UNIVERSITÉ NICE SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour l'obtention du grade de

Docteur en Science

de l'Université Nice Sophia Antipolis

Mention : Informatique

présentée et soutenue par

Johan GRANDE

CONCEPTION ET IMPLÉMENTATION D'UN LANGAGE DE PROGRAMMATION CONCURRENTE MODULAIRE

Thèse dirigée par Manuel SERRANO et Gérard BOUDOL
préparée à l'Inria Sophia Antipolis dans l'équipe Indes
soutenue le 28 septembre 2015

Jury :

Denis Caromel	Professeur	Université Nice Sophia Antipolis	*président du jury*
Emmanuel Chailloux	Professeur	Université Pierre et Marie Curie	*rapporteur*
Theo D'Hondt	Professeur	Vrije Universiteit Brussel	*rapporteur*
Jérôme Vouillon	Chargé de recherche	Université Paris Diderot	*examinateur*
Gérard Boudol	Chercheur émérite	Inria	*directeur de thèse*
Manuel Serrano	Directeur de recherche	Inria	*directeur de thèse*

Résumé

La programmation concurrente à mémoire partagée est un modèle classique de concurrence qui permet notamment de tirer parti des processeurs multicœurs aujourd'hui très répandus dans les ordinateurs personnels. Les programmes concurrents sont sujets au problème des interblocages, notoirement difficiles à prévoir et à éliminer, en particulier dans le cas de l'utilisation du mécanisme de synchronisation très populaire que sont les mutex.

Dans cette thèse nous avons travaillé à rendre plus aisée la programmation avec des mutex en étudiant des méthodes d'évitement des interblocages. Nous avons d'abord étudié une méthode utilisant une analyse statique par un système de types et d'effets, puis une variante de cette méthode dans un langage à typage dynamique.

La seconde méthode est celle que nous avons le plus développée. Elle combine prévention et évitement des interblocages pour fournir une fonction de verrouillage sans interblocages expressive et utilisable. Nous l'avons implémentée sous forme d'une bibliothèque Hop (dialecte de Scheme). Ce faisant, nous avons développé un algorithme sans famine pour l'acquisition simultanée d'un nombre arbitraire de mutex, et identifié le concept d'*interblocage asymptotique*. Nous avons également été amenés à proposer une optimisation des exceptions (blocs `finally`).

Nos tests de performances semblent indiquer un impact négligeable de l'utilisation de notre bibliothèque sur des applications concurrentes réelles. La majeure partie de notre recherche pourrait être appliquée à d'autres langages de programmation structurée tels que Java.

Abstract

Shared-memory concurrency is a classic concurrency model which, among other things, makes it possible to take advantage of multicore processors that are now widespread in personal computers. Concurrent programs are prone to deadlocks which are notoriously hard to predict and debug. Programs using mutexes, a very popular synchronization mechanism, are no exception.

In this thesis we studied deadlock avoidance methods with the aim of making programming with mutexes easier. We first studied a method that uses a static analysis by means of a type and effect system, then a variation on this method in a dynamically typed language.

We developed more the second method. It mixes deadlock prevention and avoidance to provide an easy-to-use and expressive deadlock-free locking function. We implemented it as a Hop (dialect of Scheme) library. This lead us to develop a starvation-free algorithm to simultaneously acquire an arbitrary number of mutexes, and to identify the concept of *asymptotic deadlock*. While doing so, we also developped an optimization of exceptions (`finally` blocks).

Our performance tests seem to show that using our library has negligible impact on the performance of real-life applications. Most of our work could be applied to other structured programming languages such as Java.

Remerciements

Je remercie Manuel Serrano, pour l'efficacité de nos discussions et tout le reste. Je remercie Gérard Boudol ainsi que Thomas Gazagnaire et Tamara Rezk avec qui j'ai eu l'occasion de travailler, Cyprien Nicolas pour avoir répondu à mes questions de débutant en Scheme, et tous les membres de l'équipe Indes pour l'accueil qui m'y a été fait tout au long de ma thèse.

Je remercie les membres de mon jury de thèse. Je remercie l'Inria et l'Université Nice Sophia Antipolis, qui constituent un contexte nécessaire et agréable.

Je remercie Hélène Collavizza et les autres enseignants de Polytech'Nice qui ont guidé mes premiers pas dans l'enseignement.

Je remercie Rémi Grande et toute ma famille et mes amis pour leur soutien, leurs encouragements.

Je remercie enfin les auteurs de tous les logiciels (pour la plupart gratuits et *libres*) qui ont pu m'être utiles, tant les outils de travail que les composants système, les applications courantes, les services Web, etc.

Ce document a été réalisé à l'aide de Pandoc et Latex,
ainsi que Gnumeric, gnuplot, Inkscape, KolourPaint, PDFtk et Python.

Table des matières

Chapitre 1

Introduction

Contents

Notre travail se place dans le cadre de la concurrence, c'est-à-dire la possibilité que plusieurs sous-programmes d'un programme informatique ne soient pas exécutés en séquence, mais progressent séparément les uns des autres, plus ou moins indépendamment. Cela recouvre d'une part les situations dans lesquelles plusieurs éléments d'un programme doivent sembler fonctionner en même

temps (par exemple, dans un navigateur Web, les sous-programmes chargés de la communication avec les serveurs, de l'affichage des pages, de l'interaction avec l'utilisateur, etc.), et d'autre part le *parallélisme*, qui désigne le fait que plusieurs opérations soient effectivement exécutées simultanément, avec un gain de performances.

Nous avons travaillé dans le cadre de la concurrence à mémoire partagée et à programme unique. Dans ce modèle, un programme est constitué d'un ou plusieurs *threads* [1] exécutables en parallèle et qui ont accès à un espace mémoire commun.

La concurrence à mémoire partagée permet de tirer avantage des architectures multiprocesseurs ou multicœurs à mémoire partagée (MIMD dans la taxonomie de Flynn [22]), très répandues dans les ordinateurs personnels depuis la fin des années 2000. Pour cela, l'environnement d'exécution répartit l'exécution des threads d'un programme sur le ou les processeurs qui lui sont attribués par le système d'exploitation (on parle de *vrai* parallélisme ou de parallélisme *physique*). Il est cependant possible d'implémenter des threads uniquement par du temps partagé sur un seul processeur (on parle de *pseudo-parallélisme*) ; c'est le cas du langage OCaml par exemple.

Enfin, nous avons supposé dans notre travail un ordonnancement *préemptif* des threads, dans lequel chaque thread peut être interrompu à tout moment pour être repris ultérieurement, plutôt qu'un ordonnancement *coopératif* dans lequel chaque thread s'exécute sans interruption jusqu'à ce qu'il rende la main.

La concurrence pose la question de la coordination temporelle, ou *synchronisation*, des threads. Il existe de nombreux mécanismes de synchronisation. Dans cette thèse, nous nous sommes intéressés aux mécanismes dits d'*exclusion mutuelle* et plus précisément aux *mutex*. Les mécanismes d'exclusion mutuelle sont notamment utilisés pour gérer les *conditions de concurrence* (*race conditions*), dans lesquelles plusieurs threads tentent simultanément d'accéder à une ressource partagée (telle qu'une valeur en mémoire), avec un résultat possiblement aberrant.

Un problème récurrent dans le cadre de la concurrence est celui des *interblocages*. Lors de l'utilisation de certains mécanismes de synchronisation tels que les mutex, des threads doivent attendre des actions d'autres threads, autrement dit ceux-ci sont *bloqués* par ceux-là. Un inter-blocage est une situation dans laquelle deux threads ou plus se bloquent les uns les autres sans qu'aucun ne puisse jamais progresser.

Les interblocages sont en particulier un obstacle à l'écriture de programmes complexes car la composition (mise en parallèle ou autre) de deux programmes sans interblocage peut être sujette aux interblocages. En d'autres termes, le problème des interblocages est un obstacle à la modularité.

Dans cette thèse, nous nous sommes donné pour but de faciliter l'écriture de programmes concurrents complexes en proposant un mécanisme d'exclusion mutuelle à la fois simple d'utilisation, expressif et sans interblocage. À cette fin, nous avons pris pour point de départ un travail antérieur de Boudol ([12] alias Boudol09) qui offre, dans un langage jouet, un mécanisme d'évitement des

1. On parle aussi de *fils d'exécution* ou de *processus légers*.

interblocages utilisant une analyse statique des programmes par un système de types et d'effets. Nous avons travaillé à étendre ce mécanisme à un langage généraliste « à taille réelle ».

Dans la suite de cette introduction, nous présenterons plusieurs mécanismes de synchronisation courants et poserons les premières définitions formelles sur lesquelles reposent notre travail. Nous présenterons également plusieurs méthodes connues pour empêcher les interblocages, et en particulier celle de Boudol09.

1.1 Mécanismes de synchronisation classiques

Il existe de nombreux mécanismes de synchronisation. Certains imposent des conditions simples sur l'exécution des threads. Ainsi, une *barrière* entre plusieurs threads impose que tous franchissent un certain point de leur codes sources respectifs simultanément.

Par contraste, la *mémoire transactionnelle* est un mécanisme de plus haut niveau qui permet d'imposer des conditions d'atomicité sur des opérations d'accès à la mémoire, et masque les opérations de synchronisations à proprement parler qui les implémentent. La mémoire transactionnelle gagne en popularité et est l'objet de nombreux travaux récents [27, 34, 41].

Les partisans de la mémoire transactionnelle font notamment remarquer la difficulté de programmer avec le mécanisme de plus bas niveau que sont les mutex. Nous partageons ce constat, mais avons choisi d'y répondre en étudiant la possibilité de rendre les mutex plus faciles à utiliser en éliminant le problème des interbocages.

1.1.1 Exclusion mutuelle

L'*exclusion mutuelle* est un schéma de synchronisation très utilisé dans lequel il existe une ressource ou une opération qui ne doit pas être utilisée ou effectuée par plus d'un thread à la fois. Dans ce schéma, lorsqu'un thread est en train d'utiliser une ressource, les autres threads qui tentent d'y accéder sont typiquement mis en attente jusqu'à ce que la ressource soit à nouveau disponible. Il arrive également que l'accès soit simplement refusé.

Dans le cadre de la programmation structurée, on appelle *section critique* un bloc de code dont l'exécution par un thread *exclut* l'exécution d'autres opérations par d'autres threads.

Voici quelques exemples de mécanismes d'exclusion mutuelle.

1.1.2 Algorithme de la boulangerie de Lamport

L'algorithme de la boulangerie, inventé en 1974 par Lamport ([32] alias Lamport74), présente un algorithme d'exclusion mutuelle réparti dans un contexte de concurrence à mémoire distribuée dans lequel chaque processeur (on dira *processus* comme dans [33] alias Lamport76) peut écrire sa propre mémoire et

lire la mémoire de tout processus. Lamport76 généralise cet algorithme en le rendant plus paramétrable. L'algorithme de la boulangerie a la particularité de ne pas reposer sur un mécanisme d'exclusion mutuelle sous-jacent tel que « test and set » (Section 1.1.4.4).

L'algorithme de Lamport74 implémente des *sections critiques* : le code du programme peut contenir des sections particulières dites « critiques » mutuellement exclusives : à un instant donné, au plus un processus peut être en train d'exécuter une section critique. Lamport76 généralise les sections critiques en les paramétrant par un *mode*, deux sections critiques n'étant mutuellement exclusives que si leurs modes sont en *conflit*. Cela permet en particulier aux processus d'accéder de façon mutuellement exclusive à un nombre arbitraire de ressources : il suffit d'utiliser comme mode de chaque section critique l'ensemble des ressources à réserver, et de spécifier que deux modes sont en conflit si et seulement si leur intersection est non vide.

En revanche, Lamport76 ne permet pas d'imbriquer des sections critiques (qu'un processus entre dans une section critique alors qu'il se trouve déjà dans un autre). Ceci a pour conséquence que l'algorithme de la boulangerie n'engendre pas d'*interblocages*. En effet, un processus ne peut pas à la fois être bloqué (être en attente pour entrer dans une section critique) et bloquer d'autres threads (être déjà dans une section critique).

Dans Lamport76, les sections critiques sont également paramétrées par une *condition* de sorte qu'un processus ne peut entrer dans une section critique que si la condition associée est vraie.

```
region mode when condition do
  section critique
od
```

L'algorithme de la boulangerie utilise une queue de processus en attente. On peut également le voir comme une queue. Lorsqu'un processus P commence à exécuter l'instruction region, il entre à la fin de la queue. P peut alors entrer dans sa section critique lorsque

- la condition de cette section critique est vraie ;
- aucun processus exécutant une section critique n'est en conflit avec P ;
- aucun processus devant P dans la queue n'est en conflit avec P.

Nous avons opté pour un mécanisme semblable de queue dans notre implémentation d'une opération de verrouillage de plusieurs mutex (Section 3.4).

L'algorithme de Lamport76 permet également de donner des priorités au différents processus ; ces priorités ont précédence sur l'ordre des processus dans la queue.

La queue des processus en attente n'est pas représentée par une structure de donnée FIFO partagée mais par un numéro attribué à chaque processus, représentant sa position dans la queue – d'où l'analogie avec une boulangerie ou toute boutique dans laquelle chaque client prend un numéro pour être servi. L'algorithme permet aux processus exécutant une instruction region de s'attribuer des numéros croissants (non strictement) et utilise l'ordre sur l'identifiant unique et immutable de chaque processus pour départager ceux qui obtiennent des numéros identiques.

Lorsque les conditions pour qu'un processus entre dans sa section critique ne sont pas remplies, celui-ci doit attendre. L'algorithme de la boulangerie ne spécifie pas de mécanisme d'attente particulier. Il peut utiliser un mécanisme d'attente active (une boucle vide) ou un mécanisme d'attente préexistant.

1.1.3 Moniteurs ; sémaphores

À l'inverse de l'algorithme de la boulangerie qui effectue des opérations sur l'ensemble des threads existants, il est fréquent que les mécanismes d'exclusion mutuelle dans les systèmes à mémoire partagée utilisent des structures de données partagées simples, associées à des opérations qui garantissent l'exclusion mutuelle des threads qui les utilisent. Une telle structure de données est appelée un *moniteur*. La notion de moniteur fut introduite par Brinch Hansen et Hoare en 1973-1974 [15, 28] ; nous l'entendons ici au sens large, et considérons qu'elle englobe par exemple les sémaphores, qui préexistaient à cette notion. De même, l'article de Hoare présentait les moniteurs comme des objets, au sens de la programmation orientée objet, mais nous considèrerons que le concept s'applique en dehors de ce paradigme de programmation.

Un *sémaphore*, mécanisme décrit en 1965 par Dijkstra, est une variable entière positive que les threads peuvent atomiquement incrémenter ou décrémenter, telle que lorsque sa valeur est `0`, un thread cherchant à la décrémenter soit mis en attente jusqu'à ce que ce soit possible, c'est-à-dire jusqu'à ce qu'un autre thread ait incrémenté le sémaphore.

Initialisé à une valeur strictement positive, un sémaphore peut représenter le nombre d'unités disponibles d'une certaine ressource, que les threads peuvent emprunter et restituer. Initialisé à zéro, un sémaphore peut représenter une quantité d'objets produits par un ou plusieurs threads et consommés par d'autres. On appelle *sémaphore binaire* un sémaphore qui ne peut prendre que les deux valeurs `0` et `1`. Ajouter aux sémaphores binaires la notion de *propriétaire* nous permet d'introduire le mécanisme d'exclusion mutuelle sur lequel nous avons travaillé, à savoir celui des mutex.

1.1.4 Mutex

Un **mutex** (pour **ex**clusion **mut**uelle), ou *verrou*, est un objet dont l'état peut être soit **verrouillé** (ou *pris*) par un thread unique (qui est alors le **propriétaire** du mutex), soit **déverrouillé** (ou *libre*). Un thread peut devenir propriétaire d'un mutex grâce à une opération d'**acquisition**, bloquante, qui consiste à attendre que le mutex soit libre (s'il ne l'est pas déjà) avant de le **prendre effectivement**. Le thread doit alors **relâcher** le mutex pour que d'autres puissent le prendre.

Remarque : Cette définition, que nous utiliserons, n'est pas universelle ; ainsi dans la bibliothèque standard d'OCaml un mutex n'a pas de propriétaire et est donc un simple sémaphore binaire.

1.1.4.1 Verrouillage explicite et verrouillage structuré

Les langages qui offrent des mutex peuvent proposer soit des fonctions indépendantes `lock` et `unlock` pour respectivement acquérir et libérer les mutex (on parle de **verrouillage explicite**), soit une unique construction prenant un mutex pour le temps de sa portée lexicale (appelée **section critique** [2] ou **bloc synchronisé**) et le relâchant ensuite (on parle de **verrouillage structuré** [3]). On dira qu'un thread `t` **synchronise** un mutex `m` si `t` verrouille `m` de façon structurée.

Dans le cas du verrouillage explicite, le programmeur doit s'assurer qu'à l'exécution, à chaque appel à `lock` corresponde un appel à `unlock`, alors que dans le cas du verrouillage structuré, c'est la syntaxe qui l'impose.

Les langages C et OCaml offrent du verrouillage explicite ; Java, du verrouillage structuré ; Hop [4] propose les deux avec les constructions suivantes :

```
;; Verrouillage explicite
(mutex-lock! m)
...
(mutex-unlock! m)

;; Verrouillage structuré
(synchronize m
   ... )
```

Du point de vue du programmeur, la différence entre le verrouillage structuré et le verrouillage explicite peut être comparée à celle entre la programmation structurée et la programmation avec des `goto`s. D'un côté, le verrouillage explicite donne au programmeur un contrôle fin sur l'exécution de son programme et autorise des algorithmes astucieux. D'un autre côté, ce type de verrouillage est difficile à utiliser correctement et à maintenir puisque les verrouillages et déverrouillages peuvent être très intriqués.

Une construction de verrouillage structuré doit gérer les exceptions et autres ruptures du flot d'exécution ; on pourrait donc s'attendre à des performances moins bonnes que dans le cas du verrouillage explicite. Nous verrons en Section 4.2 une implémentation efficace du verrouillage structuré qui rend cet écart de performances négligeable.

1.1.4.2 Protection de ressources

Les mutex sont typiquement utilisés pour **protéger** des ressources (par exemple, de simples valeurs en mémoire partagée), c'est-à-dire qu'un thread ne peut

2. ou parfois *région critique*
3. Le terme *structured locking* est utilisé dans un sens différent dans [36, p. 36] par exemple.
4. Dans notre travail nous avons utilisé principalement le langage Hop, un dialecte de Scheme (voir Section 3.1.1) ; une majorité des exemples sera donc donnée en Hop ou pseudo-Hop. Nous donnerons également des exemples en C, Java et OCaml ; nous ne présenterons pas ici la syntaxe de ces langages.

accéder à une certaine ressource que s'il possède un certain mutex. Ainsi deux threads ne peuvent accéder à la ressource en même temps.

Dans le cas où les ressources protégées sont des valeurs, selon le langage utilisé, un mutex est soit une ressource qui possède les caractéristiques d'un mutex énoncées ci-dessus, soit un objet à part entière associé à une ressource de façon plus ou moins formelle.

En Java, par exemple, tout objet peut être verrouillé et est donc son propre mutex :

```
// Soit x un objet quelconque, donc aussi un mutex
Object x = ...;

// On peut le verrouiller avant d'y effectuer une opération
synchronized(x) {
    ...  // accès à x
}
```

En Hop en revanche, ainsi qu'en C (avec la bibliothèque pthread) et en OCaml, les mutex sont des objets à part entière et il est de la responsabilité du programmeur de les associer (informellement) à des ressources :

```
;; Soit x une variable
(define x ...)

;; et mx le mutex informellement associé à x
(define mx (make-mutex))

(synchronize mx
   ...  ; accès à x
   )
```

Nous donnons un exemple plus complet d'utilisation des mutex en Section 1.1.6.

1.1.4.3 Verrouillage récursif

Le **verrouillage récursif** est le fait pour un thread de « reverrouiller » un mutex qu'il possède déjà. On appelle **mutex récursif** un mutex qui supporte le verrouillage récursif. Un tel mutex contient un compteur de verrouillages qui est incrémenté à chaque `lock` / entrée dans une section critique, et décrémenté à chaque `unlock` / sortie de section critique. Un mutex verrouillé plusieurs fois redevient libre lorsque le compteur vaut `0`, autrement dit lorsqu'il a été déverrouillé le même nombre de fois.

```
(synchronize m
   ...
   (synchronize m
      ...
      ))
```

Un mutex récursif fonctionne ainsi à l'inverse d'un sémaphore, qui est bloquant lorsque son compteur vaut `0`.

Le verrouillage récursif est pertinent dans le cadre de la modularité ; il permet qu'une fonction synchronisant un mutex soit appelée par du code le synchronisant également.

```
(define (f)
   ...
   (synchronize m
      ... )
   ...
   )

(f)

(synchronize m
   ...
   (f)
   ...
   )
```

Dans le cas de mutex non récursifs, une tentative par un thread d'acquérir un mutex qu'il possède déjà peut résulter (selon les implémentations) soit en un blocage permanent, soit en une erreur.

1.1.4.4 Implémentation

Si des algorithmes tels que celui de la boulangerie permettent d'implémenter des mécanismes d'exclusion mutuelle *ex nihilo*, en pratique les mutex sont généralement implémentés, pour des raisons de performances, en utilisant des mécanismes basiques tels que `test-and-set`, eux-mêmes implémentés au niveau matériel.

`test-and-set` est une instruction qui permet d'assigner une valeur à une variable tout en en lisant sa valeur précédente, et ce atomiquement. Avec cette instruction, la procédure d'acquisition d'un mutex sans propriétaire – avec simplement un attribut `locked` booléen – peut prendre la forme suivante :

```
(define (lock m)
   (while (test-and-set m.locked #t)
      ; m est verrouillé ; le test-and-set ne change pas cela
      (...)  ; procédure d'attente
      )
   ; m était libre ; le test-and-set l'a verrouillé
   )
```

On utilise généralement une procédure d'attente implémentée au niveau de l'ordonnanceur, qui permet au thread de s'endormir et d'être réveillé à un moment

opportun. Néanmoins, on rencontre également l'implémentation **spinlock** qui consiste à utiliser une technique d'*attente active*, autrement dit une procédure d'attente qui se contente de tester répétitivement l'état du mutex jusqu'à pouvoir le verrouiller :

```
(define (lock m)
   (while (test-and-set m.locked #t)
      #f ))
```

Ce comportement est généralement indésirable car il gaspille du temps processeur. Il peut cependant être utile si le temps moyen de libération d'un mutex est inférieur à celui de deux changements de contexte (mise en sommeil du thread courant et activation d'un autre, puis changement inverse).

1.1.5 Variables de conditions

Les *variables de conditions* sont un mécanisme de synchronisation fréquemment utilisé conjointement avec les mutex. Notre travail sur les interblocages ne s'applique qu'aux interblocages engendrés par des mutex, et non à ceux engendrés par des variables de conditions ; en revanche, nous nous sommes assurés que notre implémentation des mutex puisse fonctionner avec des variables de conditions (voir Section 3.2.8).

Les variables de conditions furent introduites par Hoare en même temps que les moniteurs [28]. Ce mécanisme permet à un ou plusieurs threads d'**attendre** sur une variable de condition jusqu'à ce que celle-ci soit **signalée** par un autre thread. Un signal sur une variable de condition représente habituellement le fait qu'une condition sur le programme (une propriété d'une structure de données, par exemple) devient vraie. On utilise typiquement une procédure du type :

```
TANT QUE la condition n'est pas vraie FAIRE
    attendre sur la variable de condition
FIN TANT QUE
```

Une variable de condition est toujours associée à – protégée par – un mutex ; plusieurs peuvent être associées à un même mutex. Dans une procédure d'attente de signal, le mutex associé à la variable de condition est relâché au moment de la mise en attente et réacquis une fois le signal reçu. Le déverrouillage du mutex et la mise en attente sont effectués de façon atomique, pour éviter une exécution du type :

```
1  ;; Soient deux threads A et B, une condition C et une variable de
2  ;; condition cv protégée par un mutex m
3
4  ;; thread A                ;; thread B
5  acquérir m                 ...
6  C est fausse               ...
7  libérer m                  ...
8                             acquérir m
9                             C est [devenue] vraie
10                            signaler cv
11 attendre sur cv            ...
12
13 ;; C est vraie mais A n'as pas vu le signal envoyé par B ;
14 ;; A est bloqué indéfiniment
```

On utilise généralement un même mutex pour protéger une variable de condition et toute ressource sur laquelle porte la condition associée, et on s'assure de la cohérence des opérations sur l'une et sur l'autre en les effectuant dans une même section critique (noter l'ordre des lignes `8` et `9` dans l'exemple ci-dessus).

Voici maintenant un exemple plus complet d'utilisation des mutex et des variables de conditions.

1.1.6 Problème du producteur-consommateur

Le problème du producteur-consommateur est un problème classique de synchronisation, dont nous donnons ici une solution comme exemple d'application des mutex et des variables de conditions.

Le problème est celui du passage de valeurs entre deux threads via un tampon de capacité limitée. L'un des threads produit les valeurs et les place dans le tampon ; l'autre les retire du tampon et les utilise.

En voici une implémentation utilisant un tampon `b`, un mutex `m` protégeant `b`, et deux variables de conditions `not-full` et `not-empty`. Elle suppose qu'on dispose d'un type de données `buffer` (tampon) et des opérations associées `make-buffer`, `buffer-push`, `buffer-pop`, `buffer-full?` et `buffer-empty?` avec la sémantique qu'on devine. Elle suppose également que les deux threads fonctionnent indéfiniment (pas de condition d'arrêt).

Chaque opération d'ajout ou de retrait d'une valeur dans le tampon `b` est protégée par le mutex `m`. L'algorithme est relativement symétrique. Le producteur ajoute chaque valeur produite dans le tampon. Il vérifie auparavant que le tampon ne soit pas plein ; si c'est le cas, il se met en attente sur la variable de condition `not-full`, que le consommateur signale lorsque le tampon cesse d'être plein. Inversement, le consommateur retire une par une les valeurs du tampon. Il vérifie à chaque fois que le tampon ne soit pas vide ; si c'est le cas, il se met en attente sur la variable de condition `not-empty`, que le producteur signale lorsque le tampon cesse d'être vide.

```
(define buffer (make-buffer))  ; Un tampon vide
(define m (make-mutex))
(define not-empty (make-condition-variable))
(define not-full (make-condition-variable))

;; Producteur
(thread
   (while #t  ; Pas de condition d'arrêt
      (let ((x (...)))  ; Produire une valeur x
         (synchronize m
            ; Si buffer est plein, attendre un signal
            ; du consommateur
            (if (buffer-full? buffer)
                (condition-variable-wait! not-full m) )
            ; Si buffer est vide, le consommateur pourrait
            ; être en attente ; on lui envoie un signal
            (if (buffer-empty? buffer)
                (condition-variable-signal! not-empty) )
            ; Mettre x dans buffer
            (buffer-push x buffer)
            )  ; m est relâché
         )))

;; Consommateur
(thread
   (while #t  ; Pas de condition d'arrêt
      ; Prendre une valeur x dans buffer
      (let ((x
             (synchronize m
                ; Si buffer est vide, attendre un signal
                ; du producteur
                (if (buffer-empty? buffer)
                    (condition-variable-wait! not-empty m) )
                ; Si buffer est plein, le producteur pourrait
                ; être en attente ; on lui envoie un signal
                (if (buffer-full? buffer)
                    (condition-variable-signal! not-full) )
                (buffer-pop buffer)
                )))  ; m est relâché
         ; Utiliser x
         (...)
         )))
```

1.2 Interblocages

Un inter-blocage au sens large est une situation dans laquelle plusieurs threads causent le blocage définitif les uns des autres.

Le problème des interblocages constitue une difficulté importante de la pro-

grammation concurrente. Plusieurs études [37, 23, 48], réalisées sur les bases de données de bogues d'applications telles que Mozilla Firefox, OpenOffice, le serveur Web Apache, MySQL et PostgreSQL, indiquent que les bogues dus aux interblocages représentent une part importante des bogues liés à la concurrence en général.

Dans notre travail, nous ne nous intéresserons qu'aux interblocages dus au mécanisme de synchronisation que nous avons étudié : les mutex.

1.2.1 Définitions

On dit d'un thread qu'il est **bloqué** si et seulement s'il est en attente pour verrouiller un ou plusieurs mutex. On dit qu'un thread **progresse** ssi. il n'est pas bloqué. Nous considèrerons qu'un thread en attente d'une entrée-sortie, ou sur une variable de condition, ou exécutant le code suivant, *progresse* :

```
(while #t
   #f )
```

À tout instant de l'exécution d'un programme utilisant des mutex, on définit sur les threads l'ordre partiel suivant :

$t_1 \prec t_2$ ssi. $\exists\ m.\ t_1$ possède m et t_2 attend pour prendre m.

Soient $\prec^+$ et $\prec^*$ les fermetures respectivement transitive et symétrique transitive de $\prec$.

On dit que t_1 **bloque** t_2 ssi. $t_1 \prec^+ t_2$. On dira que t_1 est une **ascendance** de t_2, t_2 une **descendance** de t_1, et que t_1 et t_2 sont des **dépendances** l'un de l'autre.

On dit qu'on programme est dans une situation d'**interblocage** ssi. $\exists\ t.\ t \prec^+ t$, autrement dit ssi. la relation $\prec^*$ n'est pas antisymétrique.

Voici un exemple archétypal de programme dont l'exécution peut résulter en un interblocage. On note `||` l'exécution en parallèle de deux expressions [5] :

```
;; thread t1                          ;; thread t2
(synchronize m  ; 1          ||       (synchronize n  ; 2
   ...                                    ...
   (synchronize n  ; 3                    (synchronize m  ; 4
      ...                                    ...
      ))                                     ))
```

Si les points du programme numérotés 1, 2, 3, 4 sont exécutés dans cet ordre, alors t_1 bloque au point 3 et on a $t_2 \prec t_1$, puis t_2 bloque au point 4 et on a $t_1 \prec t_2 \prec t_1$: interblocage.

On dit qu'un programme est **sans interblocage** si aucune de ses exécutions ne peut résulter en un interblocage.

5. En Hop, `e1 || e2` peut s'écrire `(let ((th (thread e1))) e2 (thread-join! th))`.

1.2.2 Solutions

Il existe de nombreuses méthodes pour gérer le problème des interblocages. Celles-ci sont habituellement classées en trois catégories [17] : *prévenir* statiquement les interblocages, les *éviter* à l'exécution, ou les *détecter* et les résoudre.

La troisième méthode consiste à ne rien faire pour empêcher les interblocages, mais, s'il en survient, à les détecter et à les débloquer en faisant par exemple échouer une ou plusieurs des opérations impliquées. Cette méthode peut donc être qualifiée d'*optimiste*, tandis que les deux premières, qui effectuent des calculs a priori, sont qualifiées de *pessimistes*. Éviter les interblocages consiste à les empêcher en utilisant des informations disponibles à l'exécution, telles que l'ensemble des mutex verrouillés à un instant donné ; il s'agit donc d'une forme dynamique de prévention des interblocages. Les performances des méthodes appartenant à chaque catégorie dépendent non seulement de leur caractéristiques intrinsèques, mais aussi des situations auquelles ces méthodes sont appliquées, comme le montre [9] dans le contexte des systèmes de gestion de bases de données.

L'évitement des interblocages apparaît dans l'algorithme du banquier de Dijkstra (présenté ci-après) ainsi que dans quelques travaux récents [38, 25], mais les méthodes de prévention ou de détection des interblocages sont plus populaires. Des méthodes hybrides existent également (voir Section 1.2.6), dont celle que nous présenterons dans le Chapitre 3. Le problème des interblocages se pose également dans des contextes présentant d'autres contraintes, tels que la programmation concurrente à passage de message, voir à titre d'exemple [56].

Nous présentons maintenant une méthode bien connue de prévention des interblocages, puis l'algorithme du banquier de Dijkstra en tant que représentant classique des mécanismes d'évitement des interblocages. Nous présenterons ensuite un travail de Boudol sur lequel le nôtre repose en grande partie, et enfin, brièvement, quelques autres solutions existantes au problème des interblocages.

1.2.3 Ordonner les mutex

Une méthode classique de prévention des interblocages consiste à imposer un ordre (partiel ou total) sur les mutex avec la condition suivante : un thread peut acquérir un mutex `m2` alors qu'il possède déjà un mutex `m1` si et seulement si `m1 < m2`. Cet ordre peut être conçu et gardé en tête par le programmeur, ou établi et/ou vérifié par le compilateur.

```
(synchronize m1
   ...  ; opération nécessitant m1
   (synchronize m2  ; autorisé seulement si m1 < m2
      ...  ; opération nécessitant m2
      ))
```

De nombreux algorithmes de prévention des interblocages s'inspirent de cette méthode [21, 14, 31, 49, 53]. La solution que nous proposerons en Section 3.2 l'utilise également, en la combinant avec un mécanisme d'évitement des interblocages.

La condition susmentionnée peut être vérifiée à l'exécution (c'est le cas dans notre solution), mais elle est suffisamment simple pour que le programmeur puisse dans certains cas s'assurer de lui-même qu'elle est respectée. Lorsqu'un programme contrevient à cette règle, il peut être transformé pour la respecter en déplaçant des opérations d'acquisition. Par exemple,

```
(synchronize m2
   ...  ; opération nécessitant m2
   (synchronize m1  ; interdit car m1 < m2
      ...  ; opération nécessitant m1
     ))
```

peut être transformé en

```
(synchronize m1
   (synchronize m2
      ...  ; opération nécessitant m2
      ...  ; opération nécessitant m1
     ))
```

1.2.4 Algorithme du banquier de Dijkstra

L'algorithme du banquier [18, 19] est peut-être l'exemple le plus connu de mécanisme d'évitement des interblocages. Pour cette raison, il nous a semblé intéressant de positionner par rapport à cet algorithme la sémantique de Boudol09, sur laquelle repose notre travail. Voici une brève présentation de l'algorithme du banquier ; nous le comparerons ensuite avec la sémantique de Boudol09 en Section 1.2.5.5.

Dans l'algorithme du banquier, des processus « empruntent » et « rendent » des quantités d'une ressource dont il existe une quantité totale fixe (la `capacité`). L'analogie est celle d'un banquier possédant une somme totale d'argent et pouvant accorder des prêts (sans intérêts) dans la limite de cette somme. Ce mécanisme est semblable à celui des sémaphores.

Lors de l'exécution d'un programme utilisant l'algorithme du banquier, on dit que le système est dans un état *sûr* si quel que soit l'enchaînement des futures demandes de prêts, chacune de ces demandes finira par pouvoir être satisfaite. À l'inverse, le système est dans un état non-sûr s'il existe un enchaînement possible des futures demandes de prêts tel qu'au moins une de ces demandes ne puisse jamais être satisfaite.

Voici un exemple d'état non sûr : Supposons un système à 2 processus avec une `capacité` de 4 ¤. Supposons que chacun des deux processus ait un `besoin` de 3 ¤ et ait déjà emprunté 2 ¤. Alors le système est dans un état non-sûr, autrement dit un interblocage pourrait survenir. En particulier, si chacun des deux processus demande à emprunter 1 ¤ supplémentaire, comme le « banquier » n'a plus d'argent à prêter, les processus seront bloqués indéfiniment.

À chaque fois qu'un processus demande à emprunter de l'argent, l'algorithme calcule, au vu du besoin et de la somme actuellement empruntée par chaque

processus, si en accordant le prêt le système resterait dans un état sûr. Dans le premier cas le prêt est accordé ; dans le second il est temporairement refusé (le processus est mis en attente).

L'algorithme du banquier *évite* les interblocages dans le sens où c'est bien en tenant compte des informations disponibles à l'exécution qu'il accorde les prêts demandés de façon sûre.

1.2.4.1 Calcul de la sûreté d'un état

Chaque processus déclare, une fois pour toute la durée de l'exécution, un `besoin` qui est la quantité maximale d'argent qu'il peut avoir emprunté à un instant donné. Autrement dit, un processus qui a emprunté moins que son besoin peut redemander un prêt (et se retrouver bloqué s'il ne l'obtient pas) avant de rendre ce qu'il a déjà emprunté.

On appelle `demande` d'un processus `p` à un instant donné la différence entre son `besoin` et la somme qu'il a déjà empruntée :

Pour déterminer si un état est sûr, le système simule l'exécution complète des processus en séquence, par ordre de demande croissante, autrement dit, pour chaque processus, l'emprunt de `besoin` ¤ en tout (cas le pire), puis la restitution de tout l'argent emprunté.

```
(define (sûr liste-processus somme-disponible)

   ; Trier les processus par demande croissante
   (set! liste-processus
      (sort
         (lambda (p1 p2)
            (< p1.demande p2.demande) )
         liste-processus))

   ; Simuler l'exécution séquentielle des processus
   (for-each
      (lambda (p)
         (if (> p.demande somme-disponible)
             ; then
             (return #f)  ; interblocage rencontré : état non sûr
             )
         (set! somme-disponible (+ somme-disponible p.emprunté))
      liste-processus ))

   #t)  ; pas d'interblocage rencontré : état sûr
```

Si aucun interblocage (demande impossible à satisfaire) n'est rencontré, cet ordonnancement permet de satisfaire toutes les demandes, donc l'état est sûr. Dijkstra démontre – pour l'algorithme généralisé à n types de ressources – que réciproquement, si cet ordonnancement conduit à un interblocage, l'état de départ n'est pas sûr. Nous ne détaillerons pas ici la preuve, ni l'algorithme généralisé.

1.2.5 Sémantique sans interblocage de Boudol

Dans cette section on décrit le travail précédent par Gérard Boudol, *A Deadlock-Free Semantics for Shared Memory Concurrency* ([12] alias Boudol09) qui constitue le point de départ de notre travail.

Boudol09 propose un langage jouet fonctionnel et concurrent à mémoire partagée, de type ML, avec évitement des interblocages. Ce langage offre du verrouillage structuré (`lock ... in ...`) et autorise les sections critiques imbriquées. Comme en Java, dans le langage de Boudol09 on verrouille directement les valeurs mutables (les références), et non des mutex séparés.

```
let x = ref 0 in
lock x in
  ...  (* accès à x *)
```

Le verrouillage de la « racine » d'une structure de données ayant des composantes mutables ne verrouille pas lesdites composantes :

```
let x = ref 0 in
let y = ref x in
lock y in
  ...  (* x n'est pas verrouillé *)
```

Le verrouillage récursif (`lock x in lock x in ...`) est autorisé.

1.2.5.1 Préverrouillage et évitement des interblocages

Dans Boudol09, un programme source est analysé statiquement par un système de types et d'effets, et traduit vers un langage cible dont la sémantique est sans interblocage pour les expressions dites *sûres*, les expressions bien typées du langage source étant traduites vers des expressions sûres du langage cible.

Nous ne décrirons pas ici le langage source et la traduction, mais seulement le mécanisme d'évitement des interblocages à proprement parler. Ce mécanisme consiste en l'ajout, à l'opération de verrouillage structuré classique, d'une condition suffisante pour éviter les interblocages.

Voici d'abord les constructions du langage cible. Celui-ci comprend fonctions et applications, création (νxe) et accès à des pointeurs (qui traduisent les références du langage source), lancement d'un thread, et verrouillage de pointeur.

$$
\begin{array}{lcll}
v & ::= & x \mid (\lambda x e) & \textit{valeurs} \\
e & ::= & v \mid (e_1 e_0) & \textit{expressions} \\
 & \mid & (\nu x e) \mid (\mathbf{!}\, e) \mid (e_0 := e_1) & \\
 & \mid & (\mathsf{thread}\ e) \mid (\mathsf{lock}_\varphi\ e\ \mathsf{in}\ e) \mid (e \backslash p) &
\end{array}
$$

L'expression $e\backslash p$, qui n'est jamais générée à la compilation mais qui apparaît à l'exécution, représente le fait que le pointeur p est possédé par le thread courant durant l'évaluation de e. La rôle de φ dans les expressions lock sera expliqué dans

$$
\begin{array}{rll}
(S, L, \mathbf{E}[(\lambda x e v)]) \mid\mid T) & \to (S, L, \mathbf{E}[\{x \mapsto v\}e] \mid\mid T) & \\
(S, L, \mathbf{E}[(!\, p)]) \mid\mid T) & \to (S, L, \mathbf{E}[v] \mid\mid T) & S(p) = v \\
(S, L, \mathbf{E}[(p := v)]) \mid\mid T) & \to (S[p := v], L, \mathbf{E}[()] \mid\mid T) & \\
(S, L, \mathbf{E}[(\mathsf{thread}\ e)]) \mid\mid T) & \to (S, L, \mathbf{E}[()] \mid\mid T \mid\mid e) & \\
(S, L, \mathbf{E}[(\mathsf{lock}_\varphi\ p\ \mathsf{in}\ e)]) \mid\mid T) & \to (S, L, \mathbf{E}[e] \mid\mid T) & p \in \lceil \mathbf{E} \rceil \\
(S, L, \mathbf{E}[(\mathsf{lock}_\varphi\ p\ \mathsf{in}\ e)]) \mid\mid T) & \to (S, L', \mathbf{E}[(e \backslash p)] \mid\mid T) & p \notin \lceil \mathbf{E} \rceil\ \&\ (\spadesuit) \\
(S, L, \mathbf{E}[(v \backslash p)]) \mid\mid T) & \to (S, L \setminus \{p\}, \mathbf{E}[v] \mid\mid T) & \\
(S, L, \mathbf{E}[(\nu x e)]) \mid\mid T) & \to (S + p, L, \mathbf{E}[\{x \mapsto p\}e] \mid\mid T) & p \notin \mathsf{dom}(S)
\end{array}
$$

$$
(\spadesuit) \quad L \cap (\{p\} \cup (\varphi \setminus \lceil \mathbf{E} \rceil)) = \emptyset,\ L' = L \cup \{p\}
$$

FIGURE 1.1 – Règles de la sémantique opérationnelle de Boudol09. Chaque état du système est représenté par un triplet (S, L, T) où S est la mémoire (*store*) associant à chaque pointeur une valeur, L l'ensemble des pointeurs verrouillés, et T le multiensemble des threads. Chaque thread est de la forme $\mathbf{E}[e]$ où $\mathbf{E}$ est un contexte d'évaluation et e une expression réductible ou une valeur. On désigne par $\lceil \mathbf{E} \rceil$ l'ensemble des pointeurs possédés par un tel thread. Pour une description plus détaillée de ce formalisme, voir Boudol09.

un instant. Les règles de la sémantique opérationnelle du langage sont rappelées en Figure 1.1 ; pour la description complète de ce formalisme, voir Boudol09.

La condition imposée à l'entrée dans une section critique est la suivante : pour acquérir un pointeur `x`, un thread ne doit pas seulement attendre que `x` soit libre, mais aussi que tous les pointeurs qui seront (ou pourront être) verrouillés à l'intérieur de la section critique soient libres également. Ces pointeurs ne sont pas verrouillés immédiatement, mais ils doivent être libres au moment ou `x` est verrouillé. Dans la suite, on dira qu'on verrouille `x` et qu'on **préverrouille** ces autres pointeurs. (Boudol09 n'utilise pas cette terminologie.) Le verrouillage avec préverrouillage est réalisé de façon atomique.

Considérons l'exemple type, déjà mentionné, de programme dont l'exécution peut contenir un interblocage, ici traduit dans le langage source de Boudol09 :

```
(* thread t1 *)                    ||    (* thread t2 *)
lock x in  (* 1 *)                      lock y in  (* 2 *)
  ...                                       ...
  lock y in  (* 3 *)                       lock x in  (* 4 *)
    ...                                       ...
```

Dans cet exemple, au point 1, le thread t_1 est en train d'acquérir `x` pour entrer dans une section critique dans laquelle il prendra `y`. t_1 doit donc préverrouiller `y`. Pour cela, t_1 attend qu'à la fois `x` et `y` soient libres avant d'entrer dans la section critique. De même, au point 2, t_2 ne peut prendre `y` que si `x` est libre. Ainsi, si les points 1, 2, 3, 4 sont exécutés dans cet ordre, alors au point 2 t_2 attend que t_1 sorte de sa section critique interne et l'interblocage potentiel est évité.

Dans le langage cible de Boudol09, l'ensemble des pointeurs à préverrouiller est désigné par φ dans une expression $(\mathsf{lock}_\varphi\ x\ \mathsf{in}\ e)$. Le programme source ci-dessus

serait traduit à la compilation en le programme sûr ci-dessous (en notant `[...]` l'ensemble φ) :

```
(* thread t1 *)                          (* thread t2 *)
lock [y] x in  (* 1 *)        ||         lock [x] y in  (* 2 *)
  ...                                     ...
  lock y in  (* 3 *)                      lock x in  (* 4 *)
    ...                                     ...
```

1.2.5.2 Inférence d'effets

Les préverrouillages n'apparaissent pas dans le langage source de Boudol09. Leur calcul s'effectue à la compilation ; c'est le rôle de l'analyse d'effets et de la traduction du langage source vers le langage cible. Dans cette analyse, l'effet d'une expression est l'ensemble des pointeurs qui peuvent y être verrouillés. Chaque instruction (lock x in e) est alors décorée d'un φ égal à l'effet de e.

Le langage source de Boudol09 correspond essentiellement au langage cible privé des préverrouillages. Ce langage utilise également des *types singletons* et une opération de création de référence particulière, mais nous n'avons pas repris ces éléments dans notre travail et nous ne les présenterons pas ici.

1.2.5.3 Acquérir n mutex

D'un point de vue abstrait, les pré- et postconditions de la sémantique de (lock_φ p in e) sont les mêmes que celles de l'opération d'acquérir en une fois p et tous les pointeurs de φ, puis à relâcher immédiatement tous les mutex de φ (mais pas p).

La sémantique de Boudol09 spécifie uniquement la condition que les pointeurs à préverrouiller doivent être libres lors de l'entrée d'un thread dans une section critique ; elle ne spécifie pas de procédure par laquelle s'assurer que cette condition soit remplie. Dans notre travail, nous avons effectivement considéré que cette condition correspondait à l'acquisition momentanée simultanée des pointeurs à préverrouiller ; nous proposons un algorithme pour cette opération, que nous appellerons *verrouillage multiple*, en Section 3.4.

1.2.5.4 Quantité de parallélisme

On désigne par *quantité de parallélisme* (nous ne formaliserons pas cette notion) la plus ou moins grande possibilité que des expressions d'un programme soient exécutées en parallèle. Cette quantité dépend d'abord de la façon dont le programme est organisé en threads. D'autre part, les techniques de prévention ou d'évitement des interblocages *réduisent* le parallélisme d'un programme en interdisant des ordonnancements non sûrs. Cette réduction du parallélisme doit être *suffisante* pour éviter les interblocages mais elle peut ne pas être *nécessaire*.

Par exemple, le programme suivant est sans interblocage même sans méthode particulière d'empêchement des interblocages, car seul un thread, t_2, peut être à la fois bloqueur et bloqué.

```
(* thread t1 *)                    (* thread t2 *)
lock x in  (* 1 *)        ||       lock y in  (* 2 *)
   ...                                ...
   ; ne prend pas y                   lock x in  (* 3 *)
   ...                                   ...
```

Cependant, la condition de sûreté de Boudol09 impose que `x` soit préverrouillé au point 2. Ainsi, si les points 1, 2, 3 sont exécutés dans cet ordre, t_2 ne peut passer le point 2 avant que t_1 ne sorte de sa section critique. Le parallélisme du programme est donc réduit alors que ce n'est pas nécessaire.

Dans d'autres cas, en revanche, le préverrouillage permet plus de parallélisme que d'autres méthodes d'empêchement des interblocages, et en particulier que la simple discipline qui consiste à ordonner les mutex, puisque qu'il permet d'utiliser des motifs d'imbrication de sections critiques qui seraient sinon bannis car pouvant conduire à des interblocages. Nous donnerons un exemple d'un tel programme en Section 4.3.1.2.

1.2.5.5 Comparaison avec l'algorithme du banquier

Tant la sémantique de Boudol09 que l'algorithme du banquier de Dijkstra (voir Section 1.2.4) mettent en jeu un évitement des interblocages. Les deux systèmes présentent toutefois plusieurs différences.

Problème résolu Les deux solutions ne répondent pas a priori au même problème. Pour ramener l'algorithme du banquier au même problème que Boudol09, on peut prendre la généralisation du premier à n ressources, et en considérer le cas particulier où il y a 1 unité de chaque ressource. Chaque ressource correspond alors à un mutex qui peut être « emprunté » par un seul thread à la fois.

Verrouillage explicite ou structuré Boudol09 repose sur le verrouillage structuré, c'est-à-dire l'imbrication des sections critiques `(lock ... in ...)`. Dans l'algorithme du banquier, au contraire, chaque processus qui possède des ressources peut en réemprunter et en rendre dans un ordre arbitraire à condition de ne pas dépasser son *besoin* déclaré. On peut donc considérer que l'algorithme du banquier permet le verrouillage explicite.

***Besoin* statique ou dynamique** Dans l'algorithme du banquier, chaque processus déclare une fois pour toutes la quantité maximale de chaque ressource qu'il pourrait emprunter. Dans le cas particulier considéré, ça signifie que chaque processus déclare l'ensemble des mutex qu'il est susceptible d'emprunter.

Dans Boudol09, au contraire, le besoin de chaque thread change à chaque section critique ; il s'agit de l'ensemble φ. Si on omet les créations de références, le φ d'un `lock` imbriqué est un sous-ensemble du φ du `lock` englobant. On peut donc considérer qu'à chaque section critique imbriquée, un thread peut réduire son besoin, libérant les autres mutex pour les autres threads et autorisant donc plus de parallélisme que si ce besoin était figé pour toute la durée de l'exécution.

Algorithme d'évitement des interblocages [19] présente un algorithme permettant, à chaque pas de l'algorithme du banquier, de calculer avec exactitude, en tenant compte de tout l'état du système, si un prêt possible pourrait conduire à un interblocage (le prêt est refusé) ou si l'exécution pourra se poursuivre quoi qu'il arrive (le prêt est accordé).

Boudol09, au contraire, calcule statiquement et localement (par l'analyse d'effets) une condition suffisante simple (mais pas forcément nécessaire) pour éviter les interblocages : qu'au moment d'un `lock`, tous les mutex de l'ensemble φ du `lock` soient libres.

1.2.6 Autres approches existantes

L'algorithme de [10] évite les interblocages en maintenant une représentation des dépendances entre threads sous la forme d'une matrice d'accessibilité. La méthode Gadara [55] met en œuvre une méthode issue de la théorie du contrôle, à savoir une analyse statique qui modélise un programme par un réseau de Petri.

La méthode Dimmunix [30] combine détection et évitement des interblocages : lorsqu'un interblocage est détecté, le système en calcule une signature et la stocke, puis l'utilise pour éviter des interblocages similaires au cours d'exécutions ultérieures. En d'autres termes, le système acquiert une immunité contre les motifs d'interblocages qu'il rencontre. Les auteurs en proposent une extension originale, nommée Communix [29], qui consiste à partager une base de signatures entre les différentes machines qui l'utilisent.

[57] présente une autre approche utilisant des « verrous fantômes » (*ghost locks*), qui a deux inconvénients. D'une part, le graphe modélisant le système se construit au cours de l'exécution, il est donc généralement incomplet et ne *garantit* pas l'absence d'interblocages. D'autre part, cette méthode induit des réductions de parallélisme importantes : ses auteurs donnent l'exemple du dîner des philosophes dans lequel, une fois la possibilité d'interblocage détectée, la possibilité de manger n'est plus donnée qu'à un philosophe à la fois.

1.3 Bilan

Nous avons choisi d'étudier plus particulièrement la méthode de Boudol09 et son utilisabilité en pratique. Dans la suite de ce document, nous présenterons deux approches différentes de généralisation du travail de Boudol :

Dans le Chapitre 2, nous présenterons une extension inachevée du système de types et d'effets de Boudol09 à un langage généraliste de la famille Caml et les difficultés, liées à l'approche que nous avions choisie, que nous avons rencontrées.

Dans le Chapitre 3, nous présenterons un algorithme dit de *verrouillage multiple* sans famine, que nous avons utilisé pour implémenter l'opération principale de la sémantique de Boudol09. Nous présenterons une combinaison de cette opération à un mécanisme classique de prévention des interblocages pour offrir un mécanisme expressif et modulaire, et utilisable sans analyse statique. Pour

définir la notion d'absence de famine, nous introduirons également le concept d'*interblocage asymptotique* (Section 3.3.1).

Dans le Chapitre 4, nous présenterons les grandes lignes de l'implémentation du mécanisme précédent sous forme d'une bibliothèque dans le langage Hop, ainsi que des tests de performances. Nous décrirons également une optimisation de la gestion des exceptions dans Hop que nous avons été amenés à développer (Section 4.2).

Chapitre 2

Types et effets pour éviter les interblocages dans un langage généraliste

Contents

Nous nous sommes donné pour but de généraliser le travail de Boudol09, et en particulier son système de types et d'effets, à un langage de programmation généraliste. Ce travail est resté inachevé.

Nous avons choisi de prendre pour base le langage Llama, dérivé de Caml, et d'y ajouter une inférence d'effets semblable à celle de Boudol09 pour obtenir un langage nommé **Tesard** [1]. Pour ce faire, nous avons considéré une variante du système de types et d'effets de Boudol09, utilisant notamment des *régions* à la place de types singletons, puis avons développé plusieurs fragments d'une extension de cette variante au langage Llama.

1. « a Type and Effet System to Avoid Race conditions and Deadlocks »

Nous avons identifié d'une part plusieurs difficultés liées à l'approche que nous avions choisie, et d'autre part un sous-problème, celui de l'implémentation de la sémantique du langage cible de Boudol09, auquel la recherche et le développement d'une solution – dans un contexte différent – constitue la suite de notre travail (Chapitre 3).

2.1 Le langage Llama

Llama Light [6] – nous dirons simplement *Llama* – est un langage fortement inspiré de Caml [2, 7]. Son implémentation reprend en partie celles de Caml Light et OCaml.

Les particularités de Llama qui ont motivé notre choix sont :

— un langage moins riche et donc un système de types plus simple à étudier que celui d'OCaml ;
— une implémentation conçue dans le but d'être plus modulaire que celle d'OCaml (à cette époque).

Ce langage nous a permis d'expérimenter sur le typage d'un langage de programmation généraliste ; en revanche, il ne nous aurait pas permis de diffuser notre travail et de le maintenir sur le long terme, pour des raisons purement pratiques : l'auteur n'a jamais mis son implémentation sous une licence d'utilisation claire et a cessé de la maintenir peu après que nous avons commencé à travailler dessus, et, à l'heure de la rédaction du présent document, son site Web a été mis hors-ligne.

Voici maintenant une présentation des principaux aspects du langage. Nous n'en donnerons pas une description exhaustive ; pour plus d'informations, le lecteur pourra se référer à une archive du site officiel de Llama.

2.1.1 Système de types

Llama propose des types prédéfinis (entiers, listes, `unit`, etc.) et permet de déclarer de nouveaux types. Le système de types de Llama comprend des types sommes, des enregistrements, des types flèches (fonctions), des tuples, des types paramétrés (polymorphes) et des types récursifs.

```
type s = Ident [of <type>] | ...                (* type somme *)
type r = { [mutable] ident : <type> ; ... }     (* enregistrement *)
type f = <type> -> <type>                       (* type flèche *)
type t = <type> * ...                           (* tuple *)
type ('a, ...) t = ... 'a ...                   (* type paramétré *)
type 'a t = ... 'a t ...                        (* type récursif *)
```

Les principaux types mutables de Llama sont les enregistrements ainsi que les deux types de base suivants : chaînes de caractères et tableaux.

```
let s : string = "Hello, World!"
let a : int array = [| 2; 3; 5; 7 |]
```

Les références sont un type prédéfini d'enregistrements :

```
# ref 42;;
- : int Pervasives.ref = {contents = 42}
```

Llama dispose également d'un unique type ouvert (type somme extensible) monomorphe `exn` pour les exceptions.

```
# Failure "abc";;          (* Exception prédéfinie *)
- : exn = Failure "abc"
# exception E of int;;     (* Déclaration d'une exception *)
exception E of int
# E 42;;
- : exn = E 42             (* E a été ajoutée au type exn *)
```

Enfin, Llama permet de définir des types abstraits.

```
# type t;;
type t
```

Ceux-ci peuvent par exemple correspondre, dans le cas de l'utilisation de fonctions définies en C, à des structures de données utilisées par des fonctions C et opaque depuis le code Llama.

Le système de types de Llama a, de plus, les particularités suivantes :

- Les définitions (`let`) locales ne sont pas généralisées [54]. Par exemple, le programme suivant ne type pas :
  ```
  let _ =
    let singleton x =
      [x]
    in
    singleton true, singleton 0
  ;;
  ```
  En OCaml, la fonction locale `singleton` recevrait pour type $\forall\ \alpha.\ \alpha \to \alpha$ list. En Llama, cette fonction est monomorphe et ne peut donc être appliquée à un booléen *et* à un entier.
- Il n'y a pas de variables de type faible ; les expressions toplevel sont généralisées immédiatement. En particulier, `let x = ref [];;` ne type pas (on écrira `let x : int list ref = ref [];;` par exemple).
- Le polymorphisme est restreint aux valeurs (cette restriction n'est pas relâchée au sens de [24]).

2.1.2 Constructions du langage

Nous ne détaillerons pas ici la syntaxe concrète du langage. Notons simplement qu'il s'agit d'un langage de type ML proposant les constructions suivantes : `let` / `let rec`, fonction et application, pattern matching, exceptions, `if-then-else`, séquence, boucles `for` et `while`, annotation/contrainte de type, assertion.

Ces constructions peuvent s'employer sans parenthèses, avec les mêmes règles de priorité et d'associativité qu'OCaml.

Llama offre également des modules, mais uniquement dans le cadre de la compilation séparée : chaque unité de compilation correspond à un module ; il n'y a pas de sous-modules ni de foncteurs. Comme en OCaml il est possible de définir des fichiers interface en `.mli`, contenant le type d'un module implémenté dans un fichier `.ml`. Interfaces et implémentations sont compilées en des fichiers objets (respectivement `.lmi` et `.lmo`) ; lorsqu'un module n'a pas de fichier `.mli`, son interface objet (`.lmi`) est inférée à partir de son implémentation (`.ml`).

2.1.3 Concurrence

Llama ne proposait à l'origine aucune forme de concurrence, mais nous y avons porté le système de threads d'OCaml pour ajouter au langage threads et mutex avec les constructions suivantes tirées de Boudol09 :

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \ldots \\
& | & \mathsf{thread}\ \mathit{expr} \\
& | & \mathsf{lock}\ \mathit{expr}\ \mathsf{in}\ \mathit{expr}
\end{array}
$$

$\mathsf{thread}\ e$ lance un nouveau thread et y exécute l'expression e. $\mathsf{lock}\ e_0\ \mathsf{in}\ e_1$ calcule e_0 puis verrouille l'unique mutex associé à e_0 pour exécuter e_1.

Comme dans OCaml, les threads sont implémentés par du temps partagé sur un seul processeur (pseudo-parallélisme).

2.2 Régions

Pour combiner l'inférence d'effets de Boudol09 et le polymorphisme offert par Llama, nous avons choisi d'utiliser des *régions* semblables à celles présentes dans l'analyse d'effets employée dans le ML Kit pour la gestion automatique de la mémoire [52, 51]. Ce mécanisme se substitue à celui de `cref` et des types singletons utilisé dans Boudol09.

Comme dans Boudol09, les régions ne sont pas manipulables par le programmeur ; elles sont introduites dans un programme lors de l'inférence de type et de la traduction vers un langage intermédiaire, qui est alors compilé par le *backend* de Llama. Ce langage n'apparaît que comme une représentation intermédiaire lors de la compilation ; nous en donnerons des exemples dans une syntaxe aussi proche que possible du langage source. Voici une présentation du mécanisme des régions restreint aux constructions du langage de Boudol09 ; nous décrirons sa généralisation aux autres constructions du langage Llama dans la section suivante.

Comme dans Boudol09, on autorise le verrouillage de toutes les valeurs mutables. À cet effet, on associe à chaque valeur mutable une région. Cette région apparaît comme un paramètre du type de cette valeur. Par exemple, que x ait pour type $\mathsf{int\ ref}_\rho$ signifie que x est associé à la région ρ. Le verrouillage de x se traduit alors en le verrouillage de ρ, qui est implémentée par un mutex.

Voici la règle de typage de `lock-in`. Comme dans Boudol09, les jugements sont de la forme

$$\Gamma \vdash e : \varphi, \tau \Rightarrow \bar{e}$$

où Γ est un environnement associant des types à des variables, e l'expression (du langage source) à typer, φ et τ respectivement l'effet et le type inférés de l'expression, et $\bar{e}$ la traduction, dans le langage cible, de e. Les effets sont des ensembles de régions.

$$\frac{\Gamma \vdash e_0 : \varphi_0, \theta\ \mathsf{ref}_\rho \Rightarrow \bar{e_0} \qquad \Gamma \vdash e_1 : \varphi_1, \tau \Rightarrow \bar{e_1}}{\Gamma \vdash \mathsf{lock}\ e_0\ \mathsf{in}\ e_1 : \{\rho\} \cup \varphi_0 \cup \varphi_1, \tau \Rightarrow \bar{e_0}; \mathsf{lock}_{\varphi_1}\ \rho\ \mathsf{in}\ \bar{e_1}}$$

Puisqu'il n'y a pas identité entre régions et pointeurs, on rend explicite (contrairement à Boudol09) la traduction de $\mathsf{lock}\ e_0\ \mathsf{in}\ e_1$ en une séquence : d'abord le calcul de $\bar{e_0}$, puis le verrouillage de ρ.

D'autre part, notre langage n'utilisant pas les types singletons, il ne comporte pas de construction `cref` mais une simple fonction `ref` dont la règle de typage est la suivante :

$$\frac{\Gamma \vdash e : \varphi, \theta \Rightarrow \bar{e}}{\Gamma \vdash \mathsf{ref}\ e : \varphi, \theta\ \mathsf{ref}_\rho \Rightarrow \mathsf{ref}\ \bar{e}}\ \rho \text{ fresh}$$

Au cours de l'inférence de types, lorsque deux types sont unifiés, leurs paramètres de régions le sont également. Par exemple, dans le programme suivant :

```
1 let x1 = ref 0 in
2 let x2 = ref 0 in
3 let l  = [x1; x2] in
4 ...
```

on a $(x_1 : \mathsf{int}\ \mathsf{ref}_{\rho 1})$ et $(x_2 : \mathsf{int}\ \mathsf{ref}_{\rho 2})$; l a pour type $\mathsf{int}\ \mathsf{ref}_{\rho 1}\ \mathsf{list}$ et implique que $\rho_1 = \rho_2$.

Les règles de notre système de type, restreint aux constructions du langage source de Boudol09, sont représentées en Figure 2.1.

2.2.1 Déclaration de région

L'inférence de types et d'effets associe un type et, le cas échéant, une région, à chaque variable présente (syntaxiquement) dans le code du programme. Or une variable peut correspondre à plusieurs valeurs à l'exécution dans un programme tel que :

```
while ... do
  let x = ref 0 in
  ...
done
```

Il peut être pertinent de ne pas associer la même région concrète (autrement dit, le même mutex) à toutes les valeurs représentées par une même variable dans le code du programme. À cette fin, on ajoute au langage cible la construction (let region ρ in e). Cette construction rend explicite la création d'une région ρ *dont l'expression e doit être dans la portée lexicale*, autrement dit qui apparaît comme une variable libre dans e (comme région d'un `lock` ou dans l'effet associé). Cette déclaration est possible à la condition que ρ n'apparaisse pas à l'extérieur de $\bar{e}$, c'est-à-dire ni dans le type de $\bar{e}$ ni dans l'environnement Γ.

L'ensemble des variables libres de région d'une expression n'est pas nécessairement égal à son effet. En particulier, l'expression

```
let x = ref 0 in
(thread
   lock x in
     ...)
```

a un effet vide (les effets ne se propagent pas d'un thread à l'autre), mais a pour variable libre la région de `x`.

Notons frv la fonction donnant l'ensemble des variables libres de région d'une expression, d'un type ou d'un environnement. La règle de typage correspondant à l'introduction de let region est la suivante :

$$\frac{\Gamma \vdash e : \varphi, \tau \Rightarrow \bar{e}}{\Gamma \vdash e : \varphi \setminus \{\rho\}, \tau \Rightarrow \text{let region } \rho \text{ in } \bar{e}} \quad \rho \in frv(\bar{e}) \setminus frv(\tau, \Gamma)$$

et $frv(\bar{e})$ se calcule de la façon suivante :

$$\begin{aligned} frv(\mathsf{lock}_\varphi\ \rho \text{ in } e) &= \varphi \cup \{\rho\} \cup frv(e) \\ frv(\text{let region } \rho \text{ in } e) &= frv(e) \setminus \{\rho\} \\ frv(\text{thread } e) &= frv(e) \\ \ldots &= \ldots \qquad \text{(règles de propagation standard)} \end{aligned}$$

L'introduction d'un let region permet de retirer la région déclarée ρ de l'effet de l'expression résultante ($\varphi \setminus \{\rho\}$ dans la règle ci-dessus) sans risquer d'interblocage. En effet, à l'exécution, jusqu'à l'instant de sa création, la région ρ n'est verrouillée par aucun thread, donc son préverrouillage n'a aucun impact sur le déroulement du programme. Il est donc en particulier légitime de négliger son préverrouillage dans le thread courant, autrement dit de la retirer de l'effet de l'expression considérée. Nous verrons une généralisation de ce raisonnement en Section 3.2.4.1.

2.3 Tesard

Nous avons étendu le mécanisme de régions et d'effets décrit dans la section précédente de plusieurs façons. Dans le langage intermédiaire avec régions, les types de la forme τ_ρ pourront être notés sans mise en forme `t[r]`.

$$\frac{}{\Gamma, x:\tau \vdash x : \emptyset,\ \tau \Rightarrow x} \qquad \frac{}{\Gamma \vdash () : \emptyset,\ \mathsf{unit} \Rightarrow ()}$$

$$\frac{\Gamma, x:\tau \vdash e : \varphi,\ \sigma \Rightarrow \bar{e}}{\Gamma \vdash \lambda x e : \emptyset,\ \tau \xrightarrow{\varphi} \sigma \Rightarrow \lambda x \bar{e}}$$

$$\frac{\Gamma \vdash e_0 : \varphi_0,\ \tau \xrightarrow{\varphi_2} \sigma \Rightarrow \bar{e_0} \qquad \Gamma \vdash e_1 : \varphi_1,\ \tau \Rightarrow \bar{e_1}}{\Gamma \vdash e_0\ e_1 : \varphi_0 \cup \varphi_1 \cup \varphi_2,\ \sigma \Rightarrow \bar{e_0}\ \bar{e_1}}$$

$$\frac{\Gamma \vdash e : \varphi,\ \theta \Rightarrow \bar{e}}{\Gamma \vdash \mathsf{ref}\ e : \varphi,\ \theta\ \mathsf{ref}_\rho \Rightarrow \mathsf{ref}\ \bar{e}}\ \rho \text{ fresh} \qquad \frac{\Gamma \vdash e : \varphi,\ \theta\ \mathsf{ref}_\rho \Rightarrow \bar{e}}{\Gamma \vdash\ !e : \varphi,\ \theta \Rightarrow\ !\bar{e}}$$

$$\frac{\Gamma \vdash e_0 : \varphi_0,\ \theta\ \mathsf{ref}_\rho \Rightarrow \bar{e_0} \qquad \Gamma \vdash e_1 : \varphi_1,\ \theta \Rightarrow \bar{e_1}}{\Gamma \vdash e_0 := e_1 : \varphi_0 \cup \varphi_1,\ \mathsf{unit} \Rightarrow \bar{e_0} := \bar{e_1}}$$

$$\frac{\Gamma \vdash e : \varphi,\ \mathsf{unit} \Rightarrow \bar{e}}{\Gamma \vdash \mathsf{thread}\ e : \emptyset,\ \mathsf{unit} \Rightarrow \mathsf{thread}\ \bar{e}}$$

$$\frac{\Gamma \vdash e_0 : \varphi_0,\ \theta\ \mathsf{ref}_\rho \Rightarrow \bar{e_0} \qquad \Gamma \vdash e_1 : \varphi_1,\ \tau \Rightarrow \bar{e_1}}{\Gamma \vdash \mathsf{lock}\ e_0\ \mathsf{in}\ e_1 : \{\rho\} \cup \varphi_0 \cup \varphi_1,\ \tau \Rightarrow \bar{e_0}; \mathsf{lock}_{\varphi_1}\ \rho\ \mathsf{in}\ \bar{e_1}}$$

$$\frac{\Gamma \vdash e : \varphi,\ \tau \Rightarrow \bar{e}}{\Gamma \vdash e : \varphi \setminus \{\rho\},\ \tau \Rightarrow \mathsf{let\ region}\ \rho\ \mathsf{in}\ \bar{e}}\ \rho \in frv(\bar{e}) \setminus frv(\tau,\ \Gamma)$$

FIGURE 2.1 – Système de types et d'effets de Tesard restreint aux constructions du langage source de Boudol09.

2.3.1 Types

Types verrouillables Nous avons choisi de rendre verrouillables :

- les chaînes de caractères ;
- les tableaux ;
- les enregistrements ayant au moins un champ mutable (dont les références).

On introduit une classe de types (informelle) $\mathsf{lockable}_\rho$. Il s'agit simplement d'un type pouvant être unifié avec tous les types susmentionnés, dont la région est alors unifiée avec ρ :

$$\frac{\Gamma \vdash e_0 : \varphi_0,\ \theta \Rightarrow \bar{e_0} \qquad \theta \sim \mathsf{lockable}_\rho \qquad \Gamma \vdash e_1 : \varphi_1,\ \tau \Rightarrow \bar{e_1}}{\Gamma \vdash \mathsf{lock}\ e_0\ \mathsf{in}\ e_1 : \{\rho\} \cup \varphi_0 \cup \varphi_1,\ \tau \Rightarrow \bar{e_0}; \mathsf{lock}_{\varphi_1}\ \rho\ \mathsf{in}\ \bar{e_1}}$$

où $\theta \sim \mathsf{lockable}_\rho$ si et seulement si θ est de la forme string_ρ ou $\sigma\ \mathsf{array}_\rho$ ou est un enregistrement mutable et associé à ρ.

Polymorphisme de région Les types ayant plusieurs composantes mutables peuvent contenir plusieurs variables de région, par exemple $\mathsf{int}\ \mathsf{ref}_{\rho 1}\ \mathsf{ref}_{\rho 0}$. Les types polymorphes sont quantifiés sur leurs variables de type *et* de région : si un type τ a pour variables de type $\vec{\alpha}$ et pour variables de région $\vec{\rho}$, il sera généralisé

en $\forall\vec{\alpha},\vec{\rho}.\,\tau$. Nous n'avons pas défini de généralisation des types vis-à-vis des variables d'effet.

Lors de la traduction de la déclaration d'un type `t`, ce type se voit attribuer en paramètres de région l'ensemble des paramètres de région de toutes ses composantes. Ainsi, un type somme

```
type t = A | B of int ref | C of int ref ref
```

est traduit en

```
type t['r0,'r1,'r2] = A | B of int ref['r0] | C of int ref['r2] ref['r1]
```

Enregistrements mutables Lorsqu'un enregistrement est mutable, il prend un paramètre de région qui sera utilisé pour le verrouiller, en plus des paramètres de régions de ses éventuelles composantes mutables.

```
type r =                          type r['r0,'r1] =
  { x : string;            =>       { x : string['r1];
    mutable y : int }                   mutable y : int }
```

Types récursifs Lorsqu'un type est mentionné dans sa propre déclaration, il prend comme paramètres de régions ceux qu'il prendrait en ommettant cette mention récursive. Alors la mention récursive prend les mêmes paramètres que le type déclaré. Par exemple, dans

```
type t = A of int ref | T of t
```

le type `t` prend un paramètre de région imposé par le `ref` et la déclaration est traduite en

```
type ['r] t = A of int ref['r] | T of ['r] t
```

Des types mutuellement récursifs prennent le même vecteur de paramètres de région, qui correspondent aux différentes composantes mutables de l'ensemble de la déclaration. Ainsi

```
type t = A of int ref | U of u
and  u = B of int ref | T of t
```

est traduit en

```
type ['r1, 'r2] t = A of int ref['r1] | U of ['r1, 'r2] u
and  ['r1, 'r2] u = B of int ref['r2] | T of ['r1, 'r2] t
```

Exceptions Les exceptions sont un type somme extensible. Nous avons choisi de donner à ce type un unique paramètre de région partagé par toutes les composantes de chaque exception. Par exemple, dans

```
exception E of int ref * int ref
```

les deux références partagent une même région. De même, dans

```
exception E1 of int ref
exception E2 of int ref
let e1 = E1 (ref 0)
let e2 = E2 (ref 0)
let l = [e1; e2]
```

`l` a pour type exn_ρ list et la région ρ est partagée par les deux références.

2.3.2 Expressions de module

Nous ne décrirons pas formellement l'analyse et la traduction d'un module complet, mais seulement celle des expressions de module (top-level) qui contiennent des expressions, c'est-à-dire les déclarations `let ... = ...;;` (ce qui inclut les évaluations « nues » ou déclarations de 0 variables `let _ = ...;;`). Plus précisément, nous décrivons la traduction dans un environnement Γ de `let x = e;;`, les variantes à zéro ou plusieurs variables s'en déduisant naturellement.

On suppose que e est typable avec un jugement de la forme :

$$\Gamma \vdash e : \varphi,\ \tau \Rightarrow \bar{e}$$

L'enjeu du typage et de la traduction de ce type d'expression est d'une part la généralisation de τ, et d'autre part une éventuelle déclaration (top-level, cette fois) de régions utilisées dans e, c'est-à-dire appartenant à $frv(\bar{e})$ ou apparaissant dans τ.

On impose que $frv(\bar{e}) \setminus frv(\Gamma, \tau) = \emptyset$. Ces variables de région étant locales à e, on impose qu'elles soient déclarées localement par un $\mathsf{let\ region}$.

L'effet φ de e, quant à lui, est ignoré. En effet, e se situant au top-level, elle ne peut pas se trouver dans le corps d'un lock.

On distingue alors les expressions de type fonction, dont le type est généralisé, et les autres, auxquelles on impose $ftv(\tau) \setminus ftv(\Gamma) = \emptyset$ (pas de variables de type faibles).

Pour les expressions de type fonction, la généralisation de traduit par :

$$\frac{\Gamma \vdash e : \varphi,\ \tau \Rightarrow \bar{e} \qquad \tau = \theta \xrightarrow{\psi} \sigma \qquad frv(\bar{e}) \setminus frv(\Gamma, \tau) = \emptyset \qquad \vec{\alpha} = ftv(\tau) \setminus ftv(\Gamma) \qquad \vec{\rho} = frv(\tau) \setminus frv(\Gamma)}{\Gamma \vdash \mathsf{let}\ f = e;; \Rightarrow \mathsf{let}\ f : \forall \vec{\alpha}, \vec{\rho}.\ \tau = \bar{e};;}$$

Pour les autres expressions, puisque leur type n'est pas généralisé, il peut contenir des variables de région libres et absentes de l'environnement. On déclare donc ces régions avec une construction region $\vec{\rho}$;;.

$$\frac{\Gamma \vdash e : \varphi, \tau \Rightarrow \bar{e} \qquad \tau \rightsquigarrow \theta \xrightarrow{\psi} \sigma \qquad frv(\bar{e}) \setminus frv(\Gamma, \tau) = \emptyset \qquad ftv(\tau) \setminus ftv(\Gamma) = \emptyset \qquad \vec{\rho} = frv(\tau) \setminus frv(\Gamma)}{\Gamma \vdash \mathsf{let}\ x = e;; \Rightarrow \mathsf{region}\ \vec{\rho};;\ \mathsf{let}\ x : \tau = \bar{e};;}$$

2.3.3 Problèmes et suites

Nous n'avons pas développé la généralisation des types vis-à-vis des effets latents. On pourrait chercher à ajouter à Tesard des étiquettes d'effets à la façon du ML Kit pour obtenir des types de la forme :

$$List.map : \forall \alpha, \beta, \vec{\epsilon}.\ (\alpha \xrightarrow{\epsilon_0} \beta) \xrightarrow{\epsilon_1.\emptyset} \alpha\ list \xrightarrow{\epsilon_2.\{\epsilon_0\}} \beta\ list$$

Chaque effet latent a une étiquette qui peut être référencée dans les autres. Ici, l'effet de `List.map f l` contient uniquement l'effet de `f`.

D'autre part, il serait possible d'autoriser des annotations de région, c'est-à-dire la possibilité d'expliciter régions et effets dans les annotations/contraintes de type, ce qui permettrait notamment :

- L'écriture par le programmeur de fichiers interface `.mli`. Dans notre prototype, les fichiers d'interface compilés (`.lmi`) ne peuvent qu'être générés à partir des fichiers implémentation (`.ml`).
- La création de types abstraits verrouillables. Les types abstraits apparaissent non seulement dans les fichiers interface, mais aussi dans les fichiers implémentation pour le typage de valeurs manipulées en C uniquement.

Cependant, une implémentation prototype de Tesard, basée sur celle de Llama, nous a permis d'identifier plusieurs problèmes liés à notre approche d'utilisation de l'inférence de types et d'effets pour associer valeurs mutables et régions/mutex :

Nombre de variables de région Nous avons fait le choix d'une association à grain fin entre valeurs et régions, en donnant aux types (dans certains cas) un paramètre de région par composante mutable. Des types complexes ayant un grand nombre de composantes mutables peuvent donc avoir un grand nombre de régions ; c'est le cas dans notre compilateur, qui est bootstrappé. À titre d'exemple, dans notre version d'OCamldoc, le type `Odoc_search.result_element` prend 1 970 paramètres de région – dont aucun ne correspond à une région effectivement utilisée à l'exécution, ce programme n'étant pas concurrent. L'unification de types ayant plusieurs centaines de paramètres de région a un impact visible sur les performances de compilation ; cependant notre prototype n'est pas assez abouti pour en faire une évaluation sérieuse.

Réduction du parallélisme À l'inverse, l'utilisation que nous faisons du typage pour associer automatiquement des régions aux valeurs mutables entraine parfois un partage d'une même région entre plusieurs valeurs qui est peu intuitif pour le programmeur et ne correspond pas à son intention, voire empêche d'exprimer un algorithme efficace.

Par exemple, si dans un programme des références sont stockées dans une table de hachage de type `(int, int ref[r0]) Hashtbl.t[r0,r1]`, ces références partagent la région `r0` ; le verrouillage de l'une implique le verrouillage de l'autre et elles ne peuvent donc pas être manipulées en parallèle (sous la protection des régions).

Unification de régions top-level Lorsqu'une région est déclarée au sein d'une expression par un `let region`, cette région n'apparaît pas en dehors du corps du `let region`. Inversement, des régions déclarées au top-level par des constructions `region` sont visibles globalement et sont susceptibles de devoir être unifiées entre elles après que le système de types et d'effets ait introduit lesdites constructions `region`. Par exemple, si on suppose un programme constitué de trois modules dont voici le code source et la traduction :

```
(* a.ml *)

let x = ref 0

(* b.ml *)

let y = ref 0

(* main.ml *)
... [A.x; B.y] ...
```

=>

```
(* a.ml (langage cible) *)
region r
let x : int ref[r] = ref 0

(* b.ml (langage cible) *)
region r
let y : int ref[r] = ref 0

(* main.ml (langage cible) *)
... [A.x; B.y] ...
(* A.r = B.r ? *)
```

Lors de la compilation du module `Main`, il apparaît que les deux régions déclarées dans les modules `A` et `B` doivent être égales.

Si on admet toute forme de modularité et de compilation séparée, les déclarations des régions `A.r` et `B.r` ne peuvent pas être modifiées. Elles doivent donc être unifiées a posteriori par un mécanisme qui reste à spécifier, ou la compilation du programme doit lancer une erreur d'unification de régions.

2.4 Implémentation de la sémantique

Notre généralisation du travail de Boudol09 à Llama repose sur l'implémentation de la sémantique de Boudol09 et en particulier de son opération de verrouillage avec préverrouillage, élément relativement indépendant du système de types et d'effets.

On a vu en Section 1.2.5.3 que cette opération peut se ramener à l'acquisition simultanée de tous les mutex à verrouiller et à préverrouiller, suivie du déverrouillage des mutex qui ne devaient être que préverrouillés.

Nous avons donc cherché un algorithme de verrouillage multiple, autrement dit une fonction `lock_n` prenant en paramètre une liste de mutex et les acquérant ; la construction `lock-in` se compile alors en utilisant la fonction `lock_n`. Le verrouillage multiple peut alors facilement être proposé au programmeur :

```
(lock [m; n] in ...)
```

Ce problème s'est révélé non trivial. Voici en effet deux algorithmes naïfs qui peuvent engendrer des situations de famine.

2.4.1 Algorithme naïf A

Une première idée d'algorithme consiste à acquérir de manière traditionnelle les différents mutex les uns après les autres.

Acquérir les mutex dans l'ordre dans lequel ils se présentent :

```
let lock_n (l : mutex list) =
  List.iter Mutex.lock l
```

mènerait à des interblocages dans le cas archétypal :

```
(lock [m; n] in ...)  ||  (lock [n; m] in ...)
```

Pour prévenir les interblocages, on peut utiliser la méthode classique (voir Section 1.2.3) de supposer un ordre total sur les mutex et de toujours les acquérir dans cet ordre, comme ceci :

```
let lock_n (l : mutex list) =
  List.iter Mutex.lock (List.sort l)
```

Mais cet algorithme est asymétrique vis-à-vis des mutex et peut donc être inéquitable vis-à-vis de threads qui utilisent différents mutex. Ainsi dans l'exemple suivant, en supposant que les mutex `m1`, `m2` et `m3` ont pour ordre $\texttt{m1} < \texttt{m2} < \texttt{m3}$:

```
(* t1 *)              (* t2 *)                    (* t3 *)
while true do   ||   while true do            ||   while true do
  ...                  ...                           ...
  lock m1 in           lock [m3; m1; m2] in          lock m3 in
    ...                  ...                           ...
done                 done                          done
```

le thread `t2` acquerra toujours les mutex dans l'ordre `m1`, `m2`, `m3`. Le mutex `m1` sera donc pris pendant tout le temps que `t2` pourra mettre à acquérir `m3`, et sera donc plus sollicité que `m3`. Le thread `t1` sera donc désavantagé par rapport à `t3`, quand bien même ces deux threads exécuteraient un même algorithme, différant seulement par l'objet mutex utilisé.

2.4.2 Algorithme naïf B

Une autre idée d'algorithme pour acquérir n mutex, symétrique vis-à-vis des mutex à acquérir, peut être décrite ainsi : Si les n mutex sont libres, les prendre, sinon, attendre un moment où ils seront libres.

En ignorant les questions d'atomicité, cela peut s'écrire comme suit :

```
let lock_n (l : mutex list) =
  while not (List.for_all Mutex.is_free l) do
    wait ()
  done;
  List.iter Mutex.take l
```

Le défaut de cet algorithme est qu'il favorise beaucoup les threads prenant un ou peu de mutex par rapport aux threads en prenant un plus grand nombre.

Supposons en effet un programme avec n mutex, verrouillés par n threads de façon répétitive et non-coordonnée, alors qu'un thread supplémentaire essaye d'acquérir les n mutex en même temps :

```
(thread  (* t1 *)
   while true do
     lock m1 in
         ...
   done);
...
(thread  (* tn *)
   while true do
     lock mn in
         ...
   done);
(thread  (* t_all *)
   while true do
     lock [m1; ...; mn] in
         ...
   done)
```

Il se peut très bien que le moment où `m1 .. mn` sont libres et où t_{all} peut donc progresser ne vienne jamais, où seulement après un temps très long : t_{all} est alors en situation de famine.

2.5 Bilan

Nous avons ébauché un langage de programmation combinant une inférence d'effets qui permet un évitement des interblocages à la façon de Boudol09 et plusieurs fonctionnalités d'un langage généraliste de type Caml, dont les

modules, les déclarations de types, les types polymorphes, les types récursifs et les exceptions.

Les suites possibles de ce travail incluaient l'étude de la généralisation des effets latents et de l'ajout au langage de paramètres de région, ainsi que de l'unification des régions top-level et du problème de l'explosion du nombre de paramètres de région.

Cependant, une question particulière, indépendante du système de types, a retenu notre attention : celle de l'implémentation de la sémantique de Boudol09 par une opération de verrouillage multiple. Nous avons constaté avec surprise la faiblesse de la littérature sur ce sujet relativement simple à exprimer, et avons choisi de rechercher un algorithme d'acquisition de n mutex sans interblocage ni famine. Ce projet nous a poussé à changer d'environnement technique, Llama n'implémentant pas de parallélisme physique, ce qui aurait limité la portée de mesures de performances. La solution que nous avons développée est exposée en Section 3.4.

D'autre part, nous avons identifié au cours de notre travail le problème d'un partage parfois imprévisible et/ou contraignant des régions entre les valeurs. Ce problème, plus que les autres, nous semble intrinsèque à l'approche d'analyse statique particulière que nous avons choisie. En effet, en utilisant l'analyse de types comme mécanisme d'association des régions aux valeurs, on soumet cette association à des unifications qui peuvent sembler arbitraires, et à la réduction de parallélisme qui en découle, ce qui laisse pronostiquer des difficultés d'utilisation pour implémenter des algorithmes concurrents complexes.

Pour cette raison, nous avons choisi de changer d'approche et d'étudier l'utilisabilité d'un langage offrant un évitement des interblocages à la Boudol09 sans inférence d'effets, comme nous allons le voir dans le chapitre suivant. Cette approche ne nécessitant pas d'analyse statique, nous pouvions travailler dans un langage n'étant pas de la famille ML ; nous avons choisi le langage à typage dynamique Hop, développé dans notre équipe.

Chapitre 3

Empêcher interblocages et famine dans un langage à typage dynamique

Contents

3.1 Introduction

Dans le Chapitre 2, dans le cadre de l'implémentation de la sémantique du langage cible de Boudol09, nous avons soulevé la question de la manière d'acquérir un nombre arbitraire de mutex simultanément, et sans famine. Dans ce chapitre nous proposons un algorithme qui remplit ces critères. Pour cela nous avons choisi une définition précise de la notion de famine, et dans ce cadre nous avons identifié un type de situations problématiques distinct que nous appelons *interblocage asymptotique*.

Notre algorithme peut être vu comme une implémentation du langage cible de Boudol09, qui pourrait utiliser en front-end une inférence de types et d'effets semblable à celle de Boudol09. Cependant, nous avons choisi d'étudier l'utilisabilité de ce mécanisme sans analyse statique, en exposant directement au programmeur la construction de verrouillage avec préverrouillage.

Sans inférence, la résolution des ensembles de préverrouillage, nécessaires à l'évitement des interblocages, doit être effectuée par le programmeur. Cependant, nous avons apporté au langage des modifications de façon à le rendre utilisable en tant que tel. Nous verrons en particulier que avons combiné le préverrouillage de Boudol09 avec une variante de la méthode classique de prévention des interblocage qui consiste à ordonner les mutex (Section 1.2.3).

Nous avons choisi d'implémenter notre travail dans le langage Hop (voir plus loin), et d'utiliser celui-ci comme point de départ à partir duquel concevoir nos algorithmes. Un avantage notable de Hop par rapport à Llama est qu'il offre du parallélisme physique qui permet d'exploiter les architectures multiprocesseurs et multicœurs très répandues dans les ordinateurs personnels, contrairement à la bibliothèque de threads de Llama et OCaml, implémentée par du temps partagé sur un seul processeur.

Notre implémentation prend la forme d'une bibliothèque de threads et mutex, nommée `jthread`, pour le langage Hop. Nous verrons (dans le Chapitre 4) que nous avons pu très facilement convertir deux vrais programmes (dont un de 20 000 lignes de code) d'une autre bibliothèque de mutex (`pthread`) vers la nôtre.

Dans ce chapitre on présente donc :

— en Section 3.2, notre bibliothèque `jthread`, son interface et la façon dont elle empêche les interblocages ;
— en Section 3.3, le concept d'interblocage asymptotique et notre définition de la notion de famine ;
— en Section 3.4, la solution que nous proposons au problème du verrouillage multiple ;
— en Section 3.5, une sémantique formelle faisant une synthèse des éléments présentés dans ce chapitre.

Dans le Chapitre 4 on présentera les aspects relevant de l'implémentation proprement dite de notre bibliothèque et de ses performances.

Les résultats clés de ce travail ont été publiés dans [26].

3.1.1 Hop

Hop [5, 43, 13], développé principalement par Manuel Serrano, est à la fois un langage de programmation Web multi-tier et un serveur Web. Pour notre usage, Hop est un dialecte de Scheme, c'est-à-dire un langage de programmation fonctionnel à typage dynamique. Plus précisément, nous avons travaillé avec Bigloo [1], le compilateur en code natif de Hop, qu'on peut voir comme la partie non-Web de Hop et qui est utilisable comme un langage – généraliste – à part entière.

Hop dispose d'une interface de programmation commune à plusieurs bibliothèques qui offrent threads, mutex et variables de conditions. Cette interface, proche de la norme SRFI 18 [1] [8], offre notamment une opération de verrouillage structuré :

```
(synchronize m
   ... )    ; corps exécuté en ayant verrouillé le mutex m
```

ainsi que deux fonctions de verrouillage explicite :

```
(mutex-lock! m)
(mutex-unlock! m)
```

Hop offre 2 implémentation de cette interface : `fthread`, une bibliothèque de threads coopératifs, et `pthread`, une bibliothèque de threads préemptifs qui, dans le cas de la compilation de Hop vers C, est implémentée par des appels à la bibliothèque C standard du même nom.

Nous avons choisi de fournir une troisième implémentation de l'interface, que nous appelons `jthread`. Notre implémentation repose quant à elle sur la bibliothèque `pthread`.

Dans la suite, nos exemples utiliseront les fonctions de Hop ainsi que quelques construction classiques telles que `while` et `thread` qui ne font pas partie de

1. Scheme Requests for Implementation, série de normes de bibliothèques et d'extensions de Scheme

Hop mais que l'on peut définir simplement. À titre d'exemple, voici notre implémentation de la construction `thread`, qui lance l'exécution de son corps dans un nouveau thread :

```
(define-macro (thread . exprs)
   `(thread-start-joinable!
       (instantiate::pthread  ; ou jthread
          (body (lambda () ,@exprs)) )))
```

3.2 Une bibliothèque de mutex sans interblocage

3.2.1 Vue d'ensemble

Dans cette section on décrit une bibliothèque de mutex proposant du verrouillage structuré sans interblocage. La construction principale est la construction de section critique suivante :

```
(synchronize l [:prelock p] body)
```

où `l` est la liste des mutex à verrouiller, et le paramètre optionnel `p` une autre liste de mutex dont l'utilité sera expliquée plus loin. Le corps `body` consiste en une ou plusieurs expressions. On notera les listes contenant un seul mutex « `m` » au lieu de « `(list m)` ».

L'implémentation de cette construction repose sur des primitives de verrouillage explicite, `lock-n` et `unlock-n`, qui permettent respectivement d'acquérir et libérer n mutex de façon atomique. L'implémentation de ces primitives et des mutex associés est décrite dans la Section 3.4.

Nous verrons que l'implémentation de l'opération `lock-n` repose sur un mécanisme d'exclusion mutuelle classique tel que les mutex *pthread* disponibles en Hop. Plus précisément, notre implémentation crée un mutex pthread global `big-lock`, et tous les algorithmes décrits ci-après sont exécutés alors que le thread courant possède `big-lock` (à l'exception des phases d'attente).

Notre construction est compilée en un appel à une fonction du même nom dont voici une esquisse de l'implémentation, qui sera expliquée par la suite :

```
(define (synchronize l p thunk)
   (pthread-synchronize big-lock
      [faire respecter les règles d'évitement des interblocages]
      (lock-n l p)
      (unwind-protect
         ; try
         (thunk)
         ; finally
         (unlock-n l) )))
```

Nous allons voir que notre approche combine deux techniques pour empêcher les interblocages :

— éviter les interblocage par le préverrouillage, comme dans Boudol09 ;
— vérifier, lors de l'exécution, que le programmeur a respecté certaines règles qui préviennent les interblocages, et lancer des erreurs sinon ; une approche qu'on peut rapprocher du typage dynamique.

Le lecteur pourra se référer à la sémantique formelle donnée en Section 3.5, qui synthétise la description de notre travail faite dans ce chapitre.

3.2.2 Autorisations

La fonction que nous venons de présenter empêche tout interblocage, par des moyens que nous allons expliquer maintenant. La principale particularité de notre approche est qu'à un moment donné de l'exécution d'un thread t, t peut ne pas être autorisé à acquérir certains mutex, c'est-à-dire autorisé à appeler la fonction blocante `lock-n` sur ces mutex.

Essayer de prendre avec `synchronize` un mutex qui n'est pas autorisé génère une erreur à l'exécution. Si les autorisations sont respectées dans un programmme, son exécution est sans erreur et sans interblocage.

Les autorisations sont contrôlées par le programmeur : à tout point d'un programme, tout mutex peut être autorisé si le programmeur le décide. Cependant, l'autorisation de certains mutex induit un évitement « actif » des interblocages, c'est-à-dire de la réduction du parallélisme. Par défaut, à un point donné du programme, les seuls mutex autorisés sont ceux qui n'activent pas l'évitement des interblocages.

Cas particulier : Quand un thread ne possède aucun mutex, il peut acquérir n'importe quel mutex. Seuls les verrouillages imbriqués sont soumis à autorisation.

Nos fonctions combinent 2 mécanismes d'autorisations complémentaires : les **régions** et le **préverrouillage**. Le préverrouillage, en tant que mécanisme d'autorisation, est simplement la condition à laquelle l'évitement des interblocages par préverrouillage est mis en œuvre. Le mécanisme des régions, quant à lui, est une instance de la technique classique qui consiste à ordonner les mutex (voir Section 1.2.3).

Ces deux mécanismes sont décrits dans les deux sous-sections suivantes. Le schéma de leur fonctionnement combiné est le suivant : Les régions sont des ensembles de mutex. Chaque mutex appartient à une région, la même pour toute sa durée de vie. Il y a sur les régions un ordre partiel qui interdit tout cycle dans le verrouillage de mutex qui appartiennent à différentes régions. Dans chaque région, les mutex sont considérés comme égaux et peuvent être verrouillés de façon imbriquée grâce au préverrouillage.

3.2.3 Régions

Notons avant tout que les régions dont il est question maintenant ne correspondent pas aux régions de Tesard, ni ne doivent être confondues avec les *régions critiques*,

autre appellation des *sections critiques*.

Il y a un ordre partiel dynamique sur les régions tel que pour toutes régions R_1 et R_2 d'un programme, si à un moment donné de l'exécution du programme $R_1 > R_2$, cela reste vrai pour toute la suite de l'exécution.

Règle 1 : Un thread peut prendre un mutex `n` (dans la région R_n) alors qu'il possède un mutex `m` (dans la région R_m) seulement si $R_m \geq R_n$.

```
(synchronize m
   (synchronize n  ; seulement si Rm >= Rn
      ... ))
```

Cette règle garantit l'absence d'interblocages impliquant des mutex qui appartiennent à au moins deux régions différentes (voir ci-après).

Plus en détails :

Le programmeur peut créer des régions, et choisir la région de chaque mutex au moment de sa création, à l'aide des fonctions suivantes :

```
(make-region)

(make-mutex [region])
```

La région par défaut d'un nouveau mutex `m` est une nouvelle région qui ne contient initialement que `m`. Ce choix nous semble correspondre au souhait du programmeur dans une majorité de cas.

L'ordre partiel sur les régions est construit au fil des appels à `synchronize` comme l'ordre partiel minimal pour satisfaire la Règle 1. Par exemple :

```
(define m (make-mutex))  ; nouvelle région Rm
(define n (make-mutex))  ; nouvelle région Rn
(synchronize m
   (synchronize n  ; pas encore de relation entre Rm et Rn
                   ; on pose Rm > Rn
      ... ))
(synchronize n
   (synchronize m  ; une relation existe : Rm > Rn
                   ; -> erreur
      ... ))
```

On ne considère pour l'instant que des blocs `synchronize` où tous les mutex de `l` et `p` appartiennent à la même région. On définira plus tard une extension naturelle de la construction `synchronize` pour plusieurs régions.

Décrivons maintenant comment cet ordre partiel minimal est calculé. Voici une description de l'algorithme indépendante de la représentation de l'ordre partiel :

On associe à chaque thread une *région courante*. Quand le thread est lancé, sa région courante est initialisée à Top, une région plus grande que toutes les autres. Puis, dans chaque section critique (y compris imbriquées), le thread prend pour région courante la région des mutex acquis (y compris pendant l'éventuelle attente pour l'acquisition de ces mutex). Chaque thread a donc une pile de régions, dont la première est la région courante, et sur laquelle les regions sont empilées ou dépilées à chaque fois que le thread entre ou sort d'une section critique.

À chaque `synchronize`, si le thread courant a pour région courante R_m et que les mutex acquis ont pour région courante $R_n \neq R_m$, on effectue ce qui suit : si l'ordre partiel contient déjà $R_n > R_m$ alors le changement de région (et donc toute l'opération) est interdit. Sinon, le changement de région est autorisé et la relation $R_m > R_n$ est ajoutée à l'ordre partiel si elle n'y est pas déjà.

La Règle 1 est donc équivalente à la Règle 1 bis :

Règle 1 bis : Un thread dans une région R_m ne peut prendre un mutex `n` (dans une région R_n) que si $R_m \geq R_n$.

Si $R_m > R_n$ cela suffit à conclure que le thread est autorisé à acquérir `n`. Si au contraire $R_m = R_n$, alors un mécanisme d'autorisations complémentaire s'applique.

3.2.3.1 Absence d'interblocages

La Règle 1 garantit l'absence d'interblocages impliquant des mutex qui appartiennent à au moins deux régions différentes.

En effet, rappelons qu'on appelle interblocage une situation dans laquelle $\exists\, t.\ t \prec^+ t$. Considérons donc, à un instant donné de l'exécution d'un programme, une chaîne de threads $t_0 \prec ... \prec t_n$, et prouvons que si elle implique des mutex appartenant à des régions différentes alors $t_0 \neq t_n$.

Par définition de $\prec$, $\forall\ 1 \leqslant i \leqslant n.\ \exists\ m_i.\ t_{i-1}$ possède $m_i\ \wedge\ t_i$ attend m_i.

$\forall\ 1 \leqslant i \leqslant n$, appelons R_i la région de m_i, qui est aussi la région courante de t_i. Soit R_0 la région courante de t_0.

Alors $\forall\ 1 \leqslant i < n.\ t_i$ attend $m_i\ \wedge\ t_i$ possède m_{i+1} donc, d'après la Règle 1, $R_i \leq R_{i+1}$.

D'autre part, puisque t_0 possède m_1, d'après la Règle 1 bis, $R_0 \leq R_1$. Donc $\forall\ 0 \leqslant i < n.\ R_i \leq R_{i+1}$.

Alors, si la chaîne implique au moins deux régions, au moins une des inégalités est stricte, donc $R_0 \neq R_n$, donc $t_0 \neq t_n$. □

3.2.3.2 Modularité

Par rapport au seul préverrouillage, ce mécanisme de régions facilite sensiblement la compilation séparée (par modules).

En effet, dans le cas du préverrouillage, pour la synchronisation imbriquée de mutex `m` et `n`, il est nécessaire que `n` soit connu (pour être préverrouillé) au moment de la synchronisation de `m`. Ça signifie que si `n` est synchronisé dans une fonction `f` d'un module différent, ce module doit exporter `n`, ainsi que l'information que `f` verrouille `n`.

Dans Tesard (Chapitre 2) ce problème était résolu par l'inférence de types et d'effets qui permettait de générer des interfaces de modules contenant types et régions. Dans Hop, en revanche, cette méthode nous semble a priori plus difficile à mettre en œuvre. Avec le préverrouillage seul, le programmeur aurait donc la tâche pénible d'exporter, de chaque module M, tous les mutex créés dans M, y compris dynamiquement, pour qu'ils puissent être préverrouillés dans les modules qui appellent des fonctions de M.

Les régions résolvent ce problème. En effet, il est assez naturel de prendre pour convention que les mutex d'un module M seront dans une ou des régions inférieures à celles des mutex d'un module N qui utilise M. Les verrouillages avec préverrouillage ne touchent alors que des mutex créés dans un même module, qui sont donc facilement accessible du point de vue syntaxique.

3.2.4 Préverrouillage

Pour le verrouillage imbriqué de mutex qui appartiennent à une même région, on utilise le mécanisme d'évitement des interblocages de Boudol09 à base de préverrouillage (voir Section 1.2.5).

Dans le langage de Boudol09, le programmeur n'écrit que les synchronisations nécessaires à son programme, et une analyse statique infère les préverrouillages nécessaires. Dans notre bibliothèque il n'y a pas d'inférence ; c'est donc au programmeur d'écrire les préverrouillages nécessaires, avec la contrainte suivante :

Règle 2 : Si des mutex `m` et `n` appartiennent à une même région, un thread t ne peut acquérir `n` alors qu'il possède `m` que si en verrouillant `m` il a préverrouillé `n`.

```
;; m, n et p appartiennent à la même région
(synchronize m :prelock n
   ...
   (synchronize n  ; autorisé
      ... )
   ...
   (synchronize p  ; interdit
      ... )
   ... )
```

Comme dans Boudol09, lorsqu'il y a plusieurs niveaux d'imbrication, les mutex des sections critiques intérieures doivent être préverrouillés à tous les niveaux supérieurs :

```
(synchronize m :prelock (list n p)
   (synchronize n :prelock p
      (synchronize p
         ... ))
```

En d'autres termes, les mutex qu'il est interdit de verrouiller ne peuvent pas non plus être préverrouillés :

```
(synchronize m :prelock n
   (synchronize n :prelock p  ; erreur : p non autorisé
      ... ))
```

On peut considérer que cela consiste à exposer le langage cible de Boudol09 au programmeur et à y ajouter des vérifications à l'exécution que les règles sont respectées.

La méthode utilisée pour faire ces vérifications est semblable à celle exposée précédemment pour mettre en œuvre la Règle 1 sur la hierarchie des régions.

Chaque thread a un ensemble variable de mutex explicitement autorisés. Cet ensemble est modifié et restauré respectivement au début et à la fin de chaque block `synchronize`. Chaque fois qu'un thread entre dans une nouvelle région R, cet ensemble est initialisé pour valoir l'ensemble des mutex qui appartiennent à R. Par la suite, l'ensemble de mutex explicitement autorisés est l'ensemble de mutex à préverrouiller de l'actuel bloc `synchronize` le plus intérieur.

La Règle 2 est équivalente à la Règle 2 bis :

Règle 2 bis : Un thread t ne peut verrouiller et préverrouiller des mutex qui appartiennent à sa région courante que s'ils étaient membres de l'ensemble de threads à préverrouiller du bloc `synchronize` le plus intérieur dans lequel se trouve t.

Comme dans Boudol09, cela garantit l'absence d'interblocages entre mutex qui appartiennent à une même région.

3.2.4.1 Mutex nouvellement créés

Dans le langage de Boudol09, chaque nouveau mutex, lors de sa création, est automatiquement autorisé pour sa portée lexicale dans le thread qui le crée. Cette règle est mise en œuvre par le système de types et d'effets. Voir aussi la règle d'introduction du let region en Section 2.2.1.

Dans notre bibliothèque, si un mutex `m` est créé dans une région différente de la région courante du thread qui le crée, la question ne se pose pas. En revanche, si les deux régions sont les mêmes, `m` n'est pas autorisé par la Règle 2.

C'est pourquoi on crée une troisième règle faisant exception à la Règle 2, exception en fait plus forte que dans Boudol09 :

Règle 3 Chaque mutex, lors de sa création, est automatiquement autorisé dans tous les blocs `synchronize` en cours d'exécution dans tous les threads qui se trouvent dans la même région.

```
(define R (make-region))
(define m (make-mutex R))
(define n (make-mutex R))
(define x #f)
(synchronize m :prelock n
   (synchronize n
      (set! x (make-mutex R))
      ; x est autorisé
      ... )
   ; x est autorisé
   (synchronize n
      ; x n'est pas autorisé
      ... )
   ; x est autorisé
   ... )
```

Correction Autoriser des mutex nouvellement créés ne génère aucun interblocage. En effet, du point de vue des autorisations, un nouveau mutex est équivalent à un mutex qui aurait été créé au début du programme et correctement préverrouillé par tous les threads, mais jamais verrouillé, ce qui correspond à une situation sûre.

Implémentation La Règle 3 est mise en œuvre en donnant à chaque mutex un identificateur numérique croissant lors de sa création. Alors, chaque thread entrant dans une section critique mémorise le dernier identificateur (global) attribué. Un thread t peut alors déterminer si un mutex donné a été créé avant ou après que t soit entré dans sa section critique courante.

Par exemple, si la variable globale `last-id` contient le dernier identificateur attribué à un mutex :

```
; last-id = 0
(define R (make-region))
(define m (make-mutex R))      ; reçoit id = 1
; last-id = 1
(synchronize m                 ; mémorise last-id = 1
   (let ((n (make-mutex R)))   ; reçoit id = 2
      (synchronize n
         ... )))
```

Au niveau du second `synchronize`, `n` est dans la ême région que `m` et n'a pas été explicitement autorisé, mais l'identificateur de `n` est plus grand que `last-id` au moment du `synchronize m`, donc `n` est autorisé.

3.2.4.2 Association mutex-valeur à grain fin

Le fait que l'association entre mutex et valeurs soit faite manuellement par le programmeur permet qu'elle soit plus souple que dans Tesard. Par exemple, rien n'empêche des valeurs contenues dans une table de hachage soit chacune associée à un mutex différent.

3.2.5 Verrouillage récursif

Comme dans Boudol09, nos mutex supportent le verrouillage récursif (voir Section 1.1.4.3). Le verrouillage récursif d'un mutex ne crée aucune relation de blocage, et ne peut pas contribuer à créer d'interblocage. Cette opération pourrait donc être autorisée systématiquement, sans tenir compte des trois Règles énoncées précédemment.

Néanmoins, il nous a semblé pertinent de présenter au programmeur un comportement plus régulier, quoique légèrement moins permissif, en imposant que le verrouillage récursif d'un mutex respecte lesdites Règles. Autrement dit, lors de l'exécution par un thread t de `(synchronize l :prelock p)` alors qu'il possède déjà certains mutex de `l` et `p` :

- La région des mutex de `l` et `p` doit être la région courante de t (sinon les deux régions feraient partie d'un cycle).
- Appelons `l1` et `p1` les listes de mutex de respectivement `l` et `p` que t ne possède pas déjà. Les mutex de `l1` et `p1` doivent être autorisés par les Règles 2 et 3.
- Les mutex de `l1` sont verrouillés et ceux de `p1` préverrouillés.

Le principal intérêt de ce comportement est le suivant : dans le cas où `l1` est vide, l'opération de préverrouillage n'a pas lieu d'être, mais l'ensemble de mutex explicitement autorisés de t prend tout de même pour valeur `p1`, pour ne pas que soient explicitement autorisés dans le corps du `synchronize` des mutex qui n'appartiennent pas à `p`.

3.2.6 `synchronize` impliquant plusieurs régions

Nous avons déjà défini `synchronize` sur plusieurs mutex, mais nous ne l'autorisions que pour des mutex appartenant à une même région. Voici maintenant une règle autorisant certains appels à `synchronize` impliquant des mutex appartenant à plusieurs régions.

Un `synchronize` impliquant plusieurs régions est autorisé ssi sa décomposition en plusieurs `synchronize` imbriqués, un par région, est autorisé. Par exemple, si les mutex `m1`, `m2`, `n1`, `p1` et `p2` appartiennent aux régions R_m, R_n et R_p comme suggéré par leur nom, alors

```
(synchronize (list m1 m2 n1 p1) :prelock (list p2)
   ...)
```

est autorisé ssi

```
(synchronize (list m1 m2)
   (synchronize (list n1)
      (synchronize (list p1) :prelock (list p2)
         ... )))
```

est autorisé.

Les deux formes sont sémantiquement équivalentes puisqu'elles consistent à exécuter leur corps en ayant verrouillé et préverrouillé les mêmes mutex. Cependant, la première s'exécute en un seul appel à la fonction `lock-n` au lieu de trois pour la seconde.

Un `synchronize` impliquant plusieurs régions impose la même contrainte sur leur ordre que sa forme décomposée. Ainsi, l'exemple ci-dessus implique que $R_m \geq R_n \geq R_p$.

Le thread courant prend alors pour région courante la dernière des régions concernées (celle du `synchronize` le plus intérieur dans la décomposition), qui doit être la région de tous les mutex du `prelock` s'il y en a un. Préverrouiller des mutex appartenant à plusieurs régions n'aurait pas de sens, dans la mesure où seule la région courante d'un thread peut ne pas être complètement autorisée ou complètement interdite pour ce thread. Dans l'exemple ci-dessus, à l'intérieur du bloc, la région courante du thread sera R_p et l'ensemble de mutex explicitement autorisés sera $\{p_2\}$.

3.2.7 Mode non-sûr

Dans les sections précédentes nous avons décrit des règles que le programmeur doit respecter et des algorithmes pour vérifier à l'exécution que ces règles sont respectées et produire des erreurs si ce n'est pas le cas. On peut considérer que ces vérifications sont un outil pour déboguer un programme et qu'elles sont inutiles une fois que le programme a été testé et débogué. (Les vérifications en question ne doivent pas être confondues avec l'*utilisation* des ensembles de préverrouillage qui *sont* nécessaires à l'exécution pour éviter les interblocages.)

L'impact de ces vérifications sur les performances est donc d'importance modérée puisque les programmes qui doivent avoir de bonnes performances pourront être exécutés dans un mode « non-sûr » dans lequel ces vérifications sont désactivées.

Cependant lorsque nous avons mesuré les performances de notre implémentation, le coût de nos fonctions était dominé pas le coût des fonctions de plus bas niveau (`lock-n` etc.) ; nous n'avons pas observé de différence significative entre les modes « sûr » et « non-sûr » de notre bibliothèque.

3.2.8 Variables de conditions

On a rappelé en Section 1.1.5 le principe du mécanisme de synchronisation que sont les variables de conditions. Les variables de conditions sont en particulier offertes par la bibliothèque pthread (en C et Hop) et fréquemment utilisées conjointement avec les mutex. Nous avons donc souhaité des variables de conditions fonctionnant avec nos mutex.

Or il est possible d'utiliser directement les variables de conditions pthread (structures de données et fonctions) avec les mutex jthread, à la seule condition de redéfinir la fonction `wait`, comme un wrapper autour de la version pthread de cette fonction.

En effet, il suffit que la nouvelle fonction `wait`, prenant une variable de condition pthread `cv` et un mutex jthread `m`, relâche `m` et attende sur `cv` avec un mutex pthread (ces deux opérations de façon atomique), puis reprenne `m`.

L'atomicité nécessaire ainsi que le déverrouillage et la réacquisition de `m` nécessitent la protection du verrou global `big-lock` (Section 3.2.1) ; c'est donc ce mutex pthread qui est utilisé pour tous les appels à la fonction `wait` pthread.

Le code, aux détails d'implémentation près, est le suivant. Il fait appel aux fonctions de plus bas niveau `lock-n` et `unlock-n` qui seront décrites dans la Section 3.4.

```
(define (j-condition-variable-wait! cv::pcondvar m::jmutex)
   (p-synchronize big-lock
      (unlock-n m)
      (p-condition-variable-wait! cv big-lock)
      (lock-n m) ))
```

Notons que si notre bibliothèque supporte les variables de conditions, elle ne prévient pas les blocages définitifs accidentels (inter-blocages ou autres) qui les mettent en jeu : un thread peut se mettre en attente sur une variable de condition qui ne sera jamais signalée.

3.3 Famine et interblocage asymptotique

Notre projet d'une implémentation sans famine du verrouillage multiple pose la question de la définition de la notion de famine. La définition de cette notion ne fait pas consensus, notamment dans le cas d'un langage de programmation riche tel que Hop. En particulier, de nombreuses définitions, telles que [50], ne considèrent pas la création dynamique d'un nombre arbitrairement grand de threads.

Pour spécifier ce que signifie le fait que notre bibliothèque soit sans famine, nous avons été amenés à introduire le concept d'*interblocage asymptotique*. Nous préciserons également la définition de progression à l'infini.

3.3.1 Interblocage asymptotique

3.3.1.1 Définition

Au cours d'une exécution d'un programme qui ne termine pas, un thread est en situation d'**interblocage asymptotique** si le nombre de ses ascendances [2] tend vers l'infini.

2. les threads qui le bloquent (voir Section 1.2)

Un interblocage asymptotique implique que les nombres de threads et de mutex tendent eux aussi vers l'infini.

3.3.1.2 Exemple

Pour introduire le concept d'interblocage asymptotique, voici l'exemple d'une exécution problématique d'un programme, que nous avons hésité à considérer comme une situation de famine avant de choisir d'utiliser un concept à part.

Le programme est le suivant :

```
(define (m0 (make-mutex)))
(define (P m)
   (thread
      (synchronize m
         (let ((n (make-mutex)))
            (P n)
            ...  ; prend du temps
            (synchronize n
               #f )))))
(P m0)
```

Ce programme crée une infinité de threads et autant de mutex. Chaque thread crée un mutex et lance un nouveau thread auquel il passe ledit mutex. Chaque thread verrouille du début à la fin de son exécution le mutex qu'il a reçu de son parent, et tente d'acquérir le mutex qu'il a lui-même créé.

Supposons que dans chaque thread, il se passe suffisamment de temps entre le lancement du thread enfant et l'exécution du `synchronize n` pour que ce `synchronize` ne soit exécuté qu'après que l'enfant ait pu verrouiller le mutex `n`. Le thread parent, lui, ne peut donc l'obtenir. Alors aucun mutex n'est jamais libéré et/car aucun thread ne termine.

Notons d'abord qu'il n'y a pas d'*interblocage* au sens où nous l'avons défini, c'est-à-dire de cycle de blocage entre les threads : chaque thread est bloqué par son enfant, et ainsi de suite. Cette imbrication de verrouillages est *autorisée* avec nos fonctions (voir section précédente), que les mutex soient dans des régions différentes (pas de cycle entre les régions) ou dans la même (Règle 3).

Cet exemple est semblable à une situation de famine. En effet, d'une part aucun thread n'obtient le second mutex qu'il attend, et d'autre part il existe un ordonnancement qui permet à chaque thread d'obtenir satisfaction et de terminer : il s'agit de l'ordonnancement « coopératif » dans lequel un seul thread s'exécute à la fois, et où un thread qui s'exécute n'est jamais interrompu préemptivement mais s'exécute jusqu'à terminer ou bloquer. Avec cet ordonnancement les enfants ne commencent pas à s'exécuter avant que leur parent ait terminé, donc chaque thread obtient son second mutex ; les threads s'exécutent simplement l'un après l'autre dans leur ordre de création.

Cependant nous considérons qu'il y a une différence significative entre l'exécution décrite plus haut, dans laquelle chaque thread est bloqué par un autre qui ne

relâche pas le mutex pour lequel ils sont en compétition, et une situation de famine classique dans laquelle un thread n'obtient jamais satisfaction alors que les ressources dont il a besoin sont libres ou périodiquement libres.

Dans notre exemple, pour chaque paire de threads parent/enfant, le programmeur a décidé que si l'enfant le relâchait pas le mutex créé par le parent, ce dernier se retrouvait bloqué. Dans une situation de famine classique, au contraire, le programmeur n'a que peu de moyens de prévoir qu'un thread ne progressera pas alors qu'il y est formellement autorisé. Notre exemple est en cela plus proche d'un interblocage, bien qu'il n'en soit pas un.

3.3.1.3 Prédiction

Dans un langage Turing-complet avec mutex, et sans discipline de verrouillage particulière, la question de savoir, étant donné un programme, si son exécution peut contenir un interblocage asymptotique, est indécidable ; on peut en effet y ramener le problème de l'arrêt :

Soit un programme P donné. Si on l'instrumente de façon à produire une chaîne de blocages comme dans l'exemple vu plus haut, chaque pas de l'exécution de P ajoutant un thread, et qu'on appelle π le programme instrumenté, alors P termine si et seulement si π ne génère aucun interblocage asymptotique.

3.3.1.4 Détection

Voici une façon rudimentaire d'aider un programmeur à se débarasser d'interblocages asymptotiques dans un programme : Il est possible d'instrumenter l'opération d'acquisition de mutex de façon à ce que chaque opération calcule le nombre d'ascendances du thread courant et avertissent le programmeur (par un *warning*) si le nombre d'ascendances d'un thread dépasse un certain seuil défini à l'avance.

3.3.2 Progression à l'infini

Clarifions maintenant ce que signifie le fait qu'un thread progresse indéfiniment.

Au cours de l'exécution d'un programme, on considère que les possibilités d'évolution d'un thread t peuvent se diviser en 3 cas :

- t termine, c'est-à-dire finit par terminer ;
- t finit par être bloqué définitivement (ne plus jamais progresser) ;
- t *progresse indéfiniment*, c'est-à-dire soit finit par ne plus jamais être bloqué, soit obtient satisfaction une infinité de fois. Cela revient à dire que la durée de progression de t (dans une certaine échelle de temps) est infinie, en supposant impossible que la durée des périodes où t progresse tende vers 0 et que leur somme infinie converge.

Pour une propriété P donnée, on dira que « si t progresse il finit par vérifier P » ssi. soit t finit par être bloqué définitivement, soit il finit par vérifier P.

3.3.3 Absence de famine

Ayant défini les notions d'interblocage asymptotique et de progression à l'infini, nous pouvons maintenant introduire celle de famine que nous utiliserons dans la section suivante.

Nous utiliserons la définition négative suivante de la famine : une exécution d'un programme est **sans famine** si :

Si chaque thread qui progresse finit par relâcher tous ses mutex **et si** il n'y a pas d'interblocage asymptotique, **alors** chaque thread finit par progresser.

Un programme est **sans famine** si toutes ses exécutions possibles sont sans famine. Dans la section suivante nous donnerons un algorithme de verrouillage multiple sans famine, autrement dit tel que tout programme qui l'utilise est sans famine.

3.4 Verrouillage multiple

3.4.1 Introduction

Dans cette section nous décrivons une structure de données de type mutex et une opération de **verrouillage multiple**, c'est-à-dire d'acquisition de n mutex garantissant l'absence de famine. Cette opération nous permet d'implémenter les blocs `synchronize` sans interblocage décrits en Section 3.2.

On a vu en Section 2.4 que la sémantique de Boudol09, bien qu'elle évite les interblocages, n'empêche pas a priori les situations de famine. Nous nous sommes donné pour but de concevoir une opération de verrouillage multiple (ce qui est suffisant pour implémenter la sémantique de Boudol09) qui n'entraîne pas de famines, et pour cela de définir un ordonnancement équitable des threads qui attendent pour prendre des mutex. (On parle d'ordonnancement de haut niveau sans préemption, a priori indépendant de l'ordonnancement préemptif de bas niveau des threads par le système d'exploitation [3].)

Contrairement aux blocs `synchronize` de la Section 3.2, dans cette section nos mutex offrent du verrouillage explicite, c'est-à-dire des fonctions de verrouillage et déverrouillage séparées `lock-n` et `unlock-n`. Cependant, notre preuve que notre d'algorithme de verrouillage est sans famine reposera, elle, sur le fait que les fonctions `lock-n` et `unlock-n` ne sont utilisées que pour implémenter les sections critiques imbriquées définies en Section 3.2, et jamais selon le motif

```
(lock-n a)
(lock-n b)
(unlock-n a)
(unlock-n b)
```

3. Sur les interactions entre l'implémentation de mécanismes de synchronisation de haut niveau et l'ordonnancement préemptif de bas niveau, voir par exemple [20].

Par ailleurs, notre fonction `lock-n` implémente le préverrouillage : elle tient compte du fait que certains mutex sont pris pour être relâchés immédiatement et optimise ce cas.

Nous verrons aussi que notre solution repose sur l'absence d'interblocages, et n'a tout simplement pas de sémantique définie en cas d'interblocage.

Rappelons que nos algorithmes supposent et utilisent un mécanisme d'exclusion mutuelle sous-jacent ; dans notre implémentation, la bibliothèque pthread.

Rappelons également que le lecteur pourra se référer à la sémantique formelle en Section 3.5 qui fait la synthèse de ce qu'on explique ici.

3.4.2 Fonctionnement général

À un instant donné de l'exécution d'un programme, considérons l'ensemble $\mathscr{A}$ des threads **actifs** , c'est-à-dire possédant des mutex ou attendant pour en prendre. L'idée principale de notre solution est de définir sur $\mathscr{A}$ un ordre total $<$ obéissant à l'invariant suivant [4] :

$$\text{(I)} \qquad \forall\ t_1,\ t_2.\ t_1 \prec^+ t_2\ \Rightarrow\ t_1 < t_2$$

Ceci n'est possible que si la relation $\prec^*$ est antisymétrique, autrement dit, s'il n'y a pas d'interblocage.

Cela signifie que les threads actifs sont dans une queue (file d'attente) globale. On utilisera les deux points de vue indifféremment. On écrira $Q = (\mathscr{A}, <)$. Cela n'implique pas que l'ordre total sur $\mathscr{A}$ soit implémenté au moyen d'une structure de donnée « queue » globale en pratique.

L'ordre sur les threads actifs est défini algorithmiquement : au début de l'exécution, Q est vide ; au cours de l'exécution, on modifie l'ordre à chaque fois qu'il est pertinent de le faire et en particulier lorsque Q gagne ou perd un élément.

- Quand un thread t devient actif, autrement dit quand il ne possède aucun mutex et se met à attendre pour en prendre, t est inséré à la fin de Q, c'est-à-dire qu'on l'établit comme plus grand que tous les threads précédemment actifs.
- Quand un thread t devient inactif, autrement dit quand il déverrouille le dernier mutex qu'il possédait, t est tout simplement retiré de Q.

Quand un thread en attente prend les mutex qu'il attendait, l'ordre n'est pas modifié, le thread garde sa place dans la queue.

L'ordre ainsi défini est utilisé de la façon suivante : un thread t qui attend des mutex peut les verrouiller si et seulement si aucun thread plus petit (devant t dans Q) ne les possède ou ne les attend.

Cette idée d'un ordre total sur les threads actifs est inspirée de l'algorithme de la boulangerie de Lamport (voir Section 1.1.2), bien que notre solution nécessite un mécanisme d'exclusion mutuelle sous-jacent. Contrairement à l'algorithme

4. qui est équivalent à $\forall\ t_1,\ t_2.\ t_1 \prec t_2 \Rightarrow t_1 < t_2$ (sans $\prec^+$)

de la boulangerie, notre solution autorise le verrouillage imbriqué ; l'ensemble des threads possédant des mutex et celui des threads en attente ne sont donc pas disjoints, c'est pourquoi on les traite comme un unique ensemble de threads *actifs*.

3.4.2.1 Notations sur les queues

Par la suite, on notera $Q_1 \cdot Q_2$ la concaténation de deux queues Q_1 et Q_2.

On décrira des *sous-queues* par des compréhensions de liste notées comme suit : pour $Q = (\mathscr{A}, <)$ et un prédicat p sur les threads :

$[\ t \text{ in } Q \mid p(t)\ ] = (\{t \in \mathscr{A} \mid p(t)\},\ <)$

3.4.3 Verrouillage imbriqué

Rappelons qu'on appelle verrouillage imbriqué le fait pour un thread d'acquérir des mutex alors qu'il en possède déjà d'autres. Ceci se produit lors de l'exécution de blocs `synchronize` imbriqués, mais nous ne prendrons pas en compte leur portée lexicale dans cette section.

Notre algorithme nécessite que l'on définisse l'évolution de Q quand un thread déjà actif t_X se met en attente de nouveaux mutex m_i. À cet instant, de nouvelles relations $\prec$ apparaissent entre t_X et les éventuels threads propriétaires des m_i.

On peut partitionner les threads de $\mathscr{A}$ en 4 sous-ensembles en fonction de leur relation de blocage à t_X :

- $\{t_X\}$
- $B_h = \{t \mid t_X \prec^+ t\}$: descendances de t_X, threads qui sont et doivent rester derrière t_X
- $B_f = \{t \mid t \prec^+ t_X\}$: ascendances de t_X, threads qui doivent être devant t_X
- $I = \{t \mid t \not\prec^* t_X \bigwedge t_X \not\prec^* t\}$: les threads indépendants de t_X

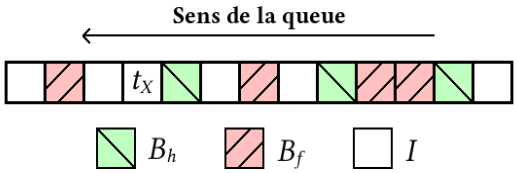


FIGURE 3.1 – Exemple de queue au moment où un thread t_X se met en attente de mutex, avant la nécessaire réorganisation de Q. Chaque case représente un thread. Les threads sont colorés en fonction de leur relation de blocage à t_X.

On prend pour acquis que $B_f \cap B_h = \emptyset$. En effet, un thread appartenant à la fois à B_f et B_h serait en situation d'interblocage, ce qui est empêché par les moyens décrits en Section 3.2. Les threads de B_h sont nécessairement déjà derrière t_X,

car le fait que t_X se mette en attente d'un ou plusieurs mutex ne rend aucun autre thread bloqué par t_X.

On appelle *stratégie* une spécification de l'évolution de Q lorsqu'un thread présent dans Q se met en attente de mutex.

Par exemple, une stratégie simple consiste à dire qu'un thread qui se met en attente se déplace à la fin de la queue, accompagné de ses éventuelles descendances :

$$Q \to [t \text{ in } Q \mid t_X \not\prec^* t] \cdot [t \text{ in } Q \mid t_X \prec^* t]$$

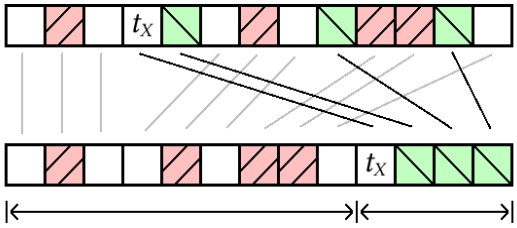


FIGURE 3.2 – Stratégie consistant à déplacer t_X et ses descendances à la fin de la queue.

Cette stratégie déplace en particulier t_X derrière ses ascendances, ce qui satisfait l'invariant (I). On peut considérer que cette stratégie s'applique également au cas d'un thread qui entre dans la queue en se mettant en attente alors qu'il ne possède aucun mutex : il n'a alors aucune descendance et est donc bien mis simplement à la fin de Q.

3.4.4 Possibilité de famine

Nous avons implémenté cette première stratégie et n'avons observé de situation de famine sur aucun des programmes que nous utilisons comme benchmarks (voir Section 4.3). Une telle situation est cependant possible ; voici un exemple de programme dont l'exécution peut contenir une situation de famine :

```
1  (define R (make-region))
2  (define m1 (make-mutex R))
3  (define m2 (make-mutex R))
4  (define m3 (make-mutex R))
5
6  (thread  ; tx
7     ...
8     (synchronize (list m1 m2 m3)
9        #f ))
10
```

```
(while #t
   (thread  ; ta(i)
      (synchronize m1 :prelock m2
         ...
         (synchronize m2
            #f )))
   (thread  ; tb(i)
      (synchronize m3 :prelock m2
         ...
         (synchronize m2
            #f )))
   )
```

Cet exemple crée 3 mutex et une infinité de threads : un thread t_x qui tente de verrouiller les 3 mutex à la fois, et une infinité de paires de threads $t_{a,i}$ et $t_{b,i}$ qui verrouillent respectivement {`m1` et `m2`} et {`m3` et `m2`} comme ci-dessus.

Lors d'une exécution de cet exemple, t_x peut se retrouver affamé. Voici comment. On suppose que t_x n'est pas le premier thread à exécuter son `synchronize` et que plusieurs $t_{a,i}$ et $t_{b,i}$ sont déjà en attente dans la queue.

Lorsqu'un thread $t_{a,n}$ exécute son `synchronize m2`, il est déplacé vers la fin de la queue avec toutes ses descendances. À ce moment $t_{a,n}$ possède `m1`, et seulement `m1`. Ses descendances incluent donc les autres $t_{a,i}$ et t_x mais pas les $t_{b,i}$. t_x se retrouve donc derrière tous les $t_{b,i}$ présents dans la queue, sans changer de position relativement aux $t_{a,i}$.

Inversement, lorsqu'un thread $t_{b,n}$ exécute son `synchronize m2`, t_x est déplacé derrière tous les $t_{a,i}$ sans changer de position relativement aux $t_{b,i}$.

Tant que des $t_{a,i}$ et $t_{b,i}$ s'exécutent, t_x recule donc perpetuellement dans la queue et n'obtient jamais satisfaction.

3.4.5 Variante sans famine

3.4.5.1 Description

Pour éviter qu'un ou plusieurs threads en attente n'obtiennent jamais satisfaction, la solution que nous proposons consiste à systématiquement privilégier le thread qui est en attente depuis le plus longtemps (s'il existe), que nous appellerons le *doyen* et noterons t_d, par rapport aux autres. Cette méthode est une variante minimaliste des techniques d'ordonnancement avec *vieillissement*, qui consistent à augmenter la priorité de processus en attente proportionnellement à la durée de leur attente [42, 46].

Nous montrerons qu'alors, sous certaines conditions, le doyen finit forcément par progresser, et que donc chaque thread du système finit par progresser.

Le doyen pouvant être bloqué par d'autre threads (ses ascendances), on stipule que cet ensemble de threads (t_d inclus) sera devant tous les autres dans Q. À l'invariant (I) défini page 59 on adjoint donc l'invariant (J) :

$$\text{(I)} \qquad \forall\, t_1, t_2.\ t_1 \prec^+ t_2 \ \Rightarrow\ t_1 < t_2$$

$$\text{(J)} \qquad \forall\, t_1, t_2.\ t_1 \prec^* t_d \ \wedge\ t_2 \not\prec^* t_d \ \Rightarrow\ t_1 < t_2$$

Ces deux invariants ne sont pas contradictoires car si $\exists\, t_1,\ t_2.\ t_1 \prec^* t_d \wedge t_2 \prec^+ t_1$ alors par définition $t_2 \prec^* t_d$.

Le thread t_d est nécessairement plus grand que toutes ses ascendances ; l'invariant (J) est donc équivalent à $\forall\, t.\ t \prec^* t_d \ \Leftrightarrow\ t \leq t_d$.

Au fil de l'exécution du programme, pendant la période où un thread donné est le doyen, des threads peuvent entrer ou sortir de l'ensemble de ses ascendances et donc passer devant ou derrière t_d dans la queue.

3.4.5.1.1 Stratégie

Les deux propriétés énoncées ci-après, dont l'absence de famine, reposent sur les invariants (I) et (J) et non sur la stratégie choisie.

Dans notre implémentation, le comportement que nous avons choisi d'adopter lorsque qu'un thread t_X déjà dans Q se met en attente de nouveaux mutex est le suivant :

— Si $t_d < t_X$, alors t_X et ses descendances sont déplacés vers la fin de la queue, comme dans la première stratégie présentée en Section 3.4.3 :
$Q \to [t \text{ in } Q \mid t_X \not\prec^* t] \cdot [t \text{ in } Q \mid t_X \prec^* t]$
— Si $t_X < t_d$, alors t_X garde sa place dans Q et, parmi ses nouvelles ascendances, celles qui doivent être déplacées parce qu'elle sont derrière t_X sont déplacées à l'avant de Q avec leurs propres ascendances :
$Q \to \text{let } S = \{t \in Q \mid t \prec^+ t_X \wedge t_X <_Q t\} \text{ in}$
$\quad [t \text{ in } Q \mid \exists\, t' \in S.\ t \prec^* t'] \cdot [t \text{ in } Q \mid \nexists\, t' \in S.\ t \prec^* t']$

Nous n'avons expérimenté qu'une seule stratégie car différentes stratégies se distinguent principalement dans le cas de relations de blocage non triviales entre threads, or, comme nous le verrons en Section 4.3.3, nous avons peiné à trouver des programmes qui présentent des imbrications complexes de sections critiques.

Nous avons choisi la présente stratégie pour sa relative simplicité compte tenu des structures de données que nous avons choisi dans notre implémentation (voir Section 4.1) : les ensembles de mutex déplacés le sont toujours vers une extrémité de la queue ; le choix des ensembles de mutex déplacés nous semble une façon relativement naturelle de satisfaire l'invariant (I).

3.4.5.1.2 Hypothèses

Les deux propriétés qui viennent sont valables dans le cas du verrouillage structuré sans variables de conditions, c'est-à-dire dans le cas où les fonctions `lock-n` et `unlock-n` ne sont utilisées que pour implémenter des blocs synchronisés.

Nos preuves reposent également sur le fait que les opérations de mise en attente, prise effective et relâche de mutex se font séquentiellement sous un verrou global.

3.4.5.2 Première propriété

Au cours de l'exécution d'un programme, pendant le temps qu'un thread t_d est le doyen, si chacun des mutex que t_d attend est relâché au moins une fois, alors t_d obtient satisfaction.

La preuve de l'absence globale de famine ne repose pas sur cette première propriété. En revanche, la preuve de cette propriété est une version simplifiée de la preuve d'absence globale de famine ; celle-ci peut donc aider à comprendre celle-là.

3.4.5.2.1 Preuve

À un instant initial donné, soit $S := \{t \mid t \prec t_d\}$. Faisons évoluer S comme suit : si à un instant donné, pour $t \in S$, on n'a plus $t \prec t_d$, t est retiré de S.

Alors à tout instant, $\forall\ t \prec^+ t_d, \exists\ t' \in S.\ t \prec^* t'$.

En effet, par récurrence sur le temps :

— c'est clairement vrai lorsqu'on initialise S
— ensuite, l'affirmation pourrait s'invalider de 3 façons :

1. Il apparaît un nouveau $t \prec^+ t_d$. On a 2 cas possibles :
   (a) $t \prec t_d$ (t acquiert m que t_d attend). Ça signifie que juste avant on avait déjà $t < t_d$ (puisque t_d attend m aussi). Donc on avait déjà $t \prec^* t_d$. Donc par hypothèse de récurrence $\exists\ t' \in S.\ t \prec^* t'$.
   (b) $t \prec t_1 \prec^+ t_d$. Alors par hypothèse de récurrence $\exists\ t' \in S.\ t_1 \prec^* t'$ et donc $t \prec^* t'$.
2. Pour t et t' tels que ci-dessus, t' sort de S. Ça signifie que t' progresse, donc t' n'est pas bloqué par t. Cas éliminé.
3. Pour t et t' tels que ci-dessus, on n'a plus $t \prec^* t'$ (donc $t' \neq t$; on avait $t \prec t_1 \prec^* t'$ et on n'a plus $t \prec t_1$), mais on a toujours $t \prec^+ t_d$. Distinguons 3 cas :
   (a) $t \prec t_2 \prec^+ t_d$. Alors par hypothèse de récurrence $\exists\ t'' \in S.\ t_2 \prec^* t''$ et donc $t \prec^* t''$.
   (b) $t \prec t_d$ et $t \in S$. Alors on a bien $t \prec^* t$.
   (c) $t \prec t_d$ et $t \notin S$. Alors $t \prec t_d$ est devenu vrai après l'initialisation de S. À ce moment on avait déjà $t < t_d$ donc $t \prec^+ t_d$ donc $\exists\ t'' \in S.\ t \prec^* t''$. Or cette relation est toujours vraie. En effet, comme on suppose du verrouillage structuré, t n'a pas encore relâché les mutex impliqués par cette relation qu'il possède, puisqu'il vient seulement de relâcher des mutex qu'il a acquis après, et les autres threads présents dans la relation n'ont donc pas progressé non plus.

Conclusion : quand chaque mutex qu'attend t_d a été relâché au moins une fois, S est vide, donc t_d n'est bloqué par personne, donc il obtient satisfaction. □

3.4.5.3 Absence globale de famine

Prouvons maintenant que notre solution est sans famine (voir Section 3.3.3), autrement dit que dans toute exécution de tout programme :

Si chaque thread qui progresse finit par relâcher tous ses mutex **et si** pour aucun thread t en attente, le nombre des ascendances de t ne tend vers l'infini, **alors** chaque thread finit par progresser.

3.4.5.3.1 Preuve

Supposons que tout thread qui progresse finit par relâcher tous ses mutex.

Lemme Si un doyen donné ne progresse jamais, alors le nombre de ses ascendances tend vers l'infini.

Preuve du lemme

En effet, supposons que t_d ne progresse jamais. Alors on construit algorithmiquement une famille infinie (S_k) d'ensembles non vides et disjoints d'ascendances de t_d.

À tout instant, soit n le nombre de S_k déjà calculés. $S_0..S_{n-1}$ sont fixés et contiennent des threads qui ne progresseront plus jamais. S_n est en cours de calcul. On fait tendre n vers $+\infty$.

À tout instant, pour tout $p \leq n$, appelons $S^p_\infty = \{t \mid t \prec^* t_d\} \setminus \bigcup_{0..p} S_k$. Notons que pour tous $p < q$, $S^q_\infty \subset S^p_\infty$.

À tout instant, on vérifiera pour tout $p \leq n$ l'invariant :

(K_p) $\forall\ t \in S^p_\infty.\ \exists\ t' \in S_p.\ t \prec^* t'$

À $n = 0$, fixons $S_0 = \{t_d\}$ (qui ne progressera plus jamais par hypothèse). (K_0) sera clairement vrai à tout instant dans le futur.

À $n > 0$, en supposant que $S_0..S_{n-1}$ ne changeront plus et ne contiennent que des threads qui n'évolueront plus, et en supposant acquis (pour toujours) $(K_0)..(K_{n-1})$,

Assignons $S_n := \{t \mid \exists\ t' \in S_{n-1}.\ t \prec t'\}$

et faisons évoluer S_n comme suit : si à un instant donné $\exists\ t \in S_n.\ \neg(\exists\ t' \in S_{n-1}.\ t \prec t')$, alors $S_n := S_n \setminus \{t\}$.

Preuve de (K_n)

Prouvons que (K_n) est vrai lorsqu'on initialise S_n et à tout instant ultérieur, par induction sur le temps.

Lorsqu'on initialise S_n, $\forall\ t \in S^n_\infty.\ t \in S^{n-1}_\infty$ donc d'après (K_{n-1}) $\exists\ t'' \in S_{n-1}.\ t \prec^* t''$ donc vu la définition de S_n on a (K_n).

Ensuite (K_n) pourrait devenir faux de 3 façons :

1. Il apparaît un nouveau $t \in S_\infty^n$. Montrons qu'$\exists\ t' \in S_n.\ t \prec^* t'$. Le présent cas peut se produire de 2 façons :

   (a) t devient ascendance de t_d. D'après (J), ce ne peut être que t acquiert `m` attendu par $t_1 \prec^* t_d$. C'est donc que $t_1 \prec^* t_d$ se met en attente de `m` que t possède. $t_1 \in S_\infty^{n-1}$ car les autres ascendances de t_d sont bloquées. Distinguons 2 sous-cas :

      i. $t_1 \in S_n$. Alors $t' = t_1$. ✓
      ii. $t_1 \in S_\infty^n$. Alors par hypothèse d'induction, c'est-à-dire (K_n) appliqué à t_1, $\exists\ t' \in S_n.\ t \prec t_1 \prec^* t'$. ✓

   (b) t sort de S_n mais on a toujours $t \prec^* t_d$. Alors d'après (K_{n-1}), $\exists\ t'' \in S_{n-1}.\ t \prec^* t''$, mais pas $t \prec t''$ puisque t sort de S_n, donc $t \prec t_1 \prec^+ t''$ et on conclut comme précédemment. ✓

2. Pour t, t' tels que dans (K_n), t' sort de S_n. Ça signifie que t' relâche des mutex, autrement dit progresse, ce qui contredit $t \prec^* t'$. ✓
3. Pour t, t' tels que dans (K_n), on n'a plus $t \prec^* t'$ mais on a toujours $t \in S_\infty^n$. Ça signifie que t relâche un ou plusieurs mutex. D'après (K_{n-1}) $\exists\ t'' \in S_{n-1}.\ t \prec^* t''$. Distinguons 2 cas :

   (a) $t \prec t_1 \prec^+ t''$ et on conclut comme précédemment. ✓
   (b) $t \prec t''$. Distinguons 2 sous-cas :

      i. Si $t \prec t''$ était vrai depuis l'initialisation de S_n ou avant, alors on aurait $t \in S_n$, ce qui n'est pas le cas par hypothèse. ✓
      ii. Sinon, au moment où $t \prec t''$ est devenu vrai on avait $t \in S_\infty^n$ donc par hypothèse d'induction on avait $\exists\ t' \in S_n.\ t \prec^* t'$. Or cette relation est toujours vraie. En effet, comme on suppose du verrouillage structuré, t n'a pas encore relâché les mutex impliqués par cette relation qu'il possède, puisqu'il vient seulement de relâcher des mutex qu'il a acquis après, et les autres threads présents dans la relation n'ont donc pas progressé non plus. ✓

Donc (K_n) est vrai et le sera à tout instant dans le futur. $\Box\ (K_n)$

Soit $t_m = min_< \bigcup_{0..n-1} S_k$.

Alors,

$$\begin{aligned} S_n = \emptyset &\overset{(K_n)}{\Longrightarrow} S_n \cup S_\infty^n = \emptyset \\ &\Longrightarrow \forall\ t \prec^* t_d.\ t \in \bigcup_{0..n-1} S_k \\ &\overset{(J)}{\Longrightarrow} \nexists\ t.\ t < t_m \\ &\overset{(I)}{\Longrightarrow} t_m\ \textit{progresse} \end{aligned}$$

Or par hypothèse de récurrence tous les threads dans $\bigcup_{0..n-1} S_k$ ne progresseront plus jamais. Donc S_n ne sera jamais vide.

Or S_n ne peut que perdre des threads, donc à un moment donné il cessera définitivement d'évoluer. À partir de ce moment, si au moins un thread $t \in S_n$ continuait à progresser indéfiniment, il finirait par relâcher tous les mutex qu'il

possède et à sortir de S_n, ce qui est impossible. Donc à un moment donné tous les threads contenus dans S_n seront définitivement bloqué.

On passe alors au calcul de S_{n+1}.

On construit bien ainsi une famille infinie d'ensembles non vides et disjoints d'ascendances de t_d, donc le nombre des ascendances de t_d tend vers $+\infty$. □ (Lemme)

Donc par contraposée, si pour aucun thread t le nombre des ascendances de t ne tend vers $+\infty$, alors t_d finit par progresser.

Alors si certains threads de l'exécution considérée ne progressaient jamais, il est clair que l'un d'entre eux finirait par être le thread en attente depuis le plus long-temps, autrement dit le doyen, et finirait donc par progresser. (Contradiction.)

Donc chaque thread du système finit par progresser. □

3.5 Sémantique formelle

Dans cette section on présente une sémantique opérationnelle des principales fonctions de notre bibliothèque, faisant la synthèse de la description qu'on en a donnée dans ce chapitre. Plus exactement, on donne la sémantique d'un langage minimaliste donc les seules constructions sont celles de notre bibliothèque.

Nous avons vu en Section 3.4 que plusieurs *stratégies* satisfont les invariants (I) et (J) sur lesquels repose l'absence de famine. Notre sémantique est paramétrée par la stratégie choisie. Cette stratégie apparaît dans les règles de la sémantique comme une fonction $\mathscr{S}$ qui prend en paramètres la queue globale Q et l'état du système et retourne une nouvelle queue Q'. Nous avons décrit en Section 3.4.5.1.1 la stratégie que nous avons choisi d'implémenter.

Dans un souci de clarté, dans cette sémantique :

- on considère des mutex non-récursifs, autrement dit on ne permet pas à un thread de verrouiller un mutex qu'il possède déjà ;
- une opération de synchronisation ne peut verrouiller que des mutex appartenant à une même région (la généralisation à une synchronisation impliquant plusieurs régions, décrite en Section 3.2.6, pourrait être appliquée ici également) ;
- on exclut également les variables de conditions.

Pour chaque aspect de la sémantique, on renvoie le lecteur aux explications plus détaillées données dans les sections précédentes.

3.5.1 Syntaxe

Dans notre langage, les seules valeurs sont des mutex, des ensembles de mutex et *unit*, et les seules opérations sont le verrouillage structuré et le lancement d'un thread. Ce langage n'a de sens qu'en tant que fragment d'un hypothétique langage plus complet avec des constructions plus généralistes telles que des fonctions.

Dans le présent langage, les *mutex* sont simplement des entiers distincts les uns des autres et attribués dans l'ordre croissant : $m_1 < m_2$ ssi. m_1 a été instancié avant m_2.

r			*régions*
x			*variables*
v	::=	$() \mid x \mid \{v*\}$	*valeurs*
e	::=	v	*expressions*
	|	$\{e*\}$	
	|	$(\mathsf{thread}\ e)$	
	|	$(\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)$	
	|	$(\mathsf{sync}\ e\ e\ e)$	

Typage informel On ne considère que les expressions dans lesquelles :

- les ensembles $\{e*\}$ ne contiennent que des expressions qui se réduisent en des mutex ;
- dans $(\mathsf{sync}\ e_0\ e_1\ e_2)$, e_0 et e_1 se réduisent toujours en des ensembles.

Le programmeur ne peut que créer des ensembles. Notre sémantique, en revanche, utilise en interne les opérations mathématiques habituelles sur ces ensembles.

3.5.2 Expressions réductibles et configurations

Les rédex incluent les constructions wait et insync qui ne sont pas exposées au programmeur, mais seulement générées par l'exécution de la sémantique.

ρ	::=	$(\mathsf{thread}\ e)$	*rédex*
	|	$(\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)$	
	|	$(\mathsf{sync}\ v\ v\ e)$	
	|	$(\mathsf{wait}\ v\ v\ e) \mid (\mathsf{insync}\ v)$	*expressions internes*
$\mathbf{E}$	::=	$[] \mid \mathbf{E}[\mathbf{F}]$	*contextes d'évaluation*
$\mathbf{F}$	::=	$[] \mid \{..., [], ...\}$	
	|	$(\mathsf{sync}\ []\ e\ e) \mid (\mathsf{sync}\ e\ []\ e)$	
	|	$(\mathsf{insync}\ [])$	

Threads

Les threads sont de la forme $\mathbf{E}[\rho]_{id}$ où

- id est un identificateur ;
- $\mathbf{E}$ est un contexte d'évaluation ;
- e est une expression réductible ou une valeur.

La seule chose qu'on attend des identificateurs de threads est qu'ils soient distincts les uns des autres.

Ensembles associatifs de piles

À chaque thread t sont associées plusieurs valeurs, dont certaines changent (uniquement) lorsque t entre dans une section critique et sont restaurées à leur valeur précédente lorsque t sort de cette section critique.

Dans les configurations, ces valeurs sont représentées par des ensembles de paires associant à chaque identificateur de thread une pile de valeurs, le sommet de la pile étant la valeur courante pour le thread.

Configurations

Chaque configuration est formée de 11 composants :

- Q est la queue (l'ensemble ordonné) des identificateurs de threads actifs (voir Section 3.4) ;
- Q^W est une queue des identificateurs de threads en attente ; son premier élément est donc le doyen (3.4.5) ;
- R^T est l'ensemble de piles associant à chaque thread sa région courante (3.2.3) ;
- R^M associe à chaque mutex sa région (non mutable) (3.2.3) ;
- R^R est un ensemble de paires de régions $(r_1,\ r_2)$ telles que $r_1 < r_2$; R^R est tel que la relation $<$ soit la fermeture transitive de cet ensemble (3.2.3) ;
- A est un ensemble de piles associant à chaque thread l'ensemble de mutex que ce thread est explicitement autorisé à acquérir, c'est-à-dire l'ensemble de préverrouillage de la section critique courante de ce thread, ou $\emptyset$ si le thread n'est pas dans une section critique (3.2.4) ;
- N est un ensemble de piles associant à chaque thread le dernier identificateur de mutex (> 0) attribué lorsque le thread est entré dans sa section critique courante, ou 0 si le thread n'est pas dans une section critique (3.2.4.1) ;
- O est un ensemble de piles associant à chaque thread l'ensemble de mutex que ce thread possède ;
- W est un ensemble de piles associant à chaque thread l'ensemble de mutex que ce thread attend actuellement de pouvoir prendre ;
- n est la valeur (entière) du dernier mutex à avoir été instancié (3.2.4.1) ;
- T est l'ensemble (non ordonné) des threads.

Style chimique

Pour simplifier légèrement les règles sémantiques, on y utilisera pour les configurations la syntaxe suivante :

$\Gamma ::= Q \mid Q^W \mid R^T \mid R^M \mid R^R \mid A \mid N \mid O \mid W \mid n \mid t \mid (\Gamma \parallel \Gamma)$ où t est un seul thread ($t \in T$).

On suppose que la composition parallèle est commutative et associative de sorte que les règles puissent être exprimées dans le « style chimique » de [11], spécifiant des « réactions » locales de la forme $\Gamma \to \Gamma'$ pouvant prendre place à n'importe quel endroit de la configuration.

Sucre syntaxique et autres notations

- $x : l$ signifie que x et l sont respectivement la tête et la queue d'une liste ou pile
- $q_1 \cdot q_2$ est la concaténation des queues q_1 et q_2
- $x * S \triangleq \{x\} \cup S$
- $x_i \triangleq (i, x)$
- Si $(i, (x : l)) \in S$ alors $S(i) \triangleq x$

Configuration initiale

Pour exécuter un programme constitué d'une expression e, on utilise la configuration initiale suivante :

$Q = [] \,;\, Q^W = [] \,;\, R^T = \{\top_0\} \,;\, R^M = \emptyset \,;\, R^R = \emptyset \,;\, A = \{\emptyset_0\} \,;\, N = \{0_0\} \,;\, O = \{\emptyset_0\} \,;\, W = \{\emptyset_0\} \,;\, n = 0 \,;\, T = \{(e)_0\}$

3.5.3 Règles

Ces règles sont regroupées Figure 3.3.

$$\frac{j \text{ fresh}}{R^T \;||\; A \;||\; N \;||\; O \;||\; W \;||\; \mathbf{E}[(\mathsf{thread}\ e)]_i \;\rightarrow\; \top_j * R^T \;||\; \emptyset_j * A \;||\; 0_j * N \;||\; \emptyset_j * O \;||\; \emptyset_j * W \;||\; \mathbf{E}[()]_i \;||\; e_j} \;\textsc{Thread}$$

Thread Lors du lancement d'un thread on crée un identificateur j, on ajoute le corps du thread (avec cet identificateur) à l'ensemble des threads en cours d'exécution, et on initialise les variables associées au nouveau thread.

$$\frac{}{R^M \;||\; R^R \;||\; n \;||\; \mathbf{E}[(\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)]_i \;\rightarrow\; r_{n+1} * R^M \;||\; (r, \top) * R^R \;||\; n+1 \;||\; \mathbf{E}[(\{x \mapsto n+1\}e)]_i} \;\textsc{LetMutex}$$

LetMutex Lors de la création d'un mutex, on incrémente le compteur global n (voir Section 3.2.4.1) et on assigne la nouvelle valeur à la variable souhaitée. On associe à ce nouveau mutex la région donnée r, dont on enregistre qu'elle est inférieure à $\top$. Si $r = \top$, cette opération laisse l'ordre partiel sur les régions inchangé.

$$\frac{\begin{array}{c}(L \cup P) \cap O(i) = \emptyset \qquad \forall\, m \in (L \cup P).\ R^M(m) = r \\ \forall\, m \in (L \cup P).\ m \in A(i) \vee m \geq N(i) \qquad Q' = \mathscr{S}(Q, Q^W, O, W, i, L, P)\end{array}}{\begin{array}{rl} & Q \;||\; Q^W \;||\; (r : rs)_i * R^T \;||\; R^M \;||\; A \;||\; O \;||\; W \;||\; \mathbf{E}[(\mathsf{sync}\ L\ P\ e)]_i \\ \rightarrow & Q' \;||\; Q^W \cdot i \;||\; (r : r : rs)_i * R^T \;||\; R^M \;||\; A \;||\; O \;||\; (L \cup P)_i * W \;||\; \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i\end{array}} \;\textsc{WaitSR}$$

$$\frac{j \text{ fresh}}{R^T \mid\mid A \mid\mid N \mid\mid O \mid\mid W \mid\mid \mathbf{E}[(\mathsf{thread}\ e)]_i \ \rightarrow\ \top_j * R^T \mid\mid \emptyset_j * A \mid\mid 0_j * N \mid\mid \emptyset_j * O \mid\mid \emptyset_j * W \mid\mid \mathbf{E}[()]_i \mid\mid e_j}\ \textsc{Thread}$$

$$\frac{}{R^M \mid\mid R^R \mid\mid n \mid\mid \mathbf{E}[(\mathsf{let}\ (x\ (\mathsf{mutex}\ r))\ e)]_i \ \rightarrow\ r_{n+1} * R^M \mid\mid (r, \top) * R^R \mid\mid n+1 \mid\mid \mathbf{E}[(\{x \mapsto n+1\}e)]_i}\ \textsc{LetMutex}$$

$$\frac{\begin{array}{c}(L \cup P) \cap O(i) = \emptyset \qquad \forall\ m \in (L \cup P).\ R^M(m) = r \\ \forall\ m \in (L \cup P).\ m \in A(i) \vee m \geq N(i) \qquad Q' = \mathscr{S}(Q, Q^W, O, W, i, L, P)\end{array}}{\begin{array}{rl} & Q \mid\mid\ Q^W \mid\mid\ (r : rs)_i * R^T \mid\mid R^M \mid\mid A \mid\mid O \mid\mid\ W \mid\mid \mathbf{E}[(\mathsf{sync}\ L\ P\ e)]_i \\ \rightarrow & Q' \mid\mid Q^W \cdot i \mid\mid (r : r : rs)_i * R^T \mid\mid R^M \mid\mid A \mid\mid O \mid\mid (L \cup P)_i * W \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i\end{array}}\ \textsc{WaitSR}$$

$$\frac{\begin{array}{c}(L \cup P) \cap O(i) = \emptyset \qquad \forall\ m \in (L \cup P).\ R^M(m) = r' \\ r \not<^* r' \qquad Q' = \mathscr{S}(Q, Q^W, O, W, i, L, P)\end{array}}{\begin{array}{rl} & Q \mid\mid\ Q^W \mid\mid\ (r : rs)_i * R^T \mid\mid R^M \mid\mid\ R^R \mid\mid O \mid\mid\ W \mid\mid \mathbf{E}[(\mathsf{sync}\ L\ P\ e)]_i \\ \rightarrow & Q' \mid\mid Q^W \cdot i \mid\mid (r' : r : rs)_i * R^T \mid\mid R^M \mid\mid (r', r) * R^R \mid\mid O \mid\mid (L \cup P)_i * W \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i\end{array}}\ \textsc{WaitCR}$$

$$\frac{(L \cup P) \cap O = \emptyset \qquad \forall\ j.\ i \leq_Q j \vee L \cap W(j) = \emptyset \qquad Q_l^W \neq [] \vee Q_r^W = []}{\begin{array}{rl} & Q \mid\mid Q_l^W \cdot i \cdot Q_r^W \mid\mid\ a_i * A \mid\mid\ ns_i * N \mid\mid\ o_i * O \mid\mid w_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i \\ \rightarrow & Q \mid\mid\ Q_l^W \cdot Q_r^W \mid\mid (P : a)_i * A \mid\mid (n : ns)_i * N \mid\mid ((L \cup o) : o)_i * O \mid\mid\ \emptyset_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{insync}\ e)]_i\end{array}}\ \textsc{EnterNND}$$

$$\frac{(L \cup P) \cap O = \emptyset \qquad \forall\ j.\ i \leq_Q j \vee L \cap W(j) = \emptyset \qquad Q' = [j \in Q \mid j \prec^* d] \cdot [j \in Q \mid j \not\prec^* d]}{\begin{array}{rl} & Q \mid\mid i \cdot d \cdot Q^W \mid\mid\ a_i * A \mid\mid\ ns_i * N \mid\mid\ o_i * O \mid\mid w_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i \\ \rightarrow & Q' \mid\mid\ d \cdot Q^W \mid\mid (P : a)_i * A \mid\mid (n : ns)_i * N \mid\mid ((L \cup o) : o)_i * O \mid\mid\ \emptyset_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{insync}\ e)]_i\end{array}}\ \textsc{EnterND}$$

$$\frac{o \neq \emptyset}{\begin{array}{rl} & (r_0 : r)_i * R^T \mid\mid (a_0 : a)_i * A \mid\mid (n_0 : ns)_i * N \mid\mid (o_0 : o)_i * O \mid\mid \mathbf{E}[(\mathsf{insync}\ v)]_i \\ \rightarrow & r_i * R^T \mid\mid\ a_i * A \mid\mid\ ns_i * N \mid\mid\ o_i * O \mid\mid \mathbf{E}[v]_i\end{array}}\ \textsc{UnlockN}$$

$$\frac{}{\begin{array}{rl} & Q_l \cdot i \cdot Q_r \mid\mid (r_0 : \top)_i * R^T \mid\mid (a_0 : \emptyset)_i * A \mid\mid (n_0 : 0)_i * N \mid\mid (o_0 : \emptyset)_i * O \mid\mid \mathbf{E}[(\mathsf{insync}\ v)]_i \\ \rightarrow & Q_l \cdot Q_r \mid\mid\ \top_i * R^T \mid\mid\ \emptyset_i * A \mid\mid\ 0_i * N \mid\mid\ \emptyset_i * O \mid\mid \mathbf{E}[v]_i\end{array}}\ \textsc{UnlockTL}$$

FIGURE 3.3 – Règles de la sémantique opérationnelle de jthread.

WAIT (*Same Region*) Lorsqu'un thread t_i exécute un `sync`, on commence par vérifier que t_i ne possède déjà aucun des mutex à verrouiller et à préverrouiller (les mutex sont non-récursifs dans cette sémantique), et que ces mutex appartiennent à une même région (autre hypothèse simplificatrice pour cette sémantique).

Si cette région est la région courante r de t_i, on vérifie que chaque mutex m est autorisé par la Règle 2 ($m \in A(i)$) ou par la Règle 3 ($m \geq N(i)$) (Section 3.2.4).

Alors le rédex `(sync ...)` est remplacé par un rédex intermédiaire `(wait ...)`, l'ensemble de mutex attendus par t_i devient $L' \cup P'$, r est rempilée comme région courante de t_i pour cette section critique, i est ajouté à la queue des threads en attente, et enfin Q est transformée selon la stratégie $\mathscr{S}$ choisie.

Les configurations qui produiraient des erreurs dans notre implémentation (acquisitions non autorisées) ne sont pas représentées dans la sémantique ; elles sont irréductibles.

$$\frac{\begin{array}{c}(L \cup P) \cap O(i) = \emptyset \qquad \forall\, m \in (L \cup P).\ R^M(m) = r' \\ r \not<^* r' \qquad Q' = \mathscr{S}(Q, Q^W, O, W, i, L, P)\end{array}}{\begin{array}{rl} & Q \mid\mid\ Q^W \mid\mid\ (r : rs)_i * R^T \mid\mid R^M \mid\mid\ R^R \mid\mid O \mid\mid\ W \mid\mid \mathbf{E}[(\mathsf{sync}\ L\ P\ e)]_i \\ \rightarrow & Q' \mid\mid Q^W \cdot i \mid\mid (r' : r : rs)_i * R^T \mid\mid R^M \mid\mid (r', r) * R^R \mid\mid O \mid\mid (L \cup P)_i * W \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i \end{array}}\ \textsc{WaitCR}$$

WAIT (*Change Region*) Cette règle est la variante de la précédente correspondant au cas où la région r' des mutex à prendre est différente de la région courante r de t_i (Règle 1, Section 3.2.3).

Alors on impose que $r \not<^* r'$ et on enregistre $r' > r$. r' devient la région courante de t_i. Les autres éléments de la configuration sont modifié comme dans la règle précédente.

$$\frac{(L \cup P) \cap O = \emptyset \qquad \forall\, j.\ i \leq_Q j \vee L \cap W(j) = \emptyset \qquad Q_l^W \neq [] \vee Q_r^W = []}{\begin{array}{rl} & Q \mid\mid Q_l^W \cdot i \cdot Q_r^W \mid\mid\ a_i * A \mid\mid\ ns_i * N \mid\mid\ o_i * O \mid\mid w_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i \\ \rightarrow & Q \mid\mid\ Q_l^W \cdot Q_r^W \mid\mid (P : a)_i * A \mid\mid (n : ns)_i * N \mid\mid ((L \cup o) : o)_i * O \mid\mid\ \emptyset_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{insync}\ e)]_i \end{array}}\ \textsc{EnterNND}$$

ENTER (*No New Doyen*) Cette règle correspond au cas où un thread t_i prend les mutex qu'il attend et entre dans sa section critique, et qu'il n'y a pas de nouveau doyen.

Un thread peut prendre les mutex qu'il attend à la condition que ceux-ci soient libres, et ne soient pas attendus par un thread $t_j < t_i$.

Il n'y a pas de nouveau doyen soit parce que t_i n'est pas le doyen ($Q_l^W \neq []$), soit parce que t_i est le doyen mais qu'il n'y a pas d'autre thread en attente pour prendre sa place ($Q_r^W = []$).

Alors le rédex intermédiaire `(wait ...)` est remplacé par un autre : `(insync ...)`. i est retiré de la queue des threads en attente et l'ensemble des mutex attendus par t_i devient $\emptyset$. Les mutex verrouillés s'ajoutent à ceux déjà possédés par t_i. Enfin, l'ensemble de mutex explicitement autorisés de t_i devient P (Règle 2, Section 3.2.4) et la valeur courante de n est sauvegardée (Règle 3, Section 3.2.4.1).

$$\frac{(L \cup P) \cap O = \emptyset \qquad \forall\, j.\ i \leq_Q j \vee L \cap W(j) = \emptyset \qquad Q' = [j \text{ in } Q \mid j \prec^* d] \cdot [j \text{ in } Q \mid j \not\prec^* d]}{\begin{array}{rl} & Q \mid\mid i \cdot d \cdot Q^W \mid\mid \quad a_i * A \mid\mid \quad ns_i * N \mid\mid \quad o_i * O \mid\mid w_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{wait}\ L\ P\ e)]_i \\ \to & Q' \mid\mid \quad d \cdot Q^W \mid\mid (P : a)_i * A \mid\mid (n : ns)_i * N \mid\mid ((L \cup o) : o)_i * O \mid\mid \emptyset_i * W \mid\mid n \mid\mid \mathbf{E}[(\mathsf{insync}\ e)]_i \end{array}}\ \textsc{EnterND}$$

ENTER (*New Doyen*) Cette règle est la variante de la précédente pour le cas où il y a un nouveau doyen, c'est-à-dire que t_i est le doyen et qu'il est remplacé par un autre, t_d.

Les conditions requises pour que le thread entre dans sa section critique sont les mêmes que pour la règle précédente.

Alors Q est réorganisée par rapport au nouveau doyen t_d de façon à satisfaire l'invariant (J) (Section 3.4.5). t_i peut être une ascendance de t_d ou non.

Les autres éléments de la configuration sont modifiés comme dans la règle précédente.

$$\frac{o \neq \emptyset}{\begin{array}{rl} & (r_0 : r)_i * R^T \mid\mid (a_0 : a)_i * A \mid\mid (n_0 : ns)_i * N \mid\mid (o_0 : o)_i * O \mid\mid \mathbf{E}[(\mathsf{insync}\ v)]_i \\ \to & \quad r_i * R^T \mid\mid \quad a_i * A \mid\mid \quad ns_i * N \mid\mid \quad o_i * O \mid\mid \mathbf{E}[v]_i \end{array}}\ \textsc{UnlockN}$$

$$\frac{}{\begin{array}{rl} & Q_l \cdot i \cdot Q_r \mid\mid (r_0 : \top)_i * R^T \mid\mid (a_0 : \emptyset)_i * A \mid\mid (n_0 : 0)_i * N \mid\mid (o_0 : \emptyset)_i * O \mid\mid \mathbf{E}[(\mathsf{insync}\ v)]_i \\ \to & \quad Q_l \cdot Q_r \mid\mid \quad \top_i * R^T \mid\mid \quad \emptyset_i * A \mid\mid \quad 0_i * N \mid\mid \quad \emptyset_i * O \mid\mid \mathbf{E}[v]_i \end{array}}\ \textsc{UnlockTL}$$

UNLOCK (*Nested / Top-Level*) Lorsqu'un thread t_i arrive à la fin d'une section critique, c'est-à-dire que le corps d'un rédex est irréductible, ce rédex est remplacé par son corps et les valeurs temporaires de R^T, A, N et O associées à t_i sont dépilées. Si t_i ne possède plus aucun mutex ($o = \emptyset$) et sort de sa section critique la plus extérieure (« top-level »), i est retiré de Q. Si t_i ne sort que d'une section critique imbriquée, Q reste inchangée.

3.5.4 Exemple de réduction

Voici, à titre d'exemple, la réduction d'un programme minimal impliquant un verrouillage :

$$(\mathsf{let}\ (m\ (\mathsf{mutex}\ \top))\ (\mathsf{sync}\ \{m\}\ \emptyset\ ()))$$

Les étapes de la réduction sont les suivantes :

$Q = []$; $Q^W = []$; $R^T = \{\top_0\}$; $R^M = \emptyset$; $R^R = \emptyset$; (Initialisation)
$A = \{\emptyset_0\}$; $N = \{0_0\}$; $O = \{\emptyset_0\}$; $W = \{\emptyset_0\}$; $n = 0$
$(\mathsf{let}\ (m\ (\mathsf{mutex}\ \top))\ (\mathsf{sync}\ \{m\}\ \emptyset\ ()))_0$

$\rightarrow$ $R^M = \{\top_1\}$; $R^R = \{(\top, \top)\}$ (LetMutex)
$(\mathsf{sync}\ \{1\}\ \emptyset\ ())_0$

$\rightarrow$ $Q = [0]$; $Q^W = [0]$; $R^T = \{(\top : \top)_0\}$; $W = \{\{1\}_0\}$ (WaitSR)
$(\mathsf{wait}\ \{1\}\ \emptyset\ ())_0$

$\rightarrow$ $Q^W = []$; $A = \{(\emptyset : \emptyset)_0\}$; $N = \{(1 : 0)_0\}$; (EnterNND)
$O = \{(\{1\} : \emptyset)_0\}$; $W = \{\emptyset_0\}$
$(\mathsf{insync}\ ())_0$

$\rightarrow$ $Q = []$; $R^T = \{\top_0\}$; $A = \{\emptyset_0\}$; $N = \{0_0\}$; $O = \{\emptyset_0\}$ (UnlockTL)
$()_0$

3.6 Bilan

Dans ce chapitre nous avons décrit une bibliothèque de mutex sans interblocage. Cette bibliothèque combine le mécanisme de Boudol09 d'évitement des interblocages par *préverrouillage* et un mécanisme de prévention des interblocages qui impose un ordre sur des ensembles de mutex appelés *régions*.

Cette bibliothèque nécessite une opération de verrouillage multiple, c'est-à-dire d'acquisition de n mutex simultanément. Nous avons décrit un algorithme de verrouillage multiple et prouvé qu'il était sans interblocage. Nous avons formalisé la combinaison des opération de haut niveau de notre bibliothèque et de notre algorithme de verrouillage multiple dans une sémantique opérationnelle.

Dans le chapitre suivant, nous décrirons quelques aspects de l'implémentation à proprement parler de notre bibliothèque et son utilisation dans des tests de performances.

Nous avons également introduit le concept problématique de *verrouillage asymptotique*, dont il reste à voir s'il aurait une pertinence dans d'autres contextes, et quels mécanismes existants ou nouveaux permettraient d'y remédier.

Chapitre 4

Implémentation et performances

Contents

Nous avons décrit dans le chapitre précédent le fonctionnement d'une bibliothèque Hop de mutex sans interblocage ni famine. Dans ce chapitre nous décrirons quelques éléments clés de notre implémentation (Section 4.1) puis présenterons des mesures de ses performances (Section 4.3).

Le développement de notre bibliothèque nous a conduit à implémenter une optimisation de la gestion des exceptions en Hop, que nous présenterons en Section 4.2. Cette optimisation pourrait être appliquée à d'autres langages de programmation structurée tels que Java.

4.1 Implémentation

4.1.1 Structures de données

Nous avons conçu nos algorithmes de verrouillage et déverrouillage pour être exécutés sous un verrou global, autrement dit sans accès concurrents aux structures de données.

Allocations mémoire Lors de tests de performances préliminaires nous avons constaté que la gestion mémoire (garbage collector) constituait une part importante du temps d'exécution de nos fonctions (plus de 30 %). Nous avons donc écrit ou réécrit nos fonctions de façon à minimiser le nombre d'allocations mémoire, en utilisant la mutabilité des structures de données, etc. Un exemple sera donné plus loin (p. 79).

4.1.1.1 Queues doublement chaînées

Pour représenter les queues de threads on crée une structure de données implémentée comme une liste doublement chaînée (type `dlq` pour *doubly linked queue*). On dira simplement une queue.

```
(class node
   (prev::node)
   (val read-only)
   (next::node)
   )
(class dlq
   (first::node)
   (last::node)
   )
```

Un thread dans une queue `q` pourra avoir un pointeur vers son nœud dans cette queue pour pouvoir la parcourir à partir de sa position, ou se retirer de `q` en temps constant.

```
(dlq-push::node  ::dlq ::obj) ; dlq-push prend une dlq et un objet
                              ; et retourne un nœud
```

4.1.1.2 Mutex et threads

Nous avons choisi de représenter l'ordre global sur les threads (Q dans la sémantique) par les structures suivantes :

- pour chaque mutex `m`, une queue contenant le propriétaire de `m` et les threads attendant `m` (s'ils existent) ;
- pour chaque thread, un entier relatif appelé le *rang* tel que l'ordre des rangs des threads corresponde à l'ordre des threads (comme dans Lamport76).

Nous avons choisi de maintenir une queue par mutex pour qu'un thread puisse parcourir ses dépendances sans parcourir totalement une queue globale pouvant contenir un nombre arbitrairement grand de mutex.

On maintient cependant une queue globale contenant tous les threads en attente appelée `waiting-threads` (Q^W dans la sémantique). Les threads sont simplement ajoutés à la fin de `waiting-threads` lorsqu'ils se mettent en attente et retirés lorsqu'ils entrent dans leur section critique. Cette queue n'est jamais réorganisée. Son premier élément, s'il existe, est le doyen.

Voici une version légèrement simplifiée des structures de données `mutex` et `jthread` de notre implémentation (pour le mode sûr) :

```
(class jmutex
   (id (default (make-mutex-id)) read-only)
   (region::region (default (instantiate::region))
      read-only )
   (locked::bool (default #f))
   (queue::dlq (default (instantiate::dlq)))
   (th-noop::pair-nil (default '()))
   (th-region::region (default region-top))
   (th-authorized (default #f))
   (th-authorized-id::int (default -1))
   (tmp::int (default -1))
   )
```

Chaque mutex a un identificateur numérique, une région, un champ indiquant son état verrouillé ou non et une queue de threads. Si un mutex `m` est verrouillé, sa queue est nécessairement non-vide et le premier élément de la queue est le propriétaire de `m`.

Chaque mutex a également un champ `tmp` qui lui permet d'être marqué par les algorithmes de parcours de dépendances, et 4 champs `th-*` dont l'utilité sera expliquée en Section 4.1.1.3.

```
(class jthread::pthread ; jthread hérite de pthread
   (condvar::condvar (default (make-condition-variable)) read-only)
   (region::region (default region-top))
   (authorized (default #f)) ; :: mutex list | #f
   (authorized-id::int (default -1))
   (rank::int (default -1))
   (owning::pair-nil (default '())) ; :: mutex list list
   (waiting::pair ; :: (mutex * node) list pair
      (default (cons '() '()))  read-only )
   (noop::pair-nil (default '()))
   (tmp::int (default -1))
   )
```

Nos algorithmes utilisent la possibilité qu'un thread s'endorme jusqu'à ce qu'il soit réveillé par un autre. En Hop, les pthreads n'offrent pas cette fonctionnalité par défaut. Nous l'avons donc implémenté en attribuant à chaque jthread une variable de condition `condvar` sur laquelle le thread peut attendre et que les autres threads peuvent signaler. Tous les threads utilisent pour attendre sur leur `condvar` le verrou global `big-lock`.

Les champs `region`, `authorized`, `authorized-id`, `owning` et `waiting` représentent respectivement :

- la région courante ;
- l'ensemble de mutex explicitement autorisés ;
- le dernier identificateur de mutex attribué lors de l'entrée dans la section critique courante ;
- les mutex possédés par le thread ;
- les mutex attendus par le thread.

(Ces champs correspondent respectivement à R^T, A, N, O et W dans la sémantique.)

`authorized` valant `#f` représente le fait que tous les mutex de la région courante sont autorisés.

La valeur par défaut de `authorized-id` doit être inférieure à tous les identificateurs de mutex possibles (attribués par la fonction `make-mutex-id`).

L'ensemble des mutex qu'un thread possède est représenté dans le champ `owning` par une pile de listes correspondant à l'emboîtement des sections critiques ; l'ensemble des mutex possédés est l'union des listes de la pile.

L'ensemble des mutex que le thread attend est représentée dans le champ `waiting` par la liste des mutex à verrouiller et la liste des mutex à préverrouiller. Chaque mutex est associé avec le nœud du thread dans la queue de ce mutex.

4.1.1.3 Passage de valeurs autour d'une section critique et champ `noop`

En Hop, un bloc `synchronize` n'est pas compilé en un appel à une fonction `synchronize`, mais en deux appels à respectivement `lock` et `unlock`, ce qui pose

le problème du nécessaire passage de valeurs de la partie `lock` à la partie `unlock` d'une exécution d'un bloc `synchronize`.

En particulier, les champs `region`, `authorized` et `authorized-id` d'un thread prennent une nouvelle valeur au début de chaque section critique et doivent être restaurés ensuite. Ces valeurs ne pouvant pas être passées comme de simples variables locales, elles pourraient être stockées dans des piles (comme dans la sémantique), mais cela impliquerait 3 `cons` (allocations mémoire) à chaque section critique.

Nous avons choisi de stocker ces valeurs dans des champs ad-hoc (notés `th-*` plus haut) du premier mutex acquis par chaque section critique. Ces valeurs sont ensuite récupérées par la fonction de déverrouillage qui reçoit ce mutex en paramètre.

Ces valeurs ne seront pas écrasées par un autre thread tant que le thread courant possèdera le mutex, c'est-à-dire jusqu'à la fin de la section critique, sauf si le mutex est relâché au cours de cette section critique, ce qui arrive en cas d'attente sur une variable de condition. Notre fonction `condition-variable-wait!` doit donc veiller à préserver ces champs du mutex qu'elle prend en paramètre.

Par ailleurs, nos mutex étant récursifs, il arrive qu'un bloc `synchronize` ne nécessite l'acquisition d'aucun mutex. Dans ce cas, parmi les 3 champs susmentionnés de la classe `jthread`, seul `authorized` change et doit être sauvegardé ; sa valeur doit donc être passée au `unlock` en plus de l'information qu'aucun nouveau mutex n'a été pris. C'est le rôle du champ `noop` (« no operation ») de la classe `jthread`, qui est une pile de sauvegardes du champ `authorized`.

Quand un thread n'a pas pour section critique courante une section de type *noop*, cette pile est vide. Quand sa section critique courante est de type noop, la hauteur de la pile correspond au nombre de sections noop directement imbriquées (sans sections non-noop entre elles) et les valeurs qu'elle contient sont les sauvegardes de `authorized` correspondantes.

Enfin, quand un thread est dans une section noop et entre dans une section critique imbriquée non-noop, son champ `noop` doit être réinitialisé à la pile vide et restauré à la fin de cette nouvelle section critique. C'est le rôle du quatrième champ `th-noop` de la classe `jmutex`.

4.1.2 Thread sortant des ascendances du doyen

Un thread t peut faire partie des ascendances du doyen (c'est-à-dire que $t \prec^* t_d$) puis en sortir ($t \not\prec^* t_d$). Cela se produit lorsque t relâche le dernier de ses mutex qui était attendu par une autre ascendance de t_d.

L'invariant (J) dicte alors que, si t ne sort pas de Q, il soit déplacé à droite de t_d dans Q.

Cependant, dans notre implémentation, nous avons choisi de tout simplement laisser un tel thread t à sa place. En effet, cela n'a pas d'impact sur les évolutions ultérieures de Q impliquant t avec la stratégie que nous avons implémenté :

— si une ascendance de t_d devient bloquée par t, t doit de toute façon être à nouveau à gauche de t_d ;

— si un thread à droite de t_d devient bloqué par t, il est donc aussi à droite de t qui n'a pas besoin d'être déplacé ;
— si t se met en attente de nouveaux mutex, comme il ne fait plus partie des ascendances de t_d, il est déplacé à la fin de la queue avec ses descendances.

4.1.3 Optimisation basique du préverrouillage

Dans l'implémentation de notre algorithme de verrouillage multiple, la fonction `lock-n` est conçue pour être utilisée comme implémentation du bloc `synchronize` défini en Section 3.2. Cela implique, dans le cas du verrouillage avec préverrouillage, d'acquérir certains mutex qui seront immédiatement relâchés.

Pour optimiser ce cas, notre fonction accepte donc 2 paramètres distincts : une liste de mutex à verrouiller, et une liste de mutex à préverrouiller.

L'optimisation la plus simple consiste, au moment où les mutex peuvent être pris effectivement, à ne pas effectuer sur les mutex à préverrouiller l'opération inutile de les prendre et de les relacher, et à la place, de seulement prendre les mutex à verrouiller.

D'autre part, au cours du processus d'attente, le thread courant n'a pas besoin d'attendre d'être le premier dans la queue des mutex à préverrouiller, il suffit pour qu'il puisse progresser qu'il soit premier dans la queue des mutex à verrouiller, et que les mutex à préverrouiller soient libre. En effet, c'est la condition suffisante à l'évitement des interblocage, et cela n'invalide pas la peuve d'absence de famine.

En revanche notre implémentation n'affecte pas de priorités différentes aux threads selon qu'ils doivent verrouiller ou préverrouiller leurs mutex, pour tenir compte de la plus faible occupation des ressources lors d'un préverrouillage. Une telle optimisation reste à développer.

4.1.4 Interface : les régions sont des symboles

Dans notre bibliothèque, la fonction de création de mutex prend un paramètre optionnel de région. En Hop, la fonction `make-mutex` existante prenait, conformément à la SRFI 18 [8], un paramètre optionnel : le nom du mutex.

Pour avoir la même interface que la fonction existante, notre fonction de création de mutex prend en paramètre le nom de sa région, c'est-à-dire un simple symbole, et la région du mutex sera la région correspondant au symbole dans une table de hashage globale si elle existe, ou une nouvelle sinon.

```
(define all-regions (make-hashtable))

(define (get-region name)
   (or (hashtable-get all-regions name)
       (let ((r (instantiate::region (name name))))
          (hashtable-put! all-regions name r)
          r )))

(define (j-make-jmutex::jmutex name)
   (if name
       (instantiate::jmutex (region (get-region name)))
       (instantiate::jmutex)
       ))
```

4.2 Exceptions : synchronization lifting et généralisation

Dans cette section on présente une optimisation du handler d'exception `unwind-protect` de Hop, équivalent d'un `try/finally` en Java.

Notre motivation vient du fait que les blocs `synchronize`, devant gérer les exceptions (avec des `unwind-protect`), étaient à l'origine significativement plus coûteux qu'une paire d'appels à `mutex-lock!` et `mutex-unlock!`. Cela désavantageait nos mutex, qui reposent sur le verrouillage structuré, par rapport aux mutex pthread, qui permettent le verrouillage explicite. La présente optimisation réduit presque totalement cette différence.

Nous présenterons d'abord l'optimisation restreinte aux seuls blocs `synchronize`, puis sa généralisation. Nous discuterons également de l'application de cette optimisation à Java.

Cette optimisation est indépendante de l'implémentation des mutex et donc relativement indépendante du reste de notre travail. Dans un soucis de simplicité, on ne considèrera dans cette section que des `synchronize` sur un seul mutex ; la généralisation au `synchronize` sur n mutex est relativement triviale.

L'optimisation restreinte aux blocs `synchronize` a été publiée dans [44].

4.2.1 Verrouillage structuré et exceptions

Lorsqu'un thread exécute un bloc synchronisé, le mutex concerné est acquis avant que le thread n'entre dans le corps du bloc, et doit être relâché lorsque le thread en sort, que le corps se termine normalement ou lance une exception. (Les autres mécanismes de rupture du flot de contrôle, tels que `return` en Java, seront assimilés à des exceptions.)

L'implémentation « naturelle » d'un bloc synchronisé consiste donc à installer un handler d'exceptions qui relâche le mutex dans tous les cas.

La JVM est la plateforme la plus commune à offrir du verrouillage structuré. Au niveau du bytecode [35], un bloc synchronisé est implémenté comme suit :

```
monitorEnter( m );
try {
  ...
} finally {
  monitorExit( m );
}
```

Rappelons le principe des handlers d'exceptions en Java :

- Un bloc `try` contient un corps à exécuter ;
- des clauses `catch`, optionnelles, définissent le rattrapage (ou non) d'une exception lancée dans le corps en fonction de son type ;
- une clause `finally` est exécutée dans tous les cas.

Le code Hop équivalent est le suivant :

```
(mutex-lock! m)
(unwind-protect
   ; try
   ( ... )
   ; finally
   (mutex-unlock! m) )
```

4.2.2 Implémentation habituelle en Hop/C

Installer un handler d'exceptions est une opération pratiquement gratuite en Java. Lancer une exception est comparativement plus coûteux. En Hop au contraire, c'est l'installation d'un handler qui est l'opération la plus coûteuse, ce qui est une conséquence de la stratégie de compilation de Hop vers C utilisée. Il est difficile d'implémenter efficacement les exceptions en code C portable car `setjmp`/`longjmp`, le mécanisme de bas niveau offert par C pour gérer les discontinuités du flot d'exécution, s'y prête mal. La fonction `setjmp` sauvegarde les arguments de la fonction courante, et potentiellement les handlers de signaux, sur la pile. Cette opération est lente, mais n'est responsable que d'une partie de l'impact sur les performances de `setjmp`. L'autre partie est dûe à la sauvegarde des variables temporaires qui ne sont pas préservées par `setjmp` lui-même. Pour les sauvegarder, Hop introduit une fonction supplémentaire dont les paramètres sont les variables libres du handler (partie « finally »). Enfin, parmi ces variables libres, celles qui sont mutées sont empaquetées dans des références.

Illustrons cette complexité en montrant la compilation de la fonction Hop suivante :

```
(define (foo z)
   (let ((x 1) (y 2))
      (set! y 4)
      (unwind-protect
         ...
         (bar (+ x y z)) )))
```

La fonction Hop `foo` est compilée en une fonction C qui prend un argument entier. Les variables `x` et `y` sont compilées en variables temporaires C, mais puisque `y` est muté et utilisé dans le handler, il est empaqueté à l'aide d'une fonction `MAKE_BOX`. Le handler utilise `x`, `y` et `z`, qui doivent donc être sauvegardés par `setjmp`. Puisque `setjmp` ne sauvegarde que les arguments de la fonction courante, on crée une fonction auxiliaire `__try53` qui prend les trois variables libres comme paramètres. Le code C est le suivant :

```
obj_t __try53( int x, obj_t y, int z ) {
   jmp_buf env;
   if( setjmp( env ) == 0 )
      ...
   else
      return bar( x + BOX_REF( y ) + z );
}

function foo( int z ) {
   int x = 1;
   obj_t y = MAKE_BOX( 2 );

   CELL_SET( y, 4 );
   __try53( x, y, z );
   ...
}
```

Ce code C est lent, en particulier comparé à l'implémentation des `try` en Java. Mais les blocs `synchronize` peuvent en fait être exécutés sans handler d'exceptions tout en gardant la même sémantique, comme nous allons le voir maintenant.

4.2.3 Synchronization lifting

La spécification des blocs `synchronize` repose implicitement sur un handler d'exceptions qui relâche le mutex. Ce handler est nécessaire mais ne doit pas obligatoirement se trouver au lieu (syntaxique) du bloc `synchronize`. Le déverrouillage du mutex peut être retardée jusqu'à la prochaine clause `catch` ou `finally` qui sera activée par l'exception. Sur la base de cette observation, nous pouvons utiliser un nouveau schéma de compilation pour les blocs synchronisés, dont les principe est illustré en Figure 4.1. Pour un bloc synchronisé donné, au lieu d'installer un handler d'exceptions dédié sur la pile, le dernier handler est amendé de façon à relâcher le mutex s'il est activé par une exception. On appelle cette technique le *synchronization lifting* .

Dans Hop, la nouvelle séquence pour un bloc `synchronize` est la suivante :

```
(mutex-lock! m)
(let ((hdl (get-current-exception-handler)))
   (handler-push-mutex! hdl m)
   (let ((res ...))
      (handler-pop-mutex! hdl)
      (mutex-unlock! m)
      res ))
```

Notons qu'il existe toujours un handler d'exceptions auquel se rattacher : un handler par défaut est installé au début de l'exécution de chaque thread et est activé en cas d'exception non rattrapée. Il est le handler courant en dehors de tout bloc `try`.

Deux autres composants de l'environnement d'exécution doivent également être modifiés :

— la structure de données des handlers d'exceptions est modifiée pour l'implémentation des fonctions `handler-push-mutex!` et `handler-pop-mutex!` ;
— l'environnement d'exécution est changé de façon à relâcher tous les mutex qui ont été empilés dans un handler d'exceptions avant d'exécuter ce dernier.

Évolution de la pile d'exécution Figure 4.1 :

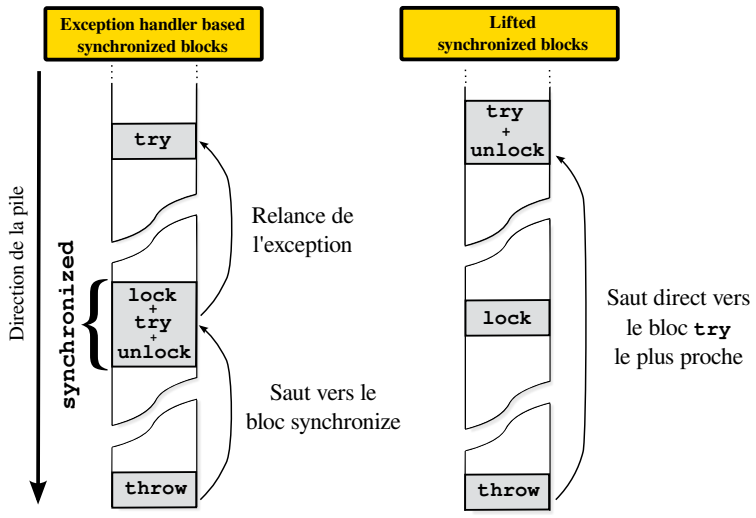


FIGURE 4.1 – Pile d'exécution lors d'une exception dans un bloc synchronisé, sans et avec synchronization lifting. À gauche, le schéma habituel : un bloc synchronisé met sont propre handler d'exception sur la pile ; en cas d'exception, le handler relâche le mutex et relance l'exception. À droite, le synchronization lifting : à l'entrée d'un bloc synchronisé, le mutex pris est empilé dans le handler d'exception courant (préexistant) ; lorsque l'exception est lancée, elle invoque directement ce handler.

Performances attendues Avec ce nouveau mécanisme, entrer dans un bloc synchronisé ne fait qu'empiler un mutex dans le handler d'exceptions courant, ce qui, nous allons le voir, revient en fait à écrire une valeur sur la pile d'exécution. Sortir du bloc synchronisé efface cette valeur. Lancer une exception déverrouille tous les mutex empilés dans le handler courant. Ce mécanisme est donc, en l'absence d'exception, pratiquement aussi efficace que l'implémentation Java à base de `try`, et plus rapide que Java dans le cas où une exception est lancée car il évite de dérouler une chaîne de clauses finally.

4.2.4 Implémentation dans Hop

Nous avons implémenté le synchronization lifting dans Hop (Bigloo $\geqslant$ `4.0a`).

Un bloc synchronisé lifté exécute 3 opérations de plus que son équivalent en verrouillage explicite :

1. il récupère le handler d'exceptions courant ;
2. il empile le mutex verrouillé ;
3. il dépile le mutex lors de son déverrouillage.

On peut rendre efficaces ces deux dernières opérations en pré-allouant l'espace requis dans le handler d'exceptions. La récupération du handler d'exceptions courant peut être implémenté comme la lecture d'une variable globale ou, quand l'environnement d'exécution le supporte, comme la lecture d'un registre du thread courant. Avec cette implémentation, le coût de ces trois opérations est à peine visible.

4.2.4.1 Cache

Pour savoir combien d'emplacements de mutex pré-allouer dans chaque handler d'exception, nous avons mesuré le nombre de mutex devant être empilés dans chaque handler. Notre expérimentation a consisté à étudier une application multimédia « réaliste », implémentée en Hop, qui utilise des mutex dans de nombreux contextes différents.

Nous avons constaté l'exécution de 15 873 blocs synchronisés. Pour 15 720 de ces blocs (99 %), le mutex verrouillé était le premier à être empilé dans le handler d'exception ; pour les 153 blocs restants (1 %) le mutex verrouillé était le second dans le handler d'exceptions. Nous n'avons observé aucune situation où plus de 2 mutex devaient être empilés dans un handler d'exception.

Par conséquent, notre implémentation du synchronization lifting dans Hop utilise un cache de 2 mutex. Si plus de 2 mutex doivent être empilés, une liste est allouée sur le tas pour stocker les mutex supplémentaires. Voici le code C correspondant, quelque peu simplifié dans un souci de lisibilité :

```
struct exc_handler {
   struct exc_handler *prev;
   /* ... */
   obj_t mutex0;
   obj_t mutex1;
   obj_t mutexN;
};

#define HOP_HANDLER_PUSH_MUTEX( exch, m ) \
   exch.mutex0 == HOP_FALSE ? exch.mutex0 = m : \
   exch.mutex1 == HOP_FALSE ? exch.mutex1 = m : \
      exch.mutexN = make_pair( m, exch.mutexN )

#define HOP_HANDLER_POP_MUTEX( exch ) \
   exch.mutex1 == HOP_FALSE ? \
      exch.mutex0 = HOP_FALSE : \
      NULLP( exch.mutexN ) ? \
         exch.mutex1 = HOP_FALSE : \
         exch.mutexN = CDR( exch.mutexN )
```

4.2.4.2 Analyse statique

Pour éliminer une partie du petit surcoût engendré par l'empilement et le dépilement des mutex, Hop met en œuvre une analyse statique simple qui détecte une partie des blocs `synchronize` *sans échec*, c'est-à-dire qui ne lancent pas d'exceptions.

Il s'agit d'une analyse d'effets naïve par module qui traverse l'arbre de syntaxe abstraite du corps de chaque `synchronize` pour voir si un lancement d'exception ou un appel à une fonction inconnue peut y être atteint. Appliquée à tout le code source de l'environnement d'exécution, cette analyse détecte 72 blocs `synchronize` sans échec parmi 264, c'est-à-dire 27 %.

Parmi les 15 873 blocs synchronisés exécutés au cours de l'expérimentation décrite plus haut (application multimédia), 1 828 (11 %) sont sans échec et évitent les opérations d'empilement et dépilement.

Cette analyse pourrait être remplacée par d'autres plus sophistiquées qui ont été développées pour Java [40, 16] ; ces optimisations et le synchronization lifting sont complémentaires.

4.2.5 Applicabilité à Java

Il nous semble que les implémentation actuelles de JVM, telles que la JVM Oracle version `6u26-1`, pourraient bénéficier du synchronization lifting. Voici pourquoi.

Les blocs synchronisés y sont implémentés comme de simples blocs `try/finally`. Quand une exception est lancée, les clauses `finally` chaînées sont exécutées, et chacune doit

1. relâcher les mutex qui doivent l'être ;
2. relancer l'exception.

Nous avons mesuré l'impact de ces sauts intermédiaires sur les performances globales de Java. Pour cela, nous avons écrit un programme Java qui exécute une boucle dans un thread. Chaque itération de la boucle exécute un bloc `try`. Selon la version du test, ce bloc synchronise plusieurs mutex pour appeler une fonction séparée (versions **sync**) ou appelle directement cette fonction sans synchronisation (versions **unsync**). D'autre part, soit cette fonction retourne directement (versions **return**), soit elle lance une exception (versions **throw**). Voici les 4 temps d'exécution obtenus :

	sync	**unsync**
return	2,21 s	0,63 s
throw	66,89 s	50,71 s

La comparaison des temps d'exécution **return/sync** et **return/unsync** montre que le coût des blocs `synchronized` est d'environ $2,21 - 0,63 = 1,58$ s. La différence entre **throw/sync** et **throw/unsync** est de 16,18 s. Cette durée, moins le temps nécessaire à l'acquisition des mutex, soit $16,18 - 1,58 = 14,60$ s, est le temps pris pour dérouler la pile des clauses `finally` lorsque l'exception est lancée. L'application du synchronization lifting à Java permettrait d'éliminer ce coût.

Le compilateur à la volée (JIT) de Java pourrait lifter la clause `finally` d'un bloc `try` lorsque celle-ci ne fait qu'appeler la fonction `monitorExit`. Cela améliorerait les performances des blocs `synchronized` quand des exceptions sont lancées, sans changer la vitesse d'exécution dans les autres cas.

4.2.6 Performances

Cette section présente une évaluation des performances du synchronization lifting, indépendamment du reste de notre travail. Dans ce but, on compare la vitesse d'exécution de trois programmes simples, chacun écrit dans trois langages : Hop (pthread, compilation vers C), C (pthread) et Java. Avec Hop, on comparera les performances avec et sans synchronization lifting. Les résultats de cette expérimentation ne doivent être lus que comme une validation de ce que nous avons affirmé précédemment et non comme une comparaison des performances des trois langages, ce qui est un exercice risqué.

Chacun de nos tests exécute un certain nombre de fois une courte séquence d'instructions :

1. Le premier test (**sync**) acquiert 2 mutex et appelle une fonction qui retourne immédiatement.
2. Le deuxième (**throw**) appelle une fonction qui, une fois par exécution, lance une exception. En C, lancer une exception est simulé par le fait de retourner un code d'erreur de fonction en fonction.
3. Le troisième (**signal**) acquiert un mutex et signale une variable de condition.

Les implémentations de ces tests sont triviales ; on ne les donnera pas ici. Les temps d'exécution de ces tests sont présentés en Table 4.1 :

	Hop	Hop/Lifting	GCC	Java
sync	38,70 s	4,06 s	4,08 s	3,38 s
throw	37,59 s	4,24 s	4,09 s	3,33 s
signal	19,56 s	3,68 s	3,56 s	5,39 s

TABLE 4.1 – Impact du synchronization lifting sur trois tests simples. Les temps rapportés sont obtenus en additionnant les temps CPU et système. Chaque valeur rapportée est la plus petite observée sur trois exécutions consécutives.

La différence la plus frappante est entre Hop (sans lifting) et Hop/Lifting. L'accélération découle de la disparition des séquences complexes à base de `setjmp`. Notre seconde observation est que Hop/Lifting et GCC ont des performances semblables. La différence entre **sync** et **throw** montre le coût d'empilement et de dépilement des mutex.

Notre dernière observation est que Java est à la fois significativement plus rapide que Hop et GCC pour les deux premiers benchmarks et plus lent pour le dernier. Cela nous semble montrer que Java sacrifie les performances de signalisation des variables de condition au profit des performances d'acquisition des mutex, reflétant le style de programmation concurrente de Java. Cela pourrait également indiquer que l'acquisition de mutex dans le cas le plus rapide est plus efficace en Java qu'avec la bibliothèque pthread. Nous n'avons pas cherché à faire cette comparaison.

4.2.7 Généralisation

En Java comme en Hop, la compilation d'un bloc synchronisé génère a priori un hander d'exception dont la seule fonction est de relâcher (dans le `finally`) les mutex acquis au début du bloc. Le synchronization lifting tel que nous l'avons défini revient à rattacher cette clause `finally` au handler d'exceptions courant précédemment installé, évitant ainsi d'en installer un supplémentaire.

Cette optimisation est permise par la certitude que ledit handler courant sera exécuté dans tous les cas où l'hypothétique handler généré pour le bloc synchronisé l'aurait été. On peut donc généraliser cette optimisation à tous les handlers d'exceptions qui ne perturbent pas le flot d'exécution lorsqu'une exception est lancée, la laissant forcément passer.

En Java, ces handlers particuliers sont les blocs `try`/`finally` sans clauses `catch`. En Hop, il s'agit de tous les blocs `unwind-protect`.

4.2.7.1 Dans Hop

Cette généralisation est implémentée dans Hop et remplace le synchronization lifting depuis Bigloo `4.0b` : un `unwind-protect` n'installe pas de handler d'exception mais empile la clause « `finally` », sous forme d'une fermeture, dans le handler d'exception courant.

Une optimisation supplémentaire est alors possible. Elle consiste, lorsqu'un `unwind-protect` est utilisé dans l'implémentation d'une opération courante, à ne pas empiler dans le handler d'exception une fermeture, mais un autre objet dont le traitement lors du dépilement sera déterminé par le type. Cela permet d'économiser le coût d'allocation et de désallocation d'une fermeture.

En particulier, un bloc `synchronize` est compilé en un `unwind-protect` qui n'empile pas une fermeture mais directement un mutex sur la même pile. En cas d'exception, lors du dépilement, si un objet dépilé n'est pas une fermeture mais un mutex, il est déverrouillé.

4.3 Performances

Dans cette section on présente des mesures des performances de notre bibliothèque comparées à celles de pthread, dans plusieurs micro-benchmarks et deux « vrais » programmes. Les résultats sont les suivants :

— pour ce qui est des performances brutes de verrouillage et déverrouillage avec et sans contention, les mutex jthread sont 10 à 15 fois plus lents que les mutex pthread ;
— dans un autre micro-benchmark, le programme utilisant jthread est légèrement plus rapide car l'évitement des interblocages permet d'utiliser un algorithme plus efficace, qui mène à des interblocages lorsqu'implémenté avec pthread ;
— en remplaçant pthread par jthread dans le serveur Web Hop et dans une application multimédia, nous n'avons pas observé de différence significative dans les performances.

Sauf mention contraire, nos tests de performance ont été effectués sur un système à noyau Linux `3.12` sur un processeur Intel Xeon E5-1660, avec la glibc `2.19-3` (qui inclut la bibliothèque C pthread) et Bigloo `4.1a`.

4.3.1 Micro-benchmarks

4.3.1.1 Performances brutes de verrouillage/déverrouillage

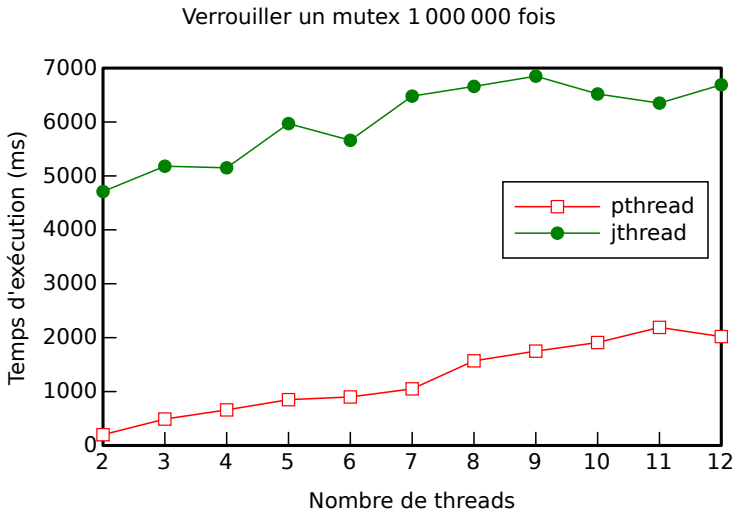
Nous avons écrit deux micro-benchmarks dans lesquels respectivement un ou plusieurs threads acquièrent un unique mutex pour effectuer une opération en virgule flottante élémentaires.

On rapporte les temps obtenus en additionnant les temps « CPU » et « système ».

Sans contention On rapporte la plus petite valeur parmi 2 exécutions consécutives.

	sync
pthread	1,06 s
jthread	13,23 s

Avec contention



On constate que pour ce qui est des performances brutes de verrouillage, notre bibliothèque est environ 12 fois plus lente que pthread sans contention, en entre 3 et 50 fois plus lente avec contention.

4.3.1.2 Benchmark avec évitement des interblocages

Ce benchmark est un petit programme d'une centaine de lignes. Dans ce benchmark, `n` threads accèdent à `p` mutex. Chacun des threads exécute de façon répétitive une tâche. Une tâche consiste à choisir aléatoirement 2 mutex `m1` et `m2` puis à effectuer un « travail » en 2 étapes, en devant posséder `m2` pour la seconde étape, et `m1` pour toute la durée des 2 étapes :

```
(synchronize m1
   (work)
   (synchronize m2
      (work)
      ))
```

Dans notre implémentation le « travail » consiste en un certain nombre d'opérations arithmétiques.

Avec la bibliothèque pthread le programme ci-dessus compile mais mène à des interblocages, par exemple lorsqu'un thread choisit des mutex `mi` et `mj` et qu'un autre choisit `mj` et `mi` (les même, inversés).

Pour ne pas avoir d'interblocages, il convient de supposer un ordre sur les mutex et de ne pas imbriquer les `synchronize` dans l'ordre inverse :

```
(if (< m1 m2)
    ; then
    (synchronize m1
       (work)
       (synchronize m2
          (work)
          ))
    ; else
    (synchronize m2
       (synchronize m1
          (work)  ; requiert m1
          (work)  ; requiert m1 et m2
          )))
```

Avec la bibliothèque jthread le premier programme mène à des cycles sur les régions et donc à des erreurs à l'exécution. Pour ne pas avoir d'erreurs, il faut assigner les `p` mutex à la même région, puis mettre les `prelock`s adéquats :

```
(synchronize m1 :prelock m2
   (work)
   (synchronize m2
      (work)
      ))
```

Résultats Nous avons exécuté le programme avec 4 threads et 4 mutex dans le but d'observer un fort taux de contention. On rapporte le temps réel d'exécution minimal parmi 3 exécutions consécutives.

	sync
pthread	9,35 s
jthread	8,34 s

Ce micro-benchmark est un exemple de cas dans lequel notre bibliothèque autorise plus de parallélisme, en verrouillant `m2` plus tardivement dans les cas où `m2` < `m1`. Dans cet exemple, le gain en parallélisme compense le coût intrinsèque de nos fonctions, rendant le programme qui utilise notre bibliothèque (légèrement) plus rapide.

4.3.2 Applications réelles

4.3.2.1 Serveur Web Hop [1]

L'environnement d'exécution du langage Hop consiste en un serveur Web à part entière, équipé de compilateurs qui compilent dynamiquement en HTML

1. Ce benchmark a été réalisé avec une version antérieure de l'algorithme de verrouillage multiple, dont nous attendons des performances sensiblement équivalentes à celles de la version la plus récente pour ce benchmark.

et JavaScript les programmes Hop destinés au côté client. Le serveur Web est optimisé pour le temps de réponse de contenu dynamique. Hop est bootstrappé. Son code source compte 75 000 lignes de code. Nous avons utilisé le serveur Web Hop pour mesurer l'impact de la bibliothèque jthread sur un logiciel de type serveur.

Hop est multi-threadé pour le traitement de multiples requêtes en parallèle. Hop utilise les pthreads. Hop utilise des mutex, par exemple pour protéger l'accès aux divers caches utilisés pour éviter de recompiler du code destiné au côté client.

Pour comparer les performances des deux versions de Hop, nous avons suivi la méthodologie traditionnelle de mesure des performances de serveurs Web. Celle-ci consiste une quantité variable de requêtes vers un serveur et à mesurer le nombre de réponsees qu'il peut fournir. Les requêtes sont générées par l'outil dédié httperf [39]. Nous avons effectués les tests sur une machine avec un Intel Xeon W3570 sous Linux `3.9`.

Nous avons comparé les performances des deux versions du serveur sur deux types de requêtes : statiques et dynamiques. Les résultats sont présentés en Figures 4.2 et 4.3.

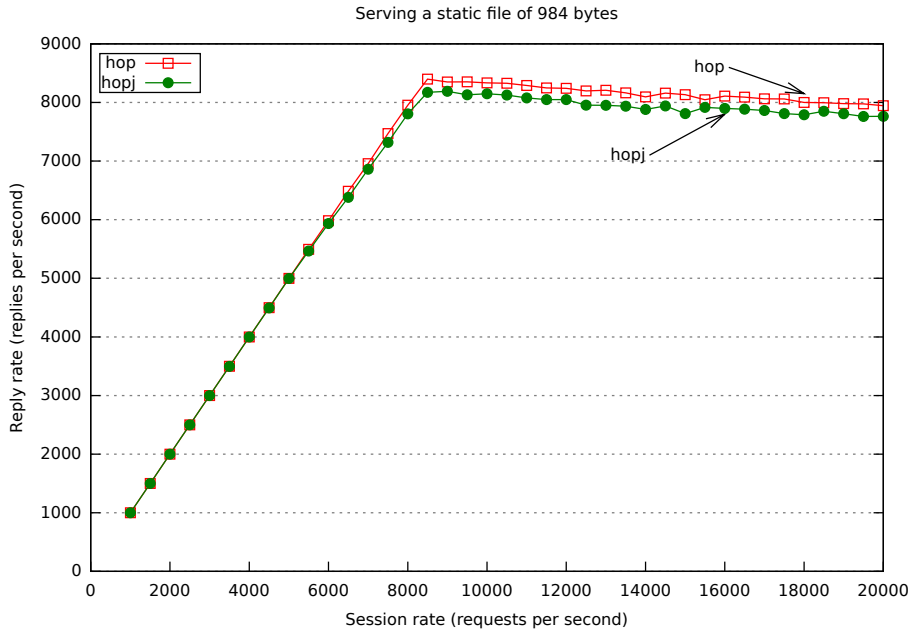


FIGURE 4.2 – Débit soutenu par Hop servant des fichiers statiques de 984 octets. Chaque session consiste en 5 requêtes consécutives envoyées en utilisant une seule connexion persistante.

Cette expérimentation ne montre pas d'impact significatif de la bibliothèque jthread sur les performances globales de Hop ; les deux versions du serveur se comportent de façons presque identiques.

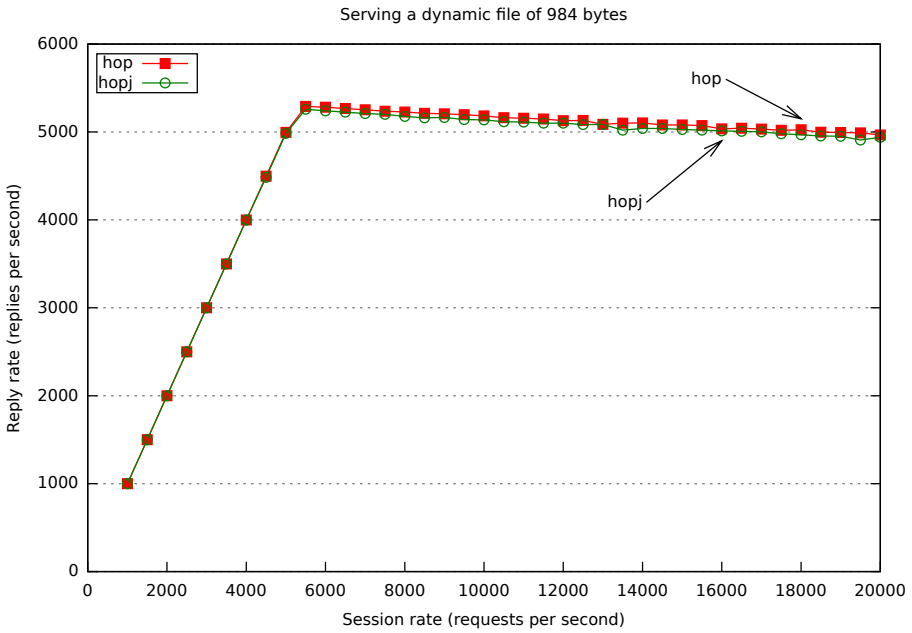


FIGURE 4.3 – Débit soutenu par Hop servant du contenu dynamique. Chaque session consiste en 5 requêtes consécutives envoyées en utilisant une seule connexion persistante.

4.3.2.2 Décodeur MP3

Notre seconde expérimentation consiste à estimer l'impact de la bibliothèque jthread sur les performances d'un décodeur de musique MP3 implémenté en Hop. Ce benchmark est une application Hop d'environ 1 300 lignes de code. Il est extrait d'une application multimédia plus importante conçue pour fonctionner sur des machines ARM bas de gamme. Pour un décodeur MP3, il est d'importance critique d'être suffisamment rapide ; le code source est donc très optimisé pour la vitesse d'exécution. En particulier, l'implémentation du parallélisme a été réalisée avec soin, et le nombre d'acquisitions de mutex minimisé autant que possible.

Le temps passé dans le système audio matériel n'étant pas pertinent pour la présente expérimentation, l'application a été modifiée pour ignorer les octets décodés au lieu de les jouer.

Le décodeur MP3 implémente un algorithme traditionnel de producteur-consommateur. Le producteur lit les octets d'un fichier de musique MP3 depuis le système de fichiers local ou sur le réseau. Le consommateur les décode en octets de son brut. Les deux threads sont synchronisés au moyen de mutex et de variables de conditions.

Nous avons fait tourner le décodeur audio sur un gros fichier MP3 de 60 minutes de musique. Environ 18 700 mutex furent verrouillés au cours de l'exécution.

Les chiffres rapportés ici sont le minimum de temps `cpu+sys` de 3 exécutions

consécutives.

	décodeur MP3
pthread	3,72 s
jthread	3,94 s

Les chiffres montrent que, dans le cas de la présente application concurrente réaliste comme dans le cas du serveur Hop, notre bibliothèque n'a pas d'impact sur les performances globales, contrairement à ce que nous avons observé sur les micro-benchmarks.

4.3.3 Benchmarks complexes

Nous n'avons pas trouvé de programme qui soit fréquemment utilisé pour mesurer les performances d'implémentations de mutex et qui comporte des imbrications complexes de sections critiques, ce qui permettrait une comparaison significative de différentes stratégies.

Par exemple, parmi SPECjvm2008 [45], les Java Grande Forum Multi-threaded Benchmarks [47], et les tests de performances du projet JSR 166 [4], plusieurs programmes utilisent le verrouillage structuré mais sans sections critiques imbriquées. Dans le Computer Language Benchmarks Game (anciennement Debian Shootout) [3], deux programmes utilisent des mutex : dans `chameneos-redux`, il n'y a pas de sections critiques imbriquées ; dans `thread-ring`, l'utilisation qui est faite des mutex ne correspond pas à du verrouillage structuré.

4.4 Bilan

Dans ce chapitre nous avons utilisé la bibliothèque jthread dans plusieurs tests de performances. Nous avons observé des performances de verrouillage brutes médiocres, mais un impact négligeable de l'utilisation de notre bibliothèque sur les performances de deux applications réelles. Nous avons également donné un exemple simple de programme dans lequel notre bibliothèque permet plus de parallélisme (c'est-à-dire, un algorithme plus efficace) et de meilleures performances que la bibliothèque pthread. Nous n'avons en revanche pas trouvé de benchmark qui nous permette de comparer finement les performances de plusieurs versions de notre bibliothèque.

Nous avons vu que notre opération de verrouillage, qui demande dans le cas général des informations supplémentaires au programmeur, pouvait être utilisée dans les deux applications susmentionnées sans les modifier.

Nous avons par ailleurs présenté une optimisation de la gestion des exceptions dans Hop, qui permet notamment une implémentation efficace du verrouillage structuré. Nous avons vu que cette optimisation pourrait être appliquée à d'autres langages et notamment à Java.

Chapitre 5

Conclusion

Nous avons travaillé à la conception et à l'implémentation d'un langage de programmation généraliste concurrent et modulaire qui offre des mutex sans interblocage. Nous avons pris pour point de départ un travail antérieur par Boudol [Boudol09] mettant en œuvre une inférence d'effets et une sémantique avec préverrouillage pour garantir un évitement des interblocages.

Dans un premier temps, nous avons ébauché une extension de l'inférence d'effets de Boudol09 à un langage de la famille Caml. Nous avons opté pour une association mutex-valeur déterminée lors de l'inférence de types et d'effets par le biais de *régions*, et avons identifié plusieurs inconvénients de cette approche.

Dans un second temps, nous nous sommes donné pour buts d'implémenter, sous forme d'une bibliothèque Hop, la sémantique avec préverrouillage de Boudol09, et de la rendre utilisable sans analyse statique préalable. Pour cela, nous avons combiné le mécanisme de préverrouillage de Boudol09 avec la méthode classique de prévention des interblocages qui consiste à ordonner les mutex. Nous avons obtenu des mutex et une opération d'acquisition qui garantit l'absence d'interblocages, en partie en soulevant des erreurs, qui sont plus faciles à déboguer que de vrais interblocages, à l'instar du typage dynamique, et en partie en évitant activement les interblocages, permettant ainsi d'exprimer des algorithmes efficaces qui, autrement, seraient susceptibles de générer des interblocages.

Dans le cas général, nos opérations sur les mutex nécessitent de la part du programmeur plus d'informations que leurs contreparties traditionnelles. Cependant, nous avons défini des comportements par défaut qui réduisent ce besoin, et avons ainsi pu utiliser notre bibliothèque dans deux applications existantes sans aucune modification de ces dernières.

Notre bibliothèque offre également des variables de conditions, mais sans la garantie d'absence d'interblocage offerte sur les mutex.

Pour implémenter l'opération d'acquisition avec préverrouillage de notre bibliothèque, nous avons proposé un algorithme de verrouillage multiple sans famine. Nous avons alors montré que bien que notre bibliothèque ait de mauvaises performances de verrouillage brutes comparé à pthread, utiliser jthread au lieu de pthread a eu un impact minime sur les performances de deux applications réelles.

Notre algorithme de verrouillage multiple pourrait recevoir des améliorations incrémentales ou être entièrement remplacé : l'interface et les opérations de haut de niveau de notre bibliothèque pourraient reposer sur un autre algorithme de verrouillage multiple.

D'autre part, notre travail pourrait être porté vers d'autres langages utilisant la programmation structurée et le verrouillage structuré, Java par exemple.

La spécification de la propriété d'absence de famine de notre bibliothèque nous a conduit à définir la notion d'*interblocage asymptotique*. Il reste à voir si cette notion aurait une pertinence dans d'autres contextes, et quels mécanismes existants ou nouveaux permettraient d'empêcher les interblocages asymptotiques.

Enfin, nous avons développé dans Hop une optimisation de la gestion des exceptions. Celle-ci pourrait être appliquée à d'autres langages de programmation structurée ; nous avons montré qu'elle permettrait par exemple d'améliorer les performances des blocs synchronisés en Java.

Index

Références

[1] Bigloo. URL http://www-sop.inria.fr/indes/fp/Bigloo/.

[2] Caml. URL http://caml.inria.fr/.

[3] The Computer Language Benchmarks Game (formerly Debian Shootout). URL https://benchmarksgame.alioth.debian.org/.

[4] Concurrency JSR-166 Interest Site. URL http://gee.cs.oswego.edu/dl/concurrency-interest/.

[5] Hop. URL http://hop.inria.fr/.

[6] Llama. URL https://web.archive.org/web/20101001184624/http://llamalabs.org/.

[7] OCaml. URL https://ocaml.org/.

[8] SRFI 18: Multithreading support. URL http://srfi.schemers.org/srfi-18/srfi-18.html.

[9] Agrawal, Rakesh, Carey, Michael J and McVoy, Larry W. 1987. The performance of alternative strategies for dealing with deadlocks in database management systems. In *IEEE Transactions on Software Engineering*, (12), pp. 1348–1363. URL http://www.minds.wisconsin.edu/bitstream/handle/1793/58618/TR590.pdf.

[10] Belik, Ference. 1990. An efficient deadlock avoidance technique. In *Computers, IEEE Transactions on*, vol. 39(7), pp. 882–888. URL http://ieeexplore.ieee.org/ielx5/12/2013/00055690.pdf?tp=&arnumber=55690&isnumber=2013.

[11] Berry, Gérard and Boudol, Gérard. 1989. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pp. 81–94. ACM. URL https://hal.inria.fr/inria-00075426/document.

[12] Boudol, Gérard. 2009. A deadlock-free semantics for shared memory concurrency. In *Theoretical Aspects of Computing-ICTAC 2009*, pp. 140–154. Springer. URL http://link.springer.com/chapter/10.1007/978-3-642-03466-4_9.

[13] Boudol, Gérard, Luo, Zhengqin, Rezk, Tamara and Serrano, Manuel. 2010. Towards reasoning for web applications: an operational semantics for hop. In *Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, pp. 3–14. ACM. URL http://dl.acm.org/citation.cfm?id=1810141.

[14] Boyapati, Chandrasekhar, Lee, Robert and Rinard, Martin. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, vol. 37, pp. 211–230. ACM. URL http://dl.acm.org/citation.cfm?id=582440.

[15] Brinch Hansen, Per. 1973. *Operating system principles*. Prentice-Hall, Inc. URL http://dl.acm.org/citation.cfm?id=540365.

[16] Choi, Jong-Deok, Gupta, Manish, Serrano, Mauricio J., Sreedhar, Vugranam C. and Midkiff, Samuel P. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25(6), pp. 876–910. URL http://dl.acm.org/citation.cfm?id=945892.

[17] Coffman, Edward G., Elphick, Melanie and Shoshani, Arie. 1971. System deadlocks. In *ACM Computing Surveys (CSUR)*, vol. 3(2), pp. 67–78. URL http://dl.acm.org/citation.cfm?id=356588.

[18] Dijkstra, Edsger W. 1964. Een algorithme ter voorkoming van de dodelijke omarming. URL http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[19] Dijkstra, Edsger W. 1982. The mathematics behind the Banker's algorithm. In *Selected Writings on Computing: A Personal Perspective*, pp. 308–312. Springer.

[20] Domaigné, Loïc. Apr. 2010. Condvars: signal with mutex locked or not? URL http://www.domaigne.com/blog/computing/condvars-signal-with-mutex-locked-or-not/.

[21] Flanagan, Cormac and Abadi, Martín. 1999. Types for safe locking. In *Programming Languages and Systems*, pp. 91–108. Springer. URL http://link.springer.com/chapter/10.1007/3-540-49099-X_7.

[22] Flynn, Michael. 1972. Some computer organizations and their effectiveness. In *IEEE Transactions on Computers*, vol. 100(9), pp. 948–960. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5009071.

[23] Fonseca, Pedro, Li, Cheng, Singhal, Vishal and Rodrigues, Rodrigo. 2010. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 221–230. IEEE. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5544315.

[24] Garrigue, Jacques. Apr. 2004. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*. Springer-Verlag. URL http://link.springer.com/chapter/10.1007/978-3-540-24754-8_15.

[25] Gerakios, Prodromos, Papaspyrou, Nikolaos and Sagonas, Konstantinos. Oct. 2011. A Type System for Unstructured Locking that Guarantees Deadlock Freedom without Imposing a Lock Ordering. In *Electronic Proceedings in Theoretical Computer Science*, vol. 69, pp. 44–58. ISSN 2075-2180. doi:10.4204/EPTCS.69.4. URL http://arxiv.org/abs/1110.4160v1.

[26] Grande, Johan, Boudol, Gérard and Serrano, Manuel. Jul. 2015. Jthread, a deadlock-free mutex library. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*, pp. 149–160. Siena, Italy. URL http://dl.acm.org/citation.cfm?id=2790523.

[27] Harris, Tim, Marlow, Simon, Peyton-Jones, Simon and Herlihy, Maurice. 2005. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 48–60. ACM. URL http://dl.acm.org/citation.cfm?id=1065952.

[28] Hoare, Charles Antony Richard. 1974. Monitors: An operating system structuring concept. In *Communications of the ACM*, vol. 17(10), pp. 549–557. URL http://dl.acm.org/citation.cfm?id=361161.

[29] Jula, Horatiu, Tözün, Pinar and Candea, George. 2011. Communix: A framework for collaborative deadlock immunity. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pp. 181–188. IEEE. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958217.

[30] Jula, Horatiu, Tralamazza, Daniel M., Zamfir, Cristian and Candea, George. 2008. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, pp. 295–308. URL https://www.usenix.org/legacy/event/osdi08/tech/full_papers/jula/jula_html/.

[31] Kobayashi, Naoki. 2006. A new type system for deadlock-free processes. In *CONCUR 2006–Concurrency Theory*, pp. 233–247. Springer. URL http://link.springer.com/chapter/10.1007/11817949_16.

[32] Lamport, Leslie. 1974. A new solution of Dijkstra's concurrent programming problem. In *Communications of the ACM*, vol. 17(8), pp. 453–455. URL http://dl.acm.org/citation.cfm?id=361093.

[33] Lamport, Leslie. 1976. The synchronization of independent processes. In *Acta Informatica*, vol. 7(1), pp. 15–34. URL http://link.springer.com/article/10.1007/BF00265219.

[34] Larus, James and Kozyrakis, Christos. Jul. 2008. Transactional memory. In *Communications of the ACM*, vol. 51(7), p. 80. ISSN 00010782. doi:10.1145/1364782.1364800. URL http://portal.acm.org/citation.cfm?doid=1364782.1364800.

[35] Lindholm, Tim and Yellin, Frank. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

[36] Lindholm, Tim, Yellin, Frank, Bracha, Gilad and Buckley, Alex. 2014. *The Java virtual machine specification*. Pearson Education.

[37] Lu, Shan, Park, Soyeon, Seo, Eunsoo and Zhou, Yuanyuan. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, vol. 43, pp. 329–339. ACM. URL http://dl.acm.org/citation.cfm?id=1346323.

[38] McCloskey, Bill, Zhou, Feng, Gay, David and Brewer, Eric. 2006. Autolocker: synchronization inference for atomic sections. In *ACM SIGPLAN Notices*, vol. 41(1), pp. 346–358. URL http://www.zhoufeng.net/eng/papers/autolocker-popl06.pdf.

[39] Mosberger, David and Jin, Tai. 1998. httperf—a tool for measuring web server performance. In *ACM SIGMETRICS Performance Evaluation Review*, vol. 26(3), pp. 31–37. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.679&rep=rep1&type=pdf.

[40] Ruf, Erik. 2000. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'00)*, pp. 208–218. ACM, New York, NY, USA. ISBN 1-58113-199-2. doi:10.1145/349299.349327. URL http://doi.acm.org/10.1145/349299.349327.

[41] Schindewolf, Martin, Cohen, Albert, Karl, Wolfgang, Marongiu, Andrea and Benini, Luca. 2009. Towards transactional memory support for gcc. In *1st GCC Research Opportunities Workshop*. URL https://hal.inria.fr/hal-00645337/.

[42] Seltzer, Margo, Chen, Peter and Ousterhout, John. 1990. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Technical Conference*, pp. 313–323. Washington, DC. URL https://www.usenix.org/publications/library/proceedings/seltzer3.pdf.

[43] Serrano, Manuel, Gallesio, Erick and Loitsch, Florian. 2006. Hop: a language for programming the web 2.0. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pp. 975–985. URL https://www.lri.fr/~conchon/TER/2012/3/dls06.pdf.

[44] Serrano, Manuel and Grande, Johan. 2014. Locking Fast. In *Symposium on Applied Computing*. Gyeongju, Korea. URL https://www-sop.inria.fr/members/Manuel.Serrano/publi/sg-sac14.pdf.

[45] Shiv, Kumar, Chow, Kingsum, Wang, Yanping and Petrochenko, Dmitry. 2009. SPECjvm2008 performance characterization. In *Computer Performance Evaluation and Benchmarking*, pp. 17–35. Springer. URL http://link.springer.com/chapter/10.1007/978-3-540-93799-9_2.

[46] Silberschatz, Abraham and Galvin, Peter Baer. 1994. *Operating system concepts*. Addison-Wesley Reading.

[47] Smith, Lorna A. and Bull, J. Mark. 2001. A Multithreaded Java Grande Benchmark Suite. In *Proceedings of the third workshop on Java for high performance computing*. URL http://www2.epcc.ed.ac.uk/~markb/docs/jhpcthread.ps.gz.

[48] Song, Xiang, Chen, Haibo and Zang, Binyu. 2010. Why software hangs and what can be done with it. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 311–316. IEEE. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5544304.

[49] Suenaga, Kohei. 2008. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Programming Languages and Systems*, pp. 155–170. Springer.

[50] Tai, Kuo-Chung. 1994. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *1994 International Conference on Parallel Processing*, vol. 2, pp. 69–72.

[51] Tofte, Mads and Birkedal, Lars. 1998. A region inference algorithm. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20(4), pp. 724–767. URL http://dl.acm.org/citation.cfm?id=291894.

[52] Tofte, Mads and Talpin, Jean-Pierre. 1997. Region-based memory management. In *Information and Computation*, vol. 132(2), pp. 109–176.

[53] Vasconcelos, Vasco T., Martins, Francisco and Cogumbreiro, Tiago. Feb. 2010. Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language. In *Electronic Proceedings in Theoretical Computer Science*, vol. 17, pp. 95–109. ISSN 2075-2180. doi: 10.4204/EPTCS.17.8. URL http://arxiv.org/abs/1002.0942v2.

[54] Vytiniotis, Dimitrios, Peyton Jones, Simon and Schrijvers, Tom. 2010. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pp. 39–50. ACM. URL http://dl.acm.org/citation.cfm?id=1708023.

[55] Wang, Yin, Kelly, Terence, Kudlur, Manjunath, Lafortune, Stéphane and Mahlke, Scott A. 2008. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, pp. 281–294. URL http://static.usenix.org/legacy/events/osdi08/tech/full_papers/wang/wang_html/.

[56] Wu, Hui, Chin, Wei-Ngan and Jaffar, Joxan. 2002. An efficient distributed deadlock avoidance algorithm for the AND model. In *IEEE Transactions on Software Engineering*, vol. 28(1), pp. 18–29. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=979987.

[57] Zeng, Fancong and Martin, Richard P. 2004. Ghost locks: Deadlock prevention for Java. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*. URL http://www.cs.rutgers.edu/~fzeng/masplas04.ps.