Ressources limitées pour la mobilité : utilisation, réutilisation, garanties.

David Teller

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'école doctorale de MathIF

par Monsieur David Teller

Ressources limitées pour la mobilité : utilisation, réutilisation, garanties

Devant la commission d'examen formée de :

Matthew	Hennessy	Examinateur/Rapporteur
Daniel	Hirschkoff	Examinateur/Directeur de Thèse
Pierre	Lescanne	Examinateur/Directeur de Thèse
Jean-Jacques	Lévy	Examinateur/Rapporteur
Jean-Bernard	Stefani	Examinateur

n^o d'ordre : 290
n^o attribué par la bibliothèque : 04ENSL0 290

Table des matières

Première partie

Préliminaires

Chapitre 1

Introduction

1.1 Présentation

Depuis longtemps déjà, les ordinateurs ne se contentent plus d'exécuter un seul programme sans autre interaction que l'impression d'un résultat à la fin d'un calcul. À l'heure actuelle, la quasi-totalité des systèmes informatiques, qu'ils soient matériels ou logiciels, évoluent et réagissent en fonction de leur environnement. Ainsi, un système d'exploitation est capable de prendre en compte des événements aussi bien logiciels que matériels, qu'il s'agisse d'appels au noyau, d'interruptions réseau ou de problèmes de batterie. De la même manière, un processus réagira aux signaux qui lui sont envoyés et pourra employer des primitives de communication de diverses natures. Enfin, un programme exécuté sur un ordinateur aura probablement été compilé sur une autre machine et pourra n'avoir été déployé que sous forme de mini-application incluse dans une page web.

Ces aspects de parallélisme, de communication et de mobilité posent de nombreux problèmes inédits dans les systèmes séquentiels et isolés. Ainsi, un ensemble de composants liés sera généralement plus sensible à des ruptures de protocole qu'un programme simple. De même, un agent qui recevra du code d'un autre agent devra décider si ce code peut être exécuté sans risquer de compromettre la stabilité du système ou le secret d'informations confidentielles. Au cours des dernières années, on a ainsi vu à plusieurs reprises des virus propagés par Internet (en fait, essentiellement par certains clients mail et certains navigateurs), des attaques destinées à pirater des ordinateurs ou à saturer de requêtes certains serveurs afin de les faire planter (attaques dites de Déni de Service ou Denial of Service).

Le problème de saturation qui rend les dénis de services possibles est lié à une contrainte importante : *tout système n'a qu'une quantité limitée de ressources.* Ceci concerne aussi bien la mémoire des téléphones portables que la place sur le disque dur d'un micro-ordinateur, le nombre de fichiers qui peuvent être ouverts simultanément ou encore la taille de la table de routage entre noeuds. Si les ressources s'épuisent, le programme, l'ordinateur ou le réseau concerné cessent de fonctionner, parfois de manière catastrophique.

Cette thèse est consacrée au problème de la gestion des ressources dans des systèmes parallèles, mobiles et distribués, à l'aide de méthodes fondées sur les

algèbres de processus : comment définir les contraintes de ressources, comment concevoir des systèmes et des protocoles capables de prendre en compte cette contrainte, comment détecter statiquement que les contraintes sont violées ou prouver qu'elles ne le seront jamais.

1.2 Le problème

Selon le dictionnaire Hachette [37], les ressources sont "[les] moyens matériels, [les] réserves dont dispose une collectivité". Dans le cadre de logiciels ou de matériels informatiques, cette définition peut prendre de nombreux sens. Les ressources peuvent désigner aussi bien la mémoire vive que le disque dur, les autorisations d'accès aux fichiers ou aux périphériques, mais aussi bien le temps processeur, le temps réel, le nombre de prises disponibles ou l'argent consacré à réaliser une opération.

Plus généralement, nous considérerons les ressources de la manière suivante : les ressources sont des *réserves* qui peuvent être *allouées* à des *entités*, qui elles-mêmes *déterminent le comportement du programme*, si bien que *celui qui contrôle l'allocation des ressources contrôle le système.*

Si de nombreux calculs, comme le λ-calcul pur, ne permettent pas d'exprimer naturellement une notion de ressources ou des politiques d'utilisation de ces ressources, d'autres calculs, ainsi que de certains langages de programmation, permettent d'écrire des programmes qui seront capables de s'adapter à des contraintes sur l'utilisation de ressources. Nous qualifierons ces formalismes et langages de *conscients des ressources.*

Définition 1 (Gestion des ressources)
Dans chaque système, nous supposons l'existence d'une ou plusieurs réserves de ressources. *Certaines opérations accessibles au système peuvent entraîner ou accompagner l'*allocation *de certaines ressources, c'est-à-dire leur disparition de la réserve. À l'inverse, dans certains systèmes, il peut aussi exister des opérations qui entraînent ou accompagnent la* désallocation *de ressources, c'est-à-dire leur retour à la réserve.*

Définition 2 (Politique de gestion des ressources)
Une politique de gestion des ressources *est un ensemble de critères qui déterminent la taille des réserves et les ressources qui sont allouées ou désallouées par les opérations. Nous parlerons notamment de l'*encombrement *d'une entité pour désigner les ressources dont cette entité a besoin pour être allouée – cet encombrement sera généralement un nombre entier.*

Définition 3 (Systèmes ressource-contrôlés)
Un système sera dit ressource-contrôlé *dans une politique de gestion des ressources donnée, ou* RC*, si aucune allocation n'a lieu depuis une réserve vide.*

Ainsi, dans le cadre d'un programme impératif écrit en `C`, si l'on s'intéresse à la mémoire, le tas fera office de réserve. Une politique de gestion des ressources spécifiera une taille maximale pour le tas et la seule opération d'allocation sera `malloc` tandis que la seule opération de désallocation sera `free`. Dans un cadre plus fonctionnel, on pourra s'intéresser à la pile. Chaque appel de fonction constituera une allocation – pour placer l'environnement local dans la pile – et chaque

retour de fonction une désallocation. On peut aussi examiner des systèmes plus complexes, tels que le tas dans un langage avec garbage-collection – la désallocation étant alors une opération automatique et essentiellement transparente.

En s'écartant des langages de programmation, si l'on considère un système de fichiers, la réserve sera constituée de l'ensemble des fichiers présents sur le disque, la politique de gestion des ressources spécifiera les autorisations d'accès en lecture, écriture, exécution, les opérations d'allocation seront les primitives d'ouverture (i.e. `fopen`) des fichiers et les opérations de désallocation seront les primitives de fermeture (i.e. `fclose`). Enfin, si l'on parle d'argent, la politique déterminera les dépenses autorisées maximales, et le fait de dépenser ou de gagner de l'argent constitueront respectivement une allocation et une désallocation.

Au cours de cet exposé, nous avons considéré un certain nombre de formalismes pour développer ou décrire des systèmes et protocoles séquentiels ou parallèles et distribués. Notre première tâche a consisté à déterminer, pour chaque formalisme, comment exprimer les contraintes de ressources, si nécessaire comment rendre ces calculs conscients des ressources et, en fonction de certains critères spécifiques à chaque système, dans quelles circonstances une ressource peut ou doit être allouée. Pour chacun de ces calculs, nous avons ensuite développé des méthodes de typage pour détecter des systèmes non ressource-contrôlés ou, au contraire, pour garantir le contrôle des ressources dans un système.

1.3 Méthodes

Tout au long de cet exposé, nous allons présenter des méthodes pour exprimer les problèmes d'utilisation de ressources, permettre de les prendre en compte et de garantir le respect de politiques de contrôle.

Pour formuler notre analyse, comme nous considérons des comportements parallèles et éventuellement distribués, nous emploierons essentiellement des algèbres de processus dites *nominales* [35] – l'aspect nominal nous permettant de nommer aisément les entités qui consomment les ressources.

En nous inspirant de [87], nous pouvons classifier les formalismes selon le type de mobilité impliqué. Ainsi, dans les algèbres à *mobilité de sites*, comme les Mobile Ambients, un système est réparti en sites distincts, liés par des relations topologiques d'appartenance, et la mobilité se caractérise par le déplacement visible de ces localités avec leur contenu. À l'inverse, dans des algèbres à *mobilité de noms*, comme le π-calcul, les déplacements sont essentiellement transparents et traduisent plutôt la transmission d'informations et de possibilités de mouvement, sans reconfiguration visible du système. D'autres types d'algèbres de processus existent mais nous ne les avons pas abordés.

Pour chacun de ces calculs, notre analyse se décompose en plusieurs étapes :

1. déterminer une notion de ressources adaptée au calcul et définir formellement cette notion, ainsi que les opérations d'allocation ou de désallocation
2. si nécessaire, étendre le calcul pour le rendre conscient des ressources
3. définir formellement le contrôle des ressources dans le calcul
4. présenter une méthode pour éliminer statiquement les systèmes qui ne sont pas ressource-contrôlés
5. valider la méthode par des preuves et des exemples non-triviaux.

1.4 Plan

Paysage du contrôle des ressources Avant de commencer à présenter nos travaux sur le contrôle des ressources dans les systèmes parallèles et distribués, nous présentons plusieurs conceptions de la notion de ressource telles qu'elles apparaissent dans la littérature. Ainsi, nous abordons des méthodes sur le contrôle de la mémoire vive en programmation impérative, fonctionnelle ou encore parallèle synchrone, ainsi que sur des protocoles d'utilisation de primitives dans des langages impératifs ou fonctionnels.

Calculs à mobilité de sites Nous nous intéressons ensuite au problème du contrôle des ressources dans les calculs à mobilité de sites. Après avoir introduit le formalisme des Mobile Ambients comme représentant de ces algèbres de processus, nous présentons BoCa [7] puis notre propre calcul des Controlled Ambients [80], deux calculs dérivés des Mobile Ambients et conçus pour le contrôle des ressources. Nous introduisons ensuite les calculs des Seals [19] et des Kells [77], deux autres formalismes de la même famille, qui mettent en jeu respectivement une sémantique différente des mouvements et des notions d'ordre supérieur.

Calculs à mobilité de noms De la même manière que nous avons présenté les calculs à mobilité de sites à l'aide des Mobile Ambients, nous introduisons les calculs à mobilité de noms par le π-calcul. Nous présentons ensuite un système de types linéaires [53] qui permet de contrôler certains aspects des ressources dans le π-calcul. Après cela, nous détaillons nos travaux sur le contrôle des ressources dans ces formalismes, avec le Controlled π-calculus [79], une esquisse d'extension vers la gestion des ressources par régions et une ébauche de version distribuée, le Controlled Distributed π-calculus.

Pour d'autres notions de ressources Tout comme il existe plusieurs notions de ressources dans les calculs séquentiels, il existe plusieurs notions de ressources dans les calculs à mobilité de noms ou de sites. Nous présentons donc une méthode simple pour étendre nos systèmes de types, de manière à gérer certaines de ces notions. Nous abordons en particulier le contrôle des communications dans le π-calcul et ses variantes, ainsi que le contrôle de la mobilité dans les Mobile Ambients. Nous introduisons de plus des travaux en cours vers d'autres extensions de nos systèmes de types, qui permettent d'observer statiquement des comportements plus complexes des processus.

Conclusions En dernier lieu, nous revenons sur les problèmes que nous avons abordés, sur les techniques que nous avons employées pour les résoudre et sur la réutilisation de nos travaux, ainsi que sur les directions que nous souhaiterions emprunter ou voir emprunter à l'avenir.

Preuves Les annexes contiennent les preuves que nous avons jugées pertinentes, en particulier pour les lemmes et théorèmes liés à la Subject Reduction et au contrôle des ressources.

Chapitre 2

Paysage du contrôle des ressources

Si le problème que nous abordons dans ce document est celui de la gestion des ressources en présence de mobilité et de parallélisme, de nombreux travaux s'intéressent aux ressources dans le cadre de calculs et de programmes séquentiels, c'est-à-dire sans mobilité ni parallélisme explicités. D'autres traitent de ressources avec une définition du parallélisme différente de la nôtre. Dans ce chapitre, nous présentons certaines des méthodes qui avaient déjà proposées pour le contrôle des ressources lorsque nous avons entrepris nos recherches ou qui ont été élaborées depuis, en parallèle à nos efforts>

Nombre de ces travaux, dans un cas comme dans l'autre, cherchent à contrôler les ressources de manière, notamment, à permettre de garantir la sûreté de code mobile. Parmi ces projets, certains visent à contrôler l'espace – et plus précisément la gestion de la mémoire vive – tandis que d'autres visent, par exemple, à borner le nombre d'utilisations d'une valeur ou à assurer que tous les objets utilisés ont été correctement initialisés ou/et seront correctement relâchés.

Plusieurs systèmes de types [25, 47] ont aussi été développés pour permettre un contrôle généraliste des ressources dans des langages de programmation – généraliste au sens où les règles d'utilisation des ressources sont définies par le programmeur. L'objectif n'est alors plus uniquement de vérifier qu'un programme n'utilise pas de ressources dont il ne dispose pas mais bel et bien de vérifier que l'utilisation des ressources par un programme respecte un protocole préétabli.

Si ces définitions de ressources ne coïncident pas forcément avec les nôtres – en fait, d'après nos définitions, nous nommerions ces ressources "entités" – et si certaines formes de contrôle dépassent l'objectif de notre travail, nous avons jugé intéressant de présenter quelques méthodes employées.

Pour la suite de ce chapitre, et pour éviter toute confusion, nous emploierons les mêmes notations que dans le reste de l'exposé. Nous conserverons le terme de ressources pour l'espace et nous nommerons donc "entités" les autres objets qui, dans certains de ces travaux, sont appelées ressources.

Contrôle de l'espace dans un cadre impératif Avant de nous intéresser au contrôle d'entités abstraites dans des calculs et des langages de programmation

de haut niveau, nous commencerons par approcher le problème du point de vue de la programmation système. Ainsi, nous discuterons du contrôle de l'espace dans les langages `C` et `C++`. En effet, les primitives d'allocation et de désallocation disponibles en `C/C++` permettent déjà un certain degré de surveillance et de gestion des ressources.

De plus, nous présenterons un bref aperçu d'une discipline typable de contrôle des ressources.

Contrôle de l'espace dans un cadre fonctionnel Dans la majeure partie des langages fonctionnels, à l'image du λ-calcul, l'allocation et la désallocation de l'espace sont des opérations essentiellement transparentes, grâce à la garbage-collection automatique. Dans le langage Camelot [60], fortement typé et de la famille ML, à l'inverse, il est aussi possible de spécifier explicitement à une valeur de prendre la place d'une autre valeur spécifique. Les ressources sont alors réallouées manuellement d'un symbole à un autre symbole.

Nous présenterons quelques-uns des mécanismes de gestion de la mémoire dans Camelot ainsi que le système de types $LF_{\diamond}$ et les garanties qu'il permet d'obtenir sur l'utilisation de l'espace.

Contrôle de l'espace dans un langage parallèle coopératif Les modèles d'exécutions synchrones coopératives [11] permettent de représenter des systèmes qui s'exécutent sous certaines hypothèses de synchronie. Grâce à ces contraintes strictes, il est possible de garantir des propriétés fortes, par exemple sur l'utilisation des références [11] ou sur celle de l'espace [3].

Nous présenterons brièvement un modèle d'exécution synchrone et quelques-unes des techniques employées pour fournir des preuves sur l'utilisation de la mémoire par un programme synchrone.

Contrôle de l'espace par un bac à sable Certains langages de programmation, notamment Java, s'exécutent dans un environnement qualifié de bac à sable : toutes les opérations considérées comme potentiellement dangereuses sont alors surveillées durant l'exécution. Ainsi, il est possible d'interdire à une application de dépasser certaines bornes d'occupation de la mémoire vive ou encore de vérifier certains aspects de la gestion des verrous par un programme.

Nous rappellerons brièvement les mécanismes derrière un bac à sable.

Contrôle des entités par des clés Vault [25] est un langage de programmation impératif proche de `C`. L'idée derrière le contrôle des entités dans Vault est que certaines opérations ne peuvent être accomplies que sur des entités dans l'état approprié. Un système de types permet alors de spécifier dans quel état est une variable et quel est l'effet d'une fonction sur l'état d'une variable.

Nous présenterons brièvement le langage Vault, ce mécanisme de contrôle et son système de types.

Analyse d'utilisation des entités Un autre travail [47], que nous désignerons Resource Usage Analysis (ou RUA) d'après le titre de l'article qui le présente, emploie une approche duale, afin de contrôler les entités dans le λ-calcul. L'idée de RUA est que chaque variable doit (ou peut) passer par un certain nombre d'opérations avant qu'on sorte de son domaine de définition.

Nous présenterons, là aussi, le langage sur lequel travaille RUA, ainsi que le système de types qui permet de garantir le respect des protocoles.

2.1 Contrôle de l'espace dans un langage impératif

Les langages de programmation impératifs de bas niveau comme le `C` sont généralement considérés comme peu appropriés dès qu'il s'agit de garantir des propriétés de programmes. Ce sont cependant ces langages qui sont, le plus souvent, employés pour développer les applications très sensibles aux ressources, telles que les applications embarquées ou les serveurs.

Si nous considérons la mémoire du tas comme ressource, les primitives d'allocation et de désallocation qu'offre `C` sont respectivement les fonctions `malloc()` et `free()`. De même, le `C++` dispose des opérateurs, plus évolués, `new`/`new[]` et `delete`/`delete[]`. Même s'il existe des variantes de ces fonctions et opérateurs, telles que `calloc()` ou `realloc()`, nous passerons ces fonctions sous silence et nous nous contenterons de considérer qu'elles ne posent pas de problèmes supplémentaires.

Nous ne présenterons pas ces primitives, que nous supposons connues.

2.1.1 Programmes conscients des ressources

Avant de présenter une discipline de programmation particulière, il peut être intéressant de remarquer que les langages `C` et `C++` permettent d'écrire des programmes partiellement conscients des ressources.

En effet, aussi bien `malloc()` que `new` et `new[]` ont un comportement précisément spécifié au cas où la mémoire vive viendrait à manquer et à rendre l'allocation impossible – comportement qui peut être utilisé pour libérer de la mémoire avant de réessayer. De plus, ces trois fonctions/opérateurs et leurs contreparties de désallocation peuvent être aisément remplacées par d'autres fonctions et opérateurs capables de comptabiliser les allocations et désallocations de ressources. Un programme peut alors vérifier manuellement l'état des réserves et choisir quelles routines exécuter pour éviter de dépasser des limites imposées.

Enfin, sous Un*x, des appels système tels que `getrusage` permettent de déterminer, pour chaque processus, la quantité de mémoire vive utilisée pour les données (ou `ru_idrss`), pour la pile (`ru_isrss`) ou partagée (`ru_ixrss`), ainsi que l'état d'utilisation d'autres ressources. Notons que l'utilisation de `getrusage` n'est, en fait, pas liée à l'aspect impératif, et que rien n'interdirait, en théorie, d'employer un appel système similaire dans un langage fonctionnel, par exemple.

Ainsi, la figure 2.1 présente un gestionnaire de mémoire simpliste en `C`. La fonction `my_malloc()` essaye d'allouer la mémoire. Si l'allocation est impossible, la mémoire de réserve en cas d'urgence `reserve` est utilisée, le temps de trouver une réponse au problème, à l'aide de la fonction `do_something_intelligent()`.

Cependant, pour les raisons habituelles qui compliquent la vérification de programmes `C` et `C++`, dans le cas général, il est impossible de prouver des bornes sur l'utilisation de la mémoire vive par un programme.

```
const int RESERVE_SIZE = ...
char* reserve;

int init_memory()
{
  reserve = malloc(sizeof(char)*RESERVE_SIZE);
}

void* my_malloc(const int size) {
  void* result = malloc(size);
  if(result==NULL) {
    free(reserve);
    do_something_intelligent();
    return malloc(size);
  } else {
    return result;
  }
}

int main(const int argc, const char** argv) {
  init_memory();
  do_something();
}
```

FIG. 2.1 – Gestionnaire de mémoire minimal en C.

2.1.2 Réutilisation immédiate de ressources linéaires

Une autre approche a été proposée [44] pour permettre de concevoir des programmes C qui n'emploient qu'une mémoire bornée.

L'idée générale consiste à gérer manuellement une liste de ressources libérables et à les réutiliser immédiatement pour y placer d'autres données, plutôt que de les désallouer et de les réallouer. Un exemple simple de cette méthode serait une implantation "en place" d'un algorithme de tri dans un tableau.

Pour vérifier et certifier cette gestion de la mémoire, l'auteur utilise un système de types linéaires – linéaires au sens où chaque paramètre ne peut être utilisé qu'une seule fois.

En fait, ce système de types n'est pas appliqué au code C lui-même mais à un code écrit dans un langage fonctionnel et utilisé pour générer le code C. Nous n'entrerons pas dans les détails car nous présentons plus avant ces méthodes dans la prochaine section et dans le cadre de la programmation fonctionnelle.

2.1.3 Discussion

L'approche qui consiste à comptabiliser manuellement l'utilisation de la mémoire ou à compter sur les mécanismes d'erreur associés à l'allocation et à la désallocation est possible et effectivement utilisée dans la majeure partie des

programmes qui ont besoin de contrôler les ressources, même pour gérer des ressources autres que l'espace. Cette méthode est malheureusement imprécise et arbitraire et son fonctionnement est impossible à garantir sauf pour les cas les plus simples.

L'approche qui tire parti de la liste de ressources est intéressante car elle vise à généraliser une pratique commune en `C`. Cependant, elle restreint énormément les possibilités de `C`, notamment en forçant trop souvent à dupliquer des données en mémoire pour pouvoir en effacer une copie. Enfin, telle quelle, cette approche semble totalement inadaptée dans un cadre parallèle, en raison de l'impossibilité de désallouer une ressource dans un processus (ou processus léger) pour la réutiliser dans un autre.

2.2 Contrôle de l'espace dans un langage fonctionnel

Camelot [60] est un langage fonctionnel de la famille ML conçu pour permettre d'écrire des applications sûres et dont les propriétés de sécurité sont certifiées sous la forme d'un code porteur de preuves. Un programme écrit en Camelot est compilé vers le langage intermédiaire Grail, dont sont extraites des obligations de preuves et les informations nécessaires pour garantir le respect de ces obligations. Enfin, le code Grail est compilé vers le langage machine Java (JVML).

L'un des aspects intéressants de Camelot est la possibilité d'écrire des programmes conscients de l'utilisation des ressources et d'extraire [48] de ces programmes des bornes sur l'utilisation de ces ressources. Précisons que, si certains procédés employés pour permettre ce contrôle des ressources sont proches de méthodes que nous avons nous-mêmes développées et que nous présenterons plus loin, notamment à la section 6, les travaux sur les ressources dans Camelot ont été menés à bien essentiellement en même temps que nos recherches.

2.2.1 Programmes conscients des ressources

Pour illustrer ce processus, considérons l'implantation d'un algorithme classique d'inversion de listes à l'aide d'un accumulateur.

```
let rev l acc =
  match l
    with Nil -> acc
       | Cons(h,t)  -> rev t (Cons(h,acc))
```

FIG. 2.2 – Inversion d'une liste en Camelot.

La figure 2.2 présente une implantation Camelot possible de cet algorithme. Dans cette version, lors d'un appel récursif de la fonction, la valeur `Cons(h,t)` n'est jamais réutilisée par la fonction, tandis que la valeur `Cons(h,acc)` est créée à une nouvelle adresse qui lui est allouée. Sans optimisations, pour inverser une

liste de longueur n et en supposant que `Cons(_,_)` occupe une ressource, il est donc nécessaire d'allouer n ressources.

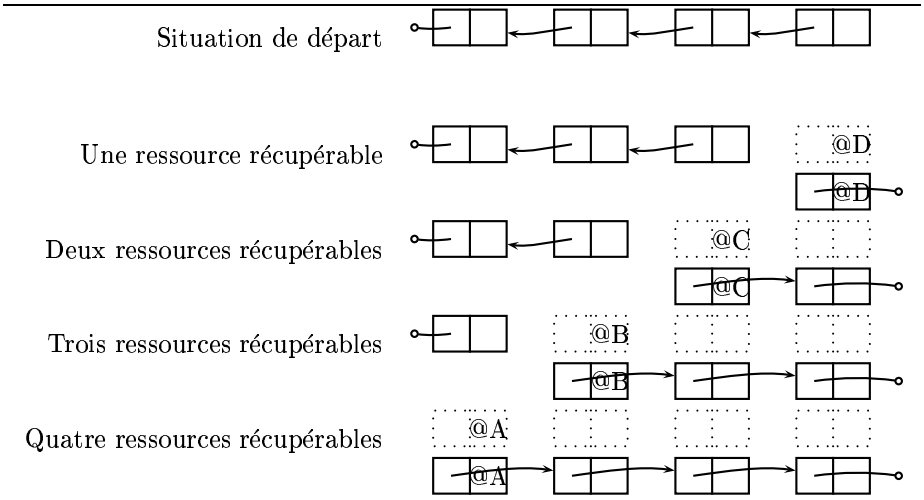


FIG. 2.3 – Inversion en place d'une liste en Camelot : utilisation des ressources.

Considérons le cas où la liste originale – que nous nommerons *source* – n'est jamais réutilisée. Comme l'illustre la figure 2.3, l'inversion aura alors nécessité n ressources et les n ressources initialement occupées par *source* finiront par être désallouées par le ramasse-miettes. Or, dans ce cas, l'opération aurait pu être exécutée sans nécessiter d'allocations supplémentaires, en réallouant au fur et à mesure les ressources occupées par *source*.

```
let rev l acc =
  match l
    with Nil@_ -> acc
       | Cons(h,t)@d  -> rev t (Cons(h,acc)@d)
```

FIG. 2.4 – Inversion en place d'une liste en Camelot.

Pour permettre cette réallocation manuelle des ressources, Camelot emploie plusieurs annotations, dont le fonctionnement est illustré sur la figure 2.4. Lorsqu'un filtrage par motifs rencontre un constructeur annoté par `@_`, Camelot déduit que la valeur reconnue par ce motif ne sera plus utilisée et que les ressources qui lui sont allouées peuvent donc être libérées. De même, lorsqu'un motif de la forme *Constructeur*`@d` reconnaît une valeur, Camelot assigne à d l'adresse à laquelle est alloué la valeur. Par la suite, si Camelot rencontre une valeur annotée par ce même `@d` cette valeur sera stockée à l'emplacement en mémoire où apparaissait le constructeur reconnu par le motif.

Ainsi, lors d'une exécution de `rev Cons(1,Nil) Nil`, le motif `Cons(h,t)@d` reconnaît `Cons(1,Nil)` instancié avec $h \leftarrow 1$ et $t \leftarrow Nil$ et d a pour va-

leur l'adresse à laquelle était alloué `Cons(h,t)@d`. Plus loin, lorsque la valeur `Cons(h,acc)@d` est construite, elle remplace donc `Cons(1,Nil)` en mémoire.

L'exécution de la fonction se poursuit par un appel récursif

$$\texttt{revNilCons}(1, \texttt{Nil})\,,$$

au cours duquel le motif `Nil@_` reconnaît *Nil* et désalloue les ressources qui lui étaient allouées.

En d'autres termes, l'annotation `@_` représente une opération de désallocation, tandis que le couple composé de l'annotation `@d` au côté du motif et de l'annotation `@d` au côté de la valeur représente une opération de désallocation et réallocation immédiate d'une ressource.

Ces opérations sont plus complexes et bien plus précises que les mécanismes d'allocation et de désallocation présents aussi bien en Caml qu'en `C`, même s'il est possible de tricher dans les deux langages pour en obtenir des approximations, au prix du système de types. Comme souvent, cette richesse est aussi un danger car, à l'inverse des garanties fournies par les ramasse-miettes, rien dans la syntaxe ne certifie que `Cons(1,Nil)` puisse effectivement être nettoyé sans risques pour l'exécution. Plusieurs systèmes de types [45, 54] peuvent alors venir compléter Camelot pour assurer des propriétés de sûreté de l'allocation et de la désallocation.

Notons que, si nous avons présenté un exemple dans lequel les données fournies à la fonction étaient désallouées, un cas probablement plus fréquent de désallocation concernera plutôt des données temporaires calculées au cours de l'exécution d'un algorithme et qui deviennent inutiles après certaines étapes.

2.2.2 Garanties sur l'utilisation des ressources

En plus de certaines opérations d'allocation et de désallocation manuelles qui permettent de concevoir des programmes conscients de l'utilisation des ressources, Camelot dispose d'un système de types [48] capable d'inférer des bornes sur la quantité de mémoire vive utilisée par une fonction – quantité qui peut être négative si la fonction libère en fait de la mémoire.

Le type de chaque fonction est ainsi complété par des informations sur sa consommation en ressources. Ainsi, si `ilist` est la structure de données d'une liste d'entiers, considérons la signature suivante :

```
val f : 2, ilist[Nil(3)|Cons(int,#,4.5)]
        -> ilist[Nil(6)|Cons(int,#,7)]
        -> ilist[Nil(0)|Cons(int,#,8)], 0;
```

Le fragment `ilist[Nil(3)|Cons(int,#,4.5)]` représente "une structure de données de type `type ilist = Nil | Cons of int*ilist` où chaque occurrence de `Nil` occupe 3 ressources et chaque occurrence de `Cons` occupe 4.5 ressources".

On peut alors déduire de cette signature que

- f est une fonction de type ML `ilist -> ilist -> ilist`
- si a et b sont des listes de longueur respective l_a et l_b, calculer `f a b` nécessite l'allocation de $2 + (3 + 4.5 \cdot l_a) + (6 + 7 \cdot l_b)$ ressources mais laisse au moins $0 + (0 + 8 \cdot l_c)$ ressources disponibles à la fin de son exécution, où l_c est la longueur du résultat.

$$\textsc{ConstBool}\ \frac{c \text{ est une constante booléenne}}{\Gamma, n \vdash_\Sigma c : \mathbf{B}, n'}\ n \geq n'$$

$$\textsc{Fun}\ \frac{\Sigma(f) = (A_1, \ldots, A_p, k) \longrightarrow (C, k')}{\Gamma, x_1 : A_1, \ldots, x_p : A_p, n \vdash_\Sigma f(x_1, \ldots, x_p) : C, n'}\ n \geq k,\ n - k + k' \geq n'$$

FIG. 2.5 – Un fragment de $LF_\diamond$.

Sans entrer dans les détails sur la manière d'obtenir le type de f, précisons quelques points sur $LF_\diamond$, le cœur de ce système de types et sur `lfd_infer`, son implantation.

$LF_\diamond$ est un système de types du premier ordre dans lequel un jugement $\Gamma, n \vdash_\Sigma e : A, n'$ se lit "sous les hypothèses Γ, à condition que n ressources soient disponibles et si le type des fonctions employées est spécifié par la signature Σ, alors l'expression e a le type A et laisse au moins n' ressources libres après avoir terminé son exécution."

La figure 2.5 présente deux des règles d'inférence de $LF_\diamond$. La règle CONST-BOOL spécifie que les expressions *true* et $false$ sont de type $\mathbf{B}$ et ne nécessitent pas de ressources. Nous retrouverons dans nos systèmes de types des règles similaires sous les noms T-NAME, T-AMBNAME, T-PROCNAME...

La règle FUN spécifie que, si la signature de f donne un type

$$(A_1, \ldots, A_p, k) \longrightarrow (C, k')$$

à f, alors l'expression $f(x_1, \ldots, x_p)$ est de type C et laisse au plus $n - k + k'$ ressources disponibles. Notons que cette règle ne vérifie pas que la fonction est appliquée à des paramètres de bon type : cette propriété est supposée avoir déjà été vérifiée par un système de types fonctionnels classique. Dans cette signature, k et k' représentent respectivement les ressources consommées par f au cours de son exécution et les ressources libérées par f à la fin de son exécution. Ce comportement est proche de ce que nous retrouverons, dans nos systèmes, dans le typage des communications par T-RCV ou, plus encore, par T-TRIG.

2.2.3 Limitations

Si $LF_\diamond$ a l'immense avantage de proposer une inférence essentiellement automatique et de fournir des résultats suffisamment précis pour de nombreuses fonctions, quelques limitations restent gênantes. En premier lieu, on peut regretter l'impossibilité de typer des fonctions d'ordre supérieur et le manque de polymorphisme.

De plus, l'approche Camelot/$LF_\diamond$ ne s'applique qu'aux fonctions qui désallouent explicitement des ressources et ne semble pas appropriée pour gérer le cas de ressources récupérées par garbage-collection. Cette limitation est surprenante car les langages ML sont dotés d'un ramasse-miette. Certaines variantes, notamment OCaml, disposent d'une fonction de finalisation [9, 57], qui permettrait d'obtenir des résultats similaires – mais peut-être plus complexes, car ils dépendraient de propriétés de garbage-collection.

Enfin, $LF_\diamond$ ne semble pas prévu non plus pour gérer les programmes parallèles. Nous verrons cependant plus loin dans cet exposé, notamment dans la section 6.3, des idées similaires appliquées dans un calcul parallèle, à l'aide de mécanismes de finalisation.

2.3 Contrôle de l'espace dans un langage synchrone

Les modèles d'exécution synchrones coopératives [11] permettent de représenter des systèmes concurrents. À la différence de tous les autres calculs que nous utiliserons dans cet exposé, ces modèles s'inspirent d'un parallélisme coopératif. Ainsi, comme sous Windows 3.1, Mac OS 8 ou Risc OS, un processus n'est jamais interrompu tant qu'il ne passe pas explicitement la main à l'ordonnanceur à l'aide d'une opération spécifique. De plus, une notion d'instants, proche de celle que l'on peut trouver dans des langages synchrones tels qu'ESTEREL [8], régule les interactions entre les threads.

Cette approche du parallélisme conserve les avantages du multitâche coopératif comme ses défauts. Ainsi, s'il est aisé d'écrire un processus qui ne sera jamais interrompu alors qu'il est dans un état instable, chaque processus doit être conçu de manière à relâcher la main régulièrement et doit s'assurer que ce passage n'entraînera pas un deadlock. Un tel processus doit donc généralement procéder à une attente active. Un intérêt de ces modèles est qu'ils sont plus proches de la programmation séquentielle que, par exemple, les algèbres de processus avec lesquelles nous travaillerons dans le reste de cet exposé et qu'ils permettent ainsi d'étendre de manière relativement naturelle des approches fonctionnelles.

Dans cette section, nous présentons un travail [3] qui étend certains résultats de réécriture et de programmation séquentielle [10, 2] sur le contrôle des ressources à un modèle d'exécutions synchrones coopératives. Précisons que ce travail, lui aussi, a été mené à bien essentiellement en même temps que nos propres recherches.

2.3.1 Un aperçu du langage

Sans entrer dans le détail de la syntaxe ou de la sémantique, un système est constitué d'un ou plusieurs processus. Les processus ressemblent à des termes d'un λ-calcul étendu notamment à l'aide de références (ou *registres*) et d'un filtrage par motifs sur la valeur d'un registre. Une instruction `yield`.b permet de passer la main puis de continuer comme b après avoir récupéré la main. Une instruction `next`.$f(\overrightarrow{e})$ permet de synchroniser le processus avec tous les autres processus – c'est-à-dire attendre la fin de l'*instant* courant – avant de continuer comme $f(\overrightarrow{e})$.

Ainsi, l'écriture et la lecture sur un canal c peuvent s'écrire, à l'aide d'une structure de données $ch(t) \triangleq$ `empty` | `full` $of\ t$:

$$
\begin{array}{lcll}
\texttt{send}(c,e).p & \triangleq & \texttt{match}\ c\ \texttt{with} & \\
 & & \quad \texttt{empty} & \Rightarrow c := \texttt{full}(e).p \\
 & & \quad [z] & \Rightarrow \texttt{fail}() \\
\texttt{receive}(c,x).q & \triangleq & \texttt{match}\ c\ \texttt{with} & \\
 & & \quad \texttt{full}(x) & \Rightarrow c := \texttt{empty}.q \\
 & & \quad [z] & \Rightarrow \texttt{fail}()\ .
\end{array}
$$

Le processus émetteur attend à l'aide de `match` que le registre c contienne une valeur `empty` puis, lorsque tel est le cas, remplace cette valeur par $\texttt{full}(e)$ et continue comme b. Si, lors de la synchronisation entre tous les processus, c ne contient pas `empty`, la fonction $\texttt{fail}()$ est invoquée. Le `match` initial ne peut être invoqué que si le processus émetteur prend la main et l'attente elle-même passe la main aux autres processus.

Le processus récepteur attend à l'aide de `match` que le registre c contienne une valeur $\texttt{full}(x)$ puis, lorsque tel est le cas, remplace cette valeur par `empty` et continue comme q. Si, lors de la synchronisation entre tous les processus, c ne contient pas $\texttt{full}(x)$, la fonction $\texttt{fail}()$ est invoquée. Le `match` initial ne peut être invoqué que si le processus récepteur prend la main et l'attente elle-même passe la main aux autres processus.

Cette forme de synchronisation systématique entre tous les processus induit une notion d'*instants* du système. Ainsi, lorsque *tous* les processus du système sont en train d'attendre à l'aide de l'instruction `next` ou d'un filtrage par motifs, l'état global du système change pour passer à l'instant suivant. Alors, dans notre exemple de communication, si l'émission n'a toujours pas pu avoir lieu, elle échoue en invoquant $\texttt{fail}()$, de même que la réception, si elle n'a toujours pas pu avoir lieu.

Notons que le modèle employé ne permet pas la création dynamique de nouveaux processus.

2.3.2 Bornes sur l'utilisation des ressources

L'analyse qui permet de borner l'utilisation de l'espace s'inspire de techniques employées pour analyser la complexité en temps et en espace dans des systèmes de réécriture [10].

Lecture unique

Une première étape de vérification permet de garantir une condition de lecture unique : entre deux synchronisations globales, chaque processus n'est autorisé à lire les registres qu'une seule fois au plus.

Ainsi, si $nat \triangleq z \mid s$ est le type des entiers et si

$$
dble \triangleq \left\{ \begin{array}{ll} dble(z) & = z \\ dble(s(n)) & = s(s(dble(n))) \end{array} \right.
$$

définit le dédoublement unaire, la propriété de lecture unique permet d'écarter les processus dont la consommation en mémoire est exponentielle, tels que

$$
\left\{ \begin{array}{lcl} exp(z) & = & \texttt{stop} \\ exp(s(n)) & = & \texttt{match}\ r\ \texttt{with}\ m \Rightarrow r := dble(m).exp(n)\ . \end{array} \right.
$$

Pour vérifier cette condition, un algorithme calcule une approximation de l'ensemble des registres lus par un processus donné pendant un instant donné et s'arrête lorsqu'un doublon potentiel est détecté.

Une conséquence de cette propriété de lecture unique est que le comportement d'un processus pendant un instant peut être décrit sous la forme d'une fonction des paramètres du processus et des valeurs des registres lus pendant cet instant.

Assignations et quasi-interprétations

La deuxième étape de l'analyse consiste à définir l'utilisation de ressources par les symboles.

Une *assignation* (que nous considérerions comme un élément d'une *politique de contrôle des ressources* dans le reste de cet exposé) définit, pour chaque symbole, une forme d'encombrement à valeur dans $\mathbf{R}^+$. Ainsi, l'encombrement d'une constante est nul, celui d'un constructeur c d'arité n est une fonction q_c telle que $q_c(x_1, \ldots, x_n) = d + \Sigma_{i \in 1..n} x_i$ avec $d \geq 1$ et celui d'une fonction f d'arité n est une fonction q_f monotone et telle que, pour tout i, $q_f(x_1, \ldots, x_n) \geq x_i$.

La notion de quasi-interprétations étend les assignations aux continuations, qui peuvent être communiquées et qui posent donc des problèmes supplémentaires, que nous ne détaillerons pas.

Bornes

La dernière étape de l'analyse consiste à déterminer le nombre d'étapes de réduction qui peuvent survenir pendant l'exécution d'un instant. Pour ce faire, plusieurs méthodes peuvent être employées.

La technique citée par les auteurs emploie un ordonnancement lexicographique du chemin (ou LPO [10]). Ainsi, s'il existe une relation d'ordre sur les termes d'un système qui soit un ordre lexicographique sur les chemins, on peut être certain que le système termine – en fait, dans le contexte des modèles d'exécutions synchrones, cela signifie que chaque instant termine.

Dès qu'un système termine par LPO et dispose d'une quasi-interprétation polynomiale, l'espace utilisé par ce système au cours d'un instant est borné par un polynôme sur les paramètres du système. Plus précisément, dans un système à m registres et n processus, si la fonction h borne toutes les quasi-interprétations du système – c'est-à-dire si $\forall f, \forall x \geq 0, h(x) \geq q_f(x, \ldots, x)$ – et si la taille des paramètres du système pour cet instant est inférieure ou égale à c, alors, durant l'instant, l'utilisation de la mémoire est bornée par $h^{n \cdot m+1}(c)$.

2.3.3 Discussion

Les méthodes exposées dans cette section permettent de fournir, de manière automatisable, des garanties polynomiales sur l'espace nécessaire pour l'exécution d'un ensemble de processus parallèles durant un instant.

Si les techniques en question sont intéressantes, le modèle employé est très différent des calculs que nous avons employés tout au long de cet exposé. Ainsi, la gestion coopérative de l'ordonnancement et la nécessité de synchronisations entre tous les processus font, à notre avis, de ce formalisme un outil moins adapté à la gestion du code mobile. Notons qu'il existe un projet [11] qui vise

à adapter les modèles d'exécution synchrones coopératives au cadre du code mobile et distribué. Cependant, la notion d'instants définis globalement reste une hypothèse forte, que nous avons préférée ne pas adopter.

Contrairement à nos méthodes, que nous présenterons dans la suite de cet exposé, et qui calculent des bornes fixes, cette analyse prouve l'existence d'une borne polynomiale sur la consommation en mémoire du système pendant un instant. Cette borne polynomiale peut se révéler importante pour de nombreuses applications dont la consommation de mémoire dépend des paramètres qui leur sont fournis. Cependant, la borne effectivement obtenue à l'aide des techniques décrites semble élevée – peut-être trop élevée, pour servir de garantie de sécurité pour du code mobile.

Enfin, une autre limitation importante de ce travail, l'impossibilité de créer de nouveaux processus durant l'exécution, nous fait préférer les algèbres de processus comme outils pour l'examen de l'utilisation des ressources par un système. Notons cependant que cette restriction est bien moins importante dans un langage qui contient déjà un λ-calcul non-déterministe que ne le serait la suppression de cet aspect du π-calcul ou des Mobile Ambients.

2.4 Contrôle de l'espace à l'aide d'un bac à sable

Nous nous sommes jusqu'à présent intéressés au contrôle des ressources dans le langage. Or, il est parfaitement possible de reporter ce contrôle à un niveau plus proche du système. C'est exactement le mécanisme utilisé, aussi bien dans la machine virtuelle Java (JVM) que sous Un*x ou encore sur des JavaCard.

Avec un mécanisme de bac à sable, l'utilisation des ressources est observée par un contrôleur de niveau supérieur. Ce contrôleur, généralement assimilé au système d'exploitation, peut à tout instant décider de refuser un opération, que ce soit en lançant une exception ou en arrêtant totalement l'application, lorsque la politique de contrôle est sur le point d'être violée.

2.4.1 Politiques de contrôle des ressources

Nous n'entrerons pas dans les détails de la conception ou de la mise en œuvre d'un bac à sable. Nous nous contenterons de remarquer que les mécanismes sont purement dynamiques et nécessitent de vérifier tous les appels dangereux – ici l'allocation et la désallocation de mémoire, ainsi que les accès à la mémoire.

Les propriétés vérifiées ainsi que les manières de fixer ces propriétés peuvent être très différentes d'un bac à sable à un autre.

Ainsi, sous Un*x ou dans une JVM, lorsqu'un programme nécessite plus de mémoire, il peut demander un bloc supplémentaire au système, qui décidera ou non de l'allouer. Sous de nombreuses variantes d'Un*x, la politique en vigueur peut être modifiée dynamiquement par l'appel système `setrlimit()`, qui permet à l'utilisateur, à l'application et au super-utilisateur de limiter, entre autres, la taille de la pile d'appels, la taille du segment de données et la taille maximale des fichiers créés. Dans une JVM, à l'heure actuelle, la politique en vigueur est fixée avant le lancement de la machine virtuelle par une option en ligne de commande.

Enfin, sous les anciens systèmes d'exploitation Mac OS, une propriété des exécutables, modifiable par l'utilisateur, détermine la taille de la *partition* de la mémoire vive allouée à une application et qui contient aussi bien la pile que

le tas [4] – cette limite peut en fait être contournée à l'aide de techniques non-vérifiées par le système.

2.4.2 Discussion

L'intérêt principal du bac à sable est la sécurité qu'il apporte, de manière essentiellement transparente. Dans un bac à sable suffisamment hermétique, quelle que soit la manière dont un programmeur a écrit son programme, le système d'exploitation l'arrêtera avec un message `Out of memory` ou une exception `OutOfMemoryException` si celui-ci viole la politique d'affectation des ressources en vigueur.

Le prix de cette sécurité est le nombre de vérifications nécessaires pendant l'exécution d'un programme, qui entraîne un surcoût, en termes de mémoire et surtout en termes de temps. Le deuxième inconvénient est qu'un bac à sable est généralement un programme complexe à écrire et à vérifier surtout, comme c'est souvent le cas en Java, en présence d'optimisations dynamiques telles que les compilateurs à la volée. Ces deux défauts justifient tous les travaux sur les garanties statiques, et ce non seulement dans le domaine du contrôle des ressources mais aussi pour tous les aspects de sécurité.

Mentionnons brièvement qu'il existe des mécanismes formels qui rappellent le principe de bacs à sable. Ainsi, le système des Guardians [26] introduit dans les Mobile Ambients un système de gardiens – des processus écrits dans un langage spécifique et qui permettent de contrôler dynamiquement chaque opération des Mobile Ambients. De même, dans le M-Calcul [74], chaque localité est dotée d'une membrane, dans laquelle des processus (programmables et reprogrammables) contrôlent et routent les déplacements.

2.5 Contrôle de protocoles avec Vault

Le langage Vault [25] est un langage impératif proche du `C`. Ce langage est conçu notamment pour permettre de garantir le respect de certains protocoles complexes utilisés pour la programmation système : ainsi, typiquement, un fichier doit être ouvert avant d'être utilisé et ne peut plus être consulté ou modifié une fois qu'il est fermé. Plus généralement, et qu'il s'agisse de gestion des entrées/sorties, de la mémoire ou même du parallélisme, un grand nombre d'opérations ne peuvent être accomplies que sur des entités qui sont dans un état approprié.

Le système de types de Vault sert notamment à garantir ces propriétés. Sans entrer dans les spécificités sans rapport avec les entités, précisons que les types de Vault sont bien plus riches que les types `C`, au sens où Vault autorise le polymorphisme paramétrique et des types union sûrs (par opposition aux `unions` de `C`).

2.5.1 Spécification de protocoles

Le langage de types de Vault permet de définir des *clés*, éventuellement associées à un ensemble d'états représentés par des noms. Ainsi, `tracked(K) FILE input` déclare la variable `input` dont la structure de donnée est `FILE`

et qui est gardée par la nouvelle clef `K`. De même, `K :FILE output` déclare la variable `output` de type `FILE`, gardée par la même clef `K`, précédemment créée.

Par définition, une variable est gardée par une clef `K` lorsque, sur tout le domaine de définition de la variable, la clef `K` doit avoir été *acquise*. Si l'annotation spécifie un état, sous la forme `K@open :FILE file`, en plus d'avoir été acquise, la clef doit être dans l'état correspondant – ici, `open`. Cette gestion des acquisitions et des états est entièrement gérée par le système de types statique.

Le type d'une fonction peut alors spécifier des pré- et post-conditions sur l'utilisation des entités. Ainsi, `void fclose(tracked(F) FILE f) [-F@open]` est la signature d'une fonction nommée `fclose`, qui ne renvoie aucun résultat, qui prend un paramètre `f` qui est une structure de données `FILE`, gardé par une clef dont le nom sera lié à `F`, qui doit être acquise dans l'état `open` lors de l'invocation de `fclose` et qui sera relâchée au retour de la fonction. Notons que le type d'une fonction peut aussi préciser que la fonction acquiert une clef connue ou fraîche ou change l'état d'une clef sans la relâcher.

Notons aussi que Vault peut statiquement gérer des clés anonymes ainsi que des clés en nombre inconnu au moment de la compilation – avec des limitations sur les opérations possibles sur ces clés.

Enfin, Vault dispose de primitives pour gérer la mémoire par *régions* [81]. Ainsi, au moment d'allouer une entité sur le tas, il est possible de préciser que cette entité appartient à une région nommée. Toutes les entités appartenant à une région sont alors désallouées simultanément, lorsque la région est elle-même désallouée. Le système de types peut vérifier statiquement que le programme ne cherche jamais à accéder à une entité lorsque la région qui la contient est détruite et que toutes les régions créées sont un jour supprimées.

2.5.2 Derrière les types de Vault

Le système de types de Vault est construit à partir du Calcul des Capacités [23] et des types d'alias [76]. Le Calcul des Capacités est un formalisme qui permet d'écrire des programmes à l'aide de continuations (CPS) dans lequel l'allocation et la désallocation de mémoire sont explicites, ainsi que la gestion de mémoire par régions.

Sans entrer dans les détails, nous nous contenterons de préciser que `void fclose(tracked(F) FILE f) [-F@open]` se traduit par

$$\forall \rho_F.\forall \epsilon.(\epsilon \oplus \{\rho_F@\texttt{open} \mapsto \texttt{FILE}\}, s(\rho_F)) \to (\epsilon, \texttt{void}) \ .$$

Dans cette formule,

- ϵ représente le reste des clés acquises, qui reste inchangé par `fclose`
- $\oplus$ ajoute à ϵ une clef supplémentaire
- $\rho_F@\texttt{open} \mapsto \texttt{FILE}$ est un motif qui accepte n'importe quelle clef associée à une valeur de type `FILE` et dans l'état `open` et lui donne le nom ρ_F
- $\epsilon \oplus \{\rho_F@\texttt{open} \mapsto \texttt{FILE}\}$ représente donc l'ensemble des clés acquises au lancement de la fonction
- $s(\rho_F)$ représente le fait que la fonction n'accepte qu'un argument qui doit être lié à la clef ρ_F.

Ce système de types, très puissant, ne dispose pas d'algorithme d'inférence automatique.

2.5.3 Discussion

L'approche employée par Vault est intéressante car elle permet effectivement de conserver un langage proche du `C` tout en assurant le respect de protocoles complexes. De plus, le système de types de Vault est suffisamment puissant pour gérer certains aspects du parallélisme – notamment la possibilité de passer explicitement des clés d'un thread à un autre ou de créer des verrous "sûrs". Enfin, même si la gestion de quantités arbitraires de clés est complexe, il ne semble pas s'agir d'une réelle limitation.

Vault n'est cependant pas exempt de défauts. Ainsi, si l'idée de prouver des propriétés de programmes écrits dans un langage proche de `C` est intéressante, on peut constater que l'apparence `C` de Vault est essentiellement une manipulation syntaxique pour cacher un langage beaucoup plus contraint – un langage manifestement sans pointeurs explicites et dans lequel, par exemple, la sémantique des types unions est sûre. En particulier, Vault ne peut donc pas être interfacé avec `C` beaucoup plus simplement qu'un langage fonctionnel et risque de ne pas être si utile pour employer les librairies systèmes de `C`.

De plus, si de nombreux protocoles peuvent être décrits pour Vault, cette description est limitée par le fait que les états des clés doivent être tous définis statiquement et ne peuvent être construits comme des ensembles. En particulier, cela signifie qu'il est difficile de compter à l'aide de clés – Vault n'est donc probablement pas approprié pour régler les problèmes qui nous intéressent dans ce document, à savoir la gestion et la réutilisation de ressources rassemblées en une réserve limitée en taille.

2.6 Analyse d'Utilisation des Ressources

L'Analyse d'Utilisation des Ressources s'applique à un λ-calcul légèrement étendu. L'objectif de cette analyse est, comme dans Vault, d'assurer que des protocoles complexes sont respectés. Ainsi, un fichier peut être ouvert, lu, écrit, refermé, à condition d'ouvrir avant de lire ou d'écrire, de refermer avant de terminer et de ne plus rien faire sur ce fichier après l'avoir refermé.

L'analyse repose sur une formulation de protocoles à l'aide de *traces* puis sur une inférence de types sur le terme obtenu.

Grossièrement, les protocoles sont décrits sous la forme de langages – finis ou infinis, réguliers ou non – puis une forme de model checking permet de vérifier que tous les mots engendrés par les exécutions possibles d'un terme entrent dans ce langage.

2.6.1 Spécification de protocoles

Le langage étudié dans RUA est un λ-calcul légèrement étendu. Si nous considérons une seule sorte d'entités, les seules primitives inhabituelles sont $\texttt{new}^{\Phi}()$ et $\texttt{acc}_i^l(M)$ – où M représente un terme. La construction $\texttt{new}^{\Phi}()$ crée une nouvelle entité avec la trace Φ tandis que $\texttt{acc}_i^l(M)$ représente un accès à l'entité désignée par M, avec l'effet l – les effets ne sont que de simples labels.

Ainsi, si nous considérons comme entités des fichiers, $\texttt{new}^{\Phi}()$ permettra de créer la structure de données pour le fichier qui devra obéir à la politique spécifiée

$$\text{UR-Zero } l \xrightarrow{l} \mathbf{0} \qquad \text{UR-SeqL } \frac{U_1 \xrightarrow{l} U_1'}{U_1; U_2 \xrightarrow{l} U_1'; U_2}$$

$$\text{T-New } \frac{[|U|] \subseteq \Phi}{\emptyset \vdash \texttt{new}^{\Phi}() : (\mathbf{R}, U)}$$

FIG. 2.6 – Un fragment du système de types de RUA.

dans Φ et nous pourrons noter

$$\begin{cases} \texttt{read} & = & \lambda x.\texttt{acc}^{c}_{reader}(M) \\ \texttt{write} & = & \lambda x.\texttt{acc}^{w}_{writer}(M) \\ \texttt{open} & = & \lambda x.\texttt{acc}^{o}_{opener}(M) \\ \texttt{close} & = & \lambda x.\texttt{acc}^{c}_{closer}(M) \ . \end{cases}$$

Spécifier qu'un fichier doit être ouvert avant toute autre chose, peut ensuite être lu et écrit et ne doit plus être touché après avoir été fermé peut se faire à l'aide d'une trace $\Phi = \{o(r+w)^* c \downarrow\}^\sharp$ – dans cette expression, $\sharp$ est un opérateur qui dénote l'ensemble des préfixes et $\downarrow$ désigne la sortie du domaine de définition de la variable. De même, $\Phi' = \{or^* c \downarrow\}^\#$ spécifiera la politique appropriée à un fichier en lecture seule.

Les fonctions et points fixes sont eux aussi annotées par une trace, tandis que les applications d'un terme à un autre sont annotées par un label – nous ne détaillerons pas le fonctionnement, complexe, de ces éléments.

Un terme respectera alors les protocoles si la trace Φ considérée en tant que langage sur des mots finis reconnaît le mot constitué des labels qui annotent les accès à la variable et terminé par $\downarrow$, lorsque la variable cesse d'exister.

2.6.2 Le système de types

Le système de types de RUA permet de prouver que certains termes respectent les protocoles. Ce système est entièrement inféré et permet d'exprimer des propriétés élémentaires (il n'est pas possible d'accéder à une entité/il n'est possible d'accéder à une entité qu'à l'aide d'une action étiquetée par un label donné), la concaténation, la disjonction, l'entrelacement, ainsi que plusieurs formes de répétition.

Un bref fragment des règles utilisées pour le typage est présenté sur la figure 2.6. Ainsi, UR-ZERO donne le comportement d'un type qui n'accepte qu'un seul label, tandis que UR-SEQL spécifie le comportement d'un type concaténé. La règle de typage T-NEW, quant à elle, précise que le type d'une entité ne peut être $(\mathbf{R}, U)$ – où $\mathbf{R}$ représente, pour résumer, le type fonctionnel de l'entité – que si tous les comportements possibles de U sont acceptables par Φ.

Nous n'entrerons pas plus dans les détails. Notons juste que la sémantique de cette règle T-NEW est en fait relativement proche de ce que nous cherchons avec T-AMB, T-SEAL, T-CELL...Cet aspect sera plus visible encore dans le chapitre 9.

2.6.3 Discussion

Si l'approche de RUA est duale par rapport à celle de Vault, les deux méthodes cherchent à prouver des propriétés essentiellement similaires. Il est à noter que le système de types de RUA semble beaucoup plus simple que celui de Vault, notamment car il ne nécessite pas de types existentiels. Si RUA ne propose pas de polymorphisme et gère imprécisément les alias, les auteurs de ce travail donnent plusieurs pistes pour étendre leurs résultats.

Tout comme Vault, RUA peut probablement s'adapter à des extensions parallèles du calcul, quoique probablement plus difficilement. De plus, RUA semble moins limité que Vault dans l'expressivité des traces, notamment lorsqu'il s'agit de compter le nombre d'accès à une ressource. Par contre, en l'absence de primitives de parallélisme, il semble pour l'instant impossible d'appliquer Vault au problème de la gestion des ressources limitées en présence de concurrence et de mobilité.

Nous retrouverons dans le chapitre 9 certains aspects de nos systèmes de types proches de RUA.

Deuxième partie

Mobilité de sites

Chapitre 3

Ambients

Dans cette partie, nous approchons le problème de la gestion des ressources dans le cadre de calculs de sites mobiles. Nous commencerons par présenter les Mobile Ambients (ou MA), en tant que représentant de cette catégorie d'algèbres de processus. Dans le cadre des MA, un système est représenté par une hiérarchie de sites – *ambients* – de profondeur arbitraire. En plus de sous-ambients, chaque ambient peut contenir des processus qui vont exécuter parallèlement leurs *capacités* et modifier localement la structure du système en causant la création d'un sous-site, la dissolution d'un sous-site (ou "ouverture") ou le déplacement du site conteneur.

Le mouvement est dit *subjectif* car le déplacement d'un ambient est déterminé uniquement par son contenu : on considère que l'ambient décide lui-même, par le biais de ses processus, de ses déplacements. De plus, l'approche du parallélisme, à l'opposé des modèles coopératifs et synchrones présentés dans la section 2.3, ne fait qu'une seule hypothèse sur l'ordonnancement des processus : les opérations de base sont atomiques et ininterruptibles.

Ainsi, si l'on cherche à représenter un réseau à l'aide des MA, chaque ordinateur sera représenté par un ambient, chaque message sur le réseau par un ambient, chaque application par un ambient et le réseau lui-même par un ambient. De même, pour représenter un canal de communication ou un fichier, nous emploierons un ambient. En d'autres termes, les ambients sont des entités qui peuvent être créées et détruites, et ce sont les entités centrales du langage, que nous pouvons gagner à contrôler.

La notion naturelle de ressource consiste donc à considérer que chaque ambient occupe un certain nombre de ressources dans l'ambient qui le contient. Ainsi, un ambient a ne peut entrer dans un ambient b que si b dispose de suffisamment de ressources pour l'accueuillir. De même, si a sort de b, b récupère des ressources. En d'autres termes, chaque déplacement d'un ambient est à la fois une désallocation dans le site d'origine et une allocation dans le site destination.

BoCa Les auteurs de BoCa [7] ont choisi de représenter explicitement les ressources dans le calcul par une primitive ▬. De plus, ils introduisent une notion de "poids" pour un processus P, qui est égal à la somme du nombre de ressources utilisées par P sous la forme de sous-ambients et du nombre de ressources offertes par P sous la forme de ▬.

Ainsi, $a^2\,[\blacksquare \mid \blacksquare]$ représente un ambient qui contient deux ressources, toutes

deux notées $\blacksquare$ et qui occupe lui-même deux ressources libres. De même, le processus $\texttt{in}\ b.a^2\,[\blacksquare \mid \blacksquare]$, qui commence par déplacer l'ambient courant vers l'ambient b puis qui crée un nouvel ambient a a aussi un poids de 2. Par définition du poids, comme a est annoté par 2, le contenu de a doit avoir un poids exactement égal à 2 et le poids de $a^2\,[\blacksquare \mid \blacksquare]$ est lui aussi égal à 2. Si a se déplace, par exemple pour entrer dans b, 2 ressources doivent être allouées dans b et 2 ressources seront libérées de l'ambient qui contenait a :

$$a^2\,[\texttt{in}\ b.P] \mid b\,[\blacksquare \mid \blacksquare] \longrightarrow b\,[a\,[2]\,P] \mid \blacksquare \mid \blacksquare$$

Afin d'éviter qu'un processus répliqué ne consomme toutes les ressources, ils introduisent un autre constructeur : $k \rhd P$, ou spawner, de poids nul et défini uniquement lorsque le poids de P et égal à k. Alors,

$$k \rhd P \mid \underbrace{\blacksquare|\blacksquare|\ldots\blacksquare}_{k} \longrightarrow P\ .$$

Ce constructeur permet de s'assurer qu'un processus répliqué ne peut être activé que s'il dispose de suffisamment de ressources.

Cette sémantique de réduction annotée permet d'exprimer naturellement des contraintes de ressources et de définir naturellement des actions qui ne seront exécutées qu'en présence de suffisamment de ressources libres. Un système de types simple complète BoCa et permet de prouver qu'un terme est ressource-contrôlé.

Controlled Ambients Le calcul des Controlled Ambients [80] (ou CA), développé à peu près en même temps que BoCa, part du principe inverse et offre des primitives de contrôle dynamique des déplacements, complétées par un contrôle statique spécifique aux ressources.

Ainsi, nous avons choisi de ne pas représenter explicitement les ressources ou l'encombrement de chaque ambient. Au lieu de cela, comme dans les Safe Ambients [58] ou ROAM [36], nous introduisons des cocapacités $\overline{\texttt{in}}$, $\overline{\texttt{out}}$, $\overline{\texttt{open}}$ qui permettent d'autoriser les déplacements. À la différence de ces formalismes, certaines synchronisations dans les Controlled Ambients nécessitent trois parties : pour que l'ambient a contenu dans l'ambient b puisse entrer dans l'ambient c, il faut que chacun des ambients a, b et c contienne un processus actif qui accepte ce déplacement. Ainsi, a doit contenir un processus à même d'exercer respectivement la capacité $\texttt{in}\ b$, de même, pour b et la cocapacité $\overline{\texttt{out}}^{\downarrow}\ a$ et pour c et la cocapacité $\overline{\texttt{in}}^{\downarrow}\ a$. Par suite, ces capacités et cocapacités permettent de déclencher des processus lors du déplacement. Ainsi, on aura

$$a\,[\texttt{in}\ b.P] \mid b\left[\overline{\texttt{in}}^{\downarrow}\ a.Q\right] \mid \overline{\texttt{out}}^{\downarrow}\ a.R \longrightarrow b\,[Q \mid a\,[P]] \mid R$$

Afin d'éviter qu'un processus répliqué ne consomme toutes les ressources, nous remplaçons la réplication par une récursion, plus aisée à contrôler. On aura donc $\texttt{rec}\ X.P \longrightarrow P\{X \leftarrow \texttt{rec}\ X.P\}$. La récursion permet de définir plus aisément des processus qui ne seront répliqués que sous conditions.

La récursion et le mécanisme des cocapacités permettent de réagir à l'allocation et à la désallocation de ressources et de n'accepter ces opérations que sous certaines conditions de ressources. Un système de types simple complète les CA et permet de spécifier statiquement des contraintes de ressources puis de prouver qu'un terme est ressource-contrôlé.

Extensions et variantes Le calcul des Seals [19] partage de nombreux aspects avec les Mobile Ambients. Un système est représenté par une hiérarchie de sites – ou *seals* – de profondeur arbitraire. En plus de sous-seals, chaque seal peut contenir des processus qui vont communiquer des informations ou des sous-sites. Cette forme de mouvement est dite *objective* car le déplacement d'un seal est déterminé depuis l'extérieur de ce seal.

En plus de mouvements objectifs, le calcul des Kells[77] ajoute aux Mobile Ambients des communications et des primitives d'ordre supérieur. Pour chacun de ces deux calculs, nous étendons les méthodes que nous avons expérimentées sur les Controlled Ambients.

3.1 Mobile Ambients

Le calcul des Mobile Ambients [16] est construit autour du concept de localité. Un système est constitué d'une hiérarchie de sites nommés – ou *ambients* – qui peuvent chacun contenir, outre d'autres ambients, un nombre quelconque de processus. Ce sont les processus qui font évoluer la structure du système en exerçant des *capacités* qui influencent tout un sous-arbre, soit en le déplaçant, soit en supprimant sa racine.

Un ambient peut représenter aussi bien une localité physique qu'un canal de communication, un processus Un*x, une instance d'objet ou un répertoire dans un système de fichiers hiérarchique. À l'inverse, l'ambient n'est pas la construction appropriée pour décrire un processus léger Un*x (i.e. thread), un composant partagé ou encore une collection d'objets rattachés par des liens uniquement logiques.

3.1.1 Le langage

La syntaxe des Mobile Ambients est présentée sur la figure 3.1. On suppose l'existence d'un ensemble strictement dénombrable a, b, c, m, n ... de noms d'ambients et d'un autre ensemble strictement dénombrable x, y, z ... de noms de variables. On emploiera P, Q ... pour désigner les processus et M, N ... pour les capacités.

Le symbole $\mathbf{0}$ représente le processus terminé, qui n'agit donc plus, tandis que $P|Q$ est la composition parallèle de P et Q. Un processus tel que $!P$ est dit *répliqué* et est équivalent à une infinité de copies de P en parallèle. La construction $n\,[P]$ dénote un processus P exécuté dans un sous-ambient nommé n. Le processus $M.P$ commence par exercer la capacité M puis continue comme P. La *restriction* de n à P, notée $(\nu n)P$, crée un nouveau nom d'ambient n, connu uniquement de P. Une communication permet de passer une capacité M d'un processus émetteur $\langle M \rangle$ à un un processus récepteur $(x).P$ parallèle et localisé dans le même site.

La capacité ϵ, à son tour, ne fait rien. Si M et N sont des capacités, $M.N$ est la concaténation de M et N, qui agit comme M puis comme N. Le détail des capacités `in` n, `out` n et `open` n est précisé sur la figure 3.4 : `in` n fait entrer l'ambient conteneur dans un ambient parallèle n, `out` n fait sortir l'ambient conteneur de son ambient parent n, tandis que `open` n dissout les frontières de l'ambient fils n, laissant son contenu intact. Enfin, x est une variable et n un nom d'ambient.

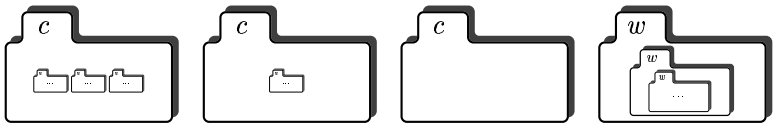
Processus			Capacités		
P, Q	::=	$\mathbf{0}$	M, N	::=	ϵ
	|	$P \mid Q$		|	$M.N$
	|	$!P$		|	`in` M
	|	$M\,[P]$		|	`out` M
	|	$M.P$		|	`open` M
	|	$(\nu n)P$		|	x
	|	$(x).P$		|	n
	|	$\langle M \rangle$			

FIG. 3.1 – Syntaxe des Mobile Ambients.

Dans toute la suite de l'exposé, on notera $n\,[\,]$ pour $n\,[\mathbf{0}]$, $(\nu n_1 \ldots n_p)P$ ou $(\nu \overrightarrow{n})P$ pour $(\nu n_1)\ldots(\nu n_p)P$ et, lorsque cela a un sens, M pour $M.\mathbf{0}$. Il est à noter qu'il existe des termes incorrectement formés. Ainsi, `out` $n\,[P]$ n'a aucun sens.

Les notions de variables libres (désignées fv) et liées (désignées bv) sont définies sur la figure 3.2 et que la substitution sur la figure 3.3.

Ainsi, si l'on veut décrire un réseau composé d'ordinateurs c et de messages w, chaque ordinateur et chaque message pouvant à son tour contenir d'autres messages w, on écrira un terme tel que



soit

$$c\,[w\,[\ldots]\ \mid w\,[\ldots]\ \mid w\,[\ldots]]\ \mid c\,[w\,[\ldots]]\ \mid c\,[\mathbf{0}]\ \mid w\,[w\,[w\,[\ldots]]]$$

La sémantique des Mobile Ambients se définit en deux temps. Tout d'abord, par une relation de *congruence structurelle*, qui regrouper en classes d'équivalence des termes considérés comme sémantiquement identiques. Comme dans tous les calculs qui suivent, on identifie notamment les termes égaux après renom-mage d'un nom lié. Ainsi, on aura $(\nu n)P = (\nu m)P\{n \leftarrow m\}$ et $(n).P = (m).P\{n \leftarrow m\}$ lorsque m est frais dans P. De même, on aura $a\,[P] \equiv a\,[Q]$ dès que $P \equiv Q$.

Définition 4 (Congruence structurelle)
La congruence structurelle est la plus petite relation $\equiv$ qui soit une relation de congruence compatible avec les règles de la figure 3.5, telle que $(P/\equiv, \mid, \mathbf{0})$ soit un monoïde commutatif et associatif et qui soit compatible avec l'α-équivalence des variables liées.

Il est à noter que, contrairement à ce que l'on peut observer dans des calculs comme Dπ, $n\,[P \mid Q]$ et $n\,[P]\ \mid n\,[Q]$ sont deux entités totalement distinctes : dans le premier cas, un ambient nommé n contient les processus P et Q, alors que dans le deuxième, deux ambients frères tous deux nommés n contiennent l'un le processus P et l'autre le processus Q.

$fv(\mathbf{0}) = \emptyset$ $fv(P\|Q) = fv(P) \cup fv(Q)$ $fv(!P) = fv(P)$ $fv(M\,[P]) = fv(P) \cup fv(M)$ $fv(M.P) = fv(P) \cup fv(M)$ $fv((\nu n)P) = fv(P) \backslash \{n\}$ $fv((x).P) = fv(P) \backslash \{x\}$ $fv(\langle M \rangle) = fv(M)$	$fn(\mathbf{0}) = \emptyset$ $fn(P\|Q) = fn(P) \cup fn(Q)$ $fn(!P) = fn(P)$ $fn(M\,[P]) = fn(P) \cup fn(M)$ $fn(M.P) = fn(P) \cup fn(M)$ $fv((\nu n)P) = fn(P) \backslash \{n\}$ $fv((x).P) = fn(P) \backslash \{x\}$ $fv(\langle M \rangle) = fn(M)$
$fv(\epsilon) = \emptyset$ $fv(M.N) = fv(M) \cup fv(N)$ $fv(\texttt{in}\ M) = fv(M)$ $fv(\texttt{out}\ M) = fv(M)$ $fv(\texttt{open}\ M) = fv(M)$ $fv(x) = \{x\}$ $fv(n) = \emptyset$	$fn(\epsilon) = \emptyset$ $fn(M.N) = fn(M) \cup fn(N)$ $fn(\texttt{in}\ M) = fn(M)$ $fn(\texttt{out}\ M) = fn(M)$ $fn(\texttt{open}\ M) = fn(M)$ $fn(x) = \emptyset$ $fn(n) = \{n\}$
$bv(\mathbf{0}) = \emptyset$ $bv(P\|Q) = bv(P) \cup bv(Q)$ $bv(!P) = bv(P)$ $bv(M\,[P]) = bv(P)$ $bv(M.P) = bv(P)$ $bv((\nu n)P) = bv(P) \cup \{n\}$ $bv((x).P) = bv(P) \cup \{x\}$ $bv(\langle M \rangle) = \emptyset$ $bv(M) = \emptyset$	

FIG. 3.2 – Variables libres, noms libres et variables liées dans les Mobile Ambients.

Le comportement des processus est alors donné par les règles de réduction de la figure 3.6.

3.1.2 Exemples

Verrous

Contrairement à des calculs comme CCS, les opérations des Mobile Ambients sont essentiellement asynchrones au sens où, même si une réduction peut impliquer plusieurs processus, seul l'un des processus est *a priori* bloqué jusqu'à ce que la réduction soit effectuée.

De manière à permettre une mesure de synchronisation, on utilise donc des constructions telles que les verrous. Un verrou est un composant partagé entre plusieurs processus. Chaque processus peut demander à acquérir le verrou puis, l'ayant acquis, à le relâcher. Par définition, à chaque instant, au plus un processus peut posséder le verrou. Les autres processus cherchant à l'acquérir attendent donc jusqu'à ce qu'il soit relâché.

$\mathbf{0}\{a \leftarrow b\}$	$= \mathbf{0}$	
$(P\|Q)\{a \leftarrow b\}$	$= P\{a \leftarrow b\} \mid Q\{a \leftarrow b\}$	
$(!P)\{a \leftarrow b\}$	$=!(P\{a \leftarrow b\})$	
$M\,[P]\,\{a \leftarrow b\}$	$= M\{a \leftarrow b\}\,[P\{a \leftarrow b\}]$	
$(M.P)\{a \leftarrow b\}$	$= (M\{a \leftarrow b\}).(P\{a \leftarrow b\})$	
$((\nu n)P)\{a \leftarrow b\}$	$= (\nu n)(P\{a \leftarrow b\})$	si $a \neq n$,$b \neq n$
$((\nu n)P)\{a \leftarrow b\}$	$= (\nu n)P$	si $a = n$
$((x).P)\{a \leftarrow b\}$	$= (x).(P\{a \leftarrow b\})$	si $x \neq a$, $x \neq b$
$((x).P)\{a \leftarrow b\}$	$= (x).P$	si $x = a$
$\langle M\rangle\{a \leftarrow b\}$	$= \langle M\{a \leftarrow b\}\rangle$	
$\epsilon\{a \leftarrow b\}$	$= \epsilon$	
$(M.N)\{a \leftarrow b\}$	$= M\{a \leftarrow b\}.N\{a \leftarrow b\}$	
$\texttt{in}\ M\{a \leftarrow b\}$	$= \texttt{in}\ (M\{a \leftarrow b\})$	
$\texttt{out}\ M\{a \leftarrow b\}$	$= \texttt{out}\ (M\{a \leftarrow b\})$	
$\texttt{open}\ M\{a \leftarrow b\}$	$= \texttt{open}\ (M\{a \leftarrow b\})$	
$x\{a \leftarrow b\}$	$= x$	si $a \neq x$
$x\{a \leftarrow b\}$	$= b$	si $a = x$
$n\{a \leftarrow b\}$	$= n$	si $n \neq a$
$n\{a \leftarrow b\}$	$= b$	si $n = a$

FIG. 3.3 – Substitutions dans les Mobile Ambients.

Cette construction peut s'écrire simplement à l'aide des macros suivantes :

$$\begin{aligned} \texttt{Acq}\ n.P &\triangleq \texttt{open}\ n.P \\ \texttt{Rel}\ n.Q &\triangleq n\,[\,] \mid Q \end{aligned}$$

Généralement, afin d'éviter des collisions sur le nom n, on s'assurera au préalable que le nom n est secret. Il est à noter que ce verrou ne fonctionne que localement. Écrire un verrou distribué entre plusieurs localités est une tâche plus compliquée, a fortiori si les sites se déplacent les uns par rapport aux autres.

On peut définir une version légèrement étendue de ces processus qui permette de relâcher le verrou à distance. Pour ce faire, au moment où l'on relâche le verrou, on lui fournit une capacité pour permettre à l'ambient qui le représente de se rendre là où il est nécessaire.

$$\texttt{Rel}\ (M,n).Q \triangleq n\,[M.\mathbf{0}] \mid Q$$

Synchronisation sur un canal

La synchronisation sur un canal est la primitive majeure du langage CCS. Il s'agit d'une réaction entre deux – et exactement deux – processus au cours de laquelle, avec des notations non-canoniques, un terme $\texttt{Up}\ n.P$ et un terme $\texttt{Down}\ n.Q$ composés en parallèle réagissent et se réduisent en $P|Q$. On dit alors qu'ils se sont synchronisés sur le canal n.

Si l'on impose que $\texttt{Up}\ n.P$ et $\texttt{Down}\ n.Q$ soient présents dans le même ambient et que, pour simplifier les communications et la vérification du protocole, le canal

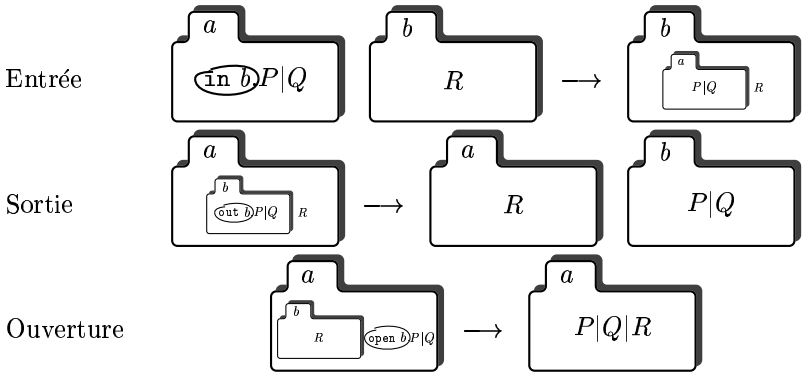


FIG. 3.4 – Le rôle des capacités.

STRUCT-REPL-PAR $P \equiv !P|P$

STRUCT-RES-RES $(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$ si $n \neq m$

STRUCT-RES-PAR $(\nu n)(P|Q) \equiv P|(\nu n)Q$ si $n \notin fv(P)$

STRUCT-RES-AMB $(\nu n)m\,[P] \equiv m\,[(\nu n)P]$ si $n \neq m$

STRUCT-ZERO-RES $(\nu n)\mathbf{0} \equiv \mathbf{0}$ STRUCT-ZERO-REPL $!\mathbf{0} \equiv \mathbf{0}$

FIG. 3.5 – Congruence structurelle dans les Mobile Ambients.

R-IN $a\,[\texttt{in}\ b.P \mid Q] \mid b\,[R] \longrightarrow b\,[a\,[P|Q] \mid R]$

R-OUT $b\,[a\,[\texttt{out}\ b.P \mid Q] \mid R] \longrightarrow a\,[P|Q] \mid b\,[R]$

R-OPEN $\texttt{open}\ b.P \mid Q \mid b\,[R] \longrightarrow P \mid Q \mid R$

R-COMM $\langle M \rangle \mid (x).P \longrightarrow P\{x \leftarrow M\}$ R-RES $\dfrac{P \longrightarrow Q}{(\nu n)P \longrightarrow (\nu n)Q}$

R-AMB $\dfrac{P \longrightarrow Q}{n\,[P] \longrightarrow n\,[Q]}$ R-PAR $\dfrac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$

R-STRUCT $\dfrac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}$

FIG. 3.6 – Réduction dans les Mobile Ambients

corresponde à un seul nom et non à un n-uplet, le terme obtenu est relativement complexe.

$$\begin{array}{lcl} \texttt{Chan}\ n & = & n\,[!\texttt{open}\ i] \\ \texttt{Up}\ n.P & = & (\nu l)(i\,[\texttt{in}\ n.\texttt{Rel}\ up.\texttt{Acq}\ dn.l\,[\texttt{out}\ n]]\ |\ \texttt{open}\ l.P) \\ \texttt{Down}\ n.Q & = & (\nu l)(i\,[\texttt{in}\ n.\texttt{Rel}\ dn.\texttt{Acq}\ up.l\,[\texttt{out}\ n]]\ |\ \texttt{open}\ l.Q) \end{array}$$

Nous représentons explicitement le canal n par un ambient. Cet ambient, nommé n, se contente de recevoir des ambients i contenant des instructions et de les ouvrir. Une synchronisation $\texttt{Up}\ n.P$ se servira de ces instructions pour relâcher un verrou nommé up et acquérir un verrou nommé dn à l'intérieur de n – up et dn sont des noms prédéfinis, contrairement à n qui est un paramètre. À l'inverse, le terme $\texttt{Down}\ n.Q$ relâche dn puis acquiert up. Enfin, les deux termes se servent de deux noms secrets tous deux notés l pour lancer effectivement P et Q.

Comme dans le cas du verrou, un processus malicieux qui connaîtrait le nom n pourrait sans difficultés saboter la communication en violant le protocole, par exemple en ouvrant n.

Communication sur un canal

Les canaux de synchronisation sur le mode de CSS peuvent être étendus en des canaux de communication proches de ceux du π-calcul :

$$\begin{array}{lcl} \texttt{Chan}\ n & = & n\,[!\texttt{open}\ i] \\ \texttt{Send}\ M\ \texttt{on}\ n.P & = & (\nu l)(\\ & & \quad i\,[\texttt{in}\ n.(\langle M\rangle\ |\ \texttt{Rel}\ up.\texttt{Acq}\ dn.l\,[\texttt{out}\ n])]\ | \\ & & \quad \texttt{open}\ l.P \\ & &) \\ \texttt{Receive}\ x\ \texttt{on}\ n.Q & = & (\nu l, m)(i[\texttt{in}\ n.\texttt{Rel}\ dn.\texttt{Acq}\ up. \\ & & \quad (x).l\,[\texttt{out}\ n.\texttt{open}\ m.Q] \\ & & \quad]\ |\ \texttt{open}\ l.m\,[] \\ & &)\ . \end{array}$$

Le principe de base reste le même. La modification la plus importante est l'ajout d'une communication à l'intérieur de l'ambient n, qui permet d'échanger les messages. Notons que cette version nécessite de déplacer le processus Q pour le placer là où la communication peut avoir un effet sur lui – dans le cadre d'une implantation distribuée des Mobile Ambients, cette opération n'est pas anodine car elle peut elle-même nécessiter des communications d'ordre supérieur.

Le protocole du taxi

Présentation du protocole Considérons maintenant un exemple plus complexe : le protocole du taxi. Même si le terme ne modélise pas, à notre connaissance, de méthode réellement utilisée dans les systèmes parallèles ou distribués, il illustre certaines des difficultés liées à la conception de protocoles robustes.

Le système, présenté sur la figure 3.7, représente une ville. La ville contient un certain nombre de sites s_1, s_2 ... et de taxis. Chaque site peut à son tour contenir des clients et des taxis. À tout instant, un client peut appeler un taxi.

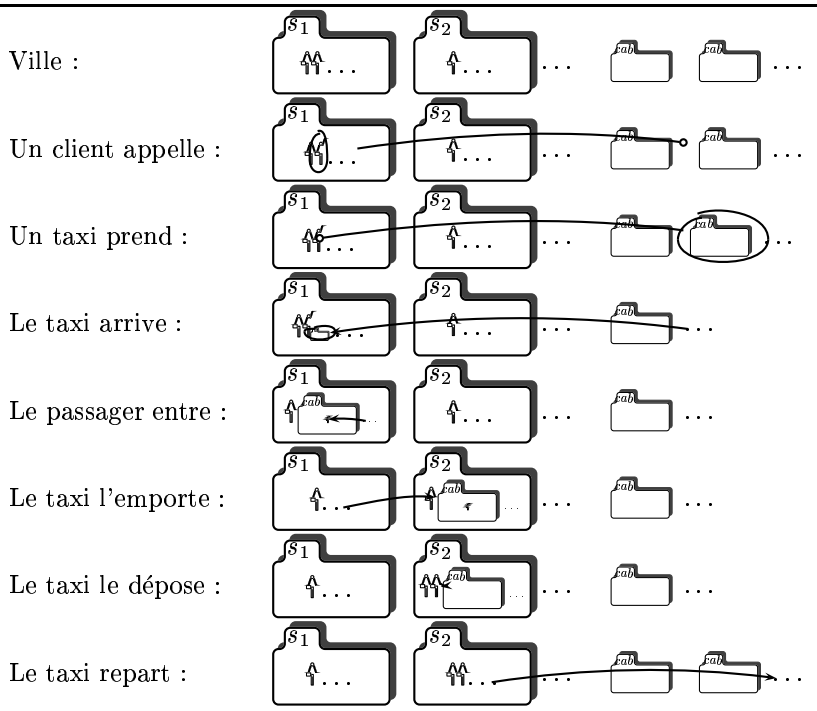


FIG. 3.7 – Le protocole du taxi.

L'un des taxis présents dans la ville doit alors venir le chercher et le déposer dans son site de destination puis repartir.

$$
\begin{array}{lcl}
\texttt{Call}\ f\ c & \triangleq & r\,[\texttt{out}\ c.\texttt{out}\ f.\texttt{in}\ cab.\texttt{in}\ f.l\,[\texttt{out}\ cab.\texttt{in}\ c]] \\
\texttt{Trip}\ f\ t\ c & \triangleq & trip\,[\texttt{out}\ c.\texttt{out}\ f.\texttt{in}\ t.u\,[\texttt{in}\ c]] \\
\texttt{Bye}\ c & \triangleq & b\,[\texttt{out}\ c.\texttt{in}\ cab.\texttt{out}\ t] \\
\texttt{Client}\ f\ t & \triangleq & (\nu c)c[\texttt{Call}\ f\ c \mid \texttt{open}\ l.\texttt{in}\ cab.\texttt{Trip}\ f\ t\ c \\
 & & \quad \mid \texttt{open}\ u.\texttt{out}\ cab.\texttt{Bye}\ c \\
\texttt{Cab} & \triangleq & cab[\texttt{Rel}\ n \mid !\texttt{Acq}\ n.\texttt{open}\ r.\texttt{open}\ trip.\texttt{open}\ b.\texttt{Rel}\ n]
\end{array}
$$

FIG. 3.8 – Le protocole du taxi – implantation en MA.

Une implantation La figure 3.8 présente une implantation possible du protocole du taxi dans les Mobile Ambients. Les sites sont représentés par des ambients nommés s_1, s_2 ... les taxis par des ambients nommés *cab* et les clients par des ambients de noms quelconques.

Le taxi se compose essentiellement d'une boucle qui attend et exécute les instructions, qui arrivent sous la forme d'un ambient nommé r (appel d'un client) puis sous la forme d'un ambient nommé *trip* (instructions données par

le client) et enfin comme un ambient nommé b (congé donné par le client). Exécuter des instructions se résume à ouvrir l'ambient qui les contient.

Un client c présent dans le site f appelle un taxi à l'aide de la macro `Call` f c : il émet un ambient nommé r qui sort de c et de f, entre dans un taxi quelconque et, après ouverture, conduit le taxi dans f. Lorsque le taxi est arrivé, le client peut entrer et donner les instructions à l'aide de la macro `Trip` f t c : il émet un nouvel ambient nommé *trip*, qui sort de c, se retrouve dans le taxi et, après ouverture, conduit le taxi hors de f et dans t puis signale l'arrivée par un ambient u, qui retourne dans le client. Une fois ce signal reçu, le client sort du taxi et signale à celui-ci, par l'ambient b, qu'il peut partir.

Limites de l'implantation Cette implantation met en évidence plusieurs difficultés que l'on rencontre lorsqu'on cherche à modéliser des protocoles non-triviaux à l'aide des Mobile Ambients. Ainsi, le mécanisme de synchronisation, assez lourd, qui emploie les ambients u et b, est nécessaire pour éviter que le client ne quitte le taxi avant que celui-ci soit arrivé à destination ou que le taxi ne reparte avant d'avoir déchargé son client.

Malheureusement, ce mécanisme ne suffit pas. En effet, considérons le code de l'ambient b : $b\,[\texttt{out}\ c.\texttt{in}\ cab.\texttt{out}\ t]$. L'ambient b quitte le client c puis entre dans un taxi *cab*. Or, si plusieurs taxis sont présents dans le site, rien ne permet *a priori* de les différencier. Un autre taxi peut donc recevoir, par erreur, le signal et quitter le site avant d'avoir déchargé son client c_2, alors que le taxi qui est effectivement vide restera sur place, en attendant un message qui ne viendra probablement jamais. Enfin, c_2, qui cherchera toujours à quitter son *cab*, pourra finir par se retrouver hors de tout site.

S'il est vrai que ces problèmes sont liés à notre manière d'implanter le protocole, ils mettent en évidence certaines difficultés caractéristiques des Mobile Ambients. Ainsi, à moins d'implanter un protocole complexe – et donc difficile à vérifier – il n'est pas possible d'imposer à c_2 de ne quitter *cab* que s'il est dans un site. Il n'est pas non plus possible de spécifier à *cab* de refuser des clients.

Le concept des *cocapacités* a été introduit dans les Safe Ambients [58] afin de simplifier la conception de protocoles sûrs et le raisonnement sur les processus. Ces constructions, dont nous retrouverons des variantes aussi bien dans BoCa (Section 3.2) que dans les Controlled Ambients (Section 3.3), répartissent la responsabilité des actions sur plusieurs sites et permettent de refuser certaines actions ou de ne les accomplir que sous certaines conditions. Ainsi, dans les Safe Ambients, un ambient a ne peut entrer dans un ambient b que si b contient la cocapacité $\overline{\texttt{in}}\ b$. Comme nous le verrons dans les sections suivantes, cette notion, qui ajoute un degré de filtrage des interactions entre ambients, simplifie grandement la conception de protocoles sûrs, notamment du point de vue des ressources.

Attaques

Plutôt que de présenter les multiples attaques possibles sur un protocole non-trivial comme celui du taxi, contentons-nous de considérer un site serveur. Ce site reçoit des requêtes, sous la forme d'ambients nommés *request*, et y répond à l'aide d'autres ambients nommés *reply*. Une requête doit contenir le chemin que doit emprunter la réponse, sous la forme de l'émission d'une capacité M composée de la suite des mouvements nécessaires pour entrer dans le client.

$$\begin{array}{lcl} \texttt{Server} & = & s\,[!\texttt{open}\ request.(M).reply\,[\texttt{out}\ s.M]] \\ \texttt{Client}\ c & = & c\,[request\,[\texttt{out}\ c.\texttt{in}\ s.\langle\texttt{in}\ c\rangle]] \end{array}$$

FIG. 3.9 – Protocole client-serveur minimal.

$\texttt{Rogue}_1$	=	$\texttt{open}\ s$	détruit le serveur
$\texttt{Rogue}_2$	=	$s\,[]$	se substitue au serveur
$\texttt{Rogue}_3$	=	$request\,[\texttt{in}\ s.!\texttt{open}\ reply]$	détruit les réponses
$\texttt{Rogue}_4$	=	$request\,[\texttt{in}\ s.!(M).\mathbf{0}]$	intercepte les requêtes

FIG. 3.10 – Attaques sur le protocole client-serveur minimal.

Ce protocole est décrit sur la figure 3.9. Un terme tel que

$$P = \texttt{Server} \mid \texttt{Client}\ c_1 \mid \texttt{Client}\ c_2 \mid \ \ldots\ \mid \texttt{Client}\ c_n$$

s'exécute alors comme spécifié. Cependant, le système est sensible à un certain nombre d'attaques, dont quelques-unes sont représentées sur la figure 3.10.

Ainsi, la composition $P|\texttt{Rogue}_1$ risque de détruire l'ambient s qui contient le serveur. La composition $P|\texttt{Rogue}_2$, à l'inverse, risque de détourner une partie des messages destinés au serveur. Un peu plus subtils, les processus $\texttt{Rogue}_3$ et $\texttt{Rogue}_4$ sont des chevaux de Troie qui s'introduisent dans le serveur et respectivement détruisent les réponses ou interceptent les requêtes. Il serait possible de réécrire `Server` de manière à empêcher ces attaques, au prix de la lisibilité du terme.

En effet, le seul moyen d'empêcher un processus de dissoudre s est de garder le nom s secret et d'introduire un agent qui permet de faire entrer les requêtes dans s sans révéler ce nom. Mais cet agent, lui-même connu, sera à son tour susceptible d'être attaqué et posera donc d'autres problèmes . Tout ceci pour se défendre contre le simple processus $\texttt{Rogue}_1$ – les attaques $\texttt{Rogue}_2$, $\texttt{Rogue}_3$ et $\texttt{Rogue}_4$ seraient bien plus complexes à gérer.

Les attaques $\texttt{Rogue}_1$, $\texttt{Rogue}_3$ et $\texttt{Rogue}_4$ entrent toutes dans la même catégorie : des processus exercent des capacités qu'ils ne devraient pas être autorisés à avoir, respectivement $\texttt{open}\ s$, $\texttt{open}\ reply$ et (M). Les attaques $\texttt{Rogue}_3$ et $\texttt{Rogue}_4$, de plus, consistent en une appropriation du protocole, profitant du fait que le serveur exécute aveuglément du code reçu d'une source falsifiable. Quant à $\texttt{Rogue}_2$, s'il s'agit encore d'une appropriation du protocole, cette fois, elle profite du fait que le client envoie une requête à une destination falsifiable.

Afin d'empêcher l'attaque $\texttt{Rogue}_1$, on peut donc imaginer utiliser des co-capacités. Contre $\texttt{Rogue}_2$, $\texttt{Rogue}_3$ et $\texttt{Rogue}_4$, il faudrait plutôt introduire des mécanismes d'authentification. Dans la section 11, nous verrons un système de types dépendants pour les Mobile Ambients qui permet de détecter statiquement certaines de ces attaques.

3.1.3 Ressources

Les ressources dans les Mobile Ambients

Considérons maintenant le client et le serveur de la figure 3.9 du point de vue des ressources. Comme un serveur réel n'a qu'une quantité de mémoire et un nombre de ports limités, il ne peut répondre à trop de requêtes simultanément. Or, `Server` n'est pas capable de rejeter des requêtes et peut donc se retrouver saturé : un processus simple tel que !`Client` constitue bel et bien une attaque de type Déni de Service.

Cette attaque peut se caractériser aisément à partir de la structure des ambients en spécifiant que `Server` dispose d'assez de ressources pour recevoir au plus un nombre r d'ambients de requêtes. Cette notion est un cas particulier de notre conception des ressources puisqu'on peut l'interpréter de la manière suivante :

- l'ambient s dispose d'une réserve de r ressources
- chaque ambient nommé *request* occupe une ressource
- lorsque *request* entre dans s, le serveur alloue des ressources
- si *request* quitte s, par exemple si l'on étend le protocole pour permettre au serveur de transmettre la requête à un autre ordinateur, ces ressources sont désallouées
- enfin, lorsque *request* est ouvert, les ressources sont désallouées de la même manière, et peut-être réallouées immédiatement, en fonction du processus libéré par cette ouverture.

Cette notion se généralise naturellement en une définition des ressources dans le contexte des ambients.

Définition 5 (Ressources – MA)
Dans les Mobile Ambients,

- *chaque ambient dispose d'une réserve de ressources – ces ressources sont toutes identiques*
- *chaque ambient peut occuper zéro, une ou plusieurs ressources dans l'ambient parent*
- *déplacer un ambient désalloue ces ressources de l'ambient dont il part et en alloue le même nombre dans l'ambient où il arrive*
- *ouvrir un ambient désalloue ces ressources puis alloue suffisamment de ressources pour accueillir le contenu de l'ambient dissous.*

Reste à concevoir des systèmes dans lesquels aucune exécution ne peut conduire à une allocation depuis une réserve vide.

Contrôle des ressources

Tel quel, le calcul des Mobile Ambients ne se prête pas naturellement à l'écriture de systèmes ressource-contrôlés. En effet, il n'existe aucune primitive qui puisse permettre de comptabiliser aisément les ressources utilisées à un moment donner et de réagir différemment en fonction de l'état de la réserve. On pourrait imaginer implanter cette fonctionnalité à l'aide de primitives qui permettraient à un ambient de prendre en compte une allocation ou une désallocation – si seulement ces primitives existaient, ce qui n'est pas le cas. En effet, dans les MA un ambient qui reçoit un sous-ambient n'en est a priori pas informé. Ceci constitue notre premier problème.

On suppose que l'ambient a nécessite k ressources et que les ambients m_1 et m_2 nécessitent respectivement e_1 et e_2 ressources.

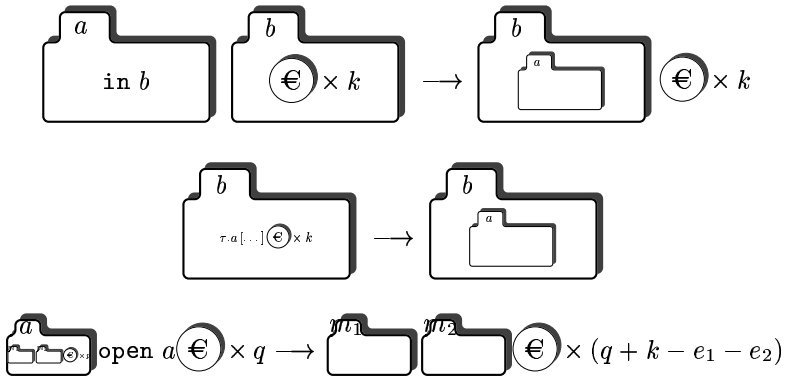


FIG. 3.11 – Allocations/désallocations de ressources dans les Mobile Ambients

Plus grave encore, il n'existe aucune primitive qui permette à un ambient de refuser une allocation. Par suite, afin de contrôler les mouvements, il serait nécessaire d'implanter des protocoles complexes, comparables à des mécanismes de transaction, et de vérifier que ces protocoles sont effectivement respectés.

Or, si l'on considère un serveur, non seulement des opérations de prise en compte d'allocation ou de désallocation existent, mais ces primitives font partie intégrante de la communication. En effet, pour recevoir un message/allouer des ressources à un nouvel ambient, il faut l'avoir explicitement demandé, de même qu'il faut avoir explicitement demandé à émettre et oublier un message/désallouer des ressources d'un ambient. Non seulement le serveur peut naturellement prendre en compte ces opérations, mais il peut aussi tout naturellement choisir de ne pas les entreprendre.

Une autre difficulté pour modéliser les systèmes ressource-contrôlés tient à la définition du constructeur de réplication !. En effet, si P est un processus, $!P$ est équivalent à une infinité de copies de P placées en parallèle. Dès que P alloue une ressource, $!P$ peut donc allouer une infinité de ressources – ce qui est une situation assez rare dans les systèmes, quels qu'ils soient.

3.2 L'approche BoCa

Dans cette section, nous introduisons BoCa [7], un calcul de processus dérivé des Mobile Ambients et dédié à l'analyse et au contrôle des ressources, développé par F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini et V. Sassone. Historiquement, ces résultats ont été publiés après les Controlled Ambients (cf. Section 3.3) et les deux calculs semblent avoir été développés à peu près simultanément. Nous avons préféré présenter BoCa en premier afin de rassembler dans la suite de cette partie les diverses méthodes que nous proposons.

Comme dans les Mobile Ambients, chaque système est constitué d'une hié-

rarchie d'ambients, qui peuvent contenir des processus. Notamment, certains processus particuliers représentent explicitement des ressources, notées $\blacksquare$. De nouvelles capacités et cocapacités permettent de déplacer ces ressources ou de forcer des processus à rester passifs tant qu'il n'y a pas suffisamment de ressources pour les exécuter. Chaque ambient est alors annoté par son encombrement. Les déplacements sont des opérations d'allocation dans le site de destination, qui ne peuvent avoir lieu qu'en présence de suffisamment de ressources, et de désallocation dans le site de départ, qui libèrent donc des ressources lorsqu'elles sont exécutées.

Un système de types complète le langage et permet de garantir statiquement les politiques de gestion des ressources.

Notons que, dans l'introduction de ce chapitre, nous avons utilisé la syntaxe que minimale du langage BoCa, alors que nous allons en présenter une version bien plus riche.

3.2.1 Le langage

La syntaxe de BoCa est présentée sur la figure 3.12. On suppose ici aussi l'existence d'un ensemble strictement dénombrable a, b, c, m, n ... de noms d'ambients, d'un autre ensemble strictement dénombrable x, y, z ... de noms de variables et d'un troisième ensemble $\mathcal{N} \cup \{*\}$ de noms de ressources. On emploiera P, Q ... pour désigner les processus, M, N ... pour les capacités et η, ρ... pour les ressources.

Nous ne nous étendrons pas sur les constructions communes avec le calcul des Mobile Ambients. Le constructeur $\blacksquare_\eta$ représente une ressource de catégorie η. Un ambient M de catégorie η, de poids k et contenant le processus P s'écrit $M^k\,[P]_\eta$. L'opération $k{\triangleright}_\eta P$ ou "spawner" permet d'attendre la présence de k ressources de catégorie η pour lancer P. La cocapacité $\overline{\texttt{opn}}$ est duale de la capacité `open` M et autorise l'ouverture d'un ambient. Enfin, les capacités `get` M et `put` permettent de transférer une ressource entre deux ambients frères, tandis que les capacités $\texttt{get}^\uparrow$ et $\texttt{put}^\downarrow$ permettent de transférer une ressource d'un ambient parent à un ambient enfant. Bien qu'utilisées dans des buts distincts, ces capacités $\texttt{get}^\uparrow$ et $\texttt{put}^\downarrow$ sont comparables aux mécanismes de communication des Boxed Ambients [13].

Comme pour les Mobile Ambients, nous noterons $M^k\,[]_\eta$ pour $M^k\,[\mathbf{0}]_\eta$. De plus, lorsque η n'est pas important, nous noterons $M^k\,[P]$ pour $M^k\,[P]_\eta$. Si k non plus n'est pas important, nous écrirons $M\,[P]$. Enfin, nous noterons $\blacksquare_\eta^k$ pour $\underbrace{\blacksquare_\eta|\ldots|\blacksquare_\eta}_{k}$ et $!^k$ pour $!k{\triangleright}$. Les définitions de variables libres et liées et de la substitution étendent naturellement celles des Mobile Ambients.

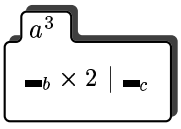
Définition 6 (Poids d'un processus)
Le poids w d'un processus est défini par
- $w(\mathbf{0}) = 0$
- $w(\blacksquare_\eta) = 1$
- $w(P|Q) = w(P) + w(Q)$
- $w(M.P) = w((x : W)P) = w(\langle M\rangle P) = w((\nu n : N)P) = w(P)$
- $w(M^k\,[P]_\eta) = k$ *si* $w(P) = k$
- $w(k \triangleright_\eta P) = 0$ *si* $w(P) = k$
- $w(!P) = 0$ *si* $w(P) = 0$.

Processus			Capacités		
P, Q	$::=$	$\mathbf{0}$	M, N	$::=$	ϵ
	$\mid$	$\blacksquare_\eta$		$\mid$	$M.N$
	$\mid$	$P \mid Q$		$\mid$	$\texttt{in}\ M$
	$\mid$	$!P$		$\mid$	$\texttt{out}\ M$
	$\mid$	$M^k\,[P]_\eta$		$\mid$	$\texttt{opn}\ M$
	$\mid$	$M.P$		$\mid$	$\overline{\texttt{opn}}$
	$\mid$	$(\nu n : N)P$		$\mid$	$\texttt{get}\ M_\eta$
	$\mid$	$k \rhd_\eta P$		$\mid$	$\texttt{get}^{\uparrow} M_\eta$
	$\mid$	$(x : N).P$		$\mid$	$\texttt{put}$
	$\mid$	$\langle M \rangle.P$		$\mid$	$\texttt{put}^{\downarrow}$
				$\mid$	n
				$\mid$	x

FIG. 3.12 – Syntaxe des BoCa.

Dans tous les autres cas, le poids n'est pas défini.

Définition 7 (Terme bien formé)
Un terme P est bien formé si $w(P)$ est défini.

Ainsi, 

ou $a^3\left[\blacksquare_b^2 \mid \blacksquare_c\right]_d$ est un terme bien formé de poids 3. L'ambient a contient trois ressources de catégories respectives b, b et c et occupe lui-même trois ressources de catégorie d.

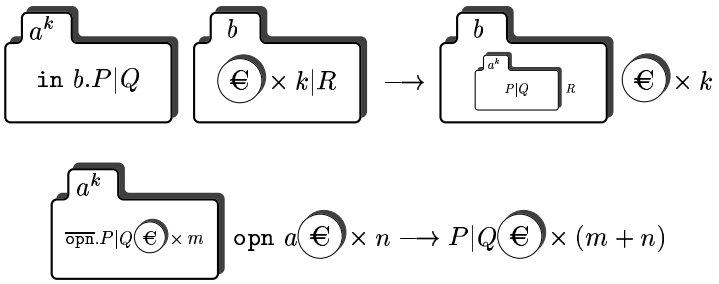


FIG. 3.13 – Quelques ressources.

La congruence structurelle est la plus petite relation $\equiv$ qui soit une relation d'équivalence compatible avec les règles de la figure 3.14, telle que $(P/\equiv, |, \mathbf{0})$ soit un monoïde commutatif et associatif et compatible avec l'α-équivalence des variables liées. Les règles de réduction sont présentées sur la figure 3.15. Ces règles ne sont définies que pour les termes correctement formés. Notons

$$(\nu a)(P|Q) \equiv ((\nu a)P)|Q \qquad (\nu a)\mathbf{0} \equiv \mathbf{0} \qquad !P \equiv !P|P \qquad (\nu a)a^0\,[\mathbf{0}] \equiv \mathbf{0}$$

$$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \qquad a\,[(\nu b)P] \equiv (\nu b)a\,[P] \text{ SI } a \neq b$$

FIG. 3.14 – Congruence structurelle des BoCa

Contextes d'évaluation $\mathbf{E} ::= \{\cdot\} \mid \mathbf{E}|P \mid (\nu m)\mathbf{E} \mid m^k\,[\mathbf{E}]_\rho$

ENTER $a^k\,[\texttt{in}\ b.P \mid Q]_\rho \mid b\,[\blacksquare_\eta^k \mid R] \longrightarrow \blacksquare_\rho^k \mid b\left[a^k\,[P|Q]_\eta \mid R\right] \quad \eta \in \{a, *\}$

EXIT $\blacksquare_\eta^k \mid b\left[P|a^k\,[\texttt{out}\ b.Q|R]_\rho\right] \longrightarrow a^k\,[Q|R]_\eta \mid b\,[P \mid \blacksquare_\rho^k] \quad \eta \in \{a, *\}$

OPEN $\texttt{opn}\ a.P \mid a\,[\overline{\texttt{opn}}.Q \mid R] \longrightarrow P|Q|R$

GETS $a^{k+1}\,[\texttt{put}.P \mid \blacksquare_\eta \mid Q] \mid b^h\,[\texttt{get}\ a_\eta.R \mid S] \longrightarrow a^k\,[P|Q] \mid b^{h+1}\,[R \mid \blacksquare_\eta \mid S]$

GETD $\texttt{put}^\downarrow.P \mid \blacksquare_\eta \mid a^{k+1}\left[\texttt{get}_\eta^\uparrow.Q \mid R\right] \longrightarrow P \mid a^k\,[\blacksquare_\eta \mid Q \mid R]$

SPAWN $k \rhd_\eta P \mid \blacksquare_\eta^k \longrightarrow P \qquad$ EXCHANGE $(x : W).P \mid \langle M\rangle Q \longrightarrow P\{x \leftarrow M\} \mid Q$

$$\text{STRUCT}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \equiv Q} \qquad \text{CONTEXT}\ \frac{P \longrightarrow Q}{\mathbf{E}\{P\} \longrightarrow \mathbf{E}\{Q\}}$$

FIG. 3.15 – Réduction dans BoCa

que quelques règles considérées comme habituelles manquent à cette définition. Ainsi, on n'a ni $!\mathbf{0} \equiv \mathbf{0}$ ni $!(P|Q) \equiv !P|!Q$.

La règle ENTER permet à un ambient a d'entrer dans un ambient b lorsque b dispose de suffisamment de ressources adaptées à a – c'est-à-dire toutes annotées par a ou toutes annotées par $*$. L'ambient a libère alors sur place les k ressources qu'il occupait. Notons que a occupait des ressources annotées par ρ, qui sont ainsi libérées, et occupe maintenant des ressources annotées par η. La règle EXIT définit, de manière symétrique, le comportement d'un ambient a qui cherche à sortir d'un ambient b. L'ouverture d'un ambient a, comme le spécifie OPEN, libère tous les processus contenus dans a – ressources comprises – ainsi que le processus Q mis en attente par $\overline{\texttt{opn}}.Q$. Par GETS, un ambient a qui dispose de $k+1$ ressources peut en communiquer une à un ambient frère b, à condition que cette ressource soit libre et porte le label attendu par b. De même, par GETD, un ambient père peut transférer une ressource à un ambient fils, dans les mêmes conditions – notons qu'il n'existe pas de primitive pour l'opération inverse mais que celle-ci peut être encodée aisément à l'aide d'une ouverture. La règle SPAWN définit le comportement d'un spawner : $k \rhd_\eta P$ attend jusqu'à ce que k ressources de catégorie η soient libres localement, puis réserve ces ressources pour P et

exécute P. Enfin, les règles EXCHANGE, STRUCT et CONTEXT permettent de généraliser ce qui précède.

3.2.2 Propriétés

Lemme 1 (Préservation des ressources)
Si P est bien formé et si $P \longrightarrow^* Q$, alors $w(P) = w(Q)$.

En d'autres termes, la taille d'une réserve de ressources (en comptant aussi bien les ressources occupées que les ressources libres) reste constante au fil du temps. Notons que ce lemme ne prouve rien sur le fait qu'un processus est ressource-contrôlé ou non.

3.2.3 Exemples

Verrous

Il est possible d'écrire des processus `Acq` et `Rel` quasiment identiques à ceux des Mobile Ambients.

$$\begin{array}{lcl} \texttt{Acq}\ n.P & \triangleq & \texttt{opn}\ n.P \\ \texttt{Rel}\ n.Q & \triangleq & n^0\,[\overline{\texttt{opn}}] \mid Q \\ \texttt{Rel}\ (M,n).Q & \triangleq & n^0\,[M.\overline{\texttt{opn}}] \mid Q \end{array}$$

De la même manière que dans les MA, on s'assurera au préalable que le nom n n'est connu que par les processus autorisés à acquérir le verrou – et à le relâcher.

On pourrait aussi concevoir d'écrire les verrous en tirant avantage de structures synchrones qu'offre déjà BoCa :

$$\begin{array}{lcl} \texttt{Acq}\ \eta.P & \triangleq & 1 \rhd_\eta .P \\ \texttt{Rel}\ \eta.Q & \triangleq & \blacksquare_\eta \mid Q \end{array}$$

Cette implantation ne permet pas de libérer simplement un verrou à distance. Par contre, notons que, si l'ambient parent est défini de manière à ne contenir qu'un seul $\blacksquare_\eta$, un terme qui libère un verrou ne sera bien formé que s'il l'a précédemment acquis. Par conséquent, le protocole d'acquisition/libération des verrous pourrait être vérifié statiquement et automatiquement.

Malheureusement, dans BoCa, il n'est pas possible de spécifier simplement une propriété telle que "l'ambient a peut contenir une seule ressource de catégorie η et trois ressources de catégories η'", ce qui limite l'intérêt de l'implantation. Ces politiques sont plus proches du contrôle statique offert par les Controlled Ambients (cf. section 3.3) ou des systèmes de types riches (cf. partie 9).

Synchronisation sur un canal

La synchronisation sur un canal est beaucoup plus simple à écrire en BoCa que dans les MA, car BoCa offre une opération synchrone (en plus de la communication) : l'ouverture d'un sous-ambient.

$$\begin{array}{lcl} \texttt{Up}\ n.P & = & \texttt{opn}\ n.P \\ \texttt{Down}\ n.Q & = & n^k\,[\overline{\texttt{opn}}.Q] \ \text{où}\ w(Q) = k \end{array}$$

De manière surprenante, cette implantation de la synchronisation ne peut être considérée comme une macro simple, car elle nécessite de prendre en compte le poids de Q. De plus, comme nous l'avions déjà fait remarquer dans le cadre des Mobile Ambients, si l'on considère une implantation distribuée du formalisme, ouvrir un ambient pour relâcher un processus peut être une opération coûteuse et risquée.

Nous préférerons donc une version un peu plus complexe :

$$\begin{array}{lcl} \texttt{Up}\ n.P & = & \texttt{opn}\ n.P \\ \texttt{Down}\ n.Q & = & (\nu l)(n^0\ [l^0\ []]\ \mid \texttt{opn}\ l.Q) \end{array}$$

Communication sur un canal

Comme la synchronisation sur un canal, la communication sur un canal est bien plus simple que dans les Mobile Ambients.

$$\begin{array}{lcl} \texttt{Send}\ M\ \texttt{on}\ n.P & = & n^k\ [\blacksquare_r^k\ \mid \texttt{opn}\ r.\langle M\rangle]\ \mid \texttt{opn}\ n.P \\ \texttt{Receive}\ x\ \texttt{on}\ n.Q & = & r^k\ [\texttt{in}\ n.\overline{\texttt{opn}}.(x).\overline{\texttt{opn}}.Q]_r \\ w(Q) & = & k \end{array}$$

L'ambient n, dont le nom doit rester privé, permet encore d'isoler l'émission et la réception, pour éviter certaines interférences. L'ambient r, dont le nom est considéré public mais réservé à ce protocole, permet d'emmener Q à sa destination. Grâce au traitement des ressources, seul un ambient r peut pénétrer n, ce qui évite de nombreuses interférences possibles.

Ici aussi, il est nécessaire d'examiner le processus Q pour déterminer la valeur de k. Cette tâche est possible car le poids de Q est indépendant de x. Dans un calcul plus riche, par exemple s'il était possible de tester l'égalité entre deux noms, le comportement de Q pourrait dépendre plus de la valeur de x et rendre la macro plus fragile – ou nécessiter l'utilisation d'un système de types.

Serveur

Le serveur présenté à la figure 3.9 s'étend naturellement lorsqu'on lui ajoute une gestion des ressources :

$$\begin{array}{lcl} \texttt{Server} & = & s[\blacksquare\ \mid \\ & & \quad !\texttt{opn}\ request.(M).1 \rhd_{reply} reply^1\ [\texttt{out}\ s.M\ \mid \blacksquare_{reply}]_{request} \\ & &] \\ \texttt{Client}\ c & = & c^1\ [request^1\ [\blacksquare_{reply}\ \mid \texttt{out}\ c.\texttt{in}\ s.\overline{\texttt{opn}}.\langle \texttt{in}\ c\rangle]] \end{array}$$

Notons que le mécanisme d'allocation des ressources de ces termes est surprenant. Ainsi, il semble que ce soit le client qui fournisse la ressource $\blacksquare_{reply}$ qui sera plus loin allouée à l'intérieur du serveur pour répondre à la requête. En fait, cette ressource est récupérée par le client dès que la requête est émise. Il n'y a donc pas eu d'échange implicite de ressources distribuées.

Notons aussi que, si ce protocole client-serveur est conscient des ressources, il ne résisterait pas à des attaques de la même famille que $\texttt{Rogue}_2$, $\texttt{Rogue}_3$, $\texttt{Rogue}_4$. Cependant, en présence de $\texttt{Rogue}_2$, l'apparition d'un deuxième serveur s modifierait le poids total du terme et pourrait peut-être être observée.

$$
\begin{array}{lcl}
\texttt{Call}\ f\ c & = & \mathit{call}^1\,[\texttt{out}\ c.\texttt{out}\ f.\texttt{in}\ \mathit{cab}.\overline{\texttt{opn}}.\texttt{in}\ f.\blacksquare_c]_* \\
\texttt{Trip}\ f\ t\ c\ b & = & \mathit{trip}^0\,[\texttt{out}\ c.\overline{\texttt{opn}}.\texttt{out}\ f.\texttt{in}\ t.u^0\,[\texttt{in}\ c.\overline{\texttt{opn}}]\ \mid 1 \rhd_c \blacksquare_b] \\
\texttt{Bye}\ b\ c & = & 1 \rhd_* b^1\,[\texttt{out}\ c.\texttt{in}\ \mathit{cab}.\overline{\texttt{opn}}.\texttt{out}\ \mathit{to}.\blacksquare_{\mathit{call}}]_b \\
\texttt{Client}\ f\ t & = & (\nu c, b)c^1[\texttt{Call}\ f\ c \mid \\
 & & \quad \texttt{in}\ \mathit{cab}.\texttt{Trip}\ f\ t\ c\ b \mid \\
 & & \quad \texttt{opn}\ u.\texttt{out}\ \mathit{cab}.\texttt{Bye}\ b\ c \\
 & &] \\
\texttt{Cab} & = & \mathit{cab}^1\,[\blacksquare_{\mathit{call}} \mid !\texttt{opn}\ \mathit{call}.\texttt{opn}\ \mathit{trip}.\texttt{opn}\ \mathit{bye}]_*
\end{array}
$$

FIG. 3.16 – Le protocole du taxi – version BoCa.

Le protocole du taxi

Une implantation du protocole du taxi est présentée sur la figure 3.16. Si la structure est globalement similaire à celle que nous avions présentée à la section 3.1.2, deux différences majeures entrent en jeu : le mécanisme des $\overline{\texttt{opn}}$ et celui des ressources.

Si les autorisations d'ouverture sont relativement peu utilisées dans l'implantation, elles permettent pour le moins d'assurer qu'aucun processus malicieux ne pourra dissoudre un taxi ou un client.

Le contrôle des ressources, quant à lui, permet d'assurer un certain respect du protocole. Ainsi, *cab* contient initialement $\blacksquare_{\mathit{call}}$ et ne peut donc recevoir qu'un ambient nommé *call*. Après l'ouverture de *call*, cette ressource est remplacée par $\blacksquare_c$, ce qui permet d'assurer que le bon client pourra entrer dans le taxi. Puis, au cours de l'exécution de `Trip`, le processus $1 \rhd_c \blacksquare_b$ est installé dans *cab*. Lorsque le client quitte *cab*, il libère une ressource $\blacksquare_c$, qui est ainsi consommée et remplacée par une ressource $\blacksquare_b$, à même de recevoir l'ambient b. Cette opération est importante car le nom b est initialement privé. En particulier, même si cette propriété n'est pas fondamentale dans cette implantation, la présence de $\blacksquare_b$ signifie que l'ambient b entrera dans le bon taxi. Un client ne peut pas chasser un taxi qui ne l'a pas amené.

3.2.4 Système de types

Si BoCa permet de représenter les ressources directement et explicitement au niveau du langage, et si la sémantique des opérations interdit les allocations depuis des réserves vides, il reste nécessaire de recourir à des outils d'analyse pour garantir que l'utilisation des ressources entre dans une politique de contrôle donnée. C'est ce pour quoi est conçu le système de types que nous présentons dans cette section.

Notons que nous avons modifié la syntaxe et la présentation de ce système de types afin de l'harmoniser avec celle des autres travaux introduits dans cet exposé.

La figure 3.17 présente la grammaire de types pour le contrôle des ressources. Le type d'un processus P est $\mathit{Proc}(d, i)[T]$ si P peut donner jusqu'à d ressources et recevoir jusqu'à i ressources suite à des échanges `put`/`get` et si toutes les com-

$$
\begin{array}{lcll}
W & ::= & Amb(u,l,d,i)[T] & (u,l,d,i) \in \mathbf{N}^4, u \leq l \\
 & | & Cap(\phi)[T] & \phi : \mathbf{N}^2 \longrightarrow \mathbf{N}^2 \\
U & ::= & Proc(d,i)[T] & (d,i) \in \mathbf{N}^4 \\
T & ::= & Ssh & \\
 & | & W &
\end{array}
$$

FIG. 3.17 – Grammaire de types pour le contrôle des ressources – BoCa.

munications dans P sont de type T. Un ambient a est de type $Amb(u,l,d,i)[T]$ si son poids est compris entre u et l et s'il peut contenir un processus de type $Proc(d,i)[T]$. Une capacité M est de type $Cap(\phi)[T]$ si, lorsqu'elle est exercée, elle peut libérer un processus dont les communications sont de type T et qui transforme les échanges de ressources selon la fonction ϕ.

Un environnement de typage Γ est une fonction partielle qui, à chaque nom de capacité pour lequel elle est définie associe un type de capacité. Nous noterons $\Gamma, n : W$ la fonction qui au nom n associe le type W et pour tout autre nom réagit comme Γ.

Dans la suite de cette présentation, nous utiliserons les fonctions Put, Get et $Open$ définies par

$$
\left\{
\begin{array}{lcl}
\forall d,i, Put(d,i) & = & d+1, max(0,i-1) \\
\forall d,i, Get(d,i) & = & max(0,d-1), i+1 \\
\forall d,i, Id(d,i) & = & d,i \\
\forall \epsilon, \forall d,i, Open(\epsilon)(d,i) & = & \epsilon + (d,i)\ .
\end{array}
\right.
$$

Les règles de typage sont sont présentées sur la figure 3.18. Sans entrer dans le détail, notons que ce système de types s'intéresse aux déplacements de ressources par `put`/`get` et aux ouvertures. À l'issue d'un typage, cependant, nous intéressés essentiellement par le type des ambients, car un ambient de type $Amb(l,u,\_,\_)[\_]$ utilisera en permanence entre l et u ressources.

On peut constater dans ce système que les effets de deux processus composés en parallèle s'additionnent, tandis que les effets de deux capacités composées en série se composent. On peut aussi remarquer que les processus $\mathbf{0}$ et $\_\_$ sont considérés comme sans effet. En fait, seules les capacités `get`, $\texttt{get}^{\uparrow}$, `put` et `open` ont un effet et seuls les processus $M.P$ – où M est l'une de ces capacités – et $M^k\,[P]$ sont considérés comme ayant un effet.

Théorème 1 (Les bons types respectent les limitations) *Si Γ est un environnement, P un processus et a un ambient, si $\Gamma \vdash P : Proc(\_)[\_]$ et si $\Gamma(a) = Amb(l,u,\_,\_)[\_]$, alors tout sous-processus de P de la forme $a^k\,[Q]$ où a n'est pas sous un lieur de a vérifie $l \leq k \leq u$.*

Notons que ce théorème n'est pas directement lié à notre notion de processus ressource-contrôlés, car la sémantique du langage interdit déjà aux processus d'allouer des ressources depuis des réserves vides. Cependant, à l'aide d'environnements Γ bien choisis, on peut examiner le nombre de ressources dont a besoin un ambient et borner le nombre de ressources dont il peut disposer au fil des transferts.

$$\text{T-Name}\ \Gamma, M : W \vdash M : W \qquad \text{T-Get}\ \frac{\Gamma \vdash M : Amb(\_)[\_]}{\Gamma \vdash \texttt{get}\ M : Cap(Get)[T]}$$

$$\text{T-GetUp}\ \Gamma \vdash \texttt{get}^{\uparrow} M : Cap(Get)[T] \qquad \text{T-Put}\ \Gamma \vdash \texttt{put} M : Cap(Put)[T]$$

$$\text{T-PutDown}\ \Gamma \vdash \texttt{put}^{\downarrow} M : Cap(Id)[T] \qquad \text{T-In}\ \frac{\Gamma \vdash M : Amb(\_)[\_]}{\Gamma \vdash \texttt{in}\ M : Cap(Id)[T]}$$

$$\text{T-Out}\ \frac{\Gamma \vdash M : Amb(\_)[\_]}{\Gamma \vdash \texttt{out}\ M : Cap(Id)[T]} \qquad \text{T-Open}\ \frac{\Gamma \vdash M : Amb(\_,\_,d,i)[T]}{\Gamma \vdash \texttt{opn}\ M : Cap(Open(d,i))[T]}$$

$$\text{T-CoOpen}\ \Gamma \vdash \texttt{opn}\ M : Cap(Id)[T]$$

$$\text{T-Path}\ \frac{\Gamma \vdash M : Cap(\phi_M)[T] \qquad \Gamma \vdash N : Cap(\phi_N)[T]}{\Gamma \vdash Cap(\phi_M \circ \phi_N)[T]}$$

$$\text{T-Slot}\ \Gamma \vdash \_ : Proc(0,0)[T] \qquad \text{T-Nil}\ \Gamma \vdash \mathbf{0} : Proc(0,0)[T]$$

$$\text{T-Prefix}\ \frac{\Gamma \vdash M : Cap(\phi)[T] \qquad \Gamma \vdash P : Proc(d,i)[T]}{\Gamma \vdash M.P : Proc(\phi(d,i))[T]}$$

$$\text{T-Par}\ \frac{\Gamma \vdash P : Proc(d_P, i_P)[T] \qquad \Gamma \vdash Q : Proc(d_Q, i_Q)[T]}{\Gamma \vdash P|Q : Proc(d_P + d_Q, i_P + i_Q)[T]}$$

$$\text{T-Snd}\ \frac{\Gamma, x : W \vdash P : U}{\Gamma \vdash (x : W)P : U} \qquad \text{T-Rcv}\ \frac{\Gamma \vdash M : W \qquad \Gamma \vdash P : U \qquad U = Proc(d,i)[W]}{\Gamma \vdash \langle M \rangle : U}$$

$$\text{T-Res}\ \frac{\Gamma, a : A \vdash P : U}{\Gamma \vdash (\nu a : A)P : U}$$

$$\text{T-Amb}\ \frac{\begin{array}{c}\Gamma \vdash M : Amb(l,u,d_M,i_M)[T] \\ \Gamma \vdash P : Proc(d_P,i_P)[T] \qquad w(P) = k \qquad d_P + i_P \geq d_M \\ min(u-l, i_P + d_P) \leq i_M \qquad max(k-d,0) \geq l \qquad k + i \leq u\end{array}}{\Gamma \vdash M^k[P]\_ : Proc(0,0)[T']}$$

$$\text{T-Spawn}\ \frac{\Gamma \vdash P : Proc(0,0)[T]}{\Gamma \vdash k \rhd\_ P : Proc(0,0)[T]}\ w(P) = k$$

$$\text{T-Bang}\ \frac{\Gamma \vdash P : Proc(0,0)[T]}{\Gamma \vdash !P : Proc(0,0)[T]}\ w(P) = 0$$

FIG. 3.18 – Règles de typage de BoCa.

3.2.5 Discussion

BoCa est un calcul d'ambients conçu autour d'un concept de ressources représentées explicitement. Dans ce formalisme, chaque ambient dispose d'une réserve de ressources, qui peuvent être allouées par les opérations $\triangleright$ et $\texttt{put}/\texttt{put}^{\downarrow}$ ou, par effets de bord, par les mouvements d'ambients et qui peuvent être désallouées par les opérations `opn`, $\texttt{get}/\texttt{get}^{\uparrow}$ ou, de nouveau par effets de bord, par des mouvements d'ambients.

À l'aide de BoCa, il est possible de représenter naturellement, au niveau du langage, des systèmes dans lesquels les ressources sont prises en compte, au sens où il n'est possible d'allouer des ressources que tant que le réserves ne sont pas vides. Un système de types permet de compléter le formalisme en garantissant des bornes sur la taille de la réserve des ambients visibles.

La construction $\triangleright$ mérite d'être soulignée car elle introduit un mécanisme intéressant de réservation des ressources. Au cours de cet exposé, nous verrons à plusieurs reprises des constructions qui réservent ainsi des ressources pour d'autres processus. De même, le mécanisme de ressources nommées, qui permet d'arbitrer les conflits de ressources, s'il est moins puissant que les cocapacités que nous présenterons dans la section suivante, est bien plus esthétique et plus agréable à lire et à utiliser.

Notons cependant que les processus écrits à l'aide de BoCa ne sont pas réellement conscients des ressources. En effet, si le calcul lui-même se charge de la comptabilité et de l'arbitration des ressources, un processus n'a pas de réel moyen de réagir à une allocation, par exemple pour libérer les ressources nécessaires pour réagir à un cas d'erreur, comme nous l'avons vu en `C`.

De plus, si un terme n'essayera jamais d'allouer des ressources depuis une réserve vide, cela s'exprimera surtout par une attente – éventuellement éternelle – de la part du processus en question. Ce comportement, courant dans le cadre de la gestion des certaines ressources, n'est cependant pas valide dans d'autres cas, par exemple lorsqu'il s'agit d'allocations de mémoire, de processus légers ou d'ouvertures de fichiers sous Un*x. En particulier, cette propriété ne vérifie qu'imparfaitement notre objectif de processus ressource-contrôlés.

Notons enfin que, s'il semble un peu trop exigeant de demander à un programmer de faire figurer explicitement toutes les ressources de tous les processus qu'il écrit, cette contrainte peut être assimilée à la publication et au retrait (dynamique) de méthodes et d'interfaces dans des langages à base de composants.

3.3 Controlled Ambients

Dans cette section, nous introduisons les Controlled Ambients [80] (CA), un calcul de processus conçu pour permettre de développer des protocoles ressource-contrôlés dans le cadre des ambients. Pour obtenir les CA, nous avons étendu les principes des Safe Ambients (SA), eux-mêmes dérivés des Mobile Ambients et développés par Davide Sangiorgi et Francesca Levi [58].

Les Safe Ambients ont été créés en réaction aux interférences qui apparaissent souvent lorsqu'on cherche à écrire un système ou un protocole dans les Mobile Ambients, comme nous l'avons vu avec le protocole du taxi. Le calcul des SA, qui permet d'écrire des termes moins "fragiles" que dans les MA, a notamment été utilisé pour étudier des questions d'équivalence comportementale.

Comme dans les MA, chaque système est constitué d'une hiérarchie d'ambients, qui peuvent contenir des processus. De nouvelles cocapacités permettent d'autoriser les déplacements et les dissolutions – qui sont a priori interdits en l'absence de ces cocapacités – et de prendre en compte ces opérations.

À l'inverse de l'approche BoCa, nous abordons le problème du contrôle des ressources sans représenter les ressources elles-mêmes dans le langage – celles-ci n'apparaissent que dans le système de types. Le type d'un ambient spécifie alors le nombre de ressources qu'il occupe dans l'ambient parent (son encombrement) et le nombre de ressources dont il dispose pour ses sous-ambients (sa réserve, locale à l'ambient). Ce système de types permet de garantir qu'un terme est ressource-contrôlé.

3.3.1 Le langage

La syntaxe des Controlled Ambients est présentée sur la figure 3.19. On suppose encore l'existence d'un ensemble strictement dénombrable a, b, c, m, $n\ldots$ de noms d'ambients et un autre ensemble strictement dénombrable x, y, $z\ldots$ de noms de variables. On emploiera P, $Q\ldots$ pour itérer sur les noms de processus, M, $N\ldots$ pour itérer sur les noms de capacités.

Nous ne nous étendrons pas sur les constructions communes avec le calcul des Mobile Ambients. La primitive $\texttt{rec}\ X.P$ permet la récursion et remplace la réplication, difficile à contrôler.

Les cocapacités $\overline{\texttt{out}}^{\downarrow}\ m$ et $\overline{\texttt{out}}^{\uparrow}\ m$ autorisent l'ambient m à quitter son ambient parent. L'ambient a contiendra $\overline{\texttt{out}}^{\downarrow}\ m$ pour permettre à m de s'enfoncer dans un ambient frère n à l'aide de $\texttt{in}\ n$ ou $\overline{\texttt{out}}^{\uparrow}\ m$ pour permettre à m de quitter a avec $\texttt{out}\ a$. De même, les cocapacités $\overline{\texttt{in}}^{\downarrow}\ m$ et $\overline{\texttt{in}}^{\uparrow}\ m$ présentes dans l'ambient n autorisent l'ambient m à entrer dans n, respectivement en venant de son ambient parent (avec un $\texttt{in}$) ou d'un ambient fils (avec un $\texttt{out}$). Un déplacement, comme illustré sur la figure 3.21, fera donc intervenir un triplet $\texttt{in}\ a$, $\overline{\texttt{out}}^{\downarrow}\ m$, $\overline{\texttt{in}}^{\downarrow}\ m$ ou un triplet $\texttt{out}\ n$, $\overline{\texttt{out}}^{\uparrow}\ m$, $\overline{\texttt{in}}^{\uparrow}\ m$. Enfin, la capacité $\overline{\texttt{open}}\ m$ présente dans l'ambient m autorise celui-ci à être ouvert.

Les définitions de variables libres et liées et de substitution des Mobile Ambients s'étendent naturellement – nous considérerons que la variable X, bien que de nature différente des variables x est liée dans $\texttt{rec}\ X.P$. De plus, nous emploierons les mêmes raccourcis syntaxiques que dans les Mobile Ambients.

Les règles de congruence structurelle sont presque les mêmes que celles des Mobile Ambients : les règles sur la réplication disparaissent car, dans les Controlled Ambients, elles n'ont pas de raison d'être, tandis que le préfixage par $\texttt{rec}\ X$ préserve la congruence.

Les règles de réduction apparaissent sur la figure 3.20. Contrairement aux Mobile Ambients, et à l'image des Safe Ambients, les déplacements des CA nécessitent des autorisations sous la forme de cocapacités. Là où les Safe Ambients introduisent les cocapacités $\overline{\texttt{in}}\ m$ et $\overline{\texttt{out}}\ m$, qui autorisent respectivement à entrer dans m et à sortir de m, nous imposons des synchronisations entre trois processus dans les règles R-In et et R-Out : respectivement $\texttt{in}\ b.P$, $\overline{\texttt{in}}^{\downarrow}\ a.R$ et $\overline{\texttt{out}}^{\downarrow}\ a.T$ pour R-In et $\texttt{out}\ a.P$, $\overline{\texttt{out}}^{\uparrow}\ b.R$ et $\overline{\texttt{in}}^{\uparrow}\ b.T$ pour R-Out.

Processus			Capacités		
P, Q	$::=$	$\mathbf{0}$	M, N	$::=$	ϵ
	$\mid$	$P \mid Q$		$\mid$	$M.N$
	$\mid$	$\texttt{rec}\ X.P$		$\mid$	$\texttt{in}\ M$
	$\mid$	$M\,[P]$		$\mid$	$\texttt{out}\ M$
	$\mid$	$M.P$		$\mid$	$\texttt{open}\ M$
	$\mid$	$(\nu n : A)P$		$\mid$	$\overline{\texttt{in}}^{\uparrow}\ M$
	$\mid$	$(x : A).P$		$\mid$	$\overline{\texttt{in}}^{\downarrow}\ M$
	$\mid$	$\langle M \rangle$		$\mid$	$\overline{\texttt{out}}^{\uparrow}\ M$
	$\mid$	X		$\mid$	$\overline{\texttt{out}}^{\downarrow}\ M$
				$\mid$	$\overline{\texttt{open}}\ M$
				$\mid$	n
				$\mid$	x

FIG. 3.19 – Syntaxe des Controlled Ambients.

3.3.2 Discussion

Création d'ambients Comme dans les Mobile Ambients et bien que ces deux situations soient intuitivement très différentes, nous ne différencions pas l'opération qui consiste à créer un nouveau site et donc à lui allouer les ressources nécessaires du marqueur qui spécifie qu'un ambient est présent et occupe déjà un certain nombre de ressources. Comme nous le verrons, la construction $a\,[P]$, qui représente aussi bien les deux cas, nous a suffi pour définir des protocoles ressource-contrôlés. Dans certains autres calculs, cependant, nous serons contraints d'employer deux constructions syntaxiques différentes afin, notamment, de pouvoir typer la création des localités et résoudre des problèmes d'identification (à ce sujet, voir la section 7.3).

Implantation Les Controlled Ambients font usage d'une synchronisation à deux parties (pour l'ouverture) ou à trois parties (pour l'entrée et la sortie d'ambients), qui s'avère réaliste du point de vue de l'implantation. Considérons, en effet, la transition suivante :

$$h\,[m\,[\texttt{in}\ n] \mid n\,[]] \longrightarrow h\,[n\,[m\,[]]] \text{ (dans les MA).}$$

Comme le suggèrent des études sur l'implantation des MA [32, 31, 73], la structure qui représente h doit être au courant de la présence de n à l'intérieur de h. Ce qui signifie, si n n'a pas été créé directement dans h, que h a du être prévenu d'une manière ou d'une autre. Plus généralement, l'exécution de ce mouvement nécessite une synchronisation entre n (qui va recevoir un nouvel élément et doit être mis au courant), m (qui va bouger et "cherche" n) et h (qui va mettre m et n en contact). De même, l'ouverture d'un ambient m par un ambient h requiert une forme de synchronisation complexe entre m et h afin de transférer tous les processus et tous les sous-ambients de m vers h et de mettre à jour des registres de présence – ou toute autre structure utilisée pour exprimer la hiérarchie des sites.

$$\text{R-In}\ a\,[\mathtt{in}\ b.P \mid Q] \mid b\left[\overline{\mathtt{in}}^{\downarrow}\ a.R \mid S\right] \mid \overline{\mathtt{out}}^{\downarrow}\ a.T \longrightarrow b\,[a\,[P \mid Q] \mid R \mid S]T$$

$$\text{R-Out}\ a\left[b\,[\mathtt{out}\ a.P \mid Q] \mid \overline{\mathtt{out}}^{\uparrow}\ b.R \mid S\right] \mid \overline{\mathtt{in}}^{\uparrow}\ b.T \longrightarrow b\,[P|Q] \mid a\,[R|S] \mid T$$

$$\text{R-Open}\ \mathtt{open}\ a.P \mid a\,[\overline{\mathtt{open}}\ a.Q \mid R] \longrightarrow P|Q|R$$

$$\text{R-Rec}\ \mathtt{rec}\ X.P \longrightarrow P\{X \leftarrow \mathtt{rec}\ X.P\}$$

$$\text{R-Comm}\ (x:W).P \mid \langle M\rangle \longrightarrow P\{x \leftarrow M\} \qquad \text{R-Res}\ \frac{P \longrightarrow Q}{(\nu n)P \longrightarrow (\nu n)Q}$$

$$\text{R-Amb}\ \frac{P \longrightarrow Q}{a\,[P] \longrightarrow a\,[Q]} \qquad \text{R-Par}\ \frac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$$

$$\text{R-Struct}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \equiv Q}$$

FIG. 3.20 – Réduction dans les CA

Cohérence Un autre intérêt des Controlled Ambients est lié à certains des problèmes que nous avons présentés autour du protocole du taxi. Notamment, afin d'assurer qu'un ambient fils ne quitte pas son ambient père trop tôt ou qu'un ambient n'en reçoit pas un autre à un mauvais moment, il peut être nécessaire d'implanter de nombreux mini-protocoles d'acquiescements, parfois complexes et difficiles à vérifier. Or, certains ces protocoles correspondent à des comportements qui, dans les langages de programmation et sur les machines actuels, sont bien plus primitifs que, par exemple, les mécanismes pour la mobilité elle-même.

Ainsi, nous avons utilisé `open` pour ouvrir un ambient et exécuter les instructions qu'il contient, ce qui peut modéliser l'ouverture d'un programme (i.e. son exécution – à ce sujet, voir par exemple les processus `Boom` et `Boom.exe` présentés dans la section 3.3.6). Cette opération nécessite le respect d'un certain

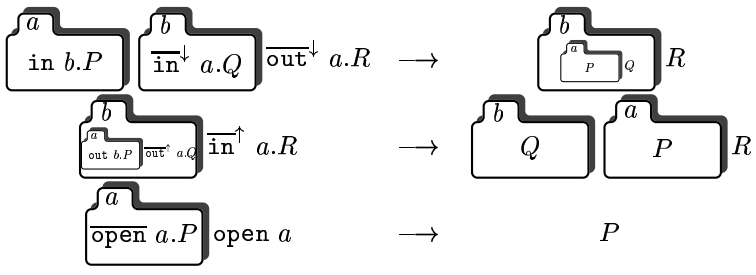


FIG. 3.21 – Quelques réductions dans les Controlled Ambients

protocole entre le système d'exploitation et le programme. En particulier, un programme ne peut pas être en train de s'exécuter avant d'avoir été effectivement chargé. Or, dans les MA, par défaut, le contenu d'un ambient s'exécute – et continue à s'exécuter, sans remarquer que le programme vient d'être ouvert. La cocapacité $\overline{\texttt{open}}\ h$ des Safe Ambients ou des Controlled Ambients permet d'introduire simplement un point d'entrée pour le programme.

De même, par défaut, dans les MA, un ambient a peut pénétrer dans un ambient b à l'aide d'un simple $\texttt{in}\ b$. Les processus présents dans b continuent à s'exécuter sans remarquer de différence. Or, typiquement, `in` peut représenter l'arrivée d'un message sur un ordinateur, en provenance du réseau – et cette action n'arrive *a priori* pas si l'ordinateur n'est pas en train d' «écouter» sur un port. La cocapacité $\overline{\texttt{in}}^{\downarrow}\ a$ représente alors le fait que le port a est ouvert et que l'ordinateur attend un message sur le port a. De même, il serait inconcevable que l'ordinateur ne soit pas au courant qu'il est en train d'émettre sur le réseau, ce que représente la cocapacité $\overline{\texttt{out}}^{\uparrow}\ a$. Dans ce contexte, les cocapacités $\overline{\texttt{in}}^{\uparrow}\ a$ et $\overline{\texttt{out}}^{\downarrow}\ a$, quant à elles, modélisent respectivement la réception et l'émission depuis/vers un sous-système, des opérations qui, elles aussi, ne se déroulent a priori pas sans avoir été explicitement demandées.

Cette manière de gérer la mobilité est intéressante en soi car elle introduit un nouveau point de vue sur les ambients : en plus de représenter des localités, les ambients représentent aussi des ports d'entrée/sortie, tandis que les cocapacités représentent les droits (dynamiques) d'accès aux ports. Cette conception de la chose rapproche le calcul des Controlled Ambients de celui des Seals, présentée à la section 4.1, ou des Nba [13]. On retrouve aussi des idées similaires, quoique poussées plus loin, dans le calcul des Kells, présenté à la section 4.2.

Ressources Enfin, du point de vue des ressources, les cocapacités permettent une réactivité dont ne disposaient pas les Mobile Ambients. En effet, lors d'un déplacement, $\overline{\texttt{in}}^{\uparrow}$ et $\overline{\texttt{in}}^{\downarrow}$ permettent de réagir aux allocations, tandis que $\overline{\texttt{out}}^{\uparrow}$ et $\overline{\texttt{out}}^{\downarrow}$ permettent de réagir aux désallocations. En d'autres termes, pour éviter de tomber à court de ressources, il suffit d'attendre que des ressources aient été libérées (par un $\overline{\texttt{out}}^{\uparrow}$, un $\overline{\texttt{out}}^{\downarrow}$ ou un `open`, par exemple) avant d'autoriser l'allocation de nouvelles ressources (par un $\overline{\texttt{in}}^{\uparrow}$, un $\overline{\texttt{in}}^{\downarrow}$ ou un `open` bien choisi) – nous illustrons cette utilisation des synchronisations dans les exemples de la section 3.3.3.

La cocapacité $\overline{\texttt{open}}$, elle, en plus d'éviter qu'un ambient soit ouvert alors qu'il devrait rester fermé, permet, par exemple, d'attendre qu'un calcul soit terminé et que les ressources qui avaient été acquises soient relâchées avant de rendre le résultat.

Revers de la médaille Malheureusement, cette structure de cocapacités multiples a plusieurs inconvénients. Le plus visible est une certaine lourdeur dans les termes écrits, car il est nécessaire de fournir tous les $\overline{\texttt{in}}$, tous les $\overline{\texttt{out}}$ et tous les $\overline{\texttt{open}}$, même dans les cas triviaux où tous les mouvements sont admis. Il est à noter que cette contrainte se trouve allégée dans les calculs des Nba et des Seals.

La deuxième complication est liée au fait qu'il est plus difficile d'écrire la sémantique étiquetée d'un système lorsque celui-ci fait appel à des transitions à trois parties. C'est un problème auquel ont été confrontés les auteurs des Seals

dans les premières versions du calcul, qui employaient un mécanisme de synchronisation à trois entre l'émetteur, le récepteur et le *portail* [20]. Ce problème, à son tour, complique toute tentative d'étude comportementale des CA.

3.3.3 Exemples

Verrous

Les verrous s'implantent presque comme dans BoCa.

$$\begin{array}{lcl} \texttt{Acq}\ n.P & \triangleq & \texttt{open}\ n.P \\ \texttt{Rel}\ n.Q & \triangleq & n\,[\overline{\texttt{open}}\ n]\ |\ Q \\ \texttt{Rel}\ (M,n).Q & \triangleq & n\,[M.\overline{\texttt{open}}\ n]\ |\ Q \end{array}$$

Synchronisation sur un canal

La synchronisation est presque identique à celle de BoCa, et donc beaucoup plus simple que dans les MA – la différence principale par rapport à BoCa étant qu'il n'est pas nécessaire d'examiner Q pour écrire la synchronisation.

$$\begin{array}{lcl} \texttt{Up}\ n.P & = & \texttt{open}\ n.P \\ \texttt{Down}\ n.Q & = & n\,[\overline{\texttt{open}}\ n.Q] \end{array}$$

Ici aussi, pour éviter d'avoir à ouvrir un ambient pour récupérer son contenu, nous utiliserons plutôt la variante suivante, un peu plus complexe :

$$\begin{array}{lcl} \texttt{Up}\ n.P & = & \texttt{open}\ n.P \\ \texttt{Down}\ n.Q & = & (\nu l)n\,[\overline{\texttt{open}}\ n.l\,[\overline{\texttt{open}}\ l]]\ |\ \texttt{open}\ l.Q \end{array}$$

Communication sur un canal

La communication sur un canal peut s'implanter de la même manière que dans BoCa.

$$\begin{array}{lcl} \texttt{Send}\ M\ \texttt{on}\ n.P & = & n\left[\overline{\texttt{in}}^{\downarrow}\ r.\texttt{open}\ r.\langle M\rangle\right]\ |\ \texttt{open}\ n.P \\ \texttt{Receive}\ x\ \texttt{on}\ n.Q & = & r\,[\texttt{in}\ n.\overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q]\ |\ \overline{\texttt{out}}^{\downarrow}\ r \end{array}$$

Là où, dans BoCa, on utilisait $\blacksquare_r^k$ pour interdire à des ambients autre que r d'entrer dans n, on utilise ici $\overline{\texttt{in}}^{\downarrow}\ r$ avec le même effet. Le reste de l'implantation est essentiellement identique.

Le protocole du taxi

À l'aide du mécanisme des cocapacités, le protocole du taxi peut être réécrit comme sur la figure 3.22. Les sites sont toujours représentées par des ambients nommés s_1, s_2..., les taxis par des ambients nommés *cab* et les clients par des ambients de noms quelconques.

La structure globale est la même que dans l'implantation MA. La première différence se situe dans la manière d'entrer dans le taxi ou d'en sortir. Alors que, dans la première version, n'importe quel ambient pouvait pénétrer *cab*, ici,

$$
\begin{array}{lcl}
\texttt{Call}\ f\ c & \triangleq & r\left[\texttt{out}\ c.\texttt{out}\ f.\texttt{in}\ cab.\overline{\texttt{open}}\ r.\texttt{in}\ f.\overline{\texttt{in}}^{\downarrow}\ c\right] \\
\texttt{Trip}\ c\ f\ to & \triangleq & trip\left[\texttt{out}\ c.\overline{\texttt{open}}\ trip.\texttt{out}\ f.\texttt{in}\ to.\overline{\texttt{out}}^{\uparrow}\ c.\texttt{Rel}\ b\right] \\
\texttt{Client}\ f\ to & \triangleq & (\nu c : A_c)c[\texttt{Call}\ f\ c \\
 & & \quad \mid \overline{\texttt{out}}^{\uparrow}\ r.\texttt{in}\ cab.\texttt{Trip}\ c\ f\ to \\
 & & \quad \mid \overline{\texttt{out}}^{\uparrow}\ trip.\texttt{out}\ cab \\
 & &] \\
\texttt{Cab} & \triangleq & cab\left[\texttt{rec}\ X.\overline{\texttt{in}}^{\downarrow}\ r.\texttt{open}\ r.\overline{\texttt{in}}^{\uparrow}\ trip.\texttt{open}\ trip.\texttt{Acq}\ b.X\right]
\end{array}
$$

FIG. 3.22 – Le protocole du taxi – implantation en CA.

$$
\begin{array}{lcl}
\texttt{Thread} & = & \texttt{rec}\ X.\overline{\texttt{in}}^{\downarrow}\ request.\texttt{open}\ request.(M).(\\
 & & \quad reply\,[\texttt{out}\ s.\texttt{in}\ M] \mid \overline{\texttt{out}}^{\uparrow}\ reply.X \\
 & &) \\
\texttt{Server} & = & s\left[\underbrace{\texttt{Thread} \mid \texttt{Thread} \mid \ldots \mid \texttt{Thread}}_{k \text{ instances}}\right] \\
\texttt{Client}\ c & = & c[\\
 & & \quad request\,[\texttt{out}\ c.\texttt{in}\ s.\overline{\texttt{open}}\ request.\langle c\rangle] \mid \\
 & & \quad \overline{\texttt{out}}^{\uparrow}\ request.\overline{\texttt{in}}^{\downarrow}\ reply \\
 & &] \mid \\
 & & \quad \overline{\texttt{in}}^{\uparrow}\ request.\overline{\texttt{out}}^{\downarrow}\ reply
\end{array}
$$

FIG. 3.23 – Protocole client-serveur minimal – version CA.

seuls les ambients r, *trip* et b y sont a priori autorisés. Lorsque r est ouvert, cette autorisation est communiquée au client – et uniquement au client – grâce à $\overline{\texttt{in}}^{\downarrow}\ c$. De même, nul ne peut sortir, jusqu'à ce que le $\overline{\texttt{out}}^{\uparrow}\ c$ apporté par $\texttt{Trip}\ c\ f\ to$ donne cette possibilité au client – et uniquement à lui.

Grâce aux cocapacités, le mécanisme de synchronisation est beaucoup plus simple à écrire que dans les MA. Ainsi, en examinant le terme, on vérifie simplement qu'aucun client ne peut sortir avant d'avoir atteint sa destination et qu'aucun client ne peut entrer dans le taxi si celui-ci est déjà plein. De même, seul le verrou b est important, pour assurer que le taxi ne part pas avant que le client soit descendu. Nous aurions, d'ailleurs, pu éviter d'utiliser b en communiquant le nom du client plutôt que des cocapacités d'entrée et de sortie mais nous avons préféré garder l'implantation CA aussi proche que possible de la version MA.

Serveur

Les attaques qui permettaient de perturber le fonctionnement du client dans le cadre des Mobile Ambients s'appliquent-elles aussi à ces processus des Controlled Ambients ? La figure 3.23 présente une version du client-serveur minimal

adaptée aux CA : chaque `Thread` consiste en une boucle qui autorise une (et une seule) requête à entrer, la traite, et attend que le traitement de la requête soit terminé pour accepter la requête suivante. Le serveur contient alors k instances de `Thread`, où k est le nombre de requêtes que le serveur peut traiter simultanément.

Comme dans BoCa, l'attaque $\texttt{Rogue}_1$ n'a aucun effet sur le système car l'ambient s n'est pas ouvrable. Cependant, des attaques $\texttt{Rogue}_2$, $\texttt{Rogue}_3$, $\texttt{Rogue}_4$ peuvent toujours empêcher le fonctionnement du serveur.

3.3.4 Des types pour contrôler les ressources

Maintenant que nous sommes capables de contrôler l'allocation et la désallocation de ressources dans chaque ambient – c'est-à-dire que nous disposons d'un calcul conscient des ressources – nous pouvons définir une notion de ressources et des méthodes pour prouver qu'un système est ressource-contrôlé.

Définition 8 (Politique de contrôle des ressources – CA)
Une politique de contrôle des ressources est une fonction partielle qui, à chaque nom d'ambient pour laquelle elle est définie, associe la taille de la réserve de l'ambient et l'encombrement de l'ambient.

Comme nous confondrons plus tard politiques de contrôle des ressources et environnements de typage, nous noterons d'ores et déjà Γ, $\Delta\ldots$ les politiques de contrôle des ressources. Si Γ est une politique de contrôle des ressources, on notera $\Gamma, n : (s, e)$ la fonction qui à n associe (s, e) et à tout autre nom a associe $\Gamma(a)$.

Définition 9 (Utilisation des ressources – CA)
Si nous considérons une politique de contrôle des ressources Γ, l'utilisation des ressources par un processus P, notée $res_\Gamma(P)$, est définie par

- $res_\Gamma(\mathbf{0}) = 0$
- $res_\Gamma(P|Q) = res_\Gamma(P) + res_\Gamma(Q)$
- $res_\Gamma(\texttt{rec}\ X.P) = res_\Gamma(X) = 0$
- $res_\Gamma(a\,[P]) = e$ *si* $res_\Gamma(P) \leq s$ *et* $\Gamma(a) = (s, e)$
- $res_\Gamma(M.P) = 0$
- $res_\Gamma((\nu n : A)P) = res_{\Gamma, n:(s_A, e_A)}(P)$ *si l'annotation A spécifie que n a une réserve de taille s et un encombrement e*
- $res_\Gamma((x : A).P) = res_\Gamma(\langle M \rangle) = 0$

et n'est pas définie dans les autres cas. Dès que $res_\Gamma(P)$ est défini, on dira que, dans l'état courant, P respecte la politique Γ.

Il est à noter que, contrairement à la définition du poids dans BoCa, qui prend en compte aussi bien les ressources déjà allouées que les ressources qui vont être allouées, res_Γ ne compte que les ressources allouées à un instant donné. Dans les CA, nous utilisons un système de types pour comptabiliser l'utilisation future des ressources.

Définition 10 (Ressource-contrôlé – CA)
Un processus P est dit ressource contrôlé dans le cadre d'une politique de contrôle des ressources Γ si, pour tout Q tel que $P \longrightarrow^* Q$, $res_\Gamma(Q)$ est défini.

$$
\begin{array}{lcll}
A & ::= & Amb(s,e)[T] & s \in \mathbf{N} \cup \{\infty\}, e \in \mathbf{N} \cup \{\infty\} \\
U & ::= & Proc(t)[T] & t \in \mathbf{N} \cup \{\infty\} \\
T & ::= & Ssh & \\
 & | & b, A & b \in \mathbf{N} \cup \{\infty\}
\end{array}
$$

FIG. 3.24 – Grammaire de types pour le contrôle des ressources – CA.

Types

La grammaire des types est présentée sur la figure 3.24.

Le type d'un ambient a est $Amb(s,e)[T]$ lorsque s est la taille de sa réserve, e son encombrement et T le type des messages qui peuvent être échangés dans a (T est aussi appelé "sujet de conversation" [17]). Un ambient peut avoir une réserve infinie et un encombrement infini. En d'autres termes, certains sites peuvent accueillir une quantité illimitée de sous-locations et certaines entités ne sont acceptables que dans de telles localités. De plus, contrairement à BoCa, par exemple, s et e sont totalement indépendants, ce qui permet notamment de modéliser des allocations dans des espaces différents. Ainsi, un ordinateur portable pourra disposer de 1Go de mémoire vive (soit, si l'on considère que notre unité de ressources est l'octet, $s = 10^9$) mais le fait accueillir cet ordinateur sur un réseau ne nécessitera que quelques octets supplémentaires chez le routeur (soit $e =$ quelques unités).

Le type d'un processus P est $Proc(t)[T]$ si l'effet de P est t, c'est-à-dire si P peut s'exécuter en utilisant au plus t ressources, et si T est le type des messages qui peuvent être échangés dans l'ambient conteneur. Un sujet de conversation peut être Ssh si aucun message n'est échangé ou b, A si les processus échangent des noms de type A et si la communication elle-même peut entraîner l'allocation de b ressources. Nous reviendrons plus tard sur le typage des communications.

Il est important de noter que, dans cette version du système de types, nous ne pouvons pas typer la communication de capacités. Bien que cette limitation ne soit pas, a priori, intrinsèque à nos méthodes, nous l'avons introduite car elle simplifie énormément le système de types, notamment ses extensions (cf. sections 3.3.6 et chapitre 9).

Un environnement de typage Γ est une fonction partielle qui, à chaque nom d'ambient pour lequel elle est définie associe un type d'ambient et à chaque nom de processus pour lequel elle est définie associe un type de processus. Nous noterons $\Gamma, n : A$ la fonction qui au nom d'ambient n associe le type A et pour tout autre nom réagit comme Γ. Nous noterons de même $\Gamma, X : U$ la fonction qui au nom de processus X associe le type U et pour tout autre nom réagit comme Γ.

Comme il est trivial de convertir un environnement de typage en une politique de contrôle de ressources, nous considérerons les environnements de typage comme des politiques de contrôle de ressources. Par suite, nous étendons la définition de res_Γ et de termes ressource-contrôlés aux cas où Γ est un environnement.

Règles de typage

Les règles de typage sont présentées sur la figure 3.25. Notons que toutes ces règles sont valables uniquement pour des entiers positifs ou infinis, ce qui explique notamment notre présentation à l'aide d'inégalités et sans soustractions.

Règles de base Comme le processus terminé **0** ne communique pas et comme il ne consomme pas de ressources, il peut accommoder de n'importe quel type, comme spécifié par T-Nil. De même, la création d'un nouveau nom n'entraîne ni allocation ni désallocation et se contente de mettre à jour l'environnement, comme l'indique T-Res – nous traiterons différemment la création des noms dans la partie III, consacrée à la mobilité des noms. Les règles T-AmbName et T-ProcName permettent d'extraire de l'environnement respectivement le type d'un ambient et celui d'un processus. La règle T-Rec spécifie le typage naturel d'une expression récursive.

Déplacements Les règles T-In et T-Out se contentent de spécifier que, lorsqu'un ambient a bouge, le mouvement n'implique aucune allocation ni désallocation à l'intérieur de a. À l'inverse, les règles T-CoIn et T-CoOut spécifient respectivement que, pour recevoir un ambient d'encombrement e, il est nécessaire de prévoir e ressources, et que, si un ambient de taille e part, il libère e ressources.

Ouverture L'ouverture se type à l'aide de T-Open et de T-CoOpen. Ainsi, ouvrir un ambient d'encombrement e et disposant d'une réserve de taille s conduit à la désallocation de e ressources et à la réallocation immédiate de s ressources, tandis qu'il ne se passe rien du point de vue de l'ambient ouvert. Les sujets de conversation des deux ambients doivent alors être les mêmes. En pratique, il est possible de typer l'ouverture de manière plus complexe et beaucoup plus fine, comme nous le montrons dans la section 3.3.6.

Structure Lorsque la politique de contrôle des ressources spécifie qu'un ambient a a s ressources en réserve et est d'encombrement e, la règle T-Amb vérifie que a ne contient pas de processus nécessitant plus de s ressources, que les sujets de conversation sont corrects, puis réserve e ressources (ou plus) dans l'ambient parent.

La règle T-Par permet de typer la composition parallèle de deux processus : si P nécessite au plus t_P ressources, si Q nécessite au plus t_Q ressources et si leur sujet de conversation est le même, alors $P|Q$ nécessite au plus $t_P + t_Q$ ressources et garde le sujet de conversation commun de P et de Q.

Communications Enfin, les règles T-Snd et T-Rcv permettent de typer une communication. Le type de $(x : A).P$ considère que P est en attente et ne consomme a priori aucune ressource tant qu'il n'est pas déclenché par un processus $\langle M \rangle$. Le processus $\langle M \rangle$, par contre, est typé de manière à prendre en compte l'effet de P. Ainsi, si P nécessite au plus b ressources pour s'exécuter, on réservera b ressources (ou plus) pour $\langle M \rangle$.

Pour permettre cette comptabilité, nous utilisons un raffinement du typage des communications entre processus par sujets de conversation [17]. Ainsi, un

T-Nil $\Gamma \vdash \mathbf{0} : T$ $\qquad$ T-Res $\dfrac{\Gamma, n : A \vdash P : U}{\Gamma \vdash (\nu n : A)P : U}$ $\qquad$ T-AmbName $\dfrac{\Gamma(n) = A}{\Gamma \vdash n : A}$

$$\text{T-ProcName}\ \frac{\Gamma(X) = Proc(t)[T]}{\Gamma \vdash X : Proc(u)[T]}\ u \geq t$$

$$\text{T-Rec}\ \frac{\Gamma, X : Proc(t)[T] \vdash P : Proc(t)[T]}{\Gamma \vdash \texttt{rec}\ X.P : Proc(u)[T]}\ u \geq t$$

$$\text{T-In}\ \frac{\Gamma \vdash P : U}{\Gamma \vdash \texttt{in}\ m.P : U} \qquad \text{T-Out}\ \frac{\Gamma \vdash P : U}{\Gamma \vdash \texttt{out}\ m.P : U}$$

$$\text{T-CoIn}\ \frac{\Gamma \vdash P : Proc(t)[T] \qquad \Gamma \vdash m : Amb(s,e)[T']}{\Gamma \vdash \overline{\texttt{in}}\ m.P : Proc(u)[T]}\ u \geq t + e$$

$$\text{T-CoOut}\ \frac{\Gamma \vdash P : Proc(t)[T] \qquad \Gamma \vdash m : Amb(s,e)[T']}{\Gamma \vdash \overline{\texttt{out}}\ m.P : Proc(u)[T]}\ u + e \geq t$$

$$\text{T-Open}\ \frac{\Gamma \vdash m : Amb(s,e)[T] \qquad \Gamma \vdash P : Proc(t)[T]}{\Gamma \vdash \texttt{open}\ m.P : Proc(u)[T]}\ u + e \geq t + s$$

$$\text{T-CoOpen}\ \frac{\Gamma \vdash m : Amb(s,e)[T] \qquad \Gamma \vdash Q : Proc(t)[T]}{\Gamma \vdash \overline{\texttt{open}}\ m.Q : Proc(t)[T]}$$

$$\text{T-Amb}\ \frac{\Gamma \vdash m : Amb(s,e)[T] \qquad \Gamma \vdash P : Proc(s)[T]}{\Gamma \vdash m\,[P] : Proc(t)[T']}\ e \leq t$$

$$\text{T-Par}\ \frac{\Gamma \vdash P : Proc(t_P)[T] \qquad \Gamma \vdash Q : Proc(t_Q)[T]}{\Gamma \vdash P \mid Q : Proc(u)[T]}\ u \geq t_P + t_Q$$

T-Snd $\dfrac{\Gamma \vdash m : A}{\Gamma \vdash \langle m \rangle : Proc(u)[b, A]}\ u \geq b$ $\qquad$ T-Rcv $\dfrac{\Gamma, x : A \vdash P : Proc(b)[b, A]}{\Gamma \vdash (x : A).P : Proc(v)[b, A]}$

FIG. 3.25 – Règles de typage des CA.

ambient a est de type $Amb(\_)[b, A]$ si et seulement si seules des communications de type A peuvent avoir lieu dans l'ambient et si cees communications déclenchent des processus qui nécessitent au plus b ressources. En particulier, tous les processus contenus dans a seront de type $Proc(\_)[b, A]$ – notamment les processus émetteurs $\langle M \rangle$ et récepteurs $(x : A).P$.

En pratique, il est possible de typer les communications de manière beaucoup plus fine, comme nous l'avons fait dans le cadre du π-calcul (cf. section 6.3.7).

Propriétés

Présentons brièvement quelques propriétés intéressantes sur le système de types des Controlled Ambients.

Lemme 2 (Sous-typage)
Soient Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P : Proc(t)[T]$. *Alors, pour tout* $u \geq t$, *on aura* $\Gamma \vdash P : Proc(u)[T]$.

Ce lemme est trivial mais fondamental. En effet, des règles telles que T-CoOut, T-Amb ou T-Rcv sont conçues pour ne typer que des processus qui nécessitent suffisamment de ressources pour garantir que les opérations restent dans $\mathbf{N} \cup \{\infty\}$. Grâce au lemme de sous-typage, nous pouvons étendre ces résultats à des processus moins gourmands. Une autre possibilité, moins esthétique et moins généralisable (cf. section 9) aurait été, comme dans le système de types de BoCa, de recadrer manuellement les résultats dans $\mathbf{N} \cup \{\infty\}$ à chaque étape, à l'aide d'opérations telles que $x \mapsto max(x, 0)$.

Corollaire 1 (Type minimal) *Soient* Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P : Proc(t)[T]$. *Alors, il existe* u *minimal tel que* $\Gamma \vdash P : Proc(u)[T]$.

Notons que u n'est minimal que pour un sujet de conversation T donné. Nous présentons une preuve de ce lemme dans l'annexe A.1.

Lemme 3 (Renforcement – noms d'ambients)
Si $\Gamma, n : A \vdash P : U$ *et* $n \notin fv(P)$, *alors* $\Gamma \vdash P : U$.

Lemme 4 (Renforcement – noms de processus)
Si $\Gamma, X : U \vdash P : V$ *et* $X \notin fv(P)$, *alors* $\Gamma \vdash P : V$.

Lemme 5 (Affaiblissement – noms d'ambients)
Si $\Gamma \vdash P : U$ *et* $n \notin fv(P)$ *alors* $\Gamma, n : A \vdash U$.

Ces trois lemmes sont habituels et leurs preuves sont standard. Nous ne les présenterons pas.

Lemme 6 (Les bons types respectent la politique)
Si $\Gamma \vdash P : U$, *alors* P *respecte* Γ.

La preuve tient en une induction structurelle sur la structure d'un certificat de $\Gamma \vdash P : U$. Nous présentons cette preuve dans l'annexe A.1.

Théorème 2 (Subject Reduction) *Si* $\Gamma \vdash P : U$ *et si* $P \longrightarrow Q$, *alors* $\Gamma \vdash Q : U$.

La preuve de ce théorème est présentée dans l'annexe A.2. Notons que, une fois de plus, nous faisons appel au lemme de sous-typage. En effet, a priori, un processus peut nécessiter moins de ressources après une étape de réduction, par exemple si cette étape a permis un nettoyage à l'aide de `open` a d'un ambient qui ne contenait que $\overline{\texttt{open}}\ a$. Le sous-typage nous permet de nous contenter de cette formulation plus simple.

Théorème 3 (Contrôle des ressources) *Soient Γ environnement et P un processus. Si P est typable dans Γ, alors P est ressource-contrôlé dans Γ.*

Il s'agit d'un corollaire direct de ce qui précède.

3.3.5 Exemples

Verrous

Considérons les verrous définis à la section 3.3.3. Considérons un environnement Γ tel que $\Gamma \vdash P : Proc(t_P)[T]$, $\Gamma \vdash Q : Proc(t_Q)[T]$ et $\Gamma(n) = Amb(0,0)[T]$.

Nous pouvons alors typer l'expression `Acq` $n.P \mid$ `Rel` $n.Q$ à l'aide de la dérivation suivante (pour des raisons de lisibilité, nous avons omis les applications de T-AmbName) :

Typage de `Acq` $n.P$		
$\Gamma \vdash n :$	$Amb(0,0)[T]$	Par hypothèse
$\Gamma \vdash P :$	$Proc(t_P)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash$ `open` $n.P :$	$Proc(t_P)[T]$	Par T-Open
Typage de $\mathbf{0}$		
$\Rightarrow \Gamma \vdash \mathbf{0} :$	$Proc(0)[T]$	Par T-Nil
Typage de $\overline{\texttt{open}}\ n.\mathbf{0}$		
$\Gamma \vdash n :$	$Amb(0,0)[T]$	Par hypothèse
$\Gamma \vdash \mathbf{0} :$	$Proc(0)[T]$	Cf. plus haut
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ n.\mathbf{0} :$	$Proc(0)[T]$	Par T-CoOpen
Typage de $n\,[\overline{\texttt{open}}\ n]$		
$\Gamma \vdash n :$	$Amb(0,0)[T]$	Par hypothèse
$\Gamma \vdash \overline{\texttt{open}}\ n.\mathbf{0} :$	$Proc(0)[T]$	Cf. plus haut
$\Rightarrow \Gamma \vdash n\,[\overline{\texttt{open}}\ n] :$	$Proc(0)[T]$	Par T-Amb
Typage de `Rel` $n.Q$		
$\Gamma \vdash n\,[\overline{\texttt{open}}\ n] :$	$Proc(0)[T]$	Cf. plus haut
$\Gamma \vdash Q :$	$Proc(t_Q)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash n\,[\overline{\texttt{open}}\ n] \mid Q :$	$Proc(t_Q)[T]$	Par T-Par
Typage de `Acq` $n.P \mid$ `Rel` $n.Q$		
$\Gamma \vdash$ `open` $n.P :$	$Proc(t_P)[T]$	Cf. plus haut
$\Gamma \vdash n\,[\overline{\texttt{open}}\ n] \mid Q :$	$Proc(t_Q)[T]$	Cf. plus haut
$\Rightarrow \Gamma \vdash$ `Acq` $n.P \mid$ `Rel` $n.Q :$	$Proc(t_P + t_Q)[T]$	Par T-Par

○

Ainsi, nous venons de prouver que `Acq` $n.P \mid$ `Rel` $n.Q$ est ressource-contrôlé et nécessite au plus autant de ressources que $P \mid Q$.

Communication sur un canal

Considérons maintenant les opérations de communication sur un canal, définies dans la section 3.3.3. Les processus, plus complexes, mettent en jeu non seulement les primitives d'émission et de réception mais aussi la mobilité et surtout l'ouverture d'un ambient pour utiliser un processus contenu dans cet ambient.

Nous noterons $T = t_Q, A$ et nous supposerons $\Gamma(r) = \Gamma(n) = Amb(t_Q, 0)[T]$, $\Gamma \vdash P : Proc(t_P)[T]$ et $\Gamma, x : A \vdash Q : Proc(t_Q)[T]$.

Nous pouvons alors typer l'émission de la manière suivante :

Typage de $\langle M \rangle$		
$\Rightarrow \Gamma \vdash \langle M \rangle :$	$Proc(t_Q)[t_Q, A]$	Par T-Send
Typage de $\texttt{open}\ r.\langle M \rangle$		
$\Gamma \vdash \langle M \rangle :$	$Proc(t_Q)[t_Q, A]$	Cf. plus haut
$\Gamma \vdash r :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \texttt{open}\ r.\langle M \rangle :$	$Proc(t_Q)[T]$	Par T-Open
Typage de $\overline{\texttt{in}}^{\downarrow}\ r.\texttt{open}\ r.\langle M \rangle$		
$\Gamma \vdash \texttt{open}\ r.\langle M \rangle :$	$Proc(t_Q)[T]$	Cf. plus haut
$\Gamma \vdash r :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \overline{\texttt{in}}^{\downarrow}\ r.\texttt{open}\ r.\langle M \rangle :$	$Proc(t_Q)[T]$	Par T-CoIn
Typage de $n\,[\ldots]$		
$\Gamma \vdash \overline{\texttt{in}}^{\downarrow}\ r.\texttt{open}\ r.\langle M \rangle :$	$Proc(t_Q)[T]$	Cf. plus haut
$\Gamma \vdash n :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash n\left[\overline{\texttt{in}}^{\downarrow}\ r.\texttt{open}\ r.\langle M \rangle\right] :$	$Proc(0)[T]$	Par T-Amb

Typage de $\texttt{open}\ n.P$		
$\Gamma \vdash P :$	$Proc(t_P)[T]$	Par hypothèse
$\Gamma \vdash n :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \texttt{open}\ n.P :$	$Proc(t_P + t_Q)[T]$	Par T-Open

Typage de $\texttt{Send}\ M\ \texttt{on}\ n.P$		
$\Gamma \vdash n\,[\ldots] :$	$Proc(0)[T]$	Cf. plus haut
$\Gamma \vdash \texttt{open}\ n.P :$	$Proc(t_P + t_Q)[T]$	Cf. plus haut
$\Rightarrow \Gamma \vdash \texttt{Send}\ M\ \texttt{on}\ n.P :$	$Proc(t_P + t_Q)[T]$	Par T-Par

De même, la réception se type à l'aide de :

Typage de $\overline{\texttt{open}}\ n.Q$		
$\Gamma, x : A \vdash Q :$	$Proc(t_Q)[T]$	Par hypothèse
$\Gamma, x : A \vdash n :$	$Amb(t_Q, 0)[T]$	Par *Affaiblissement*
$\Rightarrow \Gamma, x : A \vdash \overline{\texttt{open}}\ n.Q :$	$Proc(t_Q)[T]$	Par T-CoOpen
Typage de $(x).\overline{\texttt{open}}\ n.Q$		
$\Gamma, x : A \vdash \overline{\texttt{open}}\ n.Q :$	$Proc(t_Q)[t_Q, A]$	Cf. plus haut
$\Rightarrow \Gamma \vdash (x).\overline{\texttt{open}}\ n.Q :$	$Proc(0)[t_Q, A]$	Par T-Rcv
Typage de $\overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q$		
$\Gamma \vdash (x).\overline{\texttt{open}}\ n.Q :$	$Proc(0)[T]$	Cf. plus haut
$\Gamma \vdash r :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q :$	$Proc(0)[T]$	Par T-CoOpen
Typage de $\texttt{in}\ n.\overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q$		
$\Gamma \vdash \overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q :$	$Proc(0)[T]$	Cf. plus haut
$\Rightarrow \Gamma \vdash \texttt{in}\ n.\overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q :$	$Proc(0)[T]$	Par T-In
Typage de $r\,[\texttt{in}\ n.\overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q]$		

$\Gamma \vdash \texttt{in}\ n.\overline{\texttt{open}}\ r.(x).\overline{\texttt{open}}\ n.Q : Proc(0)[T]$		Cf. plus haut
$\Gamma \vdash r :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash r\,[\texttt{in}\ n. \ldots] :$	$Proc(0)[T]$	Par T-AMB

Typage de $\overline{\texttt{out}}^{\downarrow}\ r$		
$\Gamma \vdash r :$	$Amb(t_Q, 0)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \overline{\texttt{out}}^{\downarrow}\ r.\mathbf{0} :$	$Proc(0)[T]$	Par T-COOUT

○

Par ces réductions, nous avons prouvé que, selon Γ, lire sur le canal n est une opération gratuite, alors qu'émettre coûte au plus $t_P + t_Q$ ressources. Ce résultat est comparable au choix que nous avions fait pour typer la communication à l'aide des règles T-SEND et T-RECV. En effet, d'après notre système de types, dans le cas de base de la communication primitive, la réception $(x : A).P$ peut être typée comme une opération gratuite, alors que l'émission $\langle M \rangle$ coûte au plus t_P ressources.

Notons au passage que ces typages imposent aussi que P ne puisse communiquer qu'avec le sujet de conversation T, alors même que P ne communique pas lui-même. Nous avons donc, d'une certaine manière, pollué le sujet de conversation de l'ambient.

Le protocole du taxi

Dans le cadre du protocole du taxi, nous cherchons à prouver que chaque taxi peut contenir au plus un seul passager à chaque instant. Pour spécifier cette propriété sous forme de types, nous devons considérer que chaque passager consomme une ressource et que chaque taxi ne contient qu'une ressource en réserve. Nous pouvons donc formuler la politique de contrôle de ressources Γ :

$$\begin{cases} A_c & = & Amb(0,1)[Ssh] & \text{(le client occupe une ressource)} \\ \Gamma(cab) & = & Amb(1,0)[Ssh] & \text{(le taxi dispose d'une ressource.)} \end{cases}$$

Complétons Γ par :

$$\begin{cases} \Gamma(rr) & = & Amb(1,0)[Ssh] & \text{(l'ambient d'appel dispose d'une ressource)} \\ \Gamma(trip) & = & Amb(0,0)[Ssh] & \\ \Gamma(b) & = & Amb(0,0)[Ssh] & . \end{cases}$$

Il est alors possible de typer les termes de la figure 3.22, à l'aide d'une réduction que nous ne détaillerons pas. En d'autres termes, le protocole du taxi est donc ressource-contrôlé dans le cadre de Γ. En particulier, cela signifie que chaque taxi ne peut contenir plus d'un client à la fois.

Il est à noter que, puisque nos ressources ne sont définies qu'au niveau du système de types, nous pouvons envisager de prouver d'autres propriétés sur le terme. Sans entrer dans les détails, nous pouvons ainsi aisément prouver que, à tout instant, le taxi peut contenir au plus un ambient auxiliaire r, $trip$ ou b, qu'il ne peut contenir aucun site ou aucun autre taxi et que le client ne contiendra jamais de clients, de sites ou de taxis.

Client-serveur

Sans entrer dans les détails, le typage du protocole client serveur permet de confirmer que au plus k requêtes ou réponses sont présentes dans le serveur à un instant donné.

Nous noterons de plus que, en donnant au serveur s un encombrement non-nul, nous pouvons aisément interdire l'attaque $\texttt{Rogue}_2$.

Ainsi, supposons que s soit le seul ambient d'encombrement non-nul. Pour simplifier, supposons que cet encombrement soit 1. Le client, les requêtes et le serveur pourront alors être inclus dans un ambient parent *network* de taille 1. Dans un tel ambient, il n'y aura alors pas assez de place pour un deuxième ambient s.

3.3.6 Des analyses plus fines

Limitations du système de types Même si, comme nous l'avons vu, il est possible de prouver des propriétés sur des processus non-triviaux à l'aide du système que nous avons présenté, certains autres processus simples mettent en évidence des limitations.

Ainsi, considérons le processus `Boom.exe` défini sur la figure 3.26. Ce terme représente un fichier exécutable qui, une fois ouvert, "explose" – c'est-à-dire ici sature l'ambient qui le contient d'une infinité potentielle de $b\,[\mathbf{0}]$. Si l'on suppose que l'encombrement de b est 1, on déduit aisément des règles de typage que le processus `Boom` a pour type $Proc(\infty)[T]$. Par conséquent, l'ambient *boom* doit disposer d'une réserve de taille infinie. Or, en fait, en examinant le processus `Boom.exe`, on peut constater que le processus `Boom` n'est jamais exécuté à l'intérieur de l'ambient *boom*, mais uniquement dans un ambient qui l'ouvrirait. En d'autres termes, *boom* ne devrait pas avoir besoin d'une réserve infinie, puisqu'aucune allocation n'a jamais lieu.

$$
\begin{array}{lcl}
\texttt{Boom} & = & \texttt{rec}\ X.(X \mid b\,[\mathbf{0}]) \\
\texttt{Boom.exe} & = & \mathit{boom}\,[\overline{\texttt{open}}\ \mathit{boom}.\texttt{Boom}]
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{Buffer}.P & = & \mathit{buf}\left[\overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ \mathit{buf}\right] \mid \overline{\texttt{out}}^{\downarrow}\ n.\overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ \mathit{buf}.P \\
\texttt{Daemon} & = & \texttt{rec}\ X.\texttt{Buffer}.X \mid n\,[\texttt{rec}\ Y.\texttt{in}\ \mathit{buf}.\texttt{out}\ \mathit{buf}.Y]
\end{array}
$$

FIG. 3.26 – Limites du typage simple des CA.

De même, le processus `Daemon` représente une boucle qui, éternellement, crée un tampon, y stocke une information, en réextrait l'information puis détruit le tampon. À l'aide du système de types que nous avons présenté, en supposant que n soit d'encombrement non-nul, on déduit que la boucle $\texttt{rec}\ X.\texttt{Buffer}.X$ a pour type $Proc(\infty)[T]$. Or, un examen du processus contenu dans *buf* montre que seul l'ambient n peut entrer dans *buf* et qu'il sera ressorti avant que *buf* soit ouvert. En d'autres termes, chaque ambient *buf* est vide lorsqu'il est ouvert. Et par conséquent, $\texttt{rec}\ X.\texttt{Buffer}.X$ ne nécessite qu'un nombre fini de ressources pour être exécuté.

Ces deux limitations sont liées au traitement de l'ouverture d'un ambient : dans le cas de `Boom.exe`, des ressources qui ne sont allouées qu'après l'ouverture, comme c'est le cas dans un programme, sont comptées comme étant allouées avant l'ouverture, tandis que, dans le cas de `Daemon`, des ressources qui ont été désallouées avant l'ouverture, comme c'est le cas dès qu'il y a garbage-collection, sont comptées comme étant toujours allouées.

Pour résoudre ce problème, il est possible de raffiner le typage de l'ouverture pour lui permettre de prendre en compte séparément les ressources nécessaires avant la dissolution et les ressources nécessaires après la dissolution. Pour la discussion qui suit, nous utiliserons les notations tirées de la règle R-OPEN :

$$\texttt{open}\ a.P \mid a\,[\overline{\texttt{open}}\ a.Q \mid R] \longrightarrow P|Q|R\ .$$

Système Q

Dans l'expression de la règle R-OPEN, Q représente le processus inclus dans a et déclenché après l'ouverture. Par conséquent, les ressources allouées par Q ne sont allouées que dans l'ambient qui ouvre a.

Pour prendre en compte ce facteur, nous introduisons un paramètre q dans le type des ambients. On notera donc $\Gamma(a) = Amb(s, e, q)[T]$ où q est le nombre de ressources dont peut avoir besoin le processus Q et où s, e et T ont leurs significations habituelles.

Les règles de typage deviennent alors

$$\text{T-QOPEN}\ \frac{\Gamma \vdash m : Amb(s,e,q)[T] \quad \Gamma \vdash P : Proc(t_P + e)[T]}{\Gamma \vdash \texttt{open}\ m.P : Proc(v_P)[T]}\ v_P + e \geq u_P + s + q$$

$$\text{T-QCOOPEN}\ \frac{\Gamma \vdash m : Amb(s,e,q)[T] \qquad \Gamma \vdash Q : Proc(u)[T]}{\Gamma \vdash \overline{\texttt{open}}\ m.Q : Proc(t)[T]}\ q \geq t_Q$$

En d'autres termes, on utilise q pour borner la contribution de Q à l'allocation de ressources et on considère ensuite que $\overline{\texttt{open}}\ r.Q$ ne participe pas à l'allocation de ressources dans a.

Ainsi, avec $\Gamma(boom) = Amb(0, 0, \infty)[T]$, on peut typer `Boom.exe` dans Γ. On aura aussi $\Gamma \vdash \texttt{open}\ boom : Proc(\infty)[T]$. Plutôt que de prouver l'intégralité du typage, nous fournissons le fragment le plus important d'une dérivation. Nous supposerons déjà prouvé $\Gamma \vdash \texttt{Boom} : Proc(\infty)[T]$.

Typage de $\overline{\texttt{open}}\ boom.\texttt{Boom}$		
$\Gamma \vdash \texttt{Boom} :$	$Proc(\infty)[T]$	Par hypothèse
$\Gamma \vdash boom :$	$Amb(0,0,\infty)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ boom.\texttt{Boom} :$	$Proc(0)[T]$	Par T-QCOOPEN
Typage de `Boom.exe`		
$\Gamma \vdash \overline{\texttt{open}}\ boom.\texttt{Boom} :$	$Proc(0)[T]$	Cf. plus haut
$\Gamma \vdash boom :$	$Amb(0,0,\infty)[T]$	Par hypothèse
$\Rightarrow \Gamma \vdash \texttt{Boom.exe} :$	$Proc(0)[T]$	Par T-AMB

○

Système R

Dans l'expression de la règle R-OPEN, R représente un processus qui peut s'exécuter indifféremment avant ou après l'ouverture. En particulier, considérons un terme tel que

$$A \triangleq a\left[\overline{\mathtt{in}}^{\downarrow}\ b.\overline{\mathtt{in}}^{\downarrow}\ b.\overline{\mathtt{in}}^{\downarrow}\ b.\overline{\mathtt{open}}\ a.Q\right] \mid b\,[\mathtt{in}\ a] \mid b\,[\mathtt{in}\ a] \mid b\,[\mathtt{in}\ a] \mid \overline{\mathtt{out}}^{\downarrow}\ b.\overline{\mathtt{out}}^{\downarrow}\ b.\overline{\mathtt{out}}^{\downarrow}\ b\ .$$

Le processus A peut évoluer en $a\,[b\,[] \mid b\,[] \mid b\,[] \mid \overline{\mathtt{open}}\ a.Q]$ – en d'autres termes, si nous notons ce processus $a\,[\overline{\mathtt{open}}\ a.Q \mid R]$, dans le rôle de R, nous aurons $b\,[] \mid b\,[] \mid b\,[]$, c'est-à-dire la trace des allocations effectuées par

$$\overline{\mathtt{in}}^{\downarrow}\ b.\overline{\mathtt{in}}^{\downarrow}\ b.\overline{\mathtt{in}}^{\downarrow}\ b\ .$$

Contrôler R peut donc nous permettre d'affiner le typage des allocations et des désallocations qui ont lieu avant l'ouverture, et dont la trace reste après l'ouverture. Pour prendre en compte ce facteur, nous introduisons un paramètre r dans le type des ambients. On notera donc $\Gamma(a) = Amb(s,e,r)[T]$ où r est le nombre de ressources dont peut avoir besoin le processus R et où s, e et T ont leurs significations habituelles.

Les règles de typage deviennent alors

$$\text{T-ROPEN}\ \frac{\Gamma \vdash m : Amb(s,e,r)[T] \qquad \Gamma \vdash P : Proc(t_P)[T]}{\Gamma \vdash \mathtt{open}\ m.P : Proc(v_P)[T]}\ v_P + e \geq t_P + r$$

$$\text{T-RCOOPEN}\ \frac{\begin{array}{c}\Gamma \vdash m : Amb(s,e,r)[T] \\ \Gamma \vdash Q : Proc(t_Q)[T]\end{array}}{\Gamma \vdash \overline{\mathtt{open}}\ m.Q : Proc(v_Q)[T]}\ v_Q + r \geq t_Q + s, s \neq \infty$$

Notons la condition $s \neq \infty$. Sans entrer dans les détails de la preuve, si $s = \infty$, nous pouvons avoir $t_Q = \infty$, $v_Q = \infty$ et $r = 0$, ce qui nous permettrait, par exemple, de déduire $\Gamma \vdash \mathtt{open}\ m.P \mid m\,[\overline{\mathtt{open}}\ m.Q \mid R] : Proc(t_P - e)[\_]$ alors que $\Gamma \vdash P \mid Q \mid R : Proc(\infty)[\_]$.

À partir du moment où $s \neq \infty$, nous utilisons s, on utilise r pour borner la contribution de R à l'allocation de ressources. Ainsi, avec $\Gamma(buf) = Amb(1,1,0)[T]$ et $\Gamma(n) = Amb(0,1,0)[T]$, on peut typer `Daemon` dans Γ et avoir $\Gamma \vdash \mathtt{Daemon} : Proc(2)[T]$, au lieu de $Proc(\infty)[T]$.

Plutôt que de prouver l'intégralité du typage, nous fournissons le fragment le plus important d'une dérivation possible. Afin de typer la boucle récursive, nous noterons $\Delta = \Gamma, X : Proc(1)[T]$:

Typage de $\overline{\mathtt{open}}\ buf$		
$\Delta \vdash \mathbf{0} :$	$Proc(0)[T]$	Par T-NIL
$\Delta \vdash buf :$	$Amb(1,1,0)[T]$	Par hypothèse
$\Rightarrow \Delta \vdash \overline{\mathtt{open}}\ buf :$	$Proc(1)[T]$	Par T-RCOOPEN
Typage de $\overline{\mathtt{out}}^{\uparrow}\ n.\overline{\mathtt{open}}\ buf$		
$\Delta \vdash \overline{\mathtt{open}}\ buf :$	$Proc(1)[T]$	Cf. plus haut
$\Delta \vdash n :$	$Amb(0,1,0)[T]$	Par hypothèse
$\Rightarrow \Delta \vdash \overline{\mathtt{out}}^{\uparrow}\ n.\overline{\mathtt{open}}\ buf :$	$Proc(0)[T]$	Par T-COOUT

Typage de $\overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf$		
$\Delta \vdash \overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf :$	$Proc(0)[T]$	Cf. plus haut
$\Delta \vdash n :$	$Amb(0,1,0)[T]$	Par hypothèse
$\Rightarrow \Delta \vdash \overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf :$	$Proc(1)[T]$	Par T-CoIn
Typage de $buf\ \left[\overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf\right]$		
$\Delta \vdash \overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf :$	$Proc(1)[T]$	Cf. plus haut
$\Delta \vdash buf :$	$Amb(1,1,0)[T]$	Par hypothèse
$\Rightarrow \Delta \vdash buf\ \left[\overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf\right] :$	$Proc(1)[T]$	Par T-Amb
Typage de $\texttt{open}\ buf.X$		
$\Delta \vdash X :$	$Proc(1)[T]$	Par hypothèse
$\Delta \vdash buf :$	$Amb(1,1,0)[T]$	Par hypothèse
$\Rightarrow \Delta \vdash \texttt{open}\ buf.X :$	$Proc(0)[T]$	Par T-ROpen
Typage de $\overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ buf.X$		
$\Delta \vdash n :$	$Amb(0,1,0)[T]$	Par hypothèse
$\Delta \vdash \texttt{open}\ buf.X :$	$Proc(0)[T]$	Cf. plus haut
$\Rightarrow \Delta \vdash \overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ buf.X :$	$Proc(1)[T]$	Par T-CoIn
Typage de $\overline{\texttt{out}}^{\downarrow}\ n.\overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ buf.X$		
$\Delta \vdash \overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ buf.X :$	$Proc(1)[T]$	Cf. plus haut
$\Delta \vdash n :$	$Amb(0,1,0)[T]$	Par hypothèse
$\Rightarrow \Delta \vdash \overline{\texttt{out}}^{\downarrow}\ n.\overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ buf.X :$	$Proc(0)[T]$	Par T-CoOut
Typage de $\texttt{Buffer}.X$		
$\Delta \vdash buf\ \left[\overline{\texttt{in}}^{\downarrow}\ n.\overline{\texttt{out}}^{\uparrow}\ n.\overline{\texttt{open}}\ buf\right] :$	$Proc(1)[T]$	Cf. plus haut
$\Delta \vdash \overline{\texttt{out}}^{\downarrow}\ n.\overline{\texttt{in}}^{\uparrow}\ n.\texttt{open}\ buf.X :$	$Proc(0)[T]$	Cf. plus haut
$\Rightarrow \Delta \vdash \texttt{Buffer}.X :$	$Proc(1)[T]$	Par T-Par
Typage de $\texttt{rec}\ X.\texttt{Buffer}.X$		
$\Delta \vdash \texttt{Buffer}.X :$	$Proc(1)[T]$	Cf. plus haut
$\Gamma \vdash \texttt{rec}\ X.\texttt{Buffer}.X :$	$Proc(1)[T]$	Par T-Rec

○

Système raffiné pour l'ouverture

Il est tout à fait possible de combiner les systèmes Q et R en un système raffiné. Les règles de typage deviennent alors

$$\text{T-BetterOpen}\ \frac{\Gamma \vdash m : Amb(s,e,q,r)[T] \quad \Gamma \vdash P : Proc(t_P)[T]}{\Gamma \vdash \texttt{open}\ m.P : Proc(v_P)[T]}\ v_P + e \geq t_P + q + r$$

$$\text{T-BetterCoOpen}\ \frac{\Gamma \vdash m : Amb(s,e,q,r)[T] \quad \Gamma \vdash Q : Proc(q)[T]}{\Gamma \vdash \overline{\texttt{open}}\ m.Q : Proc(v_Q)[T]}\ v_Q + r \geq s, s \neq \infty$$

Sans entrer dans les détails techniques, le système raffiné gère correctement les termes `Boom.exe` et `Daemon` et les propriétés présentées à la section 3.3.4 restent vraies.

Notons que nous avons écrit $\Gamma \vdash Q : Proc(q)[T]$ en comptant, une fois de plus, sur la propriété de sous-typage. En d'autres termes, pour tout $t_Q \leq q$, nous pouvons accepter $\Gamma \vdash Q : Proc(t_Q)[T]$;

3.3.7 Discussion

Le calcul des Controlled Ambients est un formalisme construit à partir des Mobile Ambients par l'ajout de cocapacités strictes pour chaque capacité des MA, à l'aide de mécanismes de synchronisation à trois parties lors des déplacements.

Si, en lui-même, ce calcul ne suffit pas à exprimer des propriétés de ressources, grâce à la possibilité de réagir aux allocations et désallocations offerte par les cocapacités, il est possible d'écrire des systèmes conscients des ressources. La représentation des ressources et des politiques de contrôle de ces ressources est entièrement statique et se fait grâce au système de types que nous avons proposé.

Comparaison avec BoCa Il est à noter que, malgré les points de vue différents, les mécanismes de cocapacités sont en fait assez proches des ressources $\blacksquare\_$ de BoCa. Ainsi, $\overline{\texttt{in}}^{\downarrow}\ a$ et $\blacksquare_a$ permettent dans les deux cas à n'importe quel ambient nommé a d'entrer et uniquement aux ambients nommés a. En pratique, après avoir expérimenté les deux mécanismes, il s'avère que les cocapacités permettent un contrôle plus précis, aussi bien statique que dynamique, apparenté à des spécifications strictes de protocoles, mais imposent une syntaxe plus lourde et manquent, à l'inverse, de la généralité qu'apportent les ressources universelles $\blacksquare_*$ ou la possibilité de retirer une ressource, par exemple pour la renommer, à l'aide de $\triangleright$.

Notons aussi que, là où nous considérons que les ressources nécessaires pour allouer $a\ [P]$ sont allouées dès que ce processus n'est plus gardé par une capacité, cocapacité, communication ou préfixe de récursion, BoCa permet de décider explicitement à quel moment les ressources en question sont consommées, à l'aide de $\triangleright$.

En revanche, là où BoCa restreint les possibilités de contrôle en imposant de fixer de manière rigide une fois pour toutes le poids d'un ambient, notre système de types nous offre la liberté supplémentaire de pouvoir être reconfiguré pour prouver des propriétés différentes sur un même terme – y compris des propriétés apparemment distinctes des ressources, comme nous le verrons à la section 9.

Bilan Le langage des Controlled Ambients nous a permis de modéliser des protocoles non-triviaux (y compris un fragment de l'ordonnanceur noyau Linux, non présenté ici). Si ce formalisme reste relativement simple, comme dans les autres calculs des Ambients, la modélisation peut nécessiter de développer un certain nombre de constructions peu naturelles à implanter, comme des canaux de communication ou des barrières de synchronisation généralisées. En particulier, la duplication d'informations complexes, qui est une opération primitive dans la majeure partie des langages de programmation et qui est une fonctionnalité importante dans de nombreux protocoles, est impossible dans les CA, tout comme dans les MA ou BoCa. De même, le formalisme ne permet pas de supprimer totalement un ambient lorsque celui-ci contient des processus arbitraires, ce qui peut s'avérer gênant pour procéder au nettoyage d'un terme.

Si nous nous restreignons aux protocoles que les Mobile Ambients permettent de décrire naturellement, en revanche, les Controlled Ambients sont généralement adaptés. En particulier, nous conjecturons que le mécanisme des cocapacités omniprésentes, quoique lourd, peut être essentiellement ajouté automatiquement à une implantation Mobile Ambients du système, et retouché à la

main dans les cas problématiques afin de garantir le respect de protocoles et l'utilisation des ressources.

Quant au système de types des CA, il s'avère assez puissant pour garantir de nombreuses politiques de contrôle de ressources. Notons ainsi que les politiques de contrôle de ressources sont globalement simples à exprimer à l'aide de ce système. De plus, comme les règles ne fait appel qu'à des inéquations linéaires sur des entiers positifs ou infinis, nous supposons qu'il est possible, à l'aide de techniques habituelles de programmation linéaire, de rendre l'analyse et l'inférence totalement automatiques.

Chapitre 4

Extensions et variantes

4.1 Mouvements objectifs : les Seals

Le problème auquel nous nous attaquons dans cette section est celui du contrôle des ressources en présence de mouvements non plus subjectifs mais objectifs. Pour ce faire, nous allons employer le calcul des Seals.

Le calcul des Seals [19] est un autre calcul de la famille des ambients, développé par Jan Vitek et Giuseppe Castagna. Par son système de localités et de processus, ce formalisme se rapproche du calcul des Mobile Ambients. Cependant, à la différence des MA, les processus peuvent communiquer ou déplacer les sites – ou *seals* – non à l'aide de capacités mais à travers des canaux.

Un canal, comme dans le π-calcul ou dans l'exemple de la section 3.1.2, est une entité nommée sur laquelle un processus peut soit émettre, soit recevoir. Les mouvements sont dits *objectifs* car la procédure de déplacement d'un seal consiste à émettre ce seal sur un canal, depuis le site parent, et à recevoir ce site sur le même canal, depuis le même site ou un autre. Grâce à cette possibilité, qui permet aussi de suspendre ou d'arrêter l'exécution des processus contenus dans un site, le calcul permet de décrire les processus légers et les exceptions plus facilement que dans les MA.

4.1.1 Le langage

Nous présentons ici une version légèrement modifiée des Seals, qui emploie la récursion au lieu de la réplication. Bien que nous ayons depuis étendu nos méthodes pour gérer la réplication, nous avons préféré conserver notre travail sur les Seals tel que nous l'avons présenté à Foundations of Global Computing [78]. De plus, pour des raisons de simplicité, et comme cela ne change en rien nos résultats, nous nous contenterons d'une version monadique du calcul et nous emploierons le dialecte des Seals "à canaux partagés" [19].

La syntaxe des Seals est présentée sur la figure 4.1. On suppose l'existence d'un ensemble a, b, c, m, $n\ldots$ de noms. On emploiera P, $Q\ldots$ pour désigner les processus, α pour désigner les actions et η pour désigner les directions de communication.

Nous ne nous étendrons pas sur les constructions communes avec les MA ou les CA. L'action $\overline{a}^{\eta_1}(b)$ et l'action $a^{\eta_2}(c)$ se synchronisent : si η_1 et η_2 se correspondent, le nom b est communiqué sur le canal a et lié au nom c. De

$$
\begin{array}{rcl}
P, Q & ::= & \mathbf{0} \\
& | & P|Q \\
& | & \texttt{rec}\ X.P \\
& | & (\nu a : T)P \\
& | & a\,[P] \\
& | & \alpha.P \\
& | & X
\end{array}
\qquad
\begin{array}{rcl}
\alpha & ::= & a^{\eta}(b) \\
& | & \overline{a}^{\eta}(b) \\
& | & \overline{a}^{\eta}\{b\} \\
& | & a^{\eta}\{b_1, \ldots, b_n\}
\end{array}
\qquad
\begin{array}{rcl}
\eta & ::= & * \\
& | & \uparrow \\
& | & a
\end{array}
$$

FIG. 4.1 – Syntaxe du calcul des Seals.

$$\textsc{Struct-Zero-Res}\ (\nu n)\mathbf{0} \equiv \mathbf{0}$$

$$\textsc{Struct-Res-Res}\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \ \text{si}\ n \neq m$$

$$\textsc{Struct-Res-Par}\ (\nu n)(P|Q) \equiv P|(\nu n)Q \qquad \text{si}\ n \notin fv(P)$$

FIG. 4.2 – Congruence structurelle des Seals.

même, l'action $\overline{a}^{\eta_1}\{b\}$ et l'action $a^{\eta_2}\{b_1, \ldots, b_n\}$ se synchronisent : si η_1 et η_2 se correspondent, le seal b est supprimé du site de départ et remplacé par des copies nommées b_1, ..., b_n, dans le site de destination.

Nous emploierons les mêmes raccourcis syntaxiques que dans les MA.

La congruence structurelle est la plus petite relation de congruence $\equiv$ qui fait de $(P/\equiv, |, \mathbf{0})$ un monoïde commutatif et qui satisfait les axiomes de la figure 4.2. Comme dans les calculs précédents, on identifie les termes égaux après renommage d'un nom lié. Ainsi, on aura $(\nu x)P = (\nu y)P\{x \leftarrow y\}$ et $c\text{-}(x)P = c\text{-}(y)P\{x \leftarrow y\}$ lorsque y est frais dans P.

Les règles de réduction sont présentées sur la figure 4.4. Il est à noter que, dans les Seals et contrairement aux calculs précédents, $m\,[(\nu a)P]$ et $(\nu a)m\,[P]$ ne sont pas structurelle-ment équivalents. En effet,

$$m\,[(\nu a)P] \mid \overline{c}^{*}\{m\} \mid c^{*}\{m_1, m_2, \ldots, m_n\} \longrightarrow m_1\,[(\nu a)P] \mid \ldots \mid m_n\,[(\nu a)P]$$

alors que

$$(\nu a)(m\,[P]) \mid \overline{c}^{*}\{m\} \mid c^{*}\{m_1, m_2, \ldots, m_n\} \longrightarrow (\nu a)(m_1\,[P] \mid \ldots \mid m_n\,[P])\ .$$

Dans le premier cas, après réduction, le terme contient n noms a distincts et privés, alors que dans le deuxième cas, toutes les occurrences de P partagent le même nom a.

Les trois premières règles spécifient la communication : locale, transmission d'un nom vers un sous-seal ou réception d'un nom depuis un sous-seal. L'extrusion du nom b privé lorsqu'une communication traverse les barrières d'un seal, n'est pas gérée par la congruence structurelle mais directement par la règle R-CommUp. Les trois règles suivantes traitent le cas du déplacement ou/et de la copie d'un seal, qu'elles soient locales, vers un sous-seal ou depuis un sous-seal.

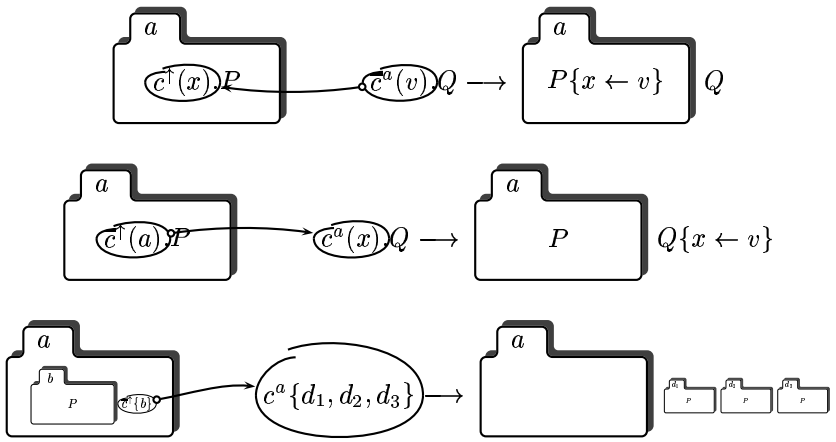


FIG. 4.3 – Quelques communications et mouvements dans les Seals

Notons que les règles de déplacement ne procèdent à aucune extrusion : pour qu'un seal b se déplace, il faut que tous les noms utilisés par b soient déjà connus dans le site de destination. En particulier, comme la seule forme d'extrusion qui traverse les barrières d'un site est liée à la communication par R-CommUp, pour faire monter un seal par R-MoveUp, il faut avoir préalablement communiqué tous les noms nécessaires au parent.

Enfin, les autres règles sont habituelles.

4.1.2 Exemples

Verrous

L'écriture d'un verrou est naturelle dans les Seals, et peut s'étendre trivialement à des verrous distants d'une membrane.

$$\begin{array}{lcl} \texttt{Acq}\ n.P & \triangleq & n^*(\_).P \\ \texttt{Rel}\ n.Q & \triangleq & Q \mid (\nu v : A)\overline{n}^*(v) \end{array}$$

En revanche, il n'est pas possible de généraliser simplement pour permettre de relâcher un verrou à distance supérieure. En effet, pour transmettre un message à distance dans les Seals, il est nécessaire de router le message, contrairement aux ambients dans les MA, qui peuvent se déplacer sans intervention de l'environnement.

Synchronisation, communication sur un canal

La synchronisation et l'émission sur un canal sont des opérations primitives dans les Seals, que nous ne détaillerons donc pas.

$$\text{R-CommLocal}\ \overline{a}^*(b).P \mid a^*(c).Q \longrightarrow P \mid Q\{c \leftarrow b\}$$

$$\text{R-CommDown}\ \frac{a \notin \overrightarrow{z} \qquad b \notin \overrightarrow{z}}{\begin{array}{r}\overline{a}^m(b).P \mid m\left[(\nu\overrightarrow{z})(a^\uparrow(c).Q \mid R)\right] \qquad \longrightarrow \\ P \mid m\left[(\nu\overrightarrow{z})(Q\{c \leftarrow b\} \mid R)\right]\end{array}}$$

$$\text{R-CommUp}\ \frac{a \notin \overrightarrow{z}}{\begin{array}{r}a^m(c).P \mid m\left[(\nu\overrightarrow{z})(\overline{a}^\uparrow(b).Q \mid R)\right] \qquad \longrightarrow \\ (\nu\overrightarrow{z} \cap \{b\})(P\{c \leftarrow b\} \mid m\left[(\nu\overrightarrow{z}\backslash\{b\})(Q \mid R)\right])\end{array}}$$

$$\text{R-MoveLocal}\ \overline{a}^*\{b\}.P \mid a^*\{c_1, \ldots, c_n\}.Q \mid b\,[R] \longrightarrow P \mid Q \mid c_1\,[R] \mid \ldots \mid c_n\,[R]$$

$$\text{R-MoveDown}\ \frac{a \notin \overrightarrow{z} \qquad fv(S) \cap \overrightarrow{z} = \emptyset}{\begin{array}{r}\overline{a}^m\{b\}.P \mid m\left[(\nu\overrightarrow{z})(a^\uparrow\{c_1, \ldots, c_n\}.Q \mid R)\right] \mid b\,[S] \qquad \longrightarrow \\ P \mid m\left[(\nu\overrightarrow{z})(Q \mid R \mid c_1\,[S] \mid \ldots \mid c_n\,[S])\right]\end{array}}$$

$$\text{R-MoveUp}\ \frac{a \notin \overrightarrow{z} \qquad fv(S) \cap \overrightarrow{z} = \emptyset}{\begin{array}{r}a^m\{c_1, \cdots, c_n\}.P \mid m\left[(\nu\overrightarrow{z})(\overline{a}^\uparrow\{b\}.Q \mid R \mid b\,[S])\right] \qquad \longrightarrow \\ P \mid c_1\,[S] \mid \ldots \mid c_n\,[S] \mid m\left[(\nu\overrightarrow{z})(Q \mid R)\right]\end{array}}$$

$$\text{R-Rec}\ \texttt{rec}\ X.P \longrightarrow P\{X \leftarrow \texttt{rec}\ X.P\} \qquad \text{R-Res}\ \frac{P \longrightarrow Q}{(\nu n)P \longrightarrow (\nu n)Q}$$

$$\text{R-Seal}\ \frac{P \longrightarrow Q}{a\,[P] \longrightarrow a\,[Q]} \qquad \text{R-Par}\ \frac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$$

$$\text{R-Struct}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \equiv Q}$$

FIG. 4.4 – Réduction dans les Seals.

Le protocole du taxi

La figure 4.5 présente une implantation possible du protocole du taxi à l'aide des Seals. Le fonctionnement est très différent de celui des versions précédentes et probablement moins intuitif, à cause du système des mouvements purement objectifs.

Un client est ici défini par deux seals. L'un de ces sites porte le nom prédéfini *caller* et sert uniquement à annoncer que le client désire appeler un taxi. L'autre site porte le nom – unique et initialement privé – du client et contient les instructions supplémentaires. Ce mini-protocole permet au client, dont le nom est initialement inconnu, de se déclarer au site, ce qui l'autorisera ensuite à communiquer directement – un mécanisme qu'on pourrait comparer à une version extrêmement simplifiée des Request Brokers tels que CORBA, COM ou XPCOM [65].

$$
\begin{array}{lcl}
\texttt{Client}\ to & = & (\nu c : A_{client})(caller\left[\overline{call}^{\uparrow}(c)\right] \mid c\left[\overline{dest}^{\uparrow}(to).\overline{name}^{\uparrow}(c)\right]) \\
\texttt{Fetcher}\ s & = & \texttt{rec}\ X.(\nu coming : C_{coming})(\nu busy : A_{cab}) \\
 & & \quad call^{caller}(x).\overline{call}^{\uparrow}(coming).coming^{\uparrow}\{busy\}. \\
 & & \quad \overline{fetch}^{busy}\{x\}.\overline{ext}^{\uparrow}(busy).\overline{goto}^{\uparrow}\{busy\}. \\
 & & \quad X \\
\texttt{Dropper}\ s & = & \texttt{rec}\ X.(\nu busy : A_{cab}) \\
 & & \quad arrive^{\uparrow}\{busy\}.name^{busy}(n).drop^{busy}\{n\}. \\
 & & \quad \overline{ext}^{\uparrow}(busy).\overline{return}^{\uparrow}\{busy\}. \\
 & & \quad X \\
\texttt{Getter}\ s & = & \texttt{rec}\ X.call^{s}(coming).\overline{coming}^{s}\{ready\}.X \\
\texttt{Gotoer}\ s & = & \texttt{rec}\ X.(\nu transit : A_{cab}). \\
 & & \quad goto^{s}\{transit\}.dest^{transit}(to).\overline{arrive}^{to}\{transit\}. \\
 & & \quad X \\
\texttt{Returner}\ s & = & \texttt{rec}\ X.return^{s}\{ready\}.X \\
\texttt{Router}\ s & = & \texttt{rec}\ X.ext^{s}(\_).X \\
\texttt{Site}\ s & = & s\,[\texttt{Fetcher}\ s \mid \texttt{Dropper}\ s \mid \ldots] \mid \\
 & & \quad \texttt{GetCaber}\ s \mid \texttt{Gotoer}\ s \mid \texttt{Returner}\ s \mid \texttt{Router}\ s \\
\texttt{Cab} & = & ready[\texttt{rec}\ X.fetch^{\uparrow}\{client\}.dest^{client}(to).\overline{dest}^{\uparrow}(to). \\
 & & \quad name^{client}(n).\overline{name}^{\uparrow}(n).\overline{drop}^{\uparrow}(client)]
\end{array}
$$

FIG. 4.5 – Une implantation du protocole du taxi en Seals.

Détails de l'implantation Une fois l'appel émis sur le canal *call*, il est routé par le processus `Fetch` jusqu'au processus `Getter`. Les processus `Getter` et `Fetch` déplacent alors un taxi libre, choisi arbitrairement, jusqu'au site, par un canal qui, pour éviter les collisions, porte un nom unique – ici, *coming*. À partir de ce moment, le taxi porte lui aussi un nom unique – ici, *busy* – toujours pour éviter les collisions.

Ensuite, le processus `Fetch` et le taxi transfèrent le client dans *busy* par le canal *fetch*. À partir de ce moment, le client s'appelle *client*. Le taxi est alors déplacé par le processus `Fetch` et le processus `Gotoer` vers l'extérieur du site, sous un nouveau nom unique (ici, *transit*).

Le client émet sur le canal *dest* le nom de son site de destination, qui est routé par le taxi jusqu'au processus `Gotoer`. À l'aide d'une communication sur le canal *arrive*, `Gotoer` et `Dropper` se synchronisent et déplacement le taxi jusqu'au site de destination, sous un nouveau nom unique (ici, *busy*). Le client émet sur le canal *name* son propre nom original, qui est routé par le taxi jusqu'à `Dropper`. Le taxi et `Dropper` déplacent alors le client et le déposent, avec son nom original, dans le site.

Enfin, les processus `Dropper` et `Returner` bougent le taxi hors du site et le renomment *ready*, pour marquer qu'il est de nouveau disponible.

Dans toute cette implantation processus `Router` s se charge de l'extrusion des noms. Ainsi, le nom de chaque seal qui doit sortir d'une localité est transmis à l'extérieur de cette localité par le canal *ext*. Même si le nom lui-même est ignoré, cela permet de forcer l'extrusion, de manière à permettre l'utilisation de

$$
\begin{array}{lcl}
\texttt{Thread} & = & \texttt{rec}\ X.(\nu x : A_x) req^{\uparrow}\{x\}.dest^{r}(to).\overline{gc}^{*}\{r\} \mid \\
 & & \quad gc^{*}\{\}.(\nu y : A_y)(y\left[\texttt{rec}\ Y.\overline{dest}^{\uparrow}(to)\right] \mid \overline{reply}^{\uparrow}\{y\}.X) \\
\texttt{Route} & = & \texttt{rec}\ X.(\nu y : A_r) reply^{s}\{y\}.dest^{y}(to).\overline{reply}^{to}\{y\}.X \\
\texttt{Server} & = & s\left[\underbrace{\texttt{Thread} \mid \texttt{Thread} \mid \ldots \mid \texttt{Thread}}_{k \text{ instances}}\right] \mid \texttt{Route} \\
\texttt{Client}\ c & = & c\left[request\left[\overline{dest}^{\uparrow}(c)\right] \mid \overline{req}^{\uparrow}\{request\}\right] \mid \\
 & & \quad \texttt{rec}\ X.(\nu x : A_x) req^{c}\{x\}.\overline{req}^{s}\{x\}.X
\end{array}
$$

FIG. 4.6 – Protocole client-serveur minimal – version Seals.

R-MOVEUP.

Discussion L'implantation du protocole est plus verbeuse que ce qu'on a pu voir dans les MA, dans BoCa ou dans les CA. Elle est aussi, peut-être, moins intuitive, car tous les mouvements sont objectifs – plutôt que d'écrire un processus qui exprime "le taxi bouge du site à la ville", on répartit les actions qui président à son déplacement entre son lieu de départ ("le site route le taxi vers l'extérieur sur le canal *goto*") et son lieu d'arrivée ("la ville reçoit le taxi sur le canal *goto* sous le nom *transit*").

En revanche, considérer les mouvements comme objectifs plutôt que subjectifs permet de modéliser de manière plus réaliste de nombreux systèmes réels. Ainsi, contrairement à ce que l'on décrit dans les Mobile Ambients, par exemple, un message ne se déplace pas spontanément d'un ordinateur vers le réseau et sur le réseau vers un autre ordinateur : le message est en fait déplacés par des actions extérieures (le système d'exploitation, la carte réseau, les routeurs ...).

Notons aussi que le client n'arrive pas complet à sa destination. En effet, le seal nommé *caller*, utilisé pour initier le protocole, n'est jamais recréé dans le site d'arrivée. Pour permettre au client de bouger de nouveau, nous aurions besoin de compliquer le terme.

Une autre différence, plus subtile, concerne le nom c. En effet, au début du protocole, ce nom est privé. Il est immédiatement transmis du client à son environnement, ce qui permet les étapes ultérieures – à partir de ce moment, le nom est public et ne redevient jamais privé. En effet, comme dans l'implantation MA, un processus initial de la forme $s\,[(\nu c)P]$ conduit après réduction à un autre processus de la forme $(\nu c)s\,[Q]$. Or, dans les Seals, il n'existe pas de mécanisme qui permette de revenir de la deuxième forme à la première. Si cette modification dans la position du (νc) ne change rien à l'exécution du protocole, on pourra tout de même noter que des informations ont été perdues au fil de l'exécution.

Client-serveur

La figure 4.6 présente une implantation Seals du protocole client-serveur minimal. Sans entrer dans le détail, cette implantation diffère des précédentes essentiellement par la nécessité de router les messages de manière plus complexe

$$
\begin{array}{lcll}
A & ::= & Seal(s,e) & s \in \mathbf{N} \cup \{\infty\}, e \in \mathbf{N} \\
U & ::= & Proc(t) & t \in \mathbf{N} \cup \{\infty\} \\
C & ::= & MChan(A) & \\
 & | & Name(W) & \\
W & ::= & A & \\
 & | & C &
\end{array}
$$

FIG. 4.7 – Grammaire de types pour le contrôle des ressources – Seals

et par nettoyage des requêtes devenues inutiles, à l'aide du canal de migration gc.

Nous pouvons remarquer qu'il serait possible d'attaquer ce système en supprimant s à l'aide d'une adaptation de $\texttt{Rogue}_1$ ou de se faire passer pour le serveur à l'aide d'une adaptation de $\texttt{Rogue}_2$. De plus, une requête boguée peut refuser de délivrer sa destination et ainsi bloquer un `Thread`. En revanche, en l'absence de la nécessité d'ouvrir un ambient sans connaître son contenu et grâce au verrouillage strict des communications, des attaques par le biais de Chevaux de Troie telles que $\texttt{Rogue}_3$ et $\texttt{Rogue}_4$ sont impossibles.

4.1.3 Des types pour contrôler les ressources

Là où le mécanisme des cocapacités permet de contrôler l'allocation et la désallocation dans les Controlled Ambients, les déplacements le long de canaux accomplit la même tâche dans les Seals. Nous pouvons donc définir, comme dans les CA, une notion de ressource et des méthodes pour prouver qu'un système est ressource-contrôlé.

Les définitions de politique d'allocation des ressources, d'utilisation de ressources et de termes ressource-contrôlés s'étendent naturellement des CA aux Seals. Tout au plus est-il nécessaire de remplacer le cas $res_\Gamma(M.P) = 0$ par $res_\Gamma(\alpha.P) = 0$ et de supprimer le cas $res_\Gamma((x : A).P) = res_\Gamma(\langle M \rangle P) = 0$, inutile dans les Seals.

La grammaire des types est présentée sur la figure 4.7.

Le type d'un seal est $Seal(s,e)$ lorsque s est la taille de sa réserve et e son encombrement. Le type d'un processus P est $Proc(t)$ lorsque l'effet de P est t, c'est-à-dire si P n'a pas besoin de plus de t ressources pour s'exécuter. Un canal c peut avoir le type $MChan(A)$ ou le type $IChan(W)$. Le canal c a le type $MChan(A)$ si c sert à déplacer des seals de type A ou $IChan(W)$ si c sert à communiquer des noms de type W.

La définition des environnements de typage est une extension naturelle de celle qui a cours dans les CA.

Règles de typage

Les règles de typage sont présentées sur la figure 4.8. Les règles T-NIL, T-RES, T-PROCNAME, T-REC, T-SEAL et T-PAR sont essentiellement identiques à leurs contreparties des Controlled Ambients desquelles on aurait supprimé le typage des message.

$$\text{T-Nil}\ \Gamma \vdash 0 : Proc(t) \qquad \text{T-Res}\ \frac{\Gamma, n : W \vdash P : U}{\Gamma \vdash (\nu n : W)P : U} \qquad \text{T-Name}\ \frac{\Gamma(n) = W}{\Gamma \vdash n : W}$$

$$\text{T-ProcName}\ \frac{\Gamma(X) = Proc(t)}{\Gamma \vdash X : Proc(u)}\ u \geq t$$

$$\text{T-Rec}\ \frac{\Gamma, X : Proc(t) \vdash P : Proc(t)}{\Gamma \vdash \texttt{rec}\ X.P : Proc(u)}\ u \geq t$$

$$\text{T-Push}\ \frac{\begin{array}{c}\Gamma \vdash P : Proc(t) \qquad \Gamma \vdash c : MChan(S) \\ \Gamma \vdash z_1 : S \qquad \cdots \qquad \Gamma \vdash z_n : S \qquad S = Seal(\_, e)\end{array}}{\Gamma \vdash c^{\eta}\{z_1, \cdots, z_k\}.P : Proc(u)}\ k \geq 0,\ u \geq t + k \cdot e$$

$$\text{T-Pop}\ \frac{\begin{array}{c}\Gamma \vdash P : Proc(t) \\ \Gamma \vdash c : MChan(S) \qquad \Gamma \vdash z : S \qquad S = Seal(\_, e)\end{array}}{\Gamma \vdash \overline{c}^{\eta}\{z\}.P : Proc(u)}\ u + e \geq t$$

$$\text{T-Rcv}\ \frac{\Gamma, z : W \vdash P : T \qquad \Gamma \vdash c : IChan(W)}{\Gamma \vdash c^{\eta}(z).P : T}$$

$$\text{T-Snd}\ \frac{\Gamma \vdash P : T \qquad \Gamma \vdash c : IChan(W) \qquad \Gamma \vdash z : W}{\Gamma \vdash \overline{c}^{\eta}(z).P : T}$$

$$\text{T-Seal}\ \frac{\Gamma \vdash m : Seal(s, e) \qquad \Gamma \vdash P : Proc(s)}{\Gamma \vdash m\,[P] : Proc(t)}\ e \leq t$$

$$\text{T-Par}\ \frac{\Gamma \vdash P : Proc(t_P) \qquad \Gamma \vdash Q : Proc(t_Q)}{\Gamma \vdash P \mid Q : Proc(u)}\ u \geq t_P + t_Q$$

FIG. 4.8 – Règles de typage pour le contrôle des ressources dans les Seals.

La mobilité est gérée par les règles T-Push et T-Pop. La première type $c^{\eta}\{z_1, \ldots, z_k\}$, qui alloue la place pour k copies d'un même seal. Il est donc nécessaire de réserver $k \cdot e$ ressources, où e est l'encombrement du seal. De plus, les noms z_1, ..., z_n doivent déjà être dans l'environnement avec le même type que celui du seal dupliqué, donné par le type du canal. De même, la deuxième règle type $\overline{c}^{\eta}\{z\}$, ce qui nécessite que le canal et z portent le même type et ce qui libère e ressources.

Enfin, les règles T-Rcv et T-Snd permettent de typer la communication d'informations, d'une manière conforme à l'intuition.

Propriétés

De la même manière que dans les CA, nous avons les propriétés suivantes.

Lemme 7 (Sous-typage)

Soient Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P : Proc(t)$. *Alors, pour tout* $u \geq t$, *on aura* $\Gamma \vdash P : Proc(u)$.

Ici aussi, le lemme est trivial. Cette propriété est nécessaire notamment pour les règles T-Pop et T-Seal.

Corollaire 2 (Type minimal) *Soient* Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P : Proc(t)$. *Alors, il existe* u *minimal tel que* $\Gamma \vdash P : Proc(u)$.

Ce lemme est trivial.

Lemme 8 (Renforcement – noms de seals)
Si $\Gamma, n : A \vdash P : U$ *et* $n \notin fv(P)$, *alors* $\Gamma \vdash P : U$.

Lemme 9 (Renforcement – noms de canaux)
Si $\Gamma, n : C \vdash P : U$ *et* $n \notin fv(P)$, *alors* $\Gamma \vdash P : U$.

Lemme 10 (Renforcement – noms de processus)
Si $\Gamma, X : U \vdash P : V$ *et* $X \notin fv(P)$, *alors* $\Gamma \vdash P : V$.

Lemme 11 (Affaiblissement – noms de seals)
Si $\Gamma \vdash P : U$ *et* $n \notin fv(P)$ *alors* $\Gamma, n : A \vdash U$.

Lemme 12 (Affaiblissement – noms de canaux)
Si $\Gamma \vdash P : U$ *et* $n \notin fv(P)$ *alors* $\Gamma, n : C \vdash U$.

Ces lemmes sont traditionnels, tout comme les preuves.

Lemme 13 (Les bons types respectent la politique)
Si $\Gamma \vdash P : U$, *alors* $res_\Gamma(P)$ *est défini.*

La preuve, simple, est présentée dans l'annexe B.1. Il s'agit, de nouveau, d'une induction sur la structure d'un certificat de $\Gamma \vdash P : U$.

Théorème 4 (Subject Reduction) *Si* $\Gamma \vdash P : U$ *et si* $P \longrightarrow Q$, *alors* $\Gamma \vdash Q : U$.

La preuve est présentée, elle aussi, dans l'annexe B.2.

Théorème 5 (Contrôle des ressources) *Soient* Γ *environnement et* P *un processus. Si* P *est typable dans* Γ, *alors* P *est ressource-contrôlé dans* Γ.

Il s'agit d'un corollaire direct de ce qui précède.

4.1.4 Exemples

Verrous

Considérons les verrous définis à la section 4.1.2 et un environnement Γ tel que $\Gamma \vdash P : Proc(t_P)$, $\Gamma \vdash Q : Proc(t_Q)$ et $\Gamma \vdash n : IChan(A)$.

Typage de P

$\Gamma \vdash P :$		$Proc(t_P)$	Par hypothèse
	Comme	$x \notin fv(P)$	
$\Rightarrow \Gamma, x : A \vdash P :$		$Proc(t_P)$	Par *Affaiblissement*
Typage de `Acq` $n.P$			
$\Gamma, x : A \vdash P :$		$Proc(t_P)$	Cf. plus haut
$\Gamma \vdash n :$		$IChan(A)$	Par hypothèse
$\Rightarrow \Gamma \vdash n^*(x).P :$		$Proc(t_P)$	Par T-Rcv
Typage de n			
$\Gamma \vdash n :$		$IChan(A)$	Par hypothèse
$\Rightarrow \Gamma, v : A \vdash n :$		$IChan(A)$	Par *Affaiblissement*
Typage de $\overline{n}^*(v)$			
$\Gamma, v : A \vdash \mathbf{0} :$		$Proc(0)$	Par T-Nil
$\Gamma, v : A \vdash v :$		A	Par T-Res
$\Gamma, v : A \vdash n :$		$IChan(A)$	Cf. plus haut
$\Rightarrow \Gamma, v : A \vdash \overline{n}^*(v).\mathbf{0} :$		$Proc(0)$	Par T-Snd
Typage de $(\nu v : A)\overline{n}^*(v)$			
$\Gamma, v : A \vdash \overline{n}^*(v).\mathbf{0} :$		$Proc(0)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu v : A)\overline{n}^*(v).\mathbf{0} :$		$Proc(0)$	Par T-Res
Typage de `Rel` $n.Q$			
$\Gamma \vdash Q :$		$Proc(t_Q)$	Par hypothèse
$\Gamma \vdash (\nu v : A)\overline{n}^*(v).\mathbf{0} :$		$Proc(0)$	Cf. plus haut
$\Rightarrow \Gamma \vdash Q \mid (\nu v : A)\overline{n}^*(v) :$		$Proc(t_Q)$	Par T-Par

○

Une nouvelle fois, le type de `Acq` $n.P \mid$ `Rel` $n.Q$ est donc le même que le type de $P \mid Q$.

Le protocole du taxi

Ici encore, nous cherchons à prouver que chaque taxi peut contenir au plus un seul passager à chaque instant. Pour spécifier cette propriété sous forme de types, nous considérons que chaque passager encombre une ressource et que chaque taxi n'a qu'une ressource en réserve. Nous pouvons donc formuler la politique de contrôle de ressources suivante :

$$\begin{cases} A_{client} & = Seal(0,1) \\ A_{cab} & = Seal(1,0) \\ A_{site} & = Seal(\infty,0) \\ \Gamma(caller) & = Seal(0,0) \\ C_{coming} & = \Gamma(goto) = \Gamma(arrive) = \Gamma(return) = MChan(A_{cab}) \\ \Gamma(call) & = \Gamma(name) = IChan(A_{client}) \\ \Gamma(fetch) & = \Gamma(drop) = MChan(A_{client}) \\ \Gamma(dest) & = IChan(A_{site}) \\ \Gamma(ready) & = A_{cab} \\ \Gamma(to) & = A_{site} \\ \Gamma(ext) & = IChan(A_{cab}) \end{cases}$$

Il est alors possible de typer le protocole du taxi. Le protocole du taxi est donc ressource-contrôlé dans le cadre de Γ. En particulier, cela signifie que chaque taxi ne peut contenir plus d'un client à la fois.

Comme dans les CA, nous pouvons aussi prouver, avec d'autres politiques de contrôle des ressources, qu'un taxi ne contiendra jamais un site, qu'un client ne contiendra jamais un taxi...

Pour des raisons de place, nous ne fournissons pas la preuve du typage.

4.1.5 Discussion

Le calcul des Seals est un formalisme conçu pour apporter la notion de mouvements objectifs dans des systèmes décrits par des hiérarchies similaires à celles des ambients. Comme les CA, le calcul en lui-même ne permet pas de spécifier des propriétés de ressources mais offre la possibilité de réagir aux allocations et aux désallocations et donc d'écrire des systèmes conscients des ressources. La représentation des ressources et des politiques de contrôle de ces ressources est entièrement statique et se fait grâce au système de types que nous avons proposé.

Comparaison De manière assez intéressante, nous pouvons remarquer que les communications le long de canaux sont apparentées aux migrations à l'aide de cocapacités. En effet, si les canaux imposent de donner un nom aux seals, $\overline{c}^{\uparrow}\{r\}$ et $\overline{\texttt{out}}^{\uparrow}\ r$ diffèrent essentiellement par le fait que l'émigration elle-même est nommée. De même, $c^{m}\{r\}$ impose un contrôle plus strict que $\overline{\texttt{in}}^{\uparrow}\ r$ mais reste une autorisation d'immigration. De nouveau, après avoir expérimenté les deux mécanismes, il s'avère que les canaux de migration permettent un contrôle encore plus précis, sous la forme de protocoles plus stricts, aussi bien statiquement que dynamiquement mais imposent, une fois de plus, une syntaxe plus lourde. Si le dialecte des Seals que nous avons présent manque de généralité, comme les CA, il faut savoir qu'il existe un dialecte légèrement différent des Seals dans lequel il est possible de communiquer avec un seal enfant sans connaître son nom – au prix d'une certaine perte rigueur de quelques cas d'interférences et de certains risques de sécurité.

Notons sinon que les mécanismes d'allocation et de désallocation des ressources sont proches de ceux des CA plus que de BoCa.

Bilan Le langage des Seals nous a permis de modéliser des protocoles non-triviaux. Si le formalisme est, de notre point de vue, plus simple que celui des CA, notamment grâce à l'absence des capacités et surtout de la possibilité de dissoudre un ambient, il nous a de plus semblé à même de représenter plus de systèmes que les MA ou CA. En particulier, les canaux de communication et de migration, malgré la lourdeur associée à ces derniers, sont des opérations qui nous semblent à la fois plus générales, plus simples et plus réalistes que l'ouverture ou les communications anonymes.

Quant au système de types des Seals, il s'agit essentiellement d'une version simplifiée de celui des CA. Notons cependant que, contrairement au système des CA, nous n'avons pas restreint arbitrairement la communication, et que le résultat reste assez puissant pour représenter de nombreuses politiques de contrôle de ressources – peut-être même plus qu'avec celui des CA, à notre intuition, grâce aux communications par des canaux. De la même manière que dans les CA, les règles ne font appel qu'à des inéquations linéaires sur des entiers positifs ou infinis, qui sont donc probablement solubles aisément à l'aide

de techniques habituelles de programmation linéaire. Cela nous permettrait, ici aussi, de rendre l'analyse et l'inférence totalement automatiques.

4.2 Ordre supérieur : les Kells

Après avoir expérimenté nos méthodes en présence de mouvements objectifs, nous nous intéressons dans cette section au problème du contrôle des ressources dans un langage bien plus complexe, qui dispose de mouvements objectifs : le calcul des Kells.

Le calcul des Kells [77] est proche de ceux des Mobile Ambients ou des Seals par son système de localités et de processus. À l'inverse de ces formalismes, cependant, les primitives des Kells sont explicitement d'ordre supérieur. Ainsi, une opération, en plus de transférer des noms ou des sites, comme dans les Seals, peut extraire un processus d'un site, le modifier, le dupliquer... De plus, là où les communications des Seals se font entre deux processus présents dans le même site ou séparés par une seule membrane, le long de canaux, les synchronisations des Kells peuvent se faire entre un nombre arbitraire de parties et autorisent de dépasser autant de membranes que nécessaire, sous certaines conditions. Enfin, certains aspects de la sémantique des communications dans les Kells sont paramétrés par un langage externe de motifs.

À cause de cette nature d'ordre supérieur, le calcul des Kells traite de manière proche les *messages* – qui ressemblent à l'émission de messages le long de canaux dans le calcul des Seals ou le π-calcul – et les *cellules* – c'est-à-dire les sites. Le mécanisme de déplacement, objectif comme dans les Seals, qui consiste à lire le contenu d'une cellule et à le réécrire dans une autre cellule, permet aussi de *passiver* une cellule, c'est-à-dire de la transformer en un message de même contenu, ou d'*activer* un message, c'est-à-dire de le transformer en cellule.

Plus encore que celui des Seals, le calcul des Kells permet de représenter naturellement les processus légers et les exceptions.

4.2.1 Langage

Comme le langage des Kells est actuellement en pleine évolution, il n'existe pas encore de sémantique de référence. Nous présentons donc ici la dernière version considérée comme stable [75], encore non publiée au moment de la rédaction de ce document.

Plutôt que de décrire rigoureusement l'intégralité du calcul, nous nous présenterons les aspects qui nous semblent nécessaires à la compréhension de l'expressivité du langage et des difficultés liées à cette richesse.

Syntaxe

La syntaxe des Kells est présentée sur la figure 4.9. On suppose l'existence d'un ensemble a, b, c, m, $n\ldots$ de noms. On emploiera P, $Q\ldots$ pour désigner les processus et ξ pour désigner les motifs.

Les processus $\mathbf{0}$, $P|Q$ et $(\nu n : N)P$ se comprennent comme dans les calculs précédents. Le processus $\xi \triangleright P$ est un *déclencheur*, préfixé par le *motif* ξ – lorsque les conditions requises par ξ sont remplies, P est déclenché. Le processus $n\,[P]\,.Q$ désigne une cellule nommée n, contenant le processus (actif) P et qui, si elle

$$
\begin{array}{rcl}
P, Q & ::= & \mathbf{0} \\
& | & P|Q \\
& | & \xi \rhd P \\
& | & (\nu n : N)P \\
& | & n\,[P]\,.Q \\
& | & n\langle P\rangle.Q \\
& | & n
\end{array}
$$

FIG. 4.9 – Syntaxe du calcul des Kells.

est consommée par un déclencheur, libère la continuation Q. De même, $n\langle P\rangle.Q$ désigne un message nommé n, contenant le processus (passif) P et qui, si il est consommé par un déclencheur, libère la continuation Q. Enfin, un nom n est un processus valide.

Les motifs ξ sont décrits dans un langage laissé comme paramètre des Kells. Pour les exemples, nous emploierons une syntaxe proche de celle des processus, dans lesquels les noms entre parenthèses sont les variables. Ainsi, le motif $a\,[(x)]$ accepte le processus $a\,[P]\,.Q$ avec $x \leftarrow P$, consomme $a\,[P]$ et déclenche la continuation Q. En particulier, le motif $a\,[(x)]$ accepte le processus $a\,[n]\,.Q$ avec $x \leftarrow n$, alors $a\,[n]$ et déclenche Q. Cependant, le processus $b\,[P]\,.Q$ est rejeté. De même, le motif $a\,[(x) \mid b\langle P\rangle]$ accepte (et consomme) $a\,[Q \mid b\langle P\rangle]$ avec $x \leftarrow Q$ aussi bien que $a\,[b\langle P\rangle]$ avec $x \leftarrow \mathbf{0}$.

Notons que la version originale du calcul était totalement asynchrone, au sens où ni les messages ni les cellules ne disposaient de continuations. Nous avions proposé [78] une extension des Kells dans laquelle les continuations étaient présentées dans un processus tiers. Ainsi, le message $a\langle x\rangle$ ne pouvait être consommé que si il apparaissait en parallèle avec un processus $a \lhd P$. La consommation libérait alors P. Si les deux manières de formuler les continuations sont essentiellement équivalentes, la méthode employée par cette version des Kells offre moins de flexibilité mais se révèle plus commode pour l'étude comportementale du calcul [75].

Nous emploierons les mêmes raccourcis syntaxiques que dans les langages précédents.

Sémantique

La sémantique du calcul des Kells se définit de manière plus complexe que dans les calculs précédents et fait intervenir une relation de congruence structurelle paramétrique, une "relation de sous-réduction" que nous ne présentons pas ici et qui permet de mettre les termes sous forme normale avant de les réduire effectivement et enfin la relation de réduction elle-même, elle aussi paramétrique.

La congruence structurelle est la plus petite congruence $\equiv$ qui fait de l'ensemble $(P/\ \equiv, |, \mathbf{0})$ un monoïde commutatif et qui satisfait les axiomes de la figure 4.10. Elle dépend d'une relation paramétrique de congruence sur les motifs, ce qui est rendu par la règle STRUCT-TRIG.

Les règles de réduction sont présentées sur la figure 4.11. Sans entrer dans tous les détails, les règles R-L et R-G résolvent les réactions entre un motif et son environnement à l'aide d'une primitive `match` paramétrique. Pour R-L, dans les trois macros auxiliaires Δ, Υ et Ψ, le premier paramètre (respectivement

$$\textsc{Struct-Zero-Res}\ (\nu n)\mathbf{0} \equiv \mathbf{0}$$

$$\textsc{Struct-Res-Res}\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \text{ si } n \neq m$$

$$\textsc{Struct-Res-Par}\ (\nu n)(P|Q) \equiv P|(\nu n)Q \text{ si } n \notin fv(P)$$

$$\textsc{Struct-Res-Kell}\ (\nu n)m\,[P]\,.Q \equiv m\,[(\nu n)P]\,.Q \qquad \text{si } n \neq m \text{ et } n \notin fv(Q)$$

$$\textsc{Struct-Trig}\ \frac{\xi \equiv \zeta}{\xi \rhd P \equiv \zeta \rhd P}$$

FIG. 4.10 – Congruence structurelle dans les Kells

U_1, U_2 et U_3) représente la situation avant la réaction, le deuxième paramètre (respectivement M_1, M_2 et M_3) représente les sous-termes consommés par la réaction et le troisième paramètre (respectivement V_1, V_2 et V_3) représente les sous-termes modifiés par cette consommation – c'est-à-dire essentiellement les continuations libérées. La macro Δ se spécialise dans le traitement des messages au même niveau que le déclencheur ou au niveau supérieur, la macro Υ gère les cellules au même niveau que le déclencheur et la macro Ψ se charge des messages imbriqués dans des sous-cellules. Les constructions ab, dn, loc sont des annotations qui permettent au motif de distinguer, parmi les sous-termes consommés, respectivement, les messages venus du père, les messages venus d'un enfant, et les messages originaires des cellules locales. Enfin, la primitive `match`, liée au langage de motifs, est un paramètre qui détermine dans quel cas un motif accepte un processus particulier et dont la valeur, lorsqu'elle est définie, est une substitution qui associe à chaque variable du motif un processus de $M_1 \mid M_2 \mid M_3$.

La relation de sous-réduction intervient uniquement sous la forme de la préconditon $U_2 \not\rightarrow$, dans R-L et R-G. Pour éviter d'entrer dans trop de détails techniques, nous nous contenterons de résumer informellement cette précondition : toutes les créations de nouveaux noms dans U_2 doivent apparaître le plus haut possible dans l'arbre syntaxique. Cela permet notamment d'éviter toutes les ambiguïtés lors de la réduction d'un terme tel que $A \stackrel{\Delta}{=} (a\,[(x)] \rhd (x|x)) \mid a\,[(\nu b).P]$. En effet, pour mettre U_2 – c'est-à-dire $a\,[(\nu b).P]$ – sous forme normale, il faut faire sortir (νb) de a, quitte à procéder à une α-conversion de b si $a = b$. Le terme effectivement réduit sera donc de la forme $(\nu b)((a\,[(x)] \rhd (x|x)) \mid a\,[P])$ et la réduction n'introduira donc pas deux (νb) distincts.

Cette notion de forme normale est à mettre en relation avec les mécanismes employés dans les Seals pour gérer l'élargissement du champ de définition des (ν) (cf. figure 4.4).

Contextes d'évaluation $\mathbf{E} ::= \{\cdot\} \mid \mathbf{E}|P \mid (\nu a)\mathbf{E} \mid a\,[\mathbf{E}]$

$$\text{R-L}\ \frac{\begin{array}{c}\xi \neq \emptyset \qquad \theta = \mathtt{match}(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3)) \\ \Delta(U_1, M_1, V_1) \qquad \Upsilon(U_2, M_2, V_2) \qquad \Psi(U_3, M_3, V_3) \qquad U_2 \not\rightarrow\end{array}}{(\xi \rhd P)|U_1|U_2|U_3 \longrightarrow P\theta|V_1|V_2|V_3}$$

$$\text{R-G}\ \frac{\begin{array}{c}\xi \neq \emptyset \\ \theta = \mathtt{match}(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3) \mid ab(M_4)) \qquad \Delta(U_1, M_1, V_1) \\ \Upsilon(U_2, M_2, V_2) \qquad \Psi(U_3, M_3, M_k) \qquad \Delta(U_4, M_4, V_4) \qquad U_2 \not\rightarrow\end{array}}{b\,[(\xi \rhd P)|U_1|U_2|U_3|R]\,.T|U_4 \longrightarrow b\,[P\theta|V_1|V_2|V_3|R]\,.T|V_4}$$

$$\text{R-Struct}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \equiv Q} \qquad \text{R-Context}\ \frac{P \longrightarrow Q}{\mathbf{E}\{P\} \longrightarrow \mathbf{E}\{Q\}}$$

$$\Delta(U, M, V) \triangleq \left\{\begin{array}{lcl} U & = & \Pi_{j\in J} a_j\langle P_j\rangle.Q_j \\ M & = & \Pi_{j\in J} a_j\langle P_j\rangle \\ V & = & \Pi_{j\in J} Q_j \end{array}\right.$$

$$\Upsilon(U, M, V) \triangleq \left\{\begin{array}{lcl} U & = & \Pi_{j\in J'} a_j\,[P_j]\,.Q_j \\ M & = & \Pi_{j\in J'} a_j\,[P_j] \\ V & = & \Pi_{j\in J'} Q_j \end{array}\right.$$

$$\Psi(U, M, V) \triangleq \left\{\begin{array}{lcl} U & = & \Pi_{j\in J''} a_j\langle R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i\rangle.Q_j \\ M & = & \Pi_{j\in J''}\Pi_{i\in I_j} a_i\langle P_i\rangle^{\downarrow a_j} \\ V & = & \Pi_{j\in J''} a_j\,[R_j \mid \Pi_{i\in I_j} S_i]\,.Q_j \end{array}\right.$$

FIG. 4.11 – Réduction dans les Kells

4.2.2 Exemples

Un langage de motifs

Avant de pouvoir écrire des termes dans le calcul des Kells, nous devons introduire un langage de motifs. Le langage que nous utiliserons est une version légèrement retouchée de l'exemple jK donné par les auteurs du calcul et globalement comparable aux join-patterns du Join-Calcul [29].

La grammaire de notre langage de motifs est présentée sur la figure 4.12. Les noms entre parenthèses sont les noms qui seront substitutués par θ dans les règles R-L et R-G – dans notre grammaire, ils sont annotés par des informations que nous utiliserons lorsqu'il s'agira de typer des termes du langage des Kells. Nous dirons de ces noms qu'ils sont *liés* par ξ. La fonction `match` est définie par induction, à l'aide des règles de la figure 4.13.

Nous illustrons le fonctionnement de ce langage de motifs dans les exemples suivants.

$$
\begin{array}{rcl}
\xi & ::= & a\,[(x:T)] \\
& | & a\langle\overline{\rho}\rangle \\
& | & a\langle\overline{\rho}\rangle^{\uparrow} \\
& | & a\langle\overline{\rho}\rangle^{\downarrow} \\
& | & \xi|\xi
\end{array}
\qquad
\begin{array}{rcl}
\overline{\rho} & ::= & (x:T) \\
& | & n \\
& | & \rho \\
& | & (a:T)\langle\overline{\rho}\rangle \\
& | & \epsilon
\end{array}
\qquad
\begin{array}{rcl}
\rho & ::= & a\langle\overline{\rho}\rangle \\
& | & \rho|\rho
\end{array}
$$

FIG. 4.12 – Un langage de motifs pour les Kells.

$$
\begin{array}{lcl}
\texttt{match}(\xi, \alpha(M_1|M_2)) & \triangleq & \texttt{match}(\xi, \alpha(M_2)|\alpha(M_2)) \\
\texttt{match}(\xi_1|\xi_2, \alpha(M_1)|\beta(M_2)) & \triangleq & \texttt{match}(\xi_1, \alpha(M_1)) \oplus \texttt{match}(\xi_2, \beta(M_2)) \\
 & & \quad \text{pour toutes annotation } \alpha, \beta \\
\texttt{match}(a\,[(x:\_)]\,, loc(a\,[P])) & \triangleq & \{x \leftarrow P\} \\
\texttt{match}(a\langle\overline{p}\rangle, loc(a\langle P\rangle)) & \triangleq & \texttt{match}(\overline{p}, loc(P)) \\
\texttt{match}(a\langle\overline{p}\rangle^{\downarrow}, dn(a\langle P\rangle)) & \triangleq & \texttt{match}(\overline{p}, loc(P)) \\
\texttt{match}(a\langle\overline{p}\rangle^{\uparrow}, ab(a\langle P\rangle)) & \triangleq & \texttt{match}(\overline{p}, loc(P)) \\
\texttt{match}(\epsilon, loc(\mathbf{0})) & \triangleq & \{\} \\
\texttt{match}(n, loc(n)) & \triangleq & \{\} \\
\texttt{match}((x:\_), loc(P)) & \triangleq & \{x \leftarrow P\} \\
\texttt{match}((a:\_)\langle\overline{p}\rangle, loc(b\,[P])) & \triangleq & \{a \leftarrow b\} \oplus \texttt{match}(\overline{p}, P)
\end{array}
$$

$$
\xi_1|\xi_2 \equiv \xi_2|\xi_1 \qquad \rho_1|\rho_2 \equiv \rho_2|\rho_1 \qquad \frac{\overline{p} \equiv \overline{p}'}{a\langle\overline{p}\rangle\text{-} \equiv a\langle\overline{p}'\rangle\text{-}}
$$

FIG. 4.13 – Acceptation des processus et congruence structurelle de notre langage de motifs.

Verrous

L'écriture d'un verrou se fait de la même manière que dans les Seals.

$$
\begin{array}{lcl}
\texttt{Acquire}\ n.P & \triangleq & n\langle(x:T)\rangle \rhd P \text{où } x \notin fv(P) \\
\texttt{Release}\ n.Q & \triangleq & Q \mid n\langle\rangle
\end{array}
$$

En raison des déplacements objectifs, il n'est pas non plus possible de généraliser simplement à des verrous distants.

Communication sur un canal

Il est très simple d'implanter une communication sur un canal à l'aide des Kells – nous pourrions même implanter une communication polyadique sans plus de difficultés.

$$
\begin{array}{lcl}
\texttt{Send}\ M\ \texttt{on}\ n.P & = & n\langle M\rangle.P \\
\texttt{Receive}\ x\ \texttt{on}\ n.Q & = & n\langle(x:T)\rangle \rhd Q
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{Repl}(\xi, P) & \triangleq & (\nu t : A_t)\texttt{Y}(P, \xi, t) \mid t\langle \texttt{Y}(P, \xi, t)\rangle \\
\texttt{Y}(P, \xi, t) & \triangleq & \xi \mid t\langle (y : T_P)\rangle \rhd P \mid y \mid t\langle y\rangle \\
 & & \text{où } t, y \notin fv(P)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{Rec}(X, P) & \triangleq & (\nu t : A_t, u : A_u)(\texttt{Z}(P, X, t, u) \mid t\langle \texttt{Z}(P, X, t, u)\rangle) \\
\texttt{Z}(P, X, t) & \triangleq & t\langle (y : T_y)\rangle \rhd u\langle y \mid t\langle y\rangle\rangle \mid u\langle (X : T_X)\rangle \rhd P \\
 & & \text{où } t, u, y \notin fv(P)
\end{array}
$$

FIG. 4.14 – Réplication et récursion dans les Kells.

Ce qui est plus intéressant est qu'il est possible d'implanter presque aussi simplement plusieurs types de communication qui ont une sémantique identique mais des implications différentes du point de vue de la distribution.

$$
\begin{array}{lcl}
\texttt{Send}_2\ M\ \texttt{on}\ n.P & = & n\langle msg\langle M\rangle \mid then\langle P\rangle\rangle \\
\texttt{Receive}_2\ x\ \texttt{on}\ n.Q & = & n\langle msg\langle (x : T)\rangle \mid then\langle (y : Proc(t_P))\rangle\rangle \rhd Q \mid y \\
 & & \quad \text{avec } y \notin fv(Q) \\
 & & \\
\texttt{Send}_3\ M\ \texttt{on}\ n.P & = & (\nu f)(n\langle (y : T_Q)\rangle \rhd (f\langle M\rangle.P \mid (f\langle (x : T)\rangle \rhd y))) \\
 & & \quad \text{avec } y \notin fv(P) \cup fv(Q) \\
\texttt{Receive}_3\ x\ \texttt{on}\ n.Q & = & n\langle Q\rangle
\end{array}
$$

En effet, même si les processus du calcul des Kells peuvent déjà être distribés partiellement explicitement entre sites nommés, rien n'interdit de considérer que des processus qui apparaissent dans la même cellule soient aussi implicitement distribués sur plusieurs localités implicites, tout comme les processus du π-calcul peuvent être distribués explicitement [41] aussi bien qu'implicitement [33]. Ainsi, si l'on considère que l'émission et la réception sont initialement exécutées sur plusieurs processeurs distincts, les macros `Send` et `Receive` conserveront cette distribution, tandis que $\texttt{Send}_2$ et $\texttt{Receive}_2$ déplaceront toute la charge de calcul vers le récepteur et $\texttt{Send}_3$ et $\texttt{Receive}_3$ porteront la charge vers l'émetteur. En plus de faciliter la distribution implicite, ces macros peuvent servir à équilibrer la charge en ressources lors d'une communication répliquée, comme nous le verrons plus loin.

Le prix de cette distribution est qu'elle nécessite des communications d'ordre supérieur.

Réplication et récursion

Contrairement aux calculs présentés précédemment, le calcul des Kells ne propose pas de primitive pour représenter la réplication ou la récursion. Il est, cependant, possible d'approximer simplement ces deux comportements, comme indiqué sur la figure 4.14.

Le processus $\texttt{Repl}(\xi, P)$, noté $\xi \diamond P$, se comporte comme le processus $\xi \rhd P$ répliqué. Plus précisément, $\xi \diamond P$ se comporte comme une implantation de la réplication gardée, avec instanciation à la demande : si le processus $\xi \diamond P$, seul,

ne se réduit pas, on peut vérifier simplement que, si $Q \mid (\xi \rhd P) \longrightarrow R$, alors $Q \mid (\xi \diamond P) \longrightarrow^* R \mid (\xi \diamond P)$.

De même, le processus $\texttt{Rec}(X, P)$ se comporte comme $P\{X \leftarrow \texttt{Rec}(X, P)\}$. Ceci est valable que X apparaisse zéro, une ou plusieurs fois dans P, éventuellement dans une sous-cellule ou un message.

Nous utiliserons fréquemment $\diamond$ et nous reviendrons plus tard sur ces macros et sur la manière de les typer.

Le protocole du taxi

La figure 4.15 présente une implantation possible du protocole du taxi à l'aide des Kells et de notre langage de motifs.

$$
\begin{array}{lcl}
\texttt{Mov}\ n\ f\ t\ c\ k\ l & = & n\,[from\langle t\rangle \mid to\langle t\rangle \mid client\langle c\rangle \mid later\langle l\rangle \mid content\langle k\rangle] \\
\texttt{Msg}\ n\ f\ t\ c & = & msg\langle from\langle t\rangle \mid to\langle t\rangle \mid client\langle c\rangle\rangle \\
\\
\texttt{Client}\ to & = & (\nu c : A_c)c\,[call\langle name\langle c\rangle \mid dest\langle to\rangle\rangle] \\
\\
\texttt{Fetcher}\ s & = & call\langle name\langle (c : A_c)\rangle \mid dest\langle (to : A_s)\rangle\rangle^{\downarrow} \diamond \\
 & & \quad \texttt{Msg}\ c\ s\ t.(\\
 & & \quad\quad (\texttt{Mov}\ busy\ s\ (t : A_s)\ c\ \mathbf{0}\ (l : T_{later}))^{\uparrow} \mid \\
 & & \quad\quad c\,[(cont : T_{cont})] \\
 & & \quad)\rhd \texttt{Mov}\ loaded\ s\ t\ c\ client\langle cont\rangle\ l \\
\\
\texttt{Getter} & = & (\texttt{Msg}\ (c : A_c)\ (f : A_s)\ (t : A_s))^{\downarrow} \diamond \texttt{Msg}\ c\ f\ t. \\
 & & cab\,[\texttt{Msg}\ accepting\ c\ f\ t \mid later\langle (l : T_{later})\rangle] \rhd \\
 & & \quad \texttt{Mov}\ busy\ f\ t\ c\ \mathbf{0}\ l \\
\\
\texttt{Gotoer} & = & loaded\langle (x : T_{loaded})\rangle^{\downarrow} \diamond loaded\langle x\rangle \\
\\
\texttt{Dropper}\ s & = & (\texttt{Mov}\ loaded\ \_\ s\ (c : A_c)\ (k : T_c)\ (l : T_{later}))^{\uparrow} \diamond \\
 & & c\,[y] \mid empty\langle later\langle l\rangle\rangle \\
\\
\texttt{Returner}\ s & = & empty\langle later\langle (x : T_{later})\rangle\rangle \diamond cab\,[later] \\
\\
\texttt{Site}\ s & = & s\,[\texttt{Fetcher}\ s \mid \texttt{Dropper}\ s \mid \ldots] \mid \texttt{Getter} \mid \texttt{Gotoer} \\
\\
\texttt{Cab} & = & cab[\texttt{Rec}(X, \\
 & & \quad call\langle from\langle (f : A_s)\rangle | dest\langle (t : A_s)\rangle | name\langle (c : A_c)\rangle\rangle^{\uparrow} \rhd \\
 & & \quad \texttt{Msg}\ accepting\ f\ t\ c \mid later\langle X\rangle \\
 & &)]
\end{array}
$$

FIG. 4.15 – Le protocole du taxi – version Kells.

Cette implantation reste raisonnablement proche de celle que nous avons présentée pour le calcul des Seals à la section 4.1.2 avec une structure similaire de processus `Fetcher`, `Dropper`, `Getter`, `Gotoer`, `Returner`. Pour simplifier la

$$
\begin{array}{lcl}
\texttt{Client} & = & (\nu c : A_{client})c\langle dest\langle to\rangle\rangle \\
\texttt{Fetcher}\ s & = & s\langle(c)\langle dest\langle to\rangle\rangle \mid (x)\rangle \mid cab\,[]\diamond \\
 & & \quad s\langle x\rangle \mid busy\,[client\langle c\rangle \mid dest\langle to\rangle] \\
\texttt{Bringer}\ s & = & s\langle(x)\rangle \mid busy\,[client\langle(c)\rangle \mid dest\langle s\rangle]\diamond \\
 & & \quad s\langle x \mid c\langle\rangle\rangle \mid cab\,[] \\
\texttt{Cab} & = & cab\,[] \\
\texttt{Site}\ s & = & s\langle\ldots\rangle \mid \texttt{Fetcher}\ s \mid \texttt{Bringer}\ s
\end{array}
$$

FIG. 4.16 – Le protocole du taxi – une autre version Kells.

lecture, nous avons introduit deux macros supplémentaires – en fait, presque des structures de données : `Mov` et `Msg`. Ainsi, `Mov` $n\ f\ t\ c\ k\ l$ représente le taxi dans un état n, appelé depuis f, pour emmener vers t un client nommé c et de contenu k, puis continuer comme l une fois que la course sera terminée. De même, `Msg` $n\ f\ t\ c$ représente un message d'appel du taxi dans un état n, qui demande au taxi d'aller en f chercher un client nommé c puis de l'emmener en t.

En fait, certains des processus qui, dans les Seals, étaient paramétrés par le nom du site, deviennent dans cette version indépendants du site. En effet, contrairement au dialecte des Seals que nous avons présenté, le calcul des Kells permet des communications entre une cellule mère et une cellule fille sans que la mère soit obligée de mentionner le nom de la cellule fille en question. Une autre conséquence de cette différence dans le traitement des échanges est qu'un site peut communiquer avec un client alors même que le nom du client est masqué par un ν. De plus, à l'inverse du calcul des Seals, dans lequel les ν ne pouvaient que monter dans l'arbre syntaxique, dans les Kells, les ν peuvent redescendre. Ainsi, après avoir été déplacé par un taxi, un client peut donc reprendre une forme proche de sa forme initiale, (νc) y compris.

Si l'on s'autorise à s'écarter de la version Seals et à tirer avantage de l'ordre supérieur, on peut obtenir une implantation bien plus simple du protocole du taxi, présentée sur la figure 4.16. Cette version, au lieu de déplacer le client par une communication entre un processus inclus dans le site et le taxi, réécrit l'arbre syntaxique qui représente le site afin d'en supprimer le client et réécrit l'arbre syntaxique qui représente le taxi afin d'y faire apparaître une copie de ce client. De même, pour emmener le client à sa destination, un processus supprime le client du taxi et en ajoute une copie dans le site cible. Cette version, qui illustre la puissance du calcul des Kells, est par contre moins probablement réaliste en tant qu'implantation, car elle implique des modifications structurelles sur des sites alors même que la gestion de ces sites peut être distribuée entre plusieurs ordinateurs ou plusieurs processeurs.

4.2.3 Ressources

Comme le calcul des Kells conserve les principes de hiérarchie entre sites et de mobilité des sites des Mobile Ambients, nous pouvons reprendre et adapter une fois de plus la définition des ressources que nous avions introduite pour les Mobile Ambients (définition 5). Ainsi, chaque cellule et chaque message dispose d'une

réserve individuelle de ressources et occupe à son tour un certain nombre de ressources dans la cellule ou le message parent. Lorsqu'une réaction consomme une cellule, des ressources sont désallouées et lorsqu'une nouvelle cellule est créée, des ressources lui sont allouées.

Même si cette formulation est assez différente de ce que nous avons fait pour les calculs précédents, les idées restent assez proches du calcul des Seals. En effet, grâce au mécanisme des continuations, écrire $a\,[P]\,.Q$ ou $a\langle P\rangle.Q$ permet de synchroniser Q sur la désallocation des ressources occupées par a, de la même manière que $\overline{c}\text{-}\{a\}.Q$ dans les Seals. De même, pour déplacer une cellule ou un message, un déclencheur créera une nouvelle cellule ou un nouveau message, de la même manière que de nouveaux Seals peuvent être créés. Le problème majeur est posé par l'absence d'informations sur le langage de motifs paramétrique qui peut, a priori, produire n'importe quelle substitution.

Définition 11 (Ressources – Kells)
Dans les Kells,

- *chaque cellule/message dispose d'une réserve de ressources – ces ressources sont toutes identiques*
- *chaque cellule/message peut occuper zéro, une ou plusieurs ressources dans la cellule/le message parent*
- *(re)créer une cellule/un message alloue ces ressources*
- *consommer une cellule ou un message désalloue ces ressources.*

La définitions d'une politique d'allocation des ressources ou d'un terme ressource-contrôlé étend naturellement le cas des Controlled Ambients.

Définition 12 (Utilisation des ressources – Kells)
Dans une politique de contrôle des ressources Γ, l'utilisation des ressources par un processus P, notée $res_\Gamma(P)$, est définie par

- $res_\Gamma(\mathbf{0}) = 0$
- $res_\Gamma(P|Q) = res_\Gamma(P) + res_\Gamma(Q)$
- $res_\Gamma(x) = 0$
- $res_\Gamma(a\,[P]\,.Q) = e$ *si* $res_\Gamma(P) \leq s$ *et* $\Gamma(a) = (s, e)$
- $res_\Gamma(a\langle P\rangle.Q) = e$ *si* $res_\Gamma(P) \leq s$ *et* $\Gamma(a) = (s, e)$
- $res_\Gamma(\xi \rhd P) = 0$
- $res_\Gamma((\nu n : A)P) = res_{\Gamma, n:(s_A, e_A)}(P)$ *si l'annotation A spécifie que n a une réserve de taille s et un encombrement e*

et n'est pas définie dans les autres cas. Dès que $res_\Gamma(P)$ est défini, on dira que, dans l'état courant, P respecte la politique Γ.

Si la majeure partie des cas étendent naturellement les définitions vues dans le cadre des CA ou des Seals, le choix de considérer $res_\Gamma(a\langle P\rangle.Q) = e$ si $res_\Gamma(P) \leq s$ et $\Gamma(a) = (s, e)$ est arbitraire. En effet, nous supposons que le site a occupe autant de ressources qu'il soit actif (si c'est une cellule) ou passif (si c'est un message) et que le processus P nécessite autant de ressources qu'il soit lui-même actif (c'est-à-dire présent dans une cellule) ou passif (soit contenu dans un message). Il serait probablement possible de distinguer les deux cas et de typer différemment selon la nature de a. Pour des raisons de simplicité et faute d'avoir une intuition précise de la consommation en ressources d'un processus passif, nous avons préféré garder cette définition de res_Γ pour établir un premier système de types, quitte à compliquer les notions dans une version ultérieure.

4.2.4 Typage – première version.

Le principe du typage reprend les méthodes que nous avons appliquées aux Controlled Ambients et au Seals et le adapte pour gérer les problèmes que posent le langage paramétrique de motifs. La grammaire des types est présentée sur la figure 4.17.

$$
\begin{array}{lcll}
T & ::= & A & \\
 & | & U & \\
A & ::= & \mathit{Site}(s,e) & s \in \mathbf{N} \cup \{\infty\}, e \in \mathbf{N} \cup \{\infty\} \\
U & ::= & \mathit{Proc}(t) & t \in \mathbf{N} \cup \{\infty\} \\
\Xi & ::= & \mathit{Pat}(\gamma) & \gamma \in \mathcal{N} \to T
\end{array}
$$

FIG. 4.17 – Grammaire de types pour le contrôle des ressources – Kells.

Le type d'un site (message ou cellule) est $\mathit{Site}(s,e)$ où s est la taille de la réserve de ce site et e le nombre de ressources qu'il occupe. Le type d'un processus P est $\mathit{Proc}(t)$ où t est une borne supérieure sur le nombre de ressources que peut nécessiter P pour s'exécuter correctement. Enfin, le type d'un motif ξ est $\mathit{Pat}(\gamma)$ où γ est une fonction partielle de l'ensemble des noms vers l'ensemble des types de sites ou de processus et qui, à chaque variable présente dans ξ, associe le type des entités qui seront substituées à ces variables par la macro `match` du langage de motifs.

Les règles de typage sont présentées sur la figure 4.18. Les règles T-NIL, T-RES et T-PAR sont essentiellement identiques à leurs contreparties des Seals. La règle T-NAMEPROCESS permet de typer un processus constitué en tout et pour tout d'un nom – nous considérons qu'un tel terme n'occupe pas de ressources, il peut dont être typé avec n'importe quel type. Les sites sont typés à l'aide des règles T-CELL et T-MSG : si n dispose de s ressources et si P n'a pas besoin de plus de s ressources, alors n peut contenir P. Le processus $n\,[P]\,.Q$ nécessitera e ressources allouées à n puis, une fois que ces ressources sont désallouées, t_Q ressources pour exécuter Q. Nous typons de la même manière $n\langle P\rangle.Q$. Il est à noter que la continuation Q d'une cellule ou d'un message a été ajoutée pour ces raisons au calcul des Kells.

Enfin, la règle T-TRIG spécifie que, pour un motif ξ dont on peut déterminer le type $\mathit{Pat}(\gamma)$, on peut utiliser les informations de type incluses dans γ pour examiner P et en déduire le type de $\xi \rhd P$. Même si nous ne spécifions pas la manière dont $\mathit{Pat}(\gamma)$ doit être obtenu, qui dépendra probablement du langage de motifs, l'intuition est que ξ contiendra probablement des annotations pour chacune des variables qui seront liées lors d'une réaction entre $\xi \rhd P$ et son environnement.

Cette règle est malheureusement fausse à moins d'être en mesure d'assurer que les processus qui seront liés lors d'une telle réaction se conformeront bien aux informations de type contenues dans γ. Pour ce faire, nous pouvons soit examiner ξ, par exemple en le typant plus complètement – ce qui nécessite de connaître le langage des motifs – soit imposer à la réduction de respecter γ. Pour commencer, c'est cette deuxième option que nous allons employer. Nous verrons plus loin comment alléger cette contrainte.

$$\text{T-Nil}\ \Gamma \vdash 0 : Proc(t) \qquad \text{T-Res}\ \frac{\Gamma, n : T \vdash P : U}{\Gamma \vdash (\nu n : T)P : U} \qquad \text{T-Name}\ \frac{\Gamma(n) = T}{\Gamma \vdash n : T}$$

$$\text{T-Par}\ \frac{\Gamma \vdash P : Proc(t_P) \qquad \Gamma \vdash Q : Proc(t_Q)}{\Gamma \vdash P \mid Q : Proc(u)}\ u \geq t_P + t_Q$$

$$\text{T-NameProcess}\ \frac{\Gamma \vdash n : Kell(\_, \_)}{\Gamma \vdash n : Proc(t)}$$

$$\text{T-Cell}\ \frac{\begin{array}{c}\Gamma \vdash n : Site(s, e) \\ \Gamma \vdash P : Proc(s) \qquad \Gamma \vdash Q : Proc(t_Q)\end{array}}{\Gamma \vdash n\,[P]\,.Q : Proc(u)}\ u \geq t_Q,\ u \geq e$$

$$\text{T-Msg}\ \frac{\begin{array}{c}\Gamma \vdash n : Site(s, e) \\ \Gamma \vdash P : Proc(s) \qquad \Gamma \vdash Q : Proc(t_Q)\end{array}}{\Gamma \vdash n\langle P\rangle.Q : Proc(u)}\ u \geq t_Q,\ u \geq e$$

$$\text{T-Trig}\ \frac{\Gamma \vdash \xi : Pat(\gamma) \qquad \Gamma, \gamma \vdash P : Proc(t_P)}{\Gamma \vdash \xi \triangleright P : Proc(t_P)}$$

FIG. 4.18 – Règles de typage pour le contrôle des ressources dans les Kells.

Pour commencer, nous allons imposer certaines contraintes sur le langage de motifs.

Définition 13 (Typage compatible avec les substitutions)
Si Γ est un environnement de typage, le typage des motifs est compatible avec les substitutions dans Γ si, pour tout ξ, dès que $\Gamma, n : W \vdash \xi : Pat(\gamma)$ et $\Gamma \vdash m : W$, alors $\Gamma \vdash \xi\{n \leftarrow m\} : Pat(\gamma)$.

Définition 14 (Typage compatible avec la congruence structurelle)
Le typage des motifs est compatible avec la congruence structurelle dans Γ si, pour tout ξ tel que $\Gamma \vdash \xi : Pat(\gamma)$ et tout ζ, si $\xi \equiv \zeta$, alors $\Gamma \vdash \zeta : Pat(\gamma)$ et si $\zeta \equiv \xi$, alors $\Gamma \vdash \zeta : Pat(\gamma)$.

Nous supposerons le typage des motifs compatible avec les substitutions et la congruence structurelle.

Nous introduisons sur la figure 4.19 des règles de réduction typée pour le calcul des Kells. Ces règles interdisent de réduire les termes mal typés et imposent que, lors de chaque réaction qui déclenche un processus gardé par un motif, le type du motif doit être respecté.

Propriétés

De la même manière que dans les CA, nous avons les propriétés suivantes.

$$\textsc{R-L-Typed}\ \frac{\begin{array}{c}\Gamma \vdash \xi : Pat(\gamma) \qquad \Gamma \vdash (\xi \rhd P)|U_1|U_2|U_3 : T \\ \xi \neq \emptyset \qquad \theta = \mathtt{match}_\Gamma(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3)) \\ Dom(\gamma) = Dom(\theta) \qquad \forall x, \theta(x) = P_x \Rightarrow \Gamma \vdash P_x : \gamma(x) \\ \Delta(U_1, M_1, V_1) \qquad \Upsilon(U_2, M_2, V_2) \qquad \Psi(U_3, M_3, V_3) \qquad U_2 \not\rightarrow\end{array}}{(\xi \rhd P)|U_1|U_2|U_3 \longrightarrow_\Gamma P\theta|V_1|V_2|V_3}$$

$$\textsc{R-G-Typed}\ \frac{\begin{array}{c}\Gamma \vdash \xi : Pat(\gamma) \qquad \Gamma \vdash (\xi \rhd P)|U_1|U_2|U_3 : T \\ \xi \neq \emptyset \qquad \theta = \mathtt{match}_\Gamma(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3) \mid ab(M_4)) \\ Dom(\gamma) = Dom(\theta) \\ \forall x, \theta(x) = P_x \Rightarrow \Gamma \vdash P_x : \gamma(x) \qquad \Delta(U_1, M_1, V_1) \\ \Upsilon(U_2, M_2, V_2) \qquad \Psi(U_3, M_3, M_k) \qquad \Delta(U_4, M_4, V_4) \qquad U_2 \not\rightarrow\end{array}}{b\,[(\xi \rhd P)|U_1|U_2|U_3|R]\,.S|U_4 \longrightarrow_\Gamma b\,[P\theta|V_1|V_2|V_3|R]\,.S|V_4}$$

$$\textsc{R-Struct-Typed}\ \frac{\Gamma \vdash P \qquad P \equiv P' \qquad P' \longrightarrow_\Gamma Q' \qquad Q' \equiv Q}{P \equiv Q}$$

$$\textsc{R-Res-Typed}\ \frac{P \longrightarrow_{\Gamma, n:T} Q \qquad \Gamma \vdash (\nu n : T)P : U}{(\nu n : T)P \longrightarrow_\Gamma (\nu n : T)Q}$$

$$\textsc{R-Par-Typed}\ \frac{P \longrightarrow_\Gamma Q \qquad \Gamma \vdash P|R : U}{P|R \longrightarrow_\Gamma Q|R}$$

$$\textsc{R-Cell-Typed}\ \frac{P \longrightarrow_\Gamma Q \qquad \Gamma \vdash a\,[P]\,.R : U}{a\,[P]\,.R \longrightarrow_\Gamma a\,[Q]\,.R}$$

FIG. 4.19 – Sémantique de réduction typée pour le contrôle des ressources dans les Kells.

Lemme 14 (Sous-typage)
Soient Γ un environnement et P un processus typable dans Γ. Notons $\Gamma \vdash P : Proc(t)$. Alors, pour tout $u \geq t$, on aura $\Gamma \vdash P : Proc(u)$.

Ici aussi, le lemme est trivial.

Corollaire 3 (Type minimal) *Soient Γ un environnement et P un processus typable dans Γ. Notons $\Gamma \vdash P : Proc(t)$. Alors, il existe u minimal tel que $\Gamma \vdash P : Proc(u)$.*

Lemme 15 (Renforcement)
Si $\Gamma, n : T \vdash P : U$ et $n \notin fv(P)$, alors $\Gamma \vdash P : U$.

Lemme 16 (Affaiblissement)
Si $\Gamma \vdash P : U$ et $n \notin fv(P)$ alors $\Gamma, nTA \vdash U$.

Ces lemmes sont traditionnels, tout comme les preuves.

Lemme 17 (Les bons types respectent la politique)
Si $\Gamma \vdash P : U$, alors $res_\Gamma(P)$ est défini.

$$
\begin{array}{lcl}
\mathtt{match}_\Gamma(\xi, \alpha(M_1|M_2)) & \triangleq & \mathtt{match}_\Gamma(\xi, \alpha(M_2)|\alpha(M_2)) \\
\mathtt{match}_\Gamma(\xi_1|\xi_2, \alpha(M_1)|\beta(M_2)) & \triangleq & \mathtt{match}_\Gamma(\xi_1, \alpha(M_1)) \oplus \mathtt{match}(\xi_2, \beta(M_2)) \\
 & & \quad \text{pour toutes annotation } \alpha, \beta \\
\mathtt{match}_\Gamma(a\,[(x:T)]\,, loc(a\,[P])) & \triangleq & \{x \leftarrow P\} \text{ si } \Gamma \vdash P : T \\
\mathtt{match}_\Gamma(a\langle\overline{\rho}\rangle, loc(a\langle P\rangle)) & \triangleq & \mathtt{match}_\Gamma(\overline{\rho}, loc(P)) \\
\mathtt{match}_\Gamma(a\langle\overline{\rho}\rangle^{\downarrow}, dn(a\langle P\rangle)) & \triangleq & \mathtt{match}_\Gamma(\overline{\rho}, loc(P)) \\
\mathtt{match}_\Gamma(a\langle\overline{\rho}\rangle^{\uparrow}, ab(a\langle P\rangle)) & \triangleq & \mathtt{match}_\Gamma(\overline{\rho}, loc(P)) \\
\mathtt{match}_\Gamma(\epsilon, loc(\mathbf{0})) & \triangleq & \{\} \\
\mathtt{match}_\Gamma(n, loc(n)) & \triangleq & \{\} \\
\mathtt{match}_\Gamma((x:T), loc(P)) & \triangleq & \{x \leftarrow P\} \text{ si } \Gamma \vdash P : T \\
\mathtt{match}_\Gamma((a:T)\langle\overline{\rho}\rangle, loc(b\,[P])) & \triangleq & \{a \leftarrow b\} \oplus \mathtt{match}_\Gamma(\overline{\rho}, P) \text{ si } \Gamma \vdash b : T
\end{array}
$$

FIG. 4.20 – Acceptation des processus par notre langage de motifs – version typée.

La preuve, simple, est présentée dans l'annexe C.1.

Théorème 6 (Subject Reduction) *Si* $\Gamma \vdash P : U$ *et si* $P \longrightarrow_\Gamma Q$, *alors* $\Gamma \vdash Q : U$.

La preuve est présentée dans l'annexe C.2.

De manière intéressante, à partir du moment où nous avons prouvé la Subject Reduction, nous pourrions alléger légèrement la sémantique typée. En effet, comme le type se préserve, il n'est pas a priori nécessaire de vérifier avant chaque pas de réduction que le terme à réduire est bien typé. Il suffirait de procéder à une vérification unique au moment de commencer la réduction – un comportement que l'on trouve par exemple dans le cadre du code Java, dont le typage est vérifié au moment du chargement, ou du Code Porteur de Preuves.

Théorème 7 (Contrôle des ressources) *Soient* Γ *environnement et* P *un processus. Si* P *est typable dans* Γ, *alors* P *est ressource-contrôlé dans* Γ.

Il s'agit d'un corollaire direct de ce qui précède.

4.2.5 Exemples

Langage de motifs

Dans cette section, nous nous intéressons de nouveau au langage de motifs présenté dans la section 4.2.2 et nous l'adaptons à la sémantique de réduction typée. Nous conservons la grammaire de la figure 4.12. Pour déterminer $Pat(\gamma)$, nous utiliserons les annotations de types prévues dans cette grammaire.

Nous n'entrerons pas dans les détails sur la manière d'obtenir $Pat(\gamma)$ à partir de ξ : il suffit de collecter toutes les annotations de type figurant dans ξ – nous supposerons que chaque variable n'apparaît qu'une fois au plus.

Nous présentons sur la figure 4.20 une adaptation de `match` à la sémantique typée. En plus des opérations précédentes, $\texttt{match}_\Gamma$ vérifie que les liaisons sont effectivement compatibles avec les informations de type fournies en annotation. En reprenant les notations de R-L-TYPED ou R-G-TYPED, on vérifie donc simplement que γ et θ ont toujours les mêmes domaines de définition et que dès que $\theta(x) = P_x$, on a $\Gamma \vdash P_x : \gamma(x)$.

Lemme 18 (Les bons motifs font de bons types)
Avec ce langage de motifs, si $\texttt{match}_\Gamma(\xi, P) = \theta$ *et si* $\Gamma \vdash \xi : Pat(\gamma)$, *alors* θ *et* γ *ont le même domaine de définition et pour tout* x, *si* $\theta(x) = P_x$, *alors* $\Gamma \vdash P_x : \gamma(x)$.

Ce lemme est trivial.
C'est ce langage de motifs que nous utiliserons pour la suite de nos exemples.

Verrous

Revenons sur l'exemple des verrous, présenté dans la section 4.2.2. Considérons un environnement Γ tel que $\Gamma \vdash P : Proc(t_P)$, $\Gamma \vdash Q : Proc(t_Q)$ et $\Gamma \vdash n : Site(s, e)$. Comme x sera $\mathbf{0}$, nous pouvons prendre $T = Proc(0)$.

Typage de P			
$\Gamma \vdash P :$		$Proc(t_P)$	Par hypothèse
	Comme	$x \notin fv(P)$	
$\Rightarrow \Gamma, x : T \vdash P :$		$Proc(t_P)$	Par *Affaiblissement*

Typage de `Acq` $n.P$			
$\Gamma, x : T \vdash P :$		$Proc(t_P)$	Cf. plus haut
$\Gamma \vdash n\langle(x : T)\rangle :$		$Pat(\{x : T\})$	Par hypothèse
$\Rightarrow \Gamma \vdash n\langle(x : T)\rangle \triangleright P :$		$Proc(t_P)$	Par T-TRIG

Typage de $n\langle\rangle$			
$\Gamma \vdash n :$		$Site(s, e)$	Par hypothèse
$\Rightarrow \Gamma \vdash n\langle\rangle :$		$Proc(e)$	Par T-MSG

Typage de `Rel` $n.Q$			
$\Gamma \vdash Q :$		$Proc(t_Q)$	Par hypothèse
$\Gamma \vdash n\langle\rangle :$		$Proc(e)$	Cf. plus haut
$\Rightarrow \Gamma \vdash Q \mid n\langle\rangle :$		$Proc(t_Q + e)$	Par T-PAR

○

Nous venons donc de prouver que les termes `Acq` $n.P$ et `Rel` $n.Q$ sont typables dans Γ. Si `Acq` $n.P$ a le même type que P, comme on pouvait l'espérer – aucune ressource n'est allouée pour l'acquisition du verrou – le processus `Rel` $n.Q$, à l'inverse, nécessite e ressources de plus que Q. Ce comportement est d'autant plus surprenant qu'il n'apparaît pas dans le calcul des Seals. Si l'on peut fixer $e = 0$ de manière à faire disparaître le symptôme, ce phénomène est conséquence de notre manière de typer les messages comme s'ils étaient des cellules, ce qui est discutable dans le cas présent.

Communication sur un canal

Considérons maintenant les différentes implantations que nous avons présentées pour la communication sur un canal.

Nous supposerons $\Gamma \vdash P : t_P$, $\Gamma, x : T \vdash Q : t_Q$, $\Gamma(n) = Site(s, e)$ et $\Gamma \vdash M : T$. De plus, pour que n puisse contenir M, nous supposerons $\Gamma \vdash M : Proc(s)$.

Typage de `Send` M `on` $n.P$			
	$\Gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
	$\Gamma \vdash M :$	$Proc(s)$	Par hypothèse
	$\Gamma \vdash n :$	$Site(s, e)$	Par hypothèse
$\Rightarrow$	$\Gamma \vdash n\langle M\rangle.P :$	$Proc(u_1)$	Par T-Msg
	Avec	$u_1 \geq t_P$	
		$u_1 \geq e$	

Typage de `Receive` x `on` $n.Q$			
$\Gamma, x : T \vdash$	$Q :$	$Proc(t_Q)$	Par hypothèse
	$\Gamma \vdash n\langle(x : T)\rangle :$	$Pat(\{x : T\})$	Par hypothèse
$\Rightarrow$	$\Gamma \vdash n\langle(x : T)\rangle \triangleright Q :$	$Proc(t_Q)$	Par T-Trig

○

Contrairement à la synchronisation sur un verrou, le coût de l'émission du message est masqué par le coût d'exécution de Q, pour des valeurs de t_Q suffisamment élevées. Comme on pouvait l'imaginer, le type de `Send` M `on` $n.P$ dans Γ est le type de P dans Γ et le type de `Receive` x `on` $n.Q$ dans Γ est le type de Q dans $\Gamma, x : T$.

Pour la suite de cet exemple, nous considérerons comme nul l'encombrement des messages.

Pour la deuxième implantation, nous considérerons $\Gamma \vdash P : Proc(t_P)$, $\Gamma, x : T \vdash Proc(Q : t_Q)$, $\Gamma(n) = Site(0, 0)$, $\Gamma(msg) = Site(s_{msg}, 0)$, $\Gamma(then) = Site(t_P, 0)$, $\Gamma \vdash M : T$. De plus, pour que msg puisse contenir M, nous supposerons $\Gamma \vdash M : Proc(s_{msg})$. Enfin, nous noterons $\Delta = \Gamma, x : T, y : Proc(t_P)$.

Typage de $then\langle P\rangle$			
	$\Gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
	$\Gamma \vdash then :$	$Site(t_P, 0)$	Par hypothèse
$\Rightarrow$	$\Gamma \vdash then\langle P\rangle :$	$Proc(0)$	Par T-Msg
Typage de $msg\langle M\rangle$			
	$\Gamma \vdash M :$	$Proc(s_{msg})$	Par hypothèse
	$\Gamma \vdash msg :$	$Site(s_{msg}, 0)$	Par hypothèse
$\Rightarrow$	$\Gamma \vdash msg\langle M\rangle :$	$Proc(0)$	Par T-Msg
Typage de $msg\langle M\rangle \mid then\langle P\rangle$			
	$\Gamma \vdash then\langle P\rangle :$	$Proc(0)$	Cf. plus haut
	$\Gamma \vdash msg\langle M\rangle :$	$Proc(0)$	Cf. plus haut
$\Rightarrow$	$\Gamma \vdash msg\langle M\rangle \mid then\langle P\rangle :$	$Proc(0)$	Par T-Par
Typage de `Send`$_2$ M `on` $n.P$			
	$\Gamma \vdash msg\langle M\rangle \mid then\langle P\rangle :$	$Proc(0)$	Cf. plus haut
	$\Gamma \vdash n :$	$Site(0, 0)$	Par hypothèse
$\Rightarrow$	$\Gamma \vdash n\langle \ldots\rangle :$	$Proc(0)$	Par T-Msg

Typage de Q			
$\Gamma, x : T \vdash$	$Q :$	$Proc(t_Q)$	Par hypothèse
$\Rightarrow$	$\Delta \vdash Q :$	$Proc(t_Q)$	Par *Affaiblissement*

Typage de $Q \mid y$		
$\Delta \vdash Q :$	$Proc(t_Q)$	Cf. plus haut
$\Delta \vdash y :$	$Proc(t_P)$	Par T-NAME
$\Rightarrow \Delta \vdash Q \mid y :$	$Proc(t_P + t_Q)$	Par T-PAR
Typage de $\texttt{Receive}_2\ x\ \texttt{on}\ n.Q$		
$\Delta \vdash Q \mid y :$	$Proc(t_P + t_Q)$	Cf. plus haut
$\Delta \vdash n\langle msg\langle(x:T)\rangle \mid then\langle(y:Proc(t_P))\rangle\rangle :$	$Pat(\{x:T, y:Proc(t_P)\})$	Par hypothèse
$\Rightarrow \Gamma \vdash \texttt{Receive}_2\ x\ \texttt{on}\ n.Q :$	$Proc(t_P + t_Q)$	Par T-TRIG

○

Pour cette implantation, le processus $\texttt{Send}_2\ M\ \texttt{on}\ n.P$ ne nécessite donc aucune allocation de ressources, tandis que le processus $\texttt{Receive}_2\ x\ \texttt{on}\ n.Q$ alloue donc aussi bien les ressources pour exécuter P que les ressources pour exécuter Q. Cette implantation a donc permis de déplacer l'intégralité du coût de la communication du côté de la réception, comme nous le faisons lorsque nous typions les Controlled Ambients, dans la section 3.3.4.

Sans entrer dans autant de détails, nous pouvons typer $\texttt{Send}_3/\texttt{Receive}_3$ et obtenir le résultat dual :

$$\begin{cases} \Gamma \vdash & \texttt{Send}_3\ M\ \texttt{on}\ n.P : Proc(t_P + t_Q) \\ \Gamma \vdash & \texttt{Receive}_2\ x\ \texttt{on}\ n.Q : Proc(0)\ . \end{cases}$$

Ce qui signifie que nous avons déplacé l'intégralité du coût de la communication du côté de l'émission.

Nous verrons dans l'exemple qui suit une application de cette propriété.

Réplication et récursion

Comme la réplication et la récursion sont deux constructions importantes dans le calcul des Kells, il peut être intéressant de considérer leurs propriétés du point de vue du contrôle des ressources.

Réplication Commençons par considérer la réplication. Sans entrer dans le détail du typage, en notant $A_t = Site(s_t, e_t)$ et $\Gamma \vdash P : Proc(t_P)$, si nous supposons $\Gamma \vdash \texttt{Y}(P, \xi, t) : Proc(t_Y)$, nous pouvons facilement déduire que $\Gamma \vdash \texttt{Y}(P, \xi, t) : Proc(t_Y + t_P + e_t)$. En d'autres termes, même en fixant $e_t = 0$, le type de $\xi \diamond P$ est le type de $P \mid \xi \diamond P$. À moins d'avoir $t_P = 0$, nous ne sommes pas en mesure de garantir que $P \mid \xi \diamond P$ nécessite moins d'une infinité de ressources.

Ce résultat, d'ailleurs cohérent avec la définition habituelle de $!P$, est d'autant moins surprenant qu'un déclencheur $\xi \triangleright P$ peut réagir avec un ou plusieurs processus extérieurs à la cellule. Ainsi, le processus $Q \triangleq a\langle\rangle^{\uparrow} \diamond P$ peut potentiellement libérer une infinité de copies de P s'il y a une boucle qui produit une infinité de messages $a\langle\rangle$ en parallèle avec la cellule qui contient Q. Reformulé dans le cadre de la modélisation d'un réseau, P peut être considéré comme un service accessible sur le port a et incontrôlé donc vulnérable à une attaque de type Déni de Service. S'il est facile, dans notre langage de motifs, de vérifier statiquement que ξ ne peut réagir qu'à des messages locaux, cela peut s'avérer plus difficile pour d'autres instances du calcul des Kells.

C'est dans ces circonstances que peuvent servir les macros $\texttt{Send}_2/\texttt{Receive}_2$ et $\texttt{Send}_3/\texttt{Receive}_3$: comme nous avons pu le vérifier en le typant, le processus $\texttt{Receive}_3.P$ n'entraîne aucune allocation. Par conséquent, ce processus peut être répliqué sans risques, à l'aide de `Repl` ou de toute autre macro similaire – le coût de la communication/synchronisation se retrouve entièrement reporté sur le processus qui émet la requête. Ce mécanisme est à mettre en rapport avec les "spawner" de BoCa, présentés à la section 3.2.1, et l'équilibrage des coûts de communication dans Cπ, que nous discuterons dans la section 6.3.7.

Récursion Sans entrer non plus dans le détail du typage, si nous considérons $T_X = Proc(t_P)$, $T_Y = Proc(t_P)$, $A_t = Site(\infty, 0)$, $A_u = Site(\infty, 0)$, $\Gamma, X : Proc(t_P) \vdash P : Proc(t_P)$, nous pouvons prouver aisément que $\Gamma \vdash \texttt{Rec}(X, P) : Proc(t_P)$. En d'autres termes, si nous supprimons les informations sur les noms privés, nous pouvons réécrire cela sous la forme

$$\text{T-Rec-Lemma}\ \frac{\Gamma, X : Proc(t_P) \vdash P : Proc(t_P)}{\Gamma \vdash \texttt{Rec}(X, P) : Proc(t_P)}$$

ce qui reprend presque exactement la règle que nous avions dans les calculs avec récursion.

Le protocole du taxi

Pour typer le protocole du taxi, nous pouvons employer l'environnement suivant :

$$\begin{cases} A_c & = Seal(\infty, 1) \\ A_s & = Seal(\infty, 0) \\ T_{later} & = Proc(0) \\ T_c & = Proc(1) \\ T_{cont} & = Proc(0) \\ \Gamma(from) & = \Gamma(to) = \Gamma(client) = \Gamma(later) = Seal(0, 0) \\ \Gamma(call) & = \Gamma(name) = \Gamma(dest) = Seal(0, 0) \\ \Gamma(to) & = \Gamma(s) = A_s \\ \Gamma(content) & = Seal(\infty, 1) \\ \Gamma(loaded) & = \Gamma(busy) = Site(1, 0) \\ \Gamma(empty) & = \Gamma(cab) = Site(0, 0)\ . \end{cases}$$

Contrairement à ce que nous avions dans les calculs précédents, ici, la structure du taxi va changer au fur et à mesure. Lorsque le taxi est dans l'état *empty* ou *cab*, il est vide – une capacité de 0 suffit donc. À l'inverse, lorsqu'il est dans l'état *loaded* ou *busy*, la place de *content* est réservée.

Le reste du typage est classique.

4.2.6 Vers un contrôle plus statique

La méthode que nous avons employée jusqu'ici pour permettre de typer le contrôle des ressources dans le calcul des Kells nécessite la définition d'une relation de réduction typée.

Dans cette section, plutôt que d'adapter la relation de réduction au système de types, nous allons présenter un travail en cours, qui cherche à restreindre un langage de motifs – celui que nous avons présenté dans la section 4.2.2 – à l'aide d'un typage sur les motifs.

$$
\begin{array}{lcll}
T & ::= & A & \\
 & | & U & \\
A & ::= & Site(e)[T] & e \in \mathbf{N} \cup \{\infty\} \\
U & ::= & Proc(t) & t \in \mathbf{N} \cup \{\infty\} \\
\Xi & ::= & Pat(\gamma) & \gamma \in \mathcal{N} \to T
\end{array}
$$

FIG. 4.21 – Grammaire de types pour le contrôle des ressources statique – Kells.

Nous présentons sur la figure 4.21 une version modifiée de la grammaire de système de types de la figure 4.21. Ainsi, le type d'un site a est $Site(e)[T]$ si a encombre e ressources et peut contenir un processus *ou un nom* de type T. Pour ce qui suit, si γ_1 et γ_2 sont des fonctions de $\mathcal{N}$ vers T de domaines de définitions disjoints, nous noterons $\gamma_1 \cup \gamma_2$ la fonction qui à tout x de $Dom(\gamma_1)$ associe $\gamma_1(x)$ et à tout x de $Dom(\gamma_2)$ associe $\gamma_2(x)$. Nous étendons, de plus, la notion de type d'un motif à certains autres éléments de la grammaire de motifs.

La figure 4.22 présente les règles adaptées à cette nouvelle grammaire ainsi que les règles employées pour typer les motifs. Ainsi, la règle T-CELL spécifie, comme sa contrepartie de la figure 4.18, qu'une cellule n peut contenir un processus P de type U seulement si le type de n spécifie que cette cellule peut accueillir des processus de type U. Le type de $n\,[P]\,.Q$ se calcule alors comme dans le cas précédent. Le fonctionnement de la règle T-MSG est identique. Dans les deux cas, la différence majeure entre la nouvelle version et l'ancienne et que le type $Site(e)[T]$ permet de différencier les messages qui contiennent des noms des messages qui contiennent d'autres types de processus.

Le typage des motifs vise à collecter les annotations de types sur les variables et à vérifier que les types des sites impliqués dans le motif sont suffisamment restrictifs pour garantir que les annotations seront respectées. Ainsi, selon T-PAT-CELL, un motif qui accepte les cellules nommées a avec un contenu x de type T n'est valable que si les cellules nommées a ne peuvent contenir que des processus ou noms de type T ; le type du motif reprend alors l'information $x : T$. La règle T-PAT-MESSAGE-BINDER est similaire pour le cas des messages. Les règles T-PAT-PAR et T-PAT-PAR2 permettent de collecter les annotations de deux motifs ou sous-motifs parallèles et vérifient au passage qu'aucune variable n'est définie simultanément dans ces deux motifs. Les motifs sans lieur sont traités par les règles T-PAT-MESSAGE-CONSTANT et T-PAT-MESSAGE-EMPTY – qui ne vérifient rien. Enfin, T-PAT-MESSAGE-SUBMESSAGE permet de traiter le cas des sous-motifs inclus dans des messages et pour lesquels il suffit de faire remonter les informations.

Une limitation importante de ce système de types est qu'il rejette tous les motifs de la forme $(a : A)\langle \ldots \rangle$: pour accepter de tels motifs, il serait nécessaire de concevoir un système plus riche et plus complexe, capable d'exprimer des propriétés telles que "tous les sous-sites de ce site sont de type A". Nous avons considéré que les motifs acceptés par le système de types sont suffisants pour une première version, quitte à compliquer la théorie ultérieurement.

$$\text{T-Cell}\ \frac{\Gamma \vdash n : Site(e)[U] \qquad \Gamma \vdash P : U \qquad \Gamma \vdash Q : Proc(t_Q)}{\Gamma \vdash n\,[P]\,.Q : Proc(u)}\ u \geq t_q,\ u \geq e$$

$$\text{T-Msg}\ \frac{\Gamma \vdash n : Site(e)[T] \qquad \Gamma \vdash P : T \qquad \Gamma \vdash Q : Proc(t_Q)}{\Gamma \vdash n\langle P\rangle.Q : Proc(u)}\ u \geq t_q,\ u \geq e$$

$$\text{T-Pat-Cell}\ \frac{\Gamma \vdash a : Site(\_)[T]}{\Gamma \vdash_{pat} a\,[(x : T)] : Pat(\{x : T\})}$$

$$\text{T-Pat-Par}\ \frac{\Gamma \vdash_{pat} \xi_1 : Pat(\gamma_1) \qquad \Gamma \vdash_{pat} \xi_2 : Pat(\gamma_2)}{\Gamma \vdash_{pat} \xi_1 \mid \xi_2 : Pat(\gamma_1 \cup \gamma_2)}\ Dom(\gamma_1) \cap Dom(\gamma_2) = \emptyset$$

$$\text{T-Pat-Par2}\ \frac{\begin{array}{c}\Gamma \vdash_{pat} \rho_1 : Pat(\gamma_1) \\ \Gamma \vdash_{pat} \rho_2 : Pat(\gamma_2)\end{array}}{\Gamma \vdash_{pat} \rho_1 \mid \rho_2 : Pat(\gamma_1 \cup \gamma_2)}\ Dom(\gamma_1) \cap Dom(\gamma_2) = \emptyset$$

$$\text{T-Pat-Message-Binder}\ \frac{\Gamma \vdash a : Site(\_)[T]}{\Gamma \vdash_{pat} a\langle(x : T)\rangle^{-} : Pat(\{x : T\})}$$

$$\text{T-Pat-Message-Constant}\ \Gamma \vdash_{pat} a\langle n\rangle^{-} : Pat(\emptyset)$$

$$\text{T-Pat-Message-Empty}\ \Gamma \vdash_{pat} a\langle\epsilon\rangle^{-} : Pat(\emptyset)$$

$$\text{T-Pat-Message-Submessage}\ \frac{\Gamma \vdash_{pat} \overline{\rho} : Pat(\gamma)}{\Gamma \vdash_{pat} a\langle\overline{\rho}\rangle^{-} : Pat(\gamma)}$$

FIG. 4.22 – Règles de typage pour le contrôle des ressources dans les Kells, instanciées pour notre langage de motifs.

Propriétés

Lemme 19 (Les bons motifs font les bons types)
Soient ξ un motif, θ une substitution, Γ en environnement, K un processus annoté et $\overline{K}$ le processus obtenu en supprimant toutes les annotations de K. Alors, si $\Gamma \vdash_{pat} \xi : Pat(\gamma)$, si $\overline{K}$ est typable dans Γ et si $\theta = \mathtt{match}(\xi, K)$, nous avons $Dom(\gamma) = Dom(\theta)$ et, pour tout x, si $\theta(x) = P_x$ alors $\Gamma \vdash P_x : \gamma(x)$.

Ce lemme, prouvé dans l'annexe C.3.3, permet de prouver que, pour des processus bien typés, les propriétés vérifiées systématiquement avant réduction par R-L-Typed ou R-G-Typed sont valides également avec une réduction non-typée.

Lemme 20 (Sous-typage)
Soient Γ un environnement et P un processus typable dans Γ. Notons $\Gamma \vdash P : Proc(t)$. Alors, pour tout $u \geq t$, on aura $\Gamma \vdash P : Proc(u)$.

Ici aussi, le lemme est trivial.

Corollaire 4 (Type minimal) *Soient* Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P : Proc(t)$. *Alors, il existe* u *minimal tel que* $\Gamma \vdash P : Proc(u)$.

Lemme 21 (Renforcement)
Si $\Gamma, n : T \vdash P : U$ *et* $n \notin fv(P)$, *alors* $\Gamma \vdash P : U$.

Lemme 22 (Affaiblissement)
Si $\Gamma \vdash P : U$ *et* $n \notin fv(P)$ *alors* $\Gamma, n : T \vdash U$.

Ces lemmes sont traditionnels, tout comme les preuves.

Théorème 8 (Subject Reduction) *Si* $\Gamma \vdash P : U$ *et si* $P \longrightarrow Q$, *alors* $\Gamma \vdash Q : U$.

La preuve est présentée dans l'annexe C.4.

Théorème 9 (Contrôle des ressources) *Soient* Γ *environnement et* P *un processus. Si* P *est typable dans* Γ, *alors* P *est ressource-contrôlé dans* Γ.

Il s'agit d'un corollaire direct de ce qui précède.

4.2.7 Eemples

Verrous

Considérons une politique de contrôle de ressources Γ adaptée de celle que nous avions utilisée avec la sémantique de réduction typée et telle que $\Gamma(n) = Site(e)[Proc(0)]$, $T = Proc(0)$, $\Gamma \vdash P : Proc(t_P)$ et $\Gamma \vdash Q : Proc(t_Q)$.

Typage de P			
$\Gamma \vdash P :$		$Proc(t_P)$	Par hypothèse
	Comme	$x \notin fv(P)$	
$\Rightarrow \Gamma, x : T \vdash P :$		$Proc(t_P)$	Par *Affaiblissement*
Typage de $n\langle(x : T)\rangle$			
$\Gamma \vdash n :$		T	Par hypothèse
$\Rightarrow \Gamma \vdash_{pat} n\langle(x : T)\rangle :$		$Pat(\{x : T\})$	Par T-Pat-Message-Binder
Typage de `Acq` $n.P$			
$\Gamma, x : T \vdash P :$		$Proc(t_P)$	Cf. plus haut
$\Gamma \vdash n\langle(x : T)\rangle :$		$Pat(\{x : T\})$	Cf. plus haut
$\Rightarrow \Gamma \vdash n\langle(x : T)\rangle \triangleright P :$		$Proc(t_P)$	Par T-Trig
Typage de $n\langle\rangle$			
$\Gamma \vdash n :$		$Site(s, e)$	Par hypothèse
$\Rightarrow \Gamma \vdash n\langle\rangle :$		$Proc(e)$	Par T-Msg
Typage de `Rel` $n.Q$			
$\Gamma \vdash Q :$		$Proc(t_Q)$	Par hypothèse
$\Gamma \vdash n\langle\rangle :$		$Proc(e)$	Cf. plus haut
$\Rightarrow \Gamma \vdash Q \mid n\langle\rangle :$		$Proc(t_Q + e)$	Par T-Par

○

Après une preuve très proche de celle que nous avons fournie dans la section 4.2.5, nous obtenons exactement le même résultat.

Communication sur un canal, réplication, récursion, protocole du taxi

Sans entrer dans les détails, ici aussi, la structure de la preuve est presque identique et nous obtenons le même résultat que dans le cas de la sémantique typée.

4.2.8 Discussion

Le calcul des Kells est un formalisme conçu pour combiner la puissance de l'ordre supérieur et de synchronisations à n parties à une description hiérarchique des systèmes, similaire à celle des ambients. Même si le calcul ne contient aucune primitive qui permette de spécifier directement des propriétés de ressources, la possibilité de modifier le langage de motifs laisse envisager des instanciations du formalisme susceptibles d'exprimer des politiques dynamiques de contrôle de ressources.

Indépendamment de cette richesse, les Kells offrent la possibilité de réagir simplement aux allocations et aux désallocations et donc d'écrire des systèmes conscients des ressources. Nous avons proposé une représentation des ressources et des politiques de contrôle qui combine typage statique et réduction typée, ainsi qu'une alternative entièrement statique mais restreinte à un langage de motifs précis.

Comparaison Là où les calculs précédents proposaient des primitives de déplacement subjectif (dans les Mobile Ambients ou BoCa), objectif (dans les Seals) ou mixtes (dans les CA), le formalisme des Kells procède selon le principe proche mais distinct de la consommation et de la (re)création. Ainsi, dans les Kells, un site ou un message qui disparaît peut réapparaître identique, modifié, dupliqué ou ne pas réapparaître du tout, localement, dans un site parent ou dans un site enfant. De plus, cette forme de mobilité est entièrement déclenchée par le processus appelant – c'est-à-dire par la localité de destination.

En particulier, les mécanismes d'allocation et de désallocation sont liés non plus à des déplacements mais uniquement à la création et à la destruction des entités.

Bilan Après avoir expérimenté les mécanismes des différents calculs, il s'avère que les motifs permettent une expressivité très forte, dans laquelle on peut aisément mélanger des déplacements *à la* Seals, ou proche des Controlled Ambients, des motifs *à la* Join [29], une élimination manuelle des processus devenus inutiles, des fonctions, des variables mutables ou liées par des `let` et d'autres structures de contrôle de flot traditionnelles (récursion, `if then else`...). Cette richesse est malheureusement aussi une faiblesse car, en plus de méthodes simples, elle offre de nombreuses méthodes compliquées pour résoudre des problèmes simples. Ainsi, de nombreuses solutions peuvent nécessiter des communications d'ordre supérieur : faut-il alors essayer de supprimer le plus possible ces situations, sachant que ces opérations sont souvent lentes et risquées ou faut-il conserver ces situations de manière à préserver la lisibilité des processus ?

En fait, une légère expérience de ce formalisme laisse plutôt l'impression d'un langage de programmation puissant que d'un calcul simple. Cette richesse permet de représenter naturellement de nombreux systèmes à l'aide d'opérations que l'on considèrera comme réalistes dans le système – ainsi, plusieurs processus

lancés par la même application et exécutés dans un même site pourront communiquer des termes d'ordre supérieur sans craintes de sécurité, tandis que des négociations sur un réseau n'impliqueront que des transferts de messages sans contenu actif et sans la possibilité d'examiner les valeurs tant qu'elles n'ont pas été effectivement reçues.

L'abstraction du calcul sur le langage de motifs est l'une des sources de cette richesse, mais pose aussi l'une des principales difficultés que nous ayons rencontrées dans notre travail. Ainsi, notre première tentative nous a conduits à introduire une sémantique typée, dans un cadre où une telle solution n'est pas pratique, alors que nos résultats statiques, eux, dépendent du langage de motifs que nous avons présenté. Heureusement, ce problème n'est pas aussi fondamental qu'il pourrait sembler, puisque les travaux actuels d'implantation des Kells sont en train de voir se stabiliser au moins le noyau du langage de motifs, proche de celui que nous avons utilisé. Lorsque cet aspect du formalisme sera figé, nous pourrons retravailler notre deuxième solution.

Chapitre 5

Bilan

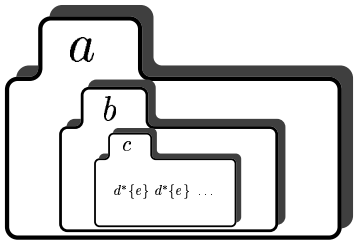
Dans cette partie, nous avons abordé le problème de la gestion des ressources dans le cadre de calculs de sites mobiles. Pour ce faire, nous avons considéré plusieurs formalismes apparentés au calcul des Mobile Ambients et nous avons, à chaque fois, proposé une notion formelle de ressources dans laquelle chaque site représente une entité qui occupe des ressources dans le site parent – et uniquement dans le site parent – et chaque création, destruction ou mouvement de site constitue une allocation ou une désallocation de ressources du site parent.

Allocations et désallocations Pour chacun des formalismes, à l'exception de BoCa, que nous nous sommes contentés de présenter à titre de comparaison, nous avons considéré les primitives du langage liées à l'allocation ou à la désallocation dans le langage – et nous en avons ajouté si nécessaire, tout en gardant la notion de ressources hors du calcul lui-même.

Ainsi, chaque formalisme proposait des opérations qui pouvaient entraîner ou accompagner, directement ou indirectement, des allocations et des désallocations. Citons la capacité `in` a des Mobile Ambients, qui, purement par effet de bord, alloue des ressources dans le site a et en désalloué dans le site source. À l'inverse, dans les Seals, l'arrivée d'un site par $c\text{-}\{b_1, \ldots, b_k\}.P$ est directement liée à une allocation de ressources à l'emplacement de ce processus. De plus, certaines constructions comme $a\,[P]$ dénotent une allocation de ressources sans préciser si cette allocation doit avoir lieu ou si elle a déjà été exécutée.

Pour chacune de ces possibilités d'allocation ou de désallocation, nous avions besoin, au minimum, d'une manière de prendre en compte l'opération et, si possible, d'y réagir. Il s'agit là des mécanismes qui nous permettent de concevoir des protocoles sensibles à l'utilisation des ressources et donc, plus tard, des protocoles ressource-contrôlés.

Il est à noter que les mécanismes en question fonctionneraient moins bien si les ressources étaient considérées comme partagées par l'arbre entier. Considérons ainsi le terme suivant dans les Seals :



Suite à une quelconque réaction, un site se trouve créé en plusieurs exemplaires à l'intérieur de c. Si les ressources de c sont un sous-ensemble des ressources de a, les processus de a doivent donc être en mesure de réagir aux $d^*\{e\}$ afin de pouvoir les refuser en cas de pénurie de ressources. Nos méthodes, à l'inverse de BoCa ou d'analyses telles que les Finite Control Ambients [22], ne sont a priori pas en mesure de traiter directement ce type de situations.

Il est à noter aussi que nous avons choisi de placer ces mécanismes dans le langage lui-même ou d'utiliser des mécanismes qui existaient déjà. Une autre possibilité aurait été d'ajouter à chacun des calculs une membrane disposant d'un langage de processus dédié au contrôle des allocations et désallocations, extension proposée dans le cadre des Guardians [26] ou dans le M-Calcul [74].

Définitions formelles des ressources Pour chacun de ces calculs, à l'exception, de nouveau, de BoCa, nous avons défini formellement la notion de ressource et de protocoles ressource-contrôlés. Ainsi, chaque nom de site se trouve annoté par le nombre de ressources dans la réserve locale du site et le nombre de ressources qu'occupe le site dans le site parent. Cette différence entre espace intérieur et espace extérieur nous permet notamment de gérer naturellement plusieurs cas de figure complexes. Ainsi, contrairement à BoCa, nous pouvons modéliser aisément certains types de bacs à sable : des localités spécifiques dans lesquelles on suppose que les ressources sont gérées par des mécanismes dynamiques indépendants de notre langage. De même, nous pouvons modéliser des systèmes dans lesquels plusieurs systèmes de ressources totalement distincts coexistent dans des lieux eux aussi totalement distincts, tels qu'un ordinateur portable, dont on peut vouloir dire qu'il peut accueillir 100 Go de données alors qu'il pèse 3 kg.

Notre définition formelle des ressources, hors du langage lui-même, nous permet de plus de conserver une certaine indépendance du langage vis-à-vis du problème de la politique de gestion des ressources. Notamment, il est possible, en fixant diverses politiques pour un même terme, de prouver plusieurs propriétés de ce terme, ce qui semble impossible avec BoCa. Nous reviendrons, d'ailleurs, sur les politiques de gestion des ressources, que nous étendrons encore, sans modifier les langages, dans la section 9.

Systèmes de types À partir de chacune des définitions formelles de la notion de ressources, nous avons établi un système de types à l'aide duquel nous pouvons prouver, sans autres vérifications dynamiques que celles introduites par le programmeur, qu'un terme est ressource-contrôlé. Ces systèmes de types ont de nombreux points communs : dans chaque cas, le type d'un site spécifie le nombre de ressources dont il dispose et le nombre de ressources qu'il occupe, alors que le type d'un processus spécifie le nombre de ressources qu'il est autorisé à allouer.

Chacun de ces systèmes de types s'est révélé suffisamment puissant pour prouver des propriétés sur un protocole non-trivial, comme le protocole du taxi. De plus, grâce à ces systèmes de types, nous sommes arrivés, dans certains cas, à mettre en évidence des schémas d'attaque élémentaire possibles (notamment sur la réplication) ainsi que des stratégies de contrôle des ressources qui permettent de contrer ces attaques.

Objectivité des opérations De manière intéressante, nous sommes partis d'un calcul dans lequel les mouvements étaient purement subjectifs (les Mobile Ambients) et, au fur et à mesure que nous avons considéré d'autres calculs, la sémantique des mouvements a changé. Ainsi, dans BoCa, on peut considérer les mouvements comme essentiellement subjectifs mais avec un très léger aspect objectif, puisque un processus est modifié dans le site source (un $\blacksquare_a$ apparaît) et un processus est modifié dans le site destination (un $\blacksquare_a$ disparaît). Cet aspect est plus développé dans les Controlled Ambients, puisque les cocapacités suffiraient à décrire des déplacements – typables et entièrement qualifiés – sans aucune capacité. Ce sont ces cocapacités que nous utilisons pour garantir le contrôle des ressources. Le calcul des Seals, quant à lui, implique des déplacements purement objectifs, le long de canaux eux aussi typables, que nous utilisons de même pour garantir le contrôle des ressources. Enfin, le calcul des Kells remplace les déplacements par des consommations/recréations, une fois de plus purement objectives, grâce à des motifs que nous typons à leur tour, et que nous utilisons pour garantir le contrôle des ressources.

Ainsi, dans chacun de ces calculs, pour permettre d'établir des protocoles ressource-contrôlés et fournir des garanties statiques de ces protocoles, nous avons tiré avantage de mécanismes de déplacement objectifs et typables qui étaient présents dans le langage ou que nous avons ajoutés. Ce qui nous amène à la question suivante : est-il possible de fournir un procédé similaire pour des calculs sans mouvements de site, comme c'est le cas dans le π-calcul [63] ? Avant de chercher à répondre à cette question, nous allons devoir commencer par essayer de déterminer à quel point notre notion de ressource reste valable dans ces formalismes. Si l'étude de calculs comme Linda [34] ou Bonita [72] dépasse le domaine de ce travail, nous présentons dans la prochaine partie nos travaux sur le contrôle des ressources dans des calculs à *mobilité de noms*, tels que le π-calcul.

Notons enfin que les méthodes que nous avons proposées dans le cadre des Mobile Ambients, des Seals ou des Kells ont aussi été expérimentées avec succès dans le cadre d'autres formalismes tels que Nomadic π [82] ou NBA [13], sous la forme de Controlled Nomadic π et des CNBA – travail publié à FGC [78]. Nous avons choisi de ne pas détailler notre travail sur ces calculs car les méthodes et les résultats sont essentiellement identiques à ceux que nous avons détaillé au cours des derniers chapitres. Néanmoins, ces études nous ont permis de confirmer la portabilité de nos méthodes.

Troisième partie

Mobilité de noms

Chapitre 6

Le π-calcul et plus

Après avoir approché les calculs dans lesquels des sites se déplacent les uns par rapport aux autres, nous abordons dans cette partie les calculs dans lesquels seuls les noms sont mobiles. Bien que ces formalismes soient, historiquement, plus anciens que, par exemple, les Mobile Ambients, nous avons préféré cet ordre de présentation car il correspond mieux à l'ordre dans lequel nous avons abordé ces problèmes.

Dans tous les formalismes que nous avons précédemment abordés, un certain nombre de processus étaient répartis explicitement entre des localités nommées et pouvaient entraîner l'allocation ou la désallocation de ressources locales pour des entités elles-mêmes nommées – et confondues avec les localités. Dans le cadre des calculs à mobilité de noms, les processus peuvent être distribués explicitement [41], implicitement [33] ou tous rassemblés dans une même localité [63]. Si l'on peut considérer que les processus allouent des ressources, par exemple en créant de nouveaux noms, en écoutant sur des canaux de communication ou en fournissant des services répliqués, il est plus difficile de déterminer de manière satisfaisante la nature des entités qui occupent les ressources.

Avant de proposer une définition formelle des ressources, nous commencerons par présenter le π-calcul, en tant que représentant des algèbres de processus à mobilité de noms. Dans ce calcul, un système est représenté comme un ensemble de processus, qui se synchronisent et évoluent en communiquant à travers des canaux nommés. Enfin, les processus peuvent être répliqués et peuvent créer de nouveaux noms/canaux.

Ainsi, le terme $(\nu c)P$ représente le processus P dans lequel est disponible un nouveau nom c. Un terme tel que $((\nu c)P) \mid Q$ représente alors un processus dans lequel P et Q s'exécutent en parallèle et où P seul connaît le nom c. Si jamais P communique c à Q, provoquant ainsi une évolution de P en P' et de Q en Q', le terme se réduira en $(\nu c)(P' \mid Q')$, marquant ainsi que le nom c est partagé entre P' et Q' – il y a eu mobilité du nom c. Si, plus tard, P' devait devenir le processus P'', dans lequel c n'est plus connu, le terme pourra se réduire, par exemple, en $P'' \mid (\nu c)Q'$ – on considère que le nom c a encore bougé. Et si Q' devient à son tour Q'' et "oublie" c, le terme pourra se réduire en $P'' \mid Q''$. En d'autres termes, la construction (νc) marque le domaine de définition du nom c, aussi bien spatialement, puisque seuls les processus inclus syntaxiquement sous (νc) connaissent c, que temporellement, puisque $(\nu c)P$ cesse de nécessiter c dès que P s'est réduit en un processus qui ne connaît plus c – à moins que ce nom

ait été préalablement communiqué à un autre processus.

Une notion naturelle de ressources, fort différente de celle que nous avons vue avec les ambients, consiste à considérer que *chaque nom occupe une ressource*. Ainsi, créer un nouveau nom alloue une ressource, alors que supprimer un nom la désalloue.

Types linéaires Le typage linéaire (ou quasi-linéaire), qu'il soit appliqué au π-calcul, au λ-calcul ou à tout autre formalisme, est une analyse qui permet de détermine que certaines entités du langage ne sont utilisées qu'une fois (ou qu'un nombre borné de fois, dans le cas quasi-linéaire). Une fois cette analyse effectuée, s'il est garanti, par exemple, qu'une variable ne peut être utilisée qu'une seule fois, cette variable peut être nettoyée immédiatement après accès par garbage-collection. De même, il est aisé de remplacer un appel à une fonction qui ne peut être invoquée qu'une seule fois par le code de cette fonction, afin d'accélérer l'exécution et de restreindre l'utilisation en mémoire d'un programme.

Dans le cadre du π-calcul, de nombreux canaux sont créés et utilisés une seule fois pour une communication, ce qui se traduit par une (unique) émission et une (unique) réception. Un système de types linéaires [53] permet de garantir cette propriété sur les canaux. Ainsi, en plus d'une polarité qui autorise à lire ou à écrire sur un canal, le type de chaque nom porte une multiplicité qui garantit une borne sur le nombre de communications.

Controlled π-calculus Dans le cadre du Controlled π-calculus, nous adoptons cette conception des ressources et nous adaptons le π-calcul pour permettre de prendre en compte les ressources dans des protocoles. L'allocation par l'opérateur (ν) est cependant une construction plus fuyante que celles que nous avons rencontrées dans les calculs à mobilité de sites, puisque (νc) lui-même se déplace d'un processus à un autre, contraint uniquement à englober les processus qui connaissent c. La désallocation, elle aussi, est une opération plus complexe que dans les formalismes précédents, car les ressources nécessaires pour un nom sont, a priori, désallouées automatiquement, lorsque le nom n'est plus connu par qui que ce soit.

Contrairement à ce que nous avons fait précédemment, et plutôt que d'introduire une opération de désallocation "forcée" assimilable au `delete` du C++, nous avons décidé de conserver cet aspect de désallocation automatique, lorsqu'un nom n'est plus utile. En d'autres termes, là où nous avions considéré les Controlled Ambients, par exemple, comme un langage où la garbage-collection doit s'effectuer manuellement, nous supposons dans le Controlled π l'existence d'une garbage-collection automatique, dont nous décrivons la sémantique.

Pour profiter de cette fonctionnalité, nous introduisons dans le π-calcul un opérateur de finalisation ($\urcorner$). La finalisation [9, 57], qui est une opération liée à la garbage-collection et que l'on trouve notamment en OCaml, Java ou C#, permet d'enregistrer, selon le langage, une fonction ou une méthode en tant que finaliseur d'un entité donnée. Cette fonction ou méthode sera alors déclenchée au moment de la garbage-collection de l'entité. Dans le Controlled π-calculus, nous adaptons cette notion pour permettre de lancer un processus lors de la désallocation d'un nom. Ainsi, si c n'est pas libre dans P, on aura $(\nu c)(\urcorner c).P \longrightarrow P$. La possibilité de synchroniser un processus sur une désallocation permet alors de concevoir des protocoles ressource-contrôlés.

Comme la finalisation d'un nom c n'entre en jeu que lorsque c n'apparaît plus que sous la forme $(\nu c)(\neg c)$, il peut être nécessaire de supprimer des occurrences de c lorsque celles-ci apparaissent dans des processus bloqués. Or, un processus peut être bloqué pour des raisons parfois complexes, telles que des deadlocks (situation de blocage en attente passive, dans laquelle plusieurs processus se bloquent mutuellement car chacun d'entre eux attend une action des autres) ou des livelocks (situation de blocage en attente active, dans laquelle plusieurs processus ne peuvent terminer car ils changent d'état en permanence en réaction à l'état d'autres processus). Pour gérer ces situations, nous considérons comme paramètre du langage une relation d'élimination des processus "morts".

Enfin, dans le cadre du Controlled π-calculus, nous introduisons un système de types qui permet de fournir des garanties sur l'utilisation des ressources.

Extensions et variantes Le mécanisme des *régions* de mémoire permet de simplifier grandement la gestion des allocations et des désallocations de mémoire sans nécessiter de mécanismes complexes de garbage-collection. La technique, utilisée aussi bien dans des calculs impératifs que dans des programmes industriels, consiste à regrouper les entités allouées en régions. Au lieu de désallouer chaque entité individuellement, on supprime la région entière dès qu'elle n'est plus nécessaire, c'est-à-dire dès que les seules références aux entités de cette région sont elles-mêmes dans la région.

Nous présentons brièvement une adaptation des régions au π-calcul, le π-*calculus with groups* [24] et nous discutons d'une possible variante de Cπ qui prendrait en compte les régions.

Le calcul Dπ [41] ajoute au π-calcul une forme de distribution explicite. Ainsi, des termes essentiellement identiques à ceux du π-calcul peuvent être répartis entre plusieurs sites et la communication entre localités se fait par le biais d'une primitive de migration de processus go. Contrairement aux calculs où les sites sont mobiles, les localités de Dπ sont toutes au même niveau, sans hiérarchie. Ce calcul pose des problèmes supplémentaires, notamment de garbage-collection distribuée et de sécurité des migrations. Ainsi, dans Dπ, on a $a\,[\mathsf{go}\ b.P] \longrightarrow b\,[P]$, ce qui signifie que le processus $\mathsf{go}\ b.P$ se comporte comme le processus P localisé dans le site b – plus encore que dans les Mobile Ambients, le site b peut donc recevoir de nouveaux processus sans aucune action de sa part et sans opportunité de refuser la migration. De plus, un nom de canal créé dans une localité peut servir à la communication entre deux processus dans une autre localité, ce qui pose le problème de la paternité du canal. De même, on peut s'interroger sur les ressources allouées par la création d'un nom de site dans une localité et sur la désallocation de ces ressources : à qui appartiennent toutes ces ressources ?

Un effort pour retravailler et étendre Dπ, notamment pour répondre à certaines de ces questions, est actuellement en cours sous le nom de SafeDPI [40]. Ce calcul propose notamment un contrôle de la mobilité, une sémantique plus complète de la création des noms, du filtrage par motifs et des communications d'ordre supérieur. Plutôt que d'étudier les notions de ressources dans SafeDPI, nous avons préféré introduire un calcul plus simple, Controlled Dπ, qui adapte les idées de Cπ à Dπ. Comme dans SafeDPI ou dans le Controlled Nomadic π-calculus [78], nous contrôlons la mobilité en imposant des migrations à travers des canaux typés et comme dans SafeDPI, nous dissocions la création de nouveaux noms de la création de sites. Comme dans Cπ, nous introduisons un

$$
\begin{array}{rcl}
P, Q & ::= & \mathbf{0} \\
& | & P|Q \\
& | & \alpha.P \\
& | & !\alpha.P \\
& | & P + Q \\
& | & (\nu c : C)P
\end{array}
\qquad\qquad
\begin{array}{rcl}
\alpha, \beta & ::= & \overline{a}\langle b \rangle \\
& | & a(b)
\end{array}
$$

FIG. 6.1 – Syntaxe du π-calcul

.

opérateur de finalisation, une notion d'élimination des processus morts et un système de types pour prouver statiquement que les ressources sont contrôlées.

Nous présentons des résultats préliminaires sur ce travail en cours.

6.1 Le π-calcul

Le π-calcul [63] est l'une des algèbres de processus les plus utilisées. Si ce formalisme, comme son ancêtre CCS [62], repose sur la juxtaposition de processus parallèles, susceptibles de se synchroniser le long de canaux nommés, il ajoute à ces bases la communication. Ainsi, les processus, concurrents, évoluent en communiquant des informations – c'est-à-dire des noms – sur des canaux de communications – eux-mêmes représentés uniquement par des noms. La structure du système n'est pas "physique" et visible directement sous la forme d'une hiérarchie de localités, comme dans les calculs à mobilité de sites, mais "logique" : on peut considérer qu'il existe un lien entre deux processus lorsque ces deux processus connaissent un nom commun de canal et sont disposés à communiquer sur ce canal.

Si l'on cherche à modéliser un système à l'aide du π-calcul, un canal peut aussi bien représenter un fichier qu'une variable, un terminal, un port de communication ou une entrée dans une table de méthodes virtuelles. Par contre, les canaux ne sont pas, a priori, une construction appropriée pour décrire un processus Un*x ou un système de fichiers hiérarchique.

Notons que le π-calcul propose, à notre avis, des primitives agréables et adaptées à la programmation et au contrôle de processus. Ainsi, ce calcul a fait l'objet d'extensions "appliquées" [28] et de plusieurs implantations sous la forme de langages de programmation à part entière, notamment sous la forme de Nomadic Pict [85] et Microsoft Biztalk [84].

6.1.1 Le langage

Pour des raisons de simplicité, nous présentons une version monadique et synchrone du π-calcul, dans laquelle la réplication est gardée et sans test d'égalité.

La syntaxe du π-calcul est présentée sur la figure 6.1. On suppose l'existence d'un ensemble strictement dénombrable a, b, $c\ldots$ de noms. On emploiera P, $Q\ldots$ pour désigner les processus.

Le seul symbole inédit est $+$, l'opérateur de choix. Un processus $\alpha.P + \beta.Q$ se comportera, au choix, comme $\alpha.P$, si l'action α peut avoir lieu, ou comme $\beta.Q$, si l'action β peut avoir lieu. Les notions de variables libres et liées et de substitution sont sans surprises.

Nous utiliserons les mêmes abréviations que dans les calculs précédents.

La congruence structurelle est la plus petite relation $\equiv$ qui soit une relation congruence compatible avec les règles de la figure 6.2, telle que $(P/\equiv, |, \mathbf{0})$ soit un monoïde commutatif et associatif et qui soit compatible avec l'α-équivalence des variables liées.

Les règles de réduction sont définies sur la figure 6.3.

$$\textsc{Struct-Repl-Par}\ P \equiv\, !P|P$$

$$\textsc{Struct-Res-Res}\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \text{ si } n \neq m$$

$$\textsc{Struct-Res-Par}\ (\nu n)(P|Q) \equiv P|(\nu n)Q \qquad \text{si } n \notin fv(P)$$

$$\textsc{Struct-Zero-Res}\ (\nu n)\mathbf{0} \equiv \mathbf{0} \qquad \textsc{Struct-Zero-Repl}\ !\mathbf{0} \equiv \mathbf{0}$$

$$\textsc{Struct-Sum-Dist}\ \frac{Q \equiv R}{P + Q \equiv P + R} \qquad \textsc{Struct-Sum-Com}\ P + Q \equiv Q + P$$

$$\textsc{Struct-Sim-Nil}\ P + \mathbf{0} \equiv P$$

FIG. 6.2 – Congruence structurelle dans le π-calcul

.

$$\textsc{Red-Comm}\ (\overline{a}\langle b\rangle.P + Q) \mid (a(x).R + S) \longrightarrow P \mid Q \mid R\{x \leftarrow b\} \mid S$$

$$\textsc{Red-Par}\ \frac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$$

$$\textsc{Red-Struct}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q} \qquad \textsc{Red-Res}\ \frac{P \longrightarrow Q}{(\nu n)P \longrightarrow (\nu n)Q}$$

FIG. 6.3 – Règles de réduction dans le π-calcul

.

6.1.2 Exemples

Communication diadique

Le π-calcul que nous avons présenté est purement monadique, au sens où une communication ne peut servir qu'à transmettre un seul nom à la fois. Cepen-

$$\blacksquare_l \triangleq (\nu c : N_c)\overline{l}\langle c\rangle \qquad \texttt{Heap}_n\ l \triangleq \underbrace{\blacksquare_l \mid \blacksquare_l \mid \ldots \mid \blacksquare_l}_{n}$$

$$\texttt{Loop}\ l \triangleq !alloc(r).(\nu d : N_d)l(c).\overline{r}\langle(c,d)\rangle.d(\_).\overline{l}\langle c\rangle$$

$$\texttt{BRM}_n \triangleq (\nu l).(\texttt{Loop}\ l \mid \texttt{Heap}_n\ l)$$

FIG. 6.4 – Gestionnaire de ressources en π-calcul.

dant, pour des raisons de lisibilité, il est souvent intéressant d'échanger plusieurs informations en une seule communication. Pour ce faire, il est possible d'écrire quelques macros très simples.

$$\begin{array}{lcl} \texttt{Send}\ (u,v)\ \texttt{on}\ a.P & = & (\nu r : T_r)\overline{a}\langle r\rangle.\overline{r}\langle u\rangle.\overline{r}\langle v\rangle.P \\ & & \text{avec } r \notin fv(P) \cup \{u,v\} \\ \texttt{Receive}\ (x,y)\ \texttt{on}\ a.Q & = & a(r).r(x).r(y).Q \\ & & \text{avec } r \notin fv(Q) \cup \{x,y\} \end{array}$$

Ainsi, pour émettre u et v sur le canal a, de manière à éviter les interférences avec d'autres communications, `Send` (u,v) `on` $a.P$ commence par créer un nouveau nom de canal r. C'est ce nom qui est communiqué sur a, permettant ainsi à l'émetteur et au récepteur d'isoler leurs échanges, puisque seuls ces deux processus connaissent r. Ensuite, les valeurs u et v sont communiquées sur ce canal.

Nous reviendrons plus tard sur le typage de ces macros.

Gestionnaire de ressources

Sous la majorité des systèmes d'exploitation, lorsqu'un agent désire obtenir une ressource, il doit la demander au système d'exploitation par le biais d'un appel système tel que `fopen` ou `fork`. À son tour, le système est responsable d'allouer la ressource – ou de refuser – puis de garder trace de ce qu'utilise l'agent et enfin, le moment venu, de procéder à la désallocation.

La figure 6.4 présente un modèle possible de gestionnaire de ressources en π-calcul – pour simplifier les notations, nous avons supposé notre π-calcul polyadique. Comme ce modèle se rapproche de certaines des constructions de BoCa, nous avons réutilisé la notation $\blacksquare$.

Le symbole $\blacksquare$ représente une unité de ressource, dont est constituée la réserve $\texttt{Heap}_n$. Un client qui désire obtenir une ressource doit émettre un nom a sur le canal prédéfini *alloc*. Deux nouveaux noms c et d sont alors créés. Le canal c représente l'entité allouée par le système tandis que le canal d représente son désallocateur. Lorsque le client a terminé d'utiliser c, il doit émettre sur d pour prévenir le système que la ressource peut être libérée. La ressource est alors replacée dans la réserve.

Comme en `C`, par exemple, même si la ressource a été rendue, le client peut continuer à l'utiliser – à ses risques et périls. Et comme en `C`, dans le cas général, il n'est pas possible de vérifier statiquement si les ressources sont effectivement

On notera τ un préfixe qui se réduit sans rien faire et on supposera que c occupe k ressources.

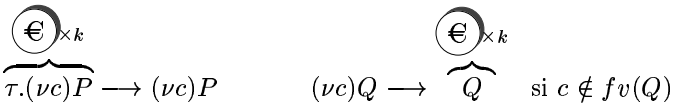


FIG. 6.5 – Ressources dans le π-calcul.

toutes rendues après avoir été utilisées et uniquement lorsque leur utilisation est terminée.

6.1.3 Ressources

Les canaux de communication du π-calcul permettent de représenter naturellement un certain nombre d'objets utilisés couramment dans la conception de logiciels ou de matériels. Ainsi, de nombreux points de vue, un socket Un*x est largement identique à un canal de communication. De même, les opérations d'entrée/sortie vers un fichier sont proches des opérations de communication sur un canal. À un plus haut niveau d'abstraction, on emploiera aussi des canaux de communication pour figurer les entrées dans une librairie dynamique ou dans une table de méthodes virtuelles, ou même pour représenter les communications entre un utilisateur et un programme par le biais d'un terminal ou d'une interface graphique.

Par conséquent, qui contrôle les communications contrôle le système [66]. Les canaux du π-calcul constituent donc une notion naturelle de ressource. Si certains canaux sont considérés comme prédéfinis, leur création dynamique peut être assimilée à la création d'un nouveau nom par la primitive (ν). Ainsi, un processus $(\nu c)P$ est un processus qui possède l'entité c, qui elle-même occupe une certaine quantité de ressources, et qui se comporte comme le processus P. C'est ce qui nous conduit à la définition suivante (illustrée sur la figure 6.5) :

Définition 15 (Ressources – π-calcul)
Dans le π-calcul, les entités allouées et désallouées sont les noms. La réserve est commune à tout le terme examiné. La seule opération d'allocation est (ν). La désallocation fait disparaître les (ν) inutile. Elle est automatique et n'est pas représentée dans le calcul.

Ainsi, même si $(\nu c)P$ et P sont deux processus qui, isolés, réagissent de la même manière, le premier nécessite une ressources pour s'exécuter. Cette conception des ressources nécessite d'apporter quelques retouches au π-calcul, notamment de supprimer la congruence $(\nu c)\mathbf{0} \equiv \mathbf{0}$. En effet, si une transition $(\nu c)\mathbf{0} \to \mathbf{0}$ reste compatible avec notre point de vue, $\mathbf{0} \to (\nu c)\mathbf{0}$ reviendrait à allouer des ressources sans raison à un processus terminé.

Notons aussi que le concept de la réserve commune à tout le terme peut se révéler problématique. En effet, contrairement à ce que nous avions dans le cas des calculs à mobilité de sites, en π-calcul, si deux processus P et Q sont ressource-contrôlés, rien n'assurera que $P|Q$ soit ressource-contrôlé.

6.2 L'approche par les types linéaires

Dans de nombreux formalismes et langages de programmation, il peut être intéressant de déterminer si une entité n'est utilisée qu'une seule fois, ou qu'un nombre fini de fois. Ainsi, une fonction définie mais jamais invoquée pourra être effacée, tandis qu'une fonction invoquée uniquement une ou deux fois pourra être recopiée en ligne (procédé d'inlining) pour éviter un certain nombre d'opérations coûteuses [56]. De même, si un programme n'accède qu'une seule fois à une donnée, il ne sert à rien de la copier dans un cache – et s'il s'agit d'une référence, il est plus utile de la passer directement au ramasse-miettes [6], une fois qu'elle a été utilisée. Enfin, si seul un processus est capable d'acquérir un verrou, il ne sert à rien de procéder effectivement aux opérations complexes d'acquisition et de libération [6].

Ainsi, il est fréquent, dans le π-calcul et ses variantes [82], de créer des canaux et de ne les employer que pour une et une seule communication. Considérons par exemple une méthode habituelle utilisée pour implanter un appel de fonction. Pour représenter la fonction f à un seul argument, on utilisera un canal diadique (i.e. par lequel on peut transmettre des couples) c_f. Lors d'une invocation de f, on émettra alors sur c_f un couple composé de l'argument x de f et d'un canal de retour r, qui sera utilisé pour permettre à f de renvoyer la valeur de retour de la fonction. Ainsi, une invocation de la fonction f prendra la forme $(\nu r)\overline{c_f}\langle x, r\rangle.r(y).P$ où r n'est pas libre dans P et n'est pas réutilisé par un autre processus.

La propriété de n'être utilisée qu'une seule fois est une propriété de linéarité, par assimilation avec les logiques linéaires, dans lesquelles chaque hypothèse ne peut être utilisée qu'un nombre limité de fois. Afin de déterminer quelles entités sont linéaires, de nombreux formalismes – et même certaines implantations du π-calcul [86] – disposent de systèmes de types linéaires [83] ou quasi-linéaires [50]. Dans ces systèmes de types, les entités sont généralement annotées par un nombre qui détermine le nombre d'accès possibles – souvent approximé par les deux catégories "au plus une fois" et "aucune borne". Nous présentons dans cette section un tel système pour le π-calcul [53], proposé par Naoki Kobayashi, Benjamin C. Pierce et David N. Turner.

6.2.1 Le langage

Avant de passer au système de types, il est nécessaire de préciser quelques mots sur le langage lui-même : il s'agit d'un π-calcul asynchrone et dont la réplication peut être gardée par une lecture. Pour cette présentation, nous nous contenterons d'une version monadique, sachant que la polyadicité n'ajoute aucune réelle complication[1].

[1] Il est tout de même nécessaire d'ajouter une règle *linéaire* pour typer les n-uplets.

$$\begin{array}{rcl} T & ::= & Ssh \\ & | & Chan(T,p,m) \quad p \in \{\emptyset,\{r\},\{w\},\{r,w\}\}, m \in \{1,\omega\} \end{array}$$

FIG. 6.6 – Grammaire des types linéaires pour le π-calcul.

$$\begin{array}{rcl} P,Q & ::= & \mathbf{0} \\ & | & P|Q \\ & | & a(b).P \\ & | & \overline{a}\langle b\rangle.P \\ & | & !a(x).P \\ & | & P+Q \\ & | & (\nu c : C)P \end{array}$$

La définition des variables libres et liées, du renommage, de la congruence structurelle et de la réduction sont identiques à celles que nous avons présentées pour le π-calcul dans la section 6.1.1.

Notons que nous avons conservé l'aspect asynchrone et cette forme de réplication gardée pour des raisons historiques plus que techniques. Enfin, remarquons que nous avons modifié en plusieurs lieux la syntaxe employée dans l'article originel [53] pour garder une certaine conformité avec le reste de cet exposé.

6.2.2 Système de types

Nous présentons sur la figure 6.6 la grammaire de types. Contrairement à la majorité des systèmes de types que nous présentons dans cet exposé, un processus sera considéré comme typable ou non mais ne se verra pas associé un type particulier. Un canal de communication c sera de type $Chan(T,p,m)$ s'il peut servir à communiquer des informations de type T, si sa polarité est p et sa multiplicité m. La polarité de c est $\cdot$ s'il est interdit d'utiliser c, r s'il est autorisé d'utiliser c pour lire mais pas pour écrire, w dans le cas dual et rw si c peut être utilisé pour lire aussi bien que pour écrire. La multiplicité de c est 1 si c ne peut être utilisé qu'une seule fois *exactement* ou ω dans le cas contraire. Notons qu'un type $Chan(\_,\emptyset,1)$ n'a pas réellement de sens, puisqu'il impose de communiquer une fois sur un canal sur lequel il est interdit de lire aussi bien que d'écrire. On assimile donc $Chan(\_,\emptyset,1)$ à $Chan(\_,\emptyset,\omega)$. Notons aussi que ce système de types ne suffit pas à garantir qu'une communication peut avoir lieu, car il ne peut prévoir les deadlocks, si bien que l'obligation de lire ou d'écrire sur un canal c se restreint en fait à une obligation de contenir une lecture ou une écriture sur c, qui peut ne pas se produire effectivement.

La notion d'environnement de typage est plus complexe que dans les autres systèmes de types que nous présentons dans cet exposé, puisqu'il s'agit d'en ensemble d'hypothèses *linéaires* – au sens de la logique linéaire – et donc utilisables une seule fois chacune. En particulier, cela signifie, comme nous le verrons avec la règle T-PAR, que pour typer $P|Q$, nous sommes obligés de répartir l'environnement de typage entre P et Q.

$$\text{T-Par}\ \frac{\Gamma_P \vdash P \qquad \Gamma_Q \vdash Q}{\Gamma_P + \Gamma_Q \vdash P|Q} \qquad \text{T-Nil}\ \frac{\Gamma \text{ illimité}}{\Gamma \vdash \mathbf{0}}$$

$$\text{T-Write}\ \frac{\Gamma \text{ illimité}}{\Gamma + x : Chan(T, \{w\}, m) + y : T \vdash \overline{x}\langle y\rangle}$$

$$\text{T-Read}\ \frac{\Gamma, z : T \vdash P}{\Gamma + x : Chan(T, \{r\}, m) \vdash x(z).P}$$

$$\text{T-ReadRepl}\ \frac{\Gamma, z : T \vdash P \qquad \Gamma \text{ illimité}}{\Gamma + x : Chan(T, \{r\}, \omega) \vdash !x(z).P}$$

$$\text{T-New}\ \frac{\Gamma, x : Chan(T, p, m) \vdash P \qquad p = rw \text{ ou } p = \cdot}{\Gamma \vdash (\nu x : Chan(T, p, m))P}$$

$$\text{T-Sum}\ \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P + Q}$$

FIG. 6.7 – Règles de typage linéaire pour le π-calcul.

L'addition sur les types se définit par

$$\begin{cases} Ssh + Ssh & = & Ssh \\ Chan(T, p, \omega) + Chan(T, q, \omega) & = & Chan(T, p \cup q, \omega) \\ Chan(T, p, 1) + Chan(T, q, 1) & = & Chan(T, p \cup q, 1) \text{ si } p \cap q = \emptyset. \end{cases}$$

Cette addition s'étend aux environnements par

$$\begin{cases} (\Gamma, x : T_1) + x : T_2 & = & \Gamma, x : T_1 + T_2 \\ \Gamma + x : T & = & \Gamma, x : T \text{ si } x \notin dom(\Gamma) \\ (\Gamma_1, x : T_1) + (\Gamma_2, x : T_2) & = & (\Gamma_1 + \Gamma_2), x : (T_1 + T_2) \\ & & \quad \text{si } dom(\Gamma_1) = dom(\Gamma_2) \\ \ldots & & \end{cases}$$

Enfin, nous dirons qu'un environnement de typage Γ est illimité si Γ ne contient aucun type $Chan(\_, p, 1)$ avec $p \neq \emptyset$. En d'autres termes, si Γ ne spécifie pas une obligation de communiquer sur un canal.

Les règles de typage sont alors présentées sur la figure 6.7.

La règle T-Par, comme nous l'avons expliqué plus haut, permet de répartir les hypothèses entre P et Q pour typer $P|Q$. Ainsi, si $\Gamma \vdash c : Chan(T, \{r, w\}, 1)$, il est possible d'écrire une seule fois sur c et de lire une seule fois sur c. Il est donc nécessaire de répartir la lecture entre P et Q – sachant que les deux opérations peuvent avoir lieu toutes les deux dans P ou toutes les deux dans Q.

La règle T-Nil spécifie que, si Γ ne garantit pas qu'une communication doit avoir lieu, le processus terminé est typable dans Γ.

La règle T-Write permet de typer un processus de la forme $\overline{x}\langle y\rangle$. Pour ce faire, y doit être du type T des noms qui peuvent transiter par x. En plus, il

est nécessaire d'avoir l'autorisation d'écrire sur x – autorisation qui est ainsi consommée si $m = 1$. De plus, comme $\overline{x}\langle y\rangle$ termine le processus, comme dans T-NIL, on vérifie qu'il ne reste aucune communication imposée.

D'après T-READ, pour lire sur un canal, il est nécessaire de disposer de l'autorisation de recevoir, qui est consommée si $m = 1$. De plus, si c'est un nom de type z qui est lu, il complète l'environnement, de manière naturelle. La lecture répliquée, elle, se type à l'aide de la règle T-READREPL. En plus des conditions précédentes, on vérifie que le nombre de lectures sur le canal n'est pas limité et que l'environnement Γ ne contient aucune obligation de communication limitée – en effet, comme P est répliqué, il est probable que toutes les communications apparaissant dans P aient lieu une infinité de fois.

Remarquons que T-NEW est un peu plus compliquée qu'on pourrait l'attendre, puisque, en plus de mettre à jour l'environnement, cette règle impose de vérifier que les canaux créés, soit ne seront jamais utilisés (si $p = \emptyset$), soit peuvent être utilisés à la fois en lecture et en écriture (si $p = \{r, w\}$). En effet, un canal utilisable uniquement en lecture ou uniquement en écriture sur tout son domaine d'existence ne sera en pratique jamais utilisé.

6.2.3 Propriétés

Afin d'exprimer la propriété suivante, il est nécessaire d'introduire une réduction typée, qui permette d'observer les communications et de consommer les autorisations. Nous ne détaillerons pas les règles qui gouvernent cette sémantique. Nous noterons $\Gamma \vdash P \xrightarrow{m,\alpha} P'$ signifie que P se réduit en P' par une communication sur le canal α (on aura $\alpha = \tau$ si le canal est caché par un (ν)) avec la multiplicité m.

Théorème 10 (Subject Reduction) *Si $\Gamma \vdash P$ et $\Gamma \vdash P \xrightarrow{m,\alpha} P'$ alors :*
- *si $\alpha = \tau$, on a $\Gamma \vdash P'$*
- *si $\alpha = x$ et si $\Gamma = \Gamma' + x : T$, on a $\Gamma' \vdash P'$.*

Théorème 11 (Sûreté des accès) *Si $\Gamma \vdash P$, si $P \longrightarrow^* Q$ et si $Q \equiv (\nu w_1 : T_1, \ldots, w_n : T_n)(R \mid S)$, alors, en notant $\Delta = \Gamma, w_1 : T_1, \ldots, w_n : T_n$,*
- *si $R = \overline{x}\langle y\rangle|x(y).R'$ ou $R = \overline{x}\langle y\rangle|!x(y).R'$, alors $\Delta(x) = Chan(\_, \{r, w\}, \_)$*
- *si $R = \overline{x}\langle y\rangle|\overline{x}\langle z\rangle$ ou $R = x(y).R_1|x(z).R_2$, alors $\Delta(x) = Chan(\_, \_, \omega)$*
- *si $R =!x(y).R'$, alors $\Delta(x) = Chan(\_, \_, \omega)$.*

En d'autres termes, les communications vérifient les hypothèses de Δ.

6.2.4 Discussion

Le système de types linéaires que nous venons de présenter permet de déterminer quand un canal ne peut être utilisé qu'une fois au plus, ce qui peut être utile pour optimiser une implantation ou étudier un terme. Si ce système ne différencie que "un seul usage" et "n'importe quel nombre de communications" et impose un calcul asynchrone, les idées peuvent être étendues à des types quasi-linéaires et à un calcul synchrone.

L'idée originale de ce travail, à savoir la possibilité de diviser une hypothèse en deux afin de séparer lecture et écriture, est intéressante et étend naturellement les principes de la logique linéaire.

L'ensemble est assez proche de nos techniques et il serait probablement possible de reformuler nos différents systèmes de types pour les rapprocher de la logique linéaire. Cependant, nos méthodes mélangent fréquemment hypothèses hypothèses locales et uniques (i.e. une borne sur la consommation en ressources d'un processus) et hypothèses réutilisables (i.e. encombrement d'une entité), ce qui risque de rendre l'ensemble peu lisible.

Nous verrons dans la section 6.3.2 une situation dans laquelle ce système de types complète de manière utile nos propres outils et dans la section 10.3 une manière d'approximer les typage linéaire à l'aide de nos propres techniques.

6.3 Controlled π-calculus

Dans cette section, nous introduisons le Controlled π-calculus, ou Cπ, une algèbre de processus créé pour permettre de développer des protocoles ressource-contrôlés dans le cadre du π-calcul.

La conception des ressources étend celle que nous avons présenté dans la définition 15 (cf. section 3.1.3). Ainsi, si les ressources ne sont pas représentées explicitement, l'opérateur (ν) délimite le domaine de définition d'un canal de communication, c'est-à-dire de l'entité qui occupe des ressources. Le π-calcul se comporte alors comme un langage de programmation muni d'un ramasse-miettes, auquel nous ajoutons une opération de finalisation ($\daleth$) (il s'agit de la lettre hébreue daleth, que nous prononcerons 'delete' pour les besoins de la cause). Ainsi, si c n'est pas libre dans P, on aura

$$(\nu c)(\daleth c).P \longrightarrow P \ .$$

En d'autres termes, de la même manière qu'en Java, OCaml ou C#, lorsqu'une entité n'est plus utile, en plus de désallouer les ressources qu'elle occupe, on invoque une méthode/une fonction/un processus conçu(e) pour cet usage. En Cπ et au contraire de ces langages, il est impossible de *ressusciter* à l'aide d'un finaliseur une entité qui était préalablement considérée comme inutilisée [5] – un événement qui peut se produire dans les langages dotés de garbage-collection et conduire à des situations ingérables.

Dans le cadre de systèmes concurrents, la garbage-collection pose des problèmes supplémentaires. Notamment, si un processus s'arrête suite à un deadlock ou entre dans une boucle infinie suite à un livelock, par exemple, une partie des ressources qu'il semble utiliser ne lui serviront en fait à rien mais ne seront *a priori* jamais libérées non plus. Ainsi, un processus aussi simple que $(\nu c)c(x).P$, dont l'effet est identique à celui de $(\nu c)\mathbf{0}$, mentionne c et empêche donc la récupération des ressources retenues par c.

Pour gérer ce type de problèmes, nous introduisons dans un deuxième temps un mécanisme de nettoyage. Plutôt que de choisir un algorithme précis pour déterminer quels sont les termes qui peuvent être supprimés, nous définissons une notion de *processus mort* à l'aide de simulations barbelées et nous paramétrons Cπ par une relation de nettoyage conforme à cette définition.

6.3.1 Langage de base

La syntaxe du Cπ est présentée sur la figure 6.9, avec les hypothèses habituelles. On suppose de plus l'existence d'un nom spécial $\circledcirc$ (ou "null"), qui est

Dans chacun des cas, on notera τ un préfixe qui se réduit sans rien faire et on supposera que c occupe k ressources.

Exemple 1

$$\overbrace{\tau.(\nu c)P}^{€\times k} \longrightarrow (\nu c)P$$

Exemple 2

$$(\nu c)\tau.(\neg c).Q \longrightarrow \overbrace{Q}^{€\times k} \quad \text{si } c \notin fv(Q)$$

Exemple 3

$$\overbrace{\tau.(\nu c)\overline{a}\langle c\rangle.(\neg c).P \mid a(x).Q}^{€\times k} \begin{array}{l} \longrightarrow (\nu c)\overline{a}\langle c\rangle.(\neg c).P \mid a(x).Q \\ \longrightarrow (\nu c)((\neg c).P \mid Q') \\ \longrightarrow^* (\nu c)(\neg c).P \\ \longrightarrow \underbrace{P}_{€\times k} \text{ si } c \notin fv(P) \text{ et } c \notin fv(Q') \end{array}$$

FIG. 6.8 – Ressources dans Cπ.

un canal sur lequel il n'est pas possible de communiquer.

Le seul nouveau symbole est ($\neg$), l'opérateur de finalisation : $(\neg c).P$ attend que c soit désalloué pour exécuter P.

La définition des variables libres et liées s'étend naturellement : c est libre dans $(\neg c)P$. Nous emploierons les raccourcis syntaxiques habituels. Notamment, si les annotations de type n'interviennent pas dans un raisonnement, nous nous réserverons le droit de les omettre de la syntaxe.

La congruence structurelle et les règles de réduction sont définies respectivement sur les figure 6.10 et 6.11. Par rapport au π-calcul, on remplace STRUCT-ZERO-RES par R-ZERO-RES, on ajoute R-DEL, qui spécifie que deux finaliseurs pour le même nom peuvent être réécrits comme un seul finaliseur combiné.

De la même manière que $(\nu c)\mathbf{0} \equiv \mathbf{0}$ n'est pas valide dans notre conception des ressources, la règle de congruence structurelle $a(x).(\nu b)P \equiv (\nu b)a(x).P$, présente dans certaines variantes du π-calcul, serait incorrecte dans Cπ. En effet, $a(x).(\nu b)P$ alloue des ressources pour contenir b après avoir communiqué sur a, tandis que $(\nu b)a(x).P$ a déjà alloué les ressources au moment de la communication. En particulier, si la communication sur a n'a jamais lieu, le premier processus ne consommera pas de ressources, contrairement au deuxième.

Enfin, la règle R-FIN spécifie le fonctionnement d'un finaliseur : si les seules occurrences de l'entité c apparaissent dans le finaliseur de c, cela signifie que c

$$
\begin{array}{rcl}
P, Q & ::= & \mathbf{0} \\
& | & P|Q \\
& | & \alpha.P \\
& | & !\alpha.P \\
& | & P + Q \\
& | & (\nu c : C)P \\
& | & (\neg c).P
\end{array}
\qquad\qquad
\begin{array}{rcl}
\alpha, \beta & ::= & \overline{a}\langle b \rangle \\
& | & a(b)
\end{array}
$$

FIG. 6.9 – Syntaxe du Cπ

STRUCT-REPL-PAR $P \equiv !P|P$

STRUCT-RES-RES $(\nu n : C)(\nu m : C)P \equiv (\nu m : C)(\nu n : C)P$ SI $n \neq m$

STRUCT-RES-PAR $(\nu n : C)(P|Q) \equiv P|(\nu n : C)Q$ si $n \notin fv(P)$

STRUCT-ZERO-REPL $!\mathbf{0} \equiv \mathbf{0}$ STRUCT-SUM-COM $P + Q \equiv Q + P$

STRUCT-SUM-NIL $P + \mathbf{0} \equiv P$

FIG. 6.10 – Congruence structurelle dans le Cπ

.

R-COMM $(\overline{a}\langle b \rangle.P + Q) \mid (a(x).R + S) \longrightarrow P \mid R\{x \leftarrow b\}$ $a \neq \circledcirc$

R-PAR $\dfrac{P \longrightarrow Q}{P|R \longrightarrow Q|R}$ R-STRUCT $\dfrac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}$

R-RES $\dfrac{P \longrightarrow Q}{(\nu n)P \longrightarrow (\nu n)Q}$ R-FIN $(\nu c : C)(\neg c)P \longrightarrow P\{c \leftarrow \circledcirc\}$

R-DEL $(\neg c)P \mid (\neg c)Q \longrightarrow (\neg c)(P \mid Q)$ R-ZERO-RES $(\nu c : C)\mathbf{0} \longrightarrow \mathbf{0}$

FIG. 6.11 – Réduction dans le Cπ

.

n'est plus utilisé et peut être nettoyé. En particulier, il faut donc exécuter le finaliseur de c. Nous substituons $\circledcirc$ à c pour assurer que l'entité est effectivement détruite. Cela permet d'éviter des situations de résurrection telles que celles qu'on peut rencontrer dans de nombreux langages dotés de garbage-collection.

Notons enfin que $\circledcirc(x).P$ est un processus qui attend éternellement sur un canal sur lequel il n'y a aucune communication et qui donc ne se réduit jamais – nous modifierons ce comportement du langage dans la section 6.3.3 pour permettre $\circledcirc(x).P \longrightarrow^* \mathbf{0}$, de la même manière qu'un processus qui essaye de lire ou d'écrire sur `null` s'achève sur un SEGFAULT.

Variantes Si nous avions cherché à nous approcher de la sémantique de ces langages, nous aurions pu définir au lieu de R-FIN une règle telle que R-FIN-RESURRECTION-REJECTED, dans laquelle le finaliseur peut effectivement ressusciter une variable :

$$\text{R-FIN-RESURRECTION-REJECTED } (\nu c : C)(\urcorner c)P \longrightarrow (\nu c : C)P \ .$$

Nous avons préféré interdire cette possibilité car, en pratique, elle nous semble ôter tout intérêt à la finalisation.

Nous aurions aussi pu imposer de ne désallouer que les entités qui ne figurent pas après le finaliseur

$$\text{R-FIN-NON-FREE-REJECTED } (\nu c : C)(\urcorner c)P \longrightarrow P \quad c \notin fv(P)$$

mais cette règle nous a semblé moins intéressante, notamment car $\circledcirc$ peut servir à lever des exceptions ou à prouver certaines propriétés intéressantes (cf. section 6.3.2). Ainsi, si en examinant un terme, on trouve un processus $\circledcirc(x).P$, on peut raisonnablement considérer que quelque chose, quelque part s'est mal passé et que le programmeur tente d'utiliser une variable déjà désallouée – en d'autres termes, sous Un*x, SEGFAULT. À l'inverse, si l'on adopte R-FIN-NON-FREE-REJECTED, lorsqu'on trouve un processus $(\nu c : C)(\urcorner c)P$, il devient nécessaire, en plus, de vérifier si $c \in fv(P)$ avant de pouvoir conclure à une erreur.

Il aurait encore été possible d'interdire de construire des termes $(\urcorner c).P$ tels que c apparaisse dans P et d'imposer que les substitutions de c ne traversent pas $(\urcorner c)$. Ainsi, on aurait $((\urcorner c).P)\{x \leftarrow c\} = (\urcorner c).P$. Il ne s'agit malheureusement pas d'une solution : comme on peut le constater sur le terme $a(x).b(y).(\urcorner y).P \mid \overline{a}\langle c\rangle.\overline{b}\langle c\rangle$, pour peu que x soit libre dans P, il est aisé de contourner la restriction et de se retrouver dans un cas problématique.

Enfin, on pourrait songer à remplacer R-FIN par une règle de congruence

$$\text{STRUCT-FIN-REJECTED } (\nu c : C)(\urcorner c)P \equiv P\{c \leftarrow \circledcirc\} \ .$$

Malheureusement, cette règle donnerait des résultats aberrants puisque nous aurions $(\nu c)(\urcorner c).P \mid (\nu d)(\urcorner d).Q \equiv (\nu e)(\urcorner e)(P\{c \leftarrow e\} \mid Q\{d \leftarrow e\})$.

Il serait probablement possible, par contre, d'écrire

$$\text{STRUCT-FIN-REJECTED } (\nu c : C)(\urcorner c)P \equiv \texttt{name}\ c.P\{c \leftarrow \circledcirc_c\}$$

où `name` c est un lieur de c similaire à (νc) mais qui n'alloue pas de ressources et qui ne se propage pas aux processus parallèles et où $\circledcirc_c$ est un canal sur lequel on ne peut communiquer. En d'autres termes, il s'agirait de découpler le masquage de nom de la création d'un nouveau canal. Cette piste nous semble envisageable, quoique plus complexe que celle que nous avons choisie pour ce travail.

6.3.2 Exemples

Exemple simple

Considérons le processus $A \triangleq (\nu c : C)(\overline{a}\langle c\rangle.(\urcorner c).P) \mid a(x).(\urcorner x).R$. Ce processus alloue des ressources à l'entité c puis émet c sur le canal prédéfini a. Grâce à STRUCT-RES-PAR, A évolue donc en $(\nu c : C)((\urcorner c).P \mid (\urcorner c).R)$. Les deux finaliseurs de c fusionnent en un seul et deviennent $(\nu c : C)(\urcorner c)(P \mid R)$. Le terme ainsi obtenu ne contient aucune occurrence de c hors du finaliseur, ce qui signifie que c peut être désalloué. Après désallocation, on obtient $P\{c \leftarrow \circledcirc\} \mid R\{c \leftarrow \circledcirc\}$.

Communication diadique

À l'aide de la finalisation, nous pouvons reprendre les macros de communication diadique développées dans la section 6.1.2.

Ainsi, considérons les macros suivantes :

$$\begin{array}{lcl} \texttt{Send}\ (u,v)\ \texttt{on}\ a.P & = & (\nu r : T_r)\overline{a}\langle r\rangle.\overline{r}\langle u\rangle.\overline{r}\langle v\rangle.(\urcorner r).P \\ \texttt{Receive}\ (x,y)\ \texttt{on}\ a.Q & = & a(r).r(x).r(y).(\urcorner r).Q\ . \end{array}$$

En plus de permettre la communication diadique, comme les macros que nous avions présentées, ces constructions permettent de garantir que le nom r ne sera plus utilisé après cette fin de communication. Cette propriété, que nous avions vérifiée manuellement dans la version précédente et qui est ici forcée par la substitution $r \leftarrow \circledcirc$, peut s'avérer importante pour plusieurs raisons.

Il peut ainsi s'agir de raisons de sécurité : dans certaines circonstances, il est primordial de garantir qu'un nom secret reste secret. La primitive ($\urcorner$) permet ici de garantir que les processus P et Q ne seront pas exécutés tant que r est en mémoire. Du point de vue d'un utilisateur, cela peut signifier une garantie que ses mots de passe seront effacés par le navigateur avant d'enregistrer les préférences.

Il peut aussi s'agir d'optimisations. En effet, de nombreux travaux, tant dans le π-calcul que dans le domaine des ramasse-miettes, pour ne citer que deux exemples, se fondent sur des propriétés de linéarité : si un nom r n'est utilisé qu'une seule fois (ou deux, ou trois...), il est aisé de déterminer à quel moment les ressources allouées pour r peuvent être désallouées. L'opérateur ($\urcorner$) restreint le domaine de r et permet de déterminer plus aisément le nombre d'utilisations de r (ici, deux communications).

Notons qu'il s'agirait aussi d'un cas parfait d'application d'un système de types quasi-linéaires. Ainsi, ($\urcorner$) peut permettre de limiter la zone d'examen d'un tel système. À l'inverse, à partir du moment où une analyse du terme peut permettre de garantir que, après la deuxième utilisation de r, qui figure dans le processus, r peut être désalloué, ($\urcorner r$) devient une opération simple et qui ne nécessite pas de mettre en jeu des mécanismes de garbage-collection complexe ni de substituer dans le terme P.

Gestionnaire de ressources

À l'aide de la finalisation, nous pouvons réécrire le gestionnaire de ressources pour un langage doté de garbage-collection, comme sur la figure 6.12. Pour simplifier le terme, nous n'avons pas donné de noms individuels aux entités

$$\blacksquare_l \triangleq \overline{l}\langle\diamond\rangle \qquad \texttt{Heap}_n\ l \triangleq \underbrace{\blacksquare_l \mid \ldots \blacksquare_l}_{n \text{ ressources}}$$

$$\texttt{Loop}\ l \triangleq !alloc(r).l(\_).(\nu c : N_c)(\overline{r}\langle c\rangle \mid (\urcorner c).\blacksquare_l)$$

$$\texttt{BRM}_n \triangleq (\nu l : N_l)(\texttt{Heap}_n\ l \mid \texttt{Loop}\ l)$$

FIG. 6.12 – Gestionnaire de ressources

Le comportement est similaire à celui de l'exemple présenté sur la figure 6.4. La différence majeure vient de la disparition du canal de désallocation, remplacé par le mécanisme, plus sûr, de finalisation.

Par suite, confronté à un client $(\nu r)\overline{alloc}\langle r\rangle.r(c).P$, le gestionnaire $\texttt{BRM}_n$ alloue une ressource, communique l'entité correspondante sur r et attend que celle-ci ne soit plus utilisée. Dès que le système réalise que tel est le cas, la ressource est retournée à la réserve $\texttt{Heap}_i$, qui redevient $\texttt{Heap}_{i+1}$ – nous avons noté $\texttt{Heap}_i$ et non $\texttt{Heap}_n$ car cette réserve peut parfaitement avoir alloué/désalloué des ressources à d'autres processus en parallèle.

Considérons un client plus spécifique $\texttt{Voleur} \triangleq (\nu r)\overline{alloc}\langle r\rangle.r(c).c(x)$. Si nous plaçons $\texttt{Voleur}$ en parallèle avec $\texttt{BRM}_n$, une fois que la ressource est allouée et communiquée, $\texttt{Voleur}$ évolue en $c(x)$. Or, seuls le voleur et le gestionnaire connaissent c et il suffit de regarder les termes pour constater qu'aucun des deux n'émettra jamais sur le canal c. Par conséquent, l'action $c(x)$ ne sera jamais exécutée et le nom c continuera à apparaître hors du finaliseur. Ce qui signifie notamment que les ressources occupées par c ne seront jamais désallouées.

D'autres circonstances peuvent empêcher des ressources d'être désallouées alors que les entités qui les occupent ne seront jamais utilisées. Ainsi, par exemple, des deadlocks ou des livelocks peuvent paralyser des processus et rendre les ressources qu'ils conservent inutiles, alors que les noms des canaux inutilisés continuent à apparaître dans le terme. Pour récupérer ces ressources, nous avons besoin d'un mécanisme d'élimination des processus morts.

6.3.3 Élimination des processus morts

Comme nous l'avons vu, dans certains cas, la finalisation dépend de la suppression de processus, lorsque ceux-ci sont "morts" mais présentent toujours des occurrences de noms, empêchant par là la désallocation de ces noms. Dans cette section, nous présentons les méthodes que nous employons dans le cadre de Cπ pour éliminer les processus que nous considérons comme morts.

Notre objectif est de "nettoyer" des systèmes, afin d'en supprimer des processus qui empêchent des ressources d'être désallouées et donc les finaliseurs d'être exécutés. S'il est possible d'envisager une solution qui consiste à considérer l'ensemble du système en train d'être exécuté afin d'en retirer sélectivement des composants, cette solution n'est généralement pas abordable, puisqu'elle implique de figer l'ensemble du système et de l'examiner dans son intégralité. Cette tâche, déjà difficile lorsque tous les processus sont exécutés sur un seul

ordinateur, devient totalement impossible dès qu'ils sont distribués sur un réseau, que ce soit explicitement (cf. section 7.2) ou implicitement. Autant que possible, nous allons donc essayer de procéder à un nettoyage local, c'est-à-dire qui fasse intervenir peu de processus.

Comme dans les programmes impératifs, il est généralement impossible de décider à l'avance si un processus va effectivement être exécuté, ne serait-ce que parce que le comportement du système peut dépendre de choix non déterministes ou de la structure de l'environnement. Ainsi, dans un processus tel que $(\nu a)(\overline{a}\langle\circledcirc\rangle \mid a(x).P \mid a(x).Q)$, il se peut que P soit exécuté et non Q, que Q soit exécuté et non P, ou même que les deux soient exécutés, selon le contenu de P et Q. Par conséquent, et même si elle peut être aidée ou approximée par des transformations statiques [52], l'élimination des processus morts est une tâche dynamique [21].

Considérons maintenant le terme

$$D \triangleq a(x).\overline{b}\langle x\rangle.P \mid b(v).\overline{a}\langle x\rangle.Q \ .$$

Ce processus présente un deadlock classique : un processus attend de recevoir un nom sur b pour émettre sur a tandis que l'autre attend de recevoir un nom sur a pour émettre sur b. Dans D, ni P ni Q ne seront exécutés. Cependant, un simple examen du processus D ne permet pas de conclure à la mort de P et Q. En effet, si l'on exécute $D \mid \overline{a}\langle v\rangle.a(x)$, les processus $P\{x \leftarrow v\}$ et $Q\{x \leftarrow v\}$ seront effectivement exécutés. Ce deadlock est donc déverrouillable. En revanche, dans $(\nu a)(\nu b)D$, P et Q ne seront effectivement jamais exécutés. Afin de déterminer si un processus est mort, il est donc nécessaire d'examiner un terme "suffisamment clos" pour que l'examen reste valable lors d'un passage au contexte.

Nous allons donc chercher à examiner des termes suffisamment clos pour qu'il soit possible de déduire qu'un processus est inutile – l'intuition, pour ce qui suit, est que, dans la portée syntaxique du nom a, le processus P est mort si exécuter P nécessite une action sur a et si supprimer P n'empêche pas le terme de s'exécuter.

Formalisons "exécuter P nécessite une action sur a" :

Définition 16 (Processus gardé)
Un processus P est dit gardé par le nom a, ce que nous noterons $P_{:a}$, si
- $P = \overline{a}\langle x\rangle.Q$, $P = a(x).Q$
- $P = !Q$ *avec* $Q_{:a}$
- $P = (\nu b : C)Q$ *avec* $Q_{:a}$ *et* $b \neq a$
- $P = Q + R$ *avec* $Q_{:a}$ *et* $R_{:a}$
- $P = (\urcorner a).Q$

Notons que le processus $(\urcorner a).P$ est considéré comme gardé par a. En effet, un terme tel que

$$(\nu a)(\nu b)((\urcorner a).(\overline{b}\langle c\rangle \mid b(x).P) \mid (\urcorner b).(\overline{a}\langle c\rangle \mid a(x).Q)) \ ,$$

présente un deadlock sur les finaliseurs. Il est ici nécessaire de supprimer l'une des deux finalisations pour pouvoir récupérer les ressources.

Les définitions suivantes sont des extensions de notions classiques du π-calcul, auxquelles on ajoute la gestion de la finalisation :

Définition 17 (Barbelé)
On notera $P \downarrow_a$ (P présente un barbelé a) si $P \equiv (\nu \overrightarrow{b})(a(x).Q + R \mid S)$ avec $a \notin \overrightarrow{b}$. On notera $P \downarrow_{\overline{a}}$ (P présente un barbelé $\overline{a}$) si $P \equiv (\nu \overrightarrow{b})(\overline{a}\langle x\rangle.Q + R \mid S)$ avec $a \notin \overrightarrow{b}$. Enfin, on notera $P \downarrow_{\neg a}$ (P présente un barbelé $\neg a$) si $P \equiv (\nu \overrightarrow{b})((\urcorner a).Q \mid S)$ avec $a \notin \overrightarrow{b}$. On désignera les barbelés par la lettre η.

Si les barbelés a et $\overline{a}$ sont habituels, la possibilité d'observer $\neg a$ est plus surprenante, puisqu'elle nous permet de distinguer $(\urcorner a).P$ de P. En effet, de notre point de vue, de la même manière que les barbelés prennent en compte les synchronisations entre $c(x).A$ et $\overline{c}\langle y\rangle.B$ en exhibant les interactions c et $\overline{c}$, les barbelés doivent prendre en compte les synchronisations entre $(\urcorner a).A$ et $(\urcorner a).B$ en exhibant l'attente de finalisation $\neg a$.

De plus, il nous semble naturel de vouloir observer une différence entre les processus $(\urcorner a).P$ et $\mathbf{0}$. Si aucun des deux ne peut évoluer seul, $(\urcorner a).P$ est un processus qui "attend" un (νa), tandis que $\mathbf{0}$ n'évoluera pas même sous un (νa).

Enfin, si nous essayons d'écrire un système de transitions étiquetées pour Cπ, nous utiliserons des règles telles que

$$(\urcorner a).P \xrightarrow{\neg a} P\{a \leftarrow \circledcirc\} \qquad \frac{P \xrightarrow{\neg a} P' \qquad Q \xrightarrow{\neg a} Q'}{P|Q \xrightarrow{\neg a} P'|Q'} \qquad \frac{P \xrightarrow{\neg a} P' \qquad a \notin fv(Q)}{P \mid Q \xrightarrow{\neg a} P' \mid Q}$$

Ces règles nécessitent une action $\neg a$, ce qui nous conduit à considérer $\neg a$ comme un barbelé naturel.

Notons cependant que, si la synchronisation doit avoir lieu, il faut que $(\urcorner a)$ apparaisse sous (νa), auquel cas $\neg a$ ne sera pas effectivement observable. Cette remarque nous laisse supposer que la simulation barbelée n'est probablement pas l'outil le plus adapté à la tâche. C'est, cependant, le meilleur que nous ayons trouvé.

Lemme 23 (Les barbelés passent à la congruence)
Soient P et Q deux processus et η un barbelé. Si $P \equiv Q$ et $P \downarrow_\eta$ alors $Q \downarrow_\eta$.

Ce lemme découle directement de la définition des barbelés [55].

Définition 18 (Simulation barbelée forte)
Soit $\mathcal{R}$ une relation entre processus. Si, pour tout P et tout Q tels que $P\mathcal{R}Q$, $P \downarrow_\eta$ implique $Q \downarrow_\eta$ et si, dès que $P \longrightarrow P'$, il existe Q' tel que $Q \longrightarrow Q'$ et $P'\mathcal{R}Q'$, alors on dira que $\mathcal{R}$ est une simulation barbelée forte. On dira alors que Q simule fortement P. On notera $P \overset{\circ}{\ll} Q$.

La simulation barbelée forte nous donne un outil qui permet de remplacer certains processus par d'autres. Cette simulation n'est, malheureusement, pas assez puissante pour nous permettre d'atteindre nos objectifs.

Considérons en effet $P \triangleq (\nu a)(a(x).P' \mid \overline{b}\langle a\rangle)$ et $Q \triangleq (\nu a)\overline{b}\langle a\rangle$, et $\mathcal{R}$ la plus petite relation telle que $P\mathcal{R}Q$. Comme P et Q présentent un seul barbelé b et comme ni P ni Q ne peuvent évoluer, nous pouvons en déduire que $\mathcal{R}$ est une simulation barbelée forte.

Considérons maintenant le processus $S \triangleq b(x).\overline{x}\langle\circledcirc\rangle$. En composant P et S, nous obtenons un processus qui se réduit après plusieurs étapes en $(\nu a)P'\{x \leftarrow$

$$\text{DPE-Null}\ \frac{\circledcirc_{:Q}}{(Q,\mathbf{0})\in\mathcal{D}pe} \qquad \text{DPE-Name}\ \frac{a_{:Q} \qquad (\nu a)(P+Q|R)\ll(\nu a)(P|R)}{((\nu a)(P+Q\mid R),(\nu a)(P|R))\in\mathcal{D}pe}$$

FIG. 6.13 – Élimination des processus morts. Dans chaque cas, le processus considéré comme mort est Q.

$\circledcirc\}$. En composant Q et S, nous obtenons le processus $(\nu a)\overline{a}\langle\circledcirc\rangle$. Pour peu que nous ayons défini P' observable – par exemple avec $P' = \overline{c}\langle\circledcirc\rangle$ – nous serons en mesure de différencier $P \mid S$ et $Q \mid S$.

En d'autres termes, même si Q simule fortement P, et même si nous avons isolé le domaine de a, nous ne pouvons pas remplacer P par Q sans supprimer des comportements possibles du système.

Définition 19 (Préordre barbelé)
Si, pour tout S, $P|S \overset{\circ}{\ll} Q|S$, on dira que Q est plus grand que P. On notera alors $P \ll Q$.

Le terme de préordre barbelé n'est, à notre connaissance, pas canonique – nous l'introduisons en nous inspirant de la dualité entre bisimilarité barbelée et équivalence barbelée.

Notons que, si $P \ll Q$, Q peut s'avérer "plus puissant" que P. Cela s'accorde avec nos objectifs puisque, lorsque nous supprimons un processus mort, nous pouvons ainsi déclencher la désallocation de ressources et donc des processus de finalisation en attente.

Ainsi, considérons $P \triangleq (\nu a)(a(x).P' \mid (\urcorner a)Q')$ et $Q \triangleq (\nu a)(\urcorner a)Q'$. Comme P ne présente aucun barbelé et ne peut pas évoluer et comme $P \mid S$ se comporte comme S, nous pouvons en déduire aisément que $P \ll Q$. Or, Q est plus puissant que P puisque $Q \longrightarrow Q'\{a \leftarrow \circledcirc\}$.

Nous considérerons donc que, pour tout processus P et tout processus Q, si Q est plus grand que P et si Q peut être obtenu uniquement en supprimant des processus morts de P, alors il peut être raisonnable de remplacer Q par P. Cette conception, que nous formalisons à l'aide de la relation $\mathcal{D}pe$, définie sur la figure 6.13, est proche de certains travaux sur l'élimination statique du code inutile dans le π-calcul [52].

Définition 20 (Élimination des processus morts)
La relation $\mathcal{D}pe$ est définie par les règles de la figure 6.13. Si un couple de processus (R,S) est lié par $(R,S) \in \mathcal{D}pe$, on dira que S s'obtient à partir de R par élimination des processus morts. Dans ce cas, le processus représenté par Q dans la règle utilisée pour vérifier que $(R,S) \in \mathcal{D}pe$ est dit mort dans le contexte R.

Ainsi, d'après DPE-Null, nous obtenons $(\circledcirc(x).R, \mathbf{0}) \in \mathcal{D}pe$, ce qui correspond à notre définition de $\circledcirc$: un processus qui essaye de lire depuis un canal désalloué s'achève immédiatement – soit parce qu'il est détecté par l'environnement et supprimé, soit par une erreur de segmentation pour ce processus. De la même manière, nous obtenons $((\nu a)(\mathbf{0} + a(x) \mid \mathbf{0}), (\nu a)(\mathbf{0} \mid \mathbf{0})) \in \mathcal{D}pe$.

Notons que la relation $\mathcal{D}pe$ n'est pas incluse dans $\ll$. En effet, si nous considérons le processus $\circledcirc(x).R$, que $\mathcal{D}pe$ met en relation avec $\mathbf{0}$, nous pouvons aisément constater que le premier de ces termes présente un barbelé $\circledcirc$, par opposition au deuxième. Cependant, nous avons la propriété suivante :

Lemme 24 (Comparaison entre $\ll$ et $\mathcal{D}pe$)
Si $(A, B) \in \mathcal{D}pe$ et si Q n'est pas gardé par $\circledcirc$, alors $A \ll B$.

Ce lemme est trivial à prouver.

La relation $\mathcal{D}pe$ n'est pas la seule possible qui permette de nettoyer des termes de sous-termes inutiles. Notamment, elle se contente d'examiner un processus précis, délimité par (νa) et un sous-processus précis, gardé par a. Il serait aussi envisageable d'examiner l'ensemble du système, garantissant ainsi que le terme examiné ne sera pas placé dans un contexte susceptible d'interagir. Cependant, comme nous l'avons déjà mentionné, si l'on considère l'utilisation possible de l'élimination des processus dans un cadre réel constitué d'un réseau de machines fonctionnant en parallèle, l'option "ensemble du système" implique d'examiner toutes les machines du réseau avant de prendre une décision, ce qui est rarement envisageable pour des réseaux réels.

De même, nous n'autorisons pas l'élimination à supprimer des sous-termes arbitraires (i.e. transformer $\alpha.P$ en $\alpha.\mathbf{0}$, par exemple), par exemple à l'aide d'une règle

$$\text{DPE-Context-Rejected}\ \frac{a_{:Q} \qquad (\nu a)\mathbf{C}\{Q\} \ll (\nu a)\mathbf{C}\{\mathbf{0}\}}{((\nu a)\mathbf{C}\{Q\}, (\nu a)\mathbf{C}\{\mathbf{0}\}) \in \mathcal{D}pe}$$

pour une certaine définition des contextes $\mathbf{C}$.

Nous avons fait ce choix pour une première phase de l'étude, afin de borner nos recherches, car ce type d'opération s'apparente plus, de notre point de vue, à une optimisation de code qu'à une étape de la garbage-collection. Dans une étape ultérieure, il serait probablement intéressant de déterminer l'utilité et la difficulté relative des deux définitions possibles.

D'autres travaux [52] ont préféré l'option contraire.

6.3.4 Exemples

Exemple simple

Considérons le processus $A \triangleq (\nu a)!a(x).B$. Comme B peut empêcher la récupération de ressources, il peut être intéressant de supprimer A. On pourra utiliser DPE-Name dès que $(\nu a)(\mathbf{0}+!a(x).B \mid \mathbf{0}) \ll (\nu a)(\mathbf{0} \mid \mathbf{0})$.

Définissons donc $\mathcal{R}$ comme la plus petite relation close par $\equiv$ et telle que pour tout S, $(\nu a)(\mathbf{0}+!a(x).B \mid \mathbf{0}) \mid S \ \mathcal{R}\ (\nu a)(\mathbf{0} \mid \mathbf{0}) \mid S$.

Cette relation est un préordre barbelé. En effet, on constate aisément que si $P\mathcal{R}Q$, alors P et Q présentent les mêmes barbelés. De plus, si $P\mathcal{R}Q$ avec $P \equiv (\nu a)(\mathbf{0}+!a(x).B \mid \mathbf{0}) \mid S$, on a $Q \equiv (\nu a)(\mathbf{0} \mid \mathbf{0}) \mid S$ et le comportement observable de P comme celui de Q est identique au comportement observable de S. Si nous notons alors $P \longrightarrow P'$, il existera S' tel que $S \longrightarrow S'$ et $P' \equiv (\nu a)(\mathbf{0}+!a(x).B \mid \mathbf{0}) \mid S'$ et $Q \longrightarrow\equiv (\nu a)(\mathbf{0} \mid \mathbf{0}) \mid S'$.

De cette définition, nous pouvons aisément constater que pour tout S,

$$(\nu a)(\mathbf{0}+!a(x).B \mid \mathbf{0}) \mid S\mathcal{R}(\nu a)(\mathbf{0} \mid \mathbf{0}) \mid S \ .$$

En d'autres termes, nous avons

$$(\nu a)(\mathbf{0}+!a(x).B \mid \mathbf{0}) \ll (\nu a)(\mathbf{0} \mid \mathbf{0}) \ .$$

Par suite, comme $!a(x).B$ est gardé par a, à l'aide de DPE-NAME, on peut considérer $!a(x).B$ comme mort et donc remplacer A par $(\nu a)(\mathbf{0} \mid \mathbf{0})$. À son tour, ce processus réduit en $\mathbf{0}$. La relation $\mathcal{D}pe$ est donc suffisamment puissante pour nettoyer A d'un terme.

Deadlock

Considérons maintenant

$$A \triangleq (\nu a, b)(a(x).\overline{b}\langle x\rangle.P \mid b(x).\overline{a}\langle x\rangle.Q \mid (\neg a).R) \ .$$

Pour nettoyer A, on pourra utiliser DPE-NAME dès que

$$(\nu a, b)(a(x).\overline{b}\langle x\rangle.P \mid b(x).\overline{a}\langle x\rangle.Q \mid (\neg a).R) \ll (\nu a, b)(\mathbf{0} \mid b(x).\overline{a}\langle x\rangle.Q) \ .$$

Définissons alors $\mathcal{R}$ comme la plus petite relation close par $\equiv$ et telle que pour tout S,

$$(\nu a, b)(a(x).\overline{b}\langle x\rangle.P|b(x).\overline{a}\langle x\rangle.Q|(\neg a).R)|S \ \mathcal{R} \ (\nu a, b)(\mathbf{0}|b(x).\overline{a}\langle x\rangle.Q|(\neg a).R)|S \ .$$

Cette relation est un préordre barbelé. Par suite, comme $a(x).\overline{b}\langle x\rangle.P$ est gardé par a, DPE-NAME spécifie que l'on peut remplacer A par

$$B \triangleq (\nu a, b)(b(x).\overline{a}\langle x\rangle.Q \mid (\neg a).R) \ .$$

Si nous étudions de nouveau ce terme à l'aide de $\mathcal{D}pe$, nous constatons, comme précédemment, que ce processus peut être remplacé par $(\nu a, b)(\neg a).R$, qui évoluera en $R\{a \leftarrow \circledcirc\}$.

La relation $\mathcal{D}pe$ est donc suffisamment puissante pour nettoyer le deadlock.

6.3.5 Le controlled π-calculus complet

À partir de la définition de base de Cπ, présentée dans la section 6.3.1, et de l'élimination des processus morts, présentée dans la section 6.3.3, nous pouvons construire une nouvelle version de Cπ – le *Cπ complet*.

Pour ce faire, nous conservons toutes les définitions du Cπ de base, auxquelles nous ajoutons une relation d'élimination des processus morts dans la sémantique de réduction. Nous pourrions ainsi ajouter la règle suivante :

$$\text{RED-DPE-REJECTED} \ \frac{(P, Q) \in \mathcal{D}pe}{P \longrightarrow Q}$$

Cependant, comme la relation $\mathcal{D}pe$ est clairement indécidable, il serait maladroit d'imposer à un compilateur ou à un environnement d'exécution de déterminer les termes liés par $\mathcal{D}pe$ afin de pouvoir les réécrire. Nous employons donc un autre mécanisme pour permettre l'élimination des processus morts.

(1) $\circledcirc(x).P \longrightarrow_{dpe1} \mathbf{0}$ (2) $\overline{\circledcirc}\langle x\rangle.P \longrightarrow_{dpe1} \mathbf{0}$

(3) $\dfrac{\text{Les seules occurrences de } a \text{ dans } P \text{ sont de la forme } (\urcorner a).R}{(\nu a)(P \mid \overline{a}\langle x\rangle.Q) \longrightarrow_{dpe1} (\nu a)P}$

(4) $\dfrac{\text{Les seules occurrences de } a \text{ dans } P \text{ sont de la forme } (\urcorner a).R}{(\nu a)(P \mid a(x).Q) \longrightarrow_{dpe1} (\nu a)P}$

(5) $\dfrac{\text{Les seules occurrences de } a \text{ dans } P \text{ sont de la forme } (\urcorner a).R}{(\nu a)(P \mid !\overline{a}\langle x\rangle.Q) \longrightarrow_{dpe1} (\nu a)P}$

(6) $\dfrac{\text{Les seules occurrences de } a \text{ dans } P \text{ sont de la forme } (\urcorner a).R}{(\nu a)(P \mid !a(x).Q) \longrightarrow_{dpe1} (\nu a)P}$

FIG. 6.14 – Un ramasse-miettes.

Définition 21 (Ramasse-miettes)
Un ramasse-miettes $\longrightarrow_{dpe}$ est une relation entre processus telle que si $P \longrightarrow_{dpe} Q$ alors $(P, Q) \in \mathcal{D}pe$.

De la même manière, nous pourrions spécifier un algorithme de ramasse-miettes dans le langage. Nous avons préféré laisser la relation d'élimination des processus morts sous la forme d'un paramètre du langage. Ainsi, si nous notons $\longrightarrow_{dpe}$ cette relation, qui doit être un ramasse-miettes, nous pouvons introduire la règle suivante :

$$\textsc{Red-Dpe}\ \frac{P \longrightarrow_{dpe} Q}{P \longrightarrow Q}\ .$$

Le calcul Cπ complet s'obtient donc en gardant l'ensemble des définitions de Cπ, avec pour seule addition RED-DPE.

6.3.6 Exemples

Définition d'un ramasse-miettes

Considérons $\longrightarrow_{dpe1}$ définie comme la plus petite relation vérifiant les règles de la figure 6.14. Il s'agit ramasse-miettes simple et facile à implanter et qui suffit à gérer quelques situations courantes.

Lemme 25 ()
La relation $\longrightarrow_{dpe1}$ est un ramasse-miettes.

Il suffit de prouver que pour tout couple A, B de processus, si $A \longrightarrow_{dpe1} B$ alors $(A, B) \in \mathcal{D}pe$. Pour ce faire, raisonnons par cas sur une preuve de $A \longrightarrow_{dpe1} B$.

Règle (1) Si la règle (1) a été utilisée, A s'écrit $\circledcirc(x).P$ et B s'écrit $\mathbf{0}$. Par conséquent, A est gardé par $\circledcirc$. La règle DPE-NULL s'applique donc et $(A, \mathbf{0}) \in \mathcal{D}pe$, ce qui prouve le cas.

Règle (2) Cf. Règle (1).

Règle (3) Soit $\mathcal{R}$ la plus petite relation close par $\equiv$ telle que, dans Cπ de base, pour tout S, si $(\nu a)(P \mid \overline{a}\langle x\rangle.Q) \mid S \longrightarrow^* \equiv (\nu a)(T \mid \overline{a}\langle x\rangle.Q) \mid U$, $((\nu a)(T \mid \overline{a}\langle x\rangle.Q) \mid U)\mathcal{R}((\nu a)T) \mid U$.

Soient alors deux termes A et B tels que $A \mathcal{R} B$. On aura

$$\begin{cases} A & \equiv (\nu a)(T \mid \overline{a}\langle x\rangle.Q) \mid U \\ B & \equiv (\nu a)T \mid U\ . \end{cases}$$

Par conséquent, A et B auront les mêmes barbes.

Montrons maintenant par récurrence que, si $(\nu a)(P \mid \overline{a}\langle x\rangle.Q) \mid S \longrightarrow^* A$, il existe T et U tels que $A \equiv (\nu a)(T \mid \overline{a}\langle x\rangle.Q) \mid U$ et tels que les seules occurrences de a dans T sont de la forme $(\urcorner a).R$.

Initialisation Si $A \equiv (\nu a)(P \mid \overline{a}\langle x\rangle.Q) \mid S$, par hypothèse, les seules occurrences de a dans P sont de la forme $(\urcorner a).R$.

Héritage Soit $A \equiv (\nu a)(T \mid \overline{a}\langle x\rangle.Q) \mid U$ tel que les seules occurrences de a dans T sont de la forme $(\urcorner a).R$. Si $A \longrightarrow A'$, alors, comme il ne peut pas y avoir de communication sur a, on aura $A' \equiv (\nu a)(T' \mid \overline{a}\langle x\rangle.Q) \mid U'$. De plus, comme ni T ni U ne peuvent communiquer le nom a, aucune variable ne peut avoir été liée à a dans T'. Par conséquent, les seules occurrences de a dans T' sont aussi de la forme $(\urcorner a).R$. Ce qui prouve la récurrence.

Fin de cas Si nous avons $A \mathcal{R} B$, alors $A \equiv (\nu a)(T \mid \overline{a}\langle x\rangle.Q) \mid U$ avec T et U tels que les seules occurrences de a dans T soient de la forme $(\urcorner a).R$. On aura alors aussi $B \equiv (\nu a)T \mid U$.

Si $A \longrightarrow A'$, notons alors $A' \equiv (\nu a)(T' \mid \overline{a}\langle x\rangle.Q) \mid U'$ où les seules occurrences de a dans T' sont de la forme $(\urcorner a).R$. En appliquant les mêmes règles à B, on trouve aisément que $B \longrightarrow B' = (\nu a)T' \mid U'$. Ce qui achève de prouver que $\mathcal{R}$ est une simulation à barbes.

Or, pour tout S, on aura $(\nu a)(P \mid \overline{a}\langle x\rangle.Q) \mid \mathcal{R}(\nu a)P \mid S$. Par conséquent, on a

$$(\nu a)(P \mid \overline{a}\langle x\rangle.Q) \ll (\nu a)P\ .$$

Par application de DPE-NAME, on conclut que $(\nu a)(P \mid \overline{a}\langle x\rangle.Q), (\nu a)P \in \mathcal{D}pe$. Ce qui prouve le cas.

Règle (4) La preuve est identique à celle du cas (3).

Règle (5) La preuve est similaire à celle du cas (3).

Règle (6) Idem.

Une fois que tous les cas sont traités, nous avons prouvé que $\longrightarrow_{dpe1}$ est un ramasse-miettes. □

Pour la suite des exemples, nous instancierons le Cπ complet avec $\longrightarrow_{dpe1}$ comme relation d'élimination des processus morts.

Retour sur les exemples d'élimination

Exemple simple Considérons de nouveau le processus $A \triangleq (\nu a)!a(x).B$. La règle (5) s'applique à A et implique $A \longrightarrow_{dpe1} (\nu a)\mathbf{0}$. Par conséquent, $A \longrightarrow^* \mathbf{0}$ – la relation $\longrightarrow_{dpe1}$ est donc suffisamment puissante pour gérer notre exemple simple.

Deadlock Si nous nous intéressons de nouveau à

$$A \triangleq (\nu a)(\nu b)(a(x).\overline{b}\langle x\rangle.P \mid b(x).\overline{a}\langle x\rangle.Q \mid (\neg a).R) \ ,$$

la situation est plus problématique. En effet, aucune règle de $\longrightarrow_{dpe1}$ ne permet de supprimer $a(x).\overline{b}\langle x\rangle.P$ ou $b(x).\overline{a}\langle x\rangle.Q$. En pratique, $\longrightarrow_{dpe1}$ n'est pas assez puissant pour nettoyer le deadlock.

Or, nous avons vu que $\mathcal{D}pe$ est suffisamment puissant pour gérer le cas du deadlock. Il existe donc une approximation de $\mathcal{D}pe$ plus puissante que $\longrightarrow_{dpe1}$ qui permette de nettoyer A. Nous n'avons pas cherché à produire une telle relation.

Gestionnaire de ressources

Retournons maintenant à notre gestionnaire de ressources défini dans la section 6.3.2. Dans le langage de base, nous avions le processus $\texttt{Voleur} \triangleq (\nu r)\overline{alloc}\langle r\rangle.r(c).c(x)$ qui, placé en parallèle avec $\texttt{BRM}_n$, empêchait la désallocation d'une ressource.

Considérons maintenant le comportement de $\texttt{BRM}_n \mid \texttt{Voleur}$ dans le langage complet. Comme précédemment, nous avons

$$\begin{array}{rcl} \texttt{BRM}_n \mid \texttt{Voleur} & = & (\nu l)(\texttt{Heap}_n\ l \mid \texttt{Loop}\ l) \mid (\nu r)\overline{alloc}\langle r\rangle.r(c).c(x) \\ & \equiv & (\nu r, l)(\texttt{Heap}_n\ l \mid \texttt{Loop}\ l) \mid \overline{alloc}\langle r\rangle.r(c).c(x) \\ & \longrightarrow^* & (\nu r, l, c)(\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\neg c).\blacksquare_l \mid c(x)) \end{array}$$

Or, par construction, la seule occurrence de c dans $\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\neg c).\blacksquare_l$ est $(\neg c).\blacksquare_l$. Donc, d'après la règle (4), nous avons

$$(\nu c)(\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\neg c).\blacksquare_l \mid c(x)) \rightarrow_{dpe1} (\nu c)(\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\neg c).\blacksquare_l) \ .$$

Par conséquent,

$$\begin{array}{rl} & (\nu r, l, c)(\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\neg c).\blacksquare_l \mid c(x)) \\ \longrightarrow & (\nu r, l, c)(\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\neg c).\blacksquare_l) \\ \equiv & (\nu l)(\texttt{Heap}_{n-1}\ l \mid \texttt{Loop}\ l \mid (\nu c)(\neg c).\blacksquare_l) \\ \longrightarrow & (\nu l)(\texttt{Loop}\ l \mid \texttt{Heap}_{n-1}\ l \mid \blacksquare_l) \\ = & \texttt{BRM}_n \end{array}$$

En d'autres termes, $\longrightarrow_{dpe1}$ est suffisamment puissant pour rendre au gestionnaire de mémoire la ressource détenue par `Voleur`. Le système résiste à cette attaque.

6.3.7 Des types pour contrôler les ressources

Maintenant que nous sommes capables d'employer le cycle d'allocation/finalisation pour concevoir des systèmes conscients de l'utilisation des ressources, nous pouvons définir formellement une notion de ressource et des méthodes pour prouver qu'un système est ressource-contrôlé. Notre conception des ressources pour Cπ reprend celle que nous avions présentée pour le π-calcul (cf. définition 15, page 115).

Définition 22 (Politique d'allocation des ressources – Cπ)
Une politique de contrôle des ressources est composée de
- *une fonction partielle qui, à chaque nom de canal pour laquelle elle est définie, associe l'encombrement du canal ;*
- *la taille de la réserve totale.*

Si Γ est une politique de contrôle de ressources, on notera $\Gamma, a : e$ la fonction qui à a associe e et à tout autre nom b associe $\Gamma(b)$.

Définition 23 (Utilisation des ressources – Cπ)
L'utilisation des ressources par un processus P, dans le cadre d'une politique de contrôle des ressources Γ, notée $res_\Gamma(P)$, est définie par
- $res_\Gamma(\mathbf{0}) = 0$
- $res_\Gamma(P|Q) = res_\Gamma(P) + res_\Gamma(Q)$
- $res_\Gamma(P + Q) = 0$
- $res_\Gamma(!P) = 0$ *si* $res_\Gamma(P) = 0$
- $res_\Gamma(\alpha.P) = res_\Gamma((\neg a).P) = 0$
- $res_\Gamma((\nu a : C)P) = res_{\Gamma, a:C}(P) + e$ *si l'annotation C associe à a un encombrement e*

Si $res_\Gamma(P)$ est défini et si Γ spécifie une réserve totale de taille s, on dira que, dans l'état courant, P respecte la politique Γ si $res_\Gamma(P) \leq s$.

Notons que dans le cas $(\nu a : C)P$ il n'est, en fait, pas nécessaire d'étendre Γ à l'aide de $a : C$ car l'annotation C n'intervient nulle part ailleurs. Nous avons cependant préféré utiliser $\Gamma, a : C$ pour garder une certaine homogénéité entre les définitions pour chaque calcul.

Notons aussi que nous n'acceptons pas $!(\nu a : C)\mathbf{0}$ si l'encombrement de c est non-nul. Nous aurions pu spécifier que l'utilisation des ressources par ce processus est infinie. Nous avons préféré considérer ce cas comme malformé, d'autant plus que l'utilisation de la valeur ∞ complique les tentatives de généralisation (cf. chapitre 9).

Définition 24 (Ressource-contrôlé – Cπ)
Un processus P est dit ressource contrôlé dans le cadre d'une politique de contrôle des ressources Γ si, pour tout Q tel que $P \longrightarrow^* Q$, Q respecte Γ.

$$
\begin{array}{lcll}
C & ::= & Name(T, e) & e \in \mathbf{N} \cup \{\infty\} \\
T & ::= & Ssh & \\
 & | & Chan(C, z) & z \in \mathbf{Z} \\
U & ::= & Proc(t)[\lambda] & t \in \mathbf{N} \cup \{\infty\}, \lambda \text{ liste de noms} \\
A & ::= & Site(s) & s \in \mathbf{N} \cup \{\infty\}
\end{array}
$$

FIG. 6.15 – Grammaire de types pour le contrôle des ressources – Cπ

Types

La grammaire des types est présentée sur la figure 6.15.

Le type d'un nom a est $Name(T, e)$ lorsque e est son encombrement et T sa fonction. T peut être Ssh si a n'est jamais utilisé en tant que canal ou $Chan(C, z)$ si a sert à transmettre des noms de type C et que le coût de la communication peut être équilibré entre émission et réception à l'aide de z (nous reviendrons sur le rôle de z). Un processus P est de type $Proc(t)[\lambda]$ si l'effet de P est t, c'est-à-dire si P peut s'exécuter en utilisant au plus t ressources, et cette information de t prend en compte la désallocation de canaux contenus dans la liste λ et aucun autre. Enfin, nous introduisons le type d'un site sous la forme $Site(s)$, où s est la taille de la réserve du site – bien que, dans Cπ, il n'existe qu'un seul site, que nous appellerons *toplevel*, nous étendrons plus tard cette notation pour le cas du π-calcul distribué.

Un environnement de typage Γ est une fonction partielle qui, à chaque nom de canal pour lequel elle est définie associe un type de canal, et à chaque nom de site pour lequel elle est définie, associe un type de site. Nous noterons $\Gamma, c : C$ la fonction qui au nom de canal c associe le type C et pour tout autre nom se comporte comme Γ. Nous noterons de même $\Gamma, l : A$ la fonction qui au nom de site l associe le type A et pour tout autre nom se comporte comme Γ.

De même que dans les calculs précédents, nous considérons qu'un environnement de typage Γ doublé d'une information sur la taille de la réserve est aussi une politique de contrôle de ressources. Par suite, nous étendons la définition de res_Γ et de termes ressource-contrôlés aux cas où Γ est un environnement contenant le type du site *toplevel*.

Règles de typage

Les règles de typage sont présentées sur la figure 6.16.

Règles usuelles Comme le processus terminé ne consomme pas de ressources et ne désalloue aucun nom, il peut s'accommoder de n'importe quel type, comme spécifié par T-NIL. Les règles T-NAME et T-NULL permettent de typer les noms – en particulier, $\circledcirc$, dont les communications ne peuvent aboutir, peut être typé avec n'importe quel type. La règle T-SUM donne le typage naturel d'un choix : les deux alternatives doivent avoir exactement le même type, qui sera aussi le type du processus somme.

$$\text{T-Nil}\ \Gamma \vdash \mathbf{0} : Proc(t)[\lambda] \qquad \text{T-Name}\ \frac{\Gamma(x) = C}{\Gamma \vdash x : C} \qquad \text{T-Null}\ \Gamma \vdash \circledcirc : C$$

$$\text{T-Res}\ \frac{\Gamma, x : (K, e) \vdash P : Proc(t_P)[\lambda, x]}{\Gamma \vdash (\nu x : K, e)P : Proc(t_P + e)[\lambda]}\ x \notin \lambda$$

$$\text{T-Sum}\ \frac{\Gamma \vdash P : U \qquad \Gamma \vdash Q : U}{\Gamma \vdash P + Q : U}$$

$$\text{T-Par}\ \frac{\Gamma \vdash P : Proc(t_P)[\lambda_P] \qquad \Gamma \vdash Q : Proc(t_Q)[\lambda_Q]}{\Gamma \vdash P|Q : Proc(u)[\lambda_P \cup \lambda_Q]}\ \lambda_P \cap \lambda_Q = \emptyset, u \geq t_P + t_Q$$

$$\text{T-Finalize1}\ \frac{\Gamma \vdash P : Proc(t_P)[\lambda_P] \qquad \Gamma \vdash x : Name(\_, e)}{\Gamma \vdash (\urcorner x).P : Proc(u)[\lambda_P, x]}\ u + e \geq t_P, x \notin \lambda_P$$

$$\text{T-Finalize2}\ \frac{\Gamma \vdash P : Proc(t_P)[\lambda_P] \qquad \Gamma \vdash x : Name(\_, \_)}{\Gamma \vdash (\urcorner x).P : Proc(t_P)[\lambda_P]}\ x \notin \lambda_P$$

$$\text{T-Repl}\ \frac{\Gamma \vdash P : Proc(0)[\emptyset]}{\Gamma \vdash !P : Proc(t)[\lambda]}$$

$$\text{T-Read}\ \frac{\begin{array}{c}\Gamma \vdash c : Name(Chan(C, z), \_) \\ \Gamma, x : C \vdash P : Proc(t_P)[\lambda_P]\end{array}}{\Gamma \vdash c(x).P : Proc(t_P + z)[\lambda_P]}\ x \notin \lambda, t_P + z \geq 0$$

$$\text{T-Write}\ \frac{\begin{array}{c}\Gamma \vdash c : Name(Chan(C, z), \_) \\ \Gamma \vdash Q : Proc(t_Q)[\lambda_Q] \qquad \Gamma \vdash y : C\end{array}}{\Gamma \vdash \overline{c}\langle y \rangle.Q : Proc(t_Q - z)[\lambda_Q]}\ t_Q - z \geq 0$$

FIG. 6.16 – Règles de typage pour le contrôle des ressources – Cπ.

Distribution des autorisations Comme dans les autres systèmes de types, T-PAR somme les contributions des deux processus P et Q composés en parallèle. En plus de cela, et de manière plus inhabituelle, cette règle vérifie qu'aucun nom n'apparaît à la fois dans λ_P et dans λ_Q.

Pour comprendre cela, considérons un nom a qui nécessite e ressources, un processus A qui nécessite t_A ressources et un processus B qui nécessite t_B ressources. Pour simplifier, supposons $t_A \geq e$ et $t_B \geq e$. Il est raisonnable de considérer que $P \triangleq (\urcorner a).A$ nécessite $t_A - e$ ressources, puisque, lors de l'exécution de A, les e ressources acquises par a ont été relâchées.. Il est tout aussi raisonnable de considérer que $Q \triangleq (\urcorner a).B$ nécessite $t_B - e$ ressources. En revanche, il est faux de considérer que $(\urcorner a).A \mid (\urcorner a).B$ nécessite $t_A + t_B - 2 \cdot e$ ressources, puisque les ressources nécessaires pour a ne sont relâchées qu'une seule fois.

Pour typer $(\neg a).A \mid (\neg a).B$ ou un processus plus complexe qui fasse intervenir deux fois ou plus $(\neg a)$, il est donc nécessaire de choisir – statiquement – quelle composante du processus disposera des e ressources. Ainsi, on pourra avoir $a \in \lambda_P$ pour signifier que P est autorisé à réutiliser les ressources utilisées par a, $a \in \lambda_Q$ pour le cas dual, ou encore $a \notin \lambda_P \cup \lambda_Q$ si aucun des deux processus ne peut réutiliser ces ressources.

On peut remarquer que cette répartition des autorisations est proche de certaines opérations présentes dans le système de types linéaires que nous avons présenté précédemment. Les autorisations de réutilisation des ressources sont délivrées ou retirées par les règles T-Res, T-Rcv et T-Repl et consommées par les règles T-Finalize1, T-Nil et T-Repl.

Allocations et désallocations Ainsi, T-Res spécifie que la création d'un nouveau nom x est une allocation – si le nom a un encombrement de e, il faut justement réserver e ressources pour pouvoir procéder à l'allocation. En plus de cela, cette règle autorise le processus à réutiliser ces ressources lorsqu'elles seront désallouées. À la fin de la vie de x, T-Finalize1 permet au processus P de $(\neg x).P$ de réutiliser ces e ressources, tandis que T-Finalize2 n'est pas autorisé à réutiliser les e ressources.

Réplications et communications Selon T-Repl, un processus ne peut être répliqué s'il consomme des ressources ou désalloue des noms qu'il n'a pas lui-même alloués – si un processus peut être répliqué, sa réplication peut alors s'accommoder de n'importe quel type. Or, de nombreux processus répliqués sont conduits à allouer des ressources de manière raisonnable. Ainsi, en pratique, si $a(x).(\nu c : C)P$ nécessite au plus t ressources pour s'exécuter, $!a(x).(\nu c : C)P \mid \overline{a}\langle v\rangle.\mathbf{0}$ ne nécessitera que t ressources, $!a(x).(\nu c : C)P \mid \overline{a}\langle v_1\rangle.\mathbf{0} \mid \overline{a}\langle v_2\rangle.\mathbf{0}$ ne nécessitera que $2 \cdot t$ ressources et plus généralement, s'il y a n émissions sur a, l'ensemble ne coûtera que $n \cdot t_Q$.

C'est ici qu'intervient l'équilibrage du coût des communications : au lieu de considérer que $a(x).(\nu c : C)P$ nécessite t ressources, on considérera que ce processus ne nécessite aucune ressource mais que chaque émission sur a coûte t ressources supplémentaires. Pour procéder à cet équilibrage, on fixe le paramètre z à $-t$. C'est ce que spécifient les règles T-Read et T-Write. Ce mécanisme est à comparer aux macros présentées aux sections 4.2.2 et 4.2.5 dans le cadre des Kells, qui avaient des effets similaires sur les garanties d'utilisation des ressources. Nous verrons dans la section 9 une version étendue de ces règles et une interprétation du paramètre z en termes d'échanges préliminaires de ressources.

Propriétés

Présentons brièvement quelques propriétés intéressantes sur le système de types de Cπ.

Lemme 26 (Sous-typage)
Soient Γ un environnement et P un processus typable dans Γ. Notons $\Gamma \vdash P : Proc(t)[\lambda]$. Alors, pour tout $u \geq t$, on aura $\Gamma \vdash P : Proc(u)[\lambda]$.

Ce lemme est trivial, une fois de plus, et s'avère nécessaire pour les règles T-Sum, T-Finalize1, T-Read et T-Write, qui cachent toutes des soustractions (ou des additions dans $\mathbf{Z}$).

Corollaire 5 (Type minimal) *Soient* Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P : Proc(t)[\lambda]$. *Alors, il existe* u *minimal tel que* $\Gamma \vdash P : Proc(u)[\lambda]$.

Notons que u n'est minimal que pour une liste de noms λ donnée. Une fois de plus, la preuve est triviale.

Lemme 27 (Renforcement)
Si $\Gamma, c : C \vdash P : U$ *et* $c \notin fv(P)$, *alors* $\Gamma \vdash P : U$.

Lemme 28 (Affaiblissement)
Si $\Gamma \vdash P : U$ *et* $c \notin fv(P)$ *alors* $\Gamma, n : A \vdash U$.

Ces lemmes sont habituels et leurs preuves sont standard.

Lemme 29 (Les bons types respectent la politique)
Dans une politique de contrôle de ressources Γ, *si* $\Gamma \vdash P : Proc(t)[\_]$ *et si* $\Gamma(toplevel) = t$, *alors* P *respecte* Γ.

La preuve, simple, est présentée dans l'annexe D.1.

Théorème 12 (Subject Reduction) *Si* $\Gamma \vdash P : U$ *et si* $P \longrightarrow Q$, *alors* $\Gamma \vdash Q : U$.

La preuve est présentée dans l'annexe D.2.

Théorème 13 (Contrôle des ressources) *Soient* Γ *un environnement et* P *un processus. Si* $\Gamma \vdash P : Proc(t)[\_]$ *et si* $\Gamma(toplevel) = t$, *alors* P *est ressource-contrôlé dans* Γ.

Il s'agit d'un corollaire direct de ce qui précède.

Notons que, à l'inverse des systèmes de types que nous avons précédemment présentés, P et Q peuvent être ressource-contrôlés sans que $P|Q$ soit ressource-contrôlé. Cela est dû au fait que, dans les calculs à mobilité de sites, nous n'introduisions pas de localité *toplevel*, alors que nous sommes contraints de le faire dans Cπ.

6.3.8 Exemples

Communication diadique

Même si les macros que nous avons présentées dans la section 6.3.2 pour permettre la communication diadique ne font que peu intervenir de gestion des ressources, il peut être intéressant de considérer leur type.

Nous supposerons

$$\begin{cases} T_r & = Name(Chan(C, z_r), e_r) \\ \Gamma & \vdash u : C \\ \Gamma & \vdash v : C \\ \Gamma & \vdash a : Name(Chan(T_r, z_a), \_) \\ \Gamma & \vdash P : Proc(t_P)[\lambda_P] \\ \Gamma, x : C, y : C & \vdash Q : Proc(t_Q)[\lambda_Q] \ . \end{cases}$$

Notamment, en l'absence de types sommes, nous sommes obligés d'imposer que u et v soient du même type C. De plus, nous supposerons

$$\begin{cases} t_P \geq e_r \\ r \notin \lambda_P \\ r \notin \lambda_Q \\ t_P - e_r - 2 \cdot z_r \geq 0 \\ t_P - e_r - 2 \cdot z_r - z_a \geq 0 \\ t_P - 2 \cdot z_r - z_a \geq 0 \\ t_Q + 2 \cdot z_r \leq 0 \\ t_Q + 2 \cdot z_r + z_a \leq 0 \ . \end{cases}$$

Voici un aperçu d'une preuve :

Typage de P		
$\Gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
$\Rightarrow \Delta \vdash P :$	$Proc(t_P)[\lambda_P]$	Par *Affaiblissement*
Avec	$\Delta \triangleq \Gamma, r : T_r$	
Typage de $(\urcorner r).P$		
$\Delta \vdash P :$	$Proc(t_P)[\lambda_P]$	Cf. plus haut
$\Delta \vdash r :$	$Name(Chan(C, z_r), e_r)$	Par T-NAME
$\Rightarrow \Delta \vdash (\urcorner r).P :$	$Proc(t_P - e_r)[\lambda_P, r]$	Par T-FINALIZE1
Typage de $\overline{r}\langle v\rangle.(\urcorner r).P$		
$\Delta \vdash (\urcorner r).P :$	$Proc(t_P - e_r)[\lambda_P, r]$	Cf. plus haut
$\Delta \vdash r :$	$Name(Chan(C, z_r), e_r)$	Par T-NAME
$\Delta \vdash v :$	C	Par hypothèse
$\Delta \vdash \overline{r}\langle v\rangle.(\urcorner r).P :$	$Proc(t_P - e_r - z_r)[\lambda_P, r]$	Par T-WRITE
Typage de $\overline{r}\langle u\rangle.\overline{r}\langle v\rangle.(\urcorner r).P$		
$\vdots$	$\vdots$	
$\Delta \vdash \overline{r}\langle u\rangle.\overline{r}\langle v\rangle.\cdots :$	$Proc(t_P - e_r - 2 \cdot z_r)[\lambda_P, r]$	Par T-WRITE
Typage de $\overline{a}\langle r\rangle.\overline{r}\langle u\rangle.\overline{r}\langle v\rangle.(\urcorner r).P$		
$\Delta \vdash \overline{r}\langle u\rangle.\overline{r}\langle v\rangle.\cdots :$	$Proc(t_P - e_r - 2 \cdot z_r)[\lambda_P, r]$	Cf. plus haut
$\Delta \vdash a :$	$Name(Chan(T_r, z_a), \_)$	Par hypothèse
$\Rightarrow \Delta \vdash \overline{a}\langle r\rangle.\overline{r}\langle u\rangle.\cdots :$	$Proc(v_1)[\lambda_P, r]$	Par T-WRITE
Avec	$v_1 \triangleq t_P - e_r - 2 \cdot z_r - z_a$	
Typage de `Send` (u, v) `on` $a.P$		
$\Delta \vdash \overline{a}\langle r\rangle.\overline{r}\langle u\rangle.\cdots :$	$Proc(v - 1)[\lambda_P, r]$	Cf. plus haut
$\Gamma \vdash$ `Send` (u, v) `on` $a.P :$	$Proc(t_P - 2 \cdot z_r - z_a)[\lambda_P]$	Par T-RES

○

De la même manière, on prouve

$$\Gamma \vdash \texttt{Receive}\ (x, y)\ \texttt{on}\ a.Q : Proc(t_Q + 2 \cdot z_r + z_a)[\lambda_Q] \ .$$

On peut donc remarquer que, du point de vue du typage, si t_P est assez élevé pour masquer e_r, ces macros sont équivalentes à une seule émission et une seule réception sur un canal pour lequel $z = 2 \cdot z_r + z_a$.

Gestionnaire automatique de ressources

Revenons au gestionnaire de ressources automatique de la section 6.3.2. Considérons la politique de contrôle de types suivante :

$$\begin{cases} N_c & : Name(\_, 1) \\ N_l & : Name(Chan(Ssh, z_l), 1) \\ N_r & : Name(Chan(N_c, 0), \_) \\ \Gamma(alloc) & = Name(Chan(N_r, z_{alloc}), \_) \\ \Gamma(toplevel) & = Site(s_{toplevel}) \end{cases}$$

Cette politique laisse volontairement en suspens les valeurs de z_l (le paramètre d'équilibrage des coûts de communication pour une ressource), de z_{alloc} (le même paramètre pour le canal de requêtes d'allocation) et $s_{toplevel}$ (l'effet maximal du processus obtenu).

Allocations progressives Une première manière de compléter la politique consiste à spécifier que $z_l = 0$ (communiquer sur l ne coûte rien), $z_{alloc} = 1$ (demander une allocation coûte une ressource au client) et $s_{toplevel} = 1$ (le système n'occupe qu'une ressource). Sans entrer dans le détail de la dérivation, nous pouvons prouver que le gestionnaire de mémoire respecte cette politique de contrôle des ressources. Schématiquement, le raisonnement est le suivant :

1. $\blacksquare_l$ ne nécessite aucune ressource
2. $(\neg c).\blacksquare_l$ ne nécessite aucune ressource
3. $\overline{r}\langle c\rangle$ ne nécessite aucune ressource
4. $(\nu c).\cdots$ nécessite 1 ressource
5. $l(\_).\cdots$ nécessite 1 ressource
6. le coût de la communication sur *alloc* est reporté sur l'émetteur, si bien que $alloc(r).\cdots$ ne nécessite donc aucune ressource
7. $\texttt{Heap}_n$ ne nécessite aucune ressource
8. par suite, $\texttt{BRM}_n$ ne nécessite qu'une ressource, allouée à l.

Bilan En d'autres termes, le gestionnaire de mémoire nécessite en tout et pour tout une ressource mais le client doit "payer" une ressource à chaque fois qu'il demande une allocation.

Pré-allocation Une deuxième manière de compléter la politique consiste à spécifier que $z_l = -1$ (écrire sur l coûte 1 ressource), $z_{alloc} = 0$ (communiquer sur *alloc* est gratuit) et $s_{toplevel} = n+1$ (le système occupe $n+1$ ressources). De nouveau, le gestionnaire de mémoire respecte aussi cette politique de contrôle des ressources. En effet,

1. $\blacksquare_l$ nécessite une ressource
2. $(\neg c).\blacksquare_l$ ne nécessite aucune ressource
3. $\overline{r}\langle c\rangle$ ne nécessite aucune ressource
4. $(\nu c).\cdots$ nécessite 1 ressource
5. le coût de la communication sur l est reporté sur l'émetteur, si bien que $l(\_).\cdots$ ne nécessite aucune ressource
6. $alloc(r).\cdots$ ne nécessite donc aucune ressource
7. $\texttt{Heap}_n$ nécessite n ressources
8. $(\nu l)\cdots$ nécessite une ressource de plus pour l
9. par la suite, $\texttt{BRM}_n$ nécessite $n+1$ ressources.

Bilan En d'autres termes, le gestionnaire de mémoire nécessite autant $n+1$ ressources mais les requêtes sont "gratuites" du point de vue du client. □

En jouant sur les stratégies d'équilibrage des coûts de communication, nous avons donc prouvé deux propriétés différentes sur l'allocation des ressources par le gestionnaire.

Gestionnaire de ressources manuel

Bien que le gestionnaire de ressources manuel de la section 6.1.2 ne procède pas à un contrôle des ressources, puisque la désallocation des ressources n'est jamais prise en compte, il peut être intéressant de l'examiner.

Considérons donc la politique de contrôle de types suivante :

$$\begin{aligned} N_c &= Name(\_, 1) \\ N_l &= Name(Chan(Ssh, z_l), 0) \\ N_r &= Name(Chan(N_c, 0), 0) \\ N_d &= Name(Chan(Ssh, 1), z_d) \\ \Gamma(alloc) &= Name(Chan(N_r, -1 - z_d), \_) \\ \Gamma(toplevel) &= s_{toplevel} \end{aligned}$$

Sans entrer dans les détails, on peut alors vérifier que $\Gamma \vdash BRM_n : n + 1$. Comme le paramètre z de *alloc* vaut $-1 - z_d$, chaque requête de ressource coûte alors $z_d + 1$ ressources, dont z_d ressources sont récupérées lorsque le canal d est utilisé pour prévenir le gestionnaire que l'opération est terminée.

En d'autres termes, et comme nous pouvions le prévoir, nous ne pouvons obtenir la garantie de mémoire finie valable pour le gestionnaire de mémoire automatique.

6.3.9 Pour améliorer Cπ

Pour vérifier l'encombrement d'un nom

De manière plus visible encore que dans les calculs à mobilité de sites, le nombre de ressources à allouer – dynamiquement – à une entité est lié à une annotation – statique – sur le nom de cette entité. Ainsi, l'opération $(\nu c : Name(\_, e))$ alloue e ressources pour le nom c.

Mais alors que penser d'un processus reçu par un canal non fiable, tel que `Untrusted` $\triangleq (\nu c : Name(\_, 0))P$? Comment être certain que c n'occupera effectivement aucune ressource, puisqu'il est impossible, même avec le code source de `Untrusted`, de vérifier quoi que ce soit sur l'encombrement de c ?

Nous ne fournissons pas de réponse définitive à cette question. Cependant, il est probable que, si le Cπ devait être implanté, l'opération (ν) serait traduite par plusieurs primitives différentes, connues et considérées comme sûres, et dont l'utilisation impose une valeur à l'encombrement des canaux créés. Ainsi, un canal de communication locale occupera toujours e_{lcc} ressources et sera toujours créé avec l'opération `newlcc`, tandis qu'un canal de sortie vers un fichier occupera toujours $e_{fileostream}$ ressources et sera toujours créé avec l'opération `newostream`. Vu d'une autre manière, on peut considérer cette transformation comme une négociation préliminaire sur l'encombrement des diverses allocations.

Pour représenter cela dans Cπ, nous pouvons imaginer utiliser un π-calcul avec groupes [24] – c'est-à-dire, grossièrement, une variante du π-calcul dans laquelle il est possible d'ajouter des types (vu comme des ensembles de noms) et dans laquelle chaque création de nom est annotée par l'un des types ainsi créés. Cette notion de groupes, même si sa syntaxe lui donne une apparence plus dynamique que les types classiques, est en fait très similaire des types tels que nous les connaissons dans les langages de programmation : dans la majeure partie des langages évolués, il est possible de déclarer de nouveaux types, à l'aide de constructions comme `struct`, `record` ou `class` puis de créer de nouvelles instances de ces types.

Nous reviendrons sur les groupes dans la section 7.1.

Répartition des ressources lors de la finalisation

Pour typer les communications, nous employons un paramètre d'équilibrage z qui, partant du principe que la communication nécessite la participation aussi bien de l'émetteur que du récepteur, permet de répartir les coûts entraînés par les communications entre l'émetteur et le récepteur.

À l'inverse, lorsque nous typons une finalisation $(\urcorner c).P$, nous devons décider si les ressources libérées par c peuvent être entièrement réutilisées par P ou si toutes ces ressources vont être utilisées par un autre finaliseur de c, sans possibilité de demi-mesure – c'est l'un des rôles des règles T-FINALIZE1 et T-FINALIZE2. Or, ce comportement est exagéré puisque, si $(\urcorner c).P$ et $(\urcorner c).Q$ sont deux finaliseurs de c, ils seront systématiquement exécutés en parallèle.

Dans une version ultérieure de notre système de types, nous pourrions aisément ajouter la possibilité d'équilibrer les gains dus à la finalisation entre les divers finaliseurs d'un même nom.

D'autres analyses statiques

Le système de types générique pour le π-calcul [46], développé par Atsushi Igarashi et Naoki Kobayashi, permet de vérifier statiquement de nombreuses propriétés d'un terme écrit en π-calcul. Quelques travaux préliminaires nous laissent supposer qu'il serait possible d'utiliser ce système de types, tel quel, pour prouver notamment qu'un processus ne contient pas et ne contiendra jamais de suffixes de la forme $(\urcorner c).P$ avec $c \in fv(P)$. Sur ce fragment de Cπ, la sémantique de la finalisation est alors la même que dans les langages OCaml, Java, C#.

Un autre système de types de Naoki Kobayashi [51] permet d'éviter les deadlocks et les livelocks dans le π-calcul. Dans des termes sans deadlocks ou livelocks, il semble probable que les opérations d'élimination des processus morts deviennent inutiles. Reste à vérifier si ce système de types et celui que nous proposons pour le contrôle des ressources peuvent coexister.

6.3.10 Discussion

Le calcul Cπ est un formalisme conçu à partir du π-calcul par l'ajout d'une primitive de finalisation, qui permet de synchroniser un processus sur la récupération de ressources allouées à une entité, et d'une formalisation de l'élimination des processus morts.

Tout comme les Controlled Ambients ou nos autres formalismes, Cπ seul ne suffit pas à exprimer directement des propriétés de ressources mais permet, grâce à l'absence d'allocations "par effets de bord" et grâce à la possibilité de réagir aux désallocations, d'écrire des systèmes conscients des ressources. La représentation des ressources et des politiques de contrôle de ces ressources, une fois de plus, est entièrement statique et se fait grâce au système de types que nous avons proposé. Enfin, si l'élimination des processus morts permet notamment de débloquer d'autres processus en attente de finalisation, elle n'influe en rien sur la représentation des ressources ou leur contrôle.

Comparaison avec les autres formalismes Même si les points communs entre les calculs à mobilité de sites et Cπ ne sont pas évidents, il existe une certaine similitude entre les mécanismes d'allocation/désallocation de ces formalismes. Ainsi, le processus $a\,[P]$, comme (νa), représente aussi bien l'action d'allouer des ressources à une entité a qu'un marqueur qui rappelle l'allocation de ces ressources. De même, les constructions $a\langle P\rangle.Q$, $a\,[P] \mid \overline{\texttt{out}}\ a.Q$ ou $a\,[\overline{\texttt{open}}\ Q]$ qui définissent aussi bien une entité a qu'un processus Q déclenché lors de la disparition de a, peuvent être rapprochées du couple $(\nu a)/(\neg a).Q$.

La cocapacité $\overline{\texttt{in}}\ a$, quant à elle, peut peut-être s'interpréter dans Cπ comme l'extrusion de (νa) lors d'une communication du nom a. En effet, si nous considérons deux agents $A = (\nu a).\overline{c}\langle a\rangle.P$ et $B = c(x).Q$ composés en un processus $A|B$, nous aurons $A|B \longrightarrow (\nu a)(A' \mid B')$ où $A' = P$ et $B' = Q\{x \leftarrow a\}$. Si nous supposons l'existence de localités implicites pour nos agents, intuitivement, l'entité a, qui n'était que dans la localité A se retrouve (aussi) dans la localité B. Du point de vue de la localité B, il y a eu entrée d'une entité, c'est-à-dire, dans les ambients, $\overline{\texttt{in}}$. Le parallèle est complexe et nous ne le pousserons pas plus loin.

Enfin, aussi surprenant que cela puisse paraître, nous pouvons imaginer identifier les processus d'ouverture $\texttt{open}\ a.P$ et $\overline{\texttt{open}}\ a.Q$ avec la communication $a(x).P$ et $\overline{a}\langle v\rangle.Q$. En effet, $\overline{\texttt{open}}\ a.Q$ est une forme d'émission d'ordre supérieur, puisqu'elle enrichit la localité qui contient $\texttt{open}\ a.P$ – le récepteur – de processus a priori arbitraires. De plus, l'équilibrage des coûts de communication peut être considéré comme une version plus esthétique du contrôle que nous sommes capables de fournir à l'ouverture.

Ces parallèles nous amènent à nous interroger sur le rôle des capacités et cocapacités. Ainsi, nous pouvons avoir plusieurs $(\neg a)$ mais un seul $\overline{\texttt{out}}\ a$/une seule continuation pour $a\langle P\rangle$. Rien n'interdirait de concevoir une variante des CA dans laquelle plusieurs $\overline{\texttt{out}}\ a$ peuvent être consommés par un seul mouvement ou une variante des Kells dans laquelle plusieurs continuations pourraient être spécifiées pour $a\langle P\rangle$.

De même, nous avions noté qu'il pourrait être intéressant de différencier la présence d'un site $a\,[P]$ d'une opération qui entraîne la création d'un site – par exemple $\tau.a\,[P]$ ou, dans BoCa, $1 \rhd a\,[P]$. De la même manière, il pourrait être intéressant de découpler le masquage d'un nom de l'allocation de ressources à ce nom, par exemple à l'aide de deux primitives `name` et `new`.

Comparaison avec les types linéaires Les types linéaires pour le π-calcul, que nous avons présentés dans la section 6.2 sont assez proches de notre travail, aussi bien par les objectifs que par certaines techniques employées. Ainsi,

là où les types linéaires limitent par le typage le nombre d'utilisations d'un canal, nous introduisons un marqueur qui interdit de réutiliser un canal, ce que nous garantissons dynamiquement par une substitution mais que nous préférerions généralement vérifier statiquement. Là où les types linéaires distribuent les autorisations de communiquer entre les composantes parallèles d'un processus, nous distribuons les autorisations d'utiliser et de réutiliser les ressources.

À l'inverse des types linéaires, cependant, nos politiques de contrôle ne se restreignent pas à des propriétés booléennes ("au plus une fois"/"un nombre quelconque de fois") et proposent un mécanisme pour traiter finement certains processus répliqués. Ainsi, dans un processus tel que $!c(x)P$, le système de types linéaires suppose que le processus P sera invoqué une infinité de fois, tandis que nous supposons que P sera ne invoqué que autant de fois qu'il peut y avoir de $\overline{c}\langle\_\rangle$ dans le système, ce qui nous permet de typer des services répliqués sans abandonner tout contrôle à cause de la réplication.

Ici aussi, les points communs sont suffisamment nombreux pour nous inviter à chercher s'il existe une présentation élégante de notre système de types sous une forme proche de ce système de types linéaires, ou l'inverse.

Bilan Nous n'avons pas cherché à modéliser à l'aide de Cπ des protocoles aussi complexes que ceux que nous avons étudiés dans les langages à mobilité de sites. Cependant, comme Cπ contient pour l'essentiel le π-calcul, nous sommes convaincus que ce formalisme est suffisamment puissant pour permettre d'implanter naturellement des protocoles complexes. De plus, d'après notre expérience, la finalisation est simple à employer et à ajouter à des emplacements utiles dans des processus du π-calcul. Il serait nécessaire d'essayer Cπ sur de plus grands projets pour vérifier si le contrôle passe à l'échelle.

Le mécanisme d'élimination des processus morts, qui forme une part importante de notre étude, est essentiellement invisible et transparent, puisqu'il se contente de remplacer des processus inutilement complexes par d'autres processus plus simples et capables de les simuler. Si ce mécanisme peut ne pas sembler important pour des processus "bien écrits" – et qui libèrent toutes les ressources acquises – cette fonctionnalité peut s'avérer nécessaire, comme nous l'avons montré dans le cas du gestionnaire de mémoire, afin de composer du code sûr avec du code non vérifié. De plus, il est assez fréquent d'écrire, par exemple pour modéliser le non-déterminisme, des termes dans lesquels deux processus ou plus sont en course ("race situation") pour communiquer sur un canal. Dans ce genre de situations, le processus perdant est généralement inutile et peut être éliminé par un mécanisme d'élimination des processus morts.

Enfin, notre système de types s'avère assez puissant pour garantir de nombreuses politiques de contrôle de ressources, qui peuvent être exprimées assez simplement à l'aide de ce système. De plus, comme dans le cadre des calculs à mobilité de sites, les règles font essentiellement appel à des inéquations linéaires sur des entiers positifs ou infinis, ce qui nous laisse supposer qu'il est possible de rendre l'analyse et l'inférence totalement automatiques. Notons cependant que l'équilibrage des coûts des communications, qui est un mécanisme puissant, risque d'être complexe à gérer, ainsi que la distribution des autorisations de réutiliser les ressources. La résolution de ce dernier problème pourrait peut-être être simplifiée si, plutôt que de réserver à l'un des finaliseurs les ressources libérées par la désallocation d'un nom, nous répartissions ces ressources entre tous

les finaliseurs de ce nom.

Chapitre 7

Extensions et variantes

7.1 Régions

Jusqu'à présent, dans le cadre de π et Cπ, nous avons considéré une notion de garbage-collection qui désalloue les entités indépendamment les unes des autres. Il existe cependant une pratique connue aussi bien en programmation que dans les calculs séquentiels [81] ou concurrents [15] et qui consiste à rassembler les entités allouées en *régions*. Au lieu de désallouer les entités une à une, c'est la région entière qui va être nettoyée, à partir du moment où les seules références à des entités allouées dans la région G sont elles-mêmes allouées dans G.

Ce mécanisme est utilisé car il permet de gérer aisément les problèmes de cycles de références, il peut être implanté plus facilement qu'un garbage-collector traditionnel, et il peut simplifier les interactions avec des langages de programmation impératifs.

Dans cette section, nous présentons brièvement le *π-calcul avec groupes* [24], un calcul introduit par Silvano Dal Zilio et Andrew Gordon notamment pour prouver la validité du typage dans le calcul des régions [81] puis nous discutons de la possibilité d'adapter Cπ aux concepts de ce calcul.

7.1.1 Le langage

La figure 7.1 présente la grammaire du π-calcul avec groupes. Cette version est asynchrone et sans choix mais pourrait probablement être étendue sans difficultés majeures à un calcul synchrone avec choix. Pour simplifier la syntaxe, nous nous limiterons à des communications monadiques.

Les constructions du langage sont essentiellement identiques à celles du π-calcul, à l'exception de $(\nu G)P$, qui déclare un groupe nommé G connu du processus P. Le processus $(\nu c : T)Q$ déclare alors un nouveau nom c de type T, où T spécifie notamment le groupe auquel appartient c, ce que nous noterons $T \subset G$.

Nous ne présentons pas le détail de la sémantique de ce langage, qui est essentiellement identique à celle du π-calcul. Les noms de groupes ne sont pas des valeurs et ne peuvent donc pas être transmis. Pour le reste, la construction (νG) est gérée essentiellement de la même manière que (νc) dans le π-calcul traditionnel. Ainsi, par exemple, de la même manière que $(\nu c : T)$ est extrudé pour

$$
\begin{array}{rcl}
P, Q & ::= & \mathbf{0} \\
& | & P|Q \\
& | & c(x).P \\
& | & \overline{c}\langle x\rangle \\
& | & !P \\
& | & (\nu G)P \\
& | & (\nu c : T)P
\end{array}
$$

FIG. 7.1 – Grammaire du π-calcul with groups.

permettre de communiquer c à un processus, (νG) est extrudé pour permettre de communiquer c à un processus, lorsque T mentionne G.

Même si le langage ne mentionne pas de règles de nettoyage de (νG) et $(\nu c : T)$, il semblerait naturel d'ajouter les règles $(\nu G)\mathbf{0} \longrightarrow \mathbf{0}$ et $(\nu c : T)\mathbf{0} \longrightarrow \mathbf{0}$ pour couvrir ce problème.

Il est à noter que les informations de groupe peuvent être effacées d'un terme sans modifier l'exécution de ce terme, par une fonction *erase* :

Lemme 30 (Dégroupage)
[15] Soit P un processus du π-calcul avec groupes. Pour tout processus Q tel que $P \longrightarrow Q$, on a $erase(P) \longrightarrow erase(Q)$ et pour tout R tel que $erase(P) \longrightarrow R$, il existe Q tel que $P \longrightarrow Q$ et $erase(Q) \equiv R$.

7.1.2 Cπ avec groupes ?

Comme les régions et les groupes contribuent à simplifier la gestion de la mémoire et comme cette approche semble pouvoir se généraliser à d'autres types de ressources, il nous a semblé intéressant d'imaginer un calcul Cπ avec groupes. Ce travail en est encore à ses débuts mais nous pouvons déjà présenter une esquisse des idées.

Les groupes s'intègrent sans difficulté dans Cπ. Ainsi, nous pouvons considérer, au choix, que (νG) alloue suffisamment de ressources pour permettre toutes les allocations $(\nu n : T)$ lorsque $T \subset G$, ou que (νG) n'alloue aucune ressource alors que chaque $(\nu n : T)$ alloue des ressources. Dans les deux cas, plutôt que de conserver la finalisation individuelle de chaque entité par $(\neg n).P$, il semble plus sensé d'introduire une finalisation de groupe $(\neg G).P$, déclenchée lorsque toutes les entités de G sont inutilisées, à l'aide d'une règle telle que

R-FIN-GROUP
$$(\nu G)(\neg G)P \longrightarrow P \ .$$

Pour éliminer les $(\nu n : T)$, on se contentera d'une règle $(\nu c : T)\mathbf{0} \longrightarrow \mathbf{0}$. Notons que ces transformations tendent en fait à simplifier le langage, puisque la valeur $\circledcirc$ devient inutile. Si un système de types s'avère nécessaire pour garantir que dans $(\neg G)P$, aucune entité n'est allouée depuis G, la propriété en question est probablement plus aisée à vérifier puisque G ne peut être communiqué comme valeur.

Reste encore à régler le problème de la garbage-collection, dont nous supposons qu'il peut être traité comme dans Cπ, sans différences majeures.

7.1.3 Ressources et régions

Si nous considérons que (νG) alloue suffisamment de ressources pour permettre toutes les allocations $(\nu n : T)$ avec $T \subset G$, le type d'un groupe G est alors constitué de l'encombrement total maximal des allocations qui auront lieu dans G. Le rôle du système de types sera alors de garantir que la somme des encombrements des noms n en question est bien limitée par l'encombrement de G. De la même manière que dans Cπ, ces ressources pourront être réutilisées après $(\neg G)$. Notons que le nombre de ressources utilisées par G n'est pas forcément identique au nombre de ressources utilisées par les noms de G. Ainsi, si G représente un fichier d'échange mémoire (swap file), créer G nécessite un peu de mémoire vive, mais les allocations de noms dans G nécessiteront un autre type de ressources – ici de l'espace disque.

Ce découplage entre espace interne et espace externe rappelle fortement notre conception des ressources dans les Controlled Ambients, présentée dans la section 3.3.4. Ainsi, de ce point de vue, G peut être assimilé à une localité virtuelle, dotée d'une capacité s et d'un encombrement e, éventuellement confondus.

Nous pouvons, à l'inverse, considérer que (νG) n'alloue pas de ressources en soi mais que chaque $(\nu n : T)$ consomme des ressources de G dès que $T \subset G$. Cette possibilité doit être approchée différemment car il reste nécessaire de déterminer combien de ressources sont désallouées au moment où l'on rencontre $(\neg G).P$. Or, s'il est relativement aisé de calculer une borne supérieure sur le nombre de ces ressources, nos méthodes ne sont absolument pas appropriées pour calculer une borne inférieure. C'est cependant une telle borne dont nous aurions besoin pour fournir des garanties lors de $(\neg G).P$. Par contre, il est possible d'envisager une utilisation de ressources explicites, qui seront retournées lors de la finalisation. Cette utilisation de ressources explicites, à son tour, n'est pas sans rapports avec les choix de conception derrière BoCa (cf. section 3.2).

Notons enfin que ces deux points de vue sont eux-mêmes assez proches des informations que nous avons pu extraire du typage du gestionnaire automatique de ressources. Ainsi, du point de vue du typage, $\texttt{BRM}_n$ peut occuper $n+1$ ressources dès sa création et n'entraîner aucun coût supplémentaire ou peut occuper une seule ressource à la création et entraîner une allocation à chaque utilisation du canal *alloc*. Cependant, dans aucun des deux cas, le système de types que nous avons présenté pour Cπ ne suffit à prendre en compte la récupération des ressources en question comme nous le souhaiterions en présence de groupes. Cette limitation, à elle seule, justifierait un travail sur un langage Cπ avec groupes.

7.2 Distribution explicite – un aperçu de Dπ

7.2.1 Le langage

Le langage Dπ [41] est un calcul dérivé du π-calcul, qui y ajoute une notion de sites et de migration de processus. Un système est composé d'un ensemble de localités sans hiérarchie spatiale et chaque localité peut contenir un nombre arbitraire de processus. Ces processus reprennent essentiellement le π-calcul traditionnel, auquel s'ajoute une action go, qui permet de faire migrer un processus. De plus, contrairement aux calculs de sites mobiles, si $a\,[P]$ signifie que le site

$$
\begin{array}{rcl@{\qquad}rcl}
 & & & P, Q & ::= & \mathbf{0} \\
 & & & & | & P|Q \\
M, N & ::= & \epsilon & & | & (\nu a : C)P \\
 & | & M|N & & | & a(x).P \\
 & | & (\nu a@b : C)M & & | & \overline{a}\langle x\rangle.P \\
 & | & a\,[P] & & | & !P \\
 & & & & | & \texttt{if } a = b \texttt{ then } P \texttt{ else } Q \\
 & & & & | & \texttt{go}\ a.P
\end{array}
$$

FIG. 7.2 – Syntaxe de Dπ.

a contient le processus P, ce terme n'interdit pas à a de contenir aussi d'autres processus. Ainsi, les termes $a\,[P] \mid a\,[Q]$ et les termes $a\,[P \mid Q]$ sont équivalents. Une des conséquences de ce point de vue distinct est qu'un nom ne peut désigner qu'une seule et unique localité : tous les processus P qui apparaissent comme $a\,[P]$ sont en fait composés en parallèle dans la même localité a.

La syntaxe de Dπ – légèrement modifiée par rapport à la syntaxe canonique [41], pour des raisons d'homogénéité de l'exposé – est présentée sur la figure 7.2. La seule primitive de migration est go. Ainsi, on a $a\,[\texttt{go}\ b.P] \longrightarrow b\,[P]$, ce qui signifie que, si le site a contient entre autres un processus de la forme go $b.P$, le système peut évoluer en un système dans lequel le processus P est dans le site b. Ainsi, la migration concerne un seul thread – ici P, parfois appelé "agent" – et non pas tout une localité (comme avec les primitives de mobilité des calculs à mobilité de site) ou tout son contenu (comme avec open/$\overline{\texttt{open}}$). Comme rien n'impose que le site b existe déjà, cette primitive de migration permet aussi de créer de nouvelles localités.

Comme les processus réagissent essentiellement comme dans le π-calcul, à des différences syntaxiques près, en particulier, les communications primitives sont toutes locales. Le seul moyen de faire passer une information d'un site à un autre consiste à faire migrer un processus, par exemple à l'aide d'une macro de communication asynchrone telle que $\overline{a@b}\langle x\rangle \triangleq \texttt{go}\ b.\overline{a}\langle x\rangle$.

Un nom de canal ou de site créé dans une localité peut, par la suite, être partagé entre plusieurs localités. La construction $(\nu a@b : C)M$ marque que le nom a a été créé dans le site b. Cependant, la signification de cette appartenance est ambiguë. En effet, considérons le terme

$$M \triangleq b\left[(\nu a)\texttt{go}\ c.\overline{d}\langle a\rangle\right] \mid c\left[d(x).(\overline{x}\langle e\rangle \mid x(f).P)\right] .$$

Dans M, un nom a est créé dans le site b puis transmis au site c, qui se sert de ce nom pour communiquer localement, sans autre intervention de b. Si la syntaxe du langage, par $(\nu a@b : C)$, semble confirmer que a appartient à b, cette notion d'appartenance demande à être définie. La question reste ouverte lorsque a est un nom de site : que signifie $(\nu a@b : C)a\,[\ldots] \mid \ldots$?

Plusieurs disciplines de types [40] et variantes de Dπ [70, 40, 82] répondent à cette question, de plusieurs manières différentes, que nous ne détaillerons pas.

7.2.2 Les ressources dans Dπ

Comme dans le π-calcul, les canaux de communication permettent de représenter de nombreux objets intéressants. D'autres objets, comme les sites physiques ou virtuels, les agents ou les applications, seront décrits plutôt en termes de localités.

Comme les canaux aussi bien que les sites peuvent être créés dynamiquement et peuvent disparaître au cours de l'exécution, nous considérerons que ces deux catégories d'objets forment l'ensemble des entités qui occupent des ressources.

Définition 25 (Ressources – Dπ)
L'ensemble des ressources de Dπ est l'ensemble des noms. Chaque site dispose d'une réserve propre. La seule opération d'allocation est la création d'un nouveau nom. La désallocation n'est pas représentée dans le calcul.

Ainsi, de la même manière que, dans π, nous considérons que le processus $(\nu c)P$ nécessite généralement plus de ressources que le processus P pour s'exécuter, dans Dπ, nous considérons que, dans le cas général, le terme $(\nu a@b)b\,[P]$ nécessite plus de ressources *de la réserve de* b que $b\,[P]$ pour s'exécuter.

7.3 Controlled Dπ

Nous présentons ici un aperçu de notre travail en cours sur Controlled Dπ (CDπ), une extension de Dπ qui vise à améliorer la sécurité et à permettre l'écriture de processus ressource-contrôlés. Il existe déjà plusieurs variantes de Dπ, à commencer par SafeDPI [40], qui introduit de nouvelles primitives et modifie la migration, notamment pour améliorer la sécurité. Notre calcul, CDπ, est en fait intermédiaire entre Dπ, qu'il complète, et SafeDPI, bien plus puissant. Notre objectif, plutôt que de concevoir un calcul à part entière, est d'étudier sur un formalisme plus simple des méthodes que nous emploierons plus tard sur SafeDPI ou peut-être d'autres calculs de la même famille.

Comme dans Dπ, chaque système est constitué d'un ensemble de localités qui peuvent contenir des processus proches du π-calcul.

La conception des ressources étend celle que nous avons présenté dans les définitions 23 (page 134) et 25 (page 149). Ainsi, les ressources ne sont pas représentées explicitement mais les opérateurs (ν_{chan}) et (ν_{loc}) délimitent le domaine de définition respectivement d'un canal de communication ou d'une localité, c'est-à-dire, dans les deux cas, d'une entité qui occupe des ressources.

Ainsi, de la même manière que Cπ se comporte comme un langage de programmation muni d'un ramasse-miettes pour des données primitives (i.e. des constructions sans pointeurs vers d'autres données), CDπ se comporte comme un langage de programmation distribuée muni d'un ramasse-miettes pour ces données primitives ainsi que pour les localités elles-mêmes.

Comme dans Cπ, nous ajoutons un opérateur de finalisation ($\neg$), dont nous étendons le fonctionnement pour lui permettre de gérer la désallocation des sites. Ainsi, si m n'est pas libre dans P, on aura

$$(\nu_{loc}\ m@l)l\,[(\neg m).P] \longrightarrow l\,[P]\ .$$

Comme la primitive de migration go a de Dπ est susceptible de provoquer l'allocation de ressources de a sans action visible de la part de a, nous la rempla-

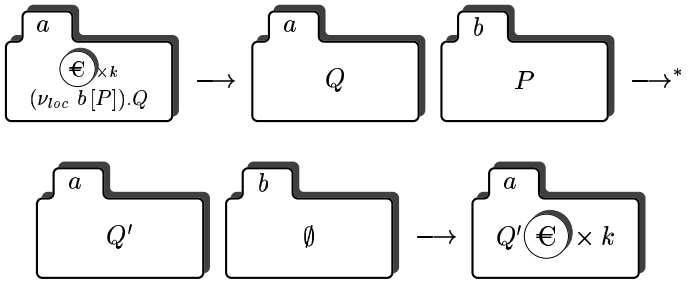


FIG. 7.3 – Cycle de vie d'une localité – CDπ.

$$
\begin{array}{rcl}
M, N & ::= & \epsilon \\
& | & M|N \\
& | & (\nu_{loc}\ a@b : A)M \\
& | & (\nu_{chan}\ a@b : C)M \\
& | & a\,[P]
\end{array}
\qquad
\begin{array}{rcl}
P, Q & ::= & \mathbf{0} \\
& | & P|Q \\
& | & \texttt{newloca}\,[\texttt{P}] : \texttt{A})\texttt{Q} \\
& | & \texttt{newchana} : \texttt{C})\texttt{P} \\
& | & (\nu a).P \\
& | & a(x).P \\
& | & \overline{a}\langle x\rangle.P \\
& | & !P \\
& | & \texttt{if}\ a = b\ \texttt{then}\ P\ \texttt{else}\ Q \\
& | & \texttt{go}\ a@b.P \\
& | & \texttt{come}\ a.P
\end{array}
$$

FIG. 7.4 – Syntaxe de CDπ.

çons par une migration avec synchronisation à l'aide d'un nom de canal. Ainsi, on aura

$$a\,[\texttt{go}\ c@b.P]\ |\ b\,[\texttt{come}\ c.Q] \longrightarrow b\,[P \mid Q]\ .$$

Ce mécanisme de synchronisation est semblable à celui qu'on peut trouver dans SafeDPI ou Controlled Nomadic π [78] et rappelle aussi les capacités/cocapacités des CA ou encore les déplacements le long de canaux dans les Seals. Une conséquence directe de cette modification est que la création d'un site n'est plus un aspect de la migration mais devient une opération à part entière, notée (`newloc`), et similaire à cette opération dans SafeDPI. Nous distinguons alors la création du site par (`newloc`) de l'existence du site, notée par (ν_{loc}). De même, nous distinguons la création d'un canal par (`newchan`) de l'existence du canal, notée (ν_{chan}).

Nous présenterons aussi quelques idées sur la gestion de la garbage-collection et sur l'élimination des processus morts – deux tâches plus complexes que dans Cπ en raison de la difficulté de concevoir un ramasse-miettes distribué [27, 68].

7.3.1 Le langage de base

La syntaxe de CDπ est présentée sur la figure 7.4, avec les hypothèses habituelles. On suppose encore l'existence d'un nom spécial ⊙, qui est aussi bien un canal sur lequel il n'est pas possible de communiquer qu'un site vers lequel il n'est pas possible de migrer.

Nous ne nous étendrons pas sur la signification des constructions habituelles. On notera M, $N\ldots$ les réseaux, composés de sites, qui peuvent avoir alloué des noms de localités et qui peuvent partager ces noms. La construction $a\,[P]$ signifie que le site a *contient* le processus P. Par suite, $a\,[P] \mid a\,[Q]$ signifie que a contient les processus P et Q. Le processus $\texttt{if}\ a = b\ \texttt{then}\ P\ \texttt{else}\ Q$ réagit, conformément à l'intuition, comme P ou Q selon l'égalité de a et b. La migration d'un processus P vers un processus b selon un canal a se fait par une synchronisation entre $\texttt{go}\ a@b.P$ et $b\,[\texttt{come}\ a.Q]$ et agit alors comme $b\,[P \mid Q]$. Enfin, $(\texttt{newloc}\ a\,[P] : A)Q$ crée le nom de site a partagé entre Q et un nouveau site $a\,[P]$. L'unicité du nom a est garantie car a reste lié par $(\nu_{loc}\ a)$ et peut être α-converti.

Notons que les constructions $(\nu_{loc}\ a@b : \_)M$ et $(\nu_{chan}\ a@b : \_)M$ imposent une notion d'appartenance aux noms. Ainsi, un nom de canal qui a été créé dans la localité b appartiendra à cette localité, de la même manière qu'un nom de site créé dans b appartiendra à b. Cela signifie que les ressources nécessaires pour allouer a ont été fournies par b et cela introduit notamment une hiérarchie entre les sites, qui n'est pas sans rappeler celle de BoPi [33].

Même si la définition des variables libres et liées et du renommage sont légèrement plus compliquées que dans les langages précédents, nous ne présenterons pas le détail de ces notions, qui s'étendent assez naturellement. Nous emploierons les raccourcis syntaxiques habituels. Nous écrirons de plus (ν) pour représenter indifféremment (ν_{loc}) et (ν_{chan}).

La règles de base la congruence structurelle sont essentiellement les mêmes que dans Dπ ou SafeDPI et étendent globalement celles du π-calcul. Nous ne les présentons donc pas ici. Contentons-nous de préciser que, pour qu'un site ne puisse pas réapparaître lorsque son nom a été détruit, nous remplaçons $a\,[\mathbf{0}] \equiv \epsilon$ par la règle de réduction Red-Stop. Les règles de réduction sont détaillées sur la figure 7.5.

Communications et migrations La règle Red-Comm détermine le déroulement d'une communication sur le canal c : cette communication ne peut avoir lieu qu'entre deux processus présents dans la même localité m, et uniquement à condition que le canal ne soit pas ⊙. De même, la règle Red-Move spécifie qu'un processus P peut migrer sur le canal c en direction de la localité m si la localité m est prête à accepter une migration sur c – à condition que le canal de ne soit pas ⊙. La migration déclenche alors un processus Q, comme le faisaient les cocapacités dans les CA, par exemple.

Nettoyage Comme dans Cπ, Red-Del permet de factoriser les finaliseurs pour un nom, tandis que Red-Fin spécifie que la finalisation d'un nom ne peut avoir lieu que lorsque les seules occurrences de ce nom sont sous son finaliseur – ce qui est valable aussi bien pour un nom de canal que pour un nom de localité.

$$\textsc{Red-Comm}\ m\,[\overline{c}\langle b\rangle.P]\ \mid m\,[c(x).Q] \longrightarrow m\,[P \mid Q\{x \leftarrow b\}]\ a \neq \circledcirc$$

$$\textsc{Red-Move}\ a\,[\texttt{go}\ c@m.P]\ \mid m\,[\texttt{come}\ c.Q] \longrightarrow m\,[P \mid Q]\ a \neq \circledcirc$$

$$\textsc{Red-Del}\ a\,[(\neg c)P]\ \mid a\,[(\neg c)Q] \longrightarrow a\,[(\neg c)(P \mid Q)]$$

$$\textsc{Red-Fin}\ (\nu c@a : C)a\,[(\neg c)P] \longrightarrow a\,[P\{c \leftarrow \circledcirc\}]$$

$$\textsc{Red-Struct}\ \frac{M \equiv M' \qquad M' \longrightarrow N' \qquad N' \equiv N}{M \longrightarrow N}$$

$$\textsc{Red-Res}\ \frac{M \longrightarrow N}{(\nu n)M \longrightarrow (\nu n)N} \qquad \textsc{Red-Par}\ \frac{M \longrightarrow N}{M|O \longrightarrow N|O}$$

$$\textsc{Red-Stop}\ a\,[\mathbf{0}] \longrightarrow \epsilon \qquad \textsc{Red-Split}\ a\,[P|Q] \longrightarrow a\,[P]\ \mid a\,[Q]$$

$$\textsc{Red-Zero-Res}\ (\nu c : C)\epsilon \longrightarrow \epsilon \qquad \textsc{Red-IfEq}\ \texttt{if}\ a = a\ \texttt{then}\ P\ \texttt{else}\ Q \longrightarrow P$$

$$\textsc{Red-IfNeq}\ \texttt{if}\ a = b\ \texttt{then}\ P\ \texttt{else}\ Q \longrightarrow Q\ a \neq b$$

$$\textsc{Red-Res-Site}\ a\,[(\texttt{newchan}\ c : C)P] \longrightarrow (\nu_{chan}\ c@a)a\,[P]$$

$$\textsc{Red-New-Site}\ a\,[(\texttt{newloc}\ b\,[P] : A)Q \longrightarrow (\nu_{loc}\ b@a)(b\,[P]\ \mid a\,[Q])]$$

$$\textsc{Red-Unwind}\ \frac{a\,[P]\ \mid M \longrightarrow N}{a\,[!P]\ \mid M \longrightarrow a\,[!P]\ \mid N}$$

FIG. 7.5 – Réduction dans CDπ.

En particulier, un nom a de localité est supprimé après que toutes les processus dans cette localité aient disparu grâce à RED-STOP et lorsqu'il n'y a plus aucune occurrence de a ou lorsque la seule occurrence de a est de la forme $b\,[(\neg a).P]$. S'il n'y a plus aucune occurrence de a, c'est RED-ZERO-RES qui détruit a. Dans le cas contraire, c'est RED-FIN qui désalloue les ressources et lance la finalisation. La majorité des règles est similaire à SafeDPI.

Le problème des effets de bord Considérons le réseau

$$\begin{aligned} M \triangleq\ & (\nu_{chan}\ b@m)(\\ & l\,[(\texttt{newchan}\ a)\texttt{go}\ p@m.\overline{b}\langle a\rangle]\ \mid \\ & m\,[\texttt{come}\ p.b(x).(\overline{x}\langle v\rangle\ \mid x(y).P)] \\ &)\ . \end{aligned}$$

Un nouveau nom a est créé dans le site l. Suite à une migration, ce nom est communiqué sur le canal $b@m$ du site m. Ainsi, on a

On considère que b nécessite k ressources.

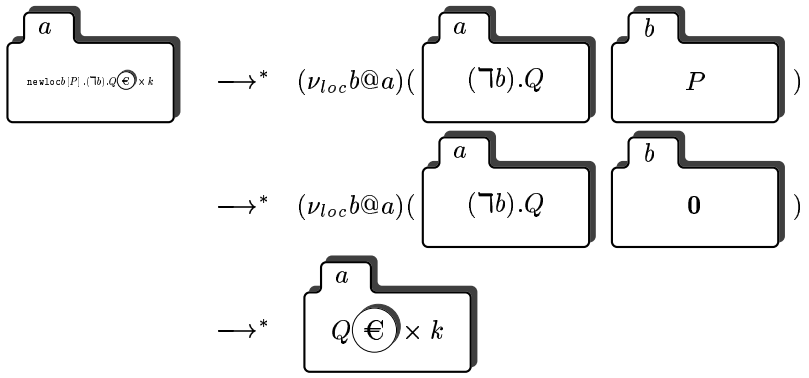


FIG. 7.6 – Utilisation des ressources dans CDπ.

$$M \longrightarrow^* (\nu_{chan}\ b@m)(\nu_{chan}\ a@l)(m\,[\overline{a}\langle v\rangle \mid a(y).P])$$

Dans ce nouveau réseau, des processus de m sont disposés à communiquer localement sur le canal a. Or, ce canal a été créé par effet de bord par l'opération $(\nu_{chan}\ a)$ dans l, qui devait servir à créer un canal a réservé à l !

De la même manière, on peut aisément obtenir un réseau de la forme

$$(\nu_{chan}\ a@l)b\,[(\urcorner a).P]\ ,$$

ce qui n'a aucun sens.

Ces deux cas sont des exemples d'effets de bord incontrôlés, qui apparaissent dans Dπ comme dans SafeDPI. Notre propos, dans ce document, n'est pas de régler ce problème, que les auteurs de SafeDPI traitent à l'aide d'un système de types. Nous supposerons simplement que ces cas ne posent pas de problèmes dans les termes que nous traitons. Si nous devions écarter nous-mêmes les termes incorrects, il nous suffirait probablement d'adapter le système de types de SafeDPI. Nous ne présenterons pas ici ce système de types, qui sort du cadre de notre travail.

7.3.2 Exemples

Canal de communications diadiques

Il est possible de profiter du système des localités pour concevoir des canaux de communications polyadiques plus riches que ceux que nous avions vu pour le π-calcul.

$$
\begin{array}{lcl}
\texttt{Send } v_1, v_2 \texttt{ on } c.P & \triangleq & (\texttt{newloc } buf[\\
& & \quad \overline{c_1}\langle v_1\rangle \mid \overline{c_2}\langle v_2\rangle \\
& & \quad \mid \texttt{come } any \\
& &] : A_{buf}).(\\
& & \quad \overline{c}\langle buf\rangle.(\neg buf).P \\
& &) \\
\texttt{Receive } x_1, x_2 \texttt{ on } c@self.Q & \triangleq & c(buf).(\\
& & \quad (\texttt{newchan } r_1 : C_1, r_2 : C_2, mig : C_{mig})(\\
& & \quad \texttt{go } buf.c_1(y_1).c_2(y_2). \\
& & \quad \texttt{go } mig@self.\overline{r_1}\langle y_1\rangle.\overline{r_2}\langle y_2\rangle \\
& & \quad \mid \texttt{come } mig.r_1(x_1)r_2(x_2). \\
& & \quad (\neg r_2).(\neg r_1).(\neg mig).(\neg buf).Q \\
& &)
\end{array}
$$

Ainsi, nous commençons par créer une localité buf, dans laquelle deux canaux c_1 et c_2 publics émettent les valeurs v_1, v_2. Le secret de la communication est garanti par le secret de buf. Un processus du site source migre vers buf par le biais du canal de migration any public, y lit les valeurs, retourne dans le site source par le biais du canal de migration mig privé et réémet ces valeurs localement sur des canaux privés r_1 et r_2. Le processus $(\neg buf).Q$ est libérable après la lecture des deux valeurs x_1 et x_2 sur r_1 et r_2. Comme l'agent migrant a quitté buf, cette localité peut être nettoyée, déclenchant ainsi P et Q.

Par rapport à ce que nous avions présenté en π ou Cπ, ces macros ont le défaut énorme d'impliquer la création et la destruction d'un nouveau site et plusieurs migrations de processus – sans compter la nécessité de connaître le nom $self$ du site source. Notons tout de même que ce site, ainsi que les processus, ne représentent pas des localités physiques et n'impliquent pas une réelle distribution. Par conséquent, dans une implantation de CDπ, un tel programme pourrait être compilé à l'aide d'une notion de "localités légères" – comme il existe des "processus légers" dans les systèmes d'exploitation modernes – dont les spécifications resteraient à déterminer.

De même, notons que l'appartenance des canaux c_1 et c_2 est ambiguë – ces canaux n'ont certainement pas été créés par buf – et pourra donc poser des problèmes du point de vue de la résolution des effets de bord.

Le seul bénéfice que nous tirons de ces macros est la possibilité de typer des communications lorsque v_1 et v_2 ne sont pas du même type.

Canal de communications distantes

Comme les communications de CDπ sont purement locales, il peut être intéressant d'encoder des macros de communications distantes.

$$
\begin{array}{lcl}
\texttt{Send } v \texttt{ to } c@s & \triangleq & \texttt{go } distant@s.\overline{c}\langle v\rangle \\
\texttt{Receive } x \texttt{ on } c.Q & \triangleq & \texttt{come } distant.c(x).Q \\
\texttt{Send } v \texttt{ to } c@s \texttt{ from } a.P & \triangleq & (\texttt{newchan } return : N_{return})(\\
& & \quad \texttt{go } distant@s.\overline{c}\langle v\rangle.\texttt{go } return@a \mid \\
& & \quad \texttt{come } return.(\neg return).P \\
& &)
\end{array}
$$

$$
\begin{array}{lcl}
\blacksquare_l & \triangleq & \overline{l}\langle\diamond\rangle \\
\texttt{Heap}_n\ l & \triangleq & \underbrace{\blacksquare_l \mid \ldots \blacksquare_l}_{n \text{ ressources}} \\
\texttt{Loop}\ l & \triangleq & !\texttt{come}\ request.from(f).rep(r).l(\_). \\
 & & \quad (\texttt{newloc}\ a\,[\texttt{come}\ baby] : A_a).(\\
 & & \quad\ \texttt{Send}\ a\ \texttt{to}\ r@f \mid (\urcorner a).\blacksquare_l \\
 & & \quad) \\
\texttt{BRM}_n & \triangleq & (\nu_{chan}\ l@brm : N_l)(brm\,[\texttt{Heap}_n\ l] \mid brm\,[\texttt{Loop}\ l])
\end{array}
$$

FIG. 7.7 – Gestionnaire de ressources automatique – version CDπ.

Les macros `Send` et `Receive`, au demeurant classiques, permettent respectivement d'émettre sur un canal c présent dans un site distant s et de recevoir sur un canal local c. La deuxième macro `Send` présente une variante synchrone de l'émission. Cette macro nécessite un paramètre a, qui précise le nom du site émetteur. Malheureusement, les deux macros `Send` nécessitent en fait des communications d'ordre supérieur entre deux sites – opération complexe, longue et risquée en présence d'un réseau non fiable – pour encoder une simple communication d'information entre ces sites.

Nous réutiliserons cependant ces macros dans les exemples suivants.

Gestionnaire de ressources distribuées automatique

Le principe du gestionnaire de ressources distribuées, présenté sur la figure 7.7 est très proche de sa contrepartie dans Cπ. La différence majeure vient du fait que, au lieu d'allouer des canaux de communication – puisque nous interdisons de partager un canal entre plusieurs sites – $\texttt{BRM}_n$ alloue de nouvelles localités. Ainsi, $(\texttt{newloc}\ a\,[\texttt{come}\ baby] : A_a)$ crée une nouvelle localité nommée a, de type A_a et qui ne contient que l'autorisation d'entrer avec la clef $baby$. À charge pour le client de remplir la localité avec un processus P au choix, par un go $baby@a.P$.

Lorsque a est vide, il est nettoyé par la règle RED-STOP puis, lorsque le seul processus contenant a est de la forme $(\nu_{loc}\ a@brm)brm\,[(\urcorner a).\blacksquare_l]$, le nom a est supprimé et les ressources sont retournées à brm.

Ici aussi, la paternité de $baby$ est ambiguë.

7.3.3 Élimination des processus morts (esquisse)

De la même manière que pour Cπ, nous devrions définir le langage complet CDπ à partir du langage de base et d'une élimination de processus morts. En l'absence de résultats, nous nous contenterons de présenter quelques réflexions sur le sujet.

Notre objectif est de définir une relation de simulation barbelée adaptée à CDπ et d'en tirer une relation d'élimination des processus morts dans un contexte distribué. Nous espérons alors pouvoir compléter CDπ de la même manière que nous avons complété Cπ. En soi, l'ajout de la distribution complique

la situation, ne serait-ce que parce qu'un ramasse-miettes ne peut se permettre de figer plusieurs ordinateurs le temps de supprimer une seule valeur.

Les théories comportementales existantes pour Dπ [39] et SafeDPI [40] formeraient un point de départ envisageable pour l'étude, auxquelles il serait nécessaire d'ajouter l'observation de la finalisation. Cependant, ces théories font appel à des systèmes de types tels que ceux présentés dans la section 10.1 pour déterminer les contextes raisonnables dans lesquels prouver les équivalences. Ainsi, si un site n'a pas distribué d'autorisations de communiquer le long d'un canal c, les équivalences considéreront qu'aucun site extérieur ne pourra envoyer de processus communiquer sur c.

D'autres propriétés, comme la notion de paternité d'un site ou d'un nom de canal ne sont pas nécessairement identiques entre nos travaux et ceux que nous désirons employer. Nous ne sommes pas encore certains que ces approches soient appropriées pour gérer les propriétés qui nous intéressent, ni qu'elle puisse s'étendre simplement pour prendre en compte nos opérations de garbage-collection et de finalisation.

7.3.4 Des types pour contrôler les ressources

Le système de types de CDπ étend naturellement celui de Cπ.

Définition 26 (Politique d'allocation des ressources – CDπ)

Une politique de contrôle des ressources est composée :
- *d'une fonction partielle qui, à chaque nom de canal pour laquelle elle est définie, associe l'encombrement du canal ;*
- *d'une fonction partielle qui, à chaque nom de site pour laquelle elle est définie, associe l'encombrement du site et la taille de sa réserve.*

La notion d'utilisation des ressources par un processus s'étend naturellement dans le cadre de CDπ. Ainsi, l'utilisation de ressources par un processus `if _ then` P `else` Q est nulle comme l'était celle de $P + Q$, l'utilisation de ressources par (`newchan` $a : C)P$ se calcule comme se calculait celle de $(\nu a : C)P$ et celle de `go` $a@c.P$ ou de `come` $a.P$ est nulle, comme celle des processus préfixés. Reste le cas de (`newloc` $a\,[P] : A)Q$. Comme dans les Controlled Ambients, nous vérifions si le site peut contenir le processus et nous comptabilisons les ressources utilisées par la localité. Ainsi, nous avons $res_\Gamma((\texttt{newloc}\ a\,[P] : A)Q) = res_{\Gamma,a:e}(Q) + e$ si l'annotation A spécifie que a a un encombrement e et une réserve de taille s et si $res_{\Gamma,a:e}(P) \leq s$.

La définition de res_Γ s'étend ensuite aux réseaux : $res_\Gamma(M)$ est alors une fonction qui, à chaque nom de site pour laquelle elle est explicitement définie associe le nombre de ressources allouées à ce site et à tous les autres noms associe 0. Si res_Γ est une telle fonction, nous noterons $res_\Gamma(M)\backslash\{x\}$ la fonction qui à x associe 0 et à tout autre y associe $res_\Gamma(M)(y)$.

Définition 27 (Utilisation des ressources par un réseau – CDπ)

L'utilisation des ressources par un réseau M, dans le cadre d'une politique de contrôle des ressources Γ, notée $res_\Gamma(M)$, est définie par
- $res_\Gamma(\epsilon) = \forall x, x \mapsto 0$
- $res_\Gamma(M|N) = res_\Gamma(M) + res_\Gamma(N)$

C	::=	$Name(T, e)$	$e \in \mathbf{N}$
T	::=	Ssh	
	|	$Chan(C, z)$	$z \in \mathbf{Z}$
	|	$Migrate(U)$	
	|	A	
U	::=	$Proc(t)[\lambda]$	$t \in \mathbf{N} \cup \{\infty\}, \lambda$ liste de noms
A	::=	$Site(s, e)$	$s \in \mathbf{N} \cup \{\infty\}, e \in \mathbf{N}$
Z	::=	$Net(f)$	$f : \mathcal{N} \longrightarrow \mathbf{N} \cup \{\infty\}$où $\mathcal{N}$ est l'ensemble des noms

FIG. 7.8 – Grammaire de types pour le contrôle des ressources – Cπ

- $res_\Gamma((\nu_{loc}\ a@b : A)M) = res_\Gamma(M)\backslash\{a\} + (b \mapsto e)$ *si l'annotation* A *spécifique que* a *a un encombrement* e *et une réserve de taille* s *et si* $res_\Gamma(M)(a) \leq s$
- $res_\Gamma((\nu_{chan}\ a@b : C)M) = res_\Gamma(M) + (b \mapsto e)$ *si l'annotation* C *spécifie que* a *a un encombrement* e
- $res_\Gamma(a\,[P]) = a \mapsto res_\Gamma(P)$.

Dans tous les autres cas, $res_\Gamma(M)$ *n'est pas défini.*

On dira alors qu'un réseau M*, dans l'état courant, respecte la politique de* Γ *si* $res_\Gamma(M)$ *est défini et si, pour tout site* a *dont* Γ *spécifie que la taille de la réserve est* e_a*, on a* $res_\Gamma(M)(a) \leq e_a$.

Pour l'essentiel, cette définition comptabilise les entités définies par chaque site. Contrairement à ce que nous avions dans Cπ, $(\nu_{loc}\ a)$ sert de borne pour le nom de site a et donc pour toutes les entités définies pour a. Ainsi, dès que nous rencontrons $(\nu_{loc}\ a)$, comme lorsque nous rencontrions une définition de localité dans les calculs à mobilité de sites, nous pouvons compter définitivement la somme de toutes les contributions à a puis les oublier. Ce mécanisme est assez proche, d'ailleurs, de ce que nous pouvions imaginer dans le contexte des régions à la section 7.1.

Définition 28 (Ressource-contrôlé – CDπ)
Un réseau M *est dit ressource contrôlé dans le cadre d'une politique de contrôle des ressources* Γ *si, pour tout* N *tel que* $M \longrightarrow^* N$, N *respecte* Γ.

La figure 7.9 présente un fragment d'un système de types pour le contrôle des ressources dans CDπ.

En l'absence d'une sémantique stable du langage, nous ne présentons pas de propriétés sur ce système de types.

7.3.5 Exemples

Comme le typage dans CDπ est très proche de ce que nous avons déjà vu dans Cπ, nous nous contenterons de présenter les exemples de manière informelle.

Canal de communications diadiques

Contrairement au canal de communications diadiques Cπ, le canal de communications diadiques CDπ se type naturellement même lorsque v_1 et v_2 sont

$$\text{T-IfThenElse}\ \frac{\Gamma \vdash P : Proc(t)[\lambda] \qquad \Gamma \vdash Q : Proc(t)[\lambda]}{\Gamma \vdash \texttt{if \_ then } P \texttt{ else } Q : Proc(t)[\lambda]}$$

$$\text{T-Go}\ \frac{\Gamma \vdash P : U \qquad \Gamma \vdash a : Migrate(U)}{\Gamma \vdash \texttt{go } a@b.P : Proc(t)[\lambda]}$$

$$\text{T-Come}\ \frac{\begin{array}{c}\Gamma \vdash a : Migrate(Proc(t_P)[\lambda_P]) \\ \Gamma \vdash Q : Proc(t_Q)[\lambda_Q]\end{array}}{\Gamma \vdash \texttt{come } a.Q : Proc(u)[\lambda_P \cup \lambda_Q]}\ u \geq t_P + t_Q,\ \lambda_P \cap \lambda_Q = \emptyset$$

$$\text{T-SpawnLoc}\ \frac{\begin{array}{c}\Gamma, a : T \vdash P : Proc(s) \\ \Gamma, a : T \vdash Q : Proc(t_Q) \qquad T = Name(Site(s), e)\end{array}}{\Gamma \vdash (\texttt{newloc } a\,[P] : T)Q : Proc(u)}\ u \geq t_Q + e$$

$$\text{T-NetEmpty}\ \Gamma \vdash \epsilon : Net(\{\})$$

$$\text{T-NetPar}\ \frac{\Gamma \vdash M : Net(Z_M) \qquad \Gamma \vdash N : Net(Z_N)}{\Gamma \vdash M|N : Net(Z_M + Z_N)}$$

$$\text{T-NetNewLoc}\ \frac{\Gamma, a : T \vdash M : Net(Z_M, a : \_) \qquad T = Name(Site(s), e)}{\Gamma \vdash (\nu_{loc}\ a@b : A)M : Net(Z_M + b \mapsto e)}$$

$$\text{T-NetNewChan}\ \frac{\Gamma, c : C \vdash M : Net(Z_M) \qquad C = Name(\_, e)}{\Gamma \vdash (\nu_{chan}\ c@b : C)M : Net(Z_P) + c \mapsto e}$$

$$\text{T-NetSite}\ \frac{\Gamma \vdash P : Proc(t_P) \qquad \Gamma \vdash a : Name(Site(\_), \_)}{\Gamma \vdash a\,[P] : Net(a \mapsto t_P)}$$

FIG. 7.9 – Typage des ressources dans CDπ – fragment.

de types différents.

Une fois de plus, nous avons le choix sur la manière d'équilibrer le coût de la communication. Pour ce qui suit, sans entrer dans le détail, nous supposerons que le coût de la création de buf est masqué par t_P – c'est-à-dire $t_P \geq e_{buf}$ – et que le coût de la création de r_1, r_2 et mig est masqué par t_Q – c'est-à-dire $t_Q \geq e_1 + e_2 + e_{mig}$. Nous pouvons alors aisément prouver que le processus émetteur nécessite $t_P - z$ ressources et que le processus récepteur coûte $t_Q + z$ ressources.

En d'autres termes, du point de vue des ressources comme du point de vue des communications, ces macros généralisent les primitives de communication de Cπ/CDπ.

Canal de communications distantes

Pour typer les communications distantes, il semble cohérent de supposer que le canal de migration *distant* est public et commun à toutes les communications. Nous supposerons donc que ce canal, pour assurer un minimum de sécurité, interdit aux processus qui migrent de procéder à des allocations. En d'autres termes, nous aurons $\Gamma \vdash distant : Migrate(Proc(0)[\emptyset])$. Par suite, nous n'avons guère de liberté pour typer c, puisque l'intégralité du coût de la communication doit reposer sur le récepteur. Nous aurons donc $z = 0$ pour c.

Nous pouvons alors prouver que le premier processus d'émission nécessite 0 ressources, tandis que le deuxième en nécessite t_P, pour peu que le coût de P masque le coût de création de *return*. Quant au processus de réception, il aura besoin d'autant de ressources que Q.

En particulier, cela signifie que, conformément à l'intuition, si Q nécessite des ressources pour s'exécuter, nous ne pouvons accepter un processus répliqué de la forme !`Receive` x `on` $c.Q$, qui permettrait une attaque de Déni de Service depuis un site extérieur.

7.3.6 Discussion

Le calcul CDπ, actuellement en chantier, est un formalisme intermédiaire entre Dπ, SafeDPI et Cπ, qui ajoute à Dπ une sémantique plus contrôlée de la création de sites et de la migration de processus, ainsi qu'une primitive de finalisation. Cette primitive permet de synchroniser un processus sur la récupération de ressources allouées à un nom, que ce nom représente un canal de communication ou de migration ou encore une localité. Idéalement, CDπ devrait être complété par une formalisation de l'élimination des processus morts, tâche que nous n'avons pas encore menée à bien.

L'essentiel des remarques qui s'appliquent à Cπ sont encore valables dans CDπ. Nous ne répéterons donc pas nos commentaires sur la manière de représenter les ressources, sur les similitudes avec les calculs précédents ou avec les types linéaires.

Ajoutons cependant que, contrairement à tous les autres formalismes que nous avons présentés, le mécanisme de création de localités différencie effectivement la création d'un site par (`newloc` $a\,[P]$) de la présence d'un site, notée $(\nu_{loc}\ a@l)a\,[P]$. De même, le mécanisme de génération de nouveaux noms différencie effectivement la création d'un nom par (`newchan` $c)P$ de la présence de ce nom, notée $(\nu_{chan}\ c@a)$. Ces distinctions, présentes aussi dans SafeDPI, en plus de rendre le sens des constructions plus clair, permettent de garantir dans la sémantique du langage l'unicité des noms pour les sites. En revanche, cette différenciation ne semble pas apporter grand chose au contrôle des ressources.

Notons aussi que nous avons préféré ne pas introduire de finalisations distribuées car elles auraient imposé une synchronisation distribuée entre un nombre de parties souvent difficile à déterminer et car nous considérons que le nom détruit appartient à une seule localité. Cependant, nous ne pensons pas que cette limitation constitue un handicap car il reste possible de déclencher un finaliseur qui, à son tour, enverra des messages pour prévenir les autres sites de la disparition du nom.

Chapitre 8

Bilan

Dans cette partie, nous avons abordé le problème de la gestion des ressources dans le cadre de calculs à mobilité de noms. Pour ce faire, nous avons considéré le π-calcul et quelques variantes et nous avons proposé une notion formelle de ressources dans laquelle chaque nom représente une entité qui occupe des ressources et chaque création / nettoyage de nom constitue une allocation/une désallocation de ressources.

Allocations, désallocations, nettoyage De la même manière que dans les calculs à mobilité de sites, la première étape dans notre approche du contrôle des ressources a consisté à mettre en évidence les opérations d'allocation et de désallocation, ou qui peuvent mettre en évidence l'allocation et la désallocation. Dans Cπ, le mécanisme d'allocation est restreint à l'opérateur de création de nom (νc) qui, comme $a\,[P]$ dans les ambients, dénote une allocation de ressources sans préciser si cette allocation doit avoir lieu ou si elle a déjà été exécutée. Cette construction est compliquée par les mécanismes d'extrusion, qui peuvent déplacer (νc) à l'intérieur d'un processus, ce qui rend l'allocation plus difficile à comprendre. À l'opposé, le mécanisme d'allocation de CDπ, noté (`newloc`) ou (`newchan`), est plus clair, puisqu'il est syntaxiquement distinct des marqueurs qui attestent de l'existence des entités ainsi créées.

Dans les deux cas, à l'inverse de ce que nous avions fait précédemment, la désallocation est automatique mais déclenche une opération de finalisation, inspirée du comportement de certains langages de programmation. Ce mécanisme, relativement simple, et d'autant plus naturel qu'il s'agit d'une fonctionnalité de nombreux ramasse-miettes, est à notre sens de beaucoup plus haut niveau que les cocapacités ou les continuations dans les Controlled Ambients ou les Kells, par exemple. En fait, la finalisation pourrait être réécrite comme une communication (répliquée) depuis un gestionnaire de ressources abstrait ou représenté, par exemple, dans une membrane. Pour le dire autrement, cette conception de la désallocation nous semble plus proche de ce que nous avions mentionné comme alternative à nos méthodes dans le cadre des calculs à mobilité de sites : l'utilisation de gardiens [26] ou de processus [74] de contrôle.

Notons que, si l'opération (ν) existe dans les calculs des ambients, les (ν) peuvent flotter librement ou presque d'un ambient à un autre. Dans les calculs à mobilité de sites, (ν) se marierait donc difficilement avec notre notion d'entités qui occupent des ressources dans une localité donnée. Ainsi, si rien n'interdit

d'essayer de compter les occurrences de (νc) dans ces formalismes, nous n'avons pas essayé d'étudier le problème.

Définition formelle des ressources La définition formelle des ressources que nous employons dans cette partie est similaire à celle que nous avons présentée pour les ambients. Ainsi, qu'il s'agisse de *toplevel* dans le Cπ, des groupes dans l'extension du calcul aux groupes ou des sites dans CDπ, chaque nom de localité, est annoté par le nombre de ressources dont dispose sa réserve et chaque nom est annoté par le nombre de ressources qu'il occupe dans la réserve de la localité qui "possède" le nom. La distinction entre capacité et encombrement d'une localité est plus naturelle encore que dans les ambients puisque, en pratique, on peut supposer qu'un site n'occupera généralement pas de ressources dans le site qui l'a créé.

Une fois de plus, notre définition formelle des ressources hors du langage nous permet de conserver une indépendance vis-à-vis de la politique de gestion des ressources, indépendance que nous exploiterons dans la section 9. Nous avons cependant noté que, si nous considérons un π-calcul avec groupes, des ressources explicitement représentées dans le langage pourraient probablement permettre une formulation dans laquelle les entités sont allouées progressivement, plutôt que toutes en une seule fois – et toutes libérées simultanément par la suite.

Système de types Les systèmes de types que nous avons proposés pour les calculs à mobilité de noms sont semblables par bien des points à leurs homologues pour la mobilité de sites.

La première différence majeure est l'équilibrage des coûts de communication, qui permet de gérer finement un grand nombre de situations, notamment les processus répliqués. Cet équilibrage nous semble être l'un des aspects les plus intéressants de notre méthode, et nous comptons le conserver dans les évolutions suivantes de nos travaux. L'une des raisons pour lesquelles nous n'avons développé cette technique qu'avec le π-calcul, et non pas dès les calculs à mobilité de sites, est que cette comptabilité nous semble n'avoir de sens que pour des communications locales.

La seconde différence majeure est la nécessité de distribuer les autorisations de réutiliser les ressources entre différents processus – ce qui pourrait, dans une version raffinée des systèmes, se transformer en une nécessité de gérer la redistribution statique de ces ressources entre les processus. Si cet aspect du système est indispensable, nous le considérons comme maladroit et nous souhaiterions trouver un moyen de l'améliorer ou de nous en dispenser, peut-être en l'unifiant avec l'équilibrage des coûts de communication.

Dans tous les cas, ici aussi, les systèmes de types se sont révélés suffisamment puissants pour prouver des propriétés sur nos exemples ou pour mettre en évidence quelques schémas d'attaques élémentaires, notamment sur la réplication.

8.1 En fin de compte

Les méthodes que nous avons proposées pour le contrôle des ressources dans les calculs à mobilité de sites s'adaptent aux calculs à mobilité de noms. Nous allons voir dans la prochaine partie comment exprimer des propriétés plus complexes à l'aide des mêmes méthodes, d'une manière commune à tous ces calculs.

Quatrième partie

Pour d'autres notions de ressources

Chapitre 9

Plus de contrôles

Jusqu'à présent, nous avons vu plusieurs méthodes pour gérer les ressources en présence de mobilité, de distribution et de parallélisme, à l'aide de formalismes adaptés à ces aspects du problème. Pour ce faire, nous avons adapté ou étendu certains des langages, et nous avons employé des systèmes de types "quantitatifs", c'est-à-dire conçus pour mesurer, essentiellement, le nombre d'occurrences de certaines opérations dans un terme donné.

Or, comme nous en avions discuté dans la section 2.4.2, il existe d'autres conceptions, bien plus "qualitatives" des ressources et des méthodes pour les contrôler : ainsi, certains travaux visent à assurer le respect de protocoles éventuellement complexes [46] et d'autres à s'assurer que certaines ressources ne sont utilisées que par des agents autorisés [59, 42]. Si ces notions ne sont pas identiques à celles que nous avons exposées, elles présentent des points communs.

Dans cette partie, nous étudions donc brièvement quelques-uns de ces travaux et la possibilité d'employer nos techniques quantitatives pour mesurer des grandeurs qualitatives et ainsi approximer certains de ces contrôles. En pratique, nous n'aborderons pas les méthodes pour contrôler le respect de protocoles complexes, telles que les techniques de (bi)simulation [1, 61], ou les systèmes de types génériques [46] : d'une part, ces approches sont trop vastes pour être présentées sérieusement dans cet exposé et d'autre part, elles fournissent une puissance en termes d'expressivité que nos méthodes ne peuvent fournir.

Pour ce qui suit, et pour reprendre un vocabulaire plus proche de celui que nous avons employé jusqu'ici, nous emploierons plutôt le terme d'"entités" pour désigner ce que les travaux en question considèrent comme ressources. Certains agents – pour une acceptation pour l'instant informelle du terme "agents" – disposent alors d'opérations qui permettent d'accéder à ces entités.

Le problème consiste à autoriser et à interdire – statiquement ou dynamiquement – les accès aux entités lorsque ces accès violent certains critères.

Vers des ressources plus riches Pour commencer, nous introduisons une généralisation de nos systèmes de types et de nos notions de ressources. Jusqu'à présent, nous avons toujours considéré que les ressources utilisées par un processus pouvaient être caractérisées uniquement par un nombre positif ou infini, signifiant ainsi implicitement que toutes les ressources sont identiques.

Dans ce chapitre, nous développons les idées et les techniques qui nous permettent d'employer dans un contexte plus général les systèmes de types que nous

avons précédemment présentés. Au lieu d'un ensemble de ressources confondu avec $\mathbf{N} \cup \{\infty\}$, nous utilisons n'importe quel ensemble vérifiant quelques critères simples. Nous présentons notamment les modifications qu'il est nécessaire d'apporter au système de types de Cπ pour permettre cette généralisation.

La communication comme accès Dans le cadre du π-calcul ou de Dπ, il est naturel d'étudier dans quelles circonstances un processus est autorisé à communiquer sur un canal de communication. Notamment, lorsqu'il s'agit d'un processus arrivé dans un site suite à une migration, il est intéressant de pouvoir contrôler l'ensemble des canaux que cet "agent" peut employer, aussi bien pour émettre que pour recevoir.

Dans le cas dynamique, ce problème peut être partiellement résolu par un masquage soigneux des noms des canaux interdits à l'extérieur. Nous présenterons ici une autre approche, statique et qui étend certains systèmes de types classiques du π-calcul. Dans cette approche [42], les autorisations de lecture ou d'écriture, ainsi que de migration, sont affectées aux processus sous la forme de types, grâce à une relation de sous-typage.

Nous comparons ensuite à cette approche quelques résultats que nous pouvons obtenir avec nos techniques, en utilisant Cπ, CDπ ou même le π-calcul habituel.

Contrôles sur les Ambients Dans le cadre des calculs des Ambients, le problème de l'accès aux entités est en fait le problème des déplacements des ambients : un ambient a peut-il entrer dans un autre ambient b ?

Cette question peut être examinée à l'aide de méthodes dynamiques, par exemple en utilisant des cocapacités telles que celles que nous avons vues dans le cadre des Controlled Ambients.

Une autre manière d'approcher le problème consiste à employer des méthodes statiques, par exemple en précisant dans le type de chaque ambient une liste d'ambients qui sont autorisés à entrer. Nous présenterons brièvement une approche statique du problème qui fait appel à des types dépendants [59] afin de spécifier de manière riche les mouvements autorisés.

Nous comparons ensuite cette méthode aux résultats que nous sommes à même d'obtenir en appliquant nos techniques, à l'aide des Controlled Ambients. Nous discutons aussi brièvement de méthodes pour employer nos techniques dans les Mobile Ambients, qui sont de fait un modèle plus difficile à contrôler.

9.1 Des ressources plus riches

En examinant les définitions de notions telles que l'utilisation des ressources ou les processus ressource-contrôlés, ainsi que les systèmes de types pour le contrôle des ressources, que nous avons présentés aux sections 3.3.4, 4.1.3, 4.2.6, 4.2.4, 6.3.7 et 7.3.4, nous pouvons constater qu'ils font appel à un nombre réduit de primitives.

Ainsi, si l'on exclut l'équilibrage des communications dans Cπ, traité dans la section 6.3.7, toutes les opérations arithmétiques sont des additions ou des comparaisons entre éléments de $\mathbf{N} \cup \{\infty\}$.

Dans cette section, au lieu de $\mathbf{N} \cup \{\infty\}$, nous emploierons des ensembles abstraits de ressources et nous essayerons de déterminer les propriétés du système

de types ainsi obtenu et les garanties que nous pouvons fournir à l'aide de ce système.

9.1.1 Ensembles de ressources plus généraux

Au lieu de $\mathbf{N} \cup \{\infty\}$, considérons des ensembles de ressources définis comme suit :

Définition 29 (Ensemble de ressources)

Un ensemble de ressources est un ensemble $(\mathcal{S}, \bot, \oplus, \preceq)$ *où* $\bot$ *est un élément de* $\mathcal{S}$, $\oplus$ *est une opération binaire et*

1. *l'ensemble* $(\mathcal{S}, \bot, \oplus)$ *est un monoïde commutatif et* $\bot$ *est un élément neutre pour la loi* $\oplus$
2. *la relation* $\preceq$ *est un ordre partiel sur* $\mathcal{S}$ *qui admet* $\bot$ *comme plus petit élément*
3. *pour tout* a, b, c, *si* $a \preceq b$, *alors* $a \oplus c \preceq b \oplus c$.

À l'aide d'un tel ensemble $\mathcal{S}$, nous pouvons réutiliser telles quelles les définitions telles que l'utilisation de ressources, les processus ressource-contrôlés et les systèmes de types que nous avons présentés – à l'exception de l'équilibrage des communications dans Cπ et CDπ, qui nécessite une légère modification, et que nous détaillerons dans la section suivante, et des systèmes raffinés pour les CA, qui nécessitent une restriction que nous présenterons un peu plus loin.

Dans tout ce qui suit, nous utiliserons couramment le symbole $\succeq$ défini par $x \succeq y$ si et seulement si $y \preceq x$.

9.1.2 Exemples

Pour commencer, précisons que les systèmes de types ainsi obtenus peuvent s'instancier de manière à obtenir les systèmes que nous avions étudiés jusqu'ici :

Lemme 31 ()
L'ensemble $(\mathcal{S}, \bot, \oplus, \preceq)$ *suivant est un ensemble de ressources :*

$$\begin{cases} \mathcal{S} & = & \mathbf{N} \cup \{\infty\} \\ \bot & = & 0 \\ \oplus & = & + \\ \preceq & = & \leq \ . \end{cases}$$

Nous ne détaillerons pas la preuve, triviale.

Si nous revenons un bref instant sur le protocole du taxi, tel qu'examiné dans la section 3.3.5, nous pouvons vouloir prouver d'autres propriétés. Ainsi, nous pouvons être intéressé par une garantie sur le fait qu'aucun client ne peut contenir de taxi. Pour ce faire, nous pouvons employer des ressources dans $\mathbf{N} \cup \{\infty\}$. Nous pouvons aussi, de manière plus lisible, employer des ressources booléennes :

Lemme 32 ()
L'ensemble $(\mathcal{B}, \bot, \oplus, \preceq)$ *suivant est un ensemble de ressources :*

$$\begin{cases} \mathcal{B} & = & \{ff, tt\} \\ \bot & = & ff \\ \oplus & = & \vee \\ & & ff \preceq tt \end{cases}$$

Une fois de plus, nous ne fournissons pas de preuve.

Ainsi, un ambient de capacité ff ne peut jamais contenir de sous-ambients de taille tt. Pour notre exemple, nous donnerons donc une taille tt aux taxis eux-mêmes et une capacité ff aux clients.

Il est tout aussi intéressant de noter que les nouveaux systèmes de types permettent de combiner plusieurs politiques en une seule et donc de garantir plusieurs propriétés en une seule vérification de types :

Lemme 33 ()
Si les ensembles $(\mathcal{S}_1, \bot_1, \oplus_1, \preceq_1)$ *et* $(\mathcal{S}_2, \bot_2, \oplus_2, \preceq_2)$ *sont des ensembles de ressources, alors l'ensemble* $(\mathcal{S}, \bot, \oplus, \preceq)$ *suivant est aussi un ensemble de ressources.*

$$\begin{cases} \mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 \\ \bot = \bot_1, \bot_2 \\ (x_1, x_2) \oplus (y_1, y_2) = (x_1 \oplus y_1, x_2 \oplus y_2) \\ (x_1, x_2) \preceq (y_1, y_2) = x_1 \preceq_1 y_1 \wedge x_2 \preceq_2 y_2 \end{cases}$$

Ce lemme est, lui aussi, trivial.

Ainsi, chaque taxi ne peut contenir, au plus, qu'un seul client, chaque taxi ne peut contenir, au plus, qu'un seul ambient auxiliaire et aucun client ne peut contenir de taxis. Grâce à ce lemme, nous pouvons combiner toutes ces propriétés en une seule politique de contrôle des ressources et donc en une seule vérification.

9.2 Le cas du système Cπ

Comme nous l'avons vu, de toutes les opérations arithmétiques que nous avons employées dans nos systèmes de types, les seules qui ne soient ni une addition ni une comparaison d'entiers positifs ou infinis concernent l'équilibrage des coûts de communication dans Cπ ou CDπ, présentés dans les sections 6.3.7 et 7.3.4.

Or, dans un ensemble plus complexe que celui des entiers, il est a priori absurde de considérer que l'on peut équilibrer les charges par une simple addition et une simple soustraction relatives, puisque ces opérations nécessitent en fait dans le cas général de plonger $\mathbf{N} \cup \{\infty\}$ dans $\mathbf{Z} \cup \{-\infty, +\infty\}$.

9.2.1 Adaptation du système Cπ

La figure 9.1 reprend les fragments problématiques de ce système de types. Pour corriger ces règles, nous devons éliminer une addition dans $\mathbf{Z}$ et une soustraction dans $\mathbf{Z}$, tout en conservant la symétrie de cet aspect du typage.

Pour supprimer les opérations sur des entiers relatifs et les remplacer par des opérations sur des entiers naturels, nous pouvons par exemple remplacer z par

$$\begin{array}{rcll} T & ::= & Ssh & \\ & | & Chan(C,z) & z \in \mathbf{Z} \end{array}$$

$$\text{T-Read}\ \frac{\Gamma \vdash c : Name(Chan(C,z),\_) \quad \Gamma, x : C \vdash P : Proc(t_P)[\lambda_P]}{\Gamma \vdash c(x).P : Proc(t_P + z)[\lambda_P]}\ x \notin \lambda, t_P + z \geq 0$$

$$\text{T-Write}\ \frac{\Gamma \vdash c : Name(Chan(C,z),\_) \quad \Gamma \vdash Q : Proc(t_Q)[\lambda_Q] \qquad \Gamma(y) = C}{\Gamma \vdash \overline{c}\langle y\rangle.Q : Proc(t_Q - z)[\lambda_Q]}\ t_Q - z \geq 0$$

FIG. 9.1 – Ancien système de types de Cπ (fragment à modifier).

un couple d'entiers r, w tels que $r - w = z$. Ainsi, $t_P + z$ devient $t_P + r - w$ et $t_Q - z$ devient $t_Q + w - r$ – ces deux opérations restent dans $\mathbf{N}$ car on impose que les résultats soient positifs. Pour éviter les soustractions, nous pouvons alors remplacer $t_P + r - w$ par u tel que $u + w \geq t_P + r$ et $t_Q + r - w$ par v tel que $v + r \geq t_Q + w$.

Bien que cette méthode tienne de l'artifice et fasse apparaître deux paramètres là où le système de types précédent n'en nécessitait qu'un seul, nous pouvons donner une interprétation à r et w. Ainsi, w représente les ressources allouées par $\overline{c}\langle y\rangle.Q$ pour permettre à P de s'exécuter, tandis que r représente les ressources allouées par $c(x).P$ pour permettre à Q de s'exécuter. Dans le cas le plus fréquent, on aura $r = w = 0$, ce qui signifie chacun des deux processus gèrent sa mémoire soi-même. Cependant, $r \neq 0$ pourra représenter le fait que Q émet une requête de ressources et $w \neq 0$ pourra représenter le fait que P ne dispose pas de ressources propres et nécessite qu'on lui fournisse des ressources pour s'exécuter. À partir du moment où r et w ne sont plus des entiers mais des éléments d'un ensemble de ressources $\mathcal{S}$ – notamment lorsque $\mathcal{S}$ est le produit de deux ensembles de ressources $\mathcal{S}_r$ et $\mathcal{S}_w$ – on peut tout à fait concevoir des circonstances dans lesquelles $r \neq \bot$ et $w \neq \bot$, lorsque les processus doivent échanger des ressources pour pouvoir communiquer.

Ainsi, si nous considérons une bibliothèques d'entrées/sorties système, $c(x).P$ peut représenter une fonction d'ouverture de fichier. Pour invoquer cette fonction, un processus $\overline{c}\langle y\rangle.Q$ devra allouer w ressources de mémoire vive pour permettre à l'ouverture de se dérouler sans problèmes. En échange, la fonction allouera r ressources de disque dur pour Q.

Le système adapté, qui apparaît sur la figure 9.2, fait usage de ces paramètres r et w dits *d'échange de ressources.*

9.2.2 Propriétés

Les propriétés principales que nous avions vues à la section 6.3.7 restent valides dans ce système étendu.

Lemme 34 (Sous-typage)
Soient Γ *un environnement et* P *un processus typable dans* Γ. *Notons* $\Gamma \vdash P :$

$$\begin{array}{rcll} T & ::= & Ssh & \\ & | & Chan(C, r, w) & r \in \mathcal{S}, w \in \mathcal{S} \end{array}$$

$$\text{T-Read-Rich}\ \frac{\Gamma \vdash c : Name(Chan(C, r, w), \_) \quad \Gamma, x : C \vdash P : Proc(u_P \oplus w)[\lambda_P]}{\Gamma \vdash c(x).P : Proc(v_P)[\lambda_P]}\ x \notin \lambda, v_P \succeq u_P \oplus r$$

$$\text{T-Write-Rich}\ \frac{\Gamma \vdash c : Name(Chan(C, r, w), \_) \quad \Gamma \vdash Q : Proc(u_Q \oplus r)[\lambda_Q] \qquad \Gamma \vdash y : C}{\Gamma \vdash \overline{c}\langle y \rangle.Q : Proc(v_Q)[\lambda_Q]}\ v_Q \succeq u_Q \oplus w$$

Fig. 9.2 – Nouveau système de types de Cπ (fragment modifié).

$Proc(t)[\lambda]$. Alors, pour tout $x \succeq t$, on aura $\Gamma \vdash P : Proc(x)[\lambda]$.

Le lemme est trivial et est utile exactement dans les mêmes circonstances que son prédécesseur sur $\mathbf{N} \cup \{\infty\}$.

Les lemmes de renforcement et d'affaiblissement, vrais une fois de plus, restent identiques et triviaux.

Lemme 35 (Les bons types respectent la politique)
Dans une politique de contrôle de ressources Γ, si $\Gamma \vdash P : Proc(t)[\_]$ et si $\Gamma(toplevel) = t$, alors P respecte Γ.

La preuve est identique à celle qui est présentée dans l'annexe D.1.

Théorème 14 (Subject Reduction) *Si $\Gamma \vdash P : U$ et si $P \longrightarrow Q$, alors $\Gamma \vdash Q : U$.*

Nous présentons le fragment intéressant de la preuve dans l'annexe E.1.

Théorème 15 (Contrôle des ressources) *Soient Γ un environnement et P un processus. Si $\Gamma \vdash P : Proc(t)[\_]$ et si $\Gamma(toplevel) = t$, alors P est ressource-contrôlé dans Γ.*

9.2.3 Discussion

Nous avons présenté une généralisation de notre système de types pour Cπ et CDπ qui permet d'exprimer des politiques plus fines que celles auxquelles nous avions accès dans l'ensemble de ressources $\mathbf{N} \cup \{\infty\}$.

Les propriétés que nous avions prouvées dans $\mathbf{N} \cup \{\infty\}$ restent valides dans des ensembles moins contraints, notamment avec des n-uplets de booléens. Ces n-uplets, en particulier, constituent une manière plus lisible de représenter certaines politiques de contrôle des ressources.

Nous verrons dans les chapitres qui suivent des exemples d'utilisation de ces nouveaux systèmes de types.

$$\text{T-BetterCoOpen}\ \frac{\Gamma \vdash m : Amb(s,e,q,r)[T] \quad \Gamma \vdash Q : Proc(q)[T]}{\Gamma \vdash \overline{\texttt{open}}\ m.Q : Proc(v_Q)[T]}\ v_Q + r \geq s, s \neq \infty$$

FIG. 9.3 – Ancien système de types raffiné de CA (fragment à modifier).

9.3 Le cas du système raffiné CA

Un second problème, plus subtil, intervient dans la définition du système de types raffiné, rappelé sur la figure 9.3. En effet, la règle T-BetterCoOpen impose $s \neq \infty$.

Comme nous l'avons vu, la condition $s \neq \infty$ est nécessaire pour assurer la validité de la propriété principale pour laquelle est conçue cette règle, à savoir $r \geq t_r$. Malheureusement, cette assertion ne peut se traduire simplement dans un ensemble de ressources $\mathcal{R}$ plus général que $\mathbf{N} \cup \{\infty\}$.

9.3.1 Adaptation du système raffiné CA

Afin de garantir $t_R \preceq r$, nous remplaçons les deux conditions $v_Q + r \geq s$ et $s \neq \infty$ par la vérification suivante : $\forall w \forall h, w \succeq v_Q \wedge w \oplus h \preceq s \Rightarrow h \preceq r$. Dans $\mathbf{N} \cup \{\infty\}$, si $s \neq \infty$ ce prédicat se traduit par $v_Q + r \geq s$, comme dans la version précédente de la règle. Plus généralement, dans tout ensemble de ressources $\mathcal{R}$, en instanciant w par v_Q et h par t_R nous pouvons déduire $t_R \preceq r$ – nous utilisons $v_Q \oplus h$ car $v_Q \oplus t_R = s$ dès que $m\,[\texttt{open}\ m.Q \mid R]$ est typable. Sans entrer dans tous les détails, le paramètre w est nécessaire pour permettre la propriété de sous-typage (cf. section 3.3.6).

C'est cette vérification que nous employons dans la règle modifiée, présentée sur la figure 9.4. Nous n'avons malheureusement pas d'interprétation plus intuitive de cette assertion.

$$\text{T-BetterCoOpen}\ \frac{\Gamma \vdash m : Amb(s,e,q,r)[T] \qquad \Gamma \vdash Q : Proc(q)[T] \qquad \forall w, \forall h, w \succeq v_Q \wedge w \oplus h \preceq s \Rightarrow h \preceq r}{\Gamma \vdash \overline{\texttt{open}}\ m.Q : Proc(v_Q)[T]}$$

FIG. 9.4 – Nouveau système de types raffiné de CA (fragment modifié).

9.3.2 Propriétés

Les propriétés principales que nous avions vues à la section 3.3.4 restent valides dans ce système étendu.

Lemme 36 (Sous-typage)
Soient Γ un environnement et P un processus typable dans Γ. Notons $\Gamma \vdash P : Proc(t)[T]$. Alors, pour tout $x \succeq t$, on aura $\Gamma \vdash P : Proc(x)[T]$.

Le lemme est trivial et est utile exactement dans les mêmes circonstances que son prédécesseur sur $\mathbf{N} \cup \{\infty\}$.

Les lemmes de renforcement et d'affaiblissement, vrais une fois de plus, restent identiques et triviaux.

Lemme 37 (Les bons types respectent la politique)
Si $\Gamma \vdash P : U$, *alors* P *respecte* Γ.

La preuve est identique à celle qui est présentée dans l'annexe A.1.

Théorème 16 (Subject Reduction) *Si* $\Gamma \vdash P : U$ *et si* $P \longrightarrow Q$, *alors* $\Gamma \vdash Q : U$.

Nous présentons le fragment intéressant de la preuve dans l'annexe E.2.

Nous verrons dans les chapitres qui suivent des exemples plus intéressants.

Chapitre 10

Contrôle d'accès aux entités dans π

Dans le cadre du π-calcul et de ses variantes, une fois de plus, le concept naturel d'entité "à contrôler" est confondu avec celui de canal de communication. Le problème du contrôle d'accès aux entités est celui des autorisations de communication le long des canaux. Notamment, dans Dπ, en présence de migration, un processus malicieux et incontrôlé peut intercepter ou parasiter des communications locales.

Dans cette section, nous nous intéresserons essentiellement à Dπ et à CDπ mais une bonne partie des méthodes sont applicables au π-calcul seul ou à Cπ. Sous le terme d'"agents", nous désignerons donc les processus migrants, tandis que "accéder" à une entité/canal reviendra à émettre ou à recevoir le long de ce canal. Nous distinguerons l'émission et la réception.

Tels quels, ces calculs ne proposent qu'un contrôle minimal des communications : à partir du moment où un processus connaît le nom d'un site et d'un canal de ce site, rien ne peut l'empêcher de migrer vers cette localité et de s'y placer comme émetteur ou/et récepteur. De même, à partir du moment où un processus connaît le nom d'un canal de communication, rien ne peut l'empêcher de lire ou d'écrire sur ce canal.

Dans une première section, nous présentons une méthode ad-hoc [42] adaptée à π et à Dπ. La technique consiste à spécifier les autorisations de communiquer sous la forme de types. Comme le type d'un nom reçu par un processus est déduit du type du canal utilisé pour communiquer ce nom, il est possible de fixer ce type de canal de manière à ce que le récepteur ne dispose pas de tous les droits d'accès au nom – c'est ce que nous appellerons "transférer un type", même si le mécanisme est, en fait, purement statique.

Nous comparerons ensuite avec les résultats que nous sommes en mesure d'obtenir à l'aide d'une application de notre système de types pour Cπ et CDπ.

Enfin, nous introduirons une autre application de notre système de types, qui approxime cette fois le système linéaire présenté à la section 10.1, et permet un contrôle beaucoup plus précis du nombre de communications.

10.1 Un système de types pour π et Dπ

Le seul moyen qui ait été, à notre connaissance, proposé pour contrôler statiquement ces accès dans π ou Dπ consiste à communiquer des autorisations d'émettre ou de recevoir, sous la forme de types. C'est ces travaux [69, 42] dont nous présentons ici un aperçu.

L'idée consiste à introduire une relation de sous-typage. Ainsi, un canal c, dont le type principal précise qu'il est utilisable sans restriction, peut aussi être considéré comme utilisable uniquement en lecture, uniquement en écriture ou pas du tout. Lorsque le nom de c est communiqué, le type du canal employé contraint le type de c du point de vue du récepteur.

10.1.1 Spécification d'une politique

Le type le plus complet d'une localité m qui contient les canaux c_1, c_2, ..., c_n est de la forme $loc\{c_1 : C_1, c_2 : C_2, \ldots, c_n : C_n\}$ si chaque C_i est le type du canal correspondant. Chaque C_i peut alors exprimer un certain nombre de propriétés, notamment le type des informations qui peuvent transiter par c_i ou la possibilité de lire ou d'écrire sur c_i.

Cependant, cette même localité m pourra aussi être considérée comme étant de type $loc\{c_2 : C_2, \ldots, c_n : C_n\}$, si l'on s'interdit de communiquer sur c_1, ou de type $loc\{c_1 : C_1', c_2 : C_2, \ldots, c_n : C_n\}$ avec C_i' "plus faible" que C_i, si l'on ne s'autorise qu'à utiliser certaines fonctionnalités de c_1.

Partant de ce principe, si l'on veut qu'aucun processus migrant ne puisse jamais communiquer sur c_1 à moins d'en avoir reçu l'autorisation localement, il suffit de s'assurer que toutes les communications qui peuvent transmettre m à un autre site donneront à m un type suffisamment restrictif.

D'autres annotations de types peuvent autoriser la création d'un canal dans un site ou la migration vers un site. Ainsi, pour permettre à un processus de connaître le nom m, l'autoriser à migrer vers m mais lui interdire de créer de nouveaux canaux dans m, on s'assurera que le type de m connu par ce processus contiendra l'annotation *go* mais pas l'annotation *newc*.

Ainsi, si nous considérons le processus

$$m\,[\mathtt{go}\ n.\overline{c}\langle m\rangle]\ \mid\ n\,[c(x).\mathtt{go}\ x.P]\,,$$

pour peu que le type de c spécifie que la valeur transportée est une localité de type $loc\{k : w\langle T\rangle, go\}$, le processus P pourra migrer vers m, ne pourra accéder qu'au canal k, uniquement en écriture et uniquement pour transmettre des informations de type T et ne pourra pas créer de nouveaux canaux dans m.

10.1.2 Vérification des types

Contrairement au système de types que nous présenterons dans la section 11 pour les Mobile Ambients, le contrôle d'accès aux canaux dans Dπ ne nécessite pas de types dépendants.

La figure 10.1 présente un fragment du système de types. La notation $\Gamma \vdash_m P$ signifie que P est typable dans le site m – seul le type de m compte, si bien que m peut être aussi bien un nom réel de site qu'une variable liée par une réception.

La règle T-W permet ainsi de typer l'émission sur le canal u d'une valeur v. Pour que cette action soit correcte, il est nécessaire de disposer du droit

$$\text{T-W}\ \frac{\Gamma \vdash_w u : w\langle T\rangle \qquad \Gamma \vdash_w v : T \qquad \Gamma \vdash_w P}{\Gamma \vdash_w \overline{u}\langle v\rangle.P}$$

$$\text{T-Go}\ \frac{\Gamma \vdash_u P \qquad \Gamma(u) : loc(go)}{\Gamma \vdash_w \mathtt{go}\ u.P}$$

$$\text{T-R}\ \frac{\Gamma, w : loc\{Z\} \vdash_w u : r\langle T\rangle \qquad \Gamma, w : loc\{x : T, Z\} \vdash_w Q}{\Gamma, w : loc\{Z\} \vdash_w u(x : T).Q}\ x \notin fv(\Gamma)$$

$$\text{T-NameLoc}\ \frac{\Gamma(w) <: loc(u : T)}{\Gamma \vdash_w u : T}$$

FIG. 10.1 – Fragment du système de types pour le contrôle des canaux dans Dπ.

d'écrire sur u des valeurs de type T et de tenter d'y écrire une valeur v qui est effectivement du type T. De même, la règle T-Go spécifie qu'une migration n'est possible que si l'on dispose de l'autorisation de procéder à une telle opération.

La règle T-R permet de typer la réception sur le canal u. Cette réception nécessite de disposer du droit de lire sur u des valeurs de type T. À partir du moment où la communication a eu lieu, la localité courante connaît l'entité x, que ce soit un canal ou un site. Cette connaissance est utilisée notamment dans la règle T-NameLoc, qui permet ainsi d'extraire le type d'un canal du type d'une localité qui connaît ce canal.

10.1.3 Exemple

La figure 10.2 rappelle les macros de communications diadiques que nous avons proposées pour CDπ et présente une adaptation à Dπ, avec des fonctionnalités similaires.

Ainsi, l'émetteur crée un site buf pour isoler les communications sur c_1 et c_2, transmet le nom buf au récepteur, qui crée un agent pour migrer vers buf, écouter les valeurs émises sur c_1 et c_2, revenir au site originel et les communiquer à son tour sur des canaux privés. Notons que le problème de l'appartenance de c_1 et c_2 se pose encore dans cette version. Dans cet exemple, nous ignorerons ce problème, car il est essentiellement orthogonal à nos préoccupations, et car la sémantique de l'appartenance imposée par le système de types pour Dπ que nous présentons est distincte de celle que nous avons employée jusqu'ici.

Afin de garantir le respect du protocole, il peut être intéressant de vérifier que seul le processus émetteur peut émettre sur les canaux c_1 et c_2 de buf et que le processus récepteur est autorisé à lire sur ces canaux.

Pour ce faire, nous utiliserons un environnement Γ tel que

Version Dπ		
Send v_1, v_2 **on** $c.P$	$\triangleq$	$(\nu buf : A_{buf})(\texttt{go}\ buf.(\overline{c_1}\langle v_1\rangle \mid \overline{c_2}\langle v_2\rangle) \mid$ $\overline{c}\langle buf\rangle.P$ $)$
Receive x_1, x_2 **on** $c@self.Q$	$\triangleq$	$c(buf).($ $(\nu r_1 : C_1, r_2 : C_2)($ $\texttt{go}\ buf.c_1(y_1).c_2(y_2).\texttt{go}\ self.\overline{r_1}\langle y_1\rangle.\overline{r_2}\langle y_2\rangle$ $\mid r_1(x_1).r_2(x_2).Q$ $)$

Version CDπ		
Send v_1, v_2 **on** $c.P$	$\triangleq$	$(\texttt{newloc}\ buf[$ $\overline{c_1}\langle v_1\rangle \mid \overline{c_2}\langle v_2\rangle$ $\mid \texttt{come}\ any$ $] : A_{buf}).($ $\overline{c}\langle buf\rangle.(\urcorner buf).P$ $)$
Receive x_1, x_2 **on** $c@self.Q$	$\triangleq$	$c(buf).($ $(\texttt{newchan}\ r_1 : C_1, r_2 : C_2, mig : C_{mig})($ $\texttt{go}\ buf.c_1(y_1).c_2(y_2).$ $\texttt{go}\ mig@self.\overline{r_1}\langle y_1\rangle.\overline{r_2}\langle y_2\rangle$ $\mid \texttt{come}\ mig.r_1(x_1)r_2(x_2).$ $(\urcorner r_2).(\urcorner r_1).(\urcorner mig).(\urcorner buf).Q$ $)$

FIG. 10.2 – Canal de communications diadiques en CDπ.

$$\begin{cases} \Gamma, buf : A_{buf} & \vdash_{self} P \\ \Gamma, r_1 : C_1, r_2 : C_2, buf : A'_{buf} & \vdash_{self} Q \\ \Gamma, buf : A_{buf} & \vdash_{self} c : res\{rw\langle A'_{buf}\rangle\} \\ A_{buf} = loc\{go, c_1 : rw\langle W_1\rangle, c_2 : rw\langle W_2\rangle\} & \\ A'_{buf} = loc\{go, c_1 : r\langle W_1\rangle, c_2 : \langle W_2\rangle\} & \end{cases}$$

Alors, s'il est possible de prouver que l'émetteur et le récepteur sont typables dans Γ dans le site $self$, nous pouvons en déduire que le récepteur n'a pas reçu de l'émetteur l'autorisation d'écrire sur un canal à l'intérieur de buf.

En fait, si cette conception d'autorisations est naturelle, le système de types ne permet pas de vérifier aussi simplement que le récepteur ne pourra jamais écrire sur c_1, c_2 ou communiquer sur d'autres canaux dans buf. En effet, pour garantir une telle propriété, il faudrait aussi examiner P et Q et s'assurer qu'aucun autre terme ne fournit une telle autorisation au récepteur.

Notons qu'il s'agit d'une tâche qui pourrait être simplifiée soit par des types linéaires – ce qui permettrait de limiter le nombre d'utilisations de buf et donc le nombre de processus qui peuvent migrer vers ce site – soit par l'opérateur ($\urcorner$) – qui interdit, dans la version CDπ des macros, à P et Q de mentionner buf.

10.2 Contrôle des ressources pour la communication

Le système de types que nous venons de présenter est conçu spécialement pour contrôler les communications dans π et Dπ. Dans cette section, plutôt que d'employer une solution ad-hoc, nous allons chercher à exercer un contrôle de nature similaire dans Cπ et CDπ à l'aide de notre système de types, par l'utilisation d'ensembles de ressources adéquats.

Notre objectif, comme précédemment, est ainsi de garantir statiquement que seuls les processus/processus migrants/sites (selon le cas) qui sont autorisés à communiquer sur un canal c communiqueront effectivement sur ce canal.

10.2.1 Comment interdire une communication

Considérons un canal c. Pour que la politique de contrôle de ressources interdise toute forme de réception sur c, dans il suffit d'assurer que $c(x).P$ nécessite des ressources qui ne sont pas présentes dans la réserve. Ainsi, s'il existe un élément $Error$ de l'ensemble de ressources employé, strictement supérieur à la taille de la réserve, il suffit de s'assurer que, pour tout processus P contenant $c(x).\_$, on aura $\Gamma \vdash P : Proc(Error)[\_]$ ou $\Gamma \not\vdash P : \_$.

Considérons alors l'ensemble $\mathcal{B}$ de ressources booléen, défini précédemment. Nous allons nous servir de la valeur tt pour déterminer dans quelles circonstances le canal c est utilisé. De manière à interdire toute utilisation de c, pour la suite de cet exemple, nous supposerons que le niveau de la réserve est ff.

Supposons maintenant que notre politique de contrôle des ressources Γ impose $\Gamma \vdash c : Chan(C, tt, w)$. Alors, si P est typable avec $\Gamma, x : C \vdash P : Proc(\_)[\_]$, nous aurons $\Gamma \vdash c(x).P : Proc(tt)[\_]$. Par suite, dans cette politique, $c(x).P$ nécessite trop de ressources pour la réserve et n'est donc pas ressource-contrôlé.

Par contre, pour peu que Q soit typable dans Γ avec le type $Proc(ff)[\_]$ et que y soit de type C, le processus dual $\overline{c}\langle y\rangle.Q$ restera typable dans Γ avec un type distinct de $Proc(tt)[\_]$ et ce processus sera ressource-contrôlé.

Nous avons donc effectivement interdit statiquement toute réception sur c, alors que toute émission correcte reste typable. Symétriquement, nous pouvons interdire les émissions en fixant $w = tt$. Enfin, si $w = r = tt$, il est impossible de communiquer sur c.

Bien entendu, nous avons interdit les émissions sur tous les canaux tels que $r = tt$ et les réceptions sur tous les canaux tels que $w = tt$. Pour différencier n groupes de canaux, on emploiera un ensemble de ressources $\mathcal{B}^n$, où chaque composante représente un groupe. Nous noterons tt_i la valeur de $\mathcal{B}^n$ dont la $i^{\grave{e}}$ composante est tt et toutes les autres ff. Alors, si nous voulons interdire ou autoriser toutes les communications à la fois, un canal c sera dans le groupe i si $\Gamma \vdash c : Chan(\_, tt_i, tt_i)$ et un site m autorisera les canaux du groupe i à communiquer si l'on peut écrire $\Gamma \vdash m : Site(tt_i \oplus y, \_)$.

10.2.2 Certains agents ne communiquent pas

En l'absence de relation de sous-typage sur les canaux, si nous pouvons passer des autorisations de communiquer sous la forme de types, nous ne sommes pas en mesure d'affaiblir ces types et donc de nous servir de ces autorisations pour limiter l'accès aux entités. Cependant, comme nous venons de le voir,

nous sommes en mesure d'interdire statiquement aux processus de communiquer sur un canal donné. Dans CDπ, nous pouvons tirer avantage du contrôle des migrations par les canaux pour fixer une politique de contrôle au niveau des agents.

Comme nous l'avons vu plus haut, nous sommes capables de garantir qu'un processus n'essaye pas de communiquer sur un canal c, en donnant à ce processus le type $Proc(ff)[\_]$. Or, la règle de typage T-Go que nous employons pour typer les tentatives de migration permet de borner l'effet du processus migrant, donc de vérifier que son type est bien $Proc(ff)[\_]$.

Ainsi, un canal de migration $accept_c$ de type $Migrate(Proc(tt)[\_])$ permettra à des processus de migrer, même s'ils communiquent sur c, tandis qu'un canal de migration $reject_c$ de type $Migrate(Proc(ff)[\_])$ ne sera ouvert qu'à des processus qui ne communiquent pas sur c.

En d'autres termes, si $\Gamma \vdash \overline{c}\langle x\rangle.Q : Proc(tt)[\_]$, le processus

$$\texttt{go}\ accept_c@a.\overline{c}\langle x\rangle.Q$$

sera typable dans Γ avec le type $Proc(ff)[\_]$, car $\overline{c}\langle x\rangle.Q$ tente de migrer le long d'un canal qui accepte des processus susceptibles de communiquer sur c, et car go $accept_c@a.\overline{c}\langle x\rangle.Q$ n'entraînera aucune communication locale sur c. De l'autre côté, le processus come $accept_c.P$ ne sera accepté dans un site m que si le type de m spécifie que les communications sur c sont autorisées, c'est-à-dire si $m : Site(tt, \_)$.

Notons que ce contrôle n'est pas le même que celui dont nous avons cherché à nous inspirer. En effet, le système de types présenté à la section 10.1 permet à un processus non seulement de communiquer sur un canal c mais aussi de passer cette autorisation. De même, si un processus P qui ne peut communiquer sur c peut éventuellement transmettre le nom c, un processus Q qui reçoit c de P ne pourra pas non plus transmettre.

Avec un contrôle tel que celui que fournissent $accept_c/reject_c$, à l'inverse, les autorisations et interdictions sont déterminées indépendamment pour chaque agent. Un processus Q peut donc notamment recevoir le nom c d'un processus P et être autorisé à communiquer sur c même si ce n'était pas le cas pour P.

Ainsi, si nous nous intéressons de nouveau au canal de communications diadiques, dont l'expression est rappelée sur la figure 10.2, nous pouvons employer un environnement Γ tel que

$$\begin{cases} \Gamma \vdash & P : Proc(ff)[\lambda_P] \text{ avec } buf \notin \lambda_Q \\ \Gamma \vdash & v_1 : W_1 \\ \Gamma \vdash & v_2 : W_2 \\ \Gamma \vdash & c_1 : Name(Chan(W_1, tt, tt), ff) \\ \Gamma \vdash & c_2 : Name(Chan(W_2, tt, tt), ff) \\ \Gamma \vdash & c : Name(Chan(A_{buf}, ff, ff), ff) \\ \Gamma \vdash & any : Name(Migrate(Proc(tt)[\emptyset])) \\ \Gamma \vdash & Q : Proc(ff)[\lambda_Q] \text{ avec } \{buf, mig, r_1, r_2\} \cap \lambda_Q = \emptyset \end{cases}$$

où

$$\begin{cases} A_{buf} = & Site(tt, ff) \\ C_1 = & Name(Chan(W_1, ff, ff), ff) \\ C_2 = & Name(Chan(W_2, ff, ff), ff) \\ C_{mig} = & Name(Migrate(Proc(ff)[\emptyset])) \ . \end{cases}$$

Cet environnement spécifie notamment que les canaux c_1 et c_2 ne peuvent être utilisés pour communiquer que dans des sites de type $Site(tt, \_)$, que buf est un tel site et que le canal de migration any accepte la migration de processus qui communiquent sur c_1 ou c_2. Comme $\Gamma \vdash \texttt{Send}\ v_1, v_2\ \texttt{on}\ c.P : Proc(ff)[\lambda_P]$ et $\Gamma \vdash \texttt{Receive}\ x_1, x_2\ \texttt{on}\ c@self.Q : Proc(ff)[\lambda_Q]$, on peut en déduire que c_1 et c_2 ne sont pas utilisés pour communiquer dans le site courant.

Cette propriété peut être intéressante à vérifier car les canaux c_1 et c_2 doivent, justement, n'être employés que dans des localités créées spécifiquement pour l'occasion.

10.2.3 Une relation de sous-typage pour les canaux

Si la possibilité d'interdire à un agent toutes les communications sur un canal est intéressante, nous pouvons nous inspirer des méthodes présentées dans la section 10.1 pour étendre notre système de types afin d'obtenir des résultats plus précis.

L'idée, de nouveau, est de distribuer des autorisations sous la forme de types. Ainsi, si un canal c permet la lecture et l'écriture, on peut espérer qu'il existe aussi une manière de typer c qui interdise soit la lecture, soit l'écriture, voire les deux. Si c'est le cas, plutôt que de transmettre c avec son type le plus général, nous pourrons transmettre c avec son type affaibli – donc avec des autorisations plus restreintes.

Première tentative

Une première approche nécessiterait de modifier légèrement le système de types de manière à permettre à un canal de type $Chan(C, r, w)$ d'être typé aussi $Chan(C, r', w')$ pour certains r' et w'. Ainsi, un processus P, qui recevrait alors ce nom c avec les informations de type $Chan(C, r', w')$, pourrait alors se voir interdit de lire sur c (si $r' = tt$) ou d'écrire sur c (si $w' = tt$).

Pour permettre une telle communication, il suffirait d'ajouter une règle de sous-typage sur les canaux telle que

$$\text{T-ChanSub-Rejected}\ \frac{\Gamma \vdash c : Name(Chan(C, r, w), e)}{\Gamma \vdash c : Name(Chan(C, r', w'), e)}\ r' \succeq r,\ w' \succeq w$$

Malheureusement, nous ne pouvons accepter cette règle et conserver les propriétés intéressantes sur le contrôle des ressources. En effet, si nous revenons dans les entiers, cette règle nous permettrait de typer le processus

$$!c(x).(\nu d : Name(\_, 1))\mathbf{0} \mid !\overline{c}\langle v\rangle.\mathbf{0}$$

avec le type $Proc(0)[\emptyset]$ dans un environnement Γ tel que $\Gamma(c) = c : Chan(\_, 0, 0)$. En d'autres termes, nous déduirons que ce processus ne nécessite aucune ressource alors qu'il peut potentiellement en allouer une infinité.

En d'autres termes, cette relation de sous-typage sur le type des canaux n'est pas appropriée.

$$\text{T-Chan-Sub}\ \frac{\Gamma \vdash c : Name((C, r, w, dr, dw), e)}{\Gamma \vdash c : Name((C, r, w, dr', dw'), e)}\ dr' \succeq dr,\ dw' \succeq dw$$

$$\text{T-Read-Sub}\ \frac{\begin{array}{c}\Gamma \vdash c : Name(Chan(C, r, w), \_) \\ \Gamma, x : C \vdash P : Proc(u_P \oplus w)[\lambda_P]\end{array}}{\Gamma \vdash c(x).P : Proc(v_P)[\lambda_P]}\ x \notin \lambda, v_P \succeq u_P \oplus r \oplus dr$$

$$\text{T-Write-Sub}\ \frac{\begin{array}{c}\Gamma \vdash c : Name(Chan(C, r, w), \_) \\ \Gamma \vdash Q : Proc(u_Q \oplus r)[\lambda_Q] \qquad \Gamma \vdash y : C\end{array}}{\Gamma \vdash \overline{c}\langle y \rangle.Q : Proc(v_Q)[\lambda_Q]}\ v_Q \succeq u_Q \oplus w \oplus dw$$

FIG. 10.3 – Système de types de Cπ avec sous-typage (fragment modifié).

Deuxième tentative

Comme notre tentative de contrôle repose sur r et w et comme il semble faux de modifier directement les valeurs de r et w, nous pouvons introduire des paramètres supplémentaires dr et dw, qui peuvent augmenter par sous-typage et qui s'ajouteront à r et w lors du typage des communications.

Ainsi, le type principal d'un canal c est $Chan(C, r, w, \bot, \bot)$, ce qui signifie que les communications seront traitées exactement comme dans notre système de types précédent. Par sous-typage, on peut transformer cette information en $\Gamma \vdash c : Chan(C, r, w, dr, dw)$ et, si nécessaire, par d'autres opérations de sous-typage, en $\Gamma \vdash c : Chan(C, r, w, dr', dw')$ avec $dr' \succeq dr$ et $dw' \succeq dw$. Les valeurs de dr et dw s'ajoutent alors respectivement au type de $c(x).P$ et à celui de $\overline{c}\langle v \rangle.Q$.

Formellement, nous obtenons les règles présentées sur la figure 10.3.

Ainsi, pour revenir dans $\mathcal{B}$, et en supposant de nouveau que tous les sites sont de capacité ff, un canal c qui peut être utilisé pour communiquer dans n'importe quel site aura pour type principal $Chan(C, ff, ff, ff, ff)$. Le nom c peut être communiqué par un canal d de type

$$Chan(Name(Chan(C, ff, ff, tt, tt), \_), \_, \_, \_, \_)\ .$$

Un processus P qui recevra ce nom en écoutant par $d(x).P'$ ne pourra pas communiquer à l'aide du nom x, pourra à son tour passer le nom x, lié à c, à d'autres processus, qui ne pourront à leur tour pas s'en servir pour communiquer, à moins de l'avoir aussi reçu par un autre biais avec des autorisations moins restrictives.

Cette forme de contrôle est donc très similaire à celle que nous cherchons à approximer, avec la différence majeure que les types des localités ne contiennent aucune information sur les types des canaux disponibles.

Nous fournissons dans l'annexe E.1 le fragment intéressant d'une preuve de Subject Reduction pour ce système de types avec sous-typage. Précisons que cette relation de sous-typage, contrairement au reste des systèmes de types enrichis, fait partie d'un travail actuellement en cours et risque de changer de forme avant d'atteindre une version stable.

10.2.4 Retour à π et SafeDPI

Les méthodes que nous avons présentées utilisent le système de types de CDπ pour garantir que des canaux ne sont pas utilisés pour communiquer par un agent ou dans une localité donnée.

Dans chacun des cas, nous avons considéré que l'encombrement d'un nom était ff, c'est-à-dire $\perp$. En d'autres termes, nous avons considéré que les noms ne nécessitaient pas de ressources selon nos critères habituels. En pratique, cela signifie que nos méthodes n'ont nulle part tiré avantage de la finalisation ou des modifications que nous avons apportées à la congruence structurelle.

Une conséquence de cette remarque est que le fragment de nos techniques applicable à Cπ est aussi applicable au π-calcul. De plus, il nous semble probable que le reste de ces techniques puisse s'appliquer à SafeDPI avec peu d'efforts d'adaptation.

10.3 Typage pseudo-linéaire

Nous avons présenté dans la section 6.2 un système de types linéaires pour le π-calcul qui permet de limiter grossièrement le nombre des émissions ou/et réceptions possibles sur un canal donné. Ce système de types reposait sur des autorisations de lecture et d'écriture proches de celles que nous venons d'exposer, couplées à un aspect linéaire, qui impose à tout processus susceptible d'exercer une de ces autorisations de la consommer.

Une fois de plus, plutôt que d'employer un système de types conçu spécialement pour cet usage, nous allons chercher à employer notre système de types de Cπ et CDπ pour contrôler le nombre de communications sur un canal et la nature de ces communications, par l'utilisation d'ensembles de ressources adéquats.

Notre objectif, comme dans la section 6.2, est de limiter le nombre de lectures ou/et d'écritures sur un canal donné. Notamment, et comme c'est déjà le cas dans Nomadic Pict [86], si nous arrivons à prouver statiquement que, après k communications sur un canal c, ce canal ne servira plus, cette information peut servir à désallouer c sans faire appel aux mécanismes coûteux de garbage-collection générique.

10.3.1 Politique de contrôle

Comme nous l'avons vu, nous sommes en mesure de détecter les émissions et les réceptions sur un canal. Or, si, dans les sections précédentes, nous nous sommes limités à un ensemble $\mathcal{B}$ de ressources, rien ne nous interdit d'employer les mêmes méthodes dans un ensemble plus vaste. Ainsi, de la même manière que nous sommes en mesure de détecter les émissions et les réceptions sur un canal, nous pouvons limiter le nombre de ces émissions et réceptions, que ce soit dans Cπ ou dans CDπ. Pour ce faire, nous pouvons utiliser l'ensemble de ressources $\mathbf{N} \cup \{\infty\} \times \mathbf{N} \cup \{\infty\}$. La première composante nous servira à noter les réceptions, la deuxième les émissions.

Pour ce qui suit, nous noterons $\mathcal{I} = (1, 0)$ et $\mathcal{O} = (0, 1)$, nous imposerons un encombrement nul pour tous les noms et nous réutiliserons le système avec sous-typage introduit dans la section 10.2.3, à la figure 10.3.

Considérons alors un canal c de type $Chan(C, \bot, \bot, \mathcal{I}, \mathcal{O})$. D'après les règles T-READ-SUB et T-WRITE-SUB, chaque lecture sur c de la forme $c(x).P$ impose au processus un surcoût de $\mathcal{I}$ ressources qui s'ajoutent aux ressources nécessaires pour exécuter P, tandis que chaque écriture sur c de la forme $\overline{c}\langle x\rangle.Q$ impose au processus un surcoût de $\mathcal{O}$ ressources qui s'ajoutent aux ressources nécessaires pour exécuter Q.

Par suite, un processus de type $Proc(i.\mathcal{I} \oplus o.\mathcal{O})[\_]$ pourra recevoir au plus i fois et émettre au plus o fois sur le canal c. Comme dans le système de types linéaires, un processus $P|Q$ autorisé à lire i fois et à écrire o fois pourra être décomposé en P et Q autorisés à lire respectivement i_P et i_Q fois et à écrire respectivement o_P et o_Q fois, avec $i_P + i_Q \leq i$ et $o_P + o_Q \leq o$.

Ainsi, pour reprendre l'exemple du canal de communications diadiques, si nous savons déjà que toutes les communications sur c_1 et c_2 ont lieu à l'intérieur de buf, nous pouvons concevoir une politique de contrôle des ressources qui garantisse que c_1 et c_2 ne sont utilisés qu'une seule fois chacun à l'intérieur de buf. Pour simplifier, nous nous concentrerons sur c_1 :

$$\left\{\begin{array}{ll}
\Gamma \vdash & P : Proc(\bot)[\lambda_P] \text{ avec } buf \notin \lambda_Q = \emptyset \\
\Gamma \vdash & v_1 : W_1 \\
\Gamma \vdash & v_2 : W_2 \\
\Gamma \vdash & c_1 : Name(Chan(W_1, \bot, \bot, \mathcal{I}, \mathcal{O}), \bot) \\
\Gamma \vdash & c_2 : Name(Chan(W_2, \bot, \bot, \bot, \bot), \bot) \\
\Gamma \vdash & c : Name(Chan(A_{buf}, \bot, \bot, \bot, \bot), \bot) \\
\Gamma \vdash & any : Name(Migrate(Proc(\mathcal{I})[\emptyset])) \ . \\
\Gamma \vdash & Q : Proc(\bot)[\lambda_Q] \text{ avec } \{buf, mig, r_1, r_2\} \cap \lambda_Q = \emptyset
\end{array}\right.$$

où

$$\left\{\begin{array}{ll}
A_{buf} = & Site(\mathcal{I} \oplus \mathcal{O}, \bot) \\
C_1 = & Name(Chan(W_1, \bot, \bot, \bot, \bot), \bot) \\
C_2 = & Name(Chan(W_2, \bot, \bot, \bot, \bot), \bot) \\
C_{mig} = & Name(Migrate(Proc(\bot)[\emptyset])) \ .
\end{array}\right.$$

Grâce à cette politique de contrôle de ressources, nous pouvons prouver que buf ne peut servir qu'à une seule lecture et une seule écriture sur le canal c_1. Une fois de plus, si nous avions voulu appliquer une politique de contrôle à plusieurs canaux simultanément, il nous aurait suffi d'utiliser un ensemble de ressources produit.

10.3.2 Retour aux sources ?

Notons que cette discipline, proche des types linéaires pour le π-calcul, n'emploie pas ($\neg$) et peut donc, justement, servir à examiner un terme de manière à placer intelligemment et automatiquement des finaliseurs.

Ainsi, intéressons-nous à r_1 dans l'expression suivante :

$$\begin{array}{rcl}
R & \triangleq & c(buf).(\\
 & & \quad (\texttt{newchan}\ r_1 : C_1, r_2 : C_2, mig : C_{mig})(\\
 & & \quad\ \texttt{go}\ buf.c_1(y_1).c_2(y_2).\texttt{go}\ mig@self.\overline{r_1}\langle y_1\rangle.\overline{r_2}\langle y_2\rangle \\
 & & \quad\quad \mid \texttt{come}\ mig.r_1(x_1)r_2(x_2).Q \\
 & & \quad) \ .
\end{array}$$

Cette expression est, en fait, identique à `Receive` x_1, x_2 `on` $c@self.Q$, sans les finalisations. Si nous sommes capables de typer R dans un environnement Γ tel que

$$\begin{cases} \Gamma \vdash & c_1 : Name(Chan(W_1, \bot, \bot, \bot, \bot), \bot) \\ \Gamma \vdash & c_2 : Name(Chan(W_2, \bot, \bot, \bot, \bot), \bot) \\ \Gamma \vdash & c : Name(Chan(A_{buf}, \bot, \bot, \bot, \bot), \bot) \\ \Gamma \vdash & Q : Proc(\bot)[\lambda_Q] \text{ avec } \{buf, mig, r_1, r_2\} \cap \lambda_Q = \emptyset \\ \Gamma \vdash & any : Name(Migrate(Proc(\bot)[\emptyset])) \ . \end{cases}$$

où

$$\begin{cases} A_{buf} = & Site(\bot, \bot) \\ C_1 = & Name(Chan(W_1, \bot, \bot, \mathcal{O}, \mathcal{I}), \bot) \\ C_2 = & Name(Chan(W_2, \bot, \bot, \bot, \bot), \bot) \\ C_{mig} = & Migrate(Proc(\mathcal{O})[\emptyset]) \ , \end{cases}$$

alors nous pouvons déduire qu'aucune communication sur r_1 n'a lieu dans Q. Cela ne signifie pas encore que nous pouvons désallouer r_1 au moment où Q commence à s'exécuter, car r_1 peut encore être utilisé ailleurs. Cependant, si le processus complet est ressource-contrôlé avec une réserve de taille $\mathcal{I} \oplus \mathcal{O}$, nous pouvons effectivement déterminer que r_1 n'est utilisé, au plus, qu'une seule fois. Comme nous pouvons déterminer syntaxiquement à quel endroit r_1 est utilisé, le ramasse-miettes est autorisé à désallouer r_1 immédiatement après la communication. Nous pouvons aussi insérer $(\neg r_1)$ n'importe où après cette communication, de manière à statiquement pouvoir prendre en compte cette désallocation. Par exemple, nous pouvons remplacer Q par $(\neg r_1).Q$, ce qui permet à Q de réutiliser les ressources allouées à r_1.

En fait, si la méthode fonctionne dans Cπ, elle n'est ni suffisante, ni même exacte dans le calcul CDπ pris tel quel, à moins de modifier notre système de types. L'erreur qui peut apparaître est liée aux opérations qui utilisent un nom de canal de communication mais qui ne "coûtent" aucune ressource : dans `if` $a = b$ `then` P `else` Q, le système de types ne nous permet pas d'observer a et b ni donc de prendre en compte le fait que a et b "vivent" encore. Pour rendre la méthode valable dans CDπ, il faudrait que nous soyons en mesure de détecter les utilisations d'un nom.

Pour ce faire, nous pouvons doter chaque nom d'un coût supplémentaire i, qui apparaît lors de chaque comparaison de ce nom. Nous aurons alors une règle de typage de la comparaison de la forme suivante :

$$\text{T-IfCost} \ \frac{\Gamma \vdash P : Proc(t)[\lambda] \qquad \Gamma \vdash Q : Proc(t)[\lambda] \qquad \Gamma \vdash a : S \qquad \Gamma \vdash b : S \qquad S = Name(T, e, f)}{\Gamma \vdash \texttt{if } a = b \texttt{ then } P \texttt{ else } Q : Proc(u)[\lambda]} \ u \succeq f \oplus t \ .$$

De même, la technique n'est pas applicable aux canaux de migration ou de sites, car il n'est pas possible de détecter leurs utilisations. Pour étendre notre approche et lui permettre de gérer tous les cas de figure, nous pourrions donner un coût aux opérations `go` et `come`, pour l'instant invisibles pour le système de types.

Nous ne détaillerons pas cette transformation du système de types qui, tout comme le sous-typage, est actuellement en cours et sujette à modifications.

10.3.3 Linéarité et équilibrage

Comme nous l'avions remarqué, le système de types linéaires pour le π-calcul ne permet pas de contrôler les processus répliqués. C'est ce pour quoi a été conçu le mécanisme d'équilibrage des coûts de communication.

Considérons ainsi un service défini par $A \triangleq !c(x).\overline{d}\langle x\rangle$, dans lequel nous cherchons à limiter le nombre de communications sur d. Le système de types linéaires considère que d peut être utilisé un nombre quelconque de fois, et ce, même si aucun processus n'émet sur c ou si aucun processus ne lit sur d. À l'inverse, à l'aide de notre système de types, le nombre de communications sur d, par exemple, sera directement lié au nombre d'émissions sur c. Ainsi, nous avons :

Typage de $\overline{d}\langle x\rangle$		
Si	$d : Name(Chan(T_x, \bot, \bot, \mathcal{I}, \mathcal{O}), \_)$	
$\Rightarrow \Gamma, x : T_x \vdash \overline{d}\langle x\rangle$	$Proc(\mathcal{O})[\emptyset]$	Par T-Write
Typage de $c(x).\overline{d}\langle x\rangle$		
Si	$c : Name(Chan(T_x, \bot, \mathcal{O}, \bot, \bot), \_)$	
$\Gamma, x : T_x \vdash \overline{d}\langle x\rangle$	$Proc(\mathcal{I})[\emptyset]$	Cf. plus haut
$\Rightarrow \Gamma \vdash c(x).\overline{d}\langle x\rangle$	$Proc(\bot)[\emptyset]$	Par T-Read
Typage de $!c(x).\overline{d}\langle x\rangle$		
$\Gamma \vdash c(x).\overline{d}\langle x\rangle$	$Proc(\bot)[\emptyset]$	Cf. plus haut
$\Rightarrow \Gamma \vdash !c(x).\overline{d}\langle x\rangle$	$Proc(\bot)[\emptyset]$	Par T-Repl

○

alors que le type d'une émission $\overline{c}\langle v\rangle.\mathbf{0}$ est $Proc(\mathcal{O})[\_]$.

Notons que, de la même manière, en combinant linéarité et sous-typage, nous pouvons communiquer un nom c tout en bornant son nombre d'utilisations en lecture ou/et en écriture.

10.4 Discussion

Le système de types pour π et Dπ que nous avons présenté permet de contrôler statiquement l'accès aux canaux et les migrations dans Dπ, à l'aide de mécanismes simples et intuitifs. Le principe de ce système est de distribuer statiquement aux processus des autorisations d'accéder à des entités, sous la forme de différents types pour ces entités. Un tel contrôle est indispensable, car il permet d'isoler les programmes mal écrits comme les chevaux de Troie, en leur interdisant d'intercepter des messages locaux ou d'émettre sur des canaux auxquels ils ne devraient pas avoir accès.

De plus, procéder à un tel contrôle par le système de types – qui pourra être vérifié par exemple à l'aide de code porteur de preuves [64] – semble plus simple et plus intuitif que de tenter de contrôler les propriétés de communication le long de canaux à l'aide de mécanismes dynamiques comparables aux cocapacités.

Il s'avère que les techniques que nous avons introduites précédemment dans cet exposé pour le contrôle des ressources dans Cπ et CDπ peuvent notamment permettre de contrôler les accès aux canaux, de plusieurs manières différentes. Nos méthodes consistent à associer un coût à chaque lecture et à chaque écriture sur un canal puis à vérifier les bornes supérieures sur le coût des processus. Cela nous permet déjà de vérifier qu'un processus donné ne peut absolument

pas communiquer sur un canal donné. En introduisant une relation de sous-typage en termes de surcoûts de communication, nous pouvons aussi affaiblir les autorisations d'accéder aux entités et transmettre ces autorisations comme dans le système de types pour π et Dπ.

Même si nous n'avons pas cherché à comparer la puissance des divers systèmes, car les politiques de contrôle et les langages employés ne sont pas exactement les mêmes, nous méthodes ont l'avantage d'être plus génériques, ce que nous illustrons en prouvant des problèmes de quasi-linéarité sur les termes. Il nous semble les technique employées, qui fonctionnent tout aussi bien dans le π-calcul que dans Cπ, et qui nécessiterait quelques modifications pour être appliquées à CDπ, permettent notamment de prouver des propriétés plus riches que celles du système de types linéaire pour le π-calcul, et d'examiner des termes plus complexes qu'avec ce système de types.

Chapitre 11

Contrôle de mobilité dans les ambients

Dans le cadre des calculs des ambients, comme nous en avions déjà discuté, le concept naturel d'entité "à contrôler" recouvre celui de localité. De même, les seuls agents susceptibles d'accéder à une entité sont des ambients : ainsi, on pourra considérer, par exemple, qu'un agent a accède à une entité b lorsque l'ambient a est susceptible de devenir un sous-ambient direct de b. On pourrait, bien entendu, aussi employer un point de vue dual, dans lequel a accède à b si a peut contenir b. Dans tous les cas, il s'agira de décider quels sont les relations père-fils autorisés entre les ambients.

Le calcul des Mobile Ambients, tel quel, ne propose qu'un contrôle extrêmement limité de la mobilité : à partir du moment où le nom d'un ambient b est connu, aucune autorisation n'est nécessaire à un autre ambient pour entrer dans b. Plusieurs extensions des MA, telles que les Safe Ambients [58], ROAM [36], les Controlled Ambients [80] ou les NBA [13], introduisent un mécanisme de cocapacités tel que celui que nous avons vu à la section 3.3, et qui permet d'autoriser ou de refuser certains mouvements ou certaines ouvertures.

Ces cocapacités, si elles permettent de contrôler efficacement la mobilité et donc, notamment, de supprimer de nombreuses erreurs de programmation, imposent une discipline parfois trop stricte et parfois difficile à vérifier. Ainsi, dans les CA, pour qu'un ambient a autorise un ambient dont le nom est initialement inconnu à devenir un de ses sous-ambients, il est nécessaire de concevoir un mini-protocole dédié. Toute implantation de ce protocole présentera un certain degré complexité puisqu'elle nécessitera l'ouverture d'au moins un ambient, ce qui n'est pas une opération anodine.

Plusieurs travaux [17, 59], à l'inverse, cherchent à contrôler la mobilité à l'aide de types. Nous commençons par présenter ici un aperçu d'une méthode [59] qui ne nécessite aucune modification au langage des Mobile Ambients : les *Types Dépendants pour le Calcul des Ambients* (ou DTMA).

Nous comparerons ensuite cette solution avec certains résultats que nous pouvons obtenir dans les Controlled Ambients, à l'aide d'une instanciation de notre système de types.

11.1 Des types dépendants pour les Mobile Ambients

Plusieurs systèmes de types ont été présentés pour contrôler statiquement les accès dans les Mobile Ambients. Nous présentons ici brièvement le mécanisme des DTMA [59]. L'idée consiste à spécifier dans le nom de chaque ambients ses liens autorisés avec les autres ambients : quels sont les ambients qu'il peut contenir, dans lesquels il peut entrer, dont il peut communiquer le nom...

11.1.1 Spécification d'une politique

Le langage des DTMA permet, pour chaque ambient a, de spécifier quels sont les noms des ambients qui peuvent entrer dans a et quels sont les ambients autorisés à contenir a – afin d'éviter toute ambiguïté, le système impose que cette relation soit toujours symétrique.

Ainsi, si l'ambient $fclose$ a pour type $amb[mob[\{system\},\{file,socket\}],\_]$, seuls des ambients nommés $file$ ou $socket$ peuvent entrer dans $fclose$ et celui-ci ne peut à son tour apparaître que comme enfant de $system$.

Le typage dépendant permet notamment de formuler des politiques qui font intervenir des noms créés dynamiquement ou liés par une communication et de spécifier des dépendances cycliques entre les types.

Les dépendances cycliques sont fondamentales car elles permettent de déclarer deux nouveaux noms a et b et de spécifier que a peut contenir b – mais aussi que b peut entrer dans a.

Ainsi, le processus

$$\begin{array}{l}(\nu\, file : amb[mob[\emptyset,\emptyset],C_{file}])\\ \quad (\nu\, fclose : amb[mob[\{system\},\{file,socket\}],C_{file}]) \quad P\end{array}$$

déclare deux nouveaux noms $file$ et $fclose$. Comme le nom $file$ est déclaré avant le nom $fclose$, $file$ ne peut référencer $fclose$. Mais comme $fclose$ peut contenir $file$, le système de types impose que $file$ puisse entrer dans $fclose$. Le type des deux noms est alors, et contrairement à ce que la syntaxe pouvait laisser supposer,

$$\left\{\begin{array}{lcl} file & : & amb[mob[\{fclose\},\emptyset]] \\ fclose & : & amb[\{system\},mob[\{file,socket\}]]\,. \end{array}\right.$$

Notons que les cycles peuvent être moins bénins. Ainsi, on peut avoir un ambient a qui peut contenir un ambient b qui, à son tour, peut contenir un ambient a.

De même, il est possible d'utiliser des noms liés dynamiquement dans une politique. Ainsi, le terme

$$\begin{array}{l}!(file)(\nu closer : amb[\\ \qquad\qquad\qquad\qquad mob[\{system\},\{file,socket\}],\\ \qquad\qquad\qquad\qquad com[\{system,linker\},\{file\}]\\ ])P\end{array}$$

crée un nom *closer* qui permet de désigner des ambients qui peuvent apparaître comme enfants de $system$, qui peuvent contenir les ambients appelés *socket* ou

$$\textsc{Amb}\ \frac{\Gamma \vdash_a^{\Xi;\Theta} P \qquad \Gamma \vdash_a^{\Xi;\Theta} a : amb[mob[\{b\} \cup \_, \_], \_]}{\Gamma \vdash_b^{\Xi;\Theta} a\,[P]}$$

$$\textsc{Output}\ \frac{\Gamma \vdash^{\Theta} V : A = amb[\_, com[\_, \{a\} \cup \_]]}{\Gamma \vdash_a^{\emptyset;\Theta} \langle V \rangle}$$

FIG. 11.1 – Fragment du système de types dépendants pour les Mobile Ambients.

$file$ – lié dynamiquement – dont le nom peut être communiqué à l'intérieur des ambients *system* et *linker* et qui peuvent à leur tour contenir des messages mentionnant le nom $file$.

11.1.2 Vérification de types

La figure 11.1 présente un bref fragment du système de types dépendants pour les Mobile Ambients. Sans entrer dans la signification de Θ et Ξ dans $\vdash_m^{\Xi;\Theta}$, précisons que $\Gamma \vdash_m^{\Xi;\Theta} P$ signifie que P est typable sous les hypothèses Γ dans l'ambient m. Ainsi, la règle AMB signifie notamment que l'ambient $a\,[P]$ ne peut être typé que si P peut être typé dans a et que $a\,[P]$ ne peut apparaître dans b que si le type de a spécifie que b peut être un parent de a.

De la même manière, OUTPUT spécifie qu'un message V ne peut être émis dans un ambient a que si V est un nom d'ambient et si le type de V autorise V à être communiqué à l'intérieur d'un ambient nommé a.

En raison de la complexité du système de types, nous n'entrerons pas plus dans les détails.

11.1.3 Exemple

La figure 11.2 présente une modélisation élémentaire d'un réseau. Le réseau peut contenir des messages et des ordinateurs, les ordinateurs peuvent contenir des messages et des applications, les applications peuvent contenir des messages et les messages peuvent eux-mêmes contenir d'autres messages.

Pour vérifier que ces spécifications sont respectées, nous écrirons donc les types suivants (en l'absence de communications) :

$$\begin{cases} \Gamma \vdash & Net : Amb(mob(\emptyset, \{Pc, Msg\}), com(\emptyset, \emptyset)) \\ \Gamma \vdash & Pc : Amb(mob(\{Net\}, \{App, Msg\}), com(\emptyset, \emptyset)) \\ \Gamma \vdash & App : Amb(mob(\{Pc\}, \{Msg\}), com(\emptyset, \emptyset)) \\ \Gamma \vdash & Msg : Amb(mob(\{Net, Pc, App, Msg\}, \{Msg\}), com(\emptyset, \emptyset)) \end{cases}$$

Nous n'entrerons pas dans le détail d'une vérification de types.

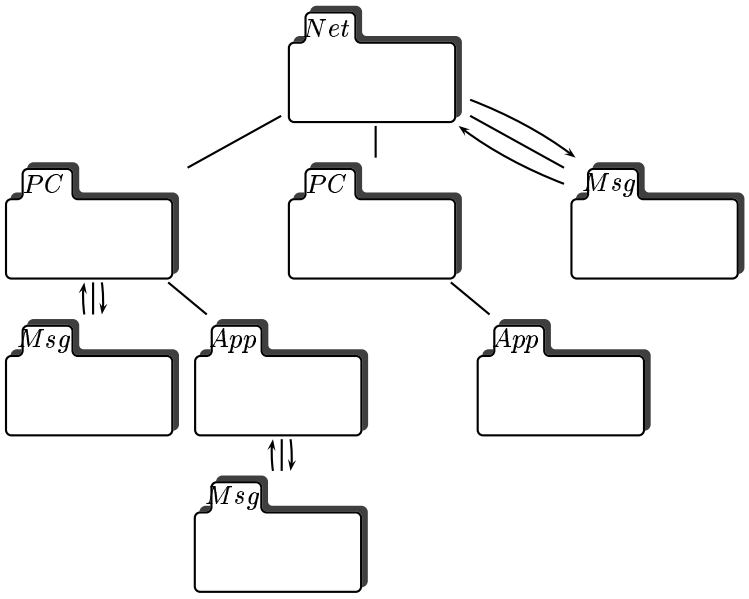


FIG. 11.2 – Modélisation élémentaire d'un réseau.

11.2 Instanciation du contrôle pour la communication

Plutôt que d'employer un système de types conçu spécialement pour cet usage, comme dans la section précédente, nous allons chercher à contrôler les déplacements dans les CA à l'aide de notre système de types, par l'utilisation d'ensembles de ressources adéquats.

Notons que les techniques peuvent aisément s'appliquer dans les Seals ou les Kells.

11.2.1 Déplacements

De la même manière que nous pouvons employer les ensembles de ressources booléens pour garantir qu'un canal n'est pas utilisé pour communiquer dans un processus Cπ, nous pouvons aisément prouver qu'un ambient ne contiendra jamais un sous-ambient donné.

Ainsi, si nous désirons caractériser à l'aide des ressources les ambients qui peuvent contenir un sous-ambient a, il suffit de s'assurer que a – et seul a – est de type $Amb(\_, tt)[\_]$. Seuls les ambients de type $Amb(tt, \_)[\_]$ sont alors autorisés à contenir a, en un ou plusieurs exemplaires. Il est, bien entendu, possible de contrôler aussi le nombre de a, comme nous l'avons fait dans $\mathbf{N} \cup \{\infty\}$.

Comme nous pouvons le constater, notre système de types n'est pas en mesure de vérifier dans quelles conditions un processus contenu dans un ambient a peut présenter des capacités de déplacement, ce que permettent certains autres

travaux [18]. Cependant, grâce aux cocapacités, nous pouvons aisément caractériser les destinations qui autorisent cet ambient a à rentrer. Si aucun autre ambient, hors celui qui le contenait initialement, ne peut accueillir a, a ne bougera pas.

Ainsi, si nous cherchons à modéliser un réseau tel que celui de la figure 11.2, dans lequel des messages se déplacent à l'intérieur d'applications, d'ordinateurs et du réseau, on emploiera un ensemble de niveaux de ressources $\mathcal{B}^4$. De ce quadruplet, le premier élément désignera les messages, le deuxième les applications, le troisième les ordinateurs et le quatrième les réseaux. Nous noterons donc $\mathcal{M} \triangleq (tt, ff, ff, ff)$ pour les messages, $\mathcal{A} \triangleq (ff, tt, ff, ff)$ pour les applications, $\mathcal{C} \triangleq (ff, ff, tt, ff)$ pour les ordinateurs et $\mathcal{N} \triangleq (ff, ff, ff, tt)$ pour le réseau.

Un message entre dans la catégorie messages et peut contenir d'autres messages. Son type sera donc $Amb(\mathcal{M}, \mathcal{M})[\_]$. De même, une application peut contenir des messages et aura le type $Amb(\mathcal{A}, \mathcal{M})[\_]$, un ordinateur peut contenir aussi bien des messages que des applications, et sera de type $Amb(\mathcal{M} \oplus \mathcal{A}, \mathcal{C})[\_]$ et le type du réseau lui-même sera $Amb(\mathcal{C} \oplus \mathcal{M}, \mathcal{N})[\_]$, puisqu'un réseau peut contenir des messages et des ordinateurs. Si processus est correctement typé, on aura donc la garantie que ces contraintes sont respectées : un ordinateur ne contiendra pas de réseau, un message ne contiendra pas d'applications...

Notons que, à l'aide de cette méthode, plus que d'autoriser ou d'interdire aux ambients à se déplacer, nous autorisons ou nous interdisons les cocapacités, qui sont elles-mêmes des formes d'autorisation ou d'interdiction. Nous nous bornons donc essentiellement à confirmer statiquement un contrôle dynamique.

Un autre défaut de cette application de notre système de types est que, si nous souhaitons ajouter une nouvelle catégorie d'entités à la modélisation, nous devrons utiliser des quintuplets au lieu de quadruplets. Plus, généralement, chaque nouvelle catégorie d'entités modifie l'ensemble de ressources sur lequel est instancié le système de types, ce qui complique l'utilisation de nos techniques. De plus, nous ne disposons pas de méthodes intelligentes pour regrouper automatiquement les ambients en catégories dans lesquelles nous pouvons prouver le respect de politiques utiles.

Par contre, comme nous l'avons vu, notre système de types peut servir à prouver des propriétés plus complexes, notamment le nombre de sous-ambients d'une certaine catégorie présents dans un ambient donné.

11.2.2 Retour aux Ambients ?

Comme nous l'avions remarqué dans le cadre de nos systèmes de types pour CDπ, nous pourrions donner un coût à certaines constructions jusqu'à présent considérées comme gratuites (i.e. `in`, `out` et `open`), de manière à être en mesure de les observer depuis le système de types. Nous pourrions alors contrôler les déplacements du point de vue des capacités, sans avoir besoin d'examiner les cocapacités.

Grâce à cela, de même que pour ($\neg$) dans Cπ ou CDπ, nous pourrions espérer utiliser ces informations pour placer automatiquement les cocapacités.

Nous n'avons pas jugé opportun de poursuivre cette voie car le langage des Controlled Ambients ne nous semble pas suffisamment pertinent pour que nous nous concentrions sur la possibilité de "compiler" les Mobile Ambients en

$$\textsc{T-BetterOpen}\ \frac{\Gamma \vdash m : Amb(s,e,q,r)[T] \quad \Gamma \vdash P : Proc(t_P)[T]}{\Gamma \vdash \texttt{open}\ m.P : Proc(v_P)[T]}\ v_P \oplus e \succeq t_P \oplus q \oplus r$$

$$\textsc{T-BetterCoOpen}\ \frac{\Gamma \vdash m : Amb(s,e,q,r)[T] \quad \Gamma \vdash Q : Proc(q)[T] \quad \forall w \forall h, w \succeq v_Q \wedge w \oplus h \preceq s \Rightarrow h \preceq r}{\Gamma \vdash \overline{\texttt{open}}\ m.Q : Proc(v_Q)[T]}$$

FIG. 11.3 – Nouveau système de types raffiné de CA (fragment concernant l'ouverture).

Controlled Ambients. De plus, il nous semble que ces transformations n'auraient aucun intérêt pour les mécanismes de synchronisation qui apparaissent dans les langages à mobilité de sites qui nous intéressent plus (i.e. les migrations le long de canaux dans les Seals et les continuations dans les Kells). En effet, ces constructions sont à la fois bien plus profondément ancrées dans le langage et bien plus transparentes, si bien que les ajouter automatiquement n'aurait aucun sens.

11.3 Contrôle d'ouverture

De même que nous pouvons contrôler les déplacements des ambients, nous pouvons avoir besoin d'interdire l'ouverture de certains ambients particuliers ou de n'autoriser que certains autres ambients à les ouvrir. Un tel contrôle prolonge le concept d'ambients verrouillés ou déverrouillés, introduit avec les premiers systèmes de types pour les Mobile Ambients [18].

Tout comme nous pouvons imaginer observer les déplacements soit en examinant les capacités, soit en examinant les cocapacités, nous pouvons étudier la dissolution d'un ambient soit du point de vue du processus ouvreur, qui exerce la capacité $\texttt{open}\ m$, soit du point de vue de l'ambient ouvert et de la cocapacité $\overline{\texttt{open}}\ m$.

Nous emploierons le système de types raffiné pour l'ouverture, rappelé sur la figure 11.3 et nous reprendrons les notations de cette figure. De la même manière que dans la section précédente, nous utiliserons $\mathcal{B}$ comme ensemble de niveaux de ressources.

Afin de mettre en évidence la possibilité d'ouvrir un ambient m, nous allons chercher une politique de contrôle de ressources qui permettra de garantir que les processus de type $Proc(ff)[\_]$ ne contiennent pas la capacité $\texttt{open}\ m$.

Afin d'éviter toute interférence, nous imposerons de nouveau qu'aucun ambient ne doit être d'encombrement tt. Alors, si $\Gamma(m) = Amb(\_, ff, tt, \_)[\_]$ et si $\texttt{open}\ m.P$ est typable dans Γ, on aura $\Gamma \vdash \texttt{open}\ m.P : Proc(tt)[\_]$. Par conséquent, dans un terme bien typé, un ambient de capacité ff ne peut pas contenir $\texttt{open}\ m.P$.

Par suite, si seuls les ambients autorisés à ouvrir m ont une taille tt, nous disposons d'un contrôle presque complet de l'ouverture de m. Presque complet, seulement, car nous ne sommes pas en mesure d'interdire ainsi au processus de

plus haut niveau d'ouvrir m. Pour ce faire, il est nécessaire, en plus, de vérifier que ce processus est typable avec le type $Proc(ff)[T]$.

Notons que l'approche duale, qui consiste à interdire les $\overline{\texttt{open}}$, ne fonctionne pas dans l'ensemble $\mathcal{B}$, pour des raisons techniques que nous ne détaillerons pas.

11.4 Discussion

Le système de types dépendants pour les Mobile Ambients que nous avons présenté permet de contrôler statiquement les mouvements d'ambients, y compris dans le cas complexe de noms d'ambients générés ou/et liés au cours de l'exécution, d'une manière souvent plus fine que ce que semblent permettre les mécanismes de cocapacités. Cette puissance a malheureusement plusieurs coûts. En premier lieu, la complexité des types et des preuves et l'absence d'inférence rendent ce système statique particulièrement complexe, ce qui est accentué par l'absence de résultats de décidabilité sur la vérification des types. De plus, l'analyse repose sur une notion de "contexte abstrait" Θ, qui modélise l'environnement du processus examiné, mais cette notion n'est pas compositionnelle, si bien que $\Gamma \vdash_a P$ et $\Gamma \vdash_a Q$ n'impliquent pas $\Gamma \vdash_a P|Q$. Cette limitation, surprenante dans le contexte de la vérification de déplacements, peut s'avérer gênante.

Ici aussi, les mécanismes que nous avons introduits précédemment dans cet exposé pour le contrôle des ressources dans les Controlled Ambients, les Seals ou les Kells, peuvent notamment permettre de contrôler les mouvements d'ambients, à partir de principes très différents. Notre méthode, comme précédemment, consiste à associer un coût à chaque déplacement d'un ambient d'une catégorie donnée – catégories qui n'interviennent qu'au moment du typage – puis à vérifier les bornes supérieures sur le coût des processus. Ainsi, comme chaque ambient impose aux processus qu'il contient une borne maximale, on peut aisément vérifier qu'un ambient ne contiendra jamais un sous-ambient donné.

Nous n'avons pas cherché à comparer formellement la puissance des deux systèmes. Si, nos techniques et celles des DTMA semblent offrir une lisibilité similaire, il semble à peu près certain que les DTMA permettent un contrôle beaucoup plus puissant. En particulier, et malgré la modification que nous avons proposée pour notre système de types et qui peut permettre d'étendre le contrôle aux capacités, nous ne pouvons prendre en compte la communication de capacités et nous ne pouvons nous passer de $\overline{\texttt{open}}$. De plus, aucun de nos outils ne semble être en mesure de gérer l'aspect dépendant des types, bien que nous ne soyons pas encore en mesure de déterminer à quel point cela affecte la puissance de notre système de types ou son utilité pour des exemples réalistes.

Notons cependant que les mécanismes de cocapacités conservent tout de même au moins une fonctionnalité absente de ce système de types, puisqu'ils permettent de spécifier des relations chronologiques sur les mouvements – un aspect important pour notre travail, et sur lequel nous reviendrons brièvement dans le prochain chapitre.

Chapitre 12

Bilan

Au début de cet exposé, nous avons défini une notion de ressources/réserves dans laquelle certaines opérations peuvent puiser pour allouer des entités et que d'autres opérations peuvent restaurer. Nous avons présenté plusieurs méthodes dynamiques et statiques pour contrôler l'allocation et la désallocation des ressources, afin d'éviter les cas où une allocation devrait puiser dans une réserve déjà vide. Alors que, dans les parties précédentes, nous nous étions contentés de considérer un ensemble de ressources caractérisé uniquement par sa taille, nous avons cherché, dans cette partie, à étendre nos méthodes pour obtenir un contrôle de propriétés plus qualitatives.

D'autres contrôles des ressources Nous avons introduit une version étendue de nos systèmes de types, dans laquelle nous ne nous restreignons plus à des réserves de ressources entières. Cette généralisation nous permet ainsi d'exprimer aisément des garanties de sécurité ou de rassembler plusieurs propriétés en une seule, afin d'avoir à établir un seul certificat pour toutes ces propriétés.

Nous avons ensuite présenté brièvement quelques autres points de vue sur la notion de contrôle des ressources, construits autour des accès plutôt qu'autour des allocations et des désallocations, puis nous avons cherché à déterminer si nous étions en mesure de caractériser ces accès, justement, à partir de notre notion de ressources, des allocations et des désallocations. Il s'est avéré que, si ces notions semblaient a priori sans rapport avec tout ce que nous avions établi dans le reste de cet exposé, nous pouvions approximer les contrôles en question à l'aide de nos méthodes.

Limitations des contrôles Même si nos techniques sont suffisamment puissantes pour permettre d'approximer des contrôles tels que les autorisations de communication de Dπ, les types linéaires du π-calcul ou le typage dépendant de la mobilité dans les Mobile Ambients, nos système de types montrent assez vite leurs limites.

Ainsi, comme nous l'avons remarqué, dans l'état actuel des choses, il est impossible de vérifier qu'un nom n'apparaîtra plus jamais dans CDπ, puisque certaines opérations sont "invisibles" du point de vue du typage. De même, nous ne pouvons typer les Controlled Ambients que du point de vue des cocapacités, sans prendre en compte les capacités, ce qui nous interdit d'appliquer notre analyse aux Mobile Ambients, par exemple.

Comme nous en avons brièvement discuté, nous pouvons parfaitement compliquer les systèmes de types de manière à rendre d'autres opérations visibles. Par exemple, nous pouvons imaginer de donner à un ambient a un co-encombrement $\overline{e}$ tel que $\texttt{in}\ a.P$ nécessite $\overline{e}$ ressources de plus que P pour s'exécuter. Cette technique est simple et peut être appliquée, notamment, aux capacités, aux migrations de CDπ ou aux comparaisons de CDπ. Dans un système de types ainsi transformé, nous pourrons, de la même manière que nous l'avons fait pour les communications ou les cocapacités, garantir l'absence de ces opérations dans certains sites ou dans certains agents.

Pour étendre encore le contrôle Il est aussi possible d'étendre le contrôle dont nous disposons déjà sur les opérations visibles. En guise d'expérience, nous avons développé dans le cadre des Controlled Ambients une version plus complexe, dans laquelle nous pouvons aussi spécifier des politiques de contrôle sur le nombre de déplacements ou l'ordre d'opérations. Ainsi, pour un ensemble de ressources bien choisi, le type d'un processus $\overline{\texttt{in}}^{\downarrow}\ a.\overline{\texttt{out}}^{\downarrow}\ a.\overline{\texttt{in}}^{\downarrow}\ b.\overline{\texttt{out}}^{\downarrow}\ b.\overline{\texttt{in}}^{\downarrow}\ a.\overline{\texttt{out}}^{\downarrow}\ a$ peut s'écrire $Proc(a \cdot b \cdot a)$ et le type d'un ambient qui n'accepte que l'entrée puis la sortie des sous-ambients a, b et a dans cet ordre peut s'écrire $Amb(a \cdot b \cdot a, \_, \_, \_)[\_]$. Plus généralement, avec des ensembles adéquats, nous sommes en mesure de donner un type proche des labels du système présenté dans la section 2.6 dans le cas séquentiel – sans, toutefois, pouvoir typer des comportements "aussi infinis". Ainsi, sans atteindre la richesse de système de types génériques [46], nos méthodes permettent de vérifier le respect de certains protocoles complexes.

Nous n'avons pas présenté cet aspect de notre travail car il est plus compliqué et moins esthétique, car nous ne sommes pas encore en mesure d'expliquer tous nos choix et car nous n'avons pas encore cherché à l'adapter aux autres calculs – notons tout de même que, dans ce système modifié, la condition sur T-BetterCoOpen se simplifie. Sans entrer dans les détails, nous dotons les entités de plusieurs niveaux d'encombrement distincts, qui entrent en jeu dans les différentes règles de typage. Ainsi, un ambient a dont nous avons jusqu'à présent considéré qu'il était d'encombrement e pourra être d'encombrement e_1 *en entrée* (ce qui signifie que $\overline{\texttt{in}}^{\downarrow}\ a$ nécessitera e_1 ressources) et d'encombrement e_2 *en sortie* (ce qui signifie que $\overline{\texttt{out}}^{\downarrow}\ a$ libérera e_2 ressources) dès que $e_1 \succeq e_2$. De bonnes instances de e_1, e_2... permettent alors de conserver dans le type d'un processus des traces d'états sur les opérations qui ont effectuées, traces qui peuvent alors être acceptées ou rejetées par le site parent, à l'image du système de types pour le λ-calcul présenté à la section 2.6.

En d'autres termes, il nous semble que nos systèmes de types pour le contrôle des ressources n'ont pas encore révélé toute leur puissance.

Inférence Comme nous pouvons le noter, même dans ces systèmes enrichis, la majeure partie des propriétés à vérifier lors de l'application des règles peut s'exprimer sous la forme d'inéquations linéaires simples. Ainsi, les preuves dont nous avons eu besoin pour la Subject Reduction, présentées en annexe, sont purement syntaxiques et totalement automatisables.

En pratique, cela signifie que, pour des ensembles de ressources simples – c'est-à-dire dès qu'une certaine classe simple de systèmes d'inéquations linéaires à coefficients entiers est soluble automatiquement – on peut espérer inférer au-

tomatiquement au moins une partie des types. C'est le cas, notamment, pour tous les ensembles que nous avons utilisés dans cette partie.

Cinquième partie

Conclusions

Chapitre 13

Bilan

L'objectif de notre démarche était d'étudier le problème de l'utilisation et de la réutilisation des ressources dans des systèmes parallèles et distribués, en présence de mobilité. Pour ce faire, nous nous sommes intéressés à plusieurs algèbres de processus, qui nous semblaient constituer des formalismes clairs et appropriés. Dans chaque cas, nous avons mis en évidence une notion de ressources ainsi que des mécanismes du langage qui permettent de contrôler l'utilisation de ces ressources, c'est-à-dire de développer des protocoles et des programmes conscients des ressources.

À l'aide de ces mécanismes, nous avons développé des systèmes de types qui garantissent que jamais un terme n'aura besoin de puiser dans une réserve de ressources vide. De plus, nous avons mis en évidence certaines propriétés complexes qui peuvent être certifiées par une généralisation de notre système de types.

13.1 Mécanismes de contrôle

Nature des mécanismes Les mécanismes de contrôle que nous avons observés peuvent être regroupés en plusieurs classes. Certaines opérations, telles que (`newloc`) ou $c\!\leftarrow\!\{\cdots\}$ (ou encore $\triangleright$ dans BoCa), provoquent des allocations, tandis que certaines autres opérations, telles que `open`, $\overline{c}\!\leftarrow\!\{\_\}$ ou les déclencheurs des Kells provoquent des désallocations. Certaines opérations, telles que $\overline{\texttt{in}}$, permettent de réagir à des allocations, tandis que d'autres, à l'image de $\overline{\texttt{out}}$, des continuations dans les Kells ou des finalisations dans Cπ ou CDπ, permettent de réagir aux désallocations. Certaines constructions, enfin, marquent des allocations sans spécifier si elles doivent avoir lieu ou si elles ont déjà eu lieu, comme $a\,[]$ ou (νa).

Dans tous les cas, ces mécanismes de contrôle permettent à l'environnement de déclencher l'allocation et la désallocation ou, au minimum, de pouvoir répondre à celles-ci. De plus, ces mécanismes imposent un certain ordre sur les opérations d'allocation et de désallocation, ce qui est la clef de l'écriture de protocoles ressource-contrôlés, puisque des processus peuvent attendre d'avoir atteint un état dans lequel des ressources sont disponibles avant d'entreprendre des actions qui vont nécessiter l'allocation de nouvelles ressources.

Complexité des mécanismes Nous avons pu constater que, avec un peu d'expérience, il devient naturel d'écrire des protocoles conscients des ressources et qui ne laissent derrière eux aucune entité allouée. Cela est, malheureusement, moins vrai dans les Controlled Ambients que dans les autres langages, en raison de l'impossibilité de supprimer un ambient arbitraire.

Notamment, par rapport aux langages de programmation habituels, nous pouvons remarquer que l'absence de partage simplifie le problème du nettoyage. Notons aussi que les mécanismes d'allocation sont assez différents de ce que l'on peut rencontrer dans les langages de programmation, à l'exception de (`newloc`) et (ν_{chan}). Les mécanismes liés à la désallocation, eux, sont plus variés. Ainsi, si les primitives des Seals et des Kells qui permettent de supprimer une entité peuvent être comparées à `delete` ou `free`() de langages impératifs, la finalisation de Cπ, les cocapacités de sortie des ambients ou les continuations des Kells sont apparentés aux finalisations des langages avec ramasse-miettes.

Notons que, comme nous l'avons mentionné, il nous semble possible d'introduire de manière automatique des cocapacités de manière à transformer un terme des Mobile Ambients bien conçu en un processus des Controlled Ambients, quitte à retoucher ce processus pour fournir des garanties plus fortes. Les allocations et désallocation des Seals et des Kells, quant à elles, sont implantées profondément dans le langage et sont donc déjà nécessaires à l'écriture du moindre terme. Enfin, il nous semble qu'un mécanisme de typage (quasi-)linéaire ou pseudo-linéaire, comme ceux que nous avons présentés aux sections 6.2 et 10.3, peut permettre de placer automatiquement des opérateurs de finalisation dans des termes Cπ ou CDπ.

Implantation des mécanismes Pour l'essentiel, les mécanismes d'allocation et de désallocation sont assez naturels à implanter ou, pour le moins, ne posent pas de problèmes supplémentaires.

Comme nous en avions discuté, les cocapacités des CA sont assez proches de ce qu'une machine virtuelle pour les Mobile Ambients peut exiger.

De la même manière, l'allocation et la désallocation sont en fait des primitives de déplacement dans les Seals ou des Kells – les considérer en plus comme des opérations d'allocation et de désallocation ne modifie en rien le calcul ou les difficultés éventuelles d'implantation.

Enfin, à notre sens, l'allocation dans Cπ et CDπ est naturelle et ne pose pas de problèmes particuliers. De même, si la désallocation nécessite un ramasse-miettes – et même un ramasse-miettes distribué dans le cas de CDπ – un tel ramasse-miettes est de toute manière nécessaire pour toute implantation du π-calcul ou de Dπ et l'ajout de la finalisation, en l'absence de résurrection, n'entraîne aucune difficulté. Les mécanismes d'élimination des processus morts, à l'inverse, peuvent par contre ajouter un fort niveau de complexité à une machine virtuelle. Cependant, il existe un certain nombre de mécanismes connus qui permettent de détecter dynamiquement les deadlocks [12] et les livelocks [38].

13.2 Systèmes de types

À l'aide des mécanismes de contrôle, nous avons développé plusieurs systèmes de types pour garantir le contrôle des ressources.

Points communs à tous ces systèmes de types Les principes de base de ces systèmes de types sont communs. Ainsi, dans chaque cas, l'environnement de typage contient les informations sur l'encombrement des entités et sur la capacité des sites. De même, le type d'un processus contient une borne supérieure sur le nombre de ressources nécessaires pour l'exécution du processus, borne qui doit, le cas échéant, être répartie entre les composantes parallèles du processus. Dans tous ces systèmes, si un préfixe d'un processus entraîne ou accompagne une désallocation locale, nous considérons que le processus peut ensuite réutiliser les ressources précédemment allouées à l'entité.

Notons ainsi que, si les mécanismes d'allocation/désallocation diffèrent dans leur sémantique, nous en extrayons une signification similaire du point de vue des ressources.

Les propriétés des systèmes sont à leur tour proches, notamment puisque les processus peuvent être sous-typés, et puisqu'un processus typé ne peut évoluer qu'en des processus ressource-contrôlés.

Spécificités Chaque système de types présente des aspects particuliers qui permettent de gérer les spécificités du calcul. Ainsi, le formalisme des Controlled Ambients propose des communications locales anonymes, dont nous notons le sujet dans le type du processus et de l'ambient, tandis que les autres langages offrent des communications plus élaborées, que nous typons à l'aide des constructions appropriées, canaux ou messages, d'une manière parfois fort élaborée. De même, dans Cπ et CDπ, nous sommes obligés de distribuer des autorisations de réutilisation des ressources pour éviter de prendre plusieurs fois en compte la désallocation d'une entité. Enfin, dans les CA et Seals, nous devons prendre en compte la récursion, alors que celle-ci se type naturellement dans les Kells et qu'elle est remplacée par une réplication dans Cπ et CDπ.

Le formalisme des Kells se distingue sur un point, car l'un des systèmes que nous avons fournis nécessite en fait une réduction typée pour pouvoir être appliqué.

Évolution Nous avons été capables d'étendre les systèmes de types de manière à raisonner avec des ensembles de niveaux de ressources au lieu de $\mathbf{N} \cup \{\infty\}$. Si l'essentiel des définitions et des règles ainsi obtenues est fondamentalement identique aux définitions et règles originelles, adapter Cπ nous a conduit à introduire des paramètres r et w supplémentaires, plus simples à interpréter que le paramètre z qu'ils remplacent. À l'inverse, pour les CA, nous avons du remplacer une condition compréhensible par une autre plus arcane.

À l'aide de ces systèmes de types enrichis, nous avons pu prouver un certain nombre de propriétés supplémentaires, parfois fort complexes, et ainsi approximer le contrôle offert par certains systèmes de types qui considèrent d'une manière différente les notions de ressources et de contrôle.

Inférence De l'ensemble des règles employées pour le typage, les seuls cas que nous considérons comme problématiques sont les messages des CA, des autorisations de réutiliser les ressources de Cπ et CDπ et, dans le cas des CA enrichis, d'une condition pour la coouverture CA. En effet, toutes les autres vérifications sont en fait des inégalités simples entre des sommes d'éléments de l'ensemble de niveaux de ressources. Si l'on cherche à inférer le type d'un système

dans lequel les cas problématiques sont déjà réglés, il suffit donc de collecter les inégalités en question et de résoudre le système ainsi obtenu.

En particulier, dans le cas des ensembles de niveaux de ressources $(\mathbf{N} \cup \{\infty\})^a \times \mathcal{B}^b$, ce qui recouvre tous nos exemples, il existe des procédures de décision sur les inégalités en question, si bien qu'il nous semble à peu près certain que l'inférence est décidable, et même aisée.

Les cas problématiques restent à régler mais nous semblent, en fait, relativement aisés à gérer – à l'exception de la condition sur les ouvertures dans les CA.

13.3 Résultats

Au cours de ce travail, nous avons étudié et modifié de nombreux calculs – citons les Mobile Ambients/Safe Ambients/Controlled Ambients, les Seals, plusieurs versions des Kells, Nomadic Pict, les NBA, le π-calcul et Dπ. Pour chacun de ces calculs, nous avons effectivement réussi à définir formellement une notion de ressource, à déterminer comment contrôler les ressources et à développer un système de types pour garantir le contrôle de cette ressource. Nous considérons donc avoir rempli avec succès notre objectif.

Tout au long de cet exposé, nous avons présenté les limitations de nos résultats, ainsi que les directions supplémentaires vers lesquelles nous avons commencé à pousser nos recherches pour dépasser ces limitations ou étendre le champ d'application de nos méthodes.

Si, nous semble-t-il, l'intérêt principal de nos contributions soit celui d'outils de tests pour nos théories, quelques aspects de notre travail méritent, à notre sens, d'être réemployés. Parmi ces aspects, nous noterons essentiellement le mécanisme des continuations que nous avons introduit dans le langage des Kells, la finalisation que nous avons apportée au π-calcul et l'équilibrage des coûts de communication, tel que nous l'avons défini dans le système de types enrichis de Cπ. À cela, nous ajouterions un système de types enrichi – probablement celui de Cπ ou CDπ, éventuellement corrigé pour améliorer la présentation des autorisations de réutilisation des ressources.

En particulier, il serait intéressant de voir quel langage pourrait être développé à partir de ces primitives – auxquelles nous aimerions ajouter le spawner $\triangleright$ de BoCa – et d'un tel système de types.

Chapitre 14

Travaux futurs

> I don't want to achieve immortality through my work. I want to achieve it through not dying.
>
> – Woody Allen

Il reste un certain nombre de pistes ouvertes que nous n'avons pas encore eu le temps de suivre et que nous aimerions explorer ou voir explorées dans l'avenir. Notamment, nous nous posons des questions sur la possibilité d'appliquer nos méthodes à des calculs éloignés de ceux que nous avons déjà étudiés ou à des variantes de certains formalismes. De même, nous souhaiterions voir jusqu'où notre système de types peut aller.

14.1 Autres formalismes

Parmi les formalismes que nous n'avons pas pu étudier, faute de temps, citons BoPi [33] et le calcul des Fusions [67], ainsi que le Join-Calculus [29].

Distribution implicite : BoPi Le langage BoPi [33] est à la fois une évolution du calcul des Fusions[67] et une variante du π-calcul qui introduit une distribution implicite et transparente des processus entre des localités essentiellement anonymes.

Une construction du langage, le répétiteur linéaire (ou linear forwarder, noté $x \multimap y$), permet de gérer de manière souple les communications distantes, sans avoir besoin de recourir à des migrations de processus. Des mécanismes simples permettent d'introduire automatiquement les répétiteurs dans les termes et chaque répétiteur n'est utilisé, au plus, qu'une seule fois, ce qui fait de cette construction une entité intéressante du point de vue des ressources.

Cette forme de distribution pose aussi d'autres questions intéressantes sur la notion de ressource. Ainsi, s'il semble aisé de déterminer à quelle localité appartient une entité/canal de communication, il est plus problématique de déterminer comment doivent fonctionner la désallocation et la finalisation. En effet, les migrations sont à la fois implicites et décidées dynamiquement, ce qui peut poser des difficultés pour décider dans quels sites placer les processus de finalisation.

Procédures distantes : le Join-Calcul Le Join Calcul [29] est un formalisme proche des langages de programmation, construit autour d'une forme de réceptions polyadiques, qui à leur tour généralisent les services RPC (Remote Procedure Call) fournis par de nombreux systèmes d'exploitation.

Cette algèbre de processus est assimilable à une variante asynchrone du π-calcul, sans construction ν explicite, dans laquelle toutes les communications sont répliquées, comme les services RPC ou assimilés, et permettent de synchroniser n parties. Ainsi, dans un service (ou *définition*), $J \triangleright P$, lorsque le motif J est reconnu de manière distribuée sur le réseau (i.e. les éléments de J ne sont pas obligés d'apparaître tous sur la même machine), le processus P est déclenché sur la machine locale. Les définitions $J \triangleright P$ ont, depuis, été adaptées et incorporées dans le calcul des Kells sous la forme de déclencheurs $\xi \triangleright P$ dans le calcul des Kells.

Il existe aussi un Distributed Join Calculus [30], qui ajoute au formalisme des notions de distribution et de mobilité distinctes aussi bien de celle des calculs à mobilité de noms que de celle des calculs à mobilité de site. Ainsi, les localités sont rassemblées en une hiérarchie mais les noms sont uniques, si bien que les messages n'ont pas à être routés.

Pour contrôler les ressources dans le (Distributed) Join Calculus, nous pourrions envisager de généraliser l'opérateur de finalisation pour permettre de gérer les sites (comme dans CDπ), et les définitions obsolètes $J \triangleright P$ elles-mêmes.

Espaces partagés : Linda... De même, nous serions intéressé par l'étude de formalismes fondés sur le concept d'espaces de n-uplets, tels que Linda [34] et les calculs dérivés. En effet, la gestion des ressources est très différente dans une architecture qui représente une mémoire partagée entre plusieurs localités, ou plusieurs mémoires, partagées de manière parfois asymétrique, ou composées à l'aide de hiérarchies parfois complexes.

À l'inverse des calculs précédents, cependant, nous n'avons absolument pas commencé à approcher le sujet.

Variantes sur les langages Comme nous l'avons mentionné plus tôt, nous aimerions déterminer les propriétés d'un langage inspiré à la fois de Cπ et de BoCa, au moins pour l'opérateur $\triangleright$, et peut-être doté d'une gestion des ressources à l'aide de groupes.

Plus immédiatement, comme nous sommes en contact avec l'équipe qui conçoit le calcul des Kells, nous avons eu droit à un aperçu d'une prochaine évolution du formalisme, qui introduit dans le langage une gestion de composants partagés entre plusieurs localités. Nous aimerions déterminer à quel point les techniques que nous avons développées dans le cadre, plus simple, où rien n'est partagé, peuvent être adaptées à ce contexte. Une première esquisse nous a laissés optimistes à ce sujet.

14.2 Systèmes de types

Ensembles de ressources Comme nous en avons brièvement parlé, nous avons en chantier un système de types plus puissant encore que ceux que nous avons présentés dans la section 9.

Les mécanismes de base, une fois de plus, sont identiques à ceux que nous avons présentés dans tout l'exposé. La différence majeure vient du fait que nous ajoutons des notions d'encombrement pour des opérations jusque là invisibles comme les capacités des ambients et les tests d'égalité et que nous distinguons plusieurs notions d'encombrement pour une même entité, selon la manière dont est utilisée cette entité.

En instanciant ce système de types pour les Controlled Ambients, nous sommes capables de garantir des propriétés bien plus complexes, telles que le nombre de déplacements d'un site ou le respect de protocoles plus fins sur l'ordre des entrées et des sorties, à l'image de 2.4.2, quoiqu'avec moins de puissance.

Cette extension nécessiterait d'être terminée, portée aux calculs qui nous intéressent plus que les Controlled Ambients et étudiée.

Généralisation des types de sites Dans tout cet exposé, nous avons considéré que chaque site était caractérisé, entre autres, par sa capacité s et son encombrement e et que ces deux quantités étaient des éléments de l'ensemble des ressources $\mathcal{S}$. À l'inverse, le travail sur contrôle des ressources dans les modèles d'exécutions synchrones et coopératives, présenté à la section 2.3, les assignations sont des fonctions qui déterminent l'encombrement d'une valeur en fonction des paramètres employés pour construire cette valeur. Nous pourrions étendre le concept de manière à obtenir des sites caractérisés par une fonction qui, à l'effet de P, associe l'effet de $a\,[P]$. Une telle flexibilité nous permettrait notamment de mélanger nos politiques de contrôle habituelles, représentées par des fonctions $s \mapsto e$, à un contrôle de ressources partagées par l'arbre entier, comme dans BoCa, à l'aide de la fonction identité $Id_{\mathcal{S}}$, et probablement à d'autres formes de ressources.

Nous espérons être aussi capables, à l'aide d'un tel système, de présenter des bornes formelles, "en fonction des entrées", à l'image des types de `lfd_infer` (cf. section 2.2).

Système de types générique Le système de types générique pour le π-calcul [46] proposé par Atsushi Igarashi et Naoki Kobayashi permet d'examiner des termes d'une variante du π-calcul à l'aide d'un système de types unique, instancié pour vérifier différentes politiques. Dans ce cadre, très puissant, il est possible d'exprimer aussi bien l'absence de deadlocks que des protocoles d'accès aux ressources similaires à ceux que nous avons présentés dans la section 2.6.

Si le π-calcul utilisé dans ce travail ne contient pas de finalisation, il est possible d'employer ce système de types pour garantir qu'un canal ne sera plus utilisé pour communiquer à partir d'un certain point, ce qui peut nous fournir une possibilité d'introduire intelligemment les opérateurs de finalisation dans un terme.

Même si les extensions de nos typages pour le π-calcul permettent de formuler des politiques comparables à celles du système générique, nous disposons d'une expressivité globalement très inférieure. Il serait donc intéressant de chercher à porter ce système à Cπ et de déterminer s'il peut ainsi permettre d'exprimer les politiques contrôle des ressources.

Présentations logiques Enfin, et même si cette préoccupation peut sembler moins primordiale, nous souhaiterions étudier d'autres manières de présenter

nos systèmes de types et nos politiques de contrôle de ressources. Nous espérons ainsi pouvoir améliorer la présentation de la distribution des autorisations de réutilisation des ressources dans Cπ mais aussi, peut-être, arriver à trouver une formulation plus proche d'une logique.

En particulier, les logiques spatiales [43, 71] se veulent conscientes de l'espace et permettent d'exprimer des propriétés sur le contenu des sites et sur leur présence. En particulier, la logique de séparation a été utilisée pour représenter la gestion de droits d'accès ou des notions de garbage-collection [14]. Nous espérons, qu'il est possible de formuler de même des politiques de contrôle des ressources sur $\mathbf{N} \cup \{\infty\}$ dans les algèbres de processus et, peut-être, dans d'autres langages.

14.3 Mise en œuvre

Au long de ce travail, nous n'avons pas pu réellement expérimenter nos techniques sur des systèmes réels, qu'il s'agisse de protocoles complexes et réellement utilisés[1] ou de codage.

Nous espérons avoir l'occasion de remédier à cette absence dans le cadre du calcul des Kells. En effet, ce formalisme est actuellement en cours d'implantation sous la forme de langage de programmation, avec les modifications que nous avons proposées. Nous comptons adapter notre système de types pour les Kells à cette implantation.

De plus, l'outil TyPiCal [49], récemment mis à disposition, fournit une implantation libre de plusieurs systèmes de types pour le π-calcul, qui pourrait nous servir de base pour expérimenter nos systèmes enrichis pour le contrôle des ressources.

[1] En fait, nous avions commencé à modéliser un fragment de l'ordonnanceur noyau de Linux à l'aide des Controlled Ambients, en coopération avec une équipe de recherche dans le domaine de la programmation système mais ce travail est resté sans suite.

Bibliographie

"The secret to creativity is knowing how to hide your sources."
– Albert Einstein

[1] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. Technical report, Inria, 1996.

[2] R. M. Amadio. Max-plus quasi-interpretations. Research report 10-2002, LIF, Marseille, France, Dec 2002.

[3] R. M. Amadio and S. D. Zilio. Resource Control for Synchronous Cooperative Threads. Research report 22-2004, LIF, Marseille, France, May 2004.

[4] Apple Computers. *Inside Macintosh : Memory*, 1992.

[5] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1998.

[6] H. G. Baker. Use-once variables and linear objects : storage management, reflection and multi-threading. *SIGPLAN Not.*, 30(1), 1995.

[7] F. Barbanera, M. Bugliesi, M. Dezani, and V. Sassone. A calculus of bounded capacities. In *Proceedings of Advances in Computing Science, 9th Asian Computing Science Conference, ASIAN'03*, volume 2896 of *Lecture Notes in Computer Science*. Springer, 2003.

[8] G. Berry. *The Foundations of Esterel*. MIT Press, 2000. Editors : G. Plotkin, C. Stirling and M. Tofte.

[9] H. Boehm. Destructors, finalizers, and synchronization. In *Proceedings of the 2003 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.

[10] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On lexicographic termination ordering with space bound certifications. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *PSI*, volume 2244 of *Lecture Notes in Computer Science*. Springer, July 2001.

[11] G. Boudol. Ulm : a core programming model for global computing. In *Proceedings of the European Symposium on Programming*, 2004.

[12] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*. ACM Press, 1984.

[13] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *Proceedings of the 22nd International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[14] C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Compututer Science*, 298(3), 2003.

[15] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*. Springer-Verlag, 2000.

[16] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures : First International Conference*. Springer-Verlag, Berlin Germany, 1998.

[17] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1999.

[18] L. Cardelli, A. D. Gordon, and G. Ghelli. Mobility types for mobile ambients. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1999.

[19] G. Castagna and F. Z. Nardelli. The seal calculus revisited : Contextual equivalence and bisimilarity. In *Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 2002.

[20] G. Castagna and J. Vitek. Commitment and confinement for the seal calculus. Technical report, University of Genova, 1999.

[21] G. Chapman, J. Cleese, E. Idle, T. Gilliam, T. Jones, and M. Palin. I'm not dead, i feel fine. In G. Almighty, editor, *5th Holy Grail acquisition and preservation workshop*, 1975.

[22] W. Charatonik, A. D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2002.

[23] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99 : The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, New York, NY, 1999.

[24] S. Dal Zilio and A. D. Gordon. Region analysis and a π-calculus with groups. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*. Springer, Aug. 2000.

[25] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[26] G. Ferrari, E. Moggi, and R. Pugliese. Guardians for ambient-based monitoring. In V. Sassone, editor, *F-WAN : Foundations of Wide Area Network Computing*, number 66 in ENTCS. Elsevier Science, 2002.

[27] F. L. Fessant, I. Piumarta, and M. Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Proceedings of te Conference on Programming Languages Design and Implementation*, Montreal (Canada), June 1998. ACM SIGPLAN.

[28] C. Fournet and M. Abadi. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, 2001.

[29] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1996.

[30] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*. Springer-Verlag, 1996.

[31] C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*. Springer-Verlag, 2000.

[32] C. Fournet and A. Schmitt. An implementation of ambients in JoCaml. In *Proceedings of the 5th ECOOP Workshop on Mobile Object Systems (MOS'99)*, Lisbon, Portugal, 1999.

[33] P. Gardner, C. Laneve, and L. Wischik. Linear forwarders. In R. Amadio and D. Lugiez, editors, *Proceedings of the 14th International Conference on Concurrency Theor*, volume 2761 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[34] D. Gelernter. Generative communication in linda. *ACM Transsactions on Programming Languages and Systems*, 7(1), 1985.

[35] A. Gordon. Notes on nominal calculi for security and mobility. In R. Focardi and R. Gorrieri, editors, *Proceedings of the 2nd International School on Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[36] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Proceedings of the International Conference on Software : Theory and Practice*, 2000.

[37] Hachette. Dictionnaire universel francophone en ligne, 2003.

[38] H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In R. Cleaveland and H. Garavel, editors, *Electronic Notes in Theoretical Computer Science*, volume 66. Elsevier, 2002.

[39] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems (extended abstract). In *Conference Record of FOSSACS03*, volume 2620 of *Lecture notes in Computer Science*, 2003.

[40] M. Hennessy, J. Rathke, and N. Yoshida. safedpi : a language for controlling mobile code. Technical Report cs02 :2003, University of Sussex, 2003.

[41] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *in Proceedings of High-Level Concurrent Languages*, volume 16.3. Elsevier Science Publishers, 1998.

[42] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173, 2002.

[43] D. Hirschkoff, Étienne Lozes, and D. Sangiorgi. Separability, expressiveness, and decidability in the ambient logic. In *Proceedings of the 17th IEEE Symposium22-25 July 2002 on Logic in Computer Science*. IEEE Computer Society, 2002.

[44] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic J. of Computing*, 7(4), 2000.

[45] M. Hofmann. A type system for bounded space and functional in-place update–extended abstract. *Nordic Journal of Computing*, 7(4), Autumn 2000. An earlier version appeared in ESOP2000.

[46] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2001.

[47] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.

[48] S. Jost. lfd_infer : an implementation of a static inference on heap space usage. In *Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.

[49] N. Kobayashi. TyPiCal : Type-based static analyzer for the pi-calculus.

[50] N. Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1999.

[51] N. Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2), 2002.

[52] N. Kobayashi. Useless code elimination and programm slicing for the pi-calculus. In *Proceedings of the first Asian Symposium on Programming Languages and Systems*, 2003.

[53] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

[54] M. Konečný. Functional in-place update with datatype sharing. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications*, Valencia, 2003. Springer Lecture Notes in Computer Science.

[55] G. Lautner, M. Audiard, L. Ventura, B. Blier, F. Blanche, C. Millot, and al. *Les Barbouzes*. Gaumont, 1964.

[56] D. Lea. Customization in C++. In *Proceedings of the USENIX C++ Conference*, 1990.

[57] X. Leroy, with Damien Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.07 Documentation and user's manual*. Inria, 2003.

[58] F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM Transsactions on Programming Languges and Systems*, 25(1), 2003.

[59] C. Lhoussaine and V. Sassone. A dependently typed ambient calculus. In *Proceedings of the European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*. Springer, 2004.

[60] K. MacKenzie and N. Wolverson. Camelot and grail : resource-aware functional programming on the JVM. 2003.

[61] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.

[62] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[63] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report -86, Penn State University, 1989.

[64] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.

[65] F. Örvill and M. Therning. Evaluation of interprocess communication methods in a component based environment.

[66] G. Orwell. *Nineteen-eighty-four*. MiniTruth, 1947.

[67] J. Parrow and B. Victor. The fusion calculus : Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*, 1998.

[68] G. Perec. Experimental demonstration of the tomatotopic organization in the soprano (cantatrix sopranica l.). Technical report, Laboratoire de physiologie, Faculté de médecine Saint-Antoine, 1974.

[69] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings 8th IEEE Logics in Computer Science*, Montreal, Canada, 1993.

[70] A. Ravara, A. Matos, V. T. Vasconcelos, and L. Lopes. A lexically scoped distributed pi-calculus. Technical Report 02–4, Department of Informatics, Faculty of Sciences, University of Lisbon, Mar. 2002.

[71] J. C. Reynolds. Separation logic : A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium22-25 July 2002 on Logic in Computer Science*. IEEE Computer Society, 2002.

[72] A. Rowstron and A. Wood. Bonita : a set of tuple space primitives for distributed coordination. In *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, 1997. IEEE Computer Society Press.

[73] D. Sangiorgi and A. Valente. A distributed abstract machine for safe ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*. Springer-Verlag, 2001.

[74] A. Schmitt and J.-B. Stefani. The M-calculus : a higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2003.

[75] A. Schmitt and J.-B. Stefani. The kell calculus, a family of higher-order distributed process calculi, 2004.

[76] F. Smith, D. Walker, and G. Morrisett. Alias types. *Lecture Notes in Computer Science*, 1782, 2000.

[77] J.-B. Stefani. A calculus of kells. In V. Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[78] D. Teller. Formalisms for mobile resource control. In *Proceedings of FGC'03*, volume 85 of *ENCS*. Elsevier, 2003.

[79] D. Teller. Resource recovery in the π-calculus. In *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science*, 2004. tbp.

[80] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proceedings of the 13th International Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[81] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[82] A. Unyapoth. *Nomadic Pi Calculi : Expressing and Verifying Infrastructure for Mobile Computation*. PhD thesis, Computer Laboratory, University of Cambridge, june 2001.

[83] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*. North Holland, 1990.

[84] L. Wischik. New directions in implementing the pi calculus. In *CaberNet Radicals Workshop*, 2002.

[85] P. Wojciechowski and P. Sewell. Nomadic pict : Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[86] P. Wojciechowski and P. Sewell. Nomadic pict : Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[87] S. D. Zilio. Mobile processes : a commented bibliography. In *Acte de l'école MOVEP'2k - Summer school on Modelling and Verification of Parallel processes*, volume 2067 of *Lecture Notes in Computer Science*. Springer-Verlag New York, Inc., 2001.

Annexe A

Controlled Ambients

A.1 Lemmes

A.1.1 Substitutions et renommages

Sous-lemme 1 (Substitution d'un nom d'ambient)
Si $\Gamma, n : A \vdash P : U$ *et* $\Gamma \vdash m : A$, *alors* $\Gamma \vdash P\{n \leftarrow m\} : U$.

Ce sous-lemme se prouve sans aucune difficulté, par induction structurelle sur une preuve de $\Gamma, n : A \vdash P : U$. Les seuls cas influencés par la substitution sont ceux qui correspondent aux applications de la règle règles T-AmbName au nom n.

Considérons donc une application de T-AmbName. Cette application permet de prouver $\Gamma, n : A \vdash n : A$. Or, par hypothèse, $\Gamma \vdash m : A$ donc $\Gamma \vdash n\{n \leftarrow m\} : A$. Le cas est donc prouvé.

Les autres cas sont triviaux. □

Sous-lemme 2 (Substitution d'un nom de processus)
Si $\Gamma, X : V \vdash P : U$ *et* $\Gamma \vdash Q : V$, *alors* $\Gamma \vdash P\{X \leftarrow Q\} : U$.

De la même manière, nous prouvons ce sous-lemme par induction structurelle sur une preuve de $\Gamma, X : V \vdash P : U$. Nous noterons $V = Proc(t_Q)[T_Q]$. Les seuls cas influencés sont les applications de T-ProcName à X.

Considérons donc une application de T-ProcName. Cette application nous permet de conclure $\Gamma, X : V \vdash X : Proc(u)[T_X]$ avec pour tout $u \geq t_Q$ et pour $T_X = T_Q$.

Or, par hypothèse, nous avons $\Gamma \vdash Q : Proc(t_Q)[T_Q]$ et d'après le lemme de sous-typage, nous pouvons en déduire $\Gamma \vdash Q : Proc(u)[T_Q]$. Le cas est donc prouvé.

Les autres cas sont triviaux. □

Sous-lemme 3 (Renommage d'un nom d'ambient lié)
Si $\Gamma \vdash (\nu n : T)P : U$ *et* $m \notin fv(P)$, *alors* $\Gamma \vdash (\nu m : T)P\{n \leftarrow m\} : U$.

La preuve est presque identique à celle du sous-lemme 1.

Sous-lemme 4 (Renommage d'une variable de processus liée)
Si $\Gamma \vdash \texttt{rec}\ X.P : U$ *et* $Y \notin fv(P)$, *alors* $\Gamma \vdash \texttt{rec}\ Y.P\{X \leftarrow Y\} : U$.

La preuve est, presque identique à celle du sous-lemme 2.

$$\textsc{Struct-Par}\ \frac{P \equiv Q}{P|R \equiv Q|R} \qquad \textsc{Struct-Rec}\ \frac{P \equiv Q}{\texttt{rec}\ X.P \equiv \texttt{rec}\ X.Q}$$

$$\textsc{Struct-Amb}\ \frac{P \equiv Q}{M\,[P] \equiv M\,[Q]}$$

$$\textsc{Struct-Pre}\ \frac{P \equiv Q}{M.P \equiv M.Q} \textsc{Struct-Res}\ \frac{P \equiv Q}{(\nu n : A)P \equiv (\nu n : A)Q}$$

$$\textsc{Struct-Rcv}\ \frac{P \equiv Q}{(x : A).P \equiv (x : A).Q}$$

FIG. A.1 – Propriétés de congruence de la congruence structurelle – CA.

A.1.2 Congruence Structurelle

Sous-lemme 5 (Les bons types supportent la congruence)
Soient deux processus A et B tels que $A \equiv B$. Si $\Gamma \vdash A : U$, alors $\Gamma \vdash B : U$ et si $\Gamma \vdash B : U$ alors $\Gamma \vdash A : U$.

Nous prouvons ce sous-lemme par induction structurelle sur une preuve de $A \equiv B$. Chaque cas correspond alors

- soit à une règle de congruence structurelle présentée sur la figure 3.5
- soit à une propriété d'équivalence de la relation $\equiv$ (transitivité, symétrie, réflexivité)
- soit à une des propriétés qui font de $(P/\equiv, |, \mathbf{0})$ un monoïde commutatif et associatif (commutativité, associativité, neutralité de $\mathbf{0}$ vis-à-vis de l'opération $|$)
- soit à l'α-équivalence des variables liées
- soit à une propriété de congruence de la relation $\equiv$, illustrées sur la figure A.1.

Struct-Repl

En l'absence de réplication, ce cas n'apparaît jamais dans les Controlled Ambients.

Struct-Repl-Par

En l'absence de réplication, ce cas n'apparaît jamais dans les Controlled Ambients.

Struct-Res-Res – Cas d'initialisation

Nous avons $A = (\nu n : T_n)(\nu m : T_m)P$ et $B = (\nu m : T_m)(\nu n : T_n)P$ avec $n \neq m$.

Comme T-RES appliquée deux fois est la seule règle qui permette de typer A, nous pouvons déduire que, si $\Gamma \vdash A : U$, alors, $\Gamma, n : T_n, m : T_m \vdash P : U$.

Or, comme $n \neq m$, les fonctions $\Gamma, n : T_n, m : T_m$ et $\Gamma, m : T_m, n : T_n$ sont identiques. Par suite, nous avons aussi $\Gamma, m : T_m, n : T_n \vdash P : U$. Par suite, en appliquant deux fois T-RES, $\Gamma \vdash B : U$.

Le cas symétrique est, justement, symétrique.

Struct-Res-Par – Cas d'initialisation

Nous avons $A = (\nu n : T)(P|Q)$ et $B = P|(\nu n : T)Q$ avec $n \notin fv(P)$.

De A vers B Supposons $\Gamma \vdash A : U$. La réduction la plus générale qui permette de typer A est alors

Typage de $P\|Q$			
	Si	$\Gamma, n : T \vdash P : Proc(t_P)[T_P]$	
		$Q : Proc(t_Q)[T_Q]$	
		$T_P = T_Q$	
		$u_A \geq t_P + t_Q$	
$\Rightarrow \Gamma, n : T \vdash P\|Q :$		$Proc(u_A)[T_P]$	Par T-PAR
Typage de $(\nu n : T)$(P|Q)			
$\Gamma, n : T \vdash P\|Q :$		$Proc(u_A)[T_P]$	Cf. plus haut
$\Gamma \vdash (\nu n : T)(P\|Q) :$		$Proc(u_A)[T_P]$	Par T-RES

○

De cette réduction, nous déduisons $U = Proc(u_A)[T_P]$, $u_A \geq t_P + t_Q$, $\Gamma, n : T \vdash P : Proc(t_P)[T_P]$, $\Gamma, n : T \vdash Q : Proc(t_Q)[T_Q]$ et $T_P = T_Q$.

Du lemme de renforcement à l'aide de noms d'ambients (lemme 3), comme $n \notin fv(P)$, nous pouvons déduire $\Gamma \vdash P : Proc(t_P)[T_P]$.

Nous pouvons alors procéder à la dérivation suivante

Typage de $(\nu n : T)Q$			
$\Gamma, n : T \vdash Q : Proc(t_Q)[T_Q]$		*Cf.plushaut*	
$\Rightarrow \Gamma \vdash (\nu n : T)Q :$		$Proc(t_Q)[T_Q]$	Par T-RES
Typage de $P\|(\nu n : T)Q$			
$\Gamma \vdash P :$		$Proc(t_P)[T_P]$	Cf. plus haut
$\Gamma \vdash (\nu n : T)Q :$		$Proc(t_Q)[T_Q]$	Par T-RES
	Comme	$T_P = T_Q$	
$\Rightarrow \Gamma \vdash P\|(\nu n : T)Q : Proc(u_A)[T_P]$		*Par*T-PAR	

○

Nous pouvons donc déduire $\Gamma \vdash B : U$.

De B vers A Le cas est essentiellement symétrique. Il suffit de remplacer le renforcement par l'affaiblissement.

Struct-Res-Amb – Cas d'initialisation

Nous avons $A = (\nu n : T)m\,[P]$ et $B = m\,[(\nu n : T)P]$ avec $m \neq n$.

De A vers B Supposons $\Gamma \vdash A : U$. La seule réduction possible est

Typage de $m\,[P]$			
	Si	$\Gamma, n : T \vdash P : Proc(t_P)[T_P]$	
		$m : Amb(s,e)[T_m]$	
		$T_P = T_m$	
		$s = t_P$	
$\Rightarrow \Gamma, n : T \vdash m\,[P] :$		$Proc(t_1)[T_1]$	Par T-Amb
	Avec	$t_1 \geq e$	

Typage de $(\nu n : T)m\,[P]$		
$\Gamma \vdash m\,[P] :$	$Proc(t_1)[T_1]$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu n : T)m\,[P] :$	$Proc(t_1)[T_1]$	Par T-Res

○

De cette réduction, nous déduisons $U = Proc(t_1)[T_1]$, $\Gamma, n : T \vdash P : Proc(t_P)[T_P]$, $\Gamma, n : T \vdash m : Amb(s,e)[T_m]$, $T_P = T_m$. De plus, $t_1 \geq e$ et $t_P = s$.

Comme $n \neq m$, par renforcement, nous déduisons $\Gamma \vdash m : Amb(s,e)[T_m]$. Nous pouvons alors procéder à la réduction suivante

Typage de $(\nu n : T)P$		
$\Gamma \vdash P :$	$Proc(t_P)[T_P]$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu n : T)P :$	$Proc(t_P)[T_P]$	Par T-Res

Typage de $m\,[(\nu n : T)P]$		
$\Gamma \vdash (\nu n : T)P :$	$Proc(t_P)[T_P]$	Cf. plus haut
$\Gamma \vdash m :$	$Amb(s,e)[T_m]$	Cf. plus haut
Comme	$T_m = T_P$	
$t_1 \geq e$		
$s = t_P$		
$\Rightarrow \Gamma \vdash m\,[(\nu n : T)P] :$	$Proc(t_1)[T_1]$	

○

Ce qui nous permet de conclure que $\Gamma \vdash B : U$.

De B vers A La preuve est symétrique.

Struct-Zero-Res – Cas d'initialisation

Ce cas est similaire à Struct-Res-Amb, en plus simple.

Struct-Zero-Repl

En l'absence de réplication, ce cas n'apparaît jamais dans les Controlled Ambients.

Transitivité – Cas d'héritage

Nous avons alors $A \equiv C$ et $C \equiv B$ pour un certain processus C.

Si nous supposons $\Gamma \vdash A : U$, par hypothèse d'induction appliquée à A et C, nous avons $\Gamma \vdash C : U$. Par hypothèse d'induction appliquée à C et B, nous avons aussi $\Gamma \vdash B : U$. Le cas symétrique est identique.

Symétrie – Cas d'héritage

Nous avons alors $A = B'$ et $B = A'$ où $A' \equiv B'$. La propriété s'obtient directement par hypothèse d'induction appliquée à A' et B'.

Réflexivité – Cas d'initialisation

Nous avons alors $A = P$ et $B = P$. Trivialement, $\Gamma \vdash A : U \iff \Gamma \vdash P : U \iff \Gamma \vdash B : U$.

Commutativité – Cas d'initialisation

Nous avons $A = P|Q$ et $B = Q|P$. La preuve du cas découle directement de la symétrie de la règle T-PAR, qui est la seule règle qui permette de typer A ou B.

Associativité – Cas d'initialisation

Nous avons $A = P|(Q|R)$ et $B = (P|Q)|R$. La preuve est similaire à ce qui précède et se résume à deux applications de T-PAR.

Renommage – Cas d'initialisation

Nous prouvons ce cas par application des sous-lemmes 3 et 4.

Struct-Par – Cas d'héritage

Si la règle STRUCT-PAR s'applique, nous avons $A \equiv A' \mid R$ et $B = B' \mid R$ avec $A' \equiv B'$.

Comme T-PAR est la seule règle qui permette de typer A et comme A est typable dans Γ, nous pouvons écrire $\Gamma \vdash A' : Proc(t'_A)[T]$, $\Gamma \vdash R : Proc(t_R)[T]$ et $\Gamma \vdash A : Proc(t_A)[T]$ avec $t_A \geq t'_A + t_R$. Or, par hypothèse d'induction, comme $\Gamma \vdash A' : Proc(t'_A)[T]$ et $A' \equiv V'$, alors $\Gamma \vdash B' : Proc(t'_A)[T]$. Par suite, comme $\Gamma \vdash R : Proc(t_R)[T]$ et $t_A \geq t'_A + t_R$, par T-PAR, nous déduisons $\Gamma \vdash B : Proc(t_A)[T]$, c'est-à-dire $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière.

Struct-Rec – Cas d'héritage

Si la règle STRUCT-REC s'applique, nous avons $A = \texttt{rec}\ X.A'$ et $B = \texttt{rec}\ X.B'$ avec $A' \equiv B'$.

Comme T-REC est la seule règle qui permette de typer A, nous pouvons déduire que, si $\Gamma \vdash A : Proc(u)[T]$, alors $\Gamma, X : Proc(t)[T] \vdash A' : Proc(t)[T]$ avec $t \leq u$. Or, si $\Gamma, X : Proc(t)[T] \vdash A' : Proc(t)[T]$, par hypothèse d'induction, nous pouvons déduire $\Gamma, X : Proc(t)[T] \vdash B' : Proc(t)[T]$ d'où, par T-RES, $\Gamma \vdash B : Proc(u)[T]$. C'est-à-dire $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière.

Struct-Amb – Cas d'héritage

Si la règle STRUCT-AMB s'applique, nous avons $A = M\,[A]$ et $B = M\,[B']$ avec $A' \equiv B'$.

Comme T-AMB est la seule règle qui permette de typer A, nous pouvons déduire que, si $\Gamma \vdash A : Proc(t)[T]$, alors $\Gamma \vdash M : Amb(s,e)[T_a]$ et $\Gamma \vdash A' : Proc(s)[T_a]$ avec $t \geq e$. Or, si $\Gamma \vdash A' : Proc(s)[T_a]$, par hypothèse d'induction, nous pouvons déduire $\Gamma \vdash B' : Proc(s)[T_a]$. Comme $\Gamma \vdash M : Amb(s,e)[T_a]$ et $t \geq e$, par T-AMB, nous obtenons $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière.

Struct-Pre – Cas d'héritage

Le cas se traite comme STRUCT-AMB.

Struct-Res – Cas d'héritage

Si la règle STRUCT-RES s'applique, nous avons $A = (\nu n : T)A'$ et $B = (\nu n : T)B'$ avec $A' \equiv B'$.

Comme T-RES est la seule règle qui permette de typer A, nous pouvons déduire que, si $\Gamma \vdash A : U$, alors $\Gamma, n : T \vdash A' : U$. Or, si $\Gamma, n : T \vdash A' : U$, par hypothèse d'induction, nous pouvons déduire $\Gamma, n : T \vdash B' : U$ d'où, par T-RES, $\Gamma \vdash B : U$.

De même, si $\Gamma \vdash B : U$, alors $\Gamma, n : T \vdash B' : U$. Or, si $\Gamma, n : T \vdash B' : U$, par hypothèse d'induction, nous pouvons déduire $\Gamma, n : T \vdash A' : U$ d'où, par T-RES, $\Gamma \vdash A : U$.

Ce qui conclut le cas.

Struct-Rcv – Cas d'héritage

Si la règle STRUCT-RCV s'applique, nous avons $A = x : T(A)'$ et $B = x : T(B)'$ avec $A' \equiv B'$.

Comme T-RCV est la seule règle qui permette de typer A, nous pouvons écrire $\Gamma \vdash A : Proc(v)[b,T]$, avec $\Gamma, x : T \vdash A' : Proc(b)[b,A]$. Or, si $\Gamma, x : T \vdash A' : Proc(b)[b,A]$, par hypothèse d'induction, nous pouvons déduire $\Gamma, x : T \vdash B' : Proc(b)[b,A]$ d'où, par T-RES, $\Gamma \vdash B : U$.

Le cas symétrique se traite de même. □

A.1.3 Type minimal

Le corollaire 1 est en fait évident.

Supposons qu'il existe $t \in \mathbf{N} \cup \{\infty\}$ et T tels que $\Gamma \vdash P : Proc(t)[T]$. Considérons alors l'ensemble $\{u \in \mathbf{N} \cup \{\infty\}, \Gamma \vdash P : Proc(u)[T]\}$. Cet ensemble est un sous-ensemble non-vide de $\mathbf{N} \cup \{\infty\}$, qui a donc un plus petit élément. Il s'agit du type minimal que nous cherchons.

A.1.4 Utilisation des ressources

Le lemme 6 se prouve par induction sur la structure d'une preuve de $\Gamma \vdash P : U$, à l'aide de l'hypothèse d'induction "Pour tout environnement Γ et tout

processus P, si P est prouvable dans Γ, alors $res_\Gamma(P)$ est défini et, si nous notons $\Gamma \vdash P : Proc(t)[\_]$, alors $res_\Gamma(P) \leq t$."

T-Nil – Cas d'initialisation

Nous avons $P = \mathbf{0}$. Par suite, $res_\Gamma(P)$ est défini et nul, donc inférieur à t.

T-Res – Cas d'héritage

Nous avons $P = (\nu n : A)P'$ et P' est typable dans $\Gamma, n : A$. De plus, le type de P' dans cet environnement est aussi $Proc(t)[\_]$.

Par hypothèse d'induction appliquée à P', nous savons que $res_{\Gamma,n:A}(P')$ est défini. Or, par définition de res_Γ, $res_\Gamma((\nu n : A)P') = res_{\Gamma,n:A}(P')$. Nous en déduisons que $res_\Gamma(P)$ est défini et plus petit que t.

T-AmbName

Il ne s'agit pas d'un processus. Le cas ne se présente donc pas.

T-ProcName – Cas d'initialisation

Nous avons $P = X$. Par suite, $res_\Gamma(P)$ est défini et nul, donc inférieur à t.

T-Rec – Cas d'initialisation

Nous avons $P = \texttt{rec}\ X.P'$. Par suite, $res_\Gamma(P)$ est défini et nul, donc inférieur à t.

T-In, T-Out, T-CoIn, T-CoOut, T-Open, T-CoOpen

Dans tous ces cas, nous avons $P = M.P'$. Par suite, $res_\Gamma(P)$ est défini et nul, donc inférieur à t.

T-Amb – Cas d'héritage

Nous avons $P = m\,[P']$. De plus, comme P est typable dans Γ par T-AMB, m et P' sont typables dans Γ avec des types que nous noterons $Amb(s, e)[T_m]$ et $Proc(s)[T_m]$, où $t \geq e$.

Par hypothèse d'induction, comme P' est typable dans Γ, nous savons que $res_\Gamma(P')$ est défini et inférieur à s. Par conséquent,par définition de res_Γ, nous pouvons en déduire que $res_\Gamma(m\,[P'])$ est défini et vaut e. Comme $t \geq e$, nous avons prouvé le cas.

T-Par – Cas d'héritage

Nous avons $P = Q|R$, avec $\Gamma \vdash Q : Proc(t_Q)[T]$ et $\Gamma \vdash R : Proc(t_R)[T]$ où $t \geq t_Q + t_R$.

Par hypothèse d'induction, $res_\Gamma(Q)$ est donc définie et majorée par t_Q, $res_\Gamma(R)$ est définie et majorée par t_R. Par définition, $res_\Gamma(Q|R)$ est donc définie comme $res_\Gamma(Q) + res_\Gamma(R)$, qui est majorée par $t_Q + t_R$ donc par t. Nous avons donc prouvé le cas.

T-Snd – Cas d'initialisation

Nous avons $P = \langle m \rangle$. Par définition, $res_\Gamma(P)$ est donc définie et nulle, donc inférieure à t.

T-Rcv – Cas d'initialisation

Nous avons $P = x : A(P)'$. Par définition, $res_\Gamma(P)$ est donc définie et nulle, donc inférieure à t. □

A.2 Théorèmes

A.2.1 Réduction du sujet

Prouvons par induction structurelle sur une preuve de $A \longrightarrow B$ que "Pour tout processus A et tout environnement Γ, si $\Gamma \vdash A : Proc(t_A)[T_A]$ et si $A \longrightarrow B$, alors $\Gamma \vdash B : Proc(t_A)[T_A]$." Dans tout ce qui suit, nous omettrons généralement les applications de la règle T-AMBNAME.

R-In – Cas d'initialisation

Notons

$$\begin{cases} A &= a\,[\texttt{in}\ b.P \mid Q] \mid b\left[\overline{\texttt{in}}^{\downarrow}\ a.R \mid S\right] \mid \overline{\texttt{out}}^{\downarrow}\ a.T \\ B &= b\,[R \mid S \mid a\,[P \mid Q]] \mid T\ . \end{cases}$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\texttt{in}\ b.P$			
	Si	$P : Proc(t_P)[T_P]$	
$\Rightarrow \Gamma \vdash \texttt{in}\ b.P :$		$Proc(t_P)[T_P]$	Par T-IN
Typage de $\texttt{in}\ b.P \mid Q$			
$\Gamma \vdash \texttt{in}\ b.P :$		$Proc(t_P)[T_P]$	Cf. plus haut
	Si	$Q : Proc(t_Q)[T_Q]$	
		$T_P = T_Q$	
		$u_1 \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash \texttt{in}\ b.P \mid Q :$		$Proc(u_1)[T_P]$	Par T-PAR
Typage de $a\,[\texttt{in}\ b.P \mid Q]$			
$\Gamma \vdash \texttt{in}\ b.P \mid Q :$		$Proc(u_1)[T_P]$	Cf. plus haut
	Si	$a : Amb(s_a, e_a)[T_a]$	
		$T_a = T_P$	
		$s_a = u_1$	
		$u_2 \geq e_a$	
$\Rightarrow \Gamma \vdash a\,[\texttt{in}\ b.P \mid Q] :$		$Proc(u_2)[T_2]$	Par T-AMB
Typage de $\overline{\texttt{in}}^{\downarrow}\ a.R$			
	Si	$R : Proc(t_R)[T_R]$	
		$u_3 \geq e_a + t_R$	
$\Gamma \vdash a :$		$Amb(s_a, e_a)[T_a]$	Cf. plus haut
$\Rightarrow \Gamma \vdash \overline{\texttt{in}}^{\downarrow}\ a.R :$		$Proc(u_3)[T_R]$	Par T-COIN
Typage de $\overline{\texttt{in}}^{\downarrow}\ a.R \mid S$			

	Si	$S : Proc(t_S)[T_S]$	
		$u_4 \geq u_3 + t_S$	
		$T_S = T_R$	
$\Rightarrow \Gamma \vdash \overline{\texttt{in}}^{\downarrow}\ a.R \mid S :$		$Proc(u_4)[T_R]$	Par T-Par
Typage de $b\left[\overline{\texttt{in}}^{\downarrow}\ a.R \mid S\right]$			
$\Gamma \vdash \overline{\texttt{in}}^{\downarrow}\ a.R \mid S :$		$Proc(u_T)[T_4]$	Cf. plus haut
	Si	$b : Amb(s_b, e_b)[T_b]$	
		$s_b = u_4$	
		$T_b = T_S$	
		$u_5 \geq e_b$	
$\Rightarrow \Gamma \vdash b\left[\overline{\texttt{in}}^{\downarrow}\ a.R \mid S\right] :$		$Proc(u_5)[T_5]$	Par T-Amb
Typage de $a\left[\texttt{in}\ b.P \mid Q\right] \mid b\left[\overline{\texttt{in}}^{\downarrow}\ a.R \mid S\right]$			
$\Gamma \vdash a\left[\texttt{in}\ b.P \mid Q\right] :$		$Proc(u_2)[T_2]$	Cf. plus haut
$\Gamma \vdash b\left[\overline{\texttt{in}}^{\downarrow}\ a.R \mid S\right] :$		$Proc(u_5)[T_5]$	Cf. plus haut
	Si	$T_2 = T_5$	
		$u_6 \geq u_2 + u_5$	
$\Rightarrow \Gamma \vdash a\left[\cdots\right] \mid b\left[\cdots\right] :$		$Proc(u_6)[T_2]$	Par T-Par
Typage de $\overline{\texttt{out}}^{\downarrow}\ a.T$			
	Si	$T : Proc(t_T)[T_T]$	
		$u_7 + e_a \geq t_T$	
$\Gamma \vdash a :$		$Amb(s_a, e_a)[T_a]$	Cf. plus haut
$\Rightarrow \Gamma \vdash \overline{\texttt{out}}^{\downarrow}\ a.T :$		$Proc(u_7)[T_T]$	Par T-CoOut
Typage de $a\left[\cdots\right] \mid b\left[\cdots\right] \mid \overline{\texttt{out}}^{\downarrow}\ a.T$			
	Si	$u_8 \geq u_6 + u_7$	
		$T_2 = T_T$	
$\Rightarrow \Gamma \vdash a\left[\cdots\right] \mid b\left[\cdots\right] \mid \overline{\texttt{out}}^{\downarrow}\ a.T :$		$Proc(u_8)[T_T]$	Par T-Par

○

Le type le plus général de A est donc $Proc(u_8)[T_T]$, avec les conditions nécessaires (pas nécessairement minimales) suivantes (nous avons supprimé les variables intermédiaires inutiles) :

$$\begin{cases} \Gamma \vdash & P : Proc(t_P)[T_a] \\ \Gamma \vdash & Q : Proc(t_Q)[T_a] \\ & s_a \geq t_P + t_Q \\ \Gamma \vdash & a : Amb(s_a, e_a)[T_a] \\ \Gamma \vdash & R : Proc(t_R)[T_b] \\ \Gamma \vdash & S : Proc(t_S)[T_b] \\ & s_b \geq e_a + t_R + t_S \\ \Gamma \vdash & b : Amb(s_b, e_b)[T_b] \\ \Gamma \vdash & T : Proc(t_T)[T_T] \\ & u_8 \geq e_b + t_T\ . \end{cases}$$

Notons que ces conditions correspondent à l'intuition du typage.
Nous allons prouver que, sous ces hypothèses, $\Gamma \vdash B : Proc(u_8)[T_T]$.

Typage de $P \mid Q$		
$\Gamma \vdash P :$	$Proc(t_P)[T_a]$	Par hypothèse
$\Gamma \vdash Q :$	$Proc(t_Q)[T_a]$	Par hypothèse

$\Rightarrow \Gamma \vdash P \mid Q :$		$Proc(s_a)[T_a]$	Par T-Par
	Comme	$s_A \geq t_P + t_Q$	
Typage de $a\,[P \mid Q]$			
$\Gamma \vdash P \mid Q :$		$Proc(s_a)[T_a]$	Cf. plus haut
$\Gamma \vdash a :$		$Amb(s_a, e_a)[T_a]$	Par hypothèse
$\Rightarrow \Gamma \vdash a\,[P \mid Q] :$		$Proc(e_a)[T_b]$	Par T-Amb
Typage de $R \mid S$			
$\Gamma \vdash R :$		$Proc(t_R)[T_b]$	Par hypothèse
$\Gamma \vdash S :$		$Proc(t_S)[T_b]$	Par hypothèse
$\Rightarrow \Gamma \vdash R \mid S :$		$Proc(t_R + t_S)[T_b]$	Par T-Par
Typage de $R \mid S \mid a\,[P \mid Q]$			
$\Gamma \vdash R \mid S :$		$Proc(t_R + t_S)[T_b]$	Cf. plus haut
$\Gamma \vdash a\,[P \mid Q] :$		$Proc(e_a)[T_b]$	Cf. plus haut
	Comme	$s_b \geq e_a + t_R + t_S$	
$\Rightarrow \Gamma \vdash R \mid S \mid a\,[P \mid Q] :$		$Proc(s_b)[T_b]$	Par T-Par
Typage de $b\,[R \mid S \mid a\,[P \mid Q]]$			
$\Gamma \vdash R \mid S \mid a\,[P \mid Q] :$		$Proc(s_b)[T_b]$	Par T-Par
$\Gamma \vdash b :$		$Amb(s_b, e_b)[T_b]$	Par hypothèse
$\Rightarrow \Gamma \vdash b\,[\cdots] :$		$Proc(e_b)[T_T]$	Par T-Amb
Typage de $b\,[\cdots] \mid T$			
$\Gamma \vdash b\,[\cdots] :$		$Proc(e_b)[T_T]$	Par T-Amb
$\Gamma \vdash T :$		$Proc(t_T)[T_T]$	Par hypothèse
	Comme	$u_8 \geq e_b + t_T$	
$\Rightarrow \Gamma \vdash b\,[\cdots] \mid T :$		$Proc(u_8)[T_T]$	Par T-Par

○

Ce qui prouve le cas.

R-Out – Cas d'initialisation

Notons

$$\begin{cases} A &= a\left[b\,[\texttt{out}\ a.P \mid Q] \mid \overline{\texttt{out}}^{\uparrow}\ b.R \mid S\right] \mid \overline{\texttt{in}}^{\uparrow}\ b.T \\ B &= b\,[P \mid Q] \mid a\,[R \mid S] \mid T \end{cases}$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\texttt{out}\ a.P$			
	Si	$P : Proc(t_P)[T_P]$	
$\Rightarrow \Gamma \vdash \texttt{out}\ a.P :$		$Proc(t_P)[T_P]$	Par T-Out
Typage de $\texttt{out}\ a.P \mid Q$			
	Si	$Q : Proc(t_Q)[T_Q]$	
		$T_Q = T_P$	
		$u_1 \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash \texttt{out}\ a.P \mid Q :$		$Proc(u_1)[T_P]$	Par T-Par
Typage de $b\,[\texttt{out}\ a.P \mid Q]$			
$\Gamma \vdash \texttt{out}\ a.P \mid Q :$		$Proc(u_1)[T_P]$	Cf. plus haut
	Si	$b : Amb(s_b, e_b)[T_b]$	
		$s_b = u_1$	
		$T_b = T_P$	
		$u_2 \geq e_b$	
$\Rightarrow \Gamma \vdash b\,[\texttt{out}\ a.P \mid Q] :$		$Proc(u_2)[T_2]$	Par T-Amb

Typage de $\overline{\texttt{out}}^{\uparrow}\ b.R$			
	Si	$R : Proc(t_R)[T_R]$	
$\Gamma \vdash b :$		$Amb(s_b, e_b)[T_b]$	Cf. plus haut
	Si	$u_3 + e_b \geq t_R$	
$\Rightarrow \Gamma \vdash \overline{\texttt{out}}^{\uparrow}\ b.R :$		$Proc(u_3)[T_R]$	Par T-CoOut
Typage de $\overline{\texttt{out}}^{\uparrow}\ b.R \mid S$			
$\Gamma \vdash \overline{\texttt{out}}^{\uparrow}\ b.R :$		$Proc(u_3)[T_R]$	Cf. plus haut
	Si	$S : Proc(t_S)[T_S]$	
		$T_S = T_R$	
		$u_4 \geq u_3 + t_S$	
$\Rightarrow \Gamma \vdash \overline{\texttt{out}}^{\uparrow}\ b.R \mid S :$		$Proc(u_4)[T_R]$	Par T-Par
Typage de $b[\cdots] \mid \overline{\texttt{out}}^{\uparrow}\ b.R \mid S$			
$\Gamma \vdash b[\cdots] :$		$Proc(u_2)[T_2]$	Cf. plus haut
$\Gamma \vdash \overline{\texttt{out}}^{\uparrow}\ b.R \mid S :$		$Proc(u_4)[T_R]$	Par T-Par
	Si	$u_5 \geq u_2 + u_4$	
		$T_R = T_2$	
$\Rightarrow \Gamma \vdash b[\cdots] \mid \overline{\texttt{out}}^{\uparrow}\ b.R \mid S :$		$Proc(u_5)[T_R]$	Par T-Par
Typage de $a\left[b[\cdots] \mid \overline{\texttt{out}}^{\uparrow}\ b.R \mid S\right]$			
$\Gamma \vdash b[\cdots] \mid \overline{\texttt{out}}^{\uparrow}\ b.R \mid S :$		$Proc(u_5)[T_R]$	Par T-Par
	Si	$a : Amb(s_a, e_a)[T_a]$	
		$s_a = u_5$	
		$u_6 \geq e_a$	
		$T_a = T_R$	
$\Rightarrow \Gamma \vdash a[\cdots] :$		$Proc(u_6)[T_6]$	Par T-Amb
Typage de $\overline{\texttt{in}}^{\uparrow}\ b.T$			
	Si	$T : Proc(t_T)[T_T]$	
$\Gamma \vdash b :$		$Amb(s_b, e_b)[T_b]$	Cf. plus haut
	Si	$u_7 \geq t_T + e_b$	
$\Rightarrow \Gamma \vdash \overline{\texttt{in}}^{\uparrow}\ b.T :$		$Proc(u_7)[T_T]$	Par T-CoIn
Typage de $a[\cdots] \mid \overline{\texttt{in}}^{\uparrow}\ b.T$			
$\Gamma \vdash a[\cdots] :$		$Proc(u_6)[T_6]$	Cf. plus haut
$\Gamma \vdash \overline{\texttt{in}}^{\uparrow}\ b.T :$		$Proc(u_7)[T_T]$	Cf. plus haut
	Si	$T_6 = T_T$	
		$u_8 \geq u_6 + u_7$	
$\Rightarrow \Gamma \vdash a[\cdots] \mid \overline{\texttt{in}}^{\uparrow}\ b.T :$		$Proc(u_8)[T_T]$	

○

Le type le plus général de A est donc $Proc(u_8)[T_T]$, avec les conditions nécessaires (pas forcément minimales) suivantes (nous avons supprimé les variables intermédiaires inutiles) :

$$\begin{cases} \Gamma \vdash & P : Proc(t_P)[T_b] \\ \Gamma \vdash & Q : Proc(t_Q)[T_b] \\ & s_b \geq t_P + t_Q \\ \Gamma \vdash & b : Amb(s_b, e_b)[T_b] \\ \Gamma \vdash & R : Proc(t_R)[T_a] \\ \Gamma \vdash & S : Proc(t_S)[T_a] \\ & s_a \geq t_R + t_S \\ \Gamma \vdash & a : Amb(s_a, e_a)[T_a] \\ \Gamma \vdash & T : Proc(t_T)[T_T] \\ & u_8 \geq e_a + e_b + t_T\ . \end{cases}$$

Nous allons prouver que, sous ces hypothèses, $\Gamma \vdash B : Proc(u_8)[T_T]$.

Typage de $R \mid S$			
$\Gamma \vdash R :$		$Proc(t_R)[T_a]$	Par hypothèse
$\Gamma \vdash S :$		$Proc(t_S)[T_a]$	Par hypothèse
	Comme	$s_a \geq t_R + t_S$	
$\Rightarrow \Gamma \vdash R \mid S :$		$Proc(s_a)[T_a]$	Par T-Par
Typage de $a\,[R \mid S]$			
$\Gamma \vdash R \mid S :$		$Proc(s_a)[T_a]$	Cf. plus haut
$\Gamma \vdash a :$		$Amb(s_a, e_a)[T_a]$	Par hypothèse
$\Rightarrow \Gamma \vdash a\,[R \mid S] :$		$Proc(e_a)[T_T]$	Par T-Amb
Typage de $P \mid Q$			
$\Gamma \vdash P :$		$Proc(t_P)[T_b]$	Par hypothèse
$\Gamma \vdash Q :$		$Proc(t_Q)[T_b]$	Par hypothèse
	Comme	$s_b \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash P \mid Q :$		$Proc(s_b)[T_b]$	Par T-Par
Typage de $a\,[P \mid Q]$			
$\Gamma \vdash P \mid Q :$		$Proc(s_b)[T_b]$	Cf. plus haut
$\Gamma \vdash b :$		$Amb(s_b, e_b)[T_b]$	Par hypothèse
$\Rightarrow \Gamma \vdash b\,[P \mid Q] :$		$Proc(e_b)[T_T]$	Par T-Amb
Typage de $b\,[\cdots] \mid a\,[\cdots]$			
$\Gamma \vdash a\,[R \mid S] :$		$Proc(e_a)[T_T]$	Cf. plus haut
$\Gamma \vdash b\,[P \mid Q] :$		$Proc(e_b)[T_T]$	Cf. plus haut
$\Rightarrow \Gamma \vdash b\,[\cdots] \mid a\,[\cdots] :$		$Proc(e_a + e_b)[T_T]$	Par T-Par
Typage de $b\,[\cdots] \mid a\,[\cdots] \mid T$			
$\Gamma \vdash b\,[\cdots] \mid a\,[\cdots] :$		$Proc(e_a + e_b)[T_T]$	Cf. plus haut
$\Gamma \vdash T :$		$Proc(t_T)[T_T]$	Par hypothèse
	Comme	$u_8 \geq e_a + e_b + t_T$	
$\Rightarrow \Gamma \vdash b\,[\cdots] \mid a\,[\cdots] \mid T :$		$Proc(u_8)[T_T]$	Par T-Par

○

R-Open – Cas d'initialisation

Notons

$$\begin{cases} A &= \texttt{open}\ a.P \mid a\,[\overline{\texttt{open}}\ a.Q \mid R] \\ B &= P|Q|R \end{cases}$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\overline{\texttt{open}}\ a.Q$			
	Si	$Q : Proc(t_Q)[T_Q]$	
		$a : Amb(s_a, e_a)[T_a]$	
		$T_Q = T_a$	
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ a.Q :$		$Proc(t_Q)[T_Q]$	Par T-CoOpen

Typage de $\overline{\texttt{open}}\ a.Q \mid R$			
	Si	$R : Proc(t_R)[T_R]$	
		$T_R = T_Q$	
		$u_1 \geq t_Q + t_R$	
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ a.Q \mid R :$		$Proc(u_1)[T_Q]$	Par T-Par
Typage de $a\,[\overline{\texttt{open}}\ a.Q \mid R]$			

$\Gamma \vdash a :$		$Amb(s_a, e_a)[T_a]$	Par hypothèse
	Si	$s_a = u_1$	
		$u_2 \geq e_a$	
	Comme	$T_a = T_Q$	
$\Rightarrow \Gamma \vdash a\,[\overline{\texttt{open}}\ a.Q \mid R] :$		$Proc(u_2)[T_2]$	Par T-Amb
Typage de $\texttt{open}\ a.P$			
	Si	$P : Proc(t_P)[T_P]$	
$\Gamma \vdash a :$		$Amb(s_a, e_a)[T_a]$	Par hypothèse
	Si	$T_a = T_P$	
		$u_3 + e_a \geq t_P + s_a$	
$\Rightarrow \Gamma \vdash \texttt{open}\ a.P :$		$Proc(u_3)[T_P]$	Par T-Open
Typage de $\texttt{open}\ a.P \mid a\,[\cdots]$			
$\Gamma \vdash a\,[\overline{\texttt{open}}\ a.Q \mid R] :$		$Proc(u_2)[T_2]$	Cf. plus haut
$\Gamma \vdash \texttt{open}\ a.P :$		$Proc(u_3)[T_P]$	Cf. plus haut
	Si	$T_P = T_2$	
		$u_4 \geq u_2 + u_3$	
$\Rightarrow \Gamma \vdash \texttt{open}\ a.P \mid a\,[\cdots] :$		$Proc(u_4)[T_P]$	Par T-Par

○

Le type le plus général de A est donc $Proc(u_4)[T_a]$, avec les conditions nécessaires (pas forcément minimales) suivantes (nous avons supprimé les variables intermédiaires inutiles) :

$$\begin{cases} \Gamma \vdash & Q : Proc(t_Q)[T_a] \\ & a : Amb(s_a, e_a)[T_a] \\ \Gamma \vdash & R : Proc(t_R)[T_a] \\ \Gamma \vdash & P : Proc(t_P)[T_a] \\ & u_4 \geq t_P + t_Q + t_R \end{cases}$$

Nous allons prouver que, sous ces hypothèses, $\Gamma \vdash B : Proc(u_4)[T_a]$.

Typage de $Q \mid R$		
$\Gamma \vdash Q :$	$Proc(t_Q)[T_a]$	Par hypothèse
$\Gamma \vdash R :$	$Proc(t_R)[T_a]$	Par hypothèse
$\Rightarrow \Gamma \vdash Q \mid R :$	$Proc(t_Q + t_R)[T_a]$	Par T-Par
Typage de $P \mid Q \mid R$		
$\Gamma \vdash P :$	$Proc(t_P)[T_a]$	Par hypothèse
$\Gamma \vdash Q \mid R :$	$Proc(t_Q + t_R)[T_a]$	Cf. plus haut
Comme	$u_4 \geq t_P + t_Q + t_R$	
$\Rightarrow \Gamma \vdash P \mid Q \mid R :$	$Proc(u_4)[T_a]$	Par T-Par

○

Ce qui prouve le cas.

R-Rec – Cas d'initialisation

Notons

$$\begin{cases} A & = & \texttt{rec}\ X.P \\ B & = & P\{X \leftarrow \texttt{rec}\ X.P\} \end{cases}$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\texttt{rec}\ X.P$		
Si	$\Gamma, X : Proc(t_X)[T_X] \vdash P : Proc(t_X)[T_X]$	
	$u_1 \geq t_X$	
$\Rightarrow \Gamma \vdash \texttt{rec}\ X.P :$	$Proc(u_1)[T_X]$	Par T-Rec

○

Le type le plus général de A est donc $Proc(u_1)[T_X]$, avec les conditions $\Gamma, X : Proc(t_X)[T_X] \vdash P : Proc(t_X)[T_X]$ et $u_1 \geq t_X$.

Sous cette hypothèse, nous allons prouver $\Gamma \vdash B : Proc(u_1)[T_X]$.

Typage de P		
$\Gamma, X : Proc(t_X)[T_X] \vdash P :$	$Proc(t_X)[T_X]$	Par hypothèse
$\Rightarrow \Gamma \vdash B :$	$Proc(t_X)[T_X]$	Par *Substitution (sous-lemme 2)*
Comme	$u_1 \geq T_X$	
$\Rightarrow \Gamma \vdash B :$	$Proc(t_X)[T_X]$	Par *Sous-typage*

○

Ce qui prouve le cas.

R-Comm – Cas d'initialisation

Notons

$$\begin{cases} A & = & (x : W).P \mid \langle M \rangle \\ B & = & P\{x \leftarrow M\} \end{cases}$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $(x : W).P$		
Si	$\Gamma, x : W \vdash P : Proc(t_P)[b_P, A_P]$	
	$W = A_P$	
	$b_P = t_P$	
$\Rightarrow \Gamma \vdash (x : W).P :$	$Proc(t_P)[b_P, A_P]$	Par T-Rcv
Typage de $\langle m \rangle$		
Si	$m : A_m$	
$\Rightarrow \Gamma \vdash \langle m \rangle :$	$Proc(u_1)[b_1, A_m]$	Par T-Snd
Typage de $(x : W).P \mid \langle m \rangle$		
$\Gamma \vdash (x : W).P :$	$Proc(t_P)[b_P, A_P]$	Cf. plus haut
$\Gamma \vdash \langle m \rangle :$	$Proc(u_1)[b_1, A_m]$	Cf. plus haut
Si	$b_1 = b_P$	
	$A_m = A_p$	
	$u_2 \geq u_1 + t_P$	
$\Rightarrow \Gamma \vdash :$	$Proc(u_2)[b_P, A_P]$	Par T-Par

○

Le type le plus général de A est donc $Proc(u_2)[b_P, A_P]$, avec les conditions nécessaires (pas forcément minimales) suivantes (nous avons supprimé les variables intermédiaires inutiles) :

$$\begin{cases} \Gamma, x : A_p \vdash & P : Proc(b_P)[b_P, A_P] \\ \Gamma \vdash & m : A_P \\ & u_2 \geq b_P \end{cases}$$

Sous ces hypothèses, nous pouvons prouver $\Gamma \vdash B : Proc(u_2)[b_P, A_P]$.

Typage de P		
$\Gamma, x : A_p \vdash P :$	$Proc(b_P)[b_P, A_P]$	Par hypothèse
$\Gamma \vdash m :$	A_P	Par hypothèse
$\Rightarrow \Gamma \vdash P\{x \leftarrow m\} :$	$Proc(b_P)[b_P, A_P]$	Par *Substitution (sous-lemme 2)*

○

Ce qui prouve le cas.

R-Res – Cas d'héritage

La preuve est triviale, directement par héritage et application de T-Res.

R-Amb – Cas d'héritage

La preuve est triviale, directement par héritage et application de T-Amb.

R-Par – Cas d'héritage

La preuve est triviale, directement par héritage et application de T-Par.

R-Struct – Cas d'héritage

La preuve est triviale, par héritage et deux applications du sous-lemme 5 (congruence structurelle).

□.

Par congruence structurelle, nous venons de prouver le théorème de Réduction du Suject pour le système de types de contrôle des ressources dans les Controlled Ambients.

Annexe B

Seals

B.1 Lemmes

B.1.1 Substitutions et renommages

Sous-lemme 6 (Substitution d'un nom)
Si $\Gamma, n : W \vdash P : U$ *et* $\Gamma \vdash m : W$, *alors* $\Gamma \vdash P\{n \leftarrow m\} : U$.

Ce sous-lemme se prouve exactement comme le sous-lemme 1 dans le cas des ambients. La règle T-Name remplace son homologue T-AmbName.

Sous-lemme 7 (Substitution d'un nom de processus)
Si $\Gamma, X : V \vdash P : U$ *et* $\Gamma \vdash Q : V$, *alors* $\Gamma \vdash P\{X \leftarrow Q\} : U$.

La preuve de ce sous-lemme est identique à celle du sous-lemme 2 pour les ambients.

Sous-lemme 8 (Renommage d'une variable de nom liée)
Si $\Gamma \vdash (\nu n : W)P : U$ *et* $m \notin fv(P)$, *alors* $\Gamma \vdash (\nu m : W)P\{n \leftarrow m\} : U$.

La preuve est presque identique à celle du sous-lemme 6.

Sous-lemme 9 (Renommage d'une variable de processus lié)
Si $\Gamma \vdash \texttt{rec}\ X.P : U$ *et* $Y \notin fv(P)$, *alors* $\Gamma \vdash \texttt{rec}\ Y.P\{X \leftarrow Y\} : U$.

La preuve est presque identique à celle du sous-lemme 7.

B.1.2 Congruence structurelle

Sous-lemme 10 (Les bons types supportent la congruence)
Soient deux processus A *et* B *tels que* $A \equiv B$. *Si* $\Gamma \vdash A : U$, *alors* $\Gamma \vdash B : U$ *et si* $\Gamma \vdash B : U$ *alors* $\Gamma \vdash A : U$.

Nous prouvons ce sous-lemme par induction structurelle sur une preuve de $A \equiv B$. Chaque cas correspond alors

- soit à une règle de congruence structurelle présentée sur la figure 4.2
- soit à une propriété qui d'équivalence de la relation $\equiv$ (transitivité, symétrie, réflexivité)

$$\textsc{Struct-Par}\ \frac{P \equiv Q}{P|R \equiv Q|R} \qquad \textsc{Struct-Rec}\ \frac{P \equiv Q}{\texttt{rec}\ X.P \equiv \texttt{rec}\ X.Q}$$

$$\textsc{Struct-Seal}\ \frac{P \equiv Q}{a\,[P] \equiv a\,[Q]} \qquad \textsc{Struct-Res}\ \frac{P \equiv Q}{(\nu n : A)P \equiv (\nu n : A)Q}$$

$$\textsc{Struct-Pre}\ \frac{P \equiv Q}{\alpha.P \equiv \alpha.Q}$$

FIG. B.1 – Propriétés de congruence de la congruence structurelle – Seals.

- soit à une des propriétés qui font de $(P/\ \equiv, |, \mathbf{0})$ un monoïde commutatif et associatif (commutativité, associativité, neutralité de $\mathbf{0}$ vis-à-vis de l'opération $|$)
- soit à l'α-équivalence des variables liées
- soit à une propriété de congruence de la relation $\equiv$, illustrées sur la figure B.1.

Comme, à l'exception de la règle STRUCT-PRE, toutes les règles qui définissent la congruence structurelle dans les Seals sont aussi valables dans les Controlled Ambients, et comme les règles de typage impliquées dans les Seals sont des versions simplifiées des règles correspondantes dans les CA, les preuves sont identiques et nous ne les présenterons pas.

Contentons-nous de traiter le cas STRUCT-PRE.

Réception

Dans le cas d'une réception, nous avons $A = a^{\eta}(b).A'$ et $B = a^{\eta}(b).B'$ où $A' \equiv B'$.

Comme T-RCV est la seule règle qui permette de typer A et comme A est typable dans Γ, nous pouvons écrire $\Gamma, b : W \vdash A' : T$, $\Gamma \vdash a : ICha\!n(W)$ et $\Gamma \vdash a^{\eta}(b).A' : T$. Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma, b : W \vdash A' : T$, nous avons $\Gamma, b : W \vdash B' : T$. En appliquant de nouveau T-RCV avec $\Gamma, b : W \vdash B' : T$ et $\Gamma \vdash a : ICha\!n(W)$, nous déduisons $\Gamma \vdash a^{\eta}(b).B' : T$ c'est-à-dire $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière.

Émission

Dans le cas d'une émission, nous avons $A = \overline{a}^{\eta}(b).A'$ et $B = \overline{a}^{\eta}(b).B'$ où $A' \equiv B'$.

Comme T-SND est la seule règle qui permette de typer A et comme A est typable dans Γ, nous pouvons écrire $\Gamma \vdash b : W$, $\Gamma \vdash A' : U$, $\Gamma \vdash a : ICha\!n(W)$ et $\Gamma \vdash \overline{a}^{\eta}(b).A' : U$. Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : U$, nous avons $\Gamma \vdash B' : U$. En appliquant de nouveau T-SND avec $\Gamma \vdash B' : U$, $\Gamma \vdash b : W$ et $\Gamma \vdash a : ICha\!n(W)$, nous déduisons $\Gamma \vdash \overline{a}^{\eta}(b).B' : U$ c'est-à-dire $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière.

Départ

Dans le cas d'un départ, nous avons $A = \overline{a}^{\eta}\{b\}.A'$ et $B = \overline{a}^{\eta}\{b\}.B'$ où $A' \equiv B'$.

Comme T-POP est la seule règle qui permette de typer A et comme A est typable dans Γ, nous pouvons écrire $\Gamma \vdash b : S$, $\Gamma \vdash A' : Proc(t)$, $\Gamma \vdash a : MChan(S)$ et $\Gamma \vdash \overline{a}^{\eta}\{b\}.A' : Proc(u)$ avec $u + e \geq t$ et $S = Seal(\_, e)$. Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : Proc(t)$, nous avons $\Gamma \vdash B' : Proc(t)$. En appliquant de nouveau T-POP avec toutes ces hypothèses, nous déduisons $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière.

Arrivée

Dans le cas d'une arrivée, nous avons

$$\begin{cases} A & = a^{\eta}\{b_1, \cdots, b_k\}.A' \\ B & = a^{\eta}\{b_1, \cdots, b_k\}.B' \end{cases}$$

où $A' \equiv B'$.

Comme T-PUSH est la seule règle qui permette de typer A et comme A est typable dans Γ, nous pouvons écrire $\Gamma \vdash b_i : S$, $\Gamma \vdash A' : Proc(t)$, $\Gamma \vdash a : MChan(S)$ et $\Gamma \vdash a^{\eta}\{b\}.A' : Proc(u)$ avec $u \geq t + k \cdot e$ et $S = Seal(\_, e)$. Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : Proc(t)$, nous avons $\Gamma \vdash B' : Proc(t)$. En appliquant de nouveau T-PUSH avec toutes ces hypothèses, nous déduisons $\Gamma \vdash B : U$.

Le cas symétrique se traite de la même manière. □

B.1.3 Ressources

La définition de l'utilisation des ressources est presque identique dans les Seals et dans les CA. Les preuves sont, elles aussi, presque identiques.

B.2 Théorèmes

B.2.1 Réduction du sujet

Prouvons par induction structurelle sur une preuve de $A \longrightarrow B$ que "Pour tout processus A et tout environnement Γ, si $\Gamma \vdash A : Proc(t_A)$ et si $A \longrightarrow B$, alors $\Gamma \vdash B : Proc(t_A)$." Dans tout ce qui suit, nous omettrons généralement les applications de la règle T-NAME et nous factoriserons les applications de T-RES.

R-Comm-Local – Cas d'initialisation

$$\begin{cases} A & = & \overline{a}^{*}(b).P \mid a^{*}(c).Q \\ B & = & P \mid Q\{c \leftarrow b\} \ . \end{cases}$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\overline{a}^*(b).P$			
	Si	$P : U_P$	
		$a : IChan(W)$	
		$b : W$	
$\Rightarrow \Gamma \vdash \overline{a}^*(b).P :$		U_P	Par T-SND
Typage de $a^*(c).Q$			
	Si	$\Gamma, c : W \vdash Q : U_Q$	
$\Gamma \vdash a :$		$IChan(W)$	Cf. plus haut
$\Rightarrow \Gamma \vdash a^*(c).Q :$		U_Q	Par T-RCV
Typage de $\overline{a}^*(b).P \mid a^*(c).Q$			
$\Gamma \vdash \overline{a}^*(b).P :$		U_P	Cf. plus haut
$\Gamma \vdash a^*(c).Q :$		U_Q	Cf. plus haut
	Si	$U_P = Proc(t_P)$	
		$U_Q = Proc(t_Q)$	
		$u \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash \overline{a}^*(b).P \mid a^*(c).Q :$		$Proc(u)$	Par T-PAR

○

Le type le plus général de A est donc $Proc(u)$, avec les conditions nécessaires (pas nécessairement minimales) suivantes (nous avons supprimé les variables intermédiaires inutiles) :

$$\begin{cases} \Gamma \vdash & P : Proc(t_P) \\ \Gamma, c : W \vdash & Q : Proc(t_Q) \\ \Gamma \vdash & a : IChan(W) \\ \Gamma \vdash & b : W \\ & u \geq t_P + t_Q \end{cases}$$

Nous allons prouver que, sous ces hypothèses, $\Gamma \vdash B : Proc(u)$.

Typage de $Q\{c \leftarrow b\}$		
$\Gamma, c : W \vdash Q :$	$Proc(t_Q)$	Par hypothèse
$\Gamma \vdash b :$	W	Par hypothèse
$\Rightarrow \Gamma \vdash Q\{c \leftarrow b\} :$	$Proc(t_Q)$	Par *Substitution*
Typage de $P \mid Q\{c \leftarrow b\}$		
$\Gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
$\Gamma \vdash Q\{c \leftarrow b\} :$	$Proc(t_Q)$	Cf. plus haut
Comme	$u \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash P \mid Q\{c \leftarrow b\} :$	$Proc(u)$	Par T-PAR

○

Ce qui prouve le cas.

R-Comm-Down – Cas d'initialisation

$$\begin{cases} A & = & \overline{a}^m(b).P \mid m\left[(\nu \overrightarrow{z} : \overrightarrow{W})(a^\uparrow(c).Q \mid R)\right] \\ B & = & P \mid m\left[(\nu \overrightarrow{z} : \overrightarrow{W})(Q\{c \leftarrow b\} \mid R)\right] \ . \end{cases}$$

où $a \notin \overrightarrow{z}$, $b \notin \overrightarrow{z}$

Dans ce qui suit, nous noterons Δ pour $\Gamma, \overrightarrow{z} : \overrightarrow{W}$.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $a^{\uparrow}(c).Q$			
	Si	$\Delta \vdash a : IChan(W)$	
		$\Delta, c : W \vdash Q : Proc(t_Q)$	
$\Rightarrow \Delta \vdash a^{\uparrow}(c).Q :$		$Proc(t_Q)$	Par T-Rcv
Typage de $a^{\uparrow}(c).Q \mid R$			
$\Delta \vdash a^{\uparrow}(c).Q :$		$Proc(t_Q)$	Cf. plus haut
	Si	$\Delta \vdash R : Proc(t_R)$	
		$u_1 \geq t_Q + t_R$	
$\Rightarrow \Delta \vdash a^{\uparrow}(c).Q \mid R :$		$Proc(u_1)$	Par T-Par
Typage de $(\nu \overrightarrow{z} : \overrightarrow{W})(a^{\uparrow}(c).Q \mid R)$			
$\Delta \vdash a^{\uparrow}(c).Q \mid R :$		$Proc(u_1)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu \overrightarrow{z} : \overrightarrow{W})(a^{\uparrow}(c).Q \mid R) :$		$Proc(u_1)$	Par T-Res
Typage de $m[\cdots]$			
$\Gamma \vdash (\nu \overrightarrow{z} : \overrightarrow{W})(a^{\uparrow}(c).Q \mid R) :$		$Proc(u_1)$	Cf. plus haut
	Si	$\Gamma \vdash m : Seal(s_m, e_m)$	
		$u_1 = s_m$	
		$u_2 \geq e_m$	
$\Rightarrow \Gamma \vdash m[\cdots] :$		$Proc(u_2)$	Par T-Seal
Typage de $\overline{a}^m(b).P$			
	Si	$\Gamma \vdash P : Proc(t_P)$	
		$a : IChan(W)$	
		$b : W$	
$\Rightarrow \Gamma \vdash \overline{a}^m(b).P :$		$Proc(t_P)$	Par T-Snd
Typage de $\overline{a}^m(b).P \mid m[\cdots]$			
$\Gamma \vdash m[\cdots] :$		$Proc(e_2)$	Cf. plus haut
$\Gamma \vdash \overline{a}^m(b).P :$		$Proc(t_P)$	Cf. plus haut
	Si	$u_3 \geq u_2 + t_P$	
$\Rightarrow \Gamma \vdash \overline{a}^m(b).P \mid m[\cdots] :$		$Proc(u_3)$	Par T-Par

○

Par suite, nous avons $\Gamma \vdash A : Proc(u_3)$ avec

$$\begin{cases} \Delta \vdash & a : IChan(W) \\ \Delta, c : W \vdash & Q : Proc(t_Q) \\ \Delta \vdash & R : Proc(t_R) \\ \Gamma \vdash & m : Seal(s_m, e_m) \\ & s_m \geq t_Q + t_R \\ \Gamma \vdash & P : Proc(t_P) \\ \Gamma \vdash & a : IChan(W) \\ \Gamma \vdash & b : W \\ & u_3 \geq e_m + t_P \end{cases}$$

Prouvons sous ces hypothèses que $\Gamma \vdash B : Proc(u_3)$.

Typage de b

$\Gamma \vdash b :$		W	Par hypothèse
$\Rightarrow \Delta \vdash b :$		W	Par *Affaiblissement*
Typage de $Q\{c \leftarrow b\}$			
$\Delta, c : W \vdash Q :$		$Proc(t_Q)$	Par hypothèse
$\Delta \vdash b :$		$WCf.plushaut$	
$\Rightarrow \Delta \vdash Q\{c \leftarrow b\} :$		$Proc(t_Q)$	Par *Substitution*
Typage de $Q\{c \leftarrow b\} \mid R$			
$\Delta \vdash Q\{c \leftarrow b\} :$		$Proc(t_Q)$	Cf. plus haut
$\Delta \vdash R :$		$Proc(t_R)$	Par hypothèse
	Comme	$s_m \geq t_Q + t_R$	
$\Rightarrow \Delta \vdash Q\{c \leftarrow b\} \mid R :$		$Proc(s_m)$	Par T-Par
Typage de $(\nu \overrightarrow{z} : \overrightarrow{W})(Q\{c \leftarrow b\} \mid R)$			
$\Delta \vdash Q\{c \leftarrow b\} \mid R :$		$Proc(s_m)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu \overrightarrow{z} : \overrightarrow{W})(Q\{c \leftarrow b\} \mid R) :$		$Proc(s_m)$	Par T-Res
Typage de $m\left[(\nu \overrightarrow{z} : \overrightarrow{W})(Q\{c \leftarrow b\} \mid R)\right]$			
$\Gamma \vdash (\nu \overrightarrow{z} : \overrightarrow{W})(Q\{c \leftarrow b\} \mid R) :$		$Proc(s_m)$	Cf. plus haut
$\Gamma \vdash m :$		$Seal(s_m, e_m)$	Par hypothèse
$\Rightarrow \Gamma \vdash m[\cdots] :$		$Proc(e_m)$	Par T-Seal
Typage de $P \mid m[\cdots]$			
$\Gamma \vdash P :$		$Proc(t_P)$	Par hypothèse
$\Gamma \vdash m[\cdots] :$		$Proc(e_m)$	Cf. plus haut
	Comme	$u_3 \geq e_m + t_P$	
$\Rightarrow \Gamma \vdash P \mid m[\cdots] :$		$Proc(u_3)$	Par T-Par

○

Ce qui prouve le cas.

R-Comm-Up – Cas d'initialisation

Notons

$$\begin{cases} A & = & a^m(c).P \mid m\left[(\nu \overrightarrow{z : W_z})(\overline{a}^{\uparrow}(b).Q \mid R)\right] \\ B & = & (\nu \overrightarrow{z : W_z} \cap \{b : W\})(P\{c \leftarrow b\} \mid m\left[(\nu \overrightarrow{z : W_z} \backslash \{b : W\})(Q \mid R)\right]) \ . \end{cases}$$

où $a \notin \overrightarrow{z}$

Dans ce qui suit, nous noterons Δ pour $\Gamma, \overrightarrow{z} : \overrightarrow{W_z}$.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\overline{a}^{\uparrow}(b).Q$			
	Si	$\Delta \vdash Q : Proc(t_Q)$	
		$a : IChan(W)$	
		$b : W$	
$\Rightarrow \Delta \vdash \overline{a}^{\uparrow}(b).Q :$		$Proc(t_Q)$	Par T-Snd
Typage de $\overline{a}^{\uparrow}(b).Q \mid R$			
	Si	$\Delta \vdash R : Proc(t_R)$	
		$u_1 \geq t_Q + t_R$	
$\Rightarrow \Delta \vdash \overline{a}^{\uparrow}(b).Q \mid R :$		$Proc(u_1)$	Par T-Par
Typage de $(\nu \overrightarrow{z : W_z})(\overline{a}^{\uparrow}(b).Q \mid R)$			

$\Delta \vdash \overline{a}^{\uparrow}(b).Q \mid R :$		$Proc(u_1)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu \overrightarrow{z : W_z})(\overline{a}^{\uparrow}(b).Q \mid R) :$		$Proc(u_1)$	Par T-Res
Typage de $m\left[(\nu \overrightarrow{z : W_z})(\overline{a}^{\uparrow}(b).Q \mid R)\right]$			
$\Gamma \vdash (\nu \overrightarrow{z})(\overline{a}^{\uparrow}(b).Q \mid R) :$		$Proc(u_1)$	Cf. plus haut
	Si	$m : Seal(s_m, e_m)$	
		$u_1 = s_m$	
		$u_2 \geq e_m$	
$\Rightarrow \Gamma \vdash m[\cdots] :$		$Proc(u_2)$	Par T-Seal
Typage de $a^m(c).P$			
	Si	$\Gamma, c : W \vdash P : Proc(t_P)$	
$\Gamma \vdash a :$		$IChan(W)$	Cf. plus haut
$\Rightarrow \Gamma \vdash a^m(c).P :$		$Proc(t_P)$	Par T-Rcv
Typage de $a^m(c).P \mid m\left[(\nu \overrightarrow{z : W_z})(\overline{a}^{\uparrow}(b).Q \mid R)\right]$			
$\Gamma \vdash m[\cdots] :$		$Proc(u_2)$	Cf. plus haut
$\Gamma \vdash a^m(c).P :$		$Proc(t_P)$	Cf. plus haut
	Si	$u_3 \geq u_2 + t_P$	
$\Rightarrow \Gamma \vdash a^m(c).P \mid m[\cdots] :$		$Proc(u_3)$	Par T-Par

○

Le type le plus général de A est donc $Proc(u_3)$, avec

$$\begin{cases} \Delta \vdash & Q : Proc(t_Q) \\ & a : IChan(W) \\ & b : W \\ \Delta \vdash & R : Proc(t_R) \\ \Gamma \vdash & m : Seal(s_m, e_m) \\ & s_m \geq t_Q + t_R \\ \Gamma, c : W \vdash & P : Proc(t_P) \\ & u_3 \geq e_m + t_P \end{cases}$$

Typons B à partir de ces hypothèses. Nous noterons Θ pour $\Gamma, \overrightarrow{z : W_z} \cap \{b : W\}$.

Typage de $Q \mid R$			
$\Delta \vdash Q :$		$Proc(t_Q)$	Par hypothèse
$\Delta \vdash R :$		$Proc(t_R)$	Par hypothèse
	Comme	$s_m \geq t_Q + t_R$	
$\Rightarrow \Delta \vdash Q \mid R :$		$Proc(s_m)$	Par T-Par
Typage de $(\nu \overrightarrow{z : W_z} \backslash \{b : W\})(Q \mid R)$			
$\Delta \vdash Q \mid R :$		$Proc(s_m)$	Par T-Par
$\Rightarrow \Theta \vdash \overrightarrow{z} \backslash \{b : W\}(Q \mid R) :$		$Proc(s_m)$	Par T-Res
Typage de $m\left[(\nu \overrightarrow{z : W_z} \backslash \{b\})(Q \mid R)\right]$			
$\Theta \vdash (\nu \overrightarrow{z : W_z} \backslash \{b\})(Q \mid R) :$		$Proc(s_m)$	Cf. plus haut
$\Rightarrow \Theta \vdash m\left[(\nu \overrightarrow{z : W_z} \backslash \{b\})(Q \mid R)\right] :$		$Proc(e_m)$	Par T-Seal
Typage de P			
$\Gamma, c : W \vdash P :$		$Proc(t_P)$	Par hypothèse
$\Rightarrow \Theta, c : W \vdash P :$		$Proc(t_P)$	Par *Affaiblissement*
Typage de $P\{c \leftarrow b\}$			

$\Theta, c : W \vdash P :$		$Proc(t_P)$	Cf. plus haut
$\Theta \vdash b : W$		ParT-Name	
$\Rightarrow \Theta \vdash P\{c \leftarrow b\} :$		$Proc(t_P)$	Par *Substitution*
Typage de $P\{c \leftarrow b\} \mid m[\cdots]$			
$\Theta \vdash P\{c \leftarrow b\} :$		$Proc(t_P)$	Cf. plus haut
$\Theta \vdash m[\cdots] :$		$Proc(e_m)$	Cf. plus haut
	Comme	$u_3 \geq t_P + e_m$	
$\Rightarrow \Theta \vdash P\{c \leftarrow b\} \mid m[\cdots] :$		$Proc(u_3)$	Par T-Par
Typage de $(\nu \overrightarrow{z : W_z} \cap \{b : W\})(P\{c \leftarrow b\} \mid m[\cdots])$			
$\Theta \vdash P\{c \leftarrow b\} \mid m[\cdots] :$		$Proc(u_3)$	Cf. plus haut
$\Rightarrow \Gamma \vdash B :$		$Proc(u_3)$	Par T-Res

○

Ce qui prouve le cas.

R-Move-Local – Cas d'initialisation

Notons

$$\begin{cases} A &= \overline{a}^*\{b\}.P \mid a^*\{c_1, \ldots, c_n\}.Q \mid b[R] \\ B &= P \mid Q \mid c_1[R] \mid \ldots \mid c_n[R] \end{cases}$$

Le typage le plus général de A dans Γ s'écrit

Typage de $\overline{a}^*\{b\}.P$			
	Si	$P : Proc(t_P)$	
		$a : MChan(Seal(s_b, e_b))$	
		$b : Seal(s_b, e_b)$	
		$u_1 + e_b \geq t_P$	
$\Rightarrow \Gamma \vdash \overline{a}^*\{b\}.P :$		$Proc(u_1)$	Par T-Pop
Typage de $a^*\{c_1, \ldots, c_n\}.Q$			
$\Gamma \vdash a :$		$MChan(Seal(s_b, e_b))$	Cf. plus haut
	Si	$Q : Proc(t_Q)$	
		$c_1 : Seal(s_b, e_b)$	
		$c_2 : Seal(s_b, e_b)$	
		$\vdots$	
		$c_n : Seal(s_b, e_b)$	
		$u_2 \geq t_Q + n \cdot e_b$	
$\Rightarrow \Gamma \vdash a^*\{c_1, \ldots, c_n\}.Q :$		$Proc(u_2)$	Par T-Push
Typage de $\overline{a}^*\{b\}.P \mid a^*\{c_1, \ldots, c_n\}.Q$			
$\Gamma \vdash \overline{a}^*\{b\}.P :$		$Proc(u_1)$	Cf. plus haut
$\Gamma \vdash a^*\{c_1, \ldots, c_n\}.Q :$		$Proc(u_2)$	Cf. plus haut
	Si	$u_3 \geq u_1 + u_2$	
$\Rightarrow \Gamma \vdash \overline{a}^*\{b\}.P \mid a^*\{c_1, \ldots, c_n\}.Q :$		$Proc(u_3)$	Par T-Par
Typage de $b[R]$			
$\Gamma \vdash b :$		$Seal(s_b, e_b)$	Cf. plus haut
	Si	$R : Proc(t_R)$	
		$t_R = s_b$	
		$u_4 \geq e_b$	
$\Rightarrow \Gamma \vdash b[R] :$		$Proc(u_4)$	Par T-Seal
Typage de A			

$\Gamma \vdash \overline{a}^*\{b\}.P \mid a^*\{c_1, \ldots, c_n\}.Q :$	$Proc(u_3)$		Cf. plus haut
$\Gamma \vdash b\,[R] :$	$Proc(u_4)$		Cf. plus haut
	Si $u_5 \geq u_4 + u_3$		
$\Rightarrow \Gamma \vdash A : Proc(u_5)$			

○

Le type le plus général de A est donc $Proc(u_5)$, avec les condition suivantes

$$\begin{cases} \Gamma \vdash & P : Proc(t_P) \\ \Gamma \vdash & a : MChan(Seal(s_b, e_b)) \\ \Gamma \vdash & b : Seal(s_b, e_b) \\ \Gamma \vdash & Q : Proc(t_Q) \\ \Gamma \vdash & c_1 : Seal(s_b, e_b) \\ \Gamma \vdash & c_2 : Seal(s_b, e_b) \\ & \vdots \\ \Gamma \vdash & c_n : Seal(s_b, e_b) \\ \Gamma \vdash & R : Proc(s_b) \\ & u_5 \geq t_P + t_Q + n \cdot e_b \end{cases}$$

Prouvons sous ces hypothèses que $\Gamma \vdash B : Proc(u_5)$.

Typage de $c_i\,[R]$			
$\Gamma \vdash R :$	$Proc(s_b)$		Par hypothèse
$\Gamma \vdash c_i :$	$Seal(s_b, e_b)$		Par hypothèse
$\Rightarrow \Gamma \vdash c_i\,[R] :$	$Proc(e_b)$		Par T-Seal
Typage de $c_1\,[R] \mid c_2\,[R] \mid \cdots \mid c_n\,[R]$			
$\Gamma \vdash c_i\,[R] :$	$Proc(e_b)$		Cf. plus haut
$\Rightarrow \Gamma \vdash c_1\,[R] \mid c_2\,[R] \mid \cdots \mid c_n\,[R] :$	$Proc(n \cdot e_b)$		Par T-Par
Typage de $P \mid Q$			
$\Gamma \vdash P :$	$Proc(t_P)$		Par hypothèse
$\Gamma \vdash Q :$	$Proc(t_Q)$		Par hypothèse
$\Rightarrow \Gamma \vdash P \mid Q :$	$Proc(t_P + t_Q)$		Par T-Par
Typage de $P \mid Q \mid c_1\,[R] \mid \ldots \mid c_n\,[R]$			
$\Gamma \vdash c_1\,[R] \mid c_2\,[R] \mid \cdots \mid c_n\,[R] :$	$Proc(n \cdot e_b)$		Cf. plus haut
$\Gamma \vdash P \mid Q :$	$Proc(t_P + t_Q)$		Cf. plus haut
	Comme	$u_5 \geq t_P + t_Q + n \cdot e_b$	
$\Rightarrow \Gamma \vdash P \mid Q \mid c_1\,[R] \mid \ldots \mid c_n\,[R] :$	$Proc(u_5)$		Par T-Par

○

Ce qui prouve le cas.

R-Move-Down – Cas d'initialisation

Notons

$$\begin{cases} A &= \overline{a}^m\{b\}.P \mid m\left[(\nu \overrightarrow{z : W_z})(a^{\uparrow}\{c_1, \ldots, c_n\}.Q \mid R)\right] \mid b\,[S] \\ B &= P \mid m\left[(\nu \overrightarrow{z : W_z})(Q \mid R \mid c_1\,[S] \mid \ldots \mid c_n\,[S])\right] \end{cases}$$

Avec $a \notin \overrightarrow{z}$, $fv(R) \cap \overrightarrow{z} = \emptyset$.

Dans ce qui suit, nous noterons Δ pour $\Gamma, \overrightarrow{z} : \overrightarrow{W}$.
Le typage le plus général de A dans Γ s'écrit

	Typage de $\overline{a}^m\{b\}.P$		
	Si	$P : Proc(t_P)$	
		$a : MChan(Seal(s_b, e_b))$	
		$b : Seal(s_b, e_b)$	
		$u_1 + e_b \geq t_P$	
$\Rightarrow \Gamma \vdash \overline{a}^m\{b\}.P :$		$Proc(u_1)$	Par T-Pop
	Typage de a		
$\Gamma \vdash a :$		$MChan(Seal(s_b, e_b))$	Cf. plus haut
$\Rightarrow \Delta \vdash a :$		$MChan(Seal(s_b, e_b))$	Par *Affaiblissement*
	Typage de $a^\uparrow\{c_1, \ldots, c_n\}.Q$		
	Si	$\Delta \vdash Q : Proc(t_Q)$	
		$c_1 : Seal(s_b, e_b)$	
		$c_2 : Seal(s_b, e_b)$	
		$\vdots$	
		$c_n : Seal(s_b, e_b)$	
		$u_2 \geq n \cdot e_b + t_Q$	
$\Delta \vdash a :$		$MChan(Seal(s_b, e_b))$	Cf. plus haut
$\Rightarrow \Delta \vdash a^\uparrow\{c_1, \ldots, c_n\}.Q :$		$Proc(u_2)$	Par T-Push
	Typage de $a^\uparrow\{c_1, \ldots, c_n\}.Q \mid R$		
$\Delta \vdash a^\uparrow\{c_1, \ldots, c_n\}.Q :$		$Proc(u_2)$	Cf. plus haut
	Si	$\Delta \vdash R : Proc(t_R)$	
		$u_3 \geq u_2 + t_R$	
$\Rightarrow \Delta \vdash a^\uparrow\{c_1, \ldots, c_n\}.Q \mid R :$		$Proc(u_3)$	Par T-Par
	Typage de $(\nu z : \overrightarrow{W_z})(a^\uparrow\{c_1, \ldots, c_n\}.Q \mid R)$		
$\Delta \vdash a^\uparrow\{c_1, \ldots, c_n\}.Q \mid R :$		$Proc(u_3)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu z : \overrightarrow{W_z})(\cdots) :$		$Proc(u_3)$	Par T-Name
	Typage de $m\left[(\nu z : \overrightarrow{W_z})(a^\uparrow\{c_1, \ldots, c_n\}.Q \mid R)\right]$		
$\Gamma \vdash (\nu z : \overrightarrow{W_z})(\cdots) :$		$Proc(u_3)$	Cf. plus haut
	Si	$\Gamma \vdash m : Seal(s_m, e_m)$	
		$u_4 \geq e_m$	
		$u_3 = s_m$	
$\Rightarrow \Gamma \vdash m[\cdots] :$		$Proc(u_4)$	Par T-Seal
	Typage de $b[S]$		
	Si	$S : Proc(s_b)$	
		$u_5 \geq e_b$	
$\Gamma \vdash b :$		$Seal(s_b, e_b)$	Cf. plus haut
$\Rightarrow \Gamma \vdash b[S] :$		$Proc(u_5)$	Par T-Amb
	Typage de $m[\cdots] \mid b[S]$		
$\Gamma \vdash m[\cdots] :$		$Proc(u_4)$	Cf. plus haut
$\Gamma \vdash b[S] :$		$Proc(u_5)$	Cf. plus haut
	Si	$u_6 \geq u_4 + u_5$	
$\Rightarrow \Gamma \vdash m[\cdots] \mid b[S] :$		$Proc(u_6)$	Par T-Par
	Typage de $\overline{a}^m\{b\}.P \mid m[\cdots] \mid b[S]$		
$\Gamma \vdash \overline{a}^m\{b\}.P :$		$Proc(u_1)$	Cf. plus haut
$\Gamma \vdash m[\cdots] \mid b[S] :$		$Proc(u_6)$	Cf. plus haut
	Si	$u_7 \geq u_6 + u_1$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u_7)$	Par T-Par

○

Le type le plus général de A est donc $Proc(u_7)$ avec

$$\begin{cases} \Gamma \vdash & P : Proc(t_P) \\ \Gamma \vdash & a : MChan(Seal(s_b, e_b)) \\ \Gamma \vdash & b : Seal(s_b, e_b) \\ \Delta \vdash & Q : Proc(t_Q) \\ \Delta \vdash & R : Proc(t_R) \\ \Delta \vdash & c_i : Seal(s_b, e_b) \\ \Gamma \vdash & m : Seal(s_m, e_m) \\ & s_m \geq n \cdot e_b + t_Q + t_R \\ \Gamma \vdash & S : Proc(s_b) \\ & u_7 \geq e_m + t_P \end{cases}$$

Avec ces hypothèses, nous pouvons typer B dans Γ

	Typage de S		
	$\Gamma \vdash S :$	$Proc(s_b)$	Par hypothèse
$\Rightarrow$	$\Delta \vdash S :$	$Proc(s_b)$	Par *Affaiblissement*
	Typage de $c_i\,[S]$		
	$\Delta \vdash S :$	$Proc(s_b)$	Cf. plus haut
	$\Delta \vdash c_i :$	$Seal(s_b, e_b)$	Par hypothèse
$\Rightarrow$	$\Delta \vdash c_i\,[S] :$	$Proc(e_b)$	Par T-Seal
	Typage de $c_1\,[S] \mid \cdots \mid c_n\,[S]$		
	$\Delta \vdash c_i\,[S] :$	$Proc(e_b)$	Cf. plus haut
$\Rightarrow$	$\Delta \vdash c_1\,[S] \mid \cdots \mid c_n\,[S] :$	$Proc(n \cdot e_b)$	Par T-Par
	Typage de $R \mid c_1\,[S] \mid \cdots \mid c_n\,[S]$		
	$\Delta \vdash R :$	$Proc(t_R)$	Par hypothèse
	$\Delta \vdash c_1\,[S] \mid \cdots \mid c_n\,[S] :$	$Proc(n \cdot e_b)$	Cf. plus haut
$\Rightarrow$	$\Delta \vdash R \mid c_1\,[S] \cdots \mid c_n\,[S] :$	$Proc(t_R + n \cdot e_b)$	Par T-Par
	Typage de $Q \mid R \mid c_1\,[S] \mid \cdots \mid c_n\,[S]$		
	$\Delta \vdash R \mid c_1\,[S] \cdots c_n\,[S] :$	$Proc(t_R + n \cdot e_b)$	Cf. plus haut
	Comme	$s_m \geq t_Q + t_R + n \cdot e_b$	
$\Rightarrow$	$\Delta \vdash Q \mid R \mid c_1\,[S] \cdots c_n\,[S] :$	$Proc(s_m)$	Par T-Par
	Typage de $(\nu \overrightarrow{z : W_z}(Q \mid R \mid c_1\,[S] \mid \cdots \mid c_n\,[S]))$		
	$\Delta \vdash Q \mid R \mid c_1\,[S] \cdots c_n\,[S] :$	$Proc(s_m)$	Cf. plus haut
$\Rightarrow$	$\Gamma \vdash (\overrightarrow{\nu z : W_z}) \cdots :$	$Proc(s_m)$	Par T-Res
	Typage de $m \mid (\overrightarrow{\nu z : W_z}) \cdots$		
	$\Gamma \vdash (\overrightarrow{\nu z : W_z}) \cdots :$	$Proc(s_m)$	Cf. plus haut
	$\Gamma \vdash m :$	$Site(s_m, e_m)$	Par hypothèse
$\Rightarrow$	$\Gamma \vdash m\,[\cdots] :$	$Proc(e_m)$	Par T-Seal
	Typage de $P \mid m \mid (\overrightarrow{\nu z : W_z}) \cdots$		
	$\Gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
	$\Gamma \vdash m\,[\cdots] :$	$Proc(e_m)$	Cf. plus haut
	Comme	$u_7 \geq e_m + t_P$	
$\Rightarrow$	$\Gamma \vdash B :$	$Proc(u_7)$	Par T-Par

○

Ce qui prouve le cas.

R-Move-Up – Cas d'initialisation

Notons

$$\begin{cases} A & = a^m\{c_1, \cdots, c_n\}.P \mid m\left[(\nu\overrightarrow{z : W_z})(\overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S])\right] \\ B & = P \mid c_1\,[S] \mid \ldots \mid c_n\,[S] \mid m\left[(\nu\overrightarrow{z : W_z})(Q \mid R)\right] \end{cases}$$

avec $a \notin \overrightarrow{z}$ et $fv(S) \cap \overrightarrow{z} = \emptyset$.
Dans ce qui suit, nous noterons Δ pour $\Gamma, \overrightarrow{z} : \overrightarrow{W}$.
Le typage le plus général de A dans Γ s'écrit

	Typage de $a^m\{c_1, \cdots, c_n\}.P$		
	Si	$\Gamma \vdash P : Proc(t_P)$	
		$c_i : Seal(s_b, e_b)$	
		$a : MChan(Seal(s_b, e_b))$	
		$u_1 \geq t_P + n \cdot e_b$	
$\Rightarrow \Gamma \vdash a^m\{c_1, \cdots, c_n\}.P :$		$Proc(u_1)$	Par T-Push
	Typage de a		
$\Gamma \vdash a :$		$MChan(Seal(s_b, e_b))$	Cf. plus haut
$\Rightarrow \Delta \vdash a :$		$MChan(Seal(s_b, e_b))$	Par *Affaiblissement*
	Typage de $\overline{a}^{\uparrow}\{b\}.Q$		
	Si	$\Delta \vdash Q : Proc(t_Q)$	
		$b : Seal(s_b, e_b)$	
		$u_2 + e_b \geq t_Q$	
$\Delta \vdash a :$		$MChan(Seal(s_b, e_b))$	Cf. plus haut
$\Rightarrow \Delta \vdash \overline{a}^{\uparrow}\{b\}.Q :$		$Proc(u_2)$	Par T-Pop
	Typage de $\overline{a}^{\uparrow}\{b\}.Q \mid R$		
$\Delta \vdash \overline{a}^{\uparrow}\{b\}.Q :$		$Proc(u_2)$	Cf. plus haut
	Si	$\Delta \vdash R : Proc(t_R)$	
		$u_3 \geq u_2 + t_R$	
$\Rightarrow \Delta \vdash \overline{a}^{\uparrow}\{b\}.Q \mid R :$		$Proc(u_3)$	Par T-Par
	Typage de $b\,[S]$		
	Si	$\Delta \vdash S : Proc(t_S)$	
		$s_b = t_S$	
		$u_4 \geq e_b$	
$\Delta \vdash b :$		$Seal(s_b, e_b)$	Cf. plus haut
$\Rightarrow \Delta \vdash b\,[S] :$		$Proc(u_4)$	Par T-Seal
	Typage de $\overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S]$		
$\Delta \vdash \overline{a}^{\uparrow}\{b\}.Q \mid R :$		$Proc(u_3)$	Cf. plus haut
$\Delta \vdash b\,[S] :$		$Proc(u_4)$	Cf. plus haut
	Si	$u_5 \geq u_3 + u_4$	
$\Rightarrow \Delta \vdash \overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S] :$		$Proc(u_5)$	Par T-Par
	Typage de $(\nu\overrightarrow{z : W_z})(\overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S])$		
$\Delta \vdash \overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S] :$		$Proc(u_5)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu\overrightarrow{z : W_z})\cdots :$		$Proc(u_5)$	Par T-Res
	Typage de $m\left[(\nu\overrightarrow{z : W_z})(\overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S])\right]$		
$\Gamma \vdash (\nu\overrightarrow{z : W_z})\cdots :$		$Proc(u_5)$	Cf. plus haut
	Si	$m : Seal(s_m, e_m)$	
		$u_5 = s_m$	
		$u_6 \geq e_m$	
$\Rightarrow \Gamma \vdash m[\cdots] :$		$Proc(u_6)$	Par T-Amb

Typage de $a^m\{c_1,\cdots,c_n\}.P \mid m\left[(\nu\overrightarrow{z:W_z})(\overline{a}^{\uparrow}\{b\}.Q \mid R \mid b\,[S])\right]$		
$\Gamma \vdash a^m\{c_1,\cdots,c_n\}.P :$	$Proc(u_1)$	Cf. plus haut
$\Gamma \vdash m\,[\cdots] :$	$Proc(u_6)$	Cf. plus haut
Si	$u_7 \geq u_6 + u_1$	
$\Rightarrow \Gamma \vdash A :$	$Proc(u_7)$	Par T-Par

○

Ainsi, le type le plus général pour A dans Γ est $Proc(u_7)$, avec

$$\begin{cases} \Gamma \vdash & P : Proc(t_P) \\ \Gamma \vdash & c_i : Seal(s_b, e_b) \\ \Gamma \vdash & a : MChan(Seal(s_b, e_b)) \\ \Gamma \vdash & m : Seal(s_m, e_m) \\ \Delta \vdash & Q : Proc(t_Q) \\ \Delta \vdash & R : Proc(t_R) \\ \Delta \vdash & S : Proc(s_b) \\ \Delta \vdash & b : Seal(s_b, e_b) \\ & s_m \geq t_Q + t_R \\ & u_7 \geq e_m + t_P + n \cdot e_b \end{cases}$$

À l'aide des hypothèses, nous pouvons typer B dans Γ :

Typage de $Q \mid R$		
$\Delta \vdash Q :$	$Proc(t_Q)$	Par hypothèse
$\Delta \vdash R :$	$Proc(t_R)$	Par hypothèse
$\Rightarrow \Delta \vdash Q \mid R :$	$Proc(t_Q + t_R)$	Par T-Par
Typage de $(\nu\overrightarrow{z:W_z})(Q \mid R)$		
$\Delta \vdash Q \mid R :$	$Proc(t_Q + t_R)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu\overrightarrow{z:W_z})(Q \mid R) :$	$Proc(t_Q + t_R)$	Par T-Res
Typage de $m\left[(\nu\overrightarrow{z:W_z})(Q \mid R)\right]$		
$\Gamma \vdash (\nu\overrightarrow{z:W_z})(Q \mid R) :$	$Proc(t_Q + t_R)$	Cf. plus haut
$\Gamma \vdash m :$	$Seal(s_m, e_m)$	Par hypothèse
Comme	$s_m \geq t_Q + t_R$	
$\Rightarrow \Gamma \vdash m\,[\cdots] :$	$Proc(e_m)$	Par T-Seal
Typage de S		
$\Delta \vdash S :$	$Proc(s_b)$	Par hypothèse
Comme	$fv(S) \cap \overrightarrow{z} = \emptyset$	
$\Rightarrow \Gamma \vdash S :$	$Proc(s_b)$	Par *Renforcement*
Typage de $c_i\,[S]$		
$\Gamma \vdash S :$	$Proc(s_b)$	Cf. plus haut
$\Gamma \vdash c_i :$	$Amb(s_b, e_b)$	Par hypothèse
$\Rightarrow \Gamma \vdash c_i\,[S] :$	$Proc(e_b)$	Par T-Seal
Typage de $c_1\,[S] \mid \ldots \mid c_n\,[S]$		
$\Gamma \vdash c_i\,[S] :$	$Proc(e_b)$	Cf. plus haut
$\Rightarrow \Gamma \vdash c_1\,[S] \mid \ldots c_n\,[S] :$	$Proc(n \cdot e_b)$	Par T-Par
Typage de $P \mid c_1\,[S] \mid \ldots \mid c_n\,[S]$		
$\Gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
$\Gamma \vdash c_1\,[S] \mid \ldots c_n\,[S] :$	$Proc(n \cdot e_b)$	Cf. plus haut
$\Rightarrow \Gamma \vdash P \mid c_1\,[S] \mid \ldots \mid c_n\,[S] :$	$Proc(t_P + n \cdot e_b)$	Par T-Par

Typage de $P \mid c_1[S] \mid \ldots \mid c_n[S] \mid m[\cdots]$			
$\Gamma \vdash P \mid c_1[S] \mid \ldots \mid c_n[S] :$		$Proc(t_P + n \cdot e_b)$	Cf. plus haut
$\Gamma \vdash m[\cdots] :$		$Proc(e_m)$	Cf. plus haut
	Comme	$u_7 \geq e_m + t_P + n \cdot e_b$	
$\Rightarrow \Gamma \vdash B :$		$Proc(u_7)$	Par T-PAR

○

Ce qui prouve le cas.

Les autres cas

Les cas R-REC, R-RES, R-SEAL, R-PAR et R-STRUCT sont identiques à leurs contreparties des Controlled Ambients. □

Annexe C

Kells

C.1 Lemmes – premier système de types

C.1.1 Substitutions et renommages

Sous-lemme 11 (Substitution d'un nom)
Si $\Gamma, n : T \vdash P : U$ *et* $\Gamma \vdash m : T$, *alors* $\Gamma \vdash P\{n \leftarrow m\} : U$.

Ce sous-lemme se prouve presque comme le sous-lemme 1 dans le cas des ambients. La règle T-Name remplace son homologue T-AmbName et nous supposons, comme spécifié dans les propriétés du système de types, que le type d'un motif est compatible avec les substitutions.

Sous-lemme 12 (Renommage d'une variable liée)
Si $\Gamma \vdash (\nu n : T)P : U$ *et* $m \notin fv(P)$, *alors* $\Gamma \vdash (\nu m : T)P\{n \leftarrow m\} : U$.

Comme T-Res est la seule règle qui permette de typer $(\nu n : T)P$ et comme $\Gamma \vdash (\nu n : T)P : U$, nous avons $\Gamma, n : T \vdash P : U$. Comme $m \notin fv(P)$, d'après le lemme de renforcement, nous pouvons supposer $m \notin \Gamma$. D'après le lemme d'affaiblissement, nous avons alors aussi $\Gamma, n : T, m : T \vdash P : U$. Or, $\Gamma, n : T, m : T \vdash m : T$. D'après le lemme de substitution, nous avons donc $\Gamma, n : T, m : T \vdash P\{n \leftarrow m\} : U$. Or, n n'est pas libre dans $P\{n \leftarrow m\}$ donc, d'après le lemme de renforcement, nous avons $\Gamma, m : T \vdash P\{n \leftarrow m\} : U$. D'après T-Res, nous pouvons conclure que $\Gamma \vdash (\nu m : T)P\{n \leftarrow m\} : U$. Ce qui prouve le sous-lemme.

C.1.2 Congruence structurelle

Sous-lemme 13 (Les bons types supportent la congruence)
Soient deux processus A *et* B *tels que* $A \equiv B$. *Si* $\Gamma \vdash A : U$, *alors* $\Gamma \vdash B : U$ *et si* $\Gamma \vdash B : U$ *alors* $\Gamma \vdash A : U$.

Nous prouvons ce sous-lemme par induction structurelle sur une preuve de $A \equiv B$. Chaque cas correspond alors

- soit à une règle de congruence structurelle présentée sur la figure 4.10
- soit à une propriété qui d'équivalence de la relation $\equiv$ (transitivité, symétrie, réflexivité)

- soit à une des propriétés qui font de $(P/\equiv, |, \mathbf{0})$ un monoïde commutatif et associatif (commutativité, associativité, neutralité de $\mathbf{0}$ vis-à-vis de l'opération $|$)
- soit à une propriété de congruence de la relation $\equiv$, illustrées sur la figure C.1.

Les cas Struct-Zero-Res, Struct-Res-Res, Struct-Res-Par, la transitivité, la symétrie, la réflexivité, la commutativité et l'associativité se traitent exactement comme dans les Controlled Ambients. Le cas Struct-Res-Kell est identique à sa contrepartie Struct-Res-Amb et le cas Struct-α est identique à la gestion du renommage.

Restent Struct-Trig et Struct-Context.

La règle Struct-Trig est une conséquence directe du fait que le typage est compatible avec la congruence structurelle.

$$\textsc{Struct-Trig-Congruence}\ \frac{P \equiv Q}{\xi \rhd P \equiv \xi \rhd Q}$$

$$\textsc{Struct-Res}\ \frac{P \equiv Q}{(\nu a : T)P \equiv (\nu a : T)Q} \qquad \textsc{Struct-Par}\ \frac{P \equiv Q}{R|P \equiv R|Q}$$

$$\textsc{Struct-Cell-In}\ \frac{P \equiv Q}{a\,[P]\,.R \equiv a\,[Q]\,.R} \qquad \textsc{Struct-Message-In}\ \frac{P \equiv Q}{a\langle P\rangle.R \equiv a\langle Q\rangle.R}$$

$$\textsc{Struct-Cell-Cont}\ \frac{P \equiv Q}{a\,[R]\,.P \equiv a\,[R]\,.Q}$$

$$\textsc{Struct-Message-Cont}\ \frac{P \equiv Q}{a\langle R\rangle.P \equiv a\langle R\rangle.Q}$$

Fig. C.1 – Propriétés de congruence de la congruence structurelle – Kells.

Cas Struct-Trig-Congruence

Si ce cas se produit, nous avons $A = \xi \rhd A'$ et $B = \xi \rhd B'$ avec $A' \equiv B'$. Comme A est typable et comme la seule règle qui permette de typer A est T-Trig, nous pouvons écrire $\Gamma \vdash \xi : Pat(\gamma)$, $\Gamma, \gamma \vdash A' : Proc(t_P)$ et $\Gamma \vdash A : Proc(t_P)$.

Par hypothèse d'induction, comme $\Gamma, \gamma \vdash A' : Proc(t_P)$ et $A' \equiv B'$, nous avons $\Gamma, \gamma \vdash B' : Proc(t_P)$. En appliquant T-Trig, comme $\Gamma \vdash \xi : Pat(\gamma)$ et $\Gamma, \gamma \vdash B' : Proc(t_P)$, nous déduisons $\Gamma \vdash B : Proc(t_P)$. Ce qui prouve le cas.

Cas Struct-Res

Ce cas reprend Struct-Res des Controlled Ambients. La preuve est identique.

Cas Struct-Par

Ce cas reprend STRUCT-PAR des Controlled Ambients. La preuve est identique.

Cas Struct-Cell-In

Ce cas reprend STRUCT-AMB des Controlled Ambients. La preuve est identique.

Cas Struct-Message-In

Ce cas reprend STRUCT-AMB des Controlled Ambients. La preuve est identique.

Cas Struct-Cell-Cont

Notons $A = a\,[P]\,.A'$ et $B = a\,[P]\,.B'$ où $A' \equiv B'$. Comme A est typable et comme la seule règle qui permette de typer A est T-CELL, nous pouvons écrire $\Gamma \vdash n : Site(s,e)$, $\Gamma \vdash P : Proc(s)$, $\Gamma \vdash A' : Proc(t_Q)$, $\Gamma \vdash n\,[P]\,.A' : Proc(u)$, $u \geq t_Q$ et $u \geq e$.

Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : Proc(t_Q)$, nous pouvons déduire $\Gamma \vdash B' : Proc(t_Q)$. En appliquant de nouveau T-CELL, avec $\Gamma \vdash n : Site(s,e)$, $\Gamma \vdash P : Proc(s)$, $\Gamma \vdash B' : Proc(t_Q)$, $u \geq t_Q$ et $u \geq e$, nous obtenons $\Gamma \vdash n\,[P]\,.A' : Proc(u)$.

Cas Struct-Message-Cont

La preuve est identique au cas STRUCT-CELL-CONT.

C.2 Théorème – premier système de types

C.2.1 Subject Reduction

Grâce au sous-lemme sur le filtrage par motif compatible avec les types

Prouvons par induction structurelle sur une preuve de $A \longrightarrow B$ que "Pour tout processus A et tout environnement Γ, si $\Gamma \vdash A : Proc(t_A)$ et si $A \longrightarrow B$, alors $\Gamma \vdash B : Proc(t_A)$." Dans tout ce qui suit, nous omettrons généralement les applications de la règle T-NAME et nous factoriserons les applications de T-RES.

R-L-Typed – Cas d'initialisation

Notons, dans un environnement Γ,

$$\begin{cases} A &= (\xi \rhd P)|U_1|U_2|U_3 \\ B &= P\theta|V_1|V_2|V_3\ . \end{cases}$$

où

$$
\left\{\begin{array}{ll}
\Gamma \vdash & \xi : Pat(\gamma) \\
\Gamma \vdash & (\xi \rhd P)|U_1|U_2|U_3 : T \\
& \xi \neq \emptyset \\
& \theta = \mathtt{match}_\Gamma(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3)) \\
& \gamma \text{ et } \theta \text{ ont le même domaine de définition} \\
& \forall x, \theta(x) = P_x \Rightarrow \Gamma \vdash P_x : \gamma(x) \\
& \Delta(U_1, M_1, V_1) \\
& \Upsilon(U_2, M_2, V_2) \\
& \Psi(U_3, M_3, V_3) \\
& U_2 \not\rightarrow \\
\Delta(U, M, V) \triangleq & \left\{\begin{array}{lcl} U & = & \Pi_{j\in J} a_j\langle P_j\rangle.Q_j \\ M & = & \Pi_{j\in J} a_j\langle P_j\rangle \\ V & = & \Pi_{j\in J} Q_j \end{array}\right. \\
\Upsilon(U, M, V) \triangleq & \left\{\begin{array}{lcl} U & = & \Pi_{j\in J'} a_j\,[P_j]\,.Q_j \\ M & = & \Pi_{j\in J'} a_j\,[P_j] \\ V & = & \Pi_{j\in J'} Q_j \end{array}\right. \\
\Psi(U, M, V) \triangleq & \left\{\begin{array}{lcl} U & = & \Pi_{j\in J''} a_j\langle R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i\rangle.Q_j \\ M & = & \Pi_{j\in J''}\Pi_{i\in I_j} a_i\langle P_i\rangle^{\downarrow a_j} \\ V & = & \Pi_{j\in J''} a_j\left[R_j \mid \Pi_{i\in I_j} S_i\right].Q_j \end{array}\right.
\end{array}\right.
$$

Pour éviter toute ambiguïté, nous supposerons que l'ensemble J ne contient pas d'entiers. De plus, nous noterons

$$w \triangleq \sum_{j\in J} u_j + \sum_{j\in J'} u_j + \sum_{j\in J''} u_h^j + t_P\ .$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $a_j\langle P_j\rangle.Q_j$			
	Si	$\Gamma \vdash P_j : Proc(s_j)$	
		$Q_j : Proc(t_Q^j)$	
		$a_j : Site(s_j, e_j)$	
		$u_j \geq e_j$	
		$u_j \geq t_Q^j$	
$\Rightarrow \Gamma \vdash a_j\langle P_j\rangle.Q_j :$		$Proc(u_j)$	Par T-Msg
Typage de $\Pi_{j\in J} a_j\langle P_j\rangle.Q_j$			
$\Gamma \vdash a_j\langle P_j\rangle.Q_j :$		$Proc(u_j)$	Cf. plus haut
	Si	$u_1 \geq \sum_{j\in J} u_j$	
$\Rightarrow \Gamma \vdash U_1 :$		$Proc(u_1)$	Par T-Par
Typage de $a_j\,[P_j]\,.Q_j$			
	Si	$\Gamma \vdash P_j : Proc(s_j)$	
		$Q_j : Proc(t_Q^j)$	
		$a_j : Site(s_j, e_j)$	
		$u_j \geq e_j$	
		$u_j \geq t_Q^j$	
$\Rightarrow \Gamma \vdash a_j\,[P_j]\,.Q_j :$		$Proc(u_j)$	Par T-Cell
Typage de $\Pi_{j\in J'} a_j\,[P_j]\,.Q_j$			
$\Gamma \vdash a_j\,[P_j]\,.Q_j :$		$Proc(u_j)$	Cf. plus haut
	Si	$u_2 \geq \sum_{j\in J'} u_j$	
$\Rightarrow \Gamma \vdash U_2 :$		$Proc(u_2)$	Par T-Par

Typage de $a\_i\langle P_i\rangle.S_i$			
	Si	$\Gamma \vdash P_i : Proc(s_i)$	
		$S_i : Proc(t_S^i)$	
		$a_i : Site(s_i, e_i)$	
		$u_i^d \geq e_i$	
		$u_i^d \geq t_S^i$	
$\Rightarrow \Gamma \vdash a\_i\langle P_i\rangle.S_i :$		$Proc(u_i^d)$	Par T-Msg
Typage de $\Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i$			
$\Gamma \vdash a\_i\langle P_i\rangle.S_i :$		$Proc(u_i^d)$	Cf. plus haut
	Si	$u_d^j \geq \sum_{i\in I_j} u_i^d$	
$\Rightarrow \Gamma \vdash \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i :$		$Proc(u_d^j)$	Par T-Par
Typage de $R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i$			
	Si	$R_j : Proc(t_R^j)$	
		$u_R^j \geq t_R^j + u_d^j$	
$\Gamma \vdash \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i :$		$Proc(u_d^j)$	Cf. plus haut
$\Rightarrow \Gamma \vdash R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i :$		$Proc(u_R^j)$	Par T-Par
Typage de $a_j\langle R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i\rangle.Q_j$			
	Si	$a_j : Site(s_j, e_j)$	
		$Q_j : Proc(t_Q^j)$	
		$u_R^j = s_j$	
		$u_h^j \geq e_j$	
		$u_h^j \geq t_Q^j$	
$\Gamma \vdash R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i :$		$Proc(u_R^j)$	Cf. plus haut
$\Rightarrow \Gamma \vdash a_j\langle \cdots \rangle :$		$Proc(u_h^j)$	Par T-Msg
Typage de $\Pi_{j\in J''} a_j\langle R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i\rangle.Q_j$			
$\Gamma \vdash a_j\langle \cdots \rangle :$		$Proc(u_h^j)$	Cf. plus haut
	Si	$u_3 \geq \sum_{j\in J''} u_h^j$	
$\Rightarrow \Gamma \vdash U_3 :$		$Proc(u_3)$	Par T-Par
Typage de $\xi \rhd P$			
	Si	$\Gamma, \gamma \vdash P : Proc(t_P)$	
$\Gamma \vdash \xi :$		$Pat(\gamma)$	Par hypothèse
$\Rightarrow \Gamma \vdash \xi \rhd P :$		$Proc(t_P)$	Par T-Trig
Typage de $(\xi \rhd P) \mid U_1$			
$\Gamma \vdash \xi \rhd P :$		$Proc(t_P)$	Cf. plus haut
$\Gamma \vdash U_1 :$		$Proc(u_1)$	Cf. plus haut
	Si	$u_4 \geq u_1 + t_P$	
$\Rightarrow \Gamma \vdash (\xi \rhd P) \mid U_1 :$		$Proc(u_4)$	Par T-Par
Typage de $(\xi \rhd P) \mid U_1 \mid U_2$			
$\Gamma \vdash (\xi \rhd P) \mid U_1 :$		$Proc(u_4)$	Cf. plus haut
$\Gamma \vdash U_2 :$		$Proc(u_2)$	Cf. plus haut
	Si	$u_5 \geq u_4 + u_2$	
$\Rightarrow \Gamma \vdash (\xi \rhd P) \mid U_1 \mid U_2 :$		$Proc(u_5)$	Par T-Par
Typage de $(\xi \rhd P) \mid U_1 \mid U_2 \mid U_3$			
$\Gamma \vdash (\xi \rhd P) \mid U_1 \mid U_2 :$		$Proc(u_5)$	Cf. plus haut
$\Gamma \vdash U_3 :$		$Proc(u_3)$	Cf. plus haut
	Si	$u_6 \geq u_5 + u_3$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u_6)$	Par T-Par

○

$$\begin{cases}
\forall j \in J & \Gamma \vdash P_j : Proc(s_j) & & \\
\forall j \in J & \Gamma \vdash Q_j : Proc(t^j_Q) & \forall j \in J & u_j \geq e_j \\
\forall j \in J & \Gamma \vdash a_j : Site(s_j, e_j) & \forall j \in J & u_j \geq t^j_Q \\
\forall j \in J' & \Gamma \vdash P_j : Proc(s_j) & \forall j \in J' & u_j \geq e_j \\
\forall j \in J' & \Gamma \vdash Q_j : Proc(t^j_Q) & \forall j \in J' & u_j \geq t^j_Q \\
\forall j \in J' & \Gamma \vdash a_j : Site(s_j, e_j) & \forall j \in J'', \forall i \in I_j & u^d_i \geq e_i \\
\forall j \in J'', \forall i \in I_j & \Gamma \vdash P_i : Proc(s_i) & \forall j \in J'', \forall i \in I_j & u^d_i \geq t^i_S \\
\forall j \in J'', \forall i \in I_j & \Gamma \vdash S_i : Proc(t^i_S) & \forall j \in J'' & u^j_d \geq \sum_{i \in I_j} u^d_i \\
\forall j \in J'', \forall i \in I_j & \Gamma \vdash a_i : Site(s_i, e_i) & \forall j \in J'' & s_j \geq t^j_R + u^j_d \\
\forall j \in J'' & \Gamma \vdash R_j : Proc(t^j_R) & \forall j \in J'' & u^j_h \geq e_j \\
\forall j \in J'' & \Gamma \vdash R_j : Proc(t^j_Q) & \forall j \in J'' & u^j_h \geq t^j_Q \\
\forall j \in J'' & \Gamma \vdash a_j : Site(s_j, e_j) & & u_6 \geq w \\
 & \Gamma, \gamma \vdash P : Proc(t_P) & &
\end{cases}$$

Sous ces hypothèses, nous pouvons typer B dans Γ.

Typage de $P\theta$		
$\Gamma, \gamma \vdash P :$	$Proc(t_P)$	Par hypothèse
$\Rightarrow \Gamma \vdash P\theta :$	$Proc(t_P)$	Par *Substitution*
Typage de V_1		
$\Gamma \vdash Q_j :$	$Proc(t^j_Q)$	Par hypothèse
$\Rightarrow \Gamma \vdash \Pi_{i \in J} Q_j :$	$Proc(\sum_{j \in J} u_j)$	Par T-Par
Typage de V_2		
$\Gamma \vdash Q_j :$	$Proc(t^j_Q)$	Par hypothèse
$\Rightarrow \Gamma \vdash \Pi_{i \in J'} Q_j :$	$Proc(\sum_{j \in J'} u_j)$	Par T-Par
Typage de $\Pi_{i \in I_j} S_i$		
$\Gamma \vdash S_i :$	$Proc(t^i_S)$	Par hypothèse
Comme	$u^j_d \geq \sum_{i \in I_j} u^d_i$	
	$u^d_i \geq t^i_S$	
$\Rightarrow \Gamma \vdash \Pi_{i \in I_j} S_i :$	$Proc(u^j_d)$	Par T-Par
Typage de $R_j \mid \Pi_{i \in I_j} S_i$		
$\Gamma \vdash \Pi_{i \in I_j} S_i :$	$Proc(u^j_d)$	Cf. plus haut
$\Gamma \vdash R_j :$	$Proc(t^j_R)$	Par hypothèse
Comme	$s_j \geq t^j_R + u^j_d$	
$\Rightarrow \Gamma \vdash R_j \mid \Pi_{i \in I_j} S_i :$	$Proc(s_j)$	Par T-Par
Typage de $a_j \, [R_j \mid \Pi_{i \in I_j} S_i] \, .Q_j$		
$\Gamma \vdash R_j \mid \Pi_{i \in I_j} S_i :$	$Proc(s_j)$	Cf. plus haut
$\Gamma \vdash a_j :$	$Site(s_j, e_j)$	Par hypothèse
$\Gamma \vdash Q_j :$	$Proc(t^j_Q)$	Par hypothèse
Comme	$u^j_h \geq e_j$	
	$u^j_h \geq t^j_Q$	
$\Rightarrow \Gamma \vdash a_j \, [\cdots] \, .Q_j :$	$Proc(u^j_h)$	Par T-Site
Typage de V_3		
$\Gamma \vdash a_j \, [\cdots] \, .Q_j :$	$Proc(u^j_h)$	Cf. plus haut
$\Rightarrow \Gamma \vdash V_3 :$	$\sum_{j \in J''} u^j_h$	Par T-Par
Typage de $P\theta \mid V_1 \mid V_2 \mid V_3$		
$\Gamma \vdash P\theta :$	$Proc(t_P)$	Cf. plus haut
$\Gamma \vdash V_1 :$	$Proc(\sum_{j \in J} u_j)$	Cf. plus haut
$\Gamma \vdash V_2 :$	$Proc(\sum_{j \in J'} u_j)$	Cf. plus haut
$\Gamma \vdash V_3 :$	$Proc(\sum_{j \in J''} u^j_h)$	Cf. plus haut

$$\Rightarrow \Gamma \vdash B : \quad \begin{array}{ll} \text{Comme} & u_6 \geq w \\ & Proc(u_6) \end{array} \qquad \text{Par T-Par}$$

○

Ce qui prouve le cas.

Notons que nous n'avons nulle part utilisé la définition de M_1, M_2 ou M_3, ni le fait que $U_2 \not\rightsquigarrow$ ou que $\Gamma \vdash (\xi \rhd P)|U_1|U_2|U_3 : T$.

R-G-Typed – Cas d'initialisation

Notons, dans un environnement Γ,

$$\left\{ \begin{array}{lcl} A & = & b\,[(\xi \rhd P)|U_1|U_2|U_3|R]\,.S|U_4 \\ B & = & b\,[P\theta|V_1|V_2|V_3|R]\,.S|V_4\ . \end{array} \right.$$

où

$$\left\{ \begin{array}{ll} \Gamma \vdash & \xi : Pat(\gamma) \\ \Gamma \vdash & (\xi \rhd P)|U_1|U_2|U_3 : T \\ & \xi \neq \emptyset \\ & \theta = \mathtt{match}_\Gamma(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3) \mid ab(M_4)) \\ & \gamma \text{ et } \theta \text{ ont le même domaine de définition} \\ & \forall x, \theta(x) = P_x \Rightarrow \Gamma \vdash P_x : \gamma(x) \\ & \Delta(U_1, M_1, V_1) \\ & \Upsilon(U_2, M_2, V_2) \\ & \Psi(U_3, M_3, V_3) \\ & \Delta(U_4, M_4, V_4) \\ & U_2 \not\rightsquigarrow \\ \Delta(U,M,V) \triangleq & \left\{ \begin{array}{lcl} U & = & \Pi_{j\in J} a_j\langle P_j\rangle.Q_j \\ M & = & \Pi_{j\in J} a_j\langle P_j\rangle \\ V & = & \Pi_{j\in J} Q_j \end{array} \right. \\ \Upsilon(U,M,V) \triangleq & \left\{ \begin{array}{lcl} U & = & \Pi_{j\in J'} a_j\,[P_j]\,.Q_j \\ M & = & \Pi_{j\in J'} a_j\,[P_j] \\ V & = & \Pi_{j\in J'} Q_j \end{array} \right. \\ \Psi(U,M,V) \triangleq & \left\{ \begin{array}{lcl} U & = & \Pi_{j\in J''} a_j\langle R_j \mid \Pi_{i\in I_j} a\_i\langle P_i\rangle.S_i\rangle.Q_j \\ M & = & \Pi_{j\in J''}\Pi_{i\in I_j} a_i\langle P_i\rangle^{\downarrow a_j} \\ V & = & \Pi_{j\in J''} a_j\left[R_j \mid \Pi_{i\in I_j} S_i\right].Q_j \end{array} \right. \end{array} \right.$$

Pour éviter toute ambiguïté, nous supposerons que l'ensemble J ne contient pas d'entiers. De plus, nous noterons $w \triangleq \sum_{j\in J} u_j + \sum_{j\in J'} u_j + \sum_{j\in J''} u_h^j + t_P + t_R$.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $a_j\langle P_j\rangle.Q_j$			
	Si	$\Gamma \vdash P_j : Proc(s_j)$	
		$Q_j : Proc(t_Q^j)$	
		$a_j : Site(s_j, e_j)$	
		$u_j \geq e_j$	
		$u_j \geq t_Q^j$	
$\Rightarrow \Gamma \vdash a_j\langle P_j\rangle.Q_j :$		$Proc(u_j)$	Par T-Msg

Typage de $\Pi_{j\in J} a_j\langle P_j\rangle.Q_j$			
$\Gamma \vdash a_j\langle P_j\rangle.Q_j :$		$Proc(u_j)$	Cf. plus haut
	Si	$v_4 \geq \sum_{j\in J^{(3)}} u_j$	
$\Rightarrow \Gamma \vdash U_4 :$		$Proc(v_4)$	Par T-Par

Typage de $(\xi \rhd P) \mid U_1 \mid U_2 \mid U_3 \mid R$			
$\Gamma \vdash (\xi \rhd P) \mid U_1 \mid U_2 \mid U_3 :$		$Proc(u_6)$	Cf. plus haut
	Si	$R : Proc(t_R)$	
		$u_7 \geq u_6 + t_R$	
$\Rightarrow \Gamma \vdash (\xi \rhd P) \mid U_1 \mid U_2 \mid U_3 \mid R :$		$Proc(u_7)$	Par T-Par

Typage de $b\,[(\xi \rhd P) \mid U_1 \mid U_2 \mid U_3 \mid R]\,.S$			
$\Gamma \vdash (\xi \rhd P) \mid U_1 \mid U_2 \mid U_3 \mid R :$		$Proc(u_7)$	Cf. plus haut
	Si	$S : Proc(t_S)$	
		$b : Site(s_b, e_b)$	
		$u_8 \geq e_b$	
		$u_8 \geq t_S$	
		$u_7 = s_b$	
$\Rightarrow \Gamma \vdash b\,[\cdots]\,.S :$		$Proc(u_8)$	Par T-Site

Typage de $b\,[(\xi \rhd P)\vert U_1\vert U_2\vert U_3\vert R]\,.S\vert U_4$			
$\Gamma \vdash b\,[\cdots]\,.S :$		$Proc(u_8)$	Cf. plus haut
$\Gamma \vdash U_4 :$		$Proc(v_4)$	Cf. plus haut
	Si	$u_9 \geq v_4 + u_8$	
$\Rightarrow \Gamma \vdash b\,[\cdots] \mid U_4 :$		$Proc(u_9)$	Par T-Par

○

$$
\begin{cases}
\forall j \in J & \Gamma \vdash P_j : Proc(s_j) & \forall j \in J & u_j \geq e_j \\
\forall j \in J & \Gamma \vdash Q_j : Proc(t^j_Q) & \forall j \in J & u_j \geq t^j_Q \\
\forall j \in J & \Gamma \vdash a_j : Site(s_j, e_j) & \forall j \in J' & u_j \geq e_j \\
\forall j \in J' & \Gamma \vdash P_j : Proc(s_j) & \forall j \in J' & u_j \geq t^j_Q \\
\forall j \in J' & \Gamma \vdash Q_j : Proc(t^j_Q) & \forall j \in J'', \forall i \in I_j & u^d_i \geq e_i \\
\forall j \in J' & \Gamma \vdash a_j : Site(s_j, e_j) & \forall j \in J'', \forall i \in I_j & u^d_i \geq t^i_S \\
\forall j \in J'', \forall i \in I_j & \Gamma \vdash P_i : Proc(s_i) & \forall j \in J'' & u^j_d \geq \sum_{i\in I_j} u^d_i \\
\forall j \in J'', \forall i \in I_j & \Gamma \vdash S_i : Proc(t^i_S) & \forall j \in J'' & s_j \geq t^j_R + u^j_d \\
\forall j \in J'', \forall i \in I_j & \Gamma \vdash a_i : Site(s_i, e_i) & \forall j \in J'' & u^j_h \geq e_j \\
\forall j \in J'' & \Gamma \vdash R_j : Proc(t^j_R) & \forall j \in J'' & u^j_h \geq t^j_Q \\
\forall j \in J'' & \Gamma \vdash R_j : Proc(t^j_Q) & \forall j \in J^{(3)} & u_j \geq e_j \\
\forall j \in J'' & \Gamma \vdash a_j : Site(s_j, e_j) & \forall j \in J^{(3)} & u_j \geq t^j_Q \\
\forall j \in J^{(3)} & \Gamma \vdash P_j : Proc(s_j) & & s_b \geq w \\
\forall j \in J^{(3)} & \Gamma \vdash Q_j : Proc(t^j_Q) & & u_8 \geq t_S \\
\forall j \in J^{(3)} & \Gamma \vdash a_j : Site(s_j, e_j) & & u_8 \geq e_b \\
 & \Gamma \vdash b : Site(s_b, e_b) & u_9 \geq u_8 + \sum_{j\in J^{(3)}} u_j & \\
 & \Gamma, \gamma \vdash P : Proc(t_P) & &
\end{cases}
$$

Sous ces hypothèses, nous pouvons typer B dans Γ.

Typage de $P\theta \mid V_1 \mid V_2 \mid V_3 \mid R$			
$\Gamma \vdash P\theta\vert V_1\vert V_2\vert V_3 :$		$Proc(w)$	Cf. plus haut
$\Gamma \vdash R :$		$Proc(t_R)$	Par hypothèse
	Comme	$s_b \geq w + t_R$	
$\Rightarrow \Gamma \vdash P\theta \mid V_1 \mid V_2 \mid V_3 \mid R : Proc(s_b)$			Par T-Par

Typage de $b\,[P\theta \mid V_1 \mid V_2 \mid V_3 \mid R]\,.S$

$\Gamma \vdash P\theta \mid V_1 \mid V_2 \mid V_3 \mid R :$		$Proc(s_b)$	Cf. plus haut
$\Gamma \vdash b :$		$Site(s_b, e_b)$	Par hypothèse
$\Gamma \vdash S :$		$Proc(t_S)$	Par hypothèse
	Comme	$u_8 \geq t_S$	
		$u_8 \geq e_b$	
$\Rightarrow \Gamma \vdash b[\cdots].S :$		$Proc(u_8)$	Par T-Site
Typage de B			
$\Gamma \vdash b[\cdots].S :$		$Proc(u_8)$	Cf. plus haut
$\Gamma \vdash V_4 :$		$Proc(\sum_{j \in J^{(3)}} u_j)$	Cf. plus haut
	Comme	$u_9 \geq u_8 + \sum_{j \in J^{(3)}} u_j$	
$\Rightarrow \Gamma \vdash B :$		$Proc(u_9)$	Par T-Par

○

Ce qui prouve le cas.

R-Struct – Cas d'héritage

Ce cas se prouve exactement comme sa contrepartie des Controlled Ambients.

R-Context – Cas d'héritage

Ce cas se décompose à son tour en quatre sous-cas, conformément à la définition de **E** :

$$\begin{array}{rcll} \mathbf{E} & ::= & \cdot & (1) \\ & \mid & (\nu a : T)\mathbf{E} & (2) \\ & \mid & a[\mathbf{E}].P & (3) \\ & \mid & P|\mathbf{E} & (4) \end{array}$$

Cas (1) Directement par hypothèse d'induction.

Cas (2) Exactement comme le cas R-Res des Controlled Ambients.

Cas (3) Exactement comme le cas R-Amb des Controlled Ambients.

Cas (4) Exactement comme le cas R-Par des Controlled Ambients.
□

C.3 Lemmes – deuxième système de types

C.3.1 Substitutions et renommages (bis)

Sous-lemme 14 (Substitution d'un nom dans un motif)
Si $\Gamma, n : T \vdash_{pat} \xi : Pat(\gamma)$ *et* $\Gamma \vdash m : T$ *avec* $m \notin Dom(\gamma)$, *alors* $\Gamma \vdash \xi\{n \leftarrow m\} : Pat(\gamma)$.

Nous prouverons ce lemme par induction sur la structure d'une preuve de $\Gamma, n : T \vdash_{pat} \xi : Pat(\gamma)$.

Règle T-Pat-Cell – Cas d'initialisation

Si la règle T-Pat-Cell s'applique, nous avons $\xi = a\,[(x : T)]$, $\Gamma, n : T \vdash a : Site(\_)[T_a]$ et $\gamma = \{x : T\}$.

Cas $a = n$ Supposons $a = n$. Comme $\Gamma \vdash m : Site(\_)[T_a]$ et comme $\xi\{n \leftarrow m\}$ s'écrit $m\,[(x : T)]$, nous pouvons appliquer T-Pat-Cell et en déduire $\Gamma \vdash \xi\{n \leftarrow m\} : Pat(\gamma)$.

Cas $a \neq n$ Supposons $a \neq n$. Alors, nous avons aussi $\Gamma \vdash a : Site(\_)[T]$, ce qui nous permet de déduire $\Gamma \vdash_{pat} \xi : Pat(\gamma)$. Comme n n'intervient pas, nous avons aussi $\xi = \xi\{n \leftarrow m\}$, ce qui conclut le cas.

Règle T-Pat-Par – Cas d'héritage

Nous avons $\xi = \xi_1 \mid \xi_2$, $\gamma = \gamma_1 \cup \gamma_2$ où $\Gamma, n : T \vdash_{pat} \xi_1 : Pat(\gamma_1)$ et $\Gamma, n : T \vdash_{pat} \xi_2 : Pat(\gamma_2)$, avec $Dom(\gamma_1) \cap Dom(\gamma_2) = \emptyset$.

Par hypothèse d'induction, comme $m \notin Dom(\gamma_1)$, nous pouvons déduire $\Gamma \vdash_{pat} \xi_1\{n \leftarrow m\} : Pat(\gamma_1)$. De même, nous trouvons $\Gamma \vdash_{pat} \xi_2\{n \leftarrow m\} : Pat(\gamma_2)$.

Comme $Dom(\gamma_1) \cap Dom(\gamma_2) = \emptyset$, $\Gamma \vdash_{pat} \xi_1\{n \leftarrow m\} : Pat(\gamma_1)$ et $\Gamma \vdash_{pat} \xi_2\{n \leftarrow m\} : Pat(\gamma_2)$, par T-Pat-Par, nous en déduisons $\Gamma \vdash_{pat} \xi_1\{n \leftarrow m\} \mid \xi_2\{n \leftarrow m\} = Pat(\gamma_1 \cup \gamma_2)$. Par conséquent, comme $\xi\{n \leftarrow m\} \stackrel{\Delta}{=} \xi_1\{n \leftarrow m\} \mid \xi_2\{n \leftarrow m\}$, nous pouvons conclure sur le cas.

Règle T-Pat-Par2 – Cas d'héritage

Le cas se gère exactement comme T-Pat-Par.

Règle T-Pat-Message-Binder – Cas d'initialisation

Le cas se gère exactement comme T-Pat-Cell.

Règle T-Pat-Message-Constant – Cas d'initialisation

Nous avons $\xi = a\langle b\rangle$- et $\gamma = \emptyset$. Le cas est trivial car Γ n'est pas utilisé par T-Pat-Message-Constant et la valeur de γ reste forcément $\emptyset$.

Règle T-Pat-Message-Empty – Cas d'initialisation

Le cas se gère exactement comme T-Pat-Message-Constant.

Règle T-Pat-Message-Submessage – Cas d'héritage

Le cas se gère exactement comme T-Pat-Cell.

Sous-lemme 15 (Substitution d'un nom)
Si $\Gamma, n : T \vdash P : U$ *et* $\Gamma \vdash m : T$, *alors* $\Gamma \vdash P\{n \leftarrow m\} : U$.

Le seul cas qui nous intéresse est celui d'un processus P de la forme $\xi \triangleright Q$. Nous supposerons, en accord avec des conventions à la Barendregt, que $m \notin Dom(\gamma)$.

La règle T-TRIG s'énonce

$$\text{T-TRIG}\ \frac{\Gamma \vdash_{pat} \xi : Pat(\gamma) \qquad \Gamma, \gamma \vdash P : Proc(t_P)}{\Gamma \vdash \xi \triangleright P : Proc(t_P)}\ .$$

Ainsi, lorsque cette règle s'applique, nous avons $\Gamma, n : T \vdash \xi \triangleright Q : Proc(t_P)$, $\Gamma, n : T \vdash_{pat} \xi : Pat(\gamma)$ et $\Gamma, n : T, \gamma \vdash Q : Proc(t_P)$. D'après le sous-lemme de substitution d'un nom dans un motif, nous avons $\Gamma \vdash_{pat} \xi\{n \leftarrow m\} : Pat(\gamma)$

Alors, par hypothèse d'induction, comme $\Gamma, \gamma, n : T \vdash Q : Proc(t_P)$, nous avons aussi $\Gamma, \gamma \vdash Q\{n \leftarrow m\} : Proc(t_P)$. Par conséquent, en appliquant T-TRIG, nous obtenons $\Gamma \vdash (\xi \triangleright P)\{n \leftarrow m\} : Proc(t_P)$, ce qui suffira à prouver le cas.

C.3.2 Congruence structurelle (bis)

Sous-lemme 16 (Les bons types supportent la congruence)
Soient deux processus A et B tels que $A \equiv B$. Si $\Gamma \vdash A : U$, alors $\Gamma \vdash B : U$ et si $\Gamma \vdash B : U$ alors $\Gamma \vdash A : U$.

Le seul cas qui nous intéresse est celui de la règle STRUCT-PAT, instanciée à son tour par les trois règles suivantes :

$$\xi_1|\xi_2 \equiv \xi_2|\xi_1 \qquad \rho_1|\rho_2 \equiv \rho_2|\rho_1 \qquad \frac{\overline{\rho} \equiv \overline{\rho}'}{a\langle\overline{\rho}\rangle^{-} \equiv a\langle\overline{\rho}'\rangle^{-}}$$

Ces trois sous-cas sont triviaux. Nous n'entrerons donc pas dans le détail de la preuve.

C.3.3 Filtrage par motif compatible avec les types (lemme 19)

Lorsque $Dom(\gamma) = Dom(\theta)$ et, pour tout x, si $\theta(x) = P_x$ alors $\Gamma \vdash P_x : \gamma(x)$, nous noterons $\gamma \sim \theta$.

Ce sous-lemme peut être prouvé par induction sur une preuve de $\Gamma \vdash_{pat} \xi : Pat(\gamma)$.

Nous numéroterons les règles de filtrage comme suit :

$$\begin{array}{lcll}
\texttt{match}(\xi, \alpha(M_1|M_2)) & \triangleq & \texttt{match}(\xi, \alpha(M_2)|\alpha(M_2)) & (1)\\
\texttt{match}(\xi_1|\xi_2, \alpha(M_1)|\beta(M_2)) & \triangleq & \texttt{match}(\xi_1, \alpha(M_1)) \oplus \texttt{match}(\xi_2, \beta(M_2)) & (2)\\
 & & \text{pour toutes annotation } \alpha, \beta & \\
\texttt{match}(a\,[(x : _)]\,, loc(a\,[P])) & \triangleq & \{x \leftarrow P\} & (3)\\
\texttt{match}(a\langle\overline{\rho}\rangle, loc(a\langle P\rangle)) & \triangleq & \texttt{match}(\overline{\rho}, loc(P)) & (4)\\
\texttt{match}(a\langle\overline{\rho}\rangle^{\downarrow}, dn(a\langle P\rangle)) & \triangleq & \texttt{match}(\overline{\rho}, loc(P)) & (5)\\
\texttt{match}(a\langle\overline{\rho}\rangle^{\uparrow}, ab(a\langle P\rangle)) & \triangleq & \texttt{match}(\overline{\rho}, loc(P)) & (6)\\
\texttt{match}(\epsilon, loc(\mathbf{0})) & \triangleq & \{\} & (7)\\
\texttt{match}(n, loc(n)) & \triangleq & \{\} & (8)\\
\texttt{match}((x : _), loc(P)) & \triangleq & \{x \leftarrow P\}(9) & \\
\texttt{match}((a : _)\,[\overline{\rho}]\,, loc(b\,[P])) & \triangleq & \{a \leftarrow b\} \oplus \texttt{match}(\overline{\rho}, P) & (10)
\end{array}$$

Règle T-Pat-Cell – Cas d'initialisation

Nous avons $\xi = a\,[(x : T)]$, $\Gamma \vdash a : \mathit{Site}(\_)[T]$ et $\gamma = \{x : T\}$. Comme $\theta = \mathtt{match}(\xi, K)$ et comme seule la règle (3) permet à ξ de reconnaître K, nous pouvons en déduire que $K = \mathit{loc}(a\,[P])$ et $\theta = \{x \leftarrow P\}$.

Par suite, nous avons $\gamma \sim \theta$.

Règle T-Pat-Par – Cas d'héritage

Nous avons $\xi = \xi_1 \mid \xi_2$, $\Gamma \vdash_{pat} \xi_1 : \mathit{Pat}(\gamma_1)$, $\Gamma \vdash_{pat} \xi_2 : \mathit{Pat}(\gamma_2)$, $\mathit{Dom}(\gamma_1) \cap \mathit{Dom}(\gamma_2) = \emptyset$ et $\gamma = \gamma_1 \cup \gamma_2$.

De plus, comme ξ reconnaît K avec la substitution θ, ce qui nécessite la règle (2), nous pouvons en déduire que $K = \alpha(M_1) \mid \beta(M_2)$, $\theta_1 = \mathtt{match}(\xi_1, \alpha(M_1))$, $\theta_2 = \mathtt{match}(\xi_2, \beta(M_2))$ et $\theta = \theta_1 \oplus \theta_2$.

De plus, comme T-PAR est la seule règle qui permette de typer $\overline{K}$ et comme $\overline{K}$ est typable dans Γ, nous pouvons déduire que M_1 et M_2 sont typables dans Γ.

Appliquons l'hypothèse d'induction à ξ_1, θ_1, Γ et $\alpha(M_1)$. Comme $\Gamma \vdash \xi_1 : \mathit{Pat}(\gamma_1)$, comme $\overline{\alpha(M_1)}$ est typable dans Γ et comme $\theta_1 = \mathtt{match}(\xi_1, \alpha(M_1))$, alors nous pouvons déduire que $\mathit{Dom}(\gamma_1) = \mathit{Dom}(\theta_1)$ et, pour tout x, si $\theta_1(x) = P_x$ alors $\Gamma \vdash P_x : \gamma_1(x)$.

De même, nous pouvons déduire que $\mathit{Dom}(\gamma_2) = \mathit{Dom}(\theta_2)$ et, pour tout x, si $\theta_2(x) = P_x$ alors $\Gamma \vdash P_x : \gamma_2(x)$.

Par suite, $\mathit{Dom}(\gamma_1 \cup \gamma_2) = \mathit{Dom}(\gamma_1) \cup \mathit{Dom}(\gamma_2) = \mathit{Dom}(\theta_1) \cup \mathit{Dom}(\theta_2) = \mathit{Dom}(\theta_1 \oplus \theta_2)$. Si nous avons, de plus, $\theta_1 \oplus \theta_2(x) = P_x$, alors soit $\theta_1(x) = P_x$, soit $\theta_2(x) = P_x$. Supposons $\theta_1(x) = P_x$. Comme nous l'avons vu plus tôt, $\Gamma \vdash P_x : \gamma_1(x)$. Or, $\mathit{Dom}(\gamma_1) \cap \mathit{Dom}(\gamma_2) = \emptyset$. Par suite comme $\gamma_1(x)$ est défini, nous pouvons en déduire que $\gamma_1 \cup \gamma_2(x) = \gamma_1(x)$, soit $\gamma(x) = \gamma_1(x)$. Donc $\Gamma \vdash P_x : \gamma(x)$. Le cas $\theta_2(x) = P_x$ est identique.

Nous avons donc prouvé le cas.

Règle T-Pat-Par2 – Cas d'héritage

Le cas se règle comme T-PAT-PAR.

Règle T-Pat-Message-Binder – Cas d'initialisation

Le cas se gère comme T-PAT-CELL.

Règle T-Pat-Message-Constant – Cas d'initialisation

Nous avons $\xi = a\langle\epsilon\rangle$ et $\gamma = \emptyset$.

De plus, comme ξ reconnaît K avec la substitution θ, ce qui nécessite la règle (7), nous pouvons déduire que $K = \mathit{loc}(\mathbf{0})$ et $\theta = \emptyset$.

Ce qui règle le cas.

Règle T-Pat-Message-Empty – Cas d'initialisation Le cas se règle comme T-PAT-MESSAGE-CONSTANT.

Règle T-Pat-Message-Submessage – Cas d'héritage Nous avons $\xi = a\langle\overline{\rho}\rangle^-$ où $\Gamma \vdash_{pat} \overline{\rho} : Pat(\gamma)$.

De plus, comme ξ reconnaît K avec la substitution θ, ce qui nécessite l'une des règles (4), (5) ou (6), nous pouvons déduire que $K = \alpha(a\,[P]\,.Q)$ pour une certaine annotation α et que $\theta = match(\overline{\rho}, loc(P))$.

Comme $a\,[P]\,.Q$ est typable et comme la seule règle qui permette de typer ce terme est T-SITE, nous pouvons déduire que P est typable. Comme, de plus, nous avons $\Gamma \vdash_{pat} \overline{\rho} : Pat(\gamma)$ et $\theta = match(\overline{\rho}, loc(P))$, par hypothèse d'induction, nous pouvons déduire que $\gamma \sim \theta$. Ce qui prouve le cas. □.

C.4 Théorème – deuxième système de types

C.4.1 Réduction du sujet

Prouvons par induction structurelle sur une preuve de $A \longrightarrow B$ que "Pour tout processus A et tout environnement Γ, si $\Gamma \vdash A : Proc(t_A)$ et si $A \longrightarrow B$, alors $\Gamma \vdash B : Proc(t_A)$." Dans tout ce qui suit, nous omettrons généralement les applications de la règle T-NAME et nous factoriserons les applications de T-RES.

Les seuls cas modifiés sont R-L-TYPED et R-G-TYPED.

R-L – Cas d'initialisation

Notons, dans un environnement Γ,

$$\begin{cases} A &= (\xi \rhd P)|U_1|U_2|U_3 \\ B &= P\theta|V_1|V_2|V_3\ . \end{cases}$$

où

$$\begin{cases} & \xi \neq \emptyset \\ & \theta = \mathtt{match}(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3)) \\ & \Delta(U_1, M_1, V_1) \\ & \Upsilon(U_2, M_2, V_2) \\ & \Psi(U_3, M_3, V_3) \\ & U_2 \not\rightarrow \\ \Delta(U,M,V) \triangleq & \begin{cases} U &= \Pi_{j\in J} a_j\langle P_j\rangle.Q_j \\ M &= \Pi_{j\in J} a_j\langle P_j\rangle \\ V &= \Pi_{j\in J} Q_j \end{cases} \\ \Upsilon(U,M,V) \triangleq & \begin{cases} U &= \Pi_{j\in J'} a_j\,[P_j]\,.Q_j \\ M &= \Pi_{j\in J'} a_j\,[P_j] \\ V &= \Pi_{j\in J'} Q_j \end{cases} \\ \Psi(U,M,V) \triangleq & \begin{cases} U &= \Pi_{j\in J''} a_j\langle R_j \mid \Pi_{i\in I_j} a_i\langle P_i\rangle.S_i\rangle.Q_j \\ M &= \Pi_{j\in J''}\Pi_{i\in I_j} a_i\langle P_i\rangle^{\downarrow a_j} \\ V &= \Pi_{j\in J''} a_j\left[R_j \mid \Pi_{i\in I_j} S_i\right].Q_j \end{cases} \end{cases}$$

Si $\Gamma \vdash A : T$, nous pouvons déduire aisément qu'il existe γ tel que $\Gamma \vdash_{pat} \xi : \gamma$ et que $U_1 \mid U_2 \mid U_3$ est typable dans Γ. Par suite, d'après le lemme 19 sur les motifs typés, comme $\theta = \mathtt{match}(\xi, loc(M_1) \mid dn(M_2) \mid loc(M_3))$, nous pouvons en déduire que γ et θ ont le même domaine de définition et, pour tout x, si

$\theta(x) = P_x$ alors $\Gamma \vdash P_x : \gamma(x)$. Par suite, toutes les hypothèses vérifiées par A dans R-L-TYPED sont aussi vérifiées par A dans R-L.

Nous pouvons donc réutiliser la preuve du cas R-L-TYPED. Ce qui prouve le cas.

R-G – Cas d'initialisation

De la même manière qu'une preuve du cas R-L-TYPED combinée avec le lemme 19 permet de gérer le cas R-L, une preuve du cas R-G-TYPED combiné avec ce même lemme permet de gérer le cas R-G.

Annexe D

Cπ

D.1 Lemmes

D.1.1 Substitutions et renommages

Sous-lemme 17 (Substitution)
Si $\Gamma, n : C \vdash P : Proc(t)[\lambda]$, *si* $n \notin \lambda$ *et si* $\Gamma \vdash m : C$, *alors* $\Gamma \vdash P\{n \leftarrow m\} : Proc(t)[\lambda]$.

Notons en particulier que $m \neq \circledcirc$.

Ce sous-lemme se prouve sans difficultés, exactement de la même manière que le sous-lemme 1 – la règle T-Name se contente de remplacer son homologue dans les ambients, T-AmbName.

Sous-lemme 18 (Renommage d'une variable liée)
Si $\Gamma \vdash (\nu n : C)P : U$ *et* $m \notin fv(P)$, *alors* $\Gamma \vdash (\nu m : C)P\{n \leftarrow m\} : U$.

La preuve est presque identique.

D.1.2 Congruence Structurelle

Sous-lemme 19 (Les bons types supportent la congruence)
Soient deux processus A *et* B *tels que* $A \equiv B$. *Si* $\Gamma \vdash A : U$, *alors* $\Gamma \vdash B : U$ *et si* $\Gamma \vdash B : U$ *alors* $\Gamma \vdash A : U$.

Comme la conception des ressources interagit avec la congruence structurelle, contrairement aux calculs précédents, nous allons devoir prouver ce lemme dans le cadre de Cπ.

Le lemme se prouve par induction structurelle sur une preuve de $A \equiv B$. Chaque cas correspond, comme précédemment,

- soit à une règle de congruence structurelle présentée sur la figure 6.10
- soit à une propriété qui d'équivalence de la relation $\equiv$ (transitivité, symétrie, réflexivité)
- soit à une des propriétés qui font de $(P/\equiv, |, \mathbf{0})$ un monoïde commutatif et associatif (commutativité, associativité, neutralité de $\mathbf{0}$ vis-à-vis de l'opération $|$)
- soit à l'α-équivalence des variables liées

$$\textsc{Struct-Par}\ \frac{P \equiv Q}{P|R \equiv Q|R} \qquad \textsc{Struct-Repl}\ \frac{P \equiv Q}{!P \equiv !Q}$$

$$\textsc{Struct-Sum-Dist}\ \frac{Q \equiv R}{P + Q \equiv P + R}$$

$$\textsc{Struct-Pre}\ \frac{P \equiv Q}{\alpha.P \equiv \alpha.Q} \textsc{Struct-Res}\ \frac{P \equiv Q}{(\nu c : C)P \equiv (\nu c : C)Q}$$

$$\textsc{Struct-Fin}\ \frac{P \equiv Q}{(\neg c)P \equiv (\neg c)Q}$$

FIG. D.1 – Propriétés de congruence de la congruence structurelle – Cπ.

– soit à une propriété de congruence de la relation ≡, illustrées sur la figure D.1.

Struct-Repl-Par – Cas d'initialisation

Si la règle STRUCT-REPL-PAR s'applique, nous avons $A = !P$ et $B = P \mid !P$. De plus, si $\Gamma \vdash A : U$, alors $U = Proc(t_A)[\lambda_A]$ et $\Gamma \vdash P : Proc(0)[\emptyset]$. Par conséquent, de $\Gamma \vdash !P : U$ et $\Gamma \vdash P : Proc(0)[\emptyset]$, comme $\emptyset \cap \lambda_A = \emptyset$ et $t_A \geq t_A + 0$, par T-PAR, nous pouvons déduire $\Gamma \vdash P \mid !P : Proc(t_A)[\lambda_A \cap \emptyset]$. C'est-à-dire $\Gamma \vdash B : U$. Ce qui prouve un sens.

Si, maintenant, nous avons $\Gamma \vdash B : U$, alors, nous pouvons déduire que $\Gamma \vdash P : Proc(0)[\emptyset]$. Par suite, en appliquant T-REPL, nous obtenons $\Gamma \vdash A : U$.

Ce qui achève le cas.

Struct-Res-Res – Cas d'initialisation

Nous avons $A = (\nu n : Name(C_n, e_n))(\nu m : Name(C_m, e_m))P$ et $A = (\nu m : Name(C_m, e_m))(\nu n : Name(C_n, e_n))P$ avec $n \neq m$.

Comme T-RES appliquée deux fois est la seule règle qui permette de typer A ou B, nous pouvons déduire que $\Gamma \vdash A : Proc(t_P + e_m + e_n)[\lambda_A]$ et $\Gamma, n : T_n, m : T_m \vdash P : Proc(t_P)[\lambda_A, n, m]$ pour t_P quelconque et λ_A ne contenant ni n ni m. Or, comme $n \neq m$, les fonctions $\Gamma, n : T_n, m : T_m$ et $\Gamma, m : T_m, n : T_n$ sont identiques. De plus, λ_A, n, m et λ_A, m, n sont les mêmes fonctions et, par commutativité de l'addition, $t_P + e_m + e_n = t_P + e_n + e_m$.

Par suite, nous avons aussi $\Gamma, m : T_m, n : T_n \vdash P : Proc(t_P)[\lambda, m, n]$. Par suite, en appliquant deux fois T-RES, $\Gamma \vdash B : Proc(t_P + e_m + e_n)[\lambda]$. Ce qui prouve un sens.

Le cas symétrique est, justement, symétrique.

Struct-Res-Par – Cas d'initialisation

Nous avons $A = (\nu n : T)(P|Q)$ et $B = P|(\nu n : T)Q$ avec $n \notin fv(P)$ et $T = Name(\_, e)$.

De A vers B Supposons $\Gamma \vdash A : U$. La réduction la plus générale qui permette de typer A est alors

Typage de $P\|Q$			
	Si	$\Gamma, n : T \vdash P : Proc(t_P)[\lambda_P]$	
		$Q : Proc(t_Q)[\lambda_Q]$	
		$\lambda_P \cap \lambda_Q = \emptyset$	
		$u_1 \geq t_P + t_Q$	
$\Rightarrow \Gamma, n : T \vdash P\|Q :$		$Proc(u_1)[\lambda_P \cup \lambda_Q]$	Par T-Par
Typage de $(\nu n : T)(P\|Q)$			
$\Gamma, n : T \vdash P\|Q :$		$Proc(u_1)[\lambda_P \cup \lambda_Q]$	Cf. plus haut
	Si	$u_A \geq u_1 + e$	
$\Gamma \vdash (\nu n : T)(P\|Q) :$		$Proc(u_A)[\lambda_P \cup \lambda_Q]$	Par T-Res

○

De cette réduction, nous déduisons $U = Proc(u_A)[\lambda_P \cup \lambda_Q]$, $\Gamma, n : T \vdash P : Proc(t_P)[\lambda_P]$, $\Gamma, n : T \vdash Q : Proc(t_Q)[\lambda_Q]$ $\lambda_P \cap \lambda_Q = \emptyset$ et $u_A \geq t_P + t_Q + e$.

Du lemme de renforcement, comme $n \notin fv(P)$, nous pouvons déduire $\Gamma \vdash P : Proc(t_P)[\lambda_P]$.

Nous pouvons alors procéder à la dérivation suivante

Typage de $(\nu n : T)Q$			
$\Gamma, n : T \vdash$		$Q : Proc(t_Q)[\lambda_Q]$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu n : T)Q :$		$Proc(t_Q + e)[\lambda_Q]$	Par T-Res
Typage de $P\|(\nu n : T)Q$			
$\Gamma \vdash P :$		$Proc(t_P)[\lambda_P]$	Cf. plus haut
$\Gamma \vdash (\nu n : T)Q :$		$Proc(t_Q + e)[\lambda_Q]$	Par T-Res
	Comme	$\lambda_P \cap \lambda_Q = \emptyset$	
		$u_A \geq t_P + t_Q + e$	
$\Rightarrow \Gamma \vdash P\|(\nu n : T)Q : Proc(u_A)[T_P]$		Par T-Par	

○

Nous pouvons donc déduire $\Gamma \vdash B : U$.

De B vers A Le cas est essentiellement symétrique. Il suffit de remplacer le renforcement par l'affaiblissement.

Struct-Zero-Repl – Cas d'initialisation

Le cas est trivial.

Struct-Sum-Com – Cas d'initialisation

Le cas est trivial. On peut le prouver en se servant du fait que la règle T-Sum est symétrique.

Struct-Sum-Nil – Cas d'initialisation

Le cas est trivial. On peut le prouver en se servant du fait que **0** peut avoir n'importe quel type, notamment le type de P.

Struct-Res – Cas d'héritage

Si la règle STRUCT-RES s'applique, nous avons $A = (\nu n : Name(T, e))A'$ et $B = (\nu n : Name(T, e))B'$ avec $A' \equiv B'$. Comme T-RES est la seule règle qui permette de typer A ou B, nous pouvons déduire que $\Gamma \vdash A : Proc(u_A)[\lambda_A]$ et $\Gamma, n : Name(T, e) \vdash A' : Proc(t_A)[\lambda_A, x]$ où $u_A \geq t_A + e$ et $x \notin \lambda_A$

Or, par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma, n : Name(T, e) \vdash A' : Proc(t_A)[\lambda_A, x]$, alors $\Gamma, n : Name(T, e) \vdash B' : Proc(t_A)[\lambda_A, x]$. Par suite, en appliquant T-RES, comme $u_A \geq t_A + e$ et $x \notin \lambda_A$, nous pouvons en déduire $\Gamma \vdash B : Proc(u_A)[\lambda_A]$. Ce qui prouve un sens.

L'autre sens est symétrique.

Struct-Repl – Cas d'héritage

Si la règle STRUCT-REPL s'applique, nous avons $A =!A'$ et $B =!B'$ où $A' \equiv B'$.

De plus, si $\Gamma \vdash A : U$, alors $U = Proc(t_A)[\lambda_A]$ et $\Gamma \vdash A' : Proc(0)[\emptyset]$. Or, par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : Proc(0)[\emptyset]$, nous avons $\Gamma \vdash B' : Proc(0)[\emptyset]$. Par suite, en appliquant T-RES, nous avons $\Gamma \vdash B : U$.

L'autre sens est symétrique.

Struct-Sum-Dist – Cas d'héritage

Nous avons $A = A' + Q$ et $B = B' + Q$ où $A' \equiv B'$.

Comme A est typable dans Γ avec $\Gamma \vdash A : U$ et comme la seule règle qui permette de typer A est T-SUM, nous déduisons que $\Gamma \vdash A' : U$ et $\Gamma \vdash Q : U$. Par hypothèse d'induction, comme $A' \equiv B'$, nous déduisons que $\Gamma \vdash B' : U$. En appliquant de nouveau T-SUM, nous concluons que $\Gamma \vdash B : U$.

Le cas symétrique est identique.

Struct-Pre – Cas d'héritage

Émission Nous avons $A = \overline{c}\langle x\rangle A'$ et $B = \overline{c}\langle x\rangle B'$ où $A' \equiv B'$.

Comme A est typable dans Γ et comme la seule règle qui permette de typer A est T-WRITE, nous pouvons écrire $\Gamma \vdash c : Name(Chan(C, z), \_)$, $\Gamma \vdash A' : Proc(t_{A'})[\lambda_{A'}]$ et $\Gamma \vdash \overline{c}\langle x\rangle.A' : Proc(t_{A'} - z)[\lambda_{A'}]$ avec $t_{A'} - z \geq 0$. Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : Proc(t_{A'})[\lambda_{A'}]$, nous déduisons $\Gamma \vdash B' : Proc(t_{A'})[\lambda_{A'}]$. En appliquant de nouveau T-WRITE avec ces mêmes hypothèses, nous concluons que $\Gamma \vdash B : U$.

Le cas symétrique est identique.

Réception Nous avons $A = c(x)A'$ et $B = c(x)B'$ où $A' \equiv B'$.

Comme A est typable dans Γ et comme la seule règle qui permette de typer A est T-READ, nous pouvons écrire $\Gamma \vdash c : Name(Chan(C, z), \_)$, $\Gamma, x : C \vdash A' : Proc(t_{A'})[\lambda_{A'}]$ et $\Gamma \vdash c(x).A' : Proc(t_{A'} + z)[\lambda_{A'}]$ avec $t_{A'} + z \geq 0$. Par hypothèse d'induction, comme $A' \equiv B'$ et $\Gamma \vdash A' : Proc(t_{A'})[\lambda_{A'}]$, nous déduisons $\Gamma \vdash B' : Proc(t_{A'})[\lambda_{A'}]$. En appliquant de nouveau T-READ avec ces mêmes hypothèses, nous concluons que $\Gamma \vdash B : U$.

Le cas symétrique est identique.

Struct-Fin – cas d'héritage

Ce cas se traite de la même manière que les autres cas d'héritage. La seule différence est qu'il existe deux manières de typer $(\neg c)P$ ou $(\neg c)Q$: T-FINALIZE1 et T-FINALIZE2.

□

D.1.3 Type minimal

Ce lemme se prouve comme dans les Controlled Ambients.

D.1.4 Utilisation des ressources

Le lemme 29 se prouve par induction sur une preuve de $\Gamma \vdash P : Proc(t)[\lambda]$.

Les cas T-NIL, T-PAR, T-READ, T-WRITE, T-FINALIZE1 et T-FINALIZE2 sont triviaux car, dès que l'une de ces règles peut être appliquée, $res_\Gamma(P)$ vaut 0.

Règle T-Res – Cas d'héritage

Par définition de T-RES, nous pouvons noter $P = (\nu c : Name(T,e))A$, $\Gamma, c : Name(T,e) \vdash A : Proc(t_A)[\_]$ et $\Gamma \vdash P : Proc(u)$ avec $u \geq t_A + e$.

Comme A est typable dans $\Gamma, c : Name(T,e)$, on peut appliquer à A l'hypothèse d'induction et en déduire que $res_{\Gamma,c:Name(T,e)}(A) \leq t_A$. Or, par définition, $res_\Gamma(P) = es_{\Gamma,c:Name(T,e)}(A)+e$. Comme, de plus, $u \geq t_A+e$, nous en déduisons $res_\Gamma(P) \leq u$. Ce qui prouve le cas.

Règle T-Par – Cas d'héritage

Le cas se règle comme dans les Controlled Ambients.

Règle T-Repl – Cas d'héritage

Par définition de T-REPL, nous avons $P = !A$, $\Gamma \vdash A : Proc(0)[\_]$.

Comme A est typable dans Γ, par hypothèse d'induction, $res_\Gamma(A) \leq 0$, c'est-à-dire $res_\Gamma(A) = 0$. Par conséquent, $res_\Gamma(P)$ est défini et vaut 0. Ce qui prouve le cas. □

D.2 Théorèmes

Prouvons par induction structurelle sur une preuve de $A \longrightarrow B$ que "Pour tout processus A et tout environnement Γ, si $\Gamma \vdash A : Proc(t_A)[\lambda_A]$ et si $A \longrightarrow B$, alors $\Gamma \vdash B : Proc(t_A)[\lambda_A]$." Dans tout ce qui suit, nous omettrons généralement les applications de la règle T-NAME.

R-Comm – Cas d'initialisation

Notons

$$\begin{cases} A &= (\overline{a}\langle b\rangle.P + Q) \mid (a(x).R + S) \\ B &= P \mid R\{x \leftarrow b\} \end{cases}$$

avec $a \neq \circledcirc$.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\overline{a}\langle b\rangle.P$		
Si	$\Gamma \vdash P : Proc(t_P)[\lambda_P]$	
	$a : Name(Chan(C, z), \_)$	
	$b : C$	
	$t_P - z \geq 0$	
$\Rightarrow \Gamma \vdash \overline{a}\langle b\rangle.P :$	$Proc(t_P - z)[\lambda_P]$	Par T-WRITE
Typage de $\overline{a}\langle b\rangle.P + Q$		
Si	$\Gamma \vdash Q : Proc(t_Q)[\lambda_Q]$	
	$t_P - z = t_Q$	
	$\lambda_P = \lambda_Q$	
$\Rightarrow \Gamma \vdash \overline{a}\langle b\rangle.P + Q :$	$Proc(t_Q)[\lambda_Q]$	Par T-SUM
Typage de $a(x).R$		
Si	$\Gamma, x : C \vdash R : Proc(t_R)[\lambda_R]$	
	$x \notin \lambda_R$	
	$t_R + z \geq 0$	
$\Gamma \vdash a :$	$Name(Chan(C, z), \_)$	Cf. plus haut
$\Rightarrow \Gamma \vdash a(x).R :$	$Proc(t_R + z)[\lambda_R]$	Par T-READ
Typage de $a(x).R + S$		
Si	$\Gamma \vdash S : Proc(t_S)[\lambda_S]$	
	$t_R + z = t_S$	
	$\lambda_S = \lambda_R$	
$\Rightarrow \Gamma \vdash a(x).R + S :$	$Proc(t_S)[\lambda_S]$	Par T-SUM
Typage de $(\overline{a}\langle b\rangle.P + Q) \mid (a(x).R + S)$		
$\Gamma \vdash \overline{a}\langle b\rangle.P + Q :$	$Proc(t_Q)[\lambda_Q]$	Cf. plus haut
$\Gamma \vdash a(x).R + S :$	$Proc(t_S)[\lambda_S]$	Cf. plus haut
Si	$\lambda_P \cap \lambda_Q = \emptyset$	
	$u \geq t_Q + t_S$	
$\Rightarrow \Gamma \vdash A :$	$Proc(u)[\lambda_P \cup \lambda_Q]$	Par T-PAR

○

Le type le plus général de A est donc $Proc(u)[\lambda_P \cup \lambda_Q]$, avec les conditions suivantes :

$$\begin{cases} \Gamma & \vdash P : Proc(t_P)[\lambda_P] \\ \Gamma & \vdash a : Name(Chan(C, z), \_) \\ \Gamma & \vdash b : C \\ \Gamma & \vdash Q : Proc(t_P - z)[\lambda_P] \\ \Gamma, x : C & \vdash R : Proc(t_R)[\lambda_R] \\ \Gamma & \vdash S : Proc(t_R + z)[\lambda_R] \\ & x \notin \lambda_R \\ & t_P - z \geq 0 \\ & t_R + z \geq 0 \\ & \lambda_P \cap \lambda_R = \emptyset \\ & u \geq t_P + t_R \end{cases}$$

De ces hypothèses, nous pouvons tirer la dérivation suivante :

Typage de $R\{x \leftarrow b\}$

$\Gamma, x : C \vdash R :$	$Proc(t_R)[\lambda_R]$	Par hypothèse
$\Gamma \vdash b :$	C	Par hypothèse
Comme	$x \notin \lambda_R$	
$\Rightarrow \Gamma \vdash R\{x \leftarrow b\} :$	$Proc(t_R)[\lambda_R]$	Par *Substitution*
Typage de $P \mid R\{x \leftarrow b\}$		
$\Gamma \vdash P :$	$Proc(t_P)[\lambda_P]$	Par hypothèse
$\Gamma \vdash R\{x \leftarrow b\} :$	$Proc(t_R)[\lambda_R]$	Cf. plus haut
Comme	$u \geq t_P + t_R$	
	$\lambda_P \cap \lambda_R = \emptyset$	
$\Rightarrow \Gamma \vdash P \mid R\{x \leftarrow b\} :$	$Proc(u)[\lambda_P \cup \lambda_R]$	Par T-Par

○

Ce qui prouve le cas.

R-Par – Cas d'héritage

La preuve est triviale, directement par héritage et application de T-Par.

R-Struct – Cas d'héritage

La preuve est triviale, par héritage et deux applications du sous-lemme 19 (congruence structurelle).

R-Res – Cas d'héritage

Notons

$$\begin{cases} A & = & (\nu n : T)P \\ B & = & (\nu n : T)Q \end{cases}$$

avec $P \longrightarrow Q$ et $T = Name(T, e)$.

Par T-Res, comme A est typable, nous avons $\Gamma, n : T \vdash P : Proc(t_P)[\lambda_A, n]$ et $\Gamma \vdash A : Proc(u_A)[\lambda_A]$ avec $u_A \geq t_P + e$ et $n \notin \lambda_A$. Par hypothèse d'induction, comme $P \longrightarrow Q$, nous avons aussi $\Gamma, n : T \vdash Q : Proc(t_Q)[\lambda_A, n]$. Par T-Res, comme $u_A \geq t_P + e$ et $n \notin \lambda_A$, nous pouvons conclure $\Gamma \vdash B : Proc(u_A)[\lambda_A]$. Ce qui prouve le cas.

R-Fin – Cas d'initialisation

Notons

$$\begin{cases} A & = & (\nu c : C)(\neg c)P \\ B & = & P\{c \leftarrow \circledcirc\} \ . \end{cases}$$

Nous noterons de plus $\Delta = \Gamma, c : C$ et $C = Name(T, e)$.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $(\neg c)P$		
Si	$\Delta \vdash P : Proc(t_P)[\lambda_P]$	
	$c \notin \lambda_P$	
	$u_1 + e \geq t_P$	
$\Delta \vdash c :$	$Name(T, e)$	Par T-Name
$\Rightarrow \Delta \vdash (\neg c)P :$	$Proc(u_1)[\lambda_P, c]$	Par T-Finalize1

Typage de $(\nu c : C)(\neg c)P$		
$\Delta\vdash (\neg c)P :$	$Proc(u_1)[\lambda_P, c]$	Cf. plus haut
Si	$u_2 \geq u_1 + e$	
Comme	$c \notin \lambda_P$	
	$c \in \lambda_P, c$	
$\Rightarrow \Gamma \vdash (\nu c : C)(\neg c)P :$	$Proc(u_2)[\lambda_P]$	Par T-Res

○

Notons qu'il n'aurait pas été possible d'employer T-Finalize2 car T-Res impose $c \in \lambda_P, c$.

Le type le plus général de A dans ce chemin est donc $Proc(u_2)[\lambda_P]$ avec les conditions

$$\begin{cases} \Delta & \vdash P : Proc(t_P)[\lambda_P] \\ & c \notin \lambda_P \\ & u_2 \geq t_P \end{cases}$$

Sous ces hypothèses, nous pouvons typer B dans Γ. En effet, comme $\Delta \vdash P : Proc(t_P)[\lambda_P]$ et $c \notin \lambda_P$, comme $\Gamma \vdash \circledcirc : C$, grâce au sous-lemme de substitution, nous pouvons déduire $\Gamma \vdash P\{c \leftarrow \circledcirc\} : Proc(t_P)[\lambda_P]$. Par suite, le lemme de sous-typage nous permet de conclure $\Gamma \vdash B : Proc(u_2)[\lambda_P]$.

Ce qui prouve le cas.

R-Del – Cas d'initialisation

Notons

$$\begin{cases} A & = & (\neg c)P \mid (\neg c)Q \\ B & = & (\neg c)(P \mid Q) \end{cases}$$

Il existe trois réductions de type "les plus générales" pour A dans un environnement Γ.

Typage de $(\neg c)P$		
Si	$P : Proc(t_P)[\lambda_P]$	
	$c \notin \lambda_P$	
	$c : Name(T, e)$	
$\Rightarrow \Gamma \vdash (\neg c)P :$	$Proc(t_P)[\lambda_P]$	Par T-Finalize2

Typage de $(\neg c)Q$		
Si	$Q : Proc(t_Q)[\lambda_Q]$	
	$c \notin \lambda_Q$	
$\Gamma \vdash c :$	$Name(T, e)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\neg c)Q :$	$Proc(t_Q)[\lambda_Q]$	Par T-Finalize2

Typage de $(\neg c)P \mid (\neg c)Q$		
$\Gamma \vdash (\neg c)P :$	$Proc(t_P)[\lambda_P]$	Cf. plus haut
$\Gamma \vdash (\neg c)Q :$	$Proc(t_Q)[\lambda_Q]$	Cf. plus haut
Si	$\lambda_P \cap \lambda_Q = \emptyset$	
	$u_1 \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash A :$	$Proc(u_1)[\lambda_P \cup \lambda_Q]$	Par T-Par

○

Première possibilité soit $\Gamma \vdash A : Proc(u_1)[\lambda_P \cup \lambda_Q]$ avec

$$\begin{cases} \Gamma & \vdash P : Proc(t_P)[\lambda_P] \\ \Gamma & \vdash Q : Proc(t_Q)[\lambda_Q] \\ \Gamma & \vdash c : Name(T, e) \\ & c \notin \lambda_P \\ & c \notin \lambda_Q \\ & \lambda_P \cap \lambda_Q = \emptyset \\ & u_1 \geq t_P + t_Q \ . \end{cases}$$

On peut alors typer B dans Γ sous ces hypothèses, par

Typage de $P \mid Q$			
$\Gamma \vdash P :$		$Proc(t_P)[\lambda_P]$	Par hypothèse
$\Gamma \vdash Q :$		$Proc(t_Q)[\lambda_Q]$	Par hypothèse
	Comme	$\lambda_P \cap \lambda_Q = \emptyset$	
		$u_1 \geq t_P + t_Q$	
$\Rightarrow \Gamma \vdash P \mid Q :$		$Proc(u_1)[\lambda_P \cup \lambda_Q]$	Par T-Par
Typage de $(\neg c)(P \mid Q)$			
$\Gamma \vdash P \mid Q :$		$Proc(u_1)[\lambda_P \cup \lambda_Q]$	Cf. plus haut
$\Gamma \vdash c :$		$Name(T, e)$	Par hypothèse
	Comme	$c \notin \lambda_P$	
		$c \notin \lambda_Q$	
$\Rightarrow \Gamma \vdash (\neg c)(P \mid Q) :$		$Proc(u_1)[\lambda_P \cup \lambda_Q]$	Par T-Finalize2

○

Donc $\Gamma \vdash B : Proc(u_1)[\lambda_P \cup \lambda_Q]$.

Typage de $(\neg c)P$			
	Si	$P : Proc(t_P)[\lambda_P]$	
		$c \notin \lambda_P$	
		$c : Name(T, e)$	
$\Rightarrow \Gamma \vdash (\neg c)P :$		$Proc(t_P)[\lambda_P]$	Par T-Finalize2
Typage de $(\neg c)Q$			
	Si	$Q : Proc(t_Q)[\lambda_Q]$	
		$c \notin \lambda_Q$	
		$u_2 + e \geq t_Q$	
$\Gamma \vdash c :$		$Name(T, e)$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\neg c)Q :$		$Proc(u_2)[\lambda_Q, c]$	Par T-Finalize1
Typage de $(\neg c)P \mid (\neg c)Q$			
$\Gamma \vdash (\neg c)P :$		$Proc(t_P)[\lambda_P]$	Cf. plus haut
$\Gamma \vdash (\neg c)Q :$		$Proc(u_2)[\lambda_Q, c]$	Cf. plus haut
	Si	$\lambda_P \cap (\lambda_Q, c) = \emptyset$	
		$u_3 \geq t_P + u_2$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u_3)[\lambda_P \cup \lambda_Q, c]$	Par T-Par

○

Deuxième possibilité soit $\Gamma \vdash A : Proc(u_3)[\lambda_P \cup \lambda_Q, c]$ avec

$$\begin{cases} \Gamma & \vdash P : Proc(t_P)[\lambda_P] \\ \Gamma & \vdash Q : Proc(t_Q)[\lambda_Q] \\ \Gamma & \vdash c : Name(T, e) \\ & c \notin \lambda_P \\ & c \notin \lambda_Q \\ & \lambda_P \cap \lambda_Q = \emptyset \\ & u_2 + e \geq t_Q \\ & u_3 \geq t_P + u_2 \end{cases}$$

Sous ces hypothèses, nous pouvons typer B dans Γ.

	Typage de $P \mid Q$		
$\Gamma \vdash P :$		$Proc(t_P)[\lambda_P]$	Par hypothèse
$\Gamma \vdash Q :$		$Proc(t_Q)[\lambda_Q]$	Par hypothèse
	Comme	$\lambda_P \cap \lambda_Q = \emptyset$	
$\Rightarrow \Gamma \vdash P \mid Q :$		$Proc(t_P + t_Q)[\lambda_P \cup \lambda_Q]$	Par T-Par
	Typage de $(\neg c)(P \mid Q)$		
$\Gamma \vdash P \mid Q :$		$Proc(t_P + t_Q)[\lambda_P \cup \lambda_Q]$	Cf. plus haut
$\Gamma \vdash c :$		$Name(T, e)$	Par hypothèse
	Comme	$u_3 + e \geq t_P + t_Q$	
		$c \notin \lambda_P$	
		$c \notin \lambda_Q$	
$\Rightarrow \Gamma \vdash (\neg c)(P \mid Q) :$		$Proc(u_3)[\lambda_P \cup \lambda_Q, c]$	Par T-Finalize1

○

En d'autres termes, $\Gamma \vdash B : Proc(u_3)[\lambda_P \cup \lambda_Q, c]$

Troisième possibilité La troisième réduction est symétrique de la deuxième. On se contente d'échanger P et Q. Nous ne détaillerons pas la preuve, essentiellement identique.

R-Zero-Res – Cas d'initialisation

On conclut immédiatement grâce au fait que **0** est toujours typable, avec n'importe quel type.

R-DPE – DPE-Null – Cas d'initialisation

Nous avons A gardé par $\circledcirc$ et $B = \mathbf{0}$. Comme B est toujours typable et peut prendre n'importe quel type, le cas est prouvé.

R-DPE – DPE-Name – Cas d'initialisation

Notons

$$\begin{cases} A & = & (\nu a : T)(P + Q \mid R) \\ B & = & (\nu a : T)(P \mid R) \ . \end{cases}$$

où Q est gardé par a et $T = Name(\_, e_a)$.
Nous noterons $\Delta = \Gamma, a : T$.
Le typage le plus général de A est

Typage de $P + Q$			
	Si	$\Delta \vdash P : Proc(t_P)[\lambda_P]$	
		$Q : Proc(t_Q)[\lambda_Q]$	
		$t_P = t_Q$	
		$\lambda_P = \lambda_Q$	
$\Rightarrow \Delta \vdash P + Q :$		$Proc(t_P)[\lambda_P]$	Par T-Sum

Typage de $P + Q \mid R$			
	Si	$\Delta \vdash R : Proc(t_R)[\lambda_R]$	
		$u_1 \geq t_P + t_R$	
		$\lambda_1 = \lambda_P \cup \lambda_R$	
		$\lambda_P \cap \lambda_R = \emptyset$	
$\Delta \vdash P + Q :$		$Proc(t_P)[\lambda_P]$	Cf. plus haut
$\Rightarrow \Delta \vdash P + Q \mid R :$		$Proc(u_1)[\lambda_1]$	Par T-Par

Typage de $(\nu a : T)(P + Q \mid R)$			
$\Delta \vdash P + Q \mid R :$		$Proc(u_1)[\lambda_1]$	Cf. plus haut
	Si	$u_2 \geq u_1 + e_a$	
		$\lambda_1 = \lambda_2, a$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u_2)[\lambda_2]$	Par T-Res

○

Nous avons donc $\Gamma \vdash A : Proc(u_2)[\lambda_2]$ avec les conditions suivantes :

$$\begin{cases} \Gamma, a : C \vdash & P : Proc(t_P)[\lambda_P] \\ \Gamma, a : C \vdash & Q : Proc(t_P)[\lambda_P] \\ \Gamma, a : C \vdash & R : Proc(t_R)[\lambda_R] \\ & \lambda_P \cap \lambda_R = \emptyset \\ & \lambda_P \cup \lambda_R = \lambda_2, a \\ & u_2 \geq t_P + t_R + e_a \end{cases}$$

Sous ces hypothèse, prouvons $\Gamma \vdash B : Proc(u_2)[\lambda_2]$.

Typage de $P \mid R$			
$\Delta \vdash P :$		$Proc(t_P)[\lambda_P]$	Par hypothèse
$\Delta \vdash R :$		$Proc(t_R)[\lambda_R]$	Par hypothèse
	Comme	$\lambda_P \cap \lambda_R = \emptyset$	
$\Delta \vdash P + R :$		$Proc(t_P + t_R)[\lambda_P \cup \lambda_R]$	Par T-Par

Typage de $(\nu a : T)(P \mid R)$			
$\Delta \vdash P \mid R :$		$Proc(t_P + t_R)[\lambda_P \cup \lambda_R]$	Cf. plus haut
	Comme	$u_2 \geq t_P + t_R + e_a$	
		$\lambda_P \cup \lambda_R = \lambda_2, a$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u_2)[\lambda_2]$	Par T-Res

○

Ce qui prouve le cas.
□

Annexe E

Généralisation

E.1 Adaptation de Cπ

Commençons par remarquer que toutes les preuves de l'annexe D sont encore valables avec un ensemble de niveaux de ressources $\mathcal{S}$ distinct de $\mathbf{N} \cup \{\infty\}$, à l'exclusion de la règle R-Comm, qui implique les seules règles que nous ayons modifiées, T-Read et T-Write.

E.1.1 Version sans sous-typage

Dans cette section, nous nous contenterons de présenter le cas de subject reduction modifié par la modification de ces règles, rappelées sur la figure 9.2, page 170.

R-Comm – Cas d'initialisation

Notons

$$\begin{cases} A &= (\overline{a}\langle b\rangle.P + Q) \mid (a(x).R + S) \\ B &= P \mid R\{x \leftarrow b\} \end{cases}$$

avec $a \neq \circledcirc$.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

	Typage de $\overline{a}\langle b\rangle.P$		
	Si	$\Gamma \vdash P : Proc(t_P)[\lambda_P]$	
		$a : Name(Chan(C, r, w), \_)$	
		$b : C$	
		$t_P = u_P \oplus r$	
		$v_P \succeq u_P \oplus w$	
$\Rightarrow \Gamma \vdash \overline{a}\langle b\rangle.P :$		$Proc(v_P)[\lambda_P]$	Par T-Write-Rich
	Typage de $\overline{a}\langle b\rangle.P + Q$		
	Si	$\Gamma \vdash Q : Proc(t_Q)[\lambda_Q]$	
		$v_P = t_Q$	
		$\lambda_P = \lambda_Q$	
$\Rightarrow \Gamma \vdash \overline{a}\langle b\rangle.P \oplus Q :$		$Proc(t_Q)[\lambda_Q]$	Par T-Sum
	Typage de $a(x).R$		

	Si	$\Gamma, x : C \vdash R : Proc(t_R)[\lambda_R]$	
		$x \notin \lambda_R$	
		$t_R = u_R \oplus w$	
		$v_R \succeq u_R \oplus r$	
$\Gamma \vdash a :$		$Name(Chan(C, z), \_)$	Cf. plus haut
$\Rightarrow \Gamma \vdash a(x).R : Proc(v_R)[\lambda_R]$		ParT-READ-RICH	
Typage de $a(x).R + S$			
	Si	$\Gamma \vdash S : Proc(t_S)[\lambda_S]$	
		$v_R = t_S$	
		$\lambda_S = \lambda_R$	
$\Rightarrow \Gamma \vdash a(x).R \oplus S :$		$Proc(t_S)[\lambda_S]$	Par T-SUM
Typage de $(\overline{a}\langle b\rangle.P + Q) \mid (a(x).R + S)$			
$\Gamma \vdash \overline{a}\langle b\rangle.P + Q :$		$Proc(t_Q)[\lambda_Q]$	Cf. plus haut
$\Gamma \vdash a(x).R + S :$		$Proc(t_S)[\lambda_S]$	Cf. plus haut
	Si	$\lambda_P \cap \lambda_Q = \emptyset$	
		$u \succeq t_Q \oplus t_S$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u)[\lambda_P \cup \lambda_Q]$	Par T-PAR

○

Le type le plus général de A est donc $Proc(u)[\lambda_P \cup \lambda_Q]$, avec les conditions suivantes :

$$\begin{cases} \Gamma \vdash & P : Proc(u_P \oplus r)[\lambda_P] \\ \Gamma \vdash & Q : Proc(u_P \oplus r)[\lambda_P] \\ \Gamma \vdash & a : Name(Chan(C, r, w), \_) \\ \Gamma \vdash & b : C \\ \Gamma, x : C \vdash & R : Proc(u_R \oplus w)[\lambda_R] \\ \Gamma \vdash & S : Proc(u_R \oplus w)[\lambda_R] \\ & c \notin \lambda_R \\ & \lambda_P \cap \lambda_R = \emptyset \\ & u \succeq u_P \oplus u_R \oplus r \oplus w \ . \end{cases}$$

De ces hypothèses, nous pouvons tirer la dérivation suivante :

Typage de $R\{x \leftarrow b\}$			
$\Gamma, x : C \vdash R :$		$Proc(u_R \oplus w)[\lambda_R]$	Par hypothèse
$\Gamma \vdash b :$		C	Par hypothèse
	Comme	$x \notin \lambda_R$	
$\Rightarrow \Gamma \vdash R\{x \leftarrow b\} :$		$Proc(u_R \oplus w)[\lambda_R]$	Par *Substitution*
Typage de $P \mid R\{x \leftarrow b\}$			
$\Gamma \vdash P :$		$Proc(u_P \oplus r)[\lambda_P]$	Par hypothèse
$\Gamma \vdash R\{x \leftarrow b\} :$		$Proc(u_P \oplus r)[\lambda_R]$	Cf. plus haut
	Comme	$u \succeq u_P \oplus r \oplus u_R \oplus w$	
		$\lambda_P \cap \lambda_R = \emptyset$	
$\Rightarrow \Gamma \vdash P \mid R\{x \leftarrow b\} :$		$Proc(u)[\lambda_P \cup \lambda_R]$	Par T-PAR

○

Ce qui prouve le cas.

E.1.2 Version avec sous-typage (fragment)

Comme seule les règles T-Chan-Sub, T-Read-Sub et T-Write-Sub utilisent les paramètres dr et dw, nous nous contenterons de présenter le cas de subject reduction modifié par la modification de ces règles, rappelées sur la figure 10.3, page 180.

R-Comm – Cas d'initialisation

Nous emploierons les mêmes notations.

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

	Typage de $\overline{a}\langle b\rangle.P$		
	Si	$\Gamma \vdash P : Proc(t_P)[\lambda_P]$	
		$a : Name(Chan(C, r, w, dr, dw), \_)$	
		$b : C$	
		$t_P = u_P \oplus r$	
		$dw' \succeq dw$	
		$v_P \succeq u_P \oplus w \oplus dw'$	
$\Rightarrow \Gamma \vdash \overline{a}\langle b\rangle.P :$		$Proc(v_P)[\lambda_P]$	Par T-Write-Sub
	Typage de $\overline{a}\langle b\rangle.P + Q$		
	Si	$\Gamma \vdash Q : Proc(t_Q)[\lambda_Q]$	
		$v_P = t_Q$	
		$\lambda_P = \lambda_Q$	
$\Rightarrow \Gamma \vdash \overline{a}\langle b\rangle.P \oplus Q :$		$Proc(t_Q)[\lambda_Q]$	Par T-Sum
	Typage de $a(x).R$		
	Si	$\Gamma, x : C \vdash R : Proc(t_R)[\lambda_R]$	
		$x \notin \lambda_R$	
		$t_R = u_R \oplus w$	
		$dr' \succeq dr$	
		$v_R \succeq u_R \oplus r \oplus dr'$	
$\Gamma \vdash a :$		$Name(Chan(C, r, w, dr, dw), \_)$	Cf. plus haut
$\Rightarrow \Gamma \vdash a(x).R :$		$Proc(v_R)[\lambda_R]$	Par T-Read-Sub
	Typage de $a(x).R + S$		
	Si	$\Gamma \vdash S : Proc(t_S)[\lambda_S]$	
		$v_R = t_S$	
		$\lambda_S = \lambda_R$	
$\Rightarrow \Gamma \vdash a(x).R \oplus S :$		$Proc(t_S)[\lambda_S]$	Par T-Sum
	Typage de $(\overline{a}\langle b\rangle.P + Q) \mid (a(x).R + S)$		
$\Gamma \vdash \overline{a}\langle b\rangle.P + Q :$		$Proc(t_Q)[\lambda_Q]$	Cf. plus haut
$\Gamma \vdash a(x).R + S :$		$Proc(t_S)[\lambda_S]$	Cf. plus haut
	Si	$\lambda_P \cap \lambda_Q = \emptyset$	
		$u \succeq t_Q \oplus t_S$	
$\Rightarrow \Gamma \vdash A :$		$Proc(u)[\lambda_P \cup \lambda_Q]$	Par T-Par

○

Le type le plus général de A est donc $Proc(u)[\lambda_P \cup \lambda_Q]$, avec les conditions suivantes :

$$\left\{\begin{array}{ll}
\Gamma \vdash & P : Proc(u_P \oplus r)[\lambda_P] \\
\Gamma \vdash & Q : Proc(u_P \oplus r \oplus dr)[\lambda_P] \\
\Gamma \vdash & a : Name(Chan(C, r, w, dr, dw), _) \\
\Gamma \vdash & b : C \\
\Gamma, x : C \vdash & R : Proc(u_R \oplus w)[\lambda_R] \\
\Gamma \vdash & S : Proc(u_R \oplus w \oplus dw)[\lambda_R] \\
 & c \notin \lambda_R \\
 & \lambda_P \cap \lambda_R = \emptyset \\
 & u \succeq u_P \oplus u_R \oplus r \oplus w \oplus dr \oplus dw \ .
\end{array}\right.$$

De ces hypothèses, nous pouvons tirer la dérivation suivante :

Typage de $R\{x \leftarrow b\}$			
$\Gamma, x : C \vdash R :$		$Proc(u_R \oplus w)[\lambda_R]$	Par hypothèse
$\Gamma \vdash b :$		C	Par hypothèse
	Comme	$x \notin \lambda_R$	
$\Rightarrow \Gamma \vdash R\{x \leftarrow b\} :$		$Proc(u_R \oplus w)[\lambda_R]$	Par *Substitution*
Typage de $P \mid R\{x \leftarrow b\}$			
$\Gamma \vdash P :$		$Proc(u_P \oplus r)[\lambda_P]$	Par hypothèse
$\Gamma \vdash R\{x \leftarrow b\} :$		$Proc(u_P \oplus r)[\lambda_R]$	Cf. plus haut
	Comme	$u \succeq u_P \oplus r \oplus u_R \oplus w$	
		$\lambda_P \cap \lambda_R = \emptyset$	
$\Rightarrow \Gamma \vdash P \mid R\{x \leftarrow b\} :$		$Proc(u)[\lambda_P \cup \lambda_R]$	Par T-Par

○

Ce qui prouve le cas.

E.2 Adaptation des CA

Commençons par remarquer que toutes les preuves de l'annexe A sont encore valables avec un ensemble de niveaux de ressources $\mathcal{S}$ distinct de $\mathbf{N} \cup \{\infty\}$.

Dans cette section, nous nous contenterons de présenter le cas de subject reduction modifié par l'ajout simultané du système raffiné et et des niveaux de ressources enrichis, rappelé sur la figure 11.3, page 192.

Cas de R-Open Notons

$$\left\{\begin{array}{lcl} A & = & \texttt{open}\ a.P \mid a\,[\overline{\texttt{open}}\ a.Q \mid R] \\ B & = & P|Q|R \end{array}\right.$$

La réduction de type la plus générale pour A dans un environnement Γ est donnée par

Typage de $\overline{\texttt{open}}\ a.Q$			
	Si	$Q : Proc(t_Q)[T_Q]$	
		$q = t_Q$	
		$a : Amb(s_a, e_a, r_a, q_a)[T_a]$	
		$T_Q = T_a$	
		$\forall w, \forall h, w \succeq v_q \wedge w \oplus h = s_a \Rightarrow h \preceq r_a$	
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ a.Q :$		$Proc(v_Q)[T_Q]$	Par T-BetterCoOpen

Typage de $\overline{\texttt{open}}\ a.Q \mid R$			
	Si	$R : Proc(t_R)[T_R]$	
		$T_R = T_Q$	
		$u_1 \succeq v_Q \oplus t_R$	
$\Rightarrow \Gamma \vdash \overline{\texttt{open}}\ a.Q \mid R :$		$Proc(u_1)[T_Q]$	Par T-BetterCoOpen
Typage de $a\,[\overline{\texttt{open}}\ a.Q \mid R]$			
$\Gamma \vdash a :$		$Amb(s_a, e_a, r_a, q_a)[T_a]$	Par hypothèse
	Si	$s_a = u_1$	
		$u_2 \succeq e_a$	
	Comme	$T_a = T_Q$	
$\Rightarrow \Gamma \vdash a\,[\overline{\texttt{open}}\ a.Q \mid R] :$		$Proc(u_2)[T_2]$	Par T-Amb
Typage de $\texttt{open}\ a.P$			
	Si	$P : Proc(t_P)[T_P]$	
$\Gamma \vdash a :$		$Amb(s_a, e_a, r_a, q_a)[T_a]$	Par hypothèse
	Si	$T_a = T_P$	
		$u_3 \oplus e_a \succeq r_a \oplus q_a$	
$\Rightarrow \Gamma \vdash \texttt{open}\ a.P :$		$Proc(u_3)[T_P]$	Par T-BetterOpen
Typage de $\texttt{open}\ a.P \mid a\,[\cdots]$			
$\Gamma \vdash a\,[\overline{\texttt{open}}\ a.Q \mid R] :$		$Proc(u_2)[T_2]$	Cf. plus haut
$\Gamma \vdash \texttt{open}\ a.P :$		$Proc(u_3)[T_P]$	Cf. plus haut
	Si	$T_P = T_2$	
		$u_4 \succeq u_2 \oplus u_3$	
$\Rightarrow \Gamma \vdash \texttt{open}\ a.P \mid a\,[\cdots] :$		$Proc(u_4)[T_P]$	Par T-Par

○

Le type le plus général de A est donc $Proc(u_4)[T_a]$, avec les conditions nécessaires (pas forcément minimales) suivantes (nous avons supprimé les variables intermédiaires inutiles) :

$$\begin{cases} \Gamma \vdash & Q : Proc(t_Q)[T_a] \\ & a : Amb(s_a, e_a, r_a, q_a)[T_a] \\ \Gamma \vdash & R : Proc(t_R)[T_a] \\ \Gamma \vdash & P : Proc(t_P)[T_a] \\ & u_4 \succeq t_P \oplus t_Q \oplus t_R \end{cases}$$

En particulier, nous avons instancié $\forall w, \forall h, w \succeq v_q \wedge w \oplus h \preceq s_a \Rightarrow h \preceq r_a$ avec $w \leftarrow v_q$ et $h \leftarrow t_R$. Comme $v_Q \oplus t_R \preceq u_1 = s_a$, nous en déduisons $t_R \preceq r_a$. Par suite, $u_4 \succeq u_2 \oplus u_3 \succeq e_a \oplus u_3 \succeq r_a \oplus q_a \succeq t_P \oplus t_Q \oplus t_R$.

Nous allons prouver que, sous ces hypothèses, $\Gamma \vdash B : Proc(u_4)[T_a]$.

Typage de $Q \mid R$			
$\Gamma \vdash Q :$		$Proc(t_Q)[T_a]$	Par hypothèse
$\Gamma \vdash R :$		$Proc(t_R)[T_a]$	Par hypothèse
$\Rightarrow \Gamma \vdash Q \mid R :$		$Proc(t_Q \oplus t_R)[T_a]$	Par T-Par
Typage de $P \mid Q \mid R$			
$\Gamma \vdash P :$		$Proc(t_P)[T_a]$	Par hypothèse
$\Gamma \vdash Q \mid R :$		$Proc(t_Q \oplus t_R)[T_a]$	Cf. plus haut
	Comme	$u_4 \succeq t_P \oplus t_Q \oplus t_R$	
$\Rightarrow \Gamma \vdash P \mid Q \mid R :$		$Proc(u_4)[T_a]$	

○

Ce qui prouve le cas.