are disjoint, no variable may have the same name as a context and vice versa. The environment Γ now handles both triples and contexts. The context ω is always present in Γ, it defines the root of the system as *world* does in the ownership types system. Context variables are not allowed to be named ω. More formally: $\omega \notin \Omega \cup \Sigma$.

$$\Gamma = \Gamma, (x, \alpha, l) \ \mid \ \Gamma, \alpha \ \mid \ \omega$$

A triple (x, α, l) binds a variable x to a context α to an ownership scheme l. Ownership schemes are the same as in Chapter 4, the only difference being they handle contexts here instead of channels.

A process P is defined as follows:

$P = \emptyset$	null process
$\mid\ P \mid P$	parallel composition
$\mid\ x\langle\overrightarrow{y}\rangle$	send channels $y_{1\ldots n}$ on channel x
$\mid\ \mathbf{def}_\alpha\ D\ \mathbf{in}\ P$	process definition

The syntax of definitions D and join patterns J is the same as the one presented in Section 4.1 The difference from the standard join calculus syntax lies in two points. First, the *process definition* rule has an added context parameter α, which is used to denote a new context for the newborn definition D. Second, the *message definition* rule has an owner parameter as in Section 4.1. The syntax of such parameters are:

$o =$ rep	syntactic sugar for the latest introduced context;
$\mid$ world	syntactic sugar for ω;
$\mid\ \alpha$	a context name.

In the following there may be reference to the variables of Figure 4.1. In addition to those we define two additional type of variables: *defined context (dc)* and *free context (fc)*. They are defined by structural induction based on the clauses defined in Figure 5.1.

5.1.1 Structural Equivalence

Most of the rules for structural congruence mentioned in Figure 3.2 in Chapter 3 hold here as well. The only different rule is presented in Figure 5.2.

$$dc(x_\alpha\langle\overrightarrow{y}\rangle) = \{\alpha\}$$

$$fc(\mathbf{def}_\alpha\ D\ \mathbf{in}\ P) = (dc(D) \cup dc(P)) \setminus \{\alpha\}$$

Figure 5.1: Definition of free context and defined context variables.

$$\mathbf{def}_\alpha\ D\ \mathbf{in}\ \mathbf{def}_\beta\ D'\ \mathbf{in}\ P \equiv \mathbf{def}_\beta\ D \wedge D'\ \mathbf{in}\ P \qquad \mathit{if}\,\alpha \notin fc(D, \mathbf{def}_\beta\ D'\ \mathbf{in}\ P)$$

Figure 5.2: Structural equivalence for CtxO.

5.2 Typing judgments

We have three typing judgments:

$\Gamma \vdash \diamond$	well formed environment Γ.
$\Gamma \vdash P$	the process P is well typed in Γ
$\Gamma \vdash D :: \Gamma'$	the definition D is well typed in Γ, with Γ' containing the bindings for its defined channels (both *dv*s and *rv*s)
$\Gamma \vdash J :: \Gamma'$	the join pattern J is well typed in Γ, with Γ' containing the bindings for its defined channels (both *dv*s and *rv*s)
$\Gamma \vdash x : \alpha : l$	channel x's first scope in Γ is α and l is its ownership scheme.
$\Gamma \vdash \alpha : l$	good context α and ownership scheme l in Γ.
$\Gamma \vdash l$	good ownership scheme l in Γ.

5.3 Typing rules

Here we present the typing rules for CtxO. Many rules do not differ from those presented in Chapter 4 so we avoid reporting them here too. Rule *Env-ctx* describes how to insert contexts into the environment. The only requirement is for the name context to be fresh. The typing rule *Type* mimics the behavior of rule *Type* in ChaO. For a pair $\alpha : l$ to be considered a well formed type we require α to be a valid context name and l to be a valid ownership scheme. The validation of l is done in the environment up to α, without considering the contexts that have been defined after α. This is made to enforce the contexts-as-dominators invariant. Rule *Pdef* is similar to ChaO's one. The only addition is that the rule enforces that the context α we introduce in a process definition is fresh.

Good Environments

$$\frac{}{\omega \vdash \diamond}\ \textit{Env-null} \qquad \frac{\Gamma \vdash \diamond \qquad \alpha \notin \Gamma}{\Gamma, \alpha \vdash \diamond}\ \textit{Env-ctx} \qquad \frac{\Gamma \vdash \diamond \qquad x \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash o : l}{\Gamma, (x, o, l) \vdash \diamond}\ \textit{Env-build}$$

Good types, ownership schemes and channels.

$$\frac{\Gamma, \alpha, \Gamma' \vdash \diamond \qquad \alpha \notin \mathit{dom}(\Gamma') \qquad \Gamma, \alpha \vdash l}{\Gamma, \alpha, \Gamma' \vdash \alpha : l}\ \textit{Type} \qquad \frac{\forall i(\Gamma \vdash \alpha_i : l_l)}{\Gamma \vdash \langle \alpha_1, l_1 \rangle, \ldots, \langle \alpha_n, l_n \rangle}\ \textit{Scheme}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \langle \emptyset \rangle}\ \textit{Scheme-null} \qquad \frac{\Gamma, (x, \alpha, l), \Gamma' \vdash \diamond \qquad x \notin \mathit{dom}(\Gamma')}{\Gamma, (x, \alpha, l), \Gamma' \vdash x : \alpha : l}\ \textit{Chan}$$

Well-typed processes.

$$\frac{\Gamma, \alpha, \Gamma' \vdash \diamond \qquad \Gamma, \alpha, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \alpha, \Gamma' \vdash P}{\Gamma \vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P}\ \textit{Pdef}$$

Figure 5.3: Typing rules for CtxO system.

5.4 Semantics

The RCHAM for this system differs slightly from the one presented in Section 4.4. Since we have to keep track of contexts, we expand the left side of the soup to contain such contexts in addition to definitions. The soup $\mathcal{D} \Vdash \mathcal{P}$ becomes $\mathcal{D}, \Delta \Vdash \mathcal{P}$, where Δ is a set of contexts. The general rule $\rightleftharpoons$ has been modified to consider contexts. We can add a set of contexts to an existing soup only if there is no clash with the already defined ones. Figure 5.4 gives the rules of the CtxO system.

Rule S-DEF mimics the scope extrusion principle inherited from π-calculus. We apply this notion not only to the variables in the definition D but also to the context α defined in $\mathbf{def}_\alpha\ D\ \mathbf{in}\ P$. This ensures no name conflict with both variables and contexts names.

5.5 Additional typing judgments and rules

Typing of programs is extended to chemical solutions. To do so, we have an additional typing judgment: $\Gamma \vdash \mathcal{D}, \Delta \Vdash \mathcal{P}$.

$\Gamma \vdash \mathcal{D}, \Delta \Vdash \mathcal{P}$	the chemical solution $\mathcal{D}, \Delta \Vdash \mathcal{P}$ is well typed in Γ.

We present the only different typing rule for CtxO in Figure 5.5.

For a soup to be well-typed, rule *Soup* checks all the definitions and all the processes in an environment augmented with the groups in the soup.

$$
\begin{array}{llllll}
 & \Vdash P_1|P_2 & \rightleftharpoons & & \Vdash P_1, P_2 & \text{S-PAR} \\
 & \Vdash \mathbf{def}_{\alpha}\ D\ \mathbf{in}\ P & \rightleftharpoons & & D, \alpha \Vdash P & \text{S-DEF} \\
 & & & & dv(D) \text{ and } \alpha \text{ are fresh} & \\
D \wedge J \triangleright P \wedge D' & \Vdash \mathcal{J} & \longrightarrow & D \wedge J \triangleright P \wedge D' & \Vdash \varphi(P, \mathcal{J}, J) & \text{R-BETA} \\
 & & & & dv(J) = dv(\mathcal{J}). &
\end{array}
$$

$$
\frac{\begin{array}{c}\mathcal{D}_1, \Delta_1 \Vdash \mathcal{P}_1 \rightleftharpoons \mathcal{D}_2, \Delta_2 \Vdash \mathcal{P}_2 \\ (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset \\ \Delta \cap (\Delta_1 \cup \Delta_2) = \emptyset\end{array}}{\mathcal{D}, \Delta, \mathcal{D}_1, \Delta_1 \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons \mathcal{D}, \Delta, \mathcal{D}_2, \Delta_2 \Vdash \mathcal{P}_2, \mathcal{P}}\ \mathit{CTX}
$$

Figure 5.4: Chemical rules for the RCHAM of CtxO.

$$
\frac{\Gamma, \Delta \vdash \diamond \qquad \Gamma, \Delta, \Gamma' \vdash \mathcal{D} :: \Gamma' \qquad \Gamma, \Delta, \Gamma' \vdash \mathcal{P}}{\Gamma \vdash \mathcal{D}, \Delta \Vdash \mathcal{P}}\ \mathit{Soup}
$$

Figure 5.5: Additional typing rule for CtxO.

5.6 Properties

For the proofs of this section we will mention some lemmas from Section 4.6. We avoid reporting them here and proving them again in the contexts-as-owners setting since the proofs would be the same. Also most of the theorems' proofs will look like those of Section 4.6, so we will write down only the most useful and different cases.

Lemma 4 (Useless contexts)**.** *Let β be a context name that is not free nor defined in $\mathcal{T}$ or Γ.*

$$\Gamma \vdash \mathcal{T} \Leftrightarrow \Gamma, \beta \vdash \mathcal{T}$$

Proof. The proof of this lemma is analogous to that of Lemma 3. We provide only the extra case that deals with contexts since the other ones are the same.

$\mathcal{T} \equiv \alpha : l$ The only matching rule is *Type*, its application gives us the following hypotheses.

$$
\frac{\Gamma, \alpha, \Gamma' \vdash \diamond \qquad \alpha \notin \mathit{dom}(\Gamma') \qquad \Gamma, \alpha \vdash l}{\Gamma, \alpha, \Gamma' \vdash \alpha : l}\ \mathit{Type}
$$

The inductive hypotheses we obtain are: $\Gamma, \alpha, \Gamma', \beta \vdash \diamond$ and $\alpha \notin \mathit{dom}(\Gamma', \beta)$. We can apply rule *Type* to these hypotheses and prove the thesis as follows.

$$\frac{\Gamma, \alpha, \Gamma', \beta \vdash \diamond \qquad \alpha \notin dom(\Gamma', \beta) \qquad \Gamma, \alpha \vdash l}{\Gamma, \alpha, \Gamma', \beta \vdash \alpha : l}\ \textit{Type}$$

□

Theorem 3 (Subject reduction). *One step chemical reductions preserve typings. If* $\Gamma \vdash \mathcal{D}, \Delta \Vdash \mathcal{P}$ *and* $\mathcal{D}, \Delta \Vdash \mathcal{P} \rightleftharpoons \mathcal{D}', \Delta' \Vdash \mathcal{P}'$*, then there exists* Γ' *such that* $\Gamma' \vdash \mathcal{D}', \Delta' \Vdash \mathcal{P}'$ *and* Γ *and* Γ' *are the same except for the names defined in* $\mathcal{D}, \Delta$ *but not in* $\mathcal{D}', \Delta'$ *and vice versa.*

Proof. The proof goes by induction on the number of applications of rule CTX in the derivation of the reduction. Note that in this proof we refer to subtrees with the notation Π. Sometimes a single derivation has been split in subproofs where the whole derivation tree would exceed the page.

Base case: Here we consider each reaction rule:

S-PAR: This case is analogous to S-PAR case in the proof of Theorem 1.

S-DEF: The reduction is: $\Vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \rightleftharpoons D, \alpha \Vdash P$.

Heating $\rightharpoonup$: In this case our hypothesis is $\Gamma \vdash\ \Vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P$ with side condition that $dv(D)$ and α are mapped to fresh names. The derivation tree is the following. First we can only apply *Soup*, then the only matching rule is *Pdef*.

$$\frac{\dfrac{\Gamma, \alpha \vdash \diamond \qquad \Gamma, \alpha, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \alpha, \Gamma' \vdash P}{\Gamma \vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P}\ \text{Pdef}}{\Gamma \vdash\ \Vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P}\ \text{Soup}$$

Now we have all the premises to apply rule *Soup* and prove the thesis $\Gamma \vdash D, \alpha \Vdash P$.

$$\frac{\Gamma, \alpha \vdash \diamond \qquad \Gamma, \alpha, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \alpha, \Gamma' \vdash P}{\Gamma \vdash D, \alpha \Vdash P}\ \text{Soup}$$

The side condition holds since we have that $\Gamma, \alpha, \Gamma' \vdash D :: \Gamma'$. This means there is no other variable whose name matches some channel name defined in D except $dv(D)$ themselves and no other context whose name clashes with α.

Cooling $\leftharpoondown$: The other way round: first apply *Soup*, then use the hypotheses to apply *Pdef* and *Soup*.

R-BETA This case is analogous to R-BETA case in the proof of Theorem 1.

Inductive case. The inductive case is proven uniformly for the context rule $\rightleftharpoons$. The proof follows the pattern of the inductive case of Theorem 1. The only difference is that we need to augment the environment not only by useless variables but also by useless contexts. The useless variables addition is allowed by Lemma 3 while adding useless contexts is permitted by Lemma 4. We omit the proof since it provides no new concepts. □

Property 3 (Owners as dominators revisited)**.** *A channel y may be sent over a channel x only if y's context α has not been declared after x's context β. This means that α precedes β in the environment Γ. More formally $\Gamma = \Gamma', \alpha, \Gamma'', \beta, \Gamma'''$. If $(x, \beta, l), (y, \alpha, l') \in \Gamma$, then:*

$$\Gamma \vdash x\langle y\rangle \Rightarrow \Gamma = \Gamma', \alpha, \Gamma'', \beta, \Gamma'''$$

Proof. The proof is analogous to that of Property 1. For $x\langle y\rangle$ to be well typed we develop the following tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots \quad \dfrac{\cdots \quad \dfrac{\Gamma'', \beta \vdash \alpha : h \quad \cdots}{\Gamma'', \beta \vdash \langle \alpha, h\rangle}\,\textit{Scheme}}{\Gamma' \vdash \beta : l}\,\textit{Type}}{\Gamma', (x, \beta, l) \vdash \diamond}\,\textit{Env-build}}{\dfrac{\vdots}{\Gamma \vdash \diamond}} \quad x \in dom(\Gamma)}{\Gamma \vdash x : \beta : l}\,\textit{Chan} \quad \Gamma \vdash y : \alpha : h \quad l = \langle \alpha, h\rangle}{\Gamma \vdash x\langle y\rangle}\,\textit{Msg}$$

As we can see the context α is not declared after β since we have $\Gamma'', \beta \vdash \alpha : h$. □

Definition 4 (Runtime errors)**.** *Consider a soup $\mathcal{D}, \Delta \Vdash \mathcal{P}$. We say that a runtime error occurs if we find either these kind of messages in the processes set $\mathcal{P}$:*

- *a message $x\langle \overrightarrow{y}\rangle$ that is not defined in $\mathcal{D}$, i.e. no join pattern J in $\mathcal{D}$ has x in its defined variables;*
- *a message $x\langle \overrightarrow{y}\rangle$ that is defined in $\mathcal{D}$ but with different arity (i.e. the defined channel x wants four parameters while we call it with three);*
- *a message $x\langle \overrightarrow{y}\rangle$ where one of the arguments' context β has been declared after x's context α.*

Property 4 (No runtime errors)**.** *However a well-typed soup evolves, it will never generate a runtime error.*

If $\Gamma \vdash \mathcal{D}, \Delta \Vdash \mathcal{P}$ and $\mathcal{D}, \Delta \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}', \Delta' \Vdash \mathcal{P}'$ then $\mathcal{P}$ does not contain runtime errors.

Proof. The proof is analogous to that of Theorem 2, the only difference being we use Property 3 to prove that no runtime errors appear. □

Property 5 (Secrecy)**.** *By declaring a channel x as owned by a context α, x is kept secret from all channels y whose context β has been declared after α.*

5.7 Examples

In the following we translate the examples from Section 4.7 to the CtxO notation.

5.7.1 Diagrammatic notation

Before showing the examples we present the notation used to show contexts and channels graphs. We represent a channel x that belongs to a context A with the form shown in Figure 5.6:

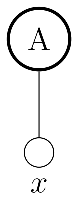

Figure 5.6: Example of the diagrammatic notation for CtxO.

By adopting this notation it is easier to relate a channel to the context it is defined in. These graphs will generally have a tree shape, showing the scope of all the channels involved in a process definition.

In the untyped join calculus we are not able to point out a channel's scope. A channel can be passed out of the definition it has been defined in turning the original channels graph into a list.

Example 8 (Example of secrecy)**.** *The processes defined in Example 4 are rewritten with contexts annotations as follows:*

$$
\begin{aligned}
\text{Sender } = {} & \textbf{\textit{def}}_S \; secret_E\langle\rangle \triangleright send\langle secret\rangle \\
& \textbf{\textit{in}} \quad secret\langle\rangle
\end{aligned}
$$

$$
\begin{aligned}
\text{Receiver } = {} & \textbf{\textit{def}}_R \; show_E\langle ch\rangle \triangleright recv\langle ch\rangle \\
& \wedge \quad chan_E\langle sec\rangle \triangleright \texttt{i have the secret} \\
& \textbf{\textit{in}} \quad show\langle chan\rangle
\end{aligned}
$$

$$
\begin{aligned}
\text{Environment } = {} & \textbf{\textit{def}}_E \; send_E\langle s\rangle \mid recv_E\langle ch\rangle \triangleright ch\langle s\rangle \\
& \textbf{\textit{in}} \quad \text{Sender} \mid \text{Receiver}
\end{aligned}
$$

Note that all the channels involved have context E as owner. Here no channel can be exported outside E. *This is the main property enforced by this type system.*

The following is the channels graph related to this example.

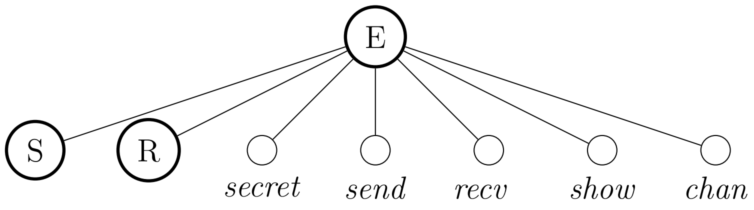

Figure 5.7: Channels graph for Example 8

As we can see, the channels graph is a tree of depth one. In more complex programs the tree will grow deeper so it will define the system and its behavior in a more precise way.

Example 9 (Visual example)**.** *Let us consider the following process.*

$$
\begin{array}{lll}
\mathrm{P} = \boldsymbol{def}_A & x_A\langle\rangle | y_A\langle\rangle \triangleright R & \\
\quad\;\; \boldsymbol{in} & \boldsymbol{def}_B \; w_B\langle\rangle \triangleright Q & \\
 & \qquad\; \boldsymbol{in} \quad T &
\end{array}
$$

Its channel graph is depicted in Figure 5.8. Here we will highlight, for each channel, which channels it is allowed to access. A straight line is a valid reference, a dashed line from y to w means that y cannot name w among its parameters.

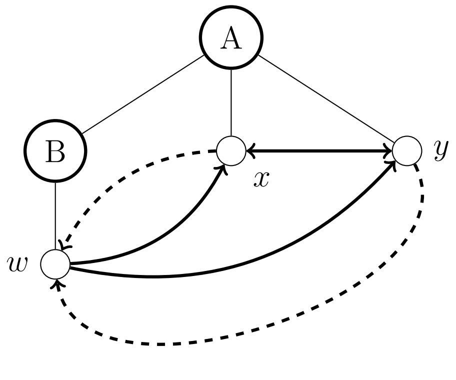

Figure 5.8: Channels graph for Example 9

Example 10 (Example of channels graph growth)**.** *The processes in this example are the translation of those from Example 7. First of all we present a one-place buffer and its activation environment.*

$$\begin{aligned}
\text{Activator} = \;& \boldsymbol{def}_A\ \mathit{makebuffer}_A\langle pub\rangle \triangleright \boldsymbol{def}_B\ put_{world}\langle x\rangle \mid empty_B\langle\rangle \triangleright full\langle x\rangle \\
& \qquad\qquad\qquad\qquad\quad \wedge \quad get_{world}\langle ch\rangle \mid full_B\langle a\rangle \triangleright ch\langle a\rangle \mid empty\langle\rangle \\
& \qquad\qquad\qquad\qquad\quad \boldsymbol{in} \quad empty\langle\rangle \mid pub\langle put, full\rangle \\
& \boldsymbol{in} \quad P_1 \mid \ldots \mid P_n
\end{aligned}$$

We then present a Sender *and a* Receiver*, linked via an* Environment *process, that can communicate by using the one-place buffer.*

$$\begin{aligned}
\text{Sender} = \;& \boldsymbol{def}_S\ snd_E\langle pt\rangle \mid secret_A\langle\rangle \triangleright pt\langle secret\rangle \\
& \boldsymbol{in} \quad sender\langle snd\rangle \mid secret\langle\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Receiver} = \;& \boldsymbol{def}_R\ rcv_E\langle gt\rangle \mid ch_E\langle c\rangle \triangleright gt\langle c\rangle \\
& \wedge \quad chan_E\langle s\rangle \triangleright \texttt{i have the secret} \\
& \boldsymbol{in} \quad receiver\langle rcv\rangle \mid ch\langle chan\rangle
\end{aligned}$$

$$\begin{aligned}
\text{Environment} = \;& \boldsymbol{def}_E\ pub_A\langle p, g\rangle \mid sender_E\langle sec\rangle \mid receiver_E\langle rec\rangle \triangleright sec\langle p\rangle \mid rec\langle g\rangle \\
& \boldsymbol{in} \quad makebuffer\langle pub\rangle \mid Sender \mid Receiver
\end{aligned}$$

The channel graph of this example is shown in figure 5.9. We can see that not all channels are a child of the first context, they are distributed along the contexts tree.

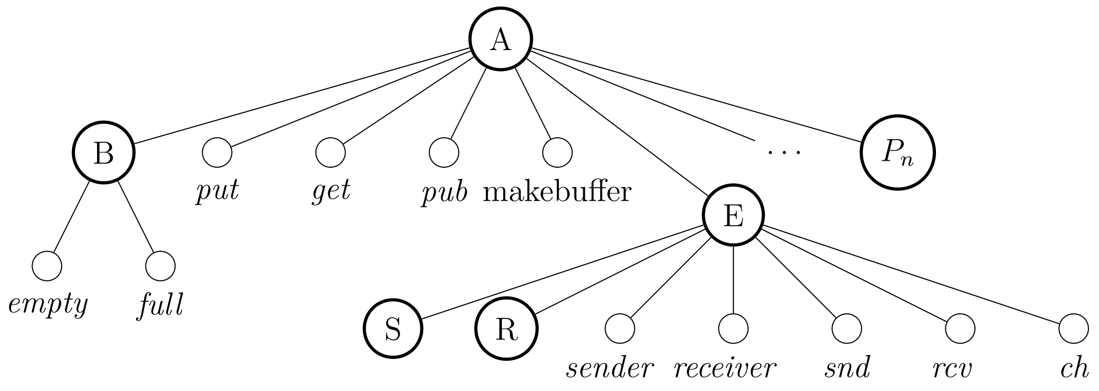

Figure 5.9: Channels graph for Example 10.

Chapter 6

Groups for the join calculus

Abstract

In questo capitolo si trasporta la nozione di gruppi dal π-calculus al join calculus. Si forniscono regole di sintassi, tipaggio e semantica oltre alla dimostrazione della correttezza del sistema in questione.

Groups for the π-calculus have been proposed by Cardelli et al in [CGG05] to enforce security properties. They have been used in [DZG02] to encode the region calculus [TT97] into the π-calculus. In this chapter we import such notion of Groups into the join calculus. What we aim is to provide a type system to compare the previously introduced ones with. Groups seem to offer a set of benefits which is quite close to the ones offered by ownership types, a comparison between the two type systems seems logical.

The original work on groups is done for the π-calculus. Since join calculus and π-calculus are proven to be equivalent, we expect that the notion of groups can be transferred to the join calculus as well. The two calculi may be equivalent in terms of expressiveness, but the core abstractions are different. Transferring the ideas did require some work.

Next comes the syntax definition, typing rules, semantics and core theorem. We leave the comparisons between the three systems presented so far for the next chapter.

6.1 Syntax

Before giving the syntax we establish a number of new conventions. If not specified, the notation made for Chapters 4 and 5 holds here as well.

Let Θ be a denumerable set of group names ranged over by: G, H, F. Θ and Σ are disjoint, no group can have the same name as a channel and vice versa. Types indicate the group a channel belongs to and the types of the variables exchanged through that channel. Types follow this syntax:

$$T = G[T_1, \dots, T_n] \mid [\,]$$

We define the environment Γ as a list of both groups G and pairs $(x : T)$ where x is a channel name and T is its type.

$$\Gamma = \emptyset \mid \Gamma, (x : T) \mid \Gamma, G$$

When defining a channel we can specify it as belonging to a group or not. For a channel to be bound to a group G we require G to be an already defined group. When adding a group we require its name not to be an already defined group name.

A process P is defined as follows:

$P = \emptyset$	null process
$\mid\ P \mid P$	parallel composition
$\mid\ x\langle\overrightarrow{y}\rangle$	send channels $y_{1 \dots n}$ on channel x
$\mid$ **def** D **in** P	process definition
$\mid$ **grp** G **for** P	group creation

A definition D is defined as follows:

$D = D \wedge D$	definition conjunction
$\mid\ J \triangleright P$	reaction pattern, if J is matched, start P

A join pattern J is defined as follows:

$J = J \mid J$	synchronization pattern
$\mid\ x_G\langle\overrightarrow{y}\rangle$	message definition.
$\mid\ x\langle\overrightarrow{y}\rangle$	message definition without group.

The difference from the standard join calculus lies in two points. As for ownership types, *message definition* rule has an added parameter that must be an already specified group. Process definition has an extra rule *group creation* for defining new groups. This rule creates a new group G whose scope is P.

Figure 6.1 defines two new kind of variables here: *defined group(dg)* and *free group(fg)*. Refer to Figure 3.1 for the other kind of variables.

6.1.1 Structural equivalence

By adding a new syntax rule for processes we have to add a set of equivalence rules that deals with them. Figure 6.2 points out such rules. The equivalence for groups are taken from the original paper [CGG05] that shows them while the other rules are just standard join calculus and can be found in Figure 3.2.

$$dg(x_g\langle\overrightarrow{y}\rangle) = \{g\}$$

$$fg(\textbf{grp } G \textbf{ for } P) = dg(P) \setminus \{G\}$$

Figure 6.1: Definition of free and defined groups.

$$\begin{array}{rcll}
\textbf{grp } G \textbf{ for grp } G' \textbf{ for } P & \equiv & \textbf{grp } G' \textbf{ for grp } G \textbf{ for } P & \\
\textbf{def } D \textbf{ in grp } G \textbf{ for } P & \equiv & \textbf{grp } G \textbf{ for def } D \textbf{ in } P & \textit{if } G \notin fg(D) \\
\textbf{grp } G \textbf{ for } (P \mid P') & \equiv & P \mid \textbf{grp } G \textbf{ for } P' & \textit{if } G \notin fg(P) \\
\textbf{grp } G \textbf{ for } P & \equiv & P & \textit{if } G \notin fg(P)
\end{array}$$

Figure 6.2: Structural equivalence for groups.

6.2 Typing judgments

These are the typing judgments:

$\Gamma \vdash \diamond$	Γ is a well typed environment.
$\Gamma \vdash P$	P is a well typed process in Γ.
$\Gamma \vdash D :: \Gamma'$	D is a well typed definition in Γ and Γ' contains the bindings for $dv(D)$.
$\Gamma \vdash J :: \Gamma'$	J is a well typed join pattern in Γ and Γ' contains the bindings for $dv(J)$.
$\Gamma \vdash x : T$	channel x has type T in Γ.
$\Gamma \vdash T$	T is a valid type in Γ.

6.3 Typing rules

Figure 6.3 shows the typing rules for this type system. We omit rules which are the same as those seen in the type systems in Chapter 4. There is only a new rule concerning environments: *Env-grp*. It states that we can add a group to the environment only if it is not a duplicate of an existing group. For a general type to be well formed, rule *Type* controls the group name G to be in the typechecking environment. All the mentioned

Good environments

$$\frac{}{\emptyset \vdash \diamond}\ \textit{Env-null} \qquad \frac{\Gamma \vdash \diamond \quad G \notin dom(\Gamma)}{\Gamma, G \vdash \diamond}\ \textit{Env-grp}$$

$$\frac{\Gamma \vdash \diamond \quad x \notin dom(\Gamma) \quad \Gamma \vdash T}{\Gamma, (x : T) \vdash \diamond}\ \textit{Env-build}$$

Well-typed channels

$$\frac{\Gamma, (x, G), \Gamma' \vdash \diamond \quad x \notin dom(\Gamma')}{\Gamma, (x, G), \Gamma' \vdash x : G}\ \textit{Chan}$$

Well formed types

$$\frac{\Gamma, G, \Gamma' \vdash \diamond \quad G \notin dom(\Gamma') \quad \forall i(\Gamma, G \vdash F_i)}{\Gamma, G, \Gamma' \vdash G[F_1, \ldots, F_n]}\ \textit{Type} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash [\,]}\ \textit{Type-null}$$

Well-typed processes

$$\frac{\Gamma, G \vdash \diamond \quad \Gamma, G \vdash P}{\mathbf{grp}\ G\ \mathbf{for}\ P}\ \textit{Pgrp} \qquad \frac{\Gamma \vdash x : G[F_1, \ldots, F_n] \quad \forall i(\Gamma \vdash y_i : F_i)}{\Gamma \vdash x\langle \overrightarrow{y} \rangle}\ \textit{Msg}$$

Well-typed join patterns

$$\frac{\Gamma \vdash x : [F_1, \ldots, F_n] \quad \forall i(\Gamma \vdash y_i : F_i)}{\Gamma \vdash x\langle \overrightarrow{y} \rangle :: (x,), (y_1, F_1), \ldots, (y_n, F_n)}\ \textit{Cdef} \quad \frac{\Gamma \vdash x : G[F_1, \ldots, F_n] \quad \forall i(\Gamma \vdash y_i : F_i)}{\Gamma \vdash x_G\langle \overrightarrow{y} \rangle :: (x, G), (y_1, F_1), \ldots, (y_n, F_n)}\ \textit{Cdef-grp}$$

Figure 6.3: Typing rules for the groups system.

types must be valid in the environment up to the group name G. For an empty type, rule *Type-null* requires the environment to be well typed. Rule *Pgrp* deals with group creation. When creating a group we ensure that the environment augmented by such group is well formed. We then check the process in scope in the environment augmented by the newly defined group. The message sending rule *Msg* as well as channel definition rules *Cdef* and *Cdef-grp* are analogous to those presented in the previous systems.

6.4 Semantics

Here we use a RCHAM similar to the one presented in Section 5.4. The modifications developed to handle groups are similar to those needed to handle contexts. Since we have a term for group creation, we introduce a structural rule S-GRP to handle it. We indicate a set of groups with the letter $\mathcal{G}$. The general rule CTX has been modified to consider groups.

The rules for the RCHAM with groups are described in Figure 6.4.

$$
\begin{array}{llcllr}
 & \Vdash P_1|P_2 & \rightleftharpoons & & \Vdash P_1, P_2 & \text{S-PAR} \\
 & \Vdash \textbf{def } D \textbf{ in } P & \rightleftharpoons & D & \Vdash P & \text{S-DEF} \\
 & & & & dv(D) \text{ are fresh.} & \\
 & \Vdash \textbf{grp } G \textbf{ for } P & \rightleftharpoons & G & \Vdash P & \text{S-GRP} \\
 & & & & G \text{ is fresh.} & \\
D \wedge J \triangleright P \wedge D' & \Vdash \mathcal{J} & \longrightarrow & D \wedge J \triangleright P \wedge D' & \Vdash \varphi(P, \mathcal{J}, J) & \text{R-BETA} \\
 & & & & dv(J){=}dv(\mathcal{J}). &
\end{array}
$$

$$
\frac{\begin{array}{c}\mathcal{D}_1, \mathcal{G}_1 \Vdash \mathcal{P}_1 \rightleftharpoons\!\!\!\!\!\rightarrow \mathcal{D}_2, \mathcal{G}_2 \Vdash \mathcal{P}_2 \\ (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset \\ \mathcal{G} \cap (\mathcal{G}_1 \cup \mathcal{G}_2) = \emptyset \end{array}}{\mathcal{D}, \mathcal{G}, \mathcal{D}_1, \mathcal{G}_1 \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons\!\!\!\!\!\rightarrow \mathcal{D}, \mathcal{G}, \mathcal{D}_2, \mathcal{G}_2 \Vdash \mathcal{P}_2, \mathcal{P}} \ CTX
$$

Figure 6.4: Chemical rules for the RCHAM with groups.

Rule S-GRP has a side condition that mimics the scope extrusion principle found in the π-calculus. When unfolding the group creation token, we map the group name G to a fresh name. This changes the group name only within its scope P avoiding conflicts with outer citations.

6.5 Additional typing judgments and rules

Since typing is extended to chemical solutions we have a new typing judgment and new typing rules. The additional typing judgment concerning well typed soups is the following.

$\Gamma \vdash \mathcal{D}, \mathcal{G} \Vdash \mathcal{P}$	The chemical solution $\mathcal{D}, \mathcal{G} \Vdash \mathcal{P}$ is well typed in Γ.

We present the only typing rule that differs from those reported in Figure 4.6.

Rule *Soup* states that a soup is well typed if the group names it defines can be added to the environment without compromising it. Also all the definitions and all the processes must be well typed in an environment augmented by both the groups of the soup and the bindings for all the defined variables in the soup's definitions.

$$\frac{\Gamma, \mathcal{G} \vdash \diamond \qquad \Gamma, \mathcal{G}, \Gamma' \vdash \mathcal{D} :: \Gamma' \qquad \Gamma, \mathcal{G}, \Gamma' \vdash \mathcal{P}}{\Gamma \vdash \mathcal{D}, \mathcal{G} \Vdash \mathcal{P}} \; \textit{Soup}$$

Figure 6.5: Additional typing rules for join calculus with groups.

6.6 Properties

The same comments that held for the proofs in Section 5.6 hold here as well. Most of the proofs of the groups system are the same of the ones developed for contexts-as-owners system, since the two are very similar. We avoid reporting such identical proofs and provide only the most useful cases.

Lemma 5 (Useless groups)**.** *Let G be a group name that is not free nor defined in $\mathcal{T}$ or Γ.*

$$\Gamma \vdash \mathcal{T} \Leftrightarrow \Gamma, G \vdash \mathcal{T}$$

Proof. The proof is analogous to that of Lemma 4. □

Theorem 4 (Subject reduction)**.** *One step chemical reductions preserve typings. If $\Gamma \vdash \mathcal{D}, \mathcal{G} \Vdash \mathcal{P}$ and $\mathcal{D}, \mathcal{G} \Vdash \mathcal{P} \rightleftharpoons\!\!\!\rightarrow \mathcal{D}', \mathcal{G}' \Vdash \mathcal{P}'$, then there exists Γ' such that $\Gamma' \vdash \mathcal{D}', \mathcal{G}' \Vdash \mathcal{P}'$ and Γ and Γ' are the same except for the names defined in $\mathcal{D}, \mathcal{G}$ but not in $\mathcal{D}', \mathcal{G}'$ and vice versa.*

Proof. The proof goes by induction on the number of applications of rule CTX in the derivation of the reduction. Note that in this proof we refer to subtrees with the notation Π. Sometimes a single derivation has been split in subproofs where the whole derivation tree would exceed the page.

Base case: Here we consider each reaction rule:

S-PAR: This case is analogous to S-PAR case in the proof of Theorem 1.

S-DEF: This case is analogous to S-DEF case in the proof of Theorem 3.

S-GRP: The reduction is: $\Vdash \textbf{grp}\ G\ \textbf{for}\ P \rightleftharpoons G \Vdash P$.

Heating: The hypothesis is $\Gamma \vdash \ \Vdash \textbf{grp}\ G\ \textbf{for}\ P$ with side condition that G is mapped to a fresh group name. Considering a general soup $\mathcal{E} \Vdash \mathcal{M}, \textbf{grp}\ G\ \textbf{for}\ P$, the following holds: $G \cap fg(\mathcal{E}, \mathcal{M}, P) = \emptyset$.

The derivation is the following: we first apply rule *Soup* and then the only matching rule is *Pgrp*.

$$\dfrac{\dfrac{\Gamma, G \vdash \diamond \qquad \Gamma, G \vdash P}{\Gamma \vdash \mathbf{grp}\ G\ \mathbf{for}\ P}\ \scriptstyle{Pgrp}}{\Gamma \vdash \quad \Vdash \mathbf{grp}\ G\ \mathbf{for}\ P}\ \scriptstyle{Soup}$$

We now have all the premises to apply rule *Soup* and prove the thesis $G \Vdash P$.

$$\dfrac{\Gamma, G \vdash \diamond \qquad \Gamma, G \vdash P}{\Gamma \vdash G \Vdash P}\ \scriptstyle{Soup}$$

The side condition holds since we have $\Gamma, G \vdash P$. This means that there is no other group whose name matches G except G itself.

Cooling: The other way round: first apply *Soup*, then use the assumptions made to apply *Pgrp* and conclude with *Soup*.

R-BETA: This case is analogous to R-BETA case in the proof of Theorem 1.

Inductive case: This case is analogous to the inductive case in the proof of Theorem 3. Of course here instead of using Lemma 4 for useless contexts, we use Lemma 5 to deal with useless groups. □

Property 6 (Secrecy property)**.** *A channel x that belongs to a group G cannot be accessed from outside G's scope. This means that channels introduced before group G cannot name x among its parameters.*

Proof. The proof is analogous to the one of Property 3. □

Chapter 7

Comparison of the type systems

Abstract

In questo capitolo viene fatto il paragone tra i tre sistemi di tipo presentati nei Capitoli 4,5,6. Si fornisce una dimostrazione formale che i tre sistemi hanno lo stesso potere espressivo: gli insiemi di termini tipabili per ogni sistema sono tra loro equivalenti.

In this chapter we compare the three systems from Chapters 4,5,6. The idea we had after formalizing both CtxO and ChaO was that they are very similar to Groups.

Comparing programming languages is not an easy subject, literature contains lots of informal claims on one language being more expressive than a different one [Fel91]. Being able to prove in a formal way the relationship, in terms of expressiveness, that exists between two languages is a tough goal. When dealing with the expressiveness of two type systems we reason about the properties enforced on the well typed programs. If two systems enforce the same properties on the well typed programs, then they are equivalent. Since properties are enforced by typing, comparing systems is made by comparing the set of typeable terms. To prove the equivalence of two different systems, the two sets of well typed terms must be the same.

7.1 Comparison overview

We introduce now some syntactic annotation that will be used in the following.

A subscript L or K is an abstraction for any of the developed system. Thus $L, K \in \{CH, CT, GR\}$. We have developed three languages so far, we shall refer to them with the names J_{CH} for the channels-as-owners model, J_{CT} for the contexts-as-owners model, J_{GR} for the groups model. References to well typed terms will be made by using names such as τ, ψ. To express their belonging to a system L we will write $\tau \in J_L$ or τ_L.

The notation $P \approx Q$ will indicate a process P having the same behavior of another process Q. The relation $\approx$ is called bisimulation and is presented in Section 7.2.

We also introduce the notation $J_K \trianglelefteq J_L$ to express the system J_L being at least as expressive as J_K.

These are two kind of functions that will be defined later, we now give a description of their semantics:

- $[\ ]_L$ is called an *erasure function*. It erases all type and type-related annotations for L;

- $[\![\ ]\!]_L$ is called a *mapping function*. It maps terms, typing environments and annotations from any system $K \neq L$ into L-related ones.

Now that all the notation has been introduced, we can show Definition 5, which states how to compare the expressiveness of two systems.

Definition 5 (System equivalence)**.** *Consider two systems J_L and J_K. If for all $\tau \in J_L$:*

1. $[\tau]_L \approx [[\![\tau]\!]_K]_K$;

2. $\Gamma \vdash \tau \Rightarrow [\![\Gamma \vdash \tau]\!]_K$,

then J_K is at least as powerful as J_L: $J_L \trianglelefteq J_K$

The basic idea is explained in the following. Consider a well typed term τ of a given system J_L, the mapping of τ into another system J_K produces a term ψ.

- If by erasing all type annotations in ψ and τ we obtain two bisimilar processes, we can say that their behavior is the same, ergo the mapping function preserves the process' behavior.

- If τ is well typed in a certain environment Γ and ψ is well typed in the environment we obtain by applying the mapping function to Γ, then we know that the mapping function preserves typing.

If the two conditions above hold for all the possible well typed terms τ of J_L, then we can infer the following property: the set of well typed terms ψ of J_K contains a subset of well typed terms each of which is bisimilar to a well typed term τ of J_L. This means that in J_K we can write at least the same programs of J_L, expressing the same properties.

Now we recap the syntax of the three systems, we introduce the notion of bisimilarity and we define both the erasure and the mapping functions. In the end we will prove the three systems to have the same expressive power.

Token	Syntax of ChaO J_{CH}	Syntax of CtxO J_{CT}	Syntax of Groups J_{GR}
P	$\emptyset$ $P\|P$ $x\langle\overrightarrow{y}\rangle$ **def** D **in** P	$\emptyset$ $P\|P$ $x\langle\overrightarrow{y}\rangle$ **def**$_\alpha$ D **in** P	$\emptyset$ $P\|P$ $x\langle\overrightarrow{y}\rangle$ **def** D **in** P **grp** G **for** P
D	$D \wedge D'$ $J \triangleright P$ $\top$	$D \wedge D'$ $J \triangleright P$ $\top$	$D \wedge D'$ $J \triangleright P$ $\top$
J	$J\|J$ $x_o\langle\overrightarrow{y}\rangle$	$J\|J$ $x_o\langle\overrightarrow{y}\rangle$	$J\|J$ $x_o\langle\overrightarrow{y}\rangle$ $x\langle\overrightarrow{y}\rangle$

Figure 7.1: Syntax definition of ChaO, CtxO and Groups.

7.1.1 Syntax recap

In Chapters 4,5,6 we introduced three denumerable sets for channels, contexts and groups: Σ, Ω, Θ. We assume the three sets to be pairwise disjoint.

Figure 7.1 recalls the syntax of each system. We now introduce a special token for empty definition $\top$ as in the early join calculus [FG00]. We avoided dealing with it in the previous chapters due to its uselessness but now it comes handy since we introduce the idea of equivalence.

The typing judgment for such token is common to all three systems:

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \top}\ \textit{Top}$$

The empty definition $\top$, like the empty process $\emptyset$, does not affect the computation of a chemical solution.

7.2 Bisimilarity equivalence

We now introduce the notion of bisimilarity [PS00] for the standard join calculus. We will use this equivalence to compare terms in pure join calculus. No owners, contexts or groups annotations will be found in the syntax of this section. A whole hierarchy of equivalences can be found in [FG00].

Definition 6 (Bisimilarity $\approx$)**.** *Two processes P and Q are bisimilar if, for every action P can produce via R-BETA, Q can produce the same action via R-BETA and the processes they turn into after the reactions are still bisimilar. And vice versa.*

More formally $P \approx Q$ iff:

- *if $P \xrightarrow{J} P'$ then $Q \rightleftharpoons^* \xrightarrow{J} Q'$ and $P' \approx Q'$;*
- *if $Q \xrightarrow{J} Q'$ then $P \rightleftharpoons^* \xrightarrow{J} P'$ and $Q' \approx P'$*

Where $\rightleftharpoons^*$ means any finite sequence of heating or cooling rules and $\xrightarrow{J}$ represents a reduction which is triggered by sending the parallel composition of messages J.

We point out some implications of the $\approx$ relationship since they will be used in the following.

$$\begin{array}{rr}
D \approx E, P \approx Q \Rightarrow \textbf{def } D \textbf{ in } P \approx \textbf{def } E \textbf{ in } Q & P \approx Q, P' \approx Q' \Rightarrow P|P' \approx Q|Q' \\
D \approx E, D' \approx E' \Rightarrow D \wedge D' \approx E \wedge E' & J \approx J'P \approx P' \Rightarrow J \rhd P \approx J' \rhd P' \\
P|\emptyset \approx P & D \wedge \top \approx D \\
\textbf{def } \top \textbf{ in } P \approx P & P \approx P \\
\textbf{def } D \wedge J \rhd \emptyset \textbf{ in } P \approx \textbf{def } D \textbf{ in } P & \\
\textit{if } dv(J) \cap fv(D, P) = \emptyset &
\end{array}$$

7.3 The erasure function

The erasure function erases anything related to owners, contexts and groups, reducing the syntax of the three systems to the one of the pure join calculus of Chapter 3. Figure 7.2 presents the function in a general flavor since most of the rules would look the same in all three systems.

$$\begin{array}{rr}
[\textbf{def } D \textbf{ in } P]_{CH} = \textbf{def } [D]_{CH} \textbf{ in } [P]_{CH} & [\textbf{def } D \textbf{ in } P]_{GR} = \textbf{def } [D]_{GR} \textbf{ in } [P]_{GR} \\
[\textbf{def}_\alpha\ D \textbf{ in } P]_{CT} = \textbf{def } [D]_{CT} \textbf{ in } [P]_{CT} & [\textbf{grp } G \textbf{ for } P]_{GR} = [P]_{GR}
\end{array}$$

$$\begin{array}{rr}
[\emptyset]_L = \emptyset & [x\langle \overrightarrow{y} \rangle]_L = x\langle \overrightarrow{y} \rangle \\
[P|P']_L = [P]_L|[P']_L & \\
[\top]_L = \top & [D \wedge D']_L = [D]_L \wedge [D']_L \\
[J \rhd P]_L = [J]_L \rhd [P]_L & \\
[J|J']_L = [J]_L|[J']_L & [x_o\langle \overrightarrow{y} \rangle]_L = x\langle \overrightarrow{y} \rangle
\end{array}$$

Figure 7.2: The erasure function.

7.4 The mapping function

In this section we provide the description of the mapping function for every system. We present only the functions used in the proofs of Section 7.5 to avoid generating confusion and to keep the document readable.

The first function maps elements of J_{CT} to elements of J_{GR}, and is stated in Figure 7.3. In the following we will use the token $\mathcal{T}$ to represent any right hand side of a typing judgment.

$$[\![\Gamma \vdash \mathcal{T}]\!]_{GR} = [\![\Gamma]\!]_{GR} \vdash [\![\mathcal{T}]\!]_{GR}$$

Mapping for environments:

$$[\![\Gamma, \alpha]\!]_{GR} = [\![\Gamma]\!]_{GR}, \alpha \qquad [\![\Gamma, (x, \alpha, l)]\!]_{GR} = [\![\Gamma]\!]_{GR}, (x : [\![\alpha : l]\!]_{GR})$$
$$[\![\emptyset]\!]_{GR} = \emptyset$$

Mapping for types and channels:

$$[\![\alpha : l]\!]_{GR} = \alpha[[\![l]\!]_{GR}] \qquad [\![x : \alpha : l]\!]_{GR} = x : [\![\alpha : l]\!]_{GR}$$

Mapping for ownership schemes:

$$[\![\langle\emptyset\rangle]\!]_{GR} = \emptyset \qquad [\![\langle\alpha_1, l_1\rangle, \dots, \langle\alpha_n, l_n\rangle]\!]_{GR} = [\![\alpha_1 : l_1]\!]_{GR}, \dots, [\![\alpha_n : l_n]\!]_{GR}$$

Mapping for syntax tokens:

$$\begin{array}{rlrl}
[\![\emptyset]\!]_{GR} &= \emptyset & [\![x\langle\overrightarrow{y}\rangle]\!]_{GR} &= x\langle\overrightarrow{y}\rangle \\
[\![P|P']\!]_{GR} &= [\![P]\!]_{GR}|[\![P']\!]_{GR} & [\![\mathbf{def}_\alpha\ D\ \mathbf{in}\ P]\!]_{GR} &= \mathbf{grp}\ \alpha\ \mathbf{for\ def}\ [\![D]\!]_{GR}\ \mathbf{in}\ [\![P]\!]_{GR} \\
[\![D \wedge D']\!]_{GR} &= [\![D]\!]_{GR} \wedge [\![D']\!]_{GR} & [\![J \triangleright P]\!]_{GR} &= [\![J]\!]_{GR} \triangleright [\![P]\!]_{GR} \\
[\![\top]\!]_{GR} &= \top & & \\
[\![J|J']\!]_{GR} &= [\![J]\!]_{GR}|[\![J']\!]_{GR} & [\![x_\alpha\langle\overrightarrow{y}\rangle]\!]_{GR} &= x_\alpha\langle\overrightarrow{y}\rangle
\end{array}$$

Figure 7.3: The mapping function from system J_{CT} to J_{GR}.

The second function maps elements of J_{GR} to elements of J_{CH}, and is stated in Figure 7.4. Here we use the notation τ to indicate any syntax token such as P, D, J. To preserve the ordering imposed by the scope of the groups, we annotate some cases with a parameter at the top right corner. Whenever a parameter v appears, it represents the rightmost group of the typechecking environment Γ, if such a group exists, or *world* otherwise.

The last mapping function maps elements of J_{CH} to elements of J_{CT}, and is stated in Figure 7.5. We indicate a context name with an uppercase letter e.g. X. We suppose such a name does not clash with other defined names, i.e is fresh.

$$[\![\Gamma \vdash \mathcal{T}]\!]_{CH} = (world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![\mathcal{T}]\!]_{CH}$$
$$[\![\Gamma \vdash \tau]\!]_{CH} = (world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![\tau]\!]^{v}_{CH}$$

Mapping for environments:

$$[\![\Gamma, G]\!]_{CH} = [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \qquad [\![\Gamma, (x : T)]\!]_{CH} = [\![\Gamma]\!]_{CH}, (x, [\![T]\!]_{CH})$$
$$[\![\emptyset]\!]_{CH} = \emptyset$$

Mapping for types and channels:

$$[\![G[T_1, \ldots, T_N]]\!]_{CH} = G : \langle [\![T_1]\!]_{CH}\rangle, \ldots, \langle [\![T_n]\!]_{CH}\rangle \qquad [\![G[\ ]]\!]_{CH} = G : \langle \emptyset \rangle$$
$$[\![\ [T_1, \ldots, T_n]]\!]_{CH} = world : \langle [\![T_1]\!]_{CH}\rangle, \ldots, \langle [\![T_n]\!]_{CH}\rangle \qquad [\![\ [\ ]]\!]_{CH} = world : \langle \emptyset \rangle$$
$$[\![x : T]\!]_{CH} = x : [\![T]\!]_{CH}$$

Mapping for syntax tokens:

$$[\![\emptyset]\!]^{v}_{CH} = \emptyset \qquad [\![x\langle \overrightarrow{y} \rangle]\!]^{v}_{CH} = x\langle \overrightarrow{y} \rangle$$
$$[\![P|P']\!]^{v}_{CH} = [\![P]\!]^{v}_{CH}|[\![P']\!]^{v}_{CH} \qquad [\![\mathbf{def}\ D\ \mathbf{in}\ P]\!]^{v}_{CH} = \mathbf{def}\ [\![D]\!]^{v}_{CH}\ \mathbf{in}\ [\![P]\!]^{v}_{CH}$$
$$[\![D \wedge D']\!]^{v}_{CH} = [\![D]\!]^{v}_{CH} \wedge [\![D']\!]^{v}_{CH} \qquad [\![J \triangleright P]\!]^{v}_{CH} = [\![J]\!]^{v}_{CH} \triangleright [\![P]\!]^{v}_{CH}$$
$$[\![\top]\!]^{v}_{CH} = \top$$
$$[\![J|J']\!]^{v}_{CH} = [\![J]\!]^{v}_{CH}|[\![J']\!]^{v}_{CH} \qquad [\![x_G\langle \overrightarrow{y} \rangle]\!]^{v}_{CH} = x_G\langle \overrightarrow{y} \rangle$$
$$[\![x\langle \overrightarrow{y} \rangle]\!]^{v}_{CH} = x_{world}\langle \overrightarrow{y} \rangle$$
$$[\![\mathbf{grp}\ G\ \mathbf{for}\ P]\!]^{v}_{CH} = \mathbf{def}\ G_v\langle\rangle \triangleright \emptyset\ \mathbf{in}\ [\![P]\!]^{G}_{CH}$$

Figure 7.4: The mapping function from system J_{GR} to J_{CH}.

$$[\![\Gamma \vdash \mathcal{T}]\!]_{CT} = [\![\Gamma]\!]_{CT} \vdash [\![\mathcal{T}]\!]_{CT}$$

Mapping for environments:

$$[\![(world, world, \emptyset)]\!]_{CT} = \omega \qquad [\![\Gamma, (x, o, l)]\!]_{CT} = [\![\Gamma]\!]_{CT}, X, (x, o, l)$$

Mapping for types and channels:

$$\begin{aligned} [\![o : l]\!]_{CT} &= O : [\![l]\!]_{CT} \\ [\![x : o : l]\!]_{CT} &= x : [\![o : l]\!]_{CT} \end{aligned}$$

Mapping for ownership schemes:

$$[\![\langle\emptyset\rangle]\!]_{CT} = \langle\emptyset\rangle \qquad [\![\langle o_1, l_1\rangle, \ldots, \langle o_n, l_n\rangle]\!]_{CT} = \langle[\![o_1 : l_1]\!]_{CT}\rangle, \ldots, \langle[\![o_n : l_n]\!]_{CT}\rangle$$

Mapping for syntax tokens:

$$\begin{aligned} [\![\emptyset]\!]_{CT} &= \emptyset & [\![x\langle\overrightarrow{y}\rangle]\!]_{CT} &= x\langle\overrightarrow{y}\rangle \\ [\![P|P']\!]_{CT} &= [\![P]\!]_{CT}|[\![P']\!]_{CT} & & \\ [\![D \wedge D']\!]_{CT} &= [\![D]\!]_{CT} \wedge [\![D']\!]_{CT} & [\![J \rhd P]\!]_{CT} &= [\![J]\!]_{CT} \rhd [\![P]\!]_{CT} \\ [\![\top]\!]_{CT} &= \top & & \\ [\![J|J']\!]_{CT} &= [\![J]\!]_{CT}|[\![J']\!]_{CT} & [\![x_o\langle\overrightarrow{y}\rangle]\!]_{CT} &= x_o\langle\overrightarrow{y}\rangle \end{aligned}$$

$$[\![\mathbf{def}\, D \,\mathbf{in}\, P]\!]_{CT} = \mathbf{def}_{X_1} \top \,\mathbf{in}\, \ldots \mathbf{in}\, \mathbf{def}_{X_n} [\![D]\!]_{CT} \,\mathbf{in}\, [\![P]\!]_{CT} \qquad \textit{where } dv(D) = \{X_1, \ldots, X_n\}$$

Figure 7.5: The mapping function from system J_{CH} to J_{CT}.

7.5 Theorems

We now state the three properties whose proof confirms the three systems having the same expressive power. Each single property is an instantiation of Definition 5 with two systems. The combination of all three properties confirms

$$\begin{tikzcd} & J_{CH} & \\ J_{CT} \arrow[ur, phantom, sloped, "\trianglelefteq"] \arrow[rr, phantom, "\trianglelefteq"] & & J_{GR} \arrow[ul, phantom, sloped, "\trianglerighteq"] \end{tikzcd}$$

which can be stated in a more linear way as: $J_{CT} \trianglelefteq J_{GR} \trianglelefteq J_{CH} \trianglelefteq J_{CT}$.

Property 7 ($J_{CT} \trianglelefteq J_{GR}$)**.** *For all $\tau \in J_{CT}$:*

1. $[\tau]_{CT} \approx [[\![\tau]\!]_{GR}]_{GR}$*;*

2. $\Gamma \vdash \tau : T \Rightarrow [\![\Gamma \vdash \tau : T]\!]_{GR}$.

Property 8 ($J_{GR} \trianglelefteq J_{CH}$)**.** *For all $\tau \in J_{GR}$:*

1. $[\tau]_{GR} \approx [[\![\tau]\!]_{CH}]_{CH}$*;*

2. $\Gamma \vdash \tau : T \Rightarrow [\![\Gamma \vdash \tau : T]\!]_{CH}$.

Property 9 ($J_{CH} \trianglelefteq J_{CT}$)**.** *For all $\tau \in J_{CH}$:*

1. $[\tau]_{CH} \approx [[\![\tau]\!]_{CT}]_{CT}$*;*

2. $\Gamma \vdash \tau : T \Rightarrow [\![\Gamma \vdash \tau : T]\!]_{CT}$.

To keep the demonstration readable we provide six theorems, each for one point of every property. For all demonstrations of point (2) we will indicate the application of the mapping function for a system L as follows.

$$\frac{[\![X, Y]\!]_L}{[\![X]\!]_L, [\![Y]\!]_L}\ [\![\ ]\!]_L$$

Where X, Y are generic expression of a given system. With this notation we are able to develop proof trees mixing both typing rules and mapping functions evaluation.

Theorem 5 (Point (1) for $J_{CT} \trianglelefteq J_{GR}$)**.** *For all $\tau \in J_{CT}$ $[\tau]_{CT} \approx [[\![\tau]\!]_{GR}]_{GR}$.*

Proof. The proof is by structural induction on τ.

Base case.

$\tau \equiv \emptyset$ We have to prove $[\emptyset]_{CT} \approx [[\![\emptyset]\!]_{GR}]_{GR}$.

Now: $[\emptyset]_{CT} = \emptyset$, and $[[\![\emptyset]\!]_{GR}]_{GR} = [\emptyset]_{GR} = \emptyset$. The thesis is immediate: $\emptyset \approx \emptyset$ holds by definition of $\approx$.

$\tau \equiv \top$ This case follows the same pattern of the one above.

$\tau \equiv x\langle\overrightarrow{y}\rangle$ We have to prove $[x\langle\overrightarrow{y}\rangle]_{CT} \approx [[\![x\langle\overrightarrow{y}\rangle]\!]_{GR}]_{GR}$.

Now $[x\langle\overrightarrow{y}\rangle]_{CT} = x\langle\overrightarrow{y}\rangle$, and $[[\![x\langle\overrightarrow{y}\rangle]\!]_{GR}]_{GR} = [x\langle\overrightarrow{y}\rangle]_{GR} = x\langle\overrightarrow{y}\rangle$. The thesis is immediate: $x\langle\overrightarrow{y}\rangle \approx x\langle\overrightarrow{y}\rangle$ holds by definition of $\approx$.

$\tau \equiv x_\alpha\langle\overrightarrow{y}\rangle$ This case follows the same pattern of the one above.

Inductive case.

$\tau \equiv P|P'$ We have to prove $[P|P']_{CT} \approx [\llbracket P|P' \rrbracket_{GR}]_{GR}$.

For the left side of the equation the following holds: $[P|P']_{CT} = [P]_{CT}|[P']_{CT}$.

By application of the inductive hypothesis we have: $[P]_{CT} \overset{IH}{\approx} [\llbracket P \rrbracket_{GR}]_{GR}$, and $[P']_{CT} \overset{IH}{\approx} [\llbracket P' \rrbracket_{GR}]_{GR}$.

Due to the IH we recall the definition of $\approx$ and we can say $[P]_{CT}|[P']_{CT} \approx [\llbracket P \rrbracket_{GR}]_{GR}|[\llbracket P' \rrbracket_{GR}]_{GR}$.

At this point the thesis follows by definition of the erasure function.

$\tau \equiv \mathbf{def}_\alpha\ D\ \mathbf{in}\ P$ We have to prove $[\mathbf{def}_\alpha\ D\ \mathbf{in}\ P]_{CT} \approx [\llbracket \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \rrbracket_{GR}]_{GR}$.

For the left side of the equation we have: $[\mathbf{def}_\alpha\ D\ \mathbf{in}\ P]_{CT} = \mathbf{def}\ [D]_{CT}\ \mathbf{in}\ [P]_{CT}$.

This gives us the inductive hypotheses $[D]_{CT} \overset{IH}{\approx} [\llbracket D \rrbracket_{GR}]_{GR}$ and $[P]_{CT} \overset{IH}{\approx} [\llbracket P \rrbracket_{GR}]_{GR}$.

On the right side we have: $[\llbracket \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \rrbracket_{GR}]_{GR} = [\mathbf{grp}\ \alpha\ \mathbf{for}\ \mathbf{def}\ \llbracket D \rrbracket_{GR}\ \mathbf{in}\ \llbracket P \rrbracket_{GR}]_{GR}$.

By application of the erasure function we obtain $\mathbf{def}\ [\llbracket D \rrbracket_{GR}]_{GR}\ \mathbf{in}\ [\llbracket P \rrbracket_{GR}]_{GR}$.

The thesis becomes $\mathbf{def}\ [D]_{CT}\ \mathbf{in}\ [P]_{CT} \approx \mathbf{def}\ [\llbracket D \rrbracket_{GR}]_{GR}\ \mathbf{in}\ [\llbracket P \rrbracket_{GR}]_{GR}$, which is true due to the inductive hypotheses by definition of $\approx$.

Other cases The remaining cases: $D \wedge D'$, $J|J'$, $J \rhd P$ follow the same pattern of the first inductive case.

□

Theorem 6 (Point (2) for $J_{CT} \trianglelefteq J_{GR}$). *For all* $\tau \in J_{CT}$ $\Gamma \vdash \tau \Rightarrow \llbracket \Gamma \vdash \tau \rrbracket_{GR}$.

Proof. The proof goes on induction of the typing tree of τ.

Base case. The hypothesis for the base case is $\omega \vdash \diamond$. We have to prove $\llbracket \omega \vdash \diamond \rrbracket_{GR}$. By evaluating the mapping function in the thesis we obtain $\llbracket \omega \vdash \diamond \rrbracket_{GR} = \llbracket \omega \rrbracket_{GR} \vdash \diamond = \omega \vdash \diamond$. We can prove the thesis by applying rule *Env-grp* and *Env-null* both of system J_{GR}.

$$\dfrac{\dfrac{}{\emptyset \vdash \diamond}\,\textit{Env-null} \qquad \omega \notin \{\emptyset\}}{\omega \vdash \diamond}\,\textit{Env-grp}$$

Inductive case. The inductive cases are shown in the list below. Every item in the list will have a label that has the following shape: *hypothesis⇒thesis*. When we use the word hypothesis we will refer to the left side of the implication, which will be a judgment in the J_{CT} system. The word thesis will refer to the right side of the implication, which will be a judgment in the J_{GR} system.

$\Gamma, \alpha \vdash \diamond \Rightarrow \llbracket \Gamma, \alpha \vdash \diamond \rrbracket_{GR}$ By applying rule *Env-ctx* to the hypothesis we develop the following tree:

$$\dfrac{\Gamma \vdash \diamond \qquad \alpha \notin \mathit{dom}(\Gamma)}{\Gamma, \alpha \vdash \diamond}\;\textit{Env-ctx}$$

That gives us these inductive hypotheses: $[\![\Gamma \vdash \diamond]\!]_{GR}$ and $\alpha \notin \mathit{dom}([\![\Gamma]\!]_{GR})$. For the definition of $[\![\ ]\!]_{GR}$, the first inductive hypothesis becomes $[\![\Gamma]\!]_{GR} \vdash \diamond$, we can use such hypotheses to apply rule *Env-grp* as follows.

$$\dfrac{\dfrac{\dfrac{[\![\Gamma]\!]_{GR} \vdash \diamond \qquad \alpha \notin \mathit{dom}([\![\Gamma]\!]_{GR})}{[\![\Gamma]\!]_{GR}, \alpha \vdash \diamond}\;\textit{Env-grp}}{[\![\Gamma, \alpha]\!]_{GR} \vdash \diamond}\;[\![\ ]\!]_{GR}}{[\![\Gamma, \alpha \vdash \diamond]\!]_{GR}}\;[\![\ ]\!]_{GR}$$

$\Gamma, (x, \alpha, l) \vdash \diamond \Rightarrow [\![\Gamma, (x, \alpha, l) \vdash \diamond]\!]_{GR}$ By applying rule *Env-build* to the hypothesis we develop the following tree:

$$\dfrac{\Gamma \vdash \diamond \qquad x \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash \alpha : l}{\Gamma, (x, \alpha, l) \vdash \diamond}\;\textit{Env-build}$$

That gives us the following inductive hypotheses: $[\![\Gamma \vdash \diamond]\!]_{GR}$, $x \notin \mathit{dom}([\![\Gamma]\!]_{GR})$ and $[\![\Gamma \vdash \alpha : l]\!]_{GR}$. We can use them to apply rule *Env-build* and develop the following tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\![\Gamma \vdash \diamond]\!]_{GR}}{[\![\Gamma]\!]_{GR} \vdash \diamond}\;[\![\ ]\!]_{GR} \qquad x \notin \mathit{dom}([\![\Gamma]\!]_{GR}) \qquad \dfrac{[\![\Gamma \vdash \alpha : l]\!]_{GR}}{[\![\Gamma]\!]_{GR} \vdash [\![\alpha : l]\!]_{GR}}\;[\![\ ]\!]_{GR}}{[\![\Gamma]\!]_{GR}, (x : [\![\alpha, l]\!]_{GR}) \vdash \diamond}\;\textit{Env-build}}{[\![\Gamma, (x, \alpha, l)]\!]_{GR} \vdash \diamond}\;[\![\ ]\!]_{GR}}{[\![\Gamma, (x, \alpha, l) \vdash \diamond]\!]_{GR}}\;[\![\ ]\!]_{GR}$$

$\Gamma, \alpha, \Gamma' \vdash \alpha : l \Rightarrow [\![\Gamma, \alpha, \Gamma' \vdash \alpha : l]\!]_{GR}$ We can apply rule *Type* to the hypothesis and develop the following tree:

$$\dfrac{\Gamma, \alpha, \Gamma' \vdash \diamond \qquad \alpha \notin \mathit{dom}(\Gamma') \qquad \Gamma, \alpha \vdash l}{\Gamma, \alpha, \Gamma' \vdash \alpha : l}\;\textit{Type}$$

That gives us the inductive hypotheses we use to apply rule *Type* as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\![\Gamma, \alpha, \Gamma' \vdash \diamond]\!]_{GR}}{[\![\Gamma, \alpha, \Gamma']\!]_{GR} \vdash \diamond}\;[\![\ ]\!]_{GR}}{[\![\Gamma]\!]_{GR}, \alpha, [\![\Gamma']\!]_{GR} \vdash \diamond}\;[\![\ ]\!]_{GR} \qquad \alpha \notin \mathit{dom}([\![\Gamma']\!]_{GR}) \qquad \dfrac{\dfrac{[\![\Gamma, \alpha \vdash l]\!]_{GR}}{[\![\Gamma, \alpha]\!]_{GR} \vdash [\![l]\!]_{GR}}\;[\![\ ]\!]_{GR}}{[\![\Gamma]\!]_{GR}, \alpha \vdash [\![l]\!]_{GR}}\;[\![\ ]\!]_{GR}}{[\![\Gamma]\!]_{GR}, \alpha, [\![\Gamma']\!]_{GR} \vdash \alpha[[\![l]\!]_{GR}]}\;\textit{Type}}{[\![\Gamma, \alpha, \Gamma']\!]_{GR} \vdash \alpha[[\![l]\!]_{GR}]}\;[\![\ ]\!]_{GR}}{[\![\Gamma, \alpha, \Gamma']\!]_{GR} \vdash [\![\alpha : l]\!]_{GR}}\;[\![\ ]\!]_{GR}}{[\![\Gamma, \alpha, \Gamma' \vdash \alpha : l]\!]_{GR}}\;[\![\ ]\!]_{GR}$$

$\Gamma \vdash \langle\emptyset\rangle \Rightarrow [\![\Gamma \vdash \langle\emptyset\rangle]\!]_{GR}$ We can apply rule *Scheme-null* to the hypothesis and develop the following tree:

$$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \langle\emptyset\rangle}\ \textit{Scheme-null}$$

That gives us the inductive hypothesis to apply rule *Scheme-null* as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\![\Gamma \vdash \diamond]\!]_{GR}}{[\![\Gamma]\!]_{GR} \vdash \diamond}\ [\![\ ]\!]_{GR}}{[\![\Gamma]\!]_{GR} \vdash \langle\emptyset\rangle}\ \textit{Scheme-null}}{[\![\Gamma]\!]_{GR} \vdash [\![\langle\emptyset\rangle]\!]_{GR}}\ [\![\ ]\!]_{GR}}{[\![\Gamma \vdash \langle\emptyset\rangle]\!]_{GR}}\ [\![\ ]\!]_{GR}$$

$\Gamma \vdash \langle\alpha_1, l_1\rangle, \ldots, \langle\alpha_n, l_n\rangle \Rightarrow [\![\Gamma \vdash \langle\alpha_1, l_1\rangle, \ldots, \langle\alpha_n, l_n\rangle]\!]_{GR}$ We can apply rule *Scheme* to the hypothesis and develop the following tree:

$$\dfrac{\forall i(\Gamma \vdash \alpha_i : l_i)}{\Gamma \vdash \langle\alpha_1, l_i\rangle, \ldots, \langle\alpha_n, l_n\rangle}\ \textit{Scheme}$$

That gives us the following inductive hypothesis: $\forall i([\![\Gamma \vdash \alpha_i : l_i]\!]_{GR})$.

By definition of $[\![\ ]\!]_{GR}$ we obtain $\forall i([\![\Gamma]\!]_{GR} \vdash [\![\alpha_i : l_i]\!]_{GR})$.

We can eliminate the universal quantifier and write the hypothesis in a more linear way as: $[\![\Gamma]\!]_{GR} \vdash [\![\alpha_1 : l_1]\!]_{GR}, \ldots, [\![\alpha_n : l_n]\!]_{GR}$.

The thesis follows for definition of $[\![\ ]\!]_{GR}$: $[\![\Gamma]\!]_{GR} \vdash [\![\alpha_1 : l_1]\!]_{GR}, \ldots, [\![\alpha_n : l_n]\!]_{GR} = [\![\Gamma]\!]_{GR} \vdash [\![\langle\alpha_1, l_1\rangle, \ldots, \langle\alpha_n, l_n\rangle]\!]_{GR} = [\![\Gamma, \vdash \langle\alpha_1, l_1\rangle, \ldots, \langle\alpha_n, l_n\rangle]\!]_{GR}$.

$\Gamma, (x, \alpha, l), \Gamma' \vdash x : \alpha : l \Rightarrow [\![\Gamma, (x, \alpha, l), \Gamma' \vdash x : \alpha : l]\!]_{GR}$ We can apply rule *Chan* and obtain the tree:

$$\dfrac{\Gamma, (x, \alpha, l), \Gamma' \vdash \diamond \qquad x \notin \textit{dom}(\Gamma')}{\Gamma, (x, \alpha, l), \Gamma' \vdash \diamond}\ \textit{Chan}$$

We now have the inductive hypotheses to apply rule *Chan* as it follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\![\Gamma, (x, \alpha, l), \Gamma' \vdash \diamond]\!]_{GR}}{[\![\Gamma, (x, \alpha, l), \Gamma']\!]_{GR} \vdash \diamond}\ [\![\ ]\!]_{GR}}{[\![\Gamma]\!]_{GR}, (x, [\![\alpha : l]\!]_{GR}), [\![\Gamma']\!]_{GR} \vdash \diamond}\ [\![\ ]\!]_{GR} \qquad x \notin \textit{dom}([\![\Gamma']\!]_{GR})}{[\![\Gamma]\!]_{GR}, (x, [\![\alpha : l]\!]_{GR}), [\![\Gamma']\!]_{GR} \vdash x : [\![\alpha : l]\!]_{GR}}\ \text{Chan}}{[\![\Gamma, (x, \alpha, l), \Gamma']\!]_{GR} \vdash x : [\![\alpha : l]\!]_{GR}}\ [\![\ ]\!]_{GR}}{[\![\Gamma, (x, \alpha, l), \Gamma' \vdash x : \alpha : l]\!]_{GR}}\ [\![\ ]\!]_{GR}$$

$\Gamma \vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \Rightarrow \llbracket \Gamma \vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \rrbracket_{GR}$ We can apply rule *Pdef* to the hypothesis and obtain the following tree:

$$\dfrac{\Gamma, \alpha, \Gamma' \vdash \diamond \qquad \Gamma, \alpha, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \alpha, \Gamma' \vdash P}{\Gamma \vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P}\ {\scriptstyle P\text{-}def}$$

That gives us the following inductive hypotheses: $\llbracket \Gamma, \alpha, \Gamma' \vdash \diamond \rrbracket_{GR}$, $\llbracket \Gamma, \alpha, \Gamma' \vdash D :: \Gamma' \rrbracket_{GR}$ and $\llbracket \Gamma, \alpha, \Gamma' \vdash P \rrbracket_{GR}$.

The derivation of the thesis is provided in the following tree, all the hypotheses are given from the induction principle so the proof holds. We skip some tedious passages that have appeared several times in the previous cases to make the proof more readable.

$$\dfrac{\dfrac{\dfrac{\dfrac{\llbracket \Gamma, \alpha, \Gamma' \rrbracket_{GR} \vdash \diamond}{\llbracket \Gamma \rrbracket_{GR}, \alpha \vdash \diamond}\ {\scriptstyle \llbracket\ \rrbracket_{GR}} \qquad \dfrac{\dfrac{\llbracket \Gamma, \alpha, \Gamma' \rrbracket_{GR} \vdash \llbracket D \rrbracket_{GR} :: \llbracket \Gamma' \rrbracket_{GR} \qquad \llbracket \Gamma, \alpha, \Gamma' \rrbracket_{GR} \vdash \llbracket P \rrbracket_{GR}}{\llbracket \Gamma, \alpha \rrbracket_{GR} \vdash \mathbf{def}\ \llbracket D \rrbracket_{GR}\ \mathbf{in}\ \llbracket P \rrbracket_{GR}}\ {\scriptstyle Pdef}}{\llbracket \Gamma \rrbracket_{GR}, \alpha \vdash \mathbf{def}\ \llbracket D \rrbracket_{GR}\ \mathbf{in}\ \llbracket P \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}}}{\llbracket \Gamma \rrbracket_{GR} \vdash \mathbf{grp}\ \alpha\ \mathbf{for}\ \mathbf{def}\ \llbracket D \rrbracket_{GR}\ \mathbf{in}\ \llbracket P \rrbracket_{GR}}\ {\scriptstyle Pgrp}}{\llbracket \Gamma \rrbracket_{GR} \vdash \llbracket \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}}}{\llbracket \Gamma \vdash \mathbf{def}_\alpha\ D\ \mathbf{in}\ P \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}}$$

$\Gamma \vdash P|P' \Rightarrow \llbracket \Gamma \vdash P|P' \rrbracket_{GR}$ We can apply rule *Par* to the hypothesis and develop the following tree:

$$\dfrac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P|P'}\ {\scriptstyle Par}$$

Such derivation gives us the inductive hypotheses to apply *Par* and prove the thesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{\llbracket \Gamma \vdash P \rrbracket_{GR}}{\llbracket \Gamma \rrbracket_{GR} \vdash \llbracket P \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}} \qquad \dfrac{\llbracket \Gamma \vdash P' \rrbracket_{GR}}{\llbracket \Gamma \rrbracket_{GR} \vdash \llbracket P' \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}}}{\llbracket \Gamma \rrbracket_{GR} \vdash \llbracket P \rrbracket_{GR} | \llbracket P' \rrbracket_{GR}}\ {\scriptstyle Par}}{\llbracket \Gamma \rrbracket_{GR} \vdash \llbracket P|P' \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}}}{\llbracket \Gamma \vdash P|P' \rrbracket_{GR}}\ {\scriptstyle \llbracket\ \rrbracket_{GR}}$$

The cases: $D \wedge D'$, $J|J'$, $J \triangleright P$, $x_o\langle \overrightarrow{y} \rangle$, $x\langle \overrightarrow{y} \rangle$, follow the same pattern.

$\square$

7.5.1 Typing rule changes

In order to keep the following proof readable, without having to introduce too many lemmas, we change the typing rule *Type* in the J_{CH} system.

$$\frac{\Gamma, (o, w, l), \Gamma' \vdash \diamond \qquad o \notin dom(\Gamma') \qquad \forall i (\Gamma, (o, w, l) \vdash l[i])}{\Gamma, (o, w, l), \Gamma' \vdash o : l} \text{ Type-new}$$

This eliminates the need for the relationship $\prec$: and all the related typing judgments and rules as well. Such relationship had been introduced to maintain the ordering explicit as it is in ownership types systems. It is easy to see that the two are encodings of the same property, a channel's scheme may not mention channels that have been introduced after the channel's owner.

Theorem 7 (Point (1) for $J_{GR} \trianglelefteq J_{CH}$). *For all* $\tau \in J_{GR}$ $[\tau]_{GR} \approx [[\![\tau]\!]_{CH}]_{CH}$.

Proof. The proof goes on structural induction on τ.

Base case. The base case follows the same structure of the one in Theorem 5.

Inductive case. Most of the inductive cases follow the structure in Theorem 5. Here we present only two significant ones.

$\tau \equiv$ **def** D **in** P We have to prove $[\textbf{def } D \textbf{ in } P]_{GR} \approx [[\![\textbf{def } D \textbf{ in } P]\!]^v_{CH}]_{CH}$.

For the left side of the equation the following holds: $[\textbf{def } D \textbf{ in } P]_{GR} = \textbf{def } [D]_{GR} \textbf{ in } [P]_{GR}$.

The inductive hypothesis tells us both: $[D]_{GR} \overset{IH}{\approx} [[\![D]\!]^v_{CH}]_{CH}$ and $[P]_{GR} \overset{IH}{\approx} [[\![P]\!]^v_{CH}]_{CH}$ for all possible values of v.

By definition of $\approx$ we are allowed to say $\textbf{def } [D]_{GR} \textbf{ in } [P]_{GR} \approx \textbf{def } [[\![D]\!]^v_{CH}]_{CH} \textbf{ in } [[\![P]\!]^v_{CH}]_{CH}$.

At this point the thesis follows by definition of the erasure function.

$\tau \equiv$ **grp** G **for** P We have to prove: $[\textbf{grp } G \textbf{ for } P]_{GR} \approx [[\![\textbf{grp } G \textbf{ for } P]\!]^v_{CH}]_{CH}$.

For the left side of the equation we have: $[\textbf{grp } G \textbf{ for } P]_{GR} = [P]_{GR}$.

This gives us the inductive hypothesis: $[P]_{GR} \overset{IH}{\approx} [[\![P]\!]^v_{CH}]_{CH}$ for all v.

For the right side of the equation we have: $[[\![\textbf{grp } G \textbf{ for } P]\!]^v_{CH}]_{CH} = [\textbf{def } G\langle v\rangle \rhd \emptyset \textbf{ in } [\![P]\!]^G_{CH}]_{CH}$.

By definition of $\approx$: $\textbf{def } G\langle v\rangle \rhd \emptyset \textbf{ in } P \approx P$ if G is free in P.

Of course this is true since G is a group name and variables are not allowed to be named as groups.

By transitivity of $\approx$ we prove the thesis: $[P]_{GR} \overset{IH}{\approx} [[\![P]\!]^v_{CH}]_{CH} \approx \textbf{def } G\langle v\rangle \rhd \emptyset \textbf{ in } [\![P]\!]^G_{CH}$

□

Theorem 8 (Point (2) for $J_{GR} \trianglelefteq J_{CH}$)**.** *For all* $\tau \in J_{GR}$ $\Gamma \vdash \tau \Rightarrow \llbracket \Gamma \vdash \tau \rrbracket_{CH}$.

Proof. The proof goes on induction of the typing tree of τ.

Base case. The hypothesis for this case is $\emptyset \vdash \diamond$. We have to prove $\llbracket \emptyset \vdash \diamond \rrbracket_{CH}$.
By evaluating the mapping function in the thesis we obtain:
$\llbracket \emptyset \vdash \diamond \rrbracket_{CH} = (\mathit{world}, \mathit{world}, \emptyset)\llbracket \emptyset \rrbracket_{CH} \vdash \diamond = (\mathit{world}, \mathit{world}, \emptyset) \vdash \diamond$.
We can prove the thesis by applying the axiom *Env-null.*

Inductive case. We will use the same notation adopted in Theorem 6.

$\Gamma, G \vdash \diamond \Rightarrow \llbracket \Gamma, G \vdash \diamond \rrbracket_{CH}$ By applying rule *Env-grp* to the hypothesis we obtain:

$$\dfrac{\Gamma \vdash \diamond \qquad G \notin \mathit{dom}(\Gamma)}{\Gamma, G \vdash \diamond}\ \scriptstyle\mathit{Env\text{-}grp}$$

This gives us the following inductive hypotheses: $\llbracket \Gamma \vdash \diamond \rrbracket_{CH}$ and $G \notin \mathit{dom}(\llbracket \Gamma \rrbracket_{CH})$.

The following tree proves the thesis.

$\Pi_1 =$ $G \notin \mathit{dom}(\llbracket \Gamma \rrbracket_{CH})$ valid by *IH*
$\Pi_2 =$ $v \in \mathit{dom}((\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH})$ valid since v is the rightmost group of Γ or world .

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\llbracket \Gamma \vdash \diamond \rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH} \vdash \diamond}\ \scriptstyle\llbracket\,\rrbracket_{CH} \qquad \dfrac{\dfrac{\dfrac{\llbracket \Gamma \vdash \diamond \rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH} \vdash \diamond}\ \scriptstyle\llbracket\,\rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH} \vdash \langle\emptyset\rangle}\ \scriptstyle\mathit{Scheme\text{-}null} \qquad \Pi_1}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH} \vdash \llbracket v : \emptyset \rrbracket_{CH}} \qquad \Pi_2}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH}, (G, v, \emptyset) \vdash \diamond}\ \scriptstyle\mathit{Env\text{-}build}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma, G \rrbracket_{CH} \vdash \diamond}\ \scriptstyle\llbracket\,\rrbracket_{CH}}{\llbracket \Gamma, G \vdash \diamond \rrbracket_{CH}}\ \scriptstyle\llbracket\,\rrbracket_{CH}}$$

$\Gamma, (x : T) \vdash \diamond \Rightarrow \llbracket \Gamma, (x : T) \vdash \diamond \rrbracket_{CH}$ By applying rule *Env-build* to the hypothesis we obtain the following tree:

$$\dfrac{\Gamma \vdash \diamond \quad x \notin \mathit{dom}(\Gamma) \quad \Gamma \vdash T}{\Gamma, (x : T) \vdash \diamond}\ \scriptstyle\mathit{Env\text{-}build}$$

That gives us the inductive hypotheses needed to prove the thesis:

$$\dfrac{\dfrac{\dfrac{\dfrac{\llbracket \Gamma \vdash \diamond \rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH} \vdash \diamond}\ \scriptstyle\llbracket\,\rrbracket_{CH} \qquad x \notin \mathit{dom}(\llbracket \Gamma \rrbracket_{CH}) \qquad \dfrac{\llbracket \Gamma \vdash T \rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH} \vdash \llbracket T \rrbracket_{CH}}\ \scriptstyle\llbracket\,\rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH}(x, \llbracket T \rrbracket_{CH}) \vdash \diamond}\ \scriptstyle\mathit{Env\text{-}build}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma, (x : T) \rrbracket_{CH} \vdash \diamond}\ \scriptstyle\llbracket\,\rrbracket_{CH}}{\llbracket \Gamma, (x : T) \vdash \diamond \rrbracket_{CH}}\ \scriptstyle\llbracket\,\rrbracket_{CH}$$

$\Gamma, G, \Gamma' \vdash G[T_1, \dots, T_n] \Rightarrow \llbracket \Gamma, G, \Gamma' \vdash G[T_1, \dots, T_n] \rrbracket_{CH}$ By applying rule *Type* to the hypothesis we obtain:

$$\dfrac{\Gamma, G, \Gamma' \vdash \diamond \qquad G \notin \mathit{dom}(\Gamma') \qquad \forall i(\Gamma, G \vdash F_i)}{\Gamma, G, \Gamma' \vdash G[F_1, \dots, F_n]}\ \mathit{Type}$$

That gives us the inductive hypotheses to prove the thesis. The proof has been split in subtrees for it would exceed the page.

$$\Pi_1 = \quad \dfrac{\llbracket \Gamma, G, \Gamma' \vdash \diamond \rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH}, (G, v, \emptyset), \llbracket \Gamma' \rrbracket_{CH} \vdash \diamond}\ \llbracket\ \rrbracket_{CH}$$

$$\Pi_2 = \quad \dfrac{\forall i(\llbracket \Gamma, G \vdash T_i \rrbracket_{CH})}{\forall i(\llbracket \Gamma \rrbracket_{CH}(G, v, \emptyset) \vdash \llbracket T_i \rrbracket_{CH})}\ \llbracket\ \rrbracket_{CH}$$

$$\dfrac{\dfrac{\dfrac{\Pi_1 \qquad G \notin \mathit{dom}(\llbracket \Gamma' \rrbracket_{CH}) \qquad \Pi_2}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH}, (G, v, \emptyset), \llbracket \Gamma' \rrbracket_{CH} \vdash G : \langle \llbracket T_1 \rrbracket_{CH} \rangle, \dots, \langle \llbracket T_n \rrbracket_{CH} \rangle}\ \mathit{Type}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma, G, \Gamma' \rrbracket_{CH} \vdash G : \langle \llbracket T_1 \rrbracket_{CH} \rangle, \dots, \langle \llbracket T_n \rrbracket_{CH} \rangle}\ \llbracket\ \rrbracket_{CH}}{\llbracket \Gamma, G, \Gamma' \vdash G[T_1, \dots, T_n] \rrbracket_{CH}}\ \llbracket\ \rrbracket_{CH}$$

Where v is the rightmost group of Γ or *world* otherwise.

$\Gamma, (x, T), \Gamma' \vdash x : T \Rightarrow \llbracket \Gamma, (x, T), \Gamma' \vdash x : T \rrbracket_{CH}$ By applying rule *Chan* to the hypothesis we obtain the following tree:

$$\dfrac{\Gamma, (x, G), \Gamma' \vdash \diamond \qquad x \notin \mathit{dom}(\Gamma')}{\Gamma, (x, G), \Gamma' \vdash x : G}\ \mathit{Chan}$$

That gives us all the inductive hypotheses that allow us to prove the thesis as follows.

$$\dfrac{\dfrac{\dfrac{\llbracket \Gamma, (x, T), \Gamma' \vdash \diamond \rrbracket_{CH}}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH}, (x, \llbracket T \rrbracket_{CH}), \llbracket \Gamma' \rrbracket^{v}_{CH} \vdash \diamond}\ \llbracket\ \rrbracket_{CH} \qquad x \notin \mathit{dom}(\llbracket \Gamma' \rrbracket^{v}_{CH})}{(\mathit{world}, \mathit{world}, \emptyset), \llbracket \Gamma \rrbracket_{CH}, (x, \llbracket T \rrbracket_{CH}), \llbracket \Gamma' \rrbracket^{v}_{CH} \vdash x : \llbracket T \rrbracket_{CH}}\ \mathit{Chan}}{\llbracket \Gamma, (x, T), \Gamma' \vdash x : T \rrbracket_{CH}}\ \llbracket\ \rrbracket_{CH}$$

Where v is the rightmost group of Γ or *world* otherwise.

$\Gamma \vdash P|P' \Rightarrow \llbracket \Gamma \vdash P|P' \rrbracket_{CH}$ By applying rule *Par* to the hypothesis we develop the following tree:

$$\dfrac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P \mid P'}\ \mathit{Par}$$

That gives us the inductive hypotheses we need to prove the thesis as follows. Note that v is the same since Γ is the same for both branches of the tree.

$$\cfrac{\cfrac{\cfrac{\cfrac{[\![\Gamma \vdash P]\!]_{CH}}{(world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![P]\!]^{v}_{CH}}\,[\![\;]\!]_{CH} \qquad \cfrac{[\![\Gamma \vdash P']\!]_{CH}}{(world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![P']\!]^{v}_{CH}}\,[\![\;]\!]_{CH}}{(world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![P]\!]^{v}_{CH} | [\![P']\!]^{v}_{CH}}\,Par}{(world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![P|P']\!]^{v}_{CH}}\,[\![\;]\!]_{CH}}{[\![\Gamma \vdash P|P']\!]_{CH}}\,[\![\;]\!]_{CH}$$

The same procedure applies also for $D \wedge D'$, $J|J'$, $J \triangleright P$, **def** D **in** P, $x_G\langle\overrightarrow{y}\rangle$, $x\langle\overrightarrow{y}\rangle$.

$\Gamma \vdash \mathbf{grp}\ G\ \mathbf{for}\ P \Rightarrow [\![\Gamma \vdash \mathbf{grp}\ G\ \mathbf{for}\ P]\!]_{CH}$ By applying rule *Pgrp* to the hypothesis we obtain:

$$\cfrac{\Gamma, G \vdash \diamond \qquad \Gamma, G \vdash P}{\mathbf{grp}\ G\ \mathbf{for}\ P}\,Pgrp$$

That gives us the following inductive hypotheses:

- $[\![\Gamma, G \vdash \diamond]\!]_{CH} = (world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash \diamond$
- $[\![\Gamma, G \vdash P]\!]_{CH} = (world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash [\![P]\!]^{G}_{CH}$ since the right-most group of the environment is G.

We now have all the hypotheses to develop the proof for the thesis. Note that the proof has been split in subtrees for it would exceed the page.

$$\Pi_1 = \cfrac{(world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash \diamond}{(world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash \emptyset}\,Null$$

$$\Pi_2 = \cfrac{\cfrac{\cfrac{(world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash \diamond}{(world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash G : v : \emptyset}\,Chan}{(world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash G_v\langle\rangle :: (G, v, \emptyset)}\,Cdef \qquad \Pi_1}{(world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash G_v\langle\rangle \triangleright \emptyset :: (G, v, \emptyset)}\,Run$$

$$\cfrac{\cfrac{\cfrac{G = G \qquad \Pi_2 \qquad (world, world, \emptyset), [\![\Gamma]\!]_{CH}, (G, v, \emptyset) \vdash [\![P]\!]^{G}_{CH}}{(world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash \mathbf{def}\ G_v\langle\rangle \triangleright \emptyset\ \mathbf{in}\ [\![P]\!]^{G}_{CH}}\,Pdef}{(world, world, \emptyset), [\![\Gamma]\!]_{CH} \vdash [\![\mathbf{grp}\ G\ \mathbf{for}\ P]\!]^{v}_{CH}}\,[\![\;]\!]_{CH}}{[\![\Gamma \vdash \mathbf{grp}\ G\ \mathbf{for}\ P]\!]_{CH}}\,[\![\;]\!]_{CH}$$

□

Theorem 9 (Point (1) for $J_{CH} \trianglelefteq J_{CT}$)**.** *For all* $\tau \in J_{CH}$ $[\tau]_{CH} \approx [[\![\tau]\!]_{CT}]_{CT}$.

Proof. The proof goes on structural induction on τ.

Base case. The base case follows the same structure of the one in Theorem 5.

Inductive case. We present the only inductive case that does not follow the structure of the ones in Theorem 5 and Theorem 7.

$\tau \equiv \mathbf{def}\ D\ \mathbf{in}\ P$ We have to prove: $[\mathbf{def}\ D\ \mathbf{in}\ P]_{CH} \approx [[\![\mathbf{def}\ D\ \mathbf{in}\ P]\!]_{CT}]_{CT}$.

For the left side of the equation we have: $[\mathbf{def}\ D\ \mathbf{in}\ P]_{CH} = \mathbf{def}\ [D]_{CH}\ \mathbf{in}\ [P]_{CH}$.

This gives us the inductive hypotheses: $[D]_{CH} \overset{IH}{\approx} [[\![D]\!]_{CT}]_{CT}$ and $[P]_{CH} \overset{IH}{\approx} [[\![P]\!]_{CT}]_{CT}$.

For the right side of the equation we have:

$[[\![\mathbf{def}\, D\, \mathbf{in}\, P]\!]_{CT}]_{CT} = [\mathbf{def}_{X_1}\ \top\ \mathbf{in}\ \ldots \mathbf{in}\ \mathbf{def}_{X_n}\ [\![D]\!]_{CT}\ \mathbf{in}\ [\![P]\!]_{CT}]_{CT} = \mathbf{def}\ \top\ \mathbf{in}\ \ldots \mathbf{in}\ \mathbf{def}\ [[\![D]\!]_{CT}]$
Where $dv(D) = \{X_1, \ldots, X_n\}$.

By definition of $\approx$ we have:

$\mathbf{def}\ \top\ \mathbf{in}\ \ldots \mathbf{in}\ \mathbf{def}\ [[\![D]\!]_{CT}]_{CT}\ \mathbf{in}\ [[\![P]\!]_{CT}]_{CT} \approx \mathbf{def}\ [[\![D]\!]_{CT}]_{CT}\ \mathbf{in}\ [[\![P]\!]_{CT}]_{CT}$.

Due to transitivity of $\approx$ we restate the thesis with the form $\mathbf{def}\ [[\![D]\!]_{CT}]_{CT}\ \mathbf{in}\ [[\![P]\!]_{CT}]_{CT}$, which holds thanks to the inductive hypotheses stated above.

$\mathbf{def}\ [D]_{CH}\ \mathbf{in}\ [P]_{CH} \overset{IH}{\approx} \mathbf{def}\ [[\![D]\!]_{CT}]_{CT}\ \mathbf{in}\ [[\![P]\!]_{CT}]_{CT}$

□

Theorem 10 (Point (2) for $J_{CH} \trianglelefteq J_{CT}$). *For all* $\tau \in J_{CH}\ \Gamma \vdash \tau \Rightarrow [\![\Gamma \vdash \tau]\!]_{CT}$.

Proof. The proof goes on induction of the typing tree of τ.

Base case. The hypothesis for the base case is $(\mathit{world}, \mathit{world}, \emptyset) \vdash \diamond$. We have to prove $[\![(\mathit{world}, \mathit{world}, \emptyset) \vdash \diamond]\!]_{CT}$. By evaluating the mapping function in the thesis we obtain: $[\![(\mathit{world}, \mathit{world}, \emptyset) \vdash \diamond]\!]_{CT} = [\![(\mathit{world}, \mathit{world}, \emptyset)]\!]_{CT} \vdash \diamond = \omega \vdash \diamond$ which is true due to axiom *Env-null* of J_{CT}.

Inductive case. We will use the same notation adopted in Theorem 6.

$\Gamma, (x, o, l) \vdash \diamond \Rightarrow [\![\Gamma, (x, o, l) \vdash \diamond]\!]_{CT}$ By applying rule *Env-build* to the hypothesis we have:

$$\frac{\Gamma \vdash \diamond \qquad x \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash o : l}{\Gamma, (x, o, l) \vdash \diamond}\ \textit{Env-build}$$

By using the inductive hypotheses given by the tree above and the definition of $[\![\ ]\!]_{CT}$, we can prove the thesis via the *Ctx* and *Env-build* rules. The hypothesis $X \notin \mathit{dom}(\Gamma)$ holds since we assume no contexts names are allowed as channels name in the J_{CH} system.

$$\dfrac{\dfrac{\dfrac{\dfrac{\llbracket \Gamma \vdash \diamond \rrbracket_{CT}}{\llbracket \Gamma \rrbracket_{CT} \vdash \diamond}\;\llbracket\;\rrbracket_{CT} \qquad X \notin dom(\llbracket \Gamma \rrbracket_{CT})}{\llbracket \Gamma \rrbracket_{CT}, X \vdash \diamond}\;\textit{Ctx} \qquad x \notin dom(\llbracket \Gamma \rrbracket_{CT}, X) \qquad \dfrac{\llbracket \Gamma, X \vdash o : l \rrbracket_{CT}}{\llbracket \Gamma \rrbracket_{CT}, X \vdash \llbracket o : l \rrbracket_{CT}}\;\llbracket\;\rrbracket_{CT}}{\dfrac{\llbracket \Gamma \rrbracket_{CT}, X, (x, \llbracket o : l \rrbracket_{CT}) \vdash \diamond}{\llbracket \Gamma, (x, o, l) \rrbracket_{CT} \vdash \diamond}\;\llbracket\;\rrbracket_{CT}}\;\textit{Env-build}}{\llbracket \Gamma, (x, o, l) \vdash \diamond \rrbracket_{CT}}\;\llbracket\;\rrbracket_{CT}$$

All of the next cases except the last one follow an analogous pattern, to shorten the derivation trees we will cut some tedious passages.

$\Gamma, (x, o, l), \Gamma' \vdash x : o : l \Rightarrow \llbracket \Gamma, (x, o, l), \Gamma' \vdash x : o : l \rrbracket_{CT}$ By applying rule *Chan* to the hypothesis we have:

$$\dfrac{\Gamma, (x, o, l), \Gamma' \vdash \diamond \qquad x \notin dom(\Gamma')}{\Gamma, (x, o, l), \Gamma' \vdash x : o : l}\;\textit{Chan}$$

By using the inductive hypotheses given by the tree above and the definition of $\llbracket\;\rrbracket_{CT}$, we can prove the thesis via the *Chan* rule.

$$\dfrac{\dfrac{\dfrac{\llbracket \Gamma, (x, o, l), \Gamma' \vdash \diamond \rrbracket_{CT}}{\llbracket \Gamma \rrbracket_{CT}, (x, \llbracket o : l \rrbracket_{CT}), \llbracket \Gamma' \rrbracket_{CT} \vdash \diamond}\;\llbracket\;\rrbracket_{CT} \qquad x \notin dom(\llbracket \Gamma' \rrbracket_{CT})}{\llbracket \Gamma, (x, o, l), \Gamma' \rrbracket_{CT} \vdash x : \llbracket o : l \rrbracket_{CT}}\;\textit{Chan}}{\llbracket \Gamma, (x, o, l), \Gamma' \vdash x : o : l \rrbracket_{CT}}\;\llbracket\;\rrbracket_{CT}$$

$\Gamma, (o, w, h), \Gamma' \vdash o : l \Rightarrow \llbracket \Gamma, (o, w, h), \Gamma' \vdash o : l \rrbracket_{CT}$ By applying rule *Type-new* to the hypothesis we have:

$$\dfrac{\Gamma, (o, w, h), \Gamma' \vdash \diamond \qquad o \notin dom(\Gamma') \qquad \forall i (\Gamma, (o, w, h) \vdash l[i])}{\Gamma, (o, w, h), \Gamma' \vdash o : l}\;\textit{Type-new}$$

By using the inductive hypotheses given by the tree above and the definition of $\llbracket\;\rrbracket_{CT}$, we can prove the thesis via the *Type* rule.

$$\dfrac{\dfrac{\llbracket \Gamma, (o, w, h), \Gamma' \rrbracket_{CT} \vdash \diamond \qquad o \notin dom(\llbracket \Gamma' \rrbracket_{CT}) \qquad \forall i (\llbracket \Gamma, (o, w, h) \rrbracket_{CT} \vdash \llbracket l[i] \rrbracket_{CT})}{\llbracket \Gamma, (o, w, h), \Gamma' \rrbracket_{CT} \vdash o : \langle \llbracket l[1] \rrbracket_{CT} \rangle, \ldots, \langle \llbracket l[n] \rrbracket_{CT} \rangle}\;\textit{Type}}{\llbracket \Gamma, (o, w, h), \Gamma' \vdash o : l \rrbracket_{CT}}\;\llbracket\;\rrbracket_{CT}$$

$\Gamma \vdash \langle \emptyset \rangle \Rightarrow \llbracket \Gamma \vdash \langle \emptyset \rangle \rrbracket_{CT}$ By applying rule *Scheme-null* to the hypothesis we have:

$$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \langle \emptyset \rangle}\;\textit{Scheme-null}$$

By using the inductive hypotheses given by the tree above and the definition of $[\![\ ]\!]_{CT}$, we can prove the thesis via the *Scheme-null* rule.

$$\dfrac{\dfrac{[\![\Gamma]\!]_{CT} \vdash \diamond}{[\![\Gamma]\!]_{CT} \vdash [\![\langle\emptyset\rangle]\!]_{CT}}\ \textit{Scheme-null}}{[\![\Gamma \vdash \langle\emptyset\rangle]\!]_{CT}}\ [\![\ ]\!]_{CT}$$

$\Gamma \vdash \langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle \Rightarrow [\![\Gamma \vdash \langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle]\!]_{CT}$ By applying rule *Scheme* to the hypothesis we have:

$$\dfrac{\forall i(\Gamma \vdash o_i : l_i)}{\Gamma \vdash \langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle}\ \textit{Scheme}$$

By using the inductive hypotheses given by the tree above and the definition of $[\![\ ]\!]_{CT}$, we can prove the thesis via the *Scheme* rule.

$$\dfrac{\dfrac{\forall i([\![\Gamma]\!]_{CT} \vdash [\![o_i : l_i]\!]_{CT})}{[\![\Gamma]\!]_{CT} \vdash [\![\langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle]\!]_{CT}}\ \textit{Scheme}}{[\![\Gamma \vdash \langle o_1, l_1\rangle, \dots, \langle o_n, l_n\rangle]\!]_{CT}}\ [\![\ ]\!]_{CT}$$

$\Gamma \vdash P|P' \Rightarrow [\![\Gamma \vdash P|P']\!]_{CT}$ By applying rule *Par* to the hypothesis we have:

$$\dfrac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P|P'}\ \textit{Par}$$

By using the inductive hypotheses given by the tree above and the definition of $[\![\ ]\!]_{CT}$, we can prove the thesis via the *Par* rule.

$$\dfrac{[\![\Gamma \vdash P]\!]_{CT} \qquad [\![\Gamma \vdash P']\!]_{CT}}{[\![\Gamma \vdash P|P']\!]_{CT}}\ \textit{Par},\ [\![\ ]\!]_{CT}$$

The cases $D \wedge D'$, $J|J'$, $J \rhd P$, $x_o\langle\overrightarrow{y}\rangle$ and $x\langle\overrightarrow{y}\rangle$ follow the same structure of this very last point so we omit their proofs.

$\Gamma \vdash \mathbf{def}\ D\ \mathbf{in}\ P \Rightarrow [\![\Gamma \vdash \mathbf{def}\ D\ \mathbf{in}\ P]\!]_{CT}$ By applying rule *Pdef* to the hypothesis we have:

$$\dfrac{\Gamma, \Gamma' \vdash D :: \Gamma' \qquad \Gamma, \Gamma' \vdash P \qquad dom(\Gamma') = dv(D)}{\Gamma \vdash \mathbf{def}\ D\ \mathbf{in}\ P}\ \textit{Pdef}$$

There is an implicit inductive hypothesis we receive from the above tree: $[\![\Gamma, \Gamma' \vdash \diamond]\!]_{CT}$.

We use this extra hypothesis to be able to apply rule *Pdef* from system J_{CT}.

$$\dfrac{\dfrac{\dfrac{[\![\Gamma, \Gamma']\!]_{CT} \vdash \diamond}{[\![\Gamma]\!]_{CT}, X_1, (x_1, o_1, l_1) \vdash \diamond}\,[\![\;]\!]_{CT}}{[\![\Gamma]\!]_{CT}, X_1 \vdash \diamond} \qquad [\![\Gamma, \Gamma']\!]_{CT} \vdash [\![D]\!]_{CT} :: [\![\Gamma']\!]_{CT} \qquad [\![\Gamma, \Gamma']\!]_{CT} \vdash [\![P]\!]_{CT}}{[\![\Gamma]\!]_{CT} \vdash \mathbf{def}_{X_1}\ [\![D]\!]_{CT}\ \mathbf{in}\ [\![P]\!]_{CT}}\,Pdef$$

We provided the proof for $\Gamma' = (x_1, o_1, l_1)$ since the general case is only syntactically more complex but requires no extra thought to be dealt. The only fact to note is that the derivation of the empty definition $\top$ comes straightforward from the extra inductive hypothesis $[\![\Gamma, \Gamma' \vdash \diamond]\!]_{CT}$ via rule *Top*.

$\square$

Chapter 8

Future work and conclusion

Abstract

In questo capitolo vi è un breve riassunto dei possibili sviluppi futuri e di ciò che contiene questa tesi. Gli sviluppi futuri prevedono l'introduzione di concetti avanzati sia per quanto riguarda i sistemi di tipo che per quanto riguarda gli ownership types. Le conclusioni riassumono il contenuto della tesi. Abbiamo inventato due type systems che formalizzano l'idea di ownership types per il join calculus e ne abbiamo dimostrato la correttezza. Abbiamo poi creato un terzo type system con cui confrontare gli altri due. Infine abbiamo dimostrato che i tre sistemi hanno lo stesso potere espressivo.

8.1 Future work

The type systems presented in this thesis show some great properties but still need to be improved to be useful for programmers. To build a powerful type system, we should bring it from first-order to a higher order one, thus allowing owner polymorphism. This would increase the expressiveness of the system making it much more suitable for real world applications. Among the type system related improvements there is of course the introduction of a mechanism for type inference. This would allow porting classical join calculus programs into ownership-annotated programs without a high syntactical overhead.

Another notion that proved its usefulness in object-oriented programming is *external uniqueness* [CW03, CWM99]. Importing such a concept in a type system would provide the language of a powerful tool for handling uniqueness and ownership toghether. This idea overcomes severe threats in dealing with uniqueness and some drawbacks of ownership types.

After laying all the theoretical foundations, it would be useful to pick one of the type system variants and to apply it to JOCaml, the programming language whose base is the join calculus. This would provide a serious and useful programming language with strong security properties.

8.2 Conclusion

We have devised two approaches to importing ownership types in the join calculus. The first uses a notion of channels-as-owners while the second uses an explicit notion of context to express a contexts-as-owners policy. We provided proofs of the soundness of both type systems in a process calculi setting, which means proving subject reduction and a no runtime error property. The second property replaces the standard progress theorem since the very definition of progress does not suit process calculi.

We proved that our system enforces the owners-as-dominators property inherited from ownership types. Such a property can be translated in a secrecy property which allows us to bind a channel to an area being sure that such a channel will not be accessed from outside that area.

Due to the similarity between the above mentioned properties and the ones enforced by groups for the π-calculus, we created a third type system for the join calculus that imports the notion of those groups.

Finally we have proven that the three systems have the same expressive power. This has been done by showing that the sets of well typed terms for the three systems coincide. The semantics of a program one can write in a system can be obtained in all other systems and vice versa for all systems introduced in this thesis.

Bibliography

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *TACS*, pages 611–638, 1997.

[Aba99] Martín Abadi. Security protocols and specifications. In *FoSSaCS*, pages 1–13, 1999.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[Bae05] Jos C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[BK89] J. A. Bergstra and J. W. Klop. Acpτ: a universal axiom system for process specification. pages 447–463, 1989.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[Boy04] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., MIT, February 2004.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin C. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI*, pages 324–337, 2003.

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, 1996.

[CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[CDD$^+$07] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Universe types for topology and encapsulation. In *FMCO*, pages 72–112, 2007.

[CDE07] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, August 2007.

[CGG05] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. *Inf. Comput.*, 196(2):127–155, 2005.

[Cla01] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[CPN98] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[CW03] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.

[CWM99] Karl Crary, David Walker, and J. Gregory Morrisett. Typed memory management in a calculus of capabilities. In *POPL*, pages 262–275, 1999.

[CWOJ08] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[DZG02] Silvano Dal-Zilio and Andrew D. Gordon. Region analysis and a pi-calculus with groups. *J. Funct. Program.*, 12(3):229–292, 2002.

[FA99] Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP*, pages 91–108, 1999.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1-3):35–75, 1991.

[FFMS02] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, pages 129–158, 2002.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL*, pages 372–385, 1996.

[FG00] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *APPSEM*, pages 268–332, 2000.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *CONCUR*, pages 196–212, 1997.

[FLMR00] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join calculus. In *FSTTCS*, pages 397–408, 2000.

[FM98] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electr. Notes Theor. Comput. Sci.*, 16(3), 1998.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Kob02] Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, pages 439–453, 2002.

[Mil92] Robin Milner. The polyadic pi-calculus (abstract). In *CONCUR*, page 1, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.

[Pie02] Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Pot07] Alex Potanin. *Generic Ownership-A Practical Approach to Ownership and Confinement in OO Programming Languages.* PhD thesis, University of Wellington, 2007.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.

[PS00] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, 2000.

[RPS$^+$04] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Y. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[Sim10] Andrea Simonetto. *Appunti del corso di Tipi e Linguaggi di Programmazione del prof. Simone Martini.* February 2010. Italian notes for the course hold by professor Simone Martini : Types and programming languages.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *J. Funct. Program.*, 2(3):245–271, 1992.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[Wri06] Tobias Wrigstad. *Ownership-Based Alias Management.* PhD thesis, Royal Institute of Technology, Kista, Stockholm, May 2006.