UNIVERSITÉ DU QUÉBEC À MONTRÉAL

EXTENSIONS PARALLÈLES POUR LE LANGAGE NIT

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SYLVAIN POIRIER

JUIN 2013

REMERCIEMENTS

Mes remerciements vont aux Professeurs Guy Tremblay et Jean Privat de l'Université du Québec à Montréal qui m'ont permis de réaliser ce travail. Qu'ils trouvent dans cet ouvrage un témoignage de ma profonde reconnaissance.

Je présente ma gratitude à tous ceux qui ont pu contribuer de près ou de loin à l'aboutissement de ce travail.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Grâce à la miniaturisation et à l'arrivée des processeurs multicoeurs dans les ordinateurs personnels, la programmation parallèle ne semble plus être réservée aux groupes restreints d'utilisateurs ayant accès à des superordinateurs ou à des grappes de calculs. De plus, avec la tendance actuelle d'augmenter constamment le nombre de coeurs des processeurs, le besoin de développer des applications pouvant s'exécuter correctement sur l'ensemble des unités de traitement disponibles est un enjeu déterminant. Sous de telles conditions, tout langage de programmation récent devrait être doté de mécanismes permettant la génération et l'exploitation de calculs parallèles. Dans le présent mémoire, nous explorons la possibilité d'intégrer des extensions parallèles à un langage orienté objet récent qui, au départ, ne possédait aucun support pour la programmation parallèle.

Le langage utilisé est Nit, un langage développé conjointement à l'UQAM et au LIRMM (Montpellier, France). Nit est un langage moderne orienté objet dont la mise en oeuvre utilise de nouvelles techniques de compilation, à la fine pointe de la recherche sur les langages orientés objet. À ce langage de base séquentiel, nous avons ajouté un module, appelé NitSpace, qui permet d'échanger de l'information entre des processus qui s'exécutent en parallèle. Ce module est fondé sur le modèle Linda proposé par Carriero et Gelernter (2). Le modèle Linda propose de partager, entre les processus, un espace de tuples par l'intermédiaire duquel les divers processus échangent de l'information, et ce à l'aide de quatre opérations principales : `out`, `in`, `rd` et `eval`. À ces quatre opérations s'ajoutent aussi deux variantes non bloquantes des opérations de lecture : `inp` et `rdp`. Ces diverses opérations permettent l'ajout, l'extraction et la lecture dans un espace partagé de tuples. Un avantage du modèle Linda est sa simplicité, mais en permettant malgré tout la définition des structures de données plus complexes telles que sémaphores, barrières de synchronisation, sacs de tâches, tableaux distribués, flux de données, etc.

Le modèle Linda, défini au départ dans le contexte du langage C, a par la suite été intégré à d'autres langages, qui ont introduit des variantes au modèle original. Quelques-unes de ces variantes seront présentées, en présentant les implémentations qui ont été réalisées dans les langages Ruby, Python, Scala et Java. Nous avons choisi de présenter ces langages car, tout comme Nit, il s'agit de langages orientés objets. Les caractéristiques de ces différentes mises en oeuvre du modèle Linda ont aussi influencé le choix de conception et de réalisation de notre module NitSpace. L'implémentation de ce module fut réalisée en trois couches que l'on présentera en détail. La majeure partie représentant la couche de bas niveau fut réalisée en C. À celle-ci s'ajoute une couche médiane qui permet de relier la couche C à la couche Nit. Finalement, la couche supérieure, écrite entièrement en Nit, représente l'API qui permet à un programme Nit de générer des calculs parallèles.

Mots-clés : Nit, NitSpace, programmation parallèle, Linda, espace de tuples.

INTRODUCTION

Nit est un langage développé conjointement à l'UQAM et au LIRMM [6]. Comme de nombreux langages récents, Nit est un langage orienté objet où *tout est objet*. En plus d'intégrer les concepts de base que tout langage objet se doit de posséder, Nit se distingue par l'intégration de plusieurs autres concepts avancés tels que l'héritage multiple, la généricité bornée et la redéfinition de classes ou de modules. Nit est un langage fortement typé, mais avec un typage *adaptatif*, c'est-à-dire que le type statique d'une variable peut être ajusté selon son utilisation.

Nit est un langage moderne qui applique de nouvelles techniques de compilation à la fine pointe de la recherche sur les langages orientés objet. Nit ne propose toutefois aucune instruction ou aucun mécanisme pouvant exploiter les nouvelles architectures parallèles. La tendance actuelle est de proposer de nouveaux PC équipés de processeurs multicoeurs, comportant 4, 8 et parfois même 16 unités de traitement. Lorsqu'on regarde vers où se dirigent les grands joueurs de l'industrie, comme Intel et AMD, on constate que la tendance est de multiplier le nombre de coeurs disponibles afin d'augmenter la puissance totale de traitement. Il ne faut pas non plus oublier la tendance vers les processeurs multicoeurs hétérogènes, combinant CPU et GPU (*Graphical Processing Unit*) sur une même puce, où le GPU peut maintenant contribuer au traitement et être utilisé pour du parallélisme de données. Pour profiter de ces nouvelles évolutions, il est essentiel que les langages offrent des mécanismes permettant d'écrire des programmes qui exploitent la capacité des processeurs multicoeurs. Le but de notre recherche est d'ajouter au langage Nit une telle capacité de traitements parallèles pouvant s'exécuter sur de telles architectures.

Pour y arriver, nous avons d'abord exploré les langages en cours de développement dans le cadre du projet «*High Productivity Computing Systems*» de la DARPA[1], notamment le langage Chapel [9] développé par Cray et le langage X10 [12] développé par IBM. Ces langages, construits sur un modèle d'espace d'adressage global partagé et partitionné (PGAS = *Partitioned Global Address Space*), amènent de nouvelles façons de développer les programmes parallèles où le

[1]La DARPA (*Defense Advanced Research Projects Agency*) est une agence du département de la défense des États-Unis chargée de la recherche et du développement de nouvelles technologies.

programmeur doit concevoir différemment les algorithmes. Bien que ces langages fournissent des mécanismes qui facilitent l'exploitation des architectures parallèles, on réalise rapidement que ce sont des langages où le parallélisme est omniprésent. En d'autres mots, ces langages sont d'abord et avant tout conçus autour de la programmation parallèle pour ensuite intégrer les concepts de la programmation orientée objet. À l'inverse, Nit est d'abord conçu autour de la programmation orientée objet. L'intégration d'instructions parallèles de haut niveau, telles que celles disponibles en Chapel ou X10, nous aurait demandé de réécrire plusieurs portions centrales du compilateur Nit. Nous avons donc choisi de nous tourner vers une approche plus indépendante du noyau du langage, plus «orthogonale».

L'approche que nous avons retenue est fondée sur les travaux de Nicholas Carriero et David Gelernter [2] qui ont proposé un modèle de programmation parallèle appelé Linda. Ce modèle propose un moyen simple de structurer l'information dans des multi-ensembles de valeurs appelées des *tuples*, et permet à des processus indépendants d'échanger ces tuples par l'intermédiaire d'un espace partagé appelé *espace de tuples*. Le modèle Linda se compose d'un petit nombre d'opérations qui permettent à un programme d'interagir avec un espace de tuples. Tel que décrit par Carriero et Gelernter, l'ajout de ce modèle à un langage de programmation séquentiel de base permet d'obtenir un langage de programmation parallèle [1].

L'implémentation du modèle Linda dans Nit fut réalisée sous forme d'une bibliothèque. Un module indépendant, appelé NitSpace, peut être importé dans les programmes Nit afin d'ajouter les instructions nécessaires aux langages pour reproduire le modèle de base de Linda. Pour comprendre comment ce module fut réalisé, il est important de savoir que lors de la compilation, les fichiers sources Nit sont traduits en C avant d'être compilés vers du code machine pour produire les fichiers exécutables. Cette transition vers du code C permet une plus grande portabilité puisque des compilateurs C sont disponibles sur la plupart des plates-formes. Mais outre la portabilité, ce mécanisme offre aussi la possibilité d'écrire du code C indépendant du compilateur Nit. C'est ce qui nous a permis d'implémenter la gestion de l'espace de tuples utilisé par le module NitSpace directement en C. De plus, le compilateur Nit est doté de mécanismes qui permettent de générer des interfaces entre le code C et le code Nit [10]. Ce support pour la génération d'interfaces entre le code C et le code Nit a grandement facilité l'intégration de la bibliothèque avec le module NitSpace.

Dans la suite de ce mémoire, nous allons présenter de quelle façon l'implémentation du module NitSpace fut réalisée. Le chapitre 1 explique la base du modèle Linda tel qu'il

fut initialement présenté par Carriero et Gelernter. Le chapitre 2 propose quelques exemples d'implémentation dans des langages orientés objet afin de voir quelques variantes existantes du modèle original; plus précisément, nous explorerons les langages Ruby, Python, Scala et Java. Le chapitre 3 explique l'API du module NitSpace; nous présenterons comment inclure notre modèle dans des programmes Nit. Finalement, le chapitre 4 explique en détail le fonctionnement des trois couches de l'implémentation du module NitSpace : la couche Nit, la couche médiane et la couche C.

CHAPITRE I

LINDA

Linda n'est pas un langage de programmation, mais plutôt un modèle de communication et de coordination entre processus qui s'exécutent en parallèle. C'est un modèle qui fait abstraction du langage sous lequel il est défini et de l'architecture sur laquelle il s'exécute.

Le module NitSpace, qui a été réalisé, est une implémentation Nit de ce modèle. Il existe plusieurs autres implémentations qui ont introduit certaines variantes au modèle original. Cependant, ce chapitre explique la base du modèle Linda[1] tel qu'il fut présenté par Carriero et Gelernter [2].

1.1 Espace de tuples

Dans un modèle Linda, la mémoire, appelée espace de tuples, est une mémoire associative partagée via laquelle des processus distincts peuvent communiquer, conserver des données et coordonner leurs actions. Plus précisément, un espace de tuples emmagasine des tuples.

Un tuple est une liste de données typées et de longueur arbitraire. Dans un système basé sur un espace de tuples, les processus sont des producteurs et des consommateurs de ces tuples. Un exemple d'un espace de tuples est illustré à la figure 1.1.

Dans un espace de tuples, les processus n'interagissent pas directement entre eux, au contraire, toutes communications s'effectuent via l'espace de tuples. Cette particularité permet aux processus d'être découplés dans le temps. Plus précisément, en raison de la persistance de l'espace de tuples, les processus ne doivent pas nécessairement exister au même moment pour

[1] Par souci de simplicité, Linda désigne à la fois le modèle et l'implémentation C-Linda de ce modèle.

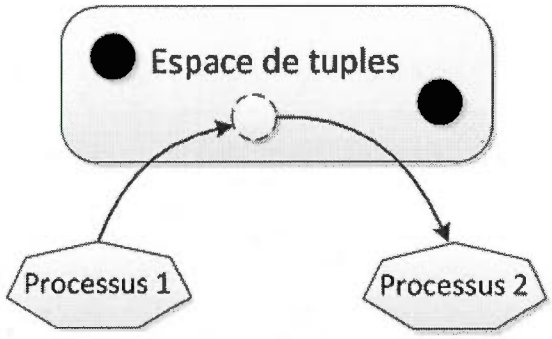

Figure 1.1 Exemple d'un système qui utilise un espace de tuples.

communiquer. Il est possible pour un processus de placer un tuple dans l'espace de tuples, puis de terminer, et qu'un autre processus, nouvellement créé, vienne consommer le tuple à un moment ultérieur.

1.2 Tuple et modèle de tuple

Il existe deux types de tuples, les tuples de processus et les tuples de données. Les tuples de processus sont des tuples actifs en cours d'évaluation, tandis que les tuples de données sont passifs — leurs valeurs sont définitives.

Un tuple de données, ou tout simplement un tuple, est la représentation d'une suite ordonnée de valeurs. Le nombre d'éléments dans un tuple est fixé et le type de ses champs est déterminé. Un tuple se distinguera alors par le nombre de champs qu'il comporte, le type de ses champs et la valeur qu'ils contiennent.

Par exemple, soit le tuple de données suivant :

```
("abc", 3.5)
```

Ce tuple contient deux champs. Le premier champ est une chaîne de caractères de valeur `"abc"`, et le second champ un nombre réel de valeur `3.5`. Lorsqu'un des champs du tuple est défini à l'aide d'une fonction qui nécessite une évaluation pour obtenir sa valeur, le tuple est appelé un tuple de processus. Les tuples de processus s'exécutent en parallèle, ils génèrent des échanges de données en consommant et produisant des tuples de données. Une fois que le tuple de processus termine son exécution, il se transforme en tuple de données et il est par la suite

impossible de distinguer, dans l'espace de tuples, le tuple processus des autres tuples de données.

Voici un exemple d'un tuple de processus pouvant être ajouté dans l'espace de tuples :

```
("a1", cos(60))
```

Le processus à l'origine de l'ajout du tuple poursuit son exécution dès que la demande est effectuée sur l'espace de tuples. L'évaluation de la fonction `cos(60)` s'effectue dans un processus parallèle. Pendant son évaluation, le tuple de processus n'est pas disponible pour une lecture ou une extraction par un autre processus. Une fois l'évaluation terminée, le tuple de processus devient un tuple de données en remplaçant la fonction par la valeur produite. Ainsi, le tuple de processus devient le tuple de données suivant :

```
("a1", 0,5)
```

Puisque son évaluation est terminée, le tuple de données produit est à présent visible dans l'espace de tuples et il est disponible pour une lecture ou une extraction par les autres processus.

Contrairement à la plupart des autres systèmes parallèles à mémoire partagée, les tuples stockés dans l'espace de tuples ne sont pas explicitement adressés ; ils sont plutôt accessibles via une recherche associative. Ce mécanisme de recherche fait appel à un modèle de tuple pour retrouver des correspondances dans l'espace de tuples. Un modèle de tuple est similaire à un tuple à l'exception que l'on retrouve, pour certains de ses champs, des indications de types au lieu de valeurs. Ces indicateurs sont les paramètres formels d'un modèle de tuple, par opposition aux paramètres réels contenant une valeur. Puisqu'il y a absence de valeurs, les paramètres formels agissent comme «*joker*» et ils peuvent ainsi avoir une correspondance valide avec toutes valeurs contenues au champ correspondant d'un tuple.

Un paramètre formel est préfixé d'un point d'interrogation, par exemple :

```
("une chaine", ?f, ?i, x)
```

Le premier et dernier champ sont des paramètres réels, tandis que les deux paramètres au centre sont des paramètres formels. Pour qu'un modèle et un tuple obtiennent une correspondance valide, ils doivent remplir les trois conditions suivantes :

1. Le tuple possède exactement le même nombre d'éléments que le modèle.

2. Les paramètres réels du modèle contiennent exactement la même valeur que ceux du tuple pour les champs respectifs.

3. Les types des champs du tuple sont de mêmes types que ceux spécifiés par les paramètres formels du modèle pour les champs respectifs.

À titre d'exemple, sachant que la variable `I` est de type `int`, considérons le modèle de tuple suivant :

```
("abc", ?I)
```

Ce modèle correspondrait au tuple `("abc", 5)` puisqu'il répond aux trois conditions énumérées précédemment. Par contre, il ne correspondrait pas avec le tuple `("xyz", 5)` parce que la valeur du premier champ est différente. Ou encore, il n'y aurait aucune correspondance possible avec le tuple `("abc", 5, 1.2)` parce qu'il y a inégalité sur le nombre de champs.

1.3 Opérations Linda

L'espace de tuples qui est fourni par un système Linda est un *multi-ensemble*. Puisqu'il peut exister plusieurs tuples identiques simultanément dans le même espace de tuples, on parle bien d'une collection de tuples et non pas d'un ensemble. Pour interagir avec cet espace, Linda fournit quatre opérations de base qui permettent de manipuler les tuples dans l'espace de tuples : `out()`, `in()`, `rd()` et `eval()`. À ces opérations s'ajoutent deux variantes `inp()` et `rdp()`.

1.3.1 Opérations d'ajout — `out(t)` et `eval(t)`

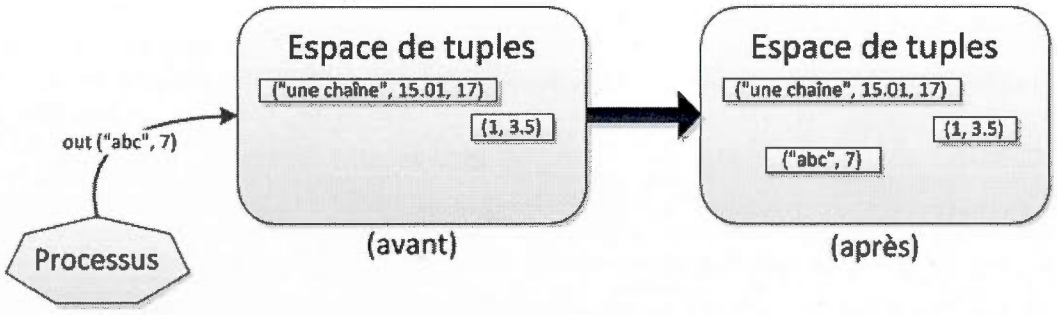

Figure 1.2 Exemple du traitement de l'opération `out()` sur un espace de tuples.

L'opération `out(t)`, telle qu'illustrée à la figure 1.2, permet d'ajouter le tuple `t` dans l'espace de tuples. Le processus appelant poursuit son exécution immédiatement après la demande.

L'opération `eval(t)` est une opération similaire à `out(t)` sauf que le tuple `t` est évalué juste avant qu'il soit ajouté dans l'espace de tuples. Une évaluation est faite pour chaque terme du tuple et chacune de ces évaluations est effectuée par un nouveau processus qui s'exécute en parallèle. La création de nouveaux processus est faite implicitement à l'appel de l'opération `eval`, et dès les nouveaux processus créés, le processus appelant poursuit immédiatement son exécution. Lorsque tous les termes du tuple ont été complètement évalués, le tuple `t` devient un tuple de données ordinaire qui peut être lu ou extrait comme n'importe quel autre tuple disponible dans l'espace de tuples.

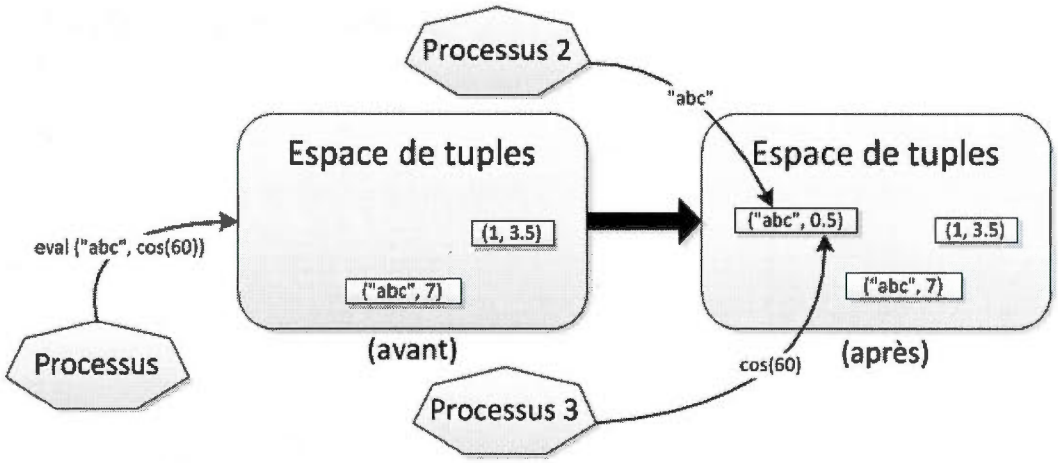

Figure 1.3 Exemple du traitement de l'opération `eval()` sur un espace de tuples.

Dans l'exemple de la figure 1.3, l'exécution de l'opération `eval("abc", cos(60))`, génère deux nouveaux processus, un pour chaque terme à évaluer. L'évaluation du premier terme est triviale puisque le champ contient déjà la valeur résultante `"abc"`. L'évaluation du second terme exigera un calcul par le processus 3 afin d'obtenir la valeur résultante `0.5`.

Bien qu'elle soit incluse dans le modèle original, l'opération `eval` est rarement présente dans les différentes implémentations de Linda que l'on peut trouver. Tout comme dans la réalisation de notre module NitSpace, la plupart des autres implémentations favorisent un mécanisme de création de processus plus traditionnel qui est influencé par le langage hôte de l'implémentation.

1.3.2 Opérations d'extraction — in(m) et inp(m)

L'opération `in(m)` permet d'extraire un tuple de l'espace de tuples. Le tuple retourné doit correspondre avec le modèle de tuple `m` qui a été demandé. Si aucun tuple correspondant au modèle `m` n'existe, alors l'opération bloque jusqu'à ce qu'un tuple avec une correspondance soit placé dans l'espace de tuples.

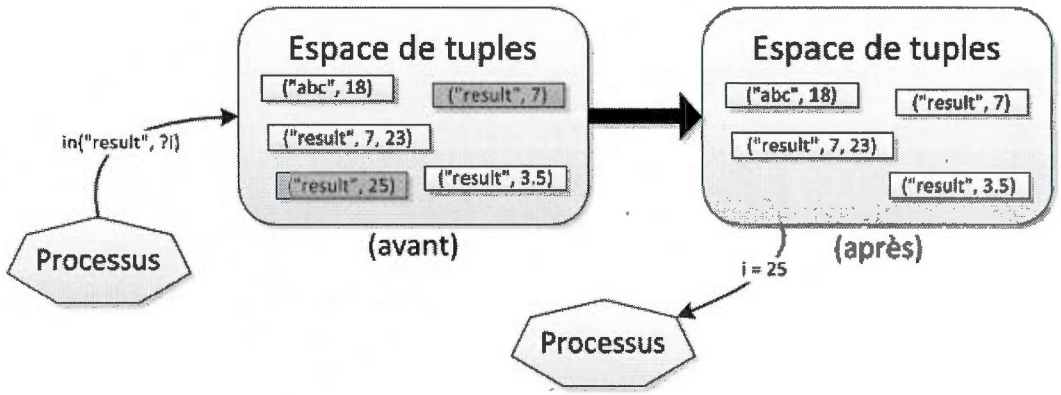

Figure 1.4 Exemple du traitement de l'opération `in()` sur un espace de tuples.

Dans l'exemple de la figure 1.4, l'exécution de l'opération `in("result", ?i)` entraine la recherche, dans l'espace de tuples, d'un tuple à deux éléments dont le premier est la chaîne de caractères `"result"` et dont le second est un élément du même type que la variable `i`, c'est-à-dire d'un élément de type `int`. Dans cet exemple, deux tuples, apparaissant sur fond grisé, ont une correspondance avec le modèle de tuple demandé. Lorsqu'il y a plusieurs correspondances disponibles, l'un des tuples est choisi arbitrairement. Une fois la sélection effectuée, le tuple est retiré de l'espace de tuples et ses valeurs réelles sont assignées aux paramètres formels du modèle de tuple. Dans ce cas-ci, c'est la valeur 25 qui est affectée à la variable `i`.

L'opération `inp(m)` est une version alternative de `in()`. Cette opération se comporte comme l'opération `in()`. Mais à l'inverse de cette dernière, elle ne bloque pas si aucun tuple correspondant au modèle n'est disponible dans l'espace de tuples. En cas d'absence de correspondance, l'opération retourne la valeur `0`, sinon elle retourne `1`.

1.3.3 Opérations de lecture — rd(m) et rdp(m)

L'opération `rd(m)`, telle qu'illustrée à la figure 1.5, est semblable à l'opération `in()` sauf que le tuple correspondant n'est pas supprimé. Les paramètres réels sont assignés aux paramètres formels comme avec l'opération `in()`, mais le tuple demeure dans l'espace de tuples.

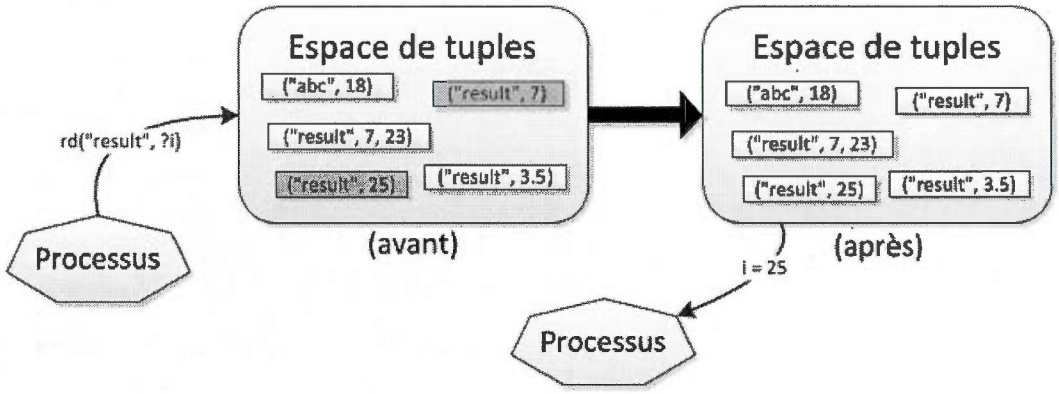

Figure 1.5 Exemple du traitement de l'opération `rd()` sur un espace de tuples.

L'opération `rdp(m)` est une version alternative de `rd()`. Cette opération tente de lire un tuple correspondant au modèle `m` et retourne 0 en cas d'échec, sinon elle retourne 1 ainsi que le tuple approprié.

1.4 Structures de données distribuées

Dans un système Linda, une structure de données distribuée est une structure de données dont les valeurs stockées sont réparties dans l'espace de tuples en encapsulant chaque élément dans un tuple. L'exemple le plus classique est un tableau où le premier champ du tuple indique le nom du tableau, le champ suivant indique l'index de l'élément et le troisième sa valeur. Outre les tableaux, il est possible de conserver, dans un espace de tuples, plusieurs autres types de structures dont quelques-unes, présentées ici, ont été introduites par Carriero et Gelernter [2].

1.4.1 Sémaphores

Dans le modèle Linda, un sémaphore généralisé peut être représenté par une collection d'éléments identiques. Ainsi, pour initialiser la valeur du sémaphore `sem` à `n`, il suffit d'exécuter l'opération `out("sem")` `n` fois. Pour acquérir une ressource, l'équivalent d'une opération `P`,

il faut exécuter l'opération d'extraction `in("sem")`. Finalement, pour libérer une ressource, équivalent à exécuter une opération V, il suffit d'utiliser l'opération d'ajout `out("sem")`.

1.4.2 Sacs de tâches

Un sac est une structure de données qui définit deux opérations: ajouter un élément et retirer un élément. Les sacs sont particulièrement importants pour la programmation parallèle, puisque c'est un type de structure qui permet de produire des algorithmes de style coordonnateur/travailleurs. Dans ce type d'algorithme, le coordonnateur ajoute des tâches dans un sac et les processus travailleurs les retirent pour obtenir une tâche à effectuer, c'est-à-dire un traitement qu'ils doivent effectuer. Ainsi, au lieu de spécifier des éléments de données, les tuples précisent les tâches qui doivent être accomplies.

Pour ajouter une tâche, on utilise l'opération d'ajout :

```
out( "tache", descriptionTache )
```

Les tâches pourront ensuite être retirées à l'aide d'une opération d'extraction :

```
in( "tache", ?nouvelleTache );
```

Une fois complété, le résultat de la tâche peut être ajouté dans l'espace de tuples afin d'être récupéré par le coordonnateur. Les tâches accomplies par chaque processus peuvent être différentes ou identiques, en fonction de la spécification de la tâche donnée dans le tuple.

1.4.3 Accès à une structure nommée

Les applications parallèles nécessitent souvent l'accès à des éléments distribués que l'on peut distinguer par leurs noms. Avec un espace de tuples, il est possible de conserver chacun de ces éléments à l'aide d'un tuple de la forme suivante : `(Nom, valeur)`. Pour lire une de ces valeurs, on utilise simplement une opération de lecture :

```
rd( nom, ?valeur );
```

Pour mettre à jour l'une de ces valeurs, on utilise une extraction, puis un ajout de la façon suivante :

```
in( nom, ?valeur );
out( nom, nouvelleValeur );
```

Tout processus qui tente de lire la valeur d'un élément pendant qu'il est mis à jour bloquera jusqu'à ce que la mise à jour soit terminée et que le tuple soit disponible à nouveau dans l'espace de tuples.

1.4.4 Barrières de synchronisation

Occasionnellement, un processus doit attendre qu'un évènement se produise, ou qu'un traitement soit complété avant de poursuivre son exécution. Certaines applications parallèles s'appuient sur des barrières pour synchroniser leurs processus. Une barrière de synchronisation est un point commun où chaque processus, dans un groupe donné, doit attendre que les autres processus du groupe aient atteint ce même point avant de poursuivre son exécution.

Avec un modèle Linda, une barrière peut être réalisée à l'aide d'un tuple qui associe le nom de la barrière avec le nombre de processus qui ne l'ont pas encore atteinte. Le nombre de processus impliqués est utilisé pour initialiser la barrière. Par exemple, si le groupe contient `n` processus, la barrière appelée `barrière01` pourra être mise en place en exécutant l'opération d'ajout du tuple suivant :

```
out( "barrière01", n );
```

Par la suite, lorsqu'un processus atteint la barrière, il décrémente la valeur contenue dans le tuple associé, puis attend jusqu'à ce que sa valeur soit égale à 0. La synchronisation par l'intermédiaire de la barrière entre les processus du groupe pourra être réalisée en ajoutant ces quelques instructions :

```
in( "barrière01", ?val );
out( "barrière01", val-1 );
rd( "barrière01", 0 );
```

1.4.5 Tableaux distribués

Les tableaux distribués sont constitués d'une série de tuples où chaque tuple représente un élément du tableau. Chacun de ces tuples contient comme premier champ le nom du tableau, ensuite l'indice ou les indices du tableau, et finalement, la valeur de l'élément du tableau qui y est stocké. Ainsi, le tuple `("V", 12, 123.4)` définit l'élément de la douzième position du vecteur `V`. Le tuple `("M", 10, 12, 3, 123.4)` contient un élément de la matrice `M` à trois dimensions, et ainsi de suite.

De cette façon, si l'on veut stocker dans l'espace de tuples le tableau `A` à deux dimensions comme une structure de données distribuées, on utiliserait une série de tuples tels que les suivants :

```
("A", 1,1, elem1), ("A", 1,2, elem2)
("A", 2,1, elem3), ("A", 2,2, elem4)
...
```

Par la suite, si on veut retrouver une copie de l'un des éléments de `A`, par exemple l'élément du tableau `A[2][2]`, on doit utiliser un modèle de tuple tel que le suivant :

```
rd( "A", 2, 2, ?data );
```

1.4.6 Flux de données (entrée/sortie)

Un flux de données est une séquence ordonnée d'éléments dans laquelle plusieurs processus peuvent ajouter des éléments à la fin du flux. Dans le cas des flux de données en entrée, les processus peuvent retirer l'élément au début du flux.

De tels flux peuvent être représentés dans un espace de tuples à l'aide d'une série de tuples numérotée séquentiellement. Chacun de ces tuples représente un élément de donnée et est constitué de trois champs : le nom du flux, le numéro de séquence et sa valeur. Ainsi, un flux nommé `"flx1"` sera constitué d'une suite de tuples représentés comme ceci :

```
("flx1", 1, valeur_1)
("flx1", 2, valeur_2)
("flx1", 3, valeur_3)
...
```

Une référence vers la queue du flux est conservée en permanence dans un tuple. Ce tuple est composé de trois champs : le nom du flux, la constante `"tail"` et la valeur du prochain numéro de séquence à utiliser pour l'ajout d'un nouvel élément dans le flux de données. Par exemple, l'indicateur de queue pour le flux de données `flx1` avec `11` éléments serait équivalent à ce tuple :

```
("flx1", "tail", 12)
```

Pour ajouter le nouvel élément `nouvElem` dans le flux de données `flx1`, le processus doit utiliser les opérations suivantes :

```
in( "flx1", "tail", ?index );
out( "flx1", "tail", index+1 );
out( "flx1", index, nouvElem );
```

Ces opérations permettent d'obtenir le numéro de séquence, de l'incrémenter et d'ajouter le nouvel élément du flux de données dans l'espace de tuples. Dans un flux de données utilisé en entrée, on doit effectuer un traitement similaire, mais en manipulant l'index du début du flux. On aura donc un tuple, identifié par la constante `"head"`, afin d'effectuer le suivi de l'index du premier élément du flux de données. Pour retirer un élément d'un flux en entrée, on utilisera donc les instructions suivantes :

```
in( "flx1", "head", ?index );
out( "flx1", "head", index+1 );
in( "flx1", index, ?elem );
```

Un processus qui tente de retirer un élément d'un flux vide bloque jusqu'à ce que le flux devienne non vide. Dans le cas de lecture, plusieurs processus peuvent lire simultanément le flux : chaque processus lit le premier élément, puis le deuxième et ainsi de suite. À la fin du flux, le processus en lecture bloque à nouveau jusqu'à ce qu'un processus en écriture vienne ajouter des éléments à la fin du flux.

Lorsqu'un flux est consommé en entrée par un seul processus, il est possible de se passer du tuple de tête et de permettre au processus qui consomme le flux de conserver l'index de début dans une variable locale uniquement. De la même façon, lorsqu'un seul processus écrit dans un flux, il est possible de se passer du tuple de queue en utilisant uniquement un index local.

1.5 Conclusion

Nous avons démontré dans ce chapitre la simplicité du modèle Linda. L'utilisation d'un espace partagé, avec quelques opérations de base pour échanger l'information entre les processus, demeure un moyen efficace, facile à comprendre et à utiliser. C'est principalement pour cette simplicité que ce modèle a retenu notre attention. De plus, comme Carriero et Gelernter ont su le démontrer [2], malgré la simplicité du modèle, il est quand même possible de réaliser, à un plus haut niveau, des structures partagées plus complexes que de simples tuples de valeurs.

En plus de sa simplicité, le choix du modèle Linda repose également sur la possibilité de réaliser ce modèle à l'aide d'une implémentation indépendante au compilateur du langage hôte. Les chapitres 3 et 4 approfondiront la présentation de cette approche, que nous avons réalisée à l'aide d'une bibliothèque du langage Nit. Cependant, la présentation du modèle du chapitre 1 est strictement basée sur l'implémentation en langage C du modèle Linda. Bien que ces exemples ont permis d'illustrer efficacement les concepts sous-jacents à Linda, force est d'admettre qu'une implémentation en langage C reste de bas niveau et n'est pas vraiment représentative des langages plus modernes. Le chapitre suivant présente justement quelques implémentations sous des langages plus modernes et orientés objets, que nous avons explorés afin d'inspirer notre implémentation Nit du modèle Linda.

CHAPITRE II

QUELQUES IMPLÉMENTATIONS DE LINDA

À l'origine, lorsque Carriero et Gelernter ont introduit Linda [2], ils ont démontré la faisabilité de leur modèle à l'aide d'une implémentation C appelée C-Linda. Depuis, plusieurs implémentations ont été réalisées dans de nombreux langages de programmation. Il aurait été fastidieux de résumer les particularités de l'ensemble de ces implémentations. Nous proposons donc quelques exemples d'implémentation dans des langages plus modernes et, comme Nit, orientés objet.

2.1 Rinda : Linda sous Ruby

Rinda est l'implémentation de Linda sous Ruby [11]. Rinda fournit un cadre d'exécution sous lequel plusieurs processus Ruby résidant sur la même machine, ou sur des machines distinctes, peuvent interagir avec l'espace de tuples. L'exécution du processus qui effectue la gestion de l'espace de tuples s'exécute dans un processus distinct des processus clients. On doit donc commencer par démarrer le processus de l'espace de tuples en tant que service en instanciant un objet de type `Rinda::TupleSpace` et en lui attribuant un URI fixe qui permettra aux autres processus clients d'interagir avec lui.

```
require 'rinda/tuplespace'
URI = "druby://localhost:67671"
DRb.start_service( URI, Rinda::TupleSpace.new )
DRb.thread.join
```

Une fois que le processus qui gère l'espace de tuples est démarré, les processus clients qui veulent interagir avec lui doivent instancier un *proxy* qui prend en charge les communications entre le client et l'espace de tuples.

```
require 'rinda/rinda'
URI = "druby://localhost:67671"
DRb.start_service
ts = Rinda::TupleSpaceProxy.new( DRbObject.new(nil, URI) )
```

Une fois l'objet `TupleSpaceProxy` instancié chez le processus client, ce dernier est en mesure de communiquer avec l'espace de tuples en appelant directement les opérations sur l'objet `TupleSpaceProxy`. Les opérations disponibles sont les suivantes :

- `write( tuple, sec=nil )`

  Permet d'écrire un tuple dans l'espace de tuples. Le dernier paramètre est optionnel et spécifie le temps avant l'expiration du tuple. Une fois le temps expiré, le tuple est supprimé de l'espace de tuples.

- `take( tuple, sec=nil )`

  Lit un tuple correspondant au modèle, le tuple retourné est retiré de l'espace de tuples. Si aucun tuple correspondant n'est trouvé, l'opération est suspendue jusqu'à ce qu'un tuple correspondant au modèle demandé soit ajouté, le tuple sera alors retiré de l'espace de tuples et retourné. Le dernier paramètre est optionnel et il spécifie le temps maximum d'attente; lorsque le temps est écoulé, une exception de type `Rinda::RequestExpiredError` est lancée.

- `read( tuple, sec=nil )`

  Identique à l'opération `take` sauf que le tuple est conservé dans l'espace de tuples.

- `read_all( tuple )`

  Lit tous les tuples de l'espace de tuples qui correspondent au modèle. Les tuples retournés sont conservés dans l'espace de tuples. Un tableau vide est retourné si aucun tuple n'est trouvé.

- `notify( event, tuple, sec=nil )`

  Permet à un client d'être averti lorsqu'une opération (`write, take, delete`) survient dans l'espace de tuples.

Avec Rinda, un tuple est un objet qui est initialisé à l'aide d'un tableau de valeurs. On peut accéder aux valeurs du tuple à l'aide de crochet `[]` ou itérer sur ces valeurs à l'aide de la méthode `each()`. Par exemple, ajouter un tuple représentant l'opération d'addition de 3 et 5 dans l'espace de tuples pourrait s'écrire de la façon suivante :

```
tuple = Rinda::Tuple.new( [ '+', 3, 5 ] )
ts.write( tuple )
```

Il est aussi possible de définir le tuple directement dans l'appel de l'opération, par exemple :

```
ts.write( ['+', 3, 5] )
```

Tout comme les tuples, les modèles de tuple sont aussi des objets. Un modèle est initialisé à l'aide d'un tableau de valeurs comprenant des valeurs réelles et des paramètres formels. Les paramètres formels peuvent être définis à l'aide du type désiré, d'un joker ou d'une expression régulière. Par exemple, on peut récupérer le tuple représentant l'opération d'addition insérée précédemment de trois façons différentes :

- À l'aide d'un modèle de tuple avec des paramètres formels typés :

  ```
  template = Rinda::Template.new( ['+', Numeric, Numeric] )
  tuple = ts.take( template )
  res = tuple[1] + tuple[2]
  ts.write( 'result', res )
  ```

- À l'aide d'un modèle de tuple qui utilise la valeur `nil` pour indiquer des paramètres formels non typés (*joker*). Dans cet exemple, les paramètres de retour peuvent recevoir n'importe quel type de valeurs :

  ```
  tuple = ts.take( ['+', nil, nil ] )
  res = tuple[1] + tuple[2]
  ts.write( 'result', res )
  ```

- À l'aide d'un modèle de tuple qui utilise une expression régulière comme paramètre formel. Cette expression régulière permet de restreindre les valeurs possibles pouvant correspondre au paramètre du tuple. Dans l'exemple suivant, le modèle de tuple spécifie qu'il recherche comme premier paramètre un caractère dont la valeur ne peut être qu'un des quatre caractères spécifiés entre les crochets, c'est-à-dire `-`, `+`, `/` ou `*`.

```
ops, a, b = ts.take( [%r^[-+/*]$, Numeric, Numeric] )
```

Dans ce dernier exemple, au lieu de retourner un tuple, l'opération `take` effectue directement l'affectation des paramètres du tuple dans les variables `ops`, `a` et `b`.

En résumé, Rinda bonifie les opérations de Linda en ajoutant un concept d'expiration de tuple. Ainsi, un tuple peut être supprimé de l'espace de tuples s'il n'est pas extrait dans un certain délai. De plus, les opérations de lecture non bloquantes de Linda ont été remplacées par un paramètre de délai maximum d'attente ; lorsque ce délai est fixé à 0, les opérations de lecture ont le même comportement que les versions non bloquantes des opérations de Linda. Un autre ajout intéressant de Rinda est l'opération `read_all`, qui permet de lire une liste de tuples en une seule opération.

Outre les opérations effectuées sur l'espace de tuples, ce qui différencie davantage Rinda des autres implémentations est l'introduction de deux nouveaux concepts pour spécifier les paramètres formels. Premièrement, le paramètre `nil` permet de spécifier un paramètre formel non typé qui se comporte comme un *joker* dans les types du tuple recherché. Deuxièment, la possibilité de définir une expression régulière comme paramètre formel. Ces deux ajouts permettent une plus grande flexibilité lors de la recherche de tuples.

2.2 PyLinda : Linda sous Python

PyLinda est l'implémentation de Linda sous Python [14] qui peut s'exécuter sur une seule machine ou dans un environnement réparti s'exécutant sur plusieurs machines. PyLinda utilise un espace de tuples universel appelé `universe`. Lorsque PyLinda s'exécute sur un environnement réparti, l'espace de tuples `universe` est distribué sur l'ensemble des noeuds. La distribution est effectuée par l'interconnexion des serveurs Linda qui s'exécutent sur chacun des noeuds.

La première chose à faire pour utiliser PyLinda est de créer le réseau de l'espace de tuples universel en démarrant un serveur Linda pour chaque ordinateur sur lequel un client sera exécuté. Un serveur peut être démarré en exécutant la commande `linda_server.py`. Si l'exécution s'effectue à travers plusieurs noeuds, chacun des noeuds démarre le serveur en spécifiant l'adresse IP ou le nom de domaine (DNS) du noeud où le serveur Linda est déjà démarré.

Pour démarrer le serveur Linda sur le premier noeud :

```
linda_server.py
```

Pour démarrer le serveur Linda sur les autres noeuds :

```
linda_server.py -c <adresse ip ou nom dns>
```

Par la suite, sur n'importe quel système où un serveur Linda est démarré, un processus client peut se joindre au réseau en important le package `linda` et en appelant l'opération `connect`. Il n'est pas nécessaire de spécifier l'adresse du serveur Linda, le client se connecte automatiquement sur le serveur local.

```
import linda
linda.connect()
```

Le processus client a maintenant accès à l'espace de tuples universel avec lequel il peut effectuer des opérations. Les opérations disponibles sont les suivantes :

- `_out( Tuple t )`

  Permet d'écrire un tuple dans l'espace de tuples.

- `_in( Template t )`

  Lit un tuple correspondant au modèle, le tuple retourné est retiré de l'espace de tuples. Si aucun tuple correspondant n'est trouvé, l'opération est suspendue jusqu'à ce qu'un tuple correspondant au modèle demandé soit ajouté, le tuple sera alors retiré de l'espace de tuples et retourné.

- `_rd( Template t )`

  Identique à l'opération `_in` sauf que le tuple est conservé dans l'espace de tuples.

- `_inp( Template t )`

  Lit un tuple correspondant au modèle, le tuple retourné est retiré de l'espace de tuples. Si aucun tuple correspondant n'est trouvé, l'opération est suspendue jusqu'à ce qu'un tuple correspondant au modèle demandé soit ajouté ou qu'une situation de *deadlock* soit détectée. Si un *deadlock* est détecté, l'opération retourne `None`.

- `_rdp( Template t )`

  Identique à l'opération `_inp` sauf que le tuple est conservé dans l'espace de tuples.

- `collect( TupleSpace dest, Template t )`

  Déplace tous les tuples qui correspondent au modèle dans un autre espace de tuples. Les tuples sont retirés de l'espace de tuple d'origine.

- `copy_collect( TupleSpace dest, Template t )`

  Copie tous les tuples qui correspondent au modèle dans un autre espace de tuples. Les tuples sont conservés dans l'espace de tuples d'origine.

Avec PyLinda, il n'y a pas de classes `Tuple` ou `Template` comme avec Rinda. Pour communiquer un tuple ou un modèle de tuple, les opérations sont appelées en énumérant directement les différents paramètres entre les parenthèses d'appel. Les éléments peuvent être de type `int`, `float`, `boolean`, `string` ou `TupleSpace`.

Par exemple, ajouter un tuple représentant une opération d'addition de 3 et 5 dans l'espace de tuples pourrait s'écrire de la façon suivante :

```
linda.universe._out( ('+', 3, 5) )
```

Par la suite, pour récupérer ce même tuple, on utilise l'opération suivante :

```
tuple = linda.universe._in( ('+', int, int) )
res = tuple[1] + tuple[2]
linda.universe._out( ('result', res) )
```

L'opération `_in`, ainsi que les autres opérations de lecture, retourne une liste de valeurs qui représente le tuple trouvé dans l'espace de tuples, correspondant au modèle de tuple demandé. Tout comme avec Rinda, on peut interroger les valeurs contenues dans la liste en spécifiant entre crochets `[]` l'index de la valeur que l'on désire accéder.

En plus de l'espace de tuples `universe`, les clients ont la possibilité de créer de nouveaux espaces de tuples. Il suffit d'instancier la classe `linda.TupleSpace` afin de créer un nouvel espace de tuples. Par la suite, le processus client peut partager son espace de tuples nouvellement créé en l'ajoutant dans l'espace de tuples `universe`.

Par exemple, ces instructions créent un nouvel espace de tuples et le partage à l'aide de l'identifiant "monEspace" :

```
ts = linda.TupleSpace()
linda.universe._out( ('monEspace', ts) )
```

Par la suite, un autre processus client peut récupérer le nouvel espace de tuples à l'aide de la chaine "monEspace".

```
ts = linda.universe._rd( ('monEspace', linda.TupleSpace) )[1]
```

En résumé, PyLinda utilise un espace de tuples global, appelé `universe`, par l'intermédiaire duquel tous les échanges entre processus clients doivent se faire. Même l'ajout d'un nouvel espace de tuples doit être partagé entre les clients par l'intermédiaire de cet espace global. Cet espace est créé automatiquement lors du démarrage du serveur Linda, qui doit s'exécuter sur chaque machine où un processus client s'exécute. D'autres implémentations peuvent fournir un espace de tuples par défaut, mais PyLinda se distingue par le fait qu'il est obligatoire de passer par celui-ci.

Ce qui différencie davantage PyLinda des autres implémentations est l'introduction de deux nouvelles opérations, `collect` et `copy_collect`, qui permettent de transférer un ensemble de tuples d'un espace à un autre.

2.3 Linda sous Scala

L'implémentation de Linda sous Scala [5] fut réalisée à l'aide d'une base de données pour représenter l'espace de tuples partagé, et d'une bibliothèque client/serveur pour permettre l'interaction entre les processus client et cet espace de tuples, c'est-à-dire la base de données sous-jacente. La tâche du serveur est d'attendre les demandes de clients et de les traiter en exécutant les requêtes vers la base de données. Il ne peut y avoir qu'un seul espace de tuples partagé, et il n'y aura donc qu'un seul serveur et une seule base de données. Une fois démarré et configuré, le processus client n'a qu'à importer les deux bibliothèques suivantes pour interagir avec eux.

```
import org.linda.LindaTypes
import org.linda.Client
```

La bibliothèque `org.linda.LindaTypes` fournit une classe abstraite, appelée `LindaType`, qui sert de type de base pour les types utilisés dans la manipulation des tuples. Chaque type utilisé dans un tuple doit hériter de cette classe de base. L'implémentation possède déjà un ensemble de types définis à partir de la classe `LindaType` et qui représentent les types de base. Il est donc possible d'utiliser les types `LindaDouble`, `LindaFloat`, `LindaInteger`, `LindaLong` et `LindaString` à la place des types `Double`, `Float`, `Int`, `Long` et `String`. Il est aussi possible de définir de nouveaux types pouvant être utilisés dans un tuple, à condition que ce type hérite de la classe de base `LindaType`.

Dans une classe `LindaType`, une variable booléenne nommée `formal` est définie et spécifie si l'instance utilisée dans un modèle de tuple est un paramètre formel ou réel. Tout paramètre réel peut être transformé en un paramètre formel et vice-versa en utilisant les méthodes `toFormal` et `toActual` définies dans la classe de base `LindaType`.

Pour comparer les types de données dérivées de `LindaType`, l'opérateur d'égalité (==) doit être redéfini ainsi qu'une méthode auxiliaire `canEqual`. Enfin de générer la signature d'un tuple, la signature de chaque élément du tuple est obtenue par la fonction `getType`; celle-ci doit également être redéfinie, au besoin, lors de la création d'un nouveau type.

```
val i1 = new LindaInteger(28) // On crée des données.
val i2 = new LindaInteger(34)
val s = new LindaString("hello")

val client = new Client // On crée un nouveau client.

client.out(i1, i2, s)  // On émet le tuple

i2.toFormal // On transforme le paramètre réel en paramètre formel.
client.in(i1, i2, s) // On lit le tuple.
```

Listage 1: Exemple Scala d'écriture et de lecture dans l'espace de tuples.

Le tuple et le modèle de tuple sont représentés par une liste d'éléments de type `LindaType`. Comme démontré dans le listage 1, un processus client doit instancier un objet `Client` afin d'appeler des opérations sur l'espace de tuples. L'objet client est responsable d'envoyer les requêtes du processus client au serveur.

Les opérations disponibles sont les suivantes :

- `out( List[LindaType] )`

  Permet d'écrire un tuple dans l'espace de tuples.

- `in( List[LindaType] )`

  Lit un tuple correspondant au modèle, le tuple retourné est retiré de l'espace de tuples. Si aucun tuple correspondant n'est trouvé, l'opération est suspendue jusqu'à ce qu'un tuple correspondant au modèle demandé soit ajouté, le tuple sera alors retiré de l'espace de tuples et retourné.

- `rd( List[LindaType] )`

  Identique à l'opération `rd` sauf que le tuple est conservé dans l'espace de tuples.

En résumé, l'implémentation de Linda sous Scala semble inachevée. Seulement trois opérations de base sont disponibles, et les opérations de lecture non bloquante, bien qu'elles soient nécessaires à la réalisation de certains algorithmes, n'étaient pas présentes lorsque nous avons exploré ce langage.

Ce qui distingue davantage cette implémentation des autres est la possibilité d'utiliser de nouveaux types de valeur dans les tuples. L'implémentation du type `LindaType` dans un nouveau type permet à celui-ci d'être manipulé dans l'espace de tuples et, conséquemment, d'être ajouté à l'intérieur d'un tuple.

2.4 JavaSpace: Linda sous Java

Avec Java, un espace de tuples est appelé un `JavaSpace`. Les JavaSpaces sont des programmes qui sont démarrés en tant que services. Ces services reposent sur l'architecture de *Jini* qui est une extension de Java pour les systèmes distribués. La technologie *Jini* fournit une infrastructure qui simplifie les connexions entre les systèmes distants. Cette infrastructure, dans un premier temps, permet aux services de faire connaître leur existence, et dans un deuxième temps permet aux clients de rechercher des services sur le réseau, d'échanger les contrats d'interfaces et d'établir des connexions entre eux.

Avant de pouvoir démarrer un service JavaSpace, il faut que l'infrastructure Jini soit en place et que les autres services connexes soit fonctionnels : serveur HTTP, *RMI Activation Daemon*, *Lookup Service* et *Transaction Manager* [8]. Par la suite, lorsqu'un service JavaSpace sera

démarré, il sera visible sur le réseau et facilement accessible par un programme client. Un programme qui désire utiliser un JavaSpace appellera simplement la méthode `SpaceAccessor.getSpace()` pour obtenir une référence sur un JavaSpace. La méthode retourne le JavaSpace par défaut, mais accepte également un nom en paramètre pour spécifier un JavaSpace spécifique.

```
JavaSpace space = SpaceAccessor.getSpace();
JavaSpace space2 = SpaceAccessor.getSpace("unJavaSpaceName");
```

Une fois la référence sur le JavaSpace obtenue, il suffit d'appeler directement les opérations sur ce JavaSpace. Ce ne sont pas tous les objets qui peuvent être utilisés par les opérations d'un JavaSpace, mais uniquement les objets qui implémentent l'interface `Entry` et que l'on appelle une entrée (*entry*). Cette interface ne contient aucune constante, méthode ou traitement spécial à implémenter. Elle ne sert qu'à marquer l'objet comme étant admissible à être déposé dans le JavaSpace. Par contre, une entrée doit respecter certaines conditions :

- Une entrée doit avoir un constructeur public sans argument requis lors de la sérialisation pour le transfert vers le JavaSpace.
- Une entrée doit posséder des attributs publics afin d'effectuer une comparaison sur ses valeurs lors de la recherche d'objet correspondant dans le JavaSpace.
- Une entrée doit utiliser uniquement des références comme attributs afin de pouvoir définir les paramètres formels d'un modèle (valeur `null`) lors des lectures dans le JavaSpace.

Voici un exemple d'une classe qui définit une entrée qui satisfait les conditions nécessaires à son utilisation dans un JavaSpace :

```
public class Message implements Entry {
      public String content;
      public Message() {}
}
```

Bien qu'elles se nomment différemment, les opérations disponibles sur un JavaSpace sont pratiquement les mêmes que celles utilisées dans le modèle Linda. Les opérations disponibles sur un JavaSpace sont les suivantes :

- `write( Entry entry, Transaction trx, long lease )`

Permet d'écrire une entrée (objet) dans le JavaSpace. Le paramètre `trx` est utilisé pour manipuler de manière atomique des opérations sur plusieurs JavaSpace, la valeur `null` est utilisée pour indiquer qu'il n'y a aucune transaction. Le dernier paramètre indique le délai d'existence de l'objet dans le JavaSpace, la constante `Lease.FOREVER` est utilisée pour indiquer qu'il n'y a aucun délai d'expiration.

- `take( Entry tmpl, Transaction trx, long timeout )`

  Lit une entrée (objet) correspondant au modèle, l'entrée retournée est retirée du JavaSpace. Si aucune entrée correspondant au modèle n'est trouvée, l'opération est suspendue jusqu'à ce qu'une entrée correspondant au modèle demandé soit ajoutée, l'entrée sera alors retirée du JavaSpace puis retournée. Le paramètre transaction est utilisé pour manipuler de manière atomique des opérations sur plusieurs JavaSpace, la valeur `null` est utilisée pour indiquer qu'il n'y a aucune transaction. Le `timeout` indique le délai maximum d'attente lorsque l'opération est suspendue, la constante `NO_WAIT` peut être utilisée pour indiquer qu'il n'y a aucun délai d'attente (0 ms).

- `read( Entry tmpl, Transaction trx, long timeout )`

  Identique à l'opération `take` sauf que l'entrée est conservée dans le JavaSpace.

- `takeIfExists( Entry tmpl, Transaction trx, long timeout )`

  Identique à l'opération `take` mais sans délai d'attente. Le `timeout` est utilisé uniquement dans un contexte de transaction.

- `readIfExists( Entry tmpl, Transaction trx, long timeout )`

  Identique à l'opération `takeIfExists` sauf que l'entrée est conservée dans le JavaSpace.

- `snapshot( Entry entry )`

  Permet d'obtenir une entrée déjà sérialisée et prête pour la transmission vers le JavaSpace, l'entrée retournée peut par la suite être réutilisé par plusieurs opérations. Ceci est utile lorsque l'entrée est un modèle qu'on utilise à plusieurs reprises dans des opérations de lecture (`read`/`take`), puisque cela permet d'éliminer le temps de préparation d'une entrée et accélère les opérations de lecture répétitives.

- `notify( Entry tmpl, Transaction trx, RemoteEventListener listener,`
  `long lease, MarshalledObject handback )`

  Permet d'obtenir une notification du JavaSpace lorsqu'un événement survient dans le JavaSpace impliquant une entrée correspondant au modèle. La correspondance de l'entrée

avec le modèle est faite comme dans une opération `read()`. Lorsqu'une entrée correspondante est écrite dans le JavaSpace, la méthode spécifiée par le `listener` en sera informée en incluant l'objet `handback`. Le paramètre `lease` indique pendant combien de temps le client en sera informé. Si une transaction est spécifiée, le client sera informé uniquement des entrées impliquées dans cette transaction.

```
public class Message implements Entry {
      public String content;
      public Message() {}
}

Message msg = new Message();
msg.content = "Hello World";

JavaSpace space = SpaceAccessor.getSpace();
space.write(msg, null, Lease.FOREVER);

Message template = new Message();
template.content = null;
Message result = (Message)space.read(template, null, Long.MAX_VALUE);

System.out.println(result.content);
```

Listage 2: Exemple d'écriture et de lecture dans un JavaSpace.

Le listage 2 donne un exemple d'une écriture et d'une lecture d'une entrée de type `Message` dans le JavaSpace par défaut. Puisque les références des attributs d'une entrée servant de modèle et qui doivent servir de paramètre formel doivent être `null`, dans l'exemple précédent c'est donc l'attribut `content` qui prend la valeur `null`.

À noter que les opérations de lecture `read` et `take` retournent un objet du type du modèle ou une spécialisation de celui-ci.

En résumé, JavaSpace introduit un nouveau concept de transaction dans les opérations de base d'un espace de tuples, afin d'effectuer de manière atomique le transfert de tuples d'un espace à un autre. De plus, deux nouvelles opérations sont ajoutées aux opérations de base. La première permet d'optimiser le temps d'exécution en fournissant un tuple déjà sérialisé et prêt

à être expédié à un espace de tuples. La deuxième permet de s'abonner à des événements qui surviennent dans l'espace de tuples. Rinda permet aussi la gestion d'événements dans un espace de tuples, par contre, JavaSpace semble plus efficace dans sa gestion, car il permet de spécifier un modèle de tuple pour lequel les événements doivent être lancés.

Ce qui différencie davantage JavaSpace des autres implémentations est l'utilisation d'objets, au lieu de tuples de valeurs, dans les opérations de communication entre les clients et un JavaSpace. Pour qu'un objet puisse être ajouté dans un JavaSpace, il suffit que le type de l'objet implémente le type `Entry`. Scala permet aussi de déclarer de nouveaux types, mais l'interface du type de base dans JavaSpace ne possède pas de méthode abstraite comme dans Scala. De plus, JavaSpace supporte l'héritage, c'est-à-dire que les objets lus dans le JavaSpace peuvent être du type demandé ou d'un sous-type de celui-ci, ce qui est utile pour un langage orienté objet comme Java.

2.5 Conclusion

Ces quelques implémentations de Linda nous ont permis d'avoir une meilleure idée de ce qu'il serait possible de réaliser. Plusieurs points ressortent, dont certains qui ont influencé la réalisation de notre propre implémentation. Tout d'abord, on constate que les différentes implémentations de Linda qui ont été explorées ont toutes été réalisées à l'aide d'une bibliothèque externe au langage. De plus, les échanges entre un client et l'espace de tuples reposent principalement sur des programmes tiers (service Ruby, serveur Linda, Jini) qui gèrent les communications. L'avantage de ces différentes approches est que ces implémentations peuvent s'exécuter sur de multiples plateformes, à la fois sur des systèmes multicoeurs ou des systèmes distribués. En contrepartie, l'abstraction de l'architecture amène une complexité dans la configuration du système avant de permettre son utilisation. L'architecture cible n'étant pas une préoccupation, et par souci de conserver son utilisation le plus simple possible, l'implémentation qui fut réalisée dans le cadre de notre mémoire se limite à une architecture à processeurs multicoeurs, et non une architecture distribuée. Comme on pourra le constater dans le chapitre 3, aucune configuration n'est nécessaire : il suffit d'importer notre bibliothèque pour être prêt à utiliser un espace de tuples.

Dans chacune des implémentations, l'espace de tuples est représenté par un objet que l'on obtient par l'appel d'une méthode. Cette méthode communique avec une tierce partie

pour obtenir une référence sur l'espace de tuples qui, rappelons-le, est déjà démarré. Avec Nit, on fera différemment. Afin de simplifier son utilisation, il suffira d'appeler le constructeur pour instancier un nouvel espace de tuples opérationnel. Aucune configuration n'est nécessaire puisqu'on utilisera uniquement des communications entre les processus résidant sur la même machine. De plus, un peu comme l'a fait Java, pour différencier un espace de tuples de Nit des autres implémentations de Linda, on utilisera un nom distinct : si Java a son JavaSpace, Nit aura son NitSpace.

Bien que certains y porteront peu d'importance, le nom des opérations disponibles sur l'espace de tuples permet de mieux saisir le but de l'opération. Scala et Python ont choisi de conserver le nom des opérations de Linda : `out`, `in`, `inp`, `rd` et `rdp`. Pour ceux qui connaissent bien Linda, ces opérations sont faciles à reconnaître, mais pour les néophytes, ces noms ne sont pas très significatifs. Tel qu'illustré au chapitre 3, nous avons opté pour des noms plus significatifs, comparables à ceux utilisés avec JavaSpace et Rinda. Les noms retenus sont `write` pour la lecture, `take` pour l'extraction et `read` pour la lecture — en plus des versions non bloquantes qui seront suffixées par `_no_wait`. Pour les autres opérations qui ont été ajoutées aux différentes implémentations, bien que certaines soient intéressantes, il nous semble inutile d'intégrer d'autres opérations lors d'une première implémentation, et on se limitera donc aux opérations de base de Linda.

Chaque implémentation doit utiliser un objet pour contenir les valeurs à échanger, c'est-à-dire un objet qui représente un tuple et un modèle de tuples. En Java, c'est l'objet qui sert de conteneur, ses attributs représentant les valeurs. Scala utilise tout simplement une liste d'objets. Sous Ruby et Python, de nouvelles classes d'objets sont ajoutées afin de représenter le tuple et le modèle de tuples. Respectivement nommés `Tuple` et `Template`, ces objets peuvent être manipulés exactement comme les tableaux. Ainsi, il est possible d'accéder aux valeurs à l'aide de l'opérateur d'indexation représenté par des crochets `[]`. Pour des raisons de lisibilité, afin de distinguer explicitement le tuple et le modèle de tuples, notre implémentation utilise la même approche que Ruby et Python.

Afin d'étendre les possibilités des types pouvant être utilisés à l'intérieur d'un tuple, notre implémentation permet d'étendre un type de base qui autorise les nouveaux sous-types à être insérés dans un tuple. L'approche retenue est donc celle de Scala, où l'on utilise un type de base qui inclut des méthodes abstraites nécessaires à sa manipulation. L'approche de Scala a été choisie, car elle était moins complexe à réaliser que celle de Java.

Le chapitre suivant présente l'ensemble des détails de l'API qui fut réalisée lors de notre implémentation Nit du modèle Linda. Outre ces quelques points inspirés des implémentations existantes, l'API réalisé fut également influencé pare le jeu d'instructions de Nit.

CHAPITRE III

NITSPACE — API

NitSpace est notre implémentation de Linda sous le langage Nit. Comme la majorité des implémentations de Linda, celle-ci a été développée sous forme de bibliothèque fournissant un ensemble de classes et d'opérations permettant à différents processus d'interagir entre eux par l'entremise d'un espace de tuple partagé que l'on nomme NitSpace.

3.1 NitSpace : un espace de tuples partagé

La première chose à faire lorsqu'on désire utiliser un NitSpace est d'importer le module qui l'implémente à l'aide de l'instruction import.

```
import nitspace
```

Suite à l'importation, il est maintenant possible d'instancier un objet NitSpace pour l'échange d'information entre processus. L'instruction `new NitSpace` crée automatiquement un nouveau processus qui gèrera les opérations sur cet espace partagé.

```
# Génère un nouveau processus avec un espace mémoire partagé.
var nitSpace = new NitSpace
```

Puisque l'exécution d'un NitSpace s'effectue à l'aide d'un processus distinct, il est important d'appeler l'opération `stop` du NitSpace lorsqu'on est prêt à en disposer, afin de terminer le processus sous lequel il s'exécute.

```
# Termine le processus et libère l'information contenue dans le NitSpace.
nitSpace.stop
```

L'échange d'information entre un programme et un NitSpace s'effectue par l'échange de tuples et de modèles de tuple. Un tuple contient uniquement une liste de valeurs, tandis qu'un modèle de tuple peut aussi contenir des types lorsqu'il y a absence de valeurs.[1] Tout comme avec Linda, l'opération d'écriture n'utilise qu'un tuple en paramètre, tandis que les opérations de lecture utilisent un modèle de tuple pour effectuer une demande au NitSpace et reçoivent en retour un tuple correspondant au modèle demandé.

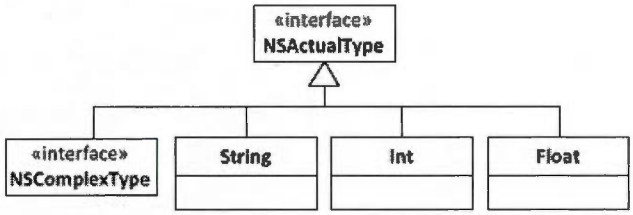

Figure 3.1 Le type NSActualType et ses spécialisations.

Pour écrire un tuple dans un NitSpace, on utilise l'opération `write`. Cette opération prend une liste de valeurs en paramètre représentant le tuple que l'on désire ajouter. Les valeurs insérées dans un tuple sont appelées des paramètres réels et sont représentées par le type `NSActualType`. Comme illustrés à la figure 3.1, les types `String`, `Int` et `Float` sont des spécialisations de `NSActualType`. À l'exception des types complexes[2], seuls ces trois types peuvent être insérés dans un tuple. Le programmeur n'a pas à se soucier de la création du tuple lors de l'appel, car c'est l'opération `write` qui s'en charge.

Par exemple, soit l'instruction suivante :

```
nitSpace1.write( "abcd", 12, 5.04 )
```

Son exécution crée un tuple constitué de la chaîne de caractères `"abcd"`, du nombre entier `12`, suivi du nombre réel `5.04`, et ensuite ce tuple est ajouté dans le NitSpace identifié par `nitSpace1`. L'instruction d'écriture est non bloquante, c'est-à-dire que le programme n'a pas besoin d'attendre une confirmation de réception, il poursuit son exécution dès qu'il envoie la demande d'ajout au NitSpace. Chaque instruction `write` ajoute un nouveau tuple dans le NitSpace, et si les mêmes valeurs sont envoyées à plusieurs reprises, il existera un exemplaire

[1] Le chapitre 1 sur Linda explique en détail les concepts de tuple et modèle de tuple.

[2] La section 3.4 explique en détail l'utilisation des types complexes.

distinct du tuple pour chaque appel d'écriture qui aura été fait — le NitSpace agit donc comme un *sac*, i.e., un *multi-ensemble.*

Pour récupérer un tuple à partir d'un NitSpace, on utilise l'opération `take`. Cette opération prend en paramètre une liste de valeurs et de types afin de définir un modèle de tuple à récupérer. Les types de valeurs autorisés sont les mêmes que ceux d'un tuple. En l'absence d'une valeur, on doit utiliser des paramètres formels qui indiquent le type de la valeur recherchée. Les paramètres formels sont représentés par le type `NSFormalType`. Les types `NSActualType` et `NSFormalType` sont regroupés sous le même type `NSType`. — voir figure 3.2.

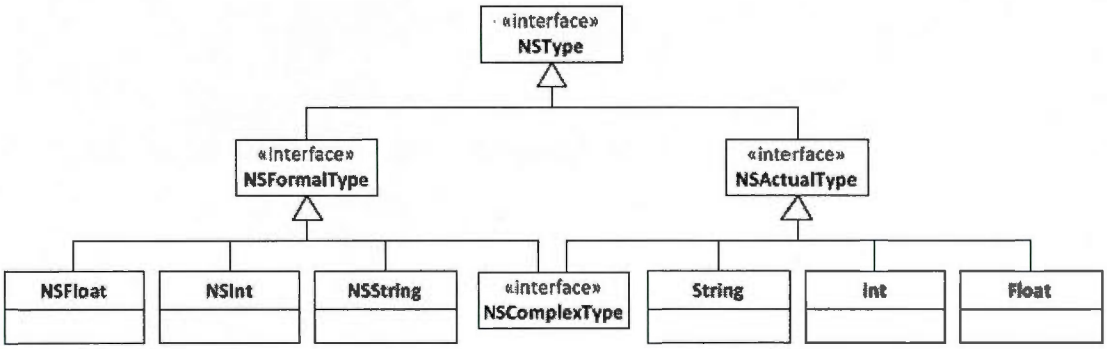

Figure 3.2 Types pouvant être utilisés dans les tuples et les modèles de tuples.

Puisqu'un modèle de tuple peut être constitué de paramètres réels et formels, le type de ses paramètres est `NSType`. Il est possible d'obtenir des instances de types formels à l'aide des méthodes `nsString`, `nsInt` et `nsFloat`. Ainsi, lors de la définition d'un modèle, les paramètres formels pourront être obtenus en appelant les méthodes `nsString` pour indiquer un type `String`, `nsInt` pour indiquer un type `Int` et `nsFloat` pour indiquer un type `Float`.

Par exemple, soit l'instruction suivante :

```
var resultat = nitSpace1.take( "abcd", nsInt, nsFloat )
```

Son exécution permet de retirer, du NitSpace identifié par `nitSpace1`, un tuple composé de trois valeurs dont la première est la chaîne de caractères `"abcd"`, suivie d'un entier et d'un nombre réel. La valeur de retour est un objet de type `Tuple`. Pour accéder aux valeurs contenues dans le tuple, on utilise l'opérateur d'accès `[]` avec l'index de l'élément désiré. L'indexation débute à la position 0, et la valeur de retour d'un élément d'un tuple est de type `NSActualType`,

c'est-à-dire des paramètres réels uniquement. Les types `String`, `Int` et `Float` sont des spécialisations de `NSActualType`. Il est toujours possible de convertir le type des éléments à l'aide du mot-clé `as`.

```
var resultat: Tuple = nitSpace.take( "abcd", nsInt, nsFloat )
var uneChaine: String = resultat[0].as(String)
var unEntier: Int = resultat[1].as(Int)
var unReel: Float =  resultat[2].as(Float)
```

Il existe une variante à l'opération `take` qui est non bloquante, l'opération `take_no_wait`. Cette opération retire également un tuple du NitSpace, mais au lieu d'être en attente, l'opération reçoit une valeur nulle lorsqu'aucun tuple correspondant n'est trouvé. Ainsi, dans le cas d'une opération `take_no_wait`, le type de la valeur de retour est `nullable Tuple`. Cette alternative offre plus de flexibilité lorsqu'on n'a pas la certitude qu'un tuple existe ou que l'on ne connait pas le nombre exact d'exemplaires disponibles.

```
var t: nullable Tuple
loop
    t = nitSpace.take_no_wait("MESSAGE", nsString)
    if t == null then break
    print t[1].as(String)
end
```

Dans l'exemple précédent, on affiche à l'écran les messages que l'on récupère dans un NitSpace à l'aide de l'opération `take_no_wait` et du modèle de tuple `("MESSAGE", nsString)`. Puisque le nombre de messages disponibles dans le NitSpace n'est pas connu à l'avance, on effectue le traitement à l'intérieur d'une boucle. Lorsque l'opération `take_no_wait` retourne une valeur nulle, indiquant ainsi qu'il n'y a plus de message disponible dans le NitSpace, on sort de la boucle à l'aide de l'instruction `break` mettant ainsi fin au traitement.

Il est aussi possible de lire un tuple à l'aide de l'opération `read`. Cette opération fonctionne exactement comme l'opération `take` à l'exception que l'exemplaire du tuple retourné est conservé dans le NitSpace. L'exemplaire ainsi conservé demeure accessible pour une prochaine lecture. L'opération `read` est bloquante; pour une lecture non bloquante il faut plutôt utiliser l'opération `read_no_wait`. Les opérations `read` et `read_no_wait` sont utiles pour lire un résultat qui est destiné à l'ensemble des processus clients et qui doit demeurer dans le NitSpace afin d'être lu

par les autres processus qui partagent le NitSpace.

```
var nitSpace: NitSpace

# Vérifie si l'instruction de contrôle d'arrêt du processus a été lancée.
fun termine: Bool
do
    var resultat: nullable Tuple = nitSpace.read_no_wait( "Control", "STOP" )
    return resultat != null
end

fun executer
do
    # Boucle tant que la commande d'arrêt n'a pas été reçue.
    while not termine do
        ... # Instructions du traitement récurrent.
    end
end
```

Listage 3: Programme Nit illustrant l'opération `read_no_wait` — extrait du programme de calcul des nombres premiers.

L'opération `read_no_wait` est bien adaptéee pour vérifier des indications de contrôle qui peuvent être diffusées par l'entremise du NitSpace. Par exemple, le listage 3 présente un extrait du programme de calcul des nombres premiers[3] où un processus boucle sur un traitement jusqu'à ce qu'il reçoive l'indication d'arrêter par un processus contrôleur.

3.2 Le type Tuple

Les objets de type `Tuple` servent de contenant pour spécifier une liste ordonnée d'éléments. Toute interaction avec un `NitSpace` doit nécessairement s'effectuer par l'entremise d'un `Tuple`. Bien que les opérations d'écriture et de lectures instancient automatiquement les tuples, il est aussi possible, dans certaines situations, de vouloir créer explicitement un tuple. Pour créer un nouveau `Tuple`, il suffit d'appeler le constructeur par défaut avec la liste des éléments en paramètre.

[3] L'exemple complet du calcul des nombres premiers est disponible à l'appendice A.

```
var unTuple = new Tuple( "abcd", 12, 5.04 )
```

Il n'est pas possible de fournir des types formels en paramètres ; tous les éléments qui sont fournis au tuple doivent être des paramètres avec des valeurs réelles représentées par le type `NSActualType`.

Il est aussi possible d'instancier un `Tuple` à partir d'un tableau d'éléments qui est passé au constructeur `with_array` ou à partir d'une liste d'éléments qui est passée au constructeur `with_list`.

```
var a = new Array[NSActualType]
a[0] = "abcd"
a[1] = 12
a[2] = 5.04
var t = new Tuple.with_array(a)
```

Sachant qu'il est possible à partir d'un `Tuple` de générer un tableau contenant la liste de ses éléments, il est possible de dupliquer un `Tuple` de la façon suivante :

```
var unTuple = new Tuple("abcd", 12, 5.04)
var unNouveauTuple = new Tuple.with_array(unTuple.to_array)
```

On peut avoir exactement le même résultat en utilisant une liste d'éléments à la place d'un tableau :

```
var unTuple = new Tuple("abcd", 12, 5.04)
var unNouveauTuple = new Tuple.with_list(unTuple.to_list)
```

Finalement, il est aussi possible d'instancier un `Tuple` vide sans élément à l'aide du constructeur `with_empty`. Les éléments peuvent, par la suite, être ajoutés au tuple avec la méthode `add_Element` ou `add_ListElement` pour ajouter plusieurs éléments à la fois.

```
var unTuple = new Tuple.with_empty
unTuple.add_Element("abcd")
unTuple.add_Element(12)
```

La méthode `add_ListElement` est pratique lorsqu'on veut concaténer deux tuples en

ajoutant au premier tuple la liste des éléments du deuxième tuple obtenu à l'aide de la méthode `to_list`.

```
tuple1.add_ListElement(tuple2.to_list)
```

Il est aussi possible de modifier ou d'ajouter un élément à l'aide de l'opérateur d'accès `[]` avec l'index de l'élément que l'on désire accéder.

```
unTuple[3] = 5.04
unTuple[2] = 15
```

Dans le cas d'une lecture, l'opérateur `[]` retourne une valeur de type `NSActualType`. Bien qu'il soit possible de convertir le type des éléments à l'aide d'une opération standard de coercition (*cast*), on peut utiliser également les méthodes `get_StringElement()`, `get_IntElement()` et `get_FloatElement()` comme alternatives. Ces méthodes prennent en argument l'index de l'élément que l'on désire accéder et effectuent la conversion vers les types `String`, `Int` et `Float`. Pour les types complexes, on peut utiliser la méthode `get_Element()` mais celle-ci, comme avec l'opérateur `[]`, exige une conversion explicite pour accéder aux méthodes de son type plus spécifique.

```
var uneChaine: String = unTuple.get_StringElement(0)
var unEntier: Int = unTuple.get_IntElement(1)
var unReel: Float =  unTuple.get_FloatElement(2)
var unMessage: Message = unTuple.get_Element(3).as(Message)
```

Le type `Tuple` offre beaucoup de flexibilité dans sa manipulation. Cette flexibilité devient intéressante, car on peut par la suite ajouter le tuple au NitSpace en le passant directement à l'opération `write`.

```
var unTuple = new Tuple("abcd", 12, 5.04)
nitSpace.write(unTuple)
```

Soulignons qu'il n'est pas possible d'ajouter plus d'un tuple à la fois, Ainsi, parce que le type `Tuple` est aussi considéré comme une spécialisation du type `NSActualType`, un programmeur pourrait être tenté d'ajouter plusieurs tuples en même temps. Cela reviendrait à construire un tuple qui contiendrait d'autres tuples et l'imbrication des tuples n'est pas supportée par notre

NitSpace. Plus précisément, bien qu'il soit possible d'imbriquer des tuples à l'intérieur d'un autre tuple, l'appel de l'opération `write` *générera une erreur lors de son exécution.*

3.3 Le type Template (modèle de tuple)

Les objets de type `Template` servent exclusivement à spécifier, lors des opérations de lecture, le modèle de tuple que l'on désire récupérer dans un NitSpace. Bien que les opérations de lecture instancient des objets de type `Template`, il peut s'avérer pratique de manipuler directement un modèle. On peut instancier un `Template` à l'aide du constructeur par défaut qui prend une liste des éléments à ajouter en paramètre.

```
var unTemplate = new Template(nsString, nsInt, nsFloat)
```

Les opérations pouvant être effectuées sur un modèle de tuple, c'est-à-dire sur un type `Template`, sont sensiblement les mêmes que celles que l'on peut effectuer sur objet de type `Tuple`. Ainsi, tout comme avec le `Tuple`, il est aussi possible d'instancier un `Template` avec les constructeurs `with_array`, `with_list` ou `with_empty`. La différence entre un `Tuple` et un `Template` est principalement le type des éléments que l'on peut y ajouter. Pour un `Template`, chacun des éléments doit obligatoirement être de type `NSType`. Le type `NSType` englobe les types réels avec `NSActualType`, et les types formels avec `NSFormalType`. Sachant que les types réels, désignés par `NSActualType`, représentent tous les types que l'on peut ajouter dans un `Tuple`, il est donc possible d'instancier un modèle `Template` avec les éléments d'un `Tuple`.

```
var unTemplate = new Template.with_list(unTuple.to_list)
```

Outre toutes les opérations disponibles pour un `Tuple`, trois méthodes additionnelles sont permises sur un `Template` afin de faciliter l'ajout d'éléments. Les méthodes `add_StringElement`, `add_IntElement` et `add_FloatElement` permettent d'ajouter un élément à la fin du `Template`. Cependant, ces méthodes ont la particularité d'accepter des types `nullable` en paramètre. Lorsqu'il y a absence de valeur et qu'une de ces méthodes reçoit un `null` en paramètre, c'est l'objet de type formel qui est ajouté au `Template`. Le programmeur n'a pas à vérifier l'état de la variable avant d'ajouter au `Template`, c'est la méthode qui s'occupe d'ajouter la valeur ou son type formel en absence de valeur. Voici un exemple de deux variables de type `nullable Int` qui sont ajoutées à un `Template` :

```
var unInt1: nullable Int = 18
var unInt2: nullable Int = null
```

```
var unTemplate  = new Template.with_empty
unTemplate.add_IntElement(unInt1)
unTemplate.add_IntElement(unInt2)
```

Dans cet exemple, la première variable `unInt1` contient la valeur 18, tandis qu'il y a absence de valeur pour la variable `unInt2`. Au final, après l'ajout des deux éléments avec la méthode `add_IntElement`, le `Template` contiendra la valeur 18 et un objet de type `NSInt`. Ceci serait équivalent à la création du `Template` suivant :

```
var unTemplate = new Template(18, nsInt)
```

Le type `Template` peut s'avérer utile lorsque l'on doit effectuer plusieurs lectures consécutives du même modèle de tuple ou d'un modèle similaire. La flexibilité qu'offre la manipulation de `Template` permet d'ajuster plus facilement les éléments qui constituent le modèle de tuple et, à la fin, d'appeler tout simplement l'opération de lecture en lui passant directement le `Template` en paramètre.

```
var unModele = new Template("abcs", nsInt, nsFloat)
var unTuple = nitSpace.take(unModele)
```

Il n'est pas possible de lire plusieurs modèles à la fois, car tout comme avec le tuple, cela reviendrait à construire un modèle qui contiendrait d'autres modèles et l'imbrication des modèles n'est pas supportée par notre NitSpace. Une erreur à l'exécution surviendra si une opération de lecture est effectuée avec un modèle qui contient d'autres modèles ou s'il contient des tuples.

3.4 Les types complexes

En plus des types de base, c'est-à-dire les types `String`, `Int` et `Float`, il est aussi possible d'ajouter dans un tuple des types complexes. Les types complexes sont des classes personnalisées qui implémentent l'interface `NSComplexType` ce qui leur permet d'être utilisés dans un tuple.

```
interface NSComplexType special NSActualType
    fun nsTypeName: String is abstract
    fun nsTemplate: Template is abstract
```

```
    fun nsNewObject(t: Tuple): NSComplexType is abstract
end
```

Afin d'être manipulé dans un tuple et dans un NitSpace, trois opérations doivent être implémentées lorsqu'un objet spécialise cette interface. L'opération `nsTypeName` doit retourner le nom du type, ce qui permet de distinguer dans le NitSpace un type d'un autre type qui aurait exactement les mêmes attributs. Il est important que ce nom soit unique et qu'il représente le type de l'objet. Il est fortement recommandé que l'opération `nsTypeName` retourne le nom de la classe, car ceci assure que le nom du type représente le type de l'objet et qu'il est unique. Par exemple, si l'on désire ajouter le nouveau type `Message` dans un tuple, on commence par définir une classe `Message` qui spécialise `NSComplexType` et on lui ajoute l'opération `nsTypeName` qui retourne le nom du type, c'est-à-dire `"Message"`.

```
class Message special NSComplexType
    redef fun nsTypeName: String
    do
        return "Message"
    end
end
```

Le type des attributs que l'on ajoute à une classe qui spécialise `NSComplexType` doit être un des trois types de base. De plus, pour permettre la génération d'un modèle de tuple, chacun de ces attributs doit pouvoir être `nullable`. Il est donc possible d'utiliser des attributs de type `nullable String`, `nullable Int` ou `nullable Float`. Il n'est pas possible d'utiliser un type complexe comme attribut d'un autre type complexe. Les restrictions des types des attributs de la classe sont dues aux opérations de sérialisation et de désérialisation à travers un tuple. Ces opérations sont réalisées par les deux autres opérations, `nsTemplate` et `nsNewObject`, qui doivent nécessairement être définies lorsqu'une classe implémente l'interface `NSComplexType`.

Le but de l'opération `nsTemplate` est de générer un modèle de tuple à partir des attributs de l'objet. Le type représentant le modèle de tuple est un `Template`. Lors de la construction d'un objet de type `Template`, il est recommandé d'utiliser l'une des trois méthodes suivantes : `add_StringElement`, `add_IntElement` ou `add_FloatElement`. Ces méthodes permettent d'ajouter la valeur d'un attribut, ou son type formel, à la fin du `Template`. Par exemple, si on poursuit la définition de notre classe `Message` en ajoutant deux attributs, le premier attribut de type `Int` appelé `senderId` qui identifie l'expéditeur du message, et un second attribut

de type `String` appelé `content` qui contient le contenu de notre message, on ferait comme suit :

```
class Message special NSComplexType
    var content: nullable String
    var senderId: nullable Int
    ...
    redef fun nsTemplate: Template
    do
        var t = new Template.with_empty
        t.add_StringElement(content)
        t.add_IntElement(senderId)
        return t
    end
end
```

La fonction `nsTemplate` est aussi utilisée pour générer le tuple qui est utilisé lors de l'écriture dans un NitSpace. Lors de l'écriture, il est important de s'assurer que chacun des attributs de l'objet possède une valeur réelle, c'est-à-dire que ses attributs ne sont pas à `null`. Des valeurs nulles provoqueraient une erreur d'exécution, car les types formels ne peuvent pas être ajoutés dans un tuple.

La dernière opération qui doit être implémentée lors de la spécialisation de `NSComplexType` est la méthode `nsNewObject`. Cette méthode reçoit un tuple de valeurs en paramètre et retourne un objet du type de la classe. L'opération `nsNewObject` permet d'instancier un nouvel objet à partir d'un tuple de valeurs lu dans le NitSpace. L'ordre des éléments qui sont fournis dans le tuple est exactement le même que celui qui est fourni par la méthode `nsTemplate`. Pour compléter l'exemple de notre classe `Message`, l'opération `nsNewObject` appelle le constructeur par défaut de la classe `Message`, en lui refilant les éléments contenus dans le tuple, et retourne directement cette nouvelle instance.

```
class Message special NSComplexType
    ...
    redef fun nsNewObject(t: Tuple): Message
    do
        return new Message(t[0].as(String), t[1].as(Int))
    end
end
```

```
# Définition d'un message
class Message special NSComplexType
  ...
  redef fun to_s: String
  do
    if content != null then
     return content.to_s
   end
   return ""
  end
end

# Création du NitSpace et ajout des messages
var nitSpace = new NitSpace
nitSpace.write ( new Message("Bonjour", 1) )
nitSpace.write ( new Message("Hello", 2) )
nitSpace.write ( new Message("Hola", 1) )

# Crée un modèle pour récupérer les messages du senderId = 1
var modele = new Template(new Message (null, 1))

# Boucle tant qu'il y a des messages à afficher pour le senderId = 1
loop
  var t = nitSpace.take_no_wait(modele)
  if t == null then break
  print t[0].as(Message)
end
```

Listage 4: Exemple d'écriture et de lecture d'objets `Message` dans un NitSpace.

Le listage 4 donne un exemple d'une écriture et d'une lecture d'un objet dans un NitSpace. Dans cet exemple, la méthode `to_s` est redéfinie pour l'objet de type `Message`. Par la suite, lorsqu'un message est retiré du NitSpace, la fonction `to_s` est appelée implicitement par l'instruction `print` afin d'afficher le contenu du message. Son exécution produira alors le résultat suivant :

```
Hola
Bonjour
```

3.5 Les processus parallèles

L'interaction avec un NitSpace est intéressante lorsque plusieurs processus s'échangent de l'information par l'intermédiaire de celui-ci. En effet, il est possible pour des processus s'exécutant en parallèle ou de façon concurrente d'utiliser des NitSpaces pour échanger de l'information et pour coordonner leurs activités. Telle qu'illustrée à la figure 3.3, chaque processus peut posséder plusieurs références sur des NitSpaces distincts où ils peuvent échanger avec des groupes de processus différents.

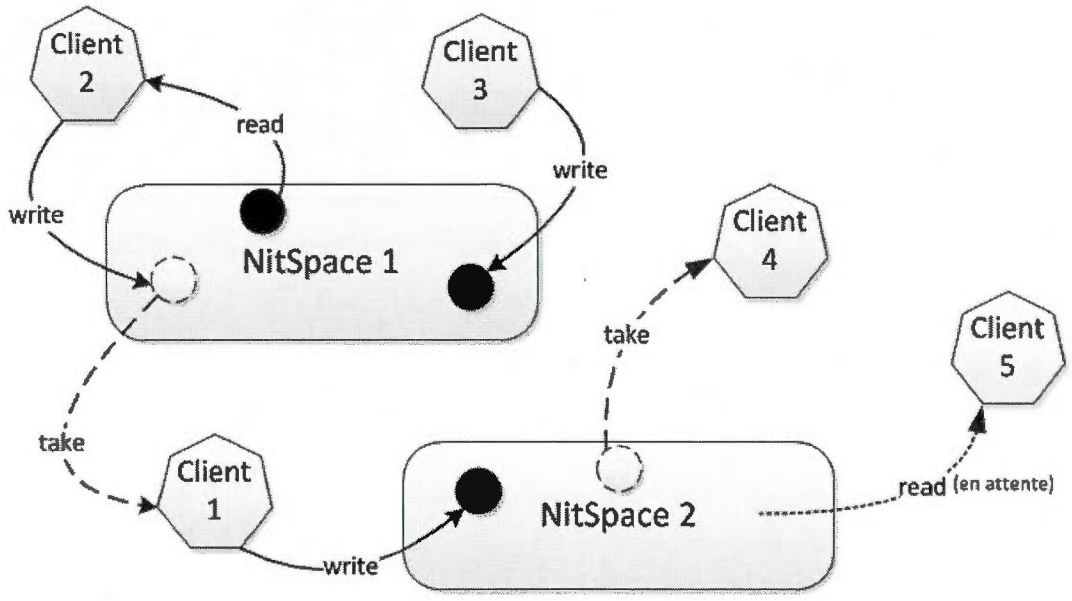

Figure 3.3 Échange et synchronisation entre processus à l'aide de NitSpaces.

C'est l'instruction `fork` qui permet de générer un ou plusieurs nouveaux processus. Elle permet de dupliquer le processus courant en une ou plusieurs copies, le nombre de copies devant être spécifié en paramètre. L'instruction retourne le numéro de la copie, ou 0 pour le processus qui est à l'origine de l'appel. Lors du traitement de l'instruction `fork`, c'est le processus dans son ensemble qui est dupliqué, incluant l'ensemble de ses variables. C'est de cette façon que deux processus Nit peuvent se transmettre une référence sur le même NitSpace. Par la suite, une fois que les processus ont une référence sur le même NitSpace, ils peuvent échanger de l'information par l'entremise de celui-ci.

```
var nitSpace = new NitSpace
var proc0 = (fork(4) == 0)
nitSpace.write( "Identification", "mon pid = " + getpid.to_s )

if proc0 then
    for i in [0..4] do
        var t = nitSpace.take( "Identification", nsString )
        print t[1].to_s
    end
    nitSpace.stop
end
```

Listage 5: Programme Nit illustrant le fonctionnement de `fork`.

Par exemple, soit l'extrait de Programme Nit présenté dans le listage 5. Son exécution produira alors le résultat suivant :

```
mon pid = 2627
mon pid = 2629
mon pid = 2630
mon pid = 2628
mon pid = 2625
```

C'est lors du démarrage du NitSpace que l'information nécessaire pour ouvrir un lien de communication est définie, il est donc important d'instancier le NitSpace avant l'instruction `fork` afin que les nouveaux processus soient en mesure de communiquer avec lui. Dans l'exemple précédent, on crée quatre nouveaux processus qui utiliseront le NitSpace pour s'identifier. Ici, les processus utilisent l'instruction `getpid` pour obtenir leurs numéros d'identification du système d'exploitation, qu'ils utilisent pour ajouter leurs identifications dans le NitSpace. Ensuite le processus de départ, celui dont l'instruction `fork` retourne la valeur 0, récupère l'identification des processus à partir du `NitSpace` et les affiche à l'écran.

Un seul processus doit appeler l'opération de fermeture du NitSpace (`stop`). Il est important de s'assurer que les autres processus ont terminé leur traitement avant d'appeler cette opération, sinon, un processus qui effectuerait une opération de lecture, après sa fermeture, resterait bloqué sur l'attente d'une réponse du NitSpace. Dans le dernier exemple, la lecture à l'aide de l'opération bloquante `take` assure que les autres processus ont terminé leur écriture.

Cette opération sert implicitement de barrière pour s'assurer que tous les processus ont terminé leur exécution. La plupart du temps, il faut explicitement ajouter des instructions de contrôle afin de coordonner le déroulement du programme. Typiquement, cela se limite à un `write` des processus qui effectuent le traitement et d'un `take` d'un processus coordonnateur, tel qu'illustré dans le listage 6.[4]

```
if fork(4) != 0 then
    ...
    nitSpace.write( "Control", "TERMINE" )
else
    ...
    for i in [1..4] do
        nitSpace.take( "Control", "TERMINE" )
    end
    nitSpace.stop
end
```

Listage 6: Programme Nit illustrant la terminaison correcte d'un programme avec une série de processus.

3.6 Conclusion

Comme il a été démontré dans ce chapitre, l'API fournie par notre module `NitSpace` permet à des processus parallèles d'interagir avec un espace partagé, le NitSpace. L'API fournit les cinq opérations de base — `write`, `take`, `take_no_wait`, `read`, et `read_no_wait` — qui permettent de communiquer avec un NitSpace. Nous avons démontré qu'il est possible, à l'aide des objets `Tuple` et `Template` combinés aux cinq opérations de base, de réaliser les paradigmes de Linda tels que présentés dans le chapitre 1.

Ce qui différencie davantage notre API des autres, c'est qu'aucune configuration n'est requise pour son utilisation. Il n'est pas nécessaire de démarrer à l'avance des processus qui gèrent les espaces de tuples à utiliser, tout est lancé à partir du programme principal. Pour démarrer un espace de tuples, il suffit d'instancier un objet `NitSpace`. Pour démarrer un processus client en parallèle, il suffit de dupliquer le processus en cours avec l'instruction `fork`.

---

[4] Le chapitre 1 sur Linda présente d'autres exemples d'algorithme de synchronisation avec un espace de tuples.

Cette capacité de dupliquer un processus en plusieurs exemplaires en partageant du même coup ses références vers des `NitSpace` existants est un procédé unique à notre implémentation. Sous les autres API, il est nécessaire d'appeler une fonction, d'un programme externe au processus client, qui retourne la référence à l'espace de tuples qui aura été identifié par la chaîne de caractères passée en paramètre. De plus, sous les autres implémentations, il n'est pas possible pour un client d'indiquer à un processus que l'espace de tuples qu'il gère n'est plus utilisé et, conséquemment, qu'il peut terminer son exécution.

Une autre caractéristique de NitSpace est l'interface `NSComplexType`, qui permet de définir de nouveaux types pouvant être utilisés dans les tuples et les modèles de tuples. L'interface `NSComplexTYpe` utilise des méthodes abstraites, car le langage Nit n'est pas pourvu d'un mécanisme de réflexion, comme Java, qui lui permettrait de déduire les valeurs de ses attributs devant servir de paramètres au tuple généré. Les méthodes abstraites obligent donc le programme à définir ses méthodes de sérialisation de l'objet au tuple et du tuple à l'objet.

L'implémentation de cette API n'est pas simple à réaliser. Plusieurs choix ont dû être faits pour permettre sa réalisation, et le prochain chapitre en explique les différents aspects.

CHAPITRE IV

NITSPACE — IMPLÉMENTATION

L'implémentation du module `nitspace` fut réalisée en trois couches. La première couche est l'API développée entièrement en Nit. Cette couche correspond à l'ensemble des classes de l'API qui a été décrite dans le chapitre précédent. La couche médiane effectue le transfert entre le module Nit et l'espace de tuples. La troisième couche, développée en C, est l'implémentation de l'espace de tuple représentant un NitSpace et les fonctions clientes qui permettent d'interagir avec lui. La figure 4.1 présente la structure générale, *statique*, de ces trois couches, alors que les figures 4.2 et 4.3 illustrent l'aspect *dynamique* en présentant les séquences d'appels effectués lors de l'exécution d'opérations `write` et `take` sur un NitSpace.

4.1 La couche Nit

4.1.1 Hiérarchie des types

Plusieurs nouvelles classes ont été introduites par le module `nitspace`. Une hiérarchie de types a dû être élaborée afin de structurer ces nouveaux types de manière à permettre une vérification lors de la compilation. En plus des nouveaux types introduits, un raffinement des types déjà existants fut nécessaire. Le type `String` et les types universels `Int` et `Float` ont dû spécialiser un des nouveaux types définis. Sachant qu'en Nit un type universel peut spécialiser uniquement une interface, l'organisation des types du module `nitspace` est donc réaliséee à partir d'interface.

Le premier type, l'interface `NSType`, représente l'ensemble des types impliqués dans le module `nitspace`. Celui-ci est spécialisé en deux types : l'interface `NSActualType` pour les types réels et l'interface `NSFormalType` pour les types formels. Les types `String`, `Int` et `Float`

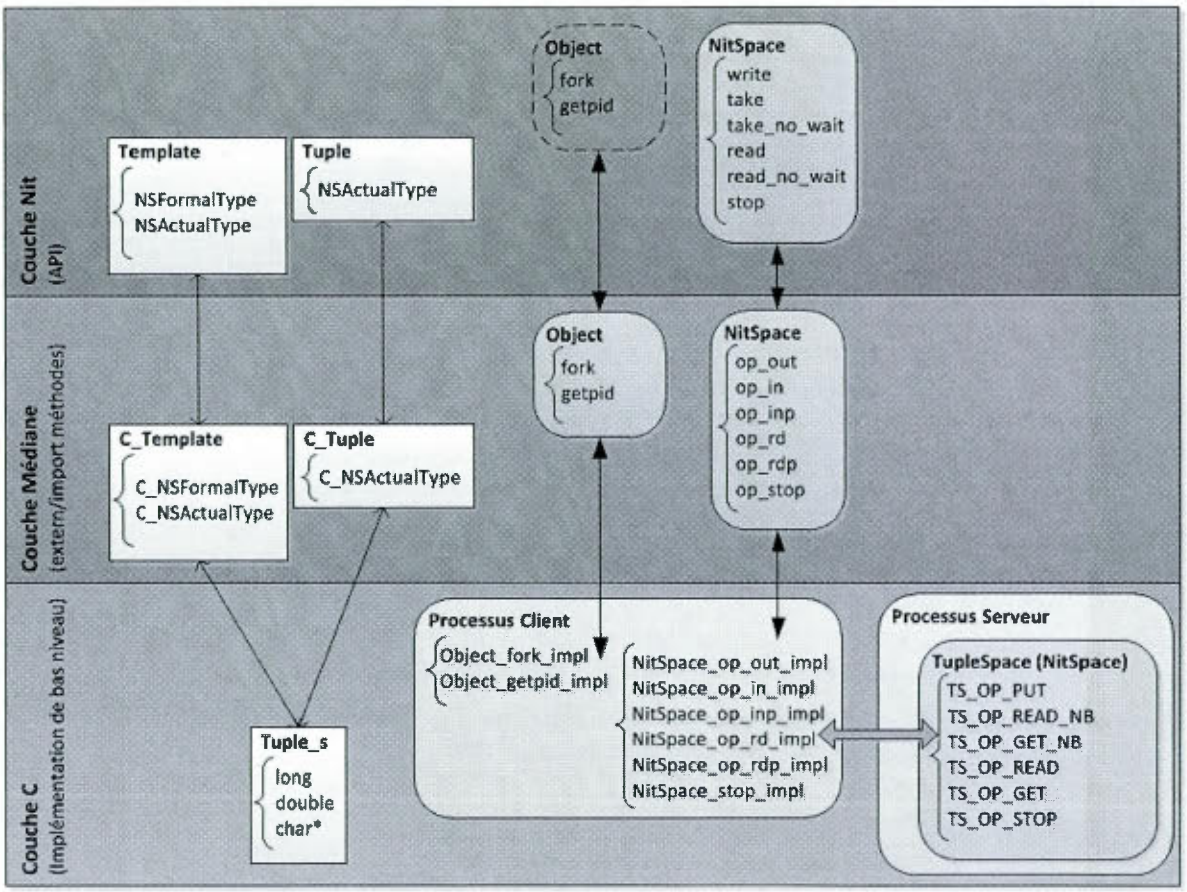

Figure 4.1 Les trois couches de l'implémentation du module `nitSpace`.

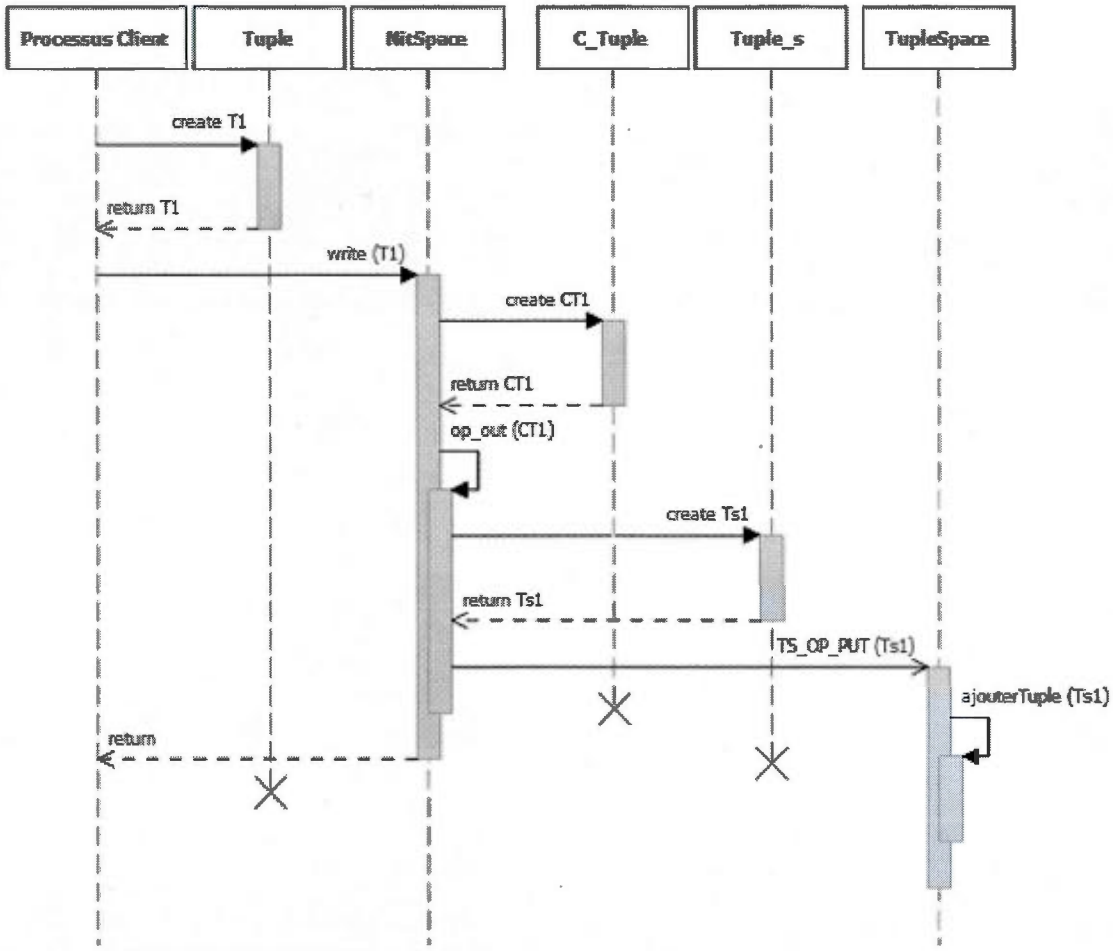

Figure 4.2 Séquence d'appels lors d'une opération `write` sur un `NitSpace`.

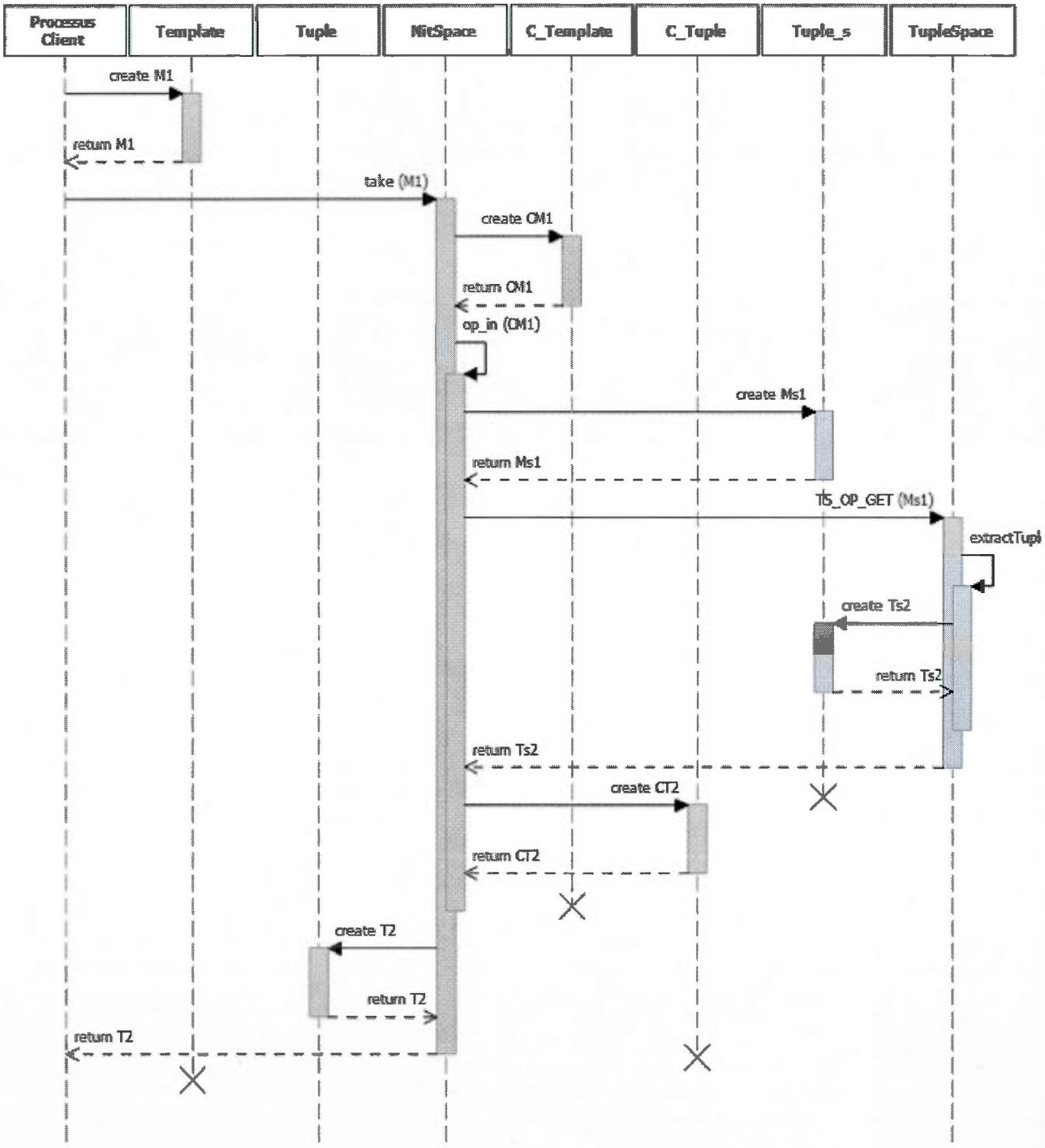

Figure 4.3 Séquence d'appels lors d'une opération `take` sur un `NitSpace`.

sont redéfinis pour spécialiser l'interface `NSActualType`. Pour la représentation formelle de ces types, trois nouvelles classes (`NSString`, `NSInt` et `NSFloat`) qui spécialisent `NSFormalType` sont ajoutées. Un raffinement à la classe `Object` permet d'obtenir des instances des types formels grâce aux méthodes `nsString`, `nsInt` et `nsFloat` qui ont été ajoutées. Puisque ces méthodes ont été ajoutées dans la classe `Object`, elles sont accessibles à partir de n'importe quel endroit du programme.

Selon leur utilisation, les types complexes peuvent représenter des valeurs réelles ou bien des valeurs formelles. Par conséquent, l'interface des types complexes `NSComplexType` spécialise les deux interfaces `NSActualType` et `NSFormalType` — voir figure 4.4.

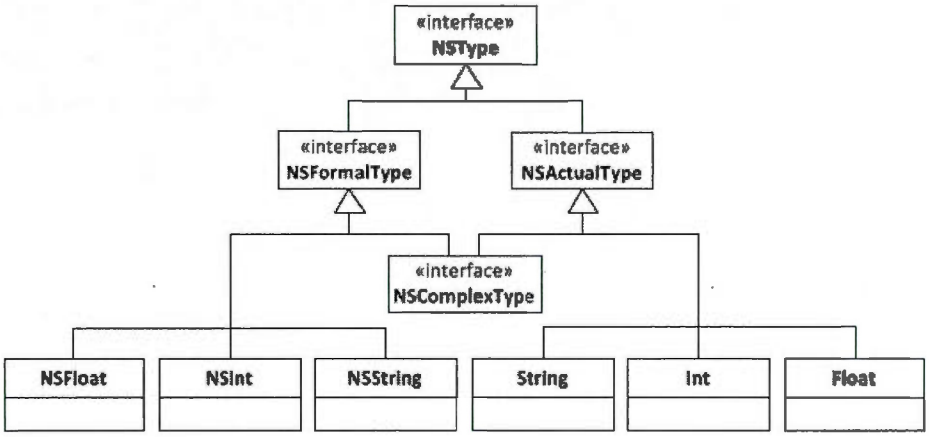

Figure 4.4 Hiérarchie des types.

4.1.2 Tuple et Template

L'implémentation de la classe `Tuple` et de la classe `Template` est réalisée par l'entremise d'une classe générique abstraite appelée `BasicTuple`. Cette classe utilise une collection de type `List` pour conserver la liste des éléments du tuple. Le type générique de la classe précise le type des éléments pouvant être insérés dans la liste. Chacune des opérations est déclarée en utilisant ce type générique. La liste des opérations disponible sur un `BasicTuple` est présentée dans le tableau 4.1.

Puisqu'un tuple doit contenir uniquement des types réels, la classe `Tuple` spécialise la classe `BasicTuple` en définissant son type générique en `NSActualType`. Puisque toutes les opérations sont déjà définies dans la classe abstraite `BasicTuple`, l'implémentation de la classe

```
# Obtient l'élément situé à la position indiquée par l'index.
fun get_Element(index: Int): T

# Ajoute un élément à la fin du tuple.
fun add_Element(elem: T)

# Ajoute une liste d'éléments.
fun add_ListElement(elems: List[T])

# Obtient le nombre d'éléments dans le tuple (ou template).
fun length: Int

# Détermine si le tuple est vide, c.-à-d., s'il ne contient aucun élément.
fun is_empty: Bool

# Accède directement a un élément par son index.
fun [](i: Int): T

# Affecte directement un élément par son index.
fun []=(i: Int, elem: T)

# Retourne les éléments du tuple (ou template) sous forme de liste.
fun to_list: List[T]

# Retourne les éléments du tuple (ou template) sous forme de tableau.
fun to_array: Array[T]
```

Tableau 4.1 Les diverses opérations associées au type `BasicTuple`.

`Tuple` ne demande que l'ajout des définitions des constructeurs. De plus, afin de permettre à un tuple d'être spécifié dans les opérations d'écriture d'un `NitSpace`, la classe `Tuple` implémente également l'interface `NSActualType`.

```
class Tuple
    super BasicTuple[NSActualType]
    super NSActualType
    ...
end
```

Le modèle de tuple, c'est-à-dire la classe `Template`, doit permettre de contenir des éléments de type formels en plus des types réels. La classe `Template` spécialise donc la classe `BasicTuple` en définissant son type générique en `NSType`, correspondant au type au sommet de la hiérarchie des types du module `nitspace`. En plus des opérations déjà définies dans la classe abstraite `BasicTuple`, l'implémentation de la classe `Template` a exigé des opérations supplémentaires pour faciliter sa manipulation. En plus des constructeurs, les opérations `add_StringElement`, `add_IntElement` et `add_FloatElement` sont ajoutées à la classe `Template` afin de faciliter l'ajout d'éléments à partir de variables. Chacune de ces fonctions vérifie que le paramètre possède une valeur ou pas et ajoute la valeur réelle ou le type formel en conséquence.

De plus, afin de permettre à un modèle d'être spécifié dans les opérations de lectures d'un `NitSpace`, la classe `Template` implémente également l'interface `NSType`.

4.1.3 NitSpace — L'espace de tuples

Bien que l'implémentation d'un NitSpace soit principalement faite en C, dans la couche Nit on doit quand même définir une classe qui représente le NitSpace avec la liste de ses opérations. Comme nous allons le voir dans les couches inférieures, tout ce que la classe `NitSpace` a besoin de connaître pour échanger avec le NitSpace est le `pid` du processus qui gère l'espace de tuples. Autrement dit, un NitSpace en Nit se limite à la définition suivante :

```
class NitSpace
    private var pid: Int
end
```

À cette définition on ajoute les cinq opérations qui permettent d'interagir avec le NitSpace,

```
# Écrit un tuple dans le NitSpace
fun write(elems: NSActualType...)

# Retire un tuple du NitSpace
fun take(elems: NSType...): Tuple

# Retire un tuple du NitSpace, mais sans blocage
# Retourne null s'il n'y a pas de correspondance dans le NitSpace.
fun take_no_wait(elems: NSType...): nullable Tuple

# Lit un tuple dans le NitSpace
# La copie du tuple demeure dans le NitSpace (pas retirée)
fun read(elems: NSType...): Tuple

# Lit un tuple dans le NitSpace sans blocage
# Retourne null s'il n'y a pas de correspondance
# La copie du tuple demeure dans le NitSpace (pas retirée)
fun read_no_wait(elems: NSType...): nullable Tuple
```

Tableau 4.2 Opérations permettant d'interagir avec un NitSpace.

présentées dans le tableau 4.2.

Toutes ces opérations effectuent sensiblement le même traitement. Tout d'abord, les arguments donnés sont vérifiés ; lorsqu'un seul paramètre est fourni et que ce paramètre est un objet de type `Tuple` ou `Template`, on l'utilise directement ; dans les autres cas, on appelle le constructeur `with_array` avec la liste des paramètres pour obtenir l'instance correspondante. Une fois que l'on possède le tuple ou le modèle de tuple demandé, on relègue le traitement à l'opération équivalente, mais au niveau de la couche médiane, qui le relèguera à son tour à la couche inférieure développée en C. Une fois que la valeur de retour est obtenue, suite au traitement par les couches inférieures, on retourne tout simplement le résultat sous forme d'un tuple. Dans le cas des opérations non bloquantes, on vérifie d'abord si le tuple reçu est vide, c'est-à-dire s'il ne possède aucun élément, et si c'est le cas, on retourne la valeur `null`.

L'introduction des types complexes a demandé quelques ajustements dans les opérations de lectures. Afin de conserver les types complexes, on déclare, au début de chaque opération, un

dictionnaire de type `HashMap[String, NSComplexType]`. À la création du modèle de tuple pour la couche C, chaque type complexe présent dans le modèle de tuple est ajouté au dictionnaire avec comme clé le nom du type retourné par la méthode `nsTypeName`. Lors de l'instanciation du `Tuple` de retour, un balayage des éléments du tuple est effectué afin de retracer les types complexes. Lorsqu'un type complexe est détecté, le dictionnaire est utilisé pour retrouver l'instance du type complexe utilisé afin d'être en mesure d'appeler la méthode `nsNewObject` qui permettra d'ajouter au tuple une nouvelle instance du type complexe à partir des éléments retournés par la couche inférieure.[1]

4.2 La couche médiane : de Nit vers C

L'interface native de Nit est utilisée afin de permettre l'interopérabilité entre la couche Nit et la couche C. «L'interface native de Nit sert à intégrer du code C à un logiciel Nit, de façon à invoquer des fonctions C depuis Nit et manipuler les objets Nit depuis le code C.» [10]. Les méthodes dites externes sont déclarées en Nit à l'aide de leur signature, mais le corps de la méthode est implémenté en C — elles permettent donc à du code Nit d'appeler des fonctions écrites en C. Inversement, les méthodes qui sont indiquées comme importées sont implémentées entièrement en Nit, mais du code C supplémentaire est généré pour que leurs invocations soit également possibles à partir de code écrit en C — elles permettent donc à du code C d'appeler des méthodes écrites en Nit. Les méthodes externes permettent de passer de la couche Nit à la couche C, tandis que les importations en C des méthodes Nit permettent à la couche C d'interagir avec la couche Nit. De nouvelles classes qui servent strictement à effectuer le passage de Nit vers C ont été ajoutées.

4.2.1 C_Tuple et C_Template

Tout d'abord, un tuple en C n'est pas aussi complexe qu'un tuple en Nit. Le tuple en C peut être composé uniquement de types de base, c'est-à-dire des éléments de type `String`, `Int` ou `Float`. De plus, la couche C utilise un caractère qui, selon son format, permet de distinguer les paramètres réels des paramètres formels. Ainsi, les caractères `s`, `i` et `f` identifient les paramètres réels de type `String`, `Int` et `Float`, tandis que les caractères `S`, `I` et `F` identifient leurs équivalents formels. Par la suite, le modèle de tuple est construit en juxtaposant la suite

[1] Le mécanisme permettant de recréer des instances de type complexe est expliqué à la section 4.2.2.

des caractères retournés par chacun de ses éléments. Pour isoler ces distinctions, une nouvelle interface `C_NSTupleParameter`, ainsi que les classes `C_Tuple` et `C_Template`, ont été ajoutées pour permettre la transition entre la couche Nit et la couche C.

L'interface `C_NSTupleParameter` permet d'isoler les types pouvant être transmis dans les tuples de la couche C. La méthode abstraite `get_NSType` est ajoutée, à cette interface, afin d'obtenir le caractère qui identifie l'élément.

Bien que seul le caractère qui est retourné par la méthode `get_NSType` permet d'identifier le type de l'élément, pour des fins de compilation en Nit, l'interface `C_NSTupleParameter` est spécialisée en deux interfaces afin de distinguer les types formels des types réels — voir figure 4.5. L'interface `C_NSActualType`, qui spécialise `NSActualType` et `C_NSTupleParameter`, permet d'isoler les types réels `String`, `Int` et `Float`. L'interface `C_NSFormalType`, qui spécialise `NSFormalType` et `C_NSTupleParameter`, permet d'isoler les types formels `NSString`, `NSInt` et `NSFloat`.

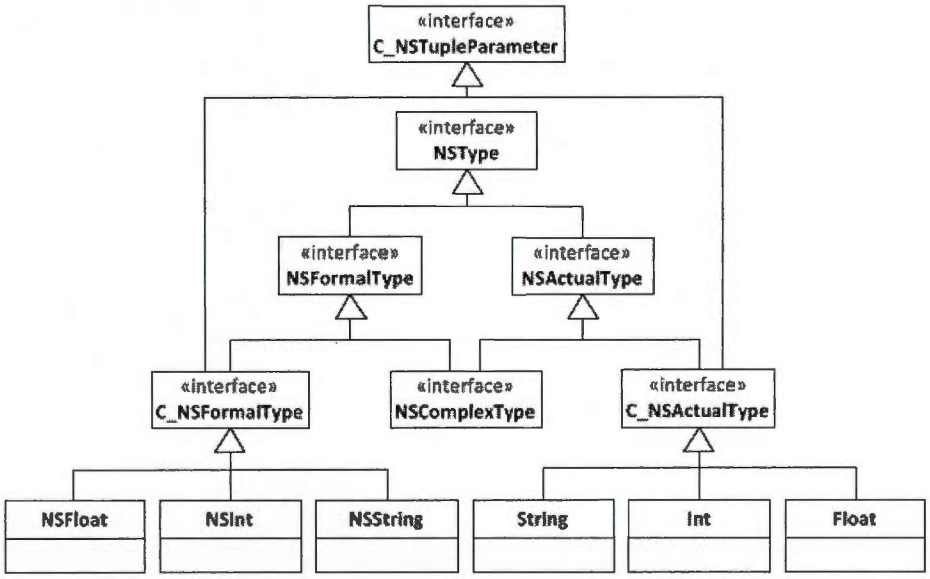

Figure 4.5 Hiérarchie des types incluant les types de transfert vers la couche C.

La classe `C_Template` permet de contenir un modèle de tuple pour la couche C. Elle spécialise la classe `BasicTuple` en définissant son type générique en `C_NSTupleParameter`. Elle introduit la méthode `get_strFormat` qui permet de récupérer le format du modèle en concaténant les caractères qui sont récupérés en appelant la méthode `get_NSType` de chacun de ses éléments.

En plus de la méthode `get_strFormat`, les méthodes `get_StringElement`, `get_IntElement` et `get_FloatElement` introduites dans la classe `BasicTuple` sont également importées dans la couche C afin de lire les éléments contenus dans le modèle. Ces méthodes sont utilisées conjointement pour lire les éléments d'un modèle, la méthode `get_strFormat` dicte quelle méthode doit être appelée pour récupérer les éléments à partir de la couche C.

Le but de la classe `C_Template` et de ses méthodes est de pouvoir convertir un modèle de tuple Nit en sa structure équivalente qui pourra être manipulée en C. Pour comprendre la façon dont la conversion est effectuée, il faut comprendre la structure C qui est utilisée.

Dans la couche `C`, toutes les chaînes de caractères sont utilisées à l'intérieur d'une structure appelée `HString`. Cette structure permet de représenter une chaîne de caractères avec sa valeur de hachage. Une valeur de hachage est utilisée dans l'espace de tuple afin d'accélérer la recherche et la comparaison des éléments. Bien que la valeur de hachage soit ignorée du côté client, on doit la mentionner, car elle fait partie de la définition de la structure de tuple qui est partagée entre l'espace de tuple et un processus client :

```
// Représentation en C d'une chaîne de caractères avec sa valeur de hachage.
typedef struct HString
{
    unsigned int hashValue;
    char* charValue;
} HString_s;
```

En C, il n'y a pas de distinction entre un tuple et un modèle de tuple : c'est exactement la même structure qui est utilisée pour représenter les deux. Sa représentation est constituée d'une chaîne de caractères représentant le format de ses éléments et d'un tableau contenant la listes des éléments. Le listage 7 présente les types utilisés.

Pour représenter les éléments du tableau, on utilise un type `union` qui comporte trois types de pointeur, un pour chaque type de base que l'on peut utiliser dans un tuple. La chaîne de caractères contenue dans l'attribut `template` correspond à la chaîne de caractères retournée par la méthode Nit `get_strFormat` de notre classe `C_Template`. C'est cette chaîne qui fournit les directives de lecture des éléments du tuple. L'index du caractère correspond à l'index du tableau et chacun de ces caractères nous indique quel est le type de l'élément, c'est-à-dire quel pointeur utiliser pour accéder à l'élément du tableau. Implicitement, le nombre de caractères

```
// Représentation d'un Template pour les tuples.
typedef HString_s cTemplate;

// Élément simple d'un tuple.
typedef union {
    long    lg;
    double  dbl;
    HString_s str;
} Tuple_element_s;

// Représentation d'un Tuple.
typedef struct {
    cTemplate template;
    Tuple_element_s* elements;
} Tuple_s;
```

Listage 7: Types C utilisés pour la représentation des tuples et des templates.

du `template` indique le nombre d'éléments contenus dans le tuple.

Pour faciliter le transfert entre un objet `C_Template` et un tuple contenu dans une structure `Tuple_s`, une fonction de conversion a été ajoutée. La fonction `Template_to_cTuple_s` reçoit un `C_Template` en paramètre, le convertit et retourne une structure `Tuple_s`. Cette méthode est appelée systématiquement dans toutes les opérations de lecture.

Pour une opération d'écriture, ainsi que pour le tuple qui est retourné lors d'une lecture, c'est la classe `C_Tuple` qui est utilisée. Cette classe spécialise la classe `BasicTuple` en utilisant `C_NSActualType` comme type générique. Tout comme la classe `C_Template`, la méthode `get_strFormat` est introduite et elle est importée en C tout comme les méthodes `get_StringElement`, `get_IntElement` et `get_FloatElement`. Pour l'opération d'écriture, la méthode `Tuple_to_cTuple_s`, similaire à la méthode `Template_to_cTuple_s`, permet de convertir le `C_Tuple` en une structure `Tuple_s` qui pourra être utilisée dans les opérations de la couche C.

La classe `C_Tuple` se distingue de la classe `C_Template` par son utilisation comme valeur de retour. Les méthodes `add_StringElement`, `add_IntElement` et `add_FloatElement` sont ajoutées et importées afin de permettre l'ajout d'éléments au `C_Tuple` à partir de la couche

C. Un constructeur, nommé `with_empty`, a aussi été ajouté afin d'en permettre l'instanciation à partir des opérations de lecture de la couche C. Une fonction de conversion d'une structure `Tuple_s` vers un `C_Tuple`, appelé `cTuple_s_to_Tuple`, utilise ces méthodes et permet, suite aux opérations de lecture sur un NitSpace, de construire le `C_Tuple` à retourner.

4.2.2 Transfert des types complexes

La conversion d'un objet de type `Tuple` ou de type `Template` vers leurs équivalents pour la couche C, c'est-à-dire en `C_Tuple` ou `C_Template`, est relativement simple. Cependant, cette simplicité est perdue lorsqu'il y a des types complexes dans les éléments qu'ils contiennent. Pour les types de base c'est une copie directe, du un pour un ; par contre, les types complexes doivent être manipulés pour être supportés dans la couche C.

L'implémentation de l'interface `NSComplexType` exige que le programmeur définisse la fonction `nsTemplate`. Cette fonction fournit déjà un modèle de tuple avec les éléments qui représentent l'état de l'instance de la classe. Par contre, il faut s'assurer qu'à l'intérieur d'un NitSpace les éléments d'un type complexe ne soient pas confondus avec les éléments d'un autre type complexe ou avec une liste d'éléments simples dont les types des éléments seraient les mêmes. Pour pallier ce problème, les éléments du type complexe sont balisés entre deux chaînes de caractère qui identifie le début et la fin des éléments d'un type complexe.

Après sa conversion, le premier élément d'un type complexe est la chaîne de caractères `_NSComplexType:Start_`, son dernier élément est la chaîne de caractères `_NSComplexType:Stop_`. Entre ces deux chaînes, on retrouve la définition des éléments du type complexe. Après la chaîne de début, on ajoute le nom du type tel que défini avec la fonction `nsTypeName`. Il est important que ce nom soit unique, car il pourrait avoir des conflits d'identification de types complexes. Ensuite, le second élément représente le nombre d'éléments contenus dans le type complexe, c'est-à-dire le nombre d'éléments contenus dans le modèle retourné par la fonction `nsTemplate`. Les éléments du modèle défini par `nsTemplate` sont ensuite insérés directement dans l'objet `C_Template` résultant. Pour faciliter sa conversion, la fonction `to_C_Template` est définie dans la classe `NSComplexType`.

Par exemple, si on effectue la conversion d'une instance du type complexe `Message`[2] avec,

---

[2]La déclaration de classe du type complexe `Message` a été expliquéee à la section 3.4,

comme attributs, les valeurs suivantes :

```
content = "Bonjour!";
senderID = 4;
```

On aurait alors comme résultat un `C_Tuple` équivalent au tuple suivant :

```
Tuple("_NSComplexType:Start_", "Message", 2, "Bonjour!", 4, "_NSComplexType:Stop_");
```

Lorsqu'on consulte un NitSpace, les éléments contenus dans le `C_Tuple` qui est retourné par une opération de lecture sont balisés. Les informations sur les types complexes sont reçues telles qu'elles sont envoyées. On doit donc détecter les types complexes et effectuer le traitement inverse pour instancier les objets du type complexe. Pour y arriver, au début d'une opération de lecture sur un NitSpace, un dictionnaire d'objet (`HashTable`) est créé. Lors de la conversion du `Template` vers le `C_Template`, tous les objets de type complexe y sont ajoutés avec le nom du type (`nsTypeName`) comme clé de recherche. Ce dictionnaire d'objet est conservé et est utilisé lors de l'opération inverse, c'est-à-dire de la conversion d'un `C_Tuple` vers un `Tuple` Nit. Cette conversion est réalisée par le constructeur `with_C_Tuple` qui a été ajouté à la classe `Tuple`. Ce constructeur reçoit deux paramètres : le `C_Tuple` transmis par le NitSpace et le dictionnaire d'objet complexe qui avait été conservé. Le constructeur consulte chaque élément avant de les ajouter dans sa liste d'éléments, et lorsqu'il détecte des balises de type complexe, il effectue le traitement d'instanciation du type avant d'ajouter celui-ci comme un nouvel élément de sa liste.

Pour instancier un objet de type complexe, le constructeur lit le nom du type, élément qui suit la balise de début, ce qui lui permet de récupérer l'objet du même type dans le dictionnaire de référence d'objet complexe. Ensuite, on obtient le nombre d'éléments contenus dans le type complexe en lisant l'élément suivant. Le nombre obtenu indiquera le nombre d'éléments successifs à récupérer pour créer un tuple temporaire. Finalement, on appelle la fonction `nsNewObject` de l'objet obtenu du dictionnaire avec comme paramètre le tuple temporaire, ce qui permet d'obtenir une nouvelle instance du type complexe, mais initialisée avec ces nouvelles valeurs. La nouvelle instance du type complexe est ensuite ajoutée dans sa liste d'éléments avant de poursuivre la lecture des éléments restants du `C_Tuple`.

4.2.3 Transfert des traitements

Toutes les opérations effectuées sur un NitSpace traitent les tuples avec le mécanisme de conversion expliqué précédemment. L'objet de type `Template` est converti vers un objet de type `C_Template` en conservant des références sur les éléments de type complexe. Ensuite, le traitement de l'opération est transféré vers la couche C. Et pour terminer, on reçoit de la couche C un objet `C_Tuple` que l'on convertit en `Tuple` à l'aide des références de type complexe obtenues précédemment.

Pour réaliser le transfert du traitement de l'opération vers la couche C, toutes les opérations qu'on peut effectuer sur un NitSpace possèdent leurs équivalents de bas niveau qui servent de relais vers la couche C. Ces procédures sont déclarées externes, car elles sont implémentées directement en C. Une fois qu'elles sont appelées, le reste du traitement se déroule exclusivement dans la couche C jusqu'à ce que l'opération retourne une valeur résultante.

Les cinq opérations de bas niveau sont les suivantes :

- `op_out(C_Tuple)` : appelée par l'opération `write`.
- `op_in(C_Template)` : appelée par l'opération `take`.
- `op_inp(C_Template)` : appelée par l'opération `take_no_wait`.
- `op_rd(C_Template)` : appelée par l'opération `read`.
- `op_rdp(C_Template)` : appelée par l'opération `read_no_wait`.

En plus des cinq opérations précédentes, on retrouve aussi trois autres méthodes externes dont l'implémentation est réalisée en C. La première méthode est l'opération `stop` sur un NitSpace. Ensuite, on a la fonction `fork` et la fonction `getpid` qui ont été ajoutées à la classe `Object` de manière à ce que leurs portées soient globales. Toutes ces méthodes externes permettent de reléguer la suite du traitement à la couche C.

4.3 La couche C

On peut diviser cette couche en trois parties. Tout d'abord, l'implémentation en C des méthodes appelées à partir de Nit représente la partie du client. toutes ces méthodes permettent d'interagir avec d'autre processus, que ce soit pour générer d'autre processus clients ou

pour communiquer avec un NitSpace. La seconde partie correspond à l'implémentation d'un espace de tuples représentant un NitSpace. Les structures et fonctions utilisées par un processus qui gère un espace de tuples représentent la partie serveur de l'implémentation. On retrouve également une partie commune qui fournit les structures et fonctions nécessaires pour l'échange d'information entre les processus client et un processus serveur.

4.3.1 Communication entre processus

La communication entre deux processus s'effectue par l'entremise de *sockets*. Pour réaliser les échanges, on utilise les *sockets* sous le domaine de communication locale, identifié par la constante `AF_UNIX`. Le domaine de communication locale limite les communications entre les processus résidant sur la même machine. La communication s'effectue entièrement dans le noyau du système d'exploitation. De ce fait, il est possible d'échanger des datagrammes en mode non connecté et ce de manière toujours fiable. Cette approche était la plus adéquate puisque le processus qui gère l'espace de tuples est constamment en attente de requêtes de la part des processus clients et certaines requêtes sont même mises en attente pendant qu'il répond à de nouvelles requêtes émises par d'autres clients.

L'utilisation de communications en mode non connecté exige que chaque *socket* soit nommé afin d'être rejoint par les autres processus. L'identification de l'adresse du *socket* peut être définie comme un nom dans le système de fichier ou comme une chaîne unique dans l'espace de nom abstrait. Nous avons préféré l'utilisation de l'espace de nom abstrait, car elle ne laisse aucune trace des allocations non libérée contrairement au système de fichiers où un fichier représentant le *socket* reste présent sur le disque lorsqu'il n'est pas libéré explicitement. L'espace de nom abstrait nous a permis de retirer l'appel explicite d'une procédure de fermeture de *socket* à la fin de l'exécution de chacun des processus.

Afin de rendre l'initiation du *socket* également transparente pour le programmeur, on appelle la procédure d'initialisation automatiquement de deux manières différentes, une pour chacune des façons qu'un processus client peut démarrer. Tout d'abord, lorsqu'un processus démarre un NitSpace, la procédure vérifie si son *socket* est déjà défini et, dans le cas contraire, initialise un nouveau *socket*. Cette procédure va s'assurer que le processus de départ est prêt à communiquer dès la création d'un NitSpace. Pour les autres processus, c'est dès leur création, lors de l'appel du `fork`, que leurs communications sont initialisées. De plus, cette façon permet

également de remplacer la définition du *socket* du parent qui a été dupliqué dans le processus enfant lors de l'appel du `fork`.

Lors de l'initialisation, le *socket* créé est directement lié dans l'espace de nom abstrait avec un nom composé de son `pid`, de cette façon on s'assure d'utiliser toujours des noms uniques dans l'espace de nom abstrait. Pour les processus clients, on utilise le préfixe `cts` avec le `pid` comme suffixe, alors que pour un NitSpace, on utilisera plutôt le préfixe `ts` avec le `pid` comme suffixe.

Puisque le nom est toujours défini à l'aide du `pid` du processus, la seule information conservée lors de l'instanciation d'un NitSpace en Nit est son `pid`. Par la suite, dans la couche C, la fonction `getAddrTupleSpace` récupère ce `pid` de la couche Nit et définit l'adresse du *socket* à rejoindre pour communiquer avec le NitSpace sur lequel l'opération a été invoquée. La fonction `getAddrTupleSpace` permet de définir l'adresse du destinataire d'un message du côté d'un processus client. En ce qui concerne le NitSpace, il obtiendra l'adresse du destinataire, pour répondre au processus client, en lisant l'adresse de l'expéditeur lors de la réception d'un message. Le type utilisé pour un message est donc le suivant :

```
typedef struct {
    int code_op;
    struct sockaddr_un addr;
    Tuple_s tuple;
} message_s;
```

Outre l'adresse du destinataire, un message est aussi constitué du code d'opération et du tuple impliqué dans la requête. Les valeurs possibles pour un code d'opération sont définies à l'aide de six constantes qui déterminent le code de la commande d'arrêt et les cinq types d'opérations, impliquant un tuple, qu'il est possible d'effectuer sur un NitSpace.

```
// Liste des codes des opérations
#define TS_OP_STOP       0
#define TS_OP_PUT       10
#define TS_OP_READ      20
#define TS_OP_READ_NB   21
#define TS_OP_GET       30
#define TS_OP_GET_NB    31
```

Puisque l'échange entre deux processus s'effectue par message sous forme de datagramme, la procédure d'envoi de message, `send_message`, convertit la structure `message_s` en datagramme avant de l'expédier avec la fonction `sendto` qui permet d'écrire directement le datagramme sur le *socket* de destination. La définition du datagramme est divisée en deux sections : l'en-tête du message et le contenu du tuple. L'en-tête du message possède une taille fixe qui est lue dès la réception du message afin de connaître l'objectif du message et la taille du tuple qu'il contient :

```
// Représentation d'un en-tête de message.
typedef struct {
    int code_op;
    int len_fmt;
    int len_msg;
} msg_header_s;
```

L'en-tête d'un message est composé de trois valeurs de type `int` indiquant les informations nécessaires au traitement de la requête effectué sur un NitSpace. La première valeur, `code_op`, indique l'opération qui est demandée. La deuxième valeur, `len_fmt`, indique le nombre de caractères inclus dans le format du tuple. La troisième et dernière valeur, `len_msg`, indique la longueur du datagramme en octets. Tout juste après l'en-tête suivra la chaîne de caractères qui spécifie le type des éléments du tuple. Le nombre de caractères contenu dans cette chaîne est défini sous la valeur `len_fmt` de l'en-tête du message. Par la suite, les éléments contenus dans le tuple sont ajoutés séquentiellement dans le datagramme.

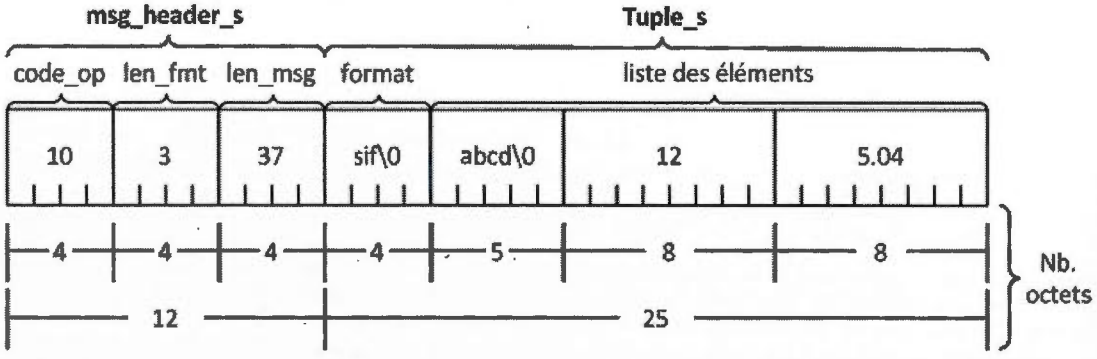

Figure 4.6 Représentation de la structure d'un datagramme.

La figure 4.6 illustre le contenu du datagramme suite à l'opération d'écriture suivante :

```
nitSpace.write("abcd", 12, 5.04)
```

Dans cet exemple, le type `int` est représenté sur 32 bits (4 octets) alors que les types `long` et `double` sont représentés sur 64 bits (8 octets).

4.3.2 Les processus clients

Outre les procédures de transformation des objets Nit en structures de données C, la majeure partie du client est réalisée en Nit. On compte huit fonctions Nit déclarées externe dont l'implémentation a été faite en C. Ces fonctions forment le côté client de l'implémentation.

Les deux premières fonctions, `getpid` et `fork`, permettent de gérer les processus clients, tandis que les autres fonctions permettent d'interagir avec un NitSpace. La fonction `getpid` retourne tout simplement la valeur retournée par la fonction C `getpid` qui permet d'obtenir le numéro d'identification du processus courant.

La version Nit de la fonction `fork` permet un peu plus de flexibilité par rapport à la fonction `fork` disponible en C, car elle permet de spécifier le nombre de fois que le processus courant doit être dupliqué. Puisque chaque processus client possède une variable d'environnement qui représente son *socket* et que celle-ci doit toujours être maintenue initialisée, la fonction doit également s'assurer d'initialiser le *socket* pour chaque nouveau processus.

Puisque le *socket* de chaque processus est déjà initialisé et prêt à communiquer avec un NitSpace, les fonctions restantes, qui permettent d'interagir avec un NitSpace, ne font qu'appeler les fonctions de conversion de types et effectuer l'appel de l'opération. Pour les opérations de lecture, la fonction doit attendre le résultat qui sera retourné par le NitSpace. Dans le cas d'une opération non bloquante, une vérification supplémentaire doit être faite afin de retourner un tuple vide lorsqu'une valeur `null` est retournée par le NitSpace. Le tuple vide sert d'indicateur entre les deux couches pour indiquer une absence de valeur; le tuple vide sera de nouveau converti en valeur `null` dans la couche supérieure de Nit.

4.3.3 Le processus serveur (`NitSpace`)

Le processus d'un NitSpace démarre dès son instanciation en Nit. La procédure `initspace`, implémentée en C, est appelée par le constructeur du NitSpace pour initialiser son environnement d'exécution. La procédure `initspace` crée un nouveau processus à l'aide de la commande `fork`

et retourne, dans la couche Nit, le `pid` du nouveau processus en tant qu'attribut de la nouvelle instance représentant le NitSpace. Ce `pid` permettra de déterminer l'adresse de destination lors d'opérations sur le NitSpace.

```
// Noeud d'une liste de tuple
typedef struct Tuple_node
{
    Tuple_s tuple;
    struct Tuple_node* next;
} Tuple_node_s;

// Liste de tuple
typedef struct Tuple_list
{
    struct Tuple_node* first;
} Tuple_list_s;
```

Listage 8: Structures de données C utilisées pour représenter la liste des tuples.

Pour initialiser son environnement, le NitSpace initialise son *socket* pour communiquer avec les autres processus, et il crée aussi une table de hachage pour conserver les tuples en mémoire. Cette table, appelée la table des listes de tuples, permet d'atteindre directement les listes de tuples qui conservent les tuples en mémoire. Une liste de tuples correspond simplement à une liste chaînée où chaque noeud de la liste possède un tuple et un pointeur sur le noeud suivant. Les insertions dans la liste s'effectuent toujours en tête de liste minimisant ainsi le coût d'insertion d'un tuple. Pour la recherche d'un tuple, on parcourt la liste séquentiellement et le premier tuple trouvé est retourné. La structure de données est présentée dans le listage 8.

Les tuples sont regroupés par modèle où chaque regroupement possède sa propre liste et seuls les tuples qui possèdent exactement le même `template` peuvent y être insérés. On a donc une liste par modèle de tuple. La table des listes de tuples permet de récupérer la bonne liste, correspondant au modèle recherché, en un temps quasi constant. Pour y arriver, on utilise la chaîne de caractères qui représente le `template` du tuple pour obtenir une valeur à l'aide d'une fonction de hachage. La chaîne n'est pas utilisée directement, parce que lors d'une lecture de tuple, la chaîne peut comporter des caractères en majuscule pour indiquer les éléments du tuple qui n'ont pas de valeur. La chaîne représentant le `template` doit être d'abord normalisée avec

uniquement des lettres minuscules, et ensuite on calcule la valeur de hachage sur cette chaîne normalisée. De cette façon, on obtient toujours la même valeur pour l'ensemble des combinaisons possibles d'un même modèle de tuple.

Pour retrouver la liste de tuples correspondant à un modèle, on divise la valeur de hachage du `template` du tuple par le nombre d'éléments de la table qui nous fournira l'index de l'élément du tableau qui contient la liste de tuples recherchés. La table de liste de tuples est créée avec 1009 éléments; ce nombre est arbitraire et semblait être assez grand et approprié pour minimiser le nombre de collisions dans notre table — en plus d'être un nombre premier.

```
// Noeud du HashTable contenant l'ensemble des listes de tuple
// hstr correspond au Template des tuples contenu dans la liste
typedef struct StrHT_node
{
    HString_s hstr;
    Tuple_list_s tuple_list;
    struct StrHT_node* next;
} StrHT_node_s;

// Nombre de noeud StrHT_node_s dans le HashTable
// Valeur assez grande pour éviter les collisions!
#define NUMELEMSTRHT 1009

// HashTable contenant l'ensemble des listes de tuples.
StrHT_node_s StringHashTable[NUMELEMSTRHT];
```

Listage 9: Structures de données C pour la table de hachage des tuples.

Même si le nombre de collisions reste minime, on doit les gérer, car elles peuvent quand même survenir. Pour ce faire, les listes de tuples sont conservées dans des listes où chaque noeud comporte une liste de tuples, le `template` du modèle de tuple conservé dans cette liste et un pointeur sur la prochaine liste de tuples. On retrouvera donc comme élément dans notre table de hachage, des pointeurs vers le premier noeud d'une liste contenant des listes de tuples — voir listage 9.

Une fois démarré, lorsque la table de liste de tuples est créée, le NitSpace commence son traitement en lisant sur son *socket* le prochain message à traiter, il se met en attente

jusqu'à la réception de la prochaine opération. Dès la réception d'un message, avant même de vérifier le type d'opération demandé, le tuple impliqué dans l'opération est traité par la fonction `findHashValue`. Cette fonction permet d'obtenir des valeurs de hachage pour chaque texte contenu dans le tuple, incluant la chaîne de caractères qui détermine le `template` du tuple. Ces valeurs de hachage permettrons de comparer les chaines de caractères entre elles plus rapidement. Pour la chaine représentant le format du tuple, sa valeur de hachage va permettre de déterminer l'emplacement de la liste de tuples dans la table de hachage, c'est-à-dire la table des listes de tuples.

À partir du moment où les valeurs de hachage des chaînes de caractère du tuple ont été définies, on retrouve le pointeur de la liste de tuples correspondant au modèle du tuple impliqué dans l'opération. Dès que le NitSpace obtient ces informations, on vérifie le code de l'opération demandé pour effectuer le traitement nécessaire. Trois fonctions permettent de réaliser les opérations sur une liste de tuple :

- `ajouterTuple(Tuple_s tuple, Tuple_list_s* ptr_tuple_list)` Cette fonction permet d'ajouter en tête de la liste un nouveau tuple.
- `readTuple(Tuple_s partial_tuple, Tuple_list_s* ptr_tuple_list)` Cette fonction recherche séquentiellement le tuple correspondant au `partial_tuple` dans la liste et retourne une copie du tuple recherché, retourne un tuple vide si ne trouve aucun tuple.
- `extractTuple(Tuple_s partial_tuple, Tuple_list_s* p_tuple_list)` Cette fonction recherche séquentiellement le tuple correspondant au `partial_tuple` dans la liste et retourne le tuple trouvé après l'avoir retiré de la liste, retourne un tuple vide si ne trouve aucun tuple.

Pour les opérations de lecture `TS_OP_READ` et `TS_OP_READ_NB`, la fonction `readTuple` sera effectuée sur la liste de tuples. Pour les opérations `TS_OP_GET` et `TS_OP_GET_NB`, c'est la fonction `extractTuple` qui sera utilisée. Finalement, on utilisera la fonction `ajouterTuple` sur la liste pour l'opération `TS_OP_PUT`.

Pour les opérations `TS_OP_READ` et `TS_OP_GET`, une vérification est faite sur le tuple de retour afin de s'assurer que ce n'est pas un tuple vide, ce qui correspondrait à n'avoir trouvé aucun tuple correspondant dans la liste. Dans ce cas, au lieu de retourner le tuple vide au client, le message qui avait été reçu est mis en attente. Les messages en attente sont conservés dans

une liste de messages se rapprochant d'une liste de type FIFO (*First In, First Out*), dans le sens où les messages en attente sont insérés en fin de liste et que le premier message inséré sera le premier traité pour le modèle tuple concerné — voir listage 10.

```
// Noeud d'un message en attente
typedef struct Msg_node
{
    message_s msg;
    struct Msg_node* next;
} Msg_node_s;

// Liste des messages en attente
typedef struct Msg_list
{
    struct Msg_node* first;
    struct Msg_node* last;
} Msg_list_s;
Msg_list_s msg_list;
```

Listage 10: Structures de données C pour conserver les messages en attentes pour les opérations bloquantes.

Lors de l'ajout d'un nouveau tuple, c'est-à-dire lors de l'appel de l'opération `TS_OP_PUT`, la liste des messages en attente est d'abord parcourue afin de vérifier si le tuple qu'on vient d'ajouter ne correspondrait pas au modèle de tuple d'un des messages en attente. Lorsqu'il y a effectivement une correspondance, le tuple est immédiatement retourné au client qui est toujours en attente du tuple précédemment demandé. Une fois le tuple retourné, on retire le message de la liste des messages en attente.

Lorsque le message en attente est un message de lecture sans extraction (`TS_OP_READ`), le traitement se poursuit jusqu'à la fin de la liste des messages en attente et ensuite le tuple est inséré dans la liste de tuple correspondant. Par contre, si le message en attente est une opération d'extraction (`TS_OP_GET`), le traitement s'arrête dès que la réponse est envoyée au client et le tuple est supprimé sans être conservé dans la liste de tuples.

Finalement, lorsque l'opération reçue est la commande d'arrêt `TS_OP_STOP`, le traitement effectué supprime tous les tuples et les listes conservés dans la table de liste de tuples, ainsi que

les messages dans la liste des messages en attente, avant de sortir de la boucle de traitement des opérations et de terminer proprement l'exécution du processus.

4.4 Conclusion

La plus grande partie du travail d'implémentation fut réalisé en C, dans la couche de bas niveau. La partie serveur, c'est-à-dire le `NitSpace` représentant l'espace de tuples, se situe à ce niveau. Le démarrage du processus, l'échange de messages avec les clients, la recherche et la gestion des tuples, toute la partie serveur de l'implémentation est entièrement réalisée en C. La partie client a été développée, toujours en C, pour communiquer avec le serveur. Les premières versions de cette implémentation furent uniquement en C. Ce n'est qu'une fois que notre implémentation C du modèle Linda fut fonctionnelle que nous avons entrepris l'intégration Nit de notre implémentation.

L'utilisation de l'interface native de Nit pour intégrer notre implémentation C a facilité sa réalisation, mais a directement influencé le code Nit généré dans la couche intermédiaire. C'est la partie client de l'implémentation C qui s'est vue intégrée presque intégralement en Nit. Dans ces conditions, les classes `C_Tuple` et `C_Template`, ainsi que les opérations de communication avec le NitSpace, c'est-à-dire les opérations `op_out`, `op_in`, `op_rd`, `op_inp` et `op_rdp`, se sont retrouvées dans la couche médiane. À partir de ce moment, nous avions une implémentation fonctionnelle du modèle Linda accessible à partir du langage Nit.

Par la suite, l'ajout d'une couche de plus haut niveau, correspondant à notre API, a été réalisée dans le but de restructurer les classes qui constituent le client afin qu'elles se rapprochent davantage du langage Nit et, du même coup, permettre de s'éloigner du modèle Linda de base. C'est la raison pour laquelle on retrouve principalement dans cette couche des déclarations de types, du raffinement de classes, ainsi que la mise en place d'une hiérarchie de types afin de permettre une validation par le compilateur Nit. Il serait donc possible de modifier l'API à faible coût, afin d'utiliser des structures plus proches du langage Nit. Par exemple, il serait possible de remplacer les classes `Tuple` et `Template` par un raffinement de la classe `Sequence`, classe bien connue des programmeurs Nit.

Bien que cette approche nous ait permis de réaliser une implémentation fonctionnelle du modèle Linda, le fait est qu'une réalisation du bas vers le haut a inévitablement fait en sorte que la couche de bas niveau a influencé la couche Nit et, implicitement, l'API résultante. Si nous

devions refaire une nouvelle implémentation, une ébauche de l'API avant l'implémentation de bas niveau aiderait à réaliser une API plus proche du langage hôte. Idéalement, il s'agirait de revoir l'API avec des programmeurs ayant une bonne expérience du langage, ce qui aiderait à être plus fidèle à la nature du langage. Malgré que la couche Nit soit maniable, dans le sens où l'on peut restructurer les classes de l'API sans modifier le coeur de l'implémentation en C, il n'en demeure pas moins qu'une modification plus fonctionnelle que structurelle, par exemple l'ajout d'un type booléen dans les types de base, impliquerait que ce type de valeur soit supporté dans la couche de plus bas niveau. Il en serait de même si on voulait supporter l'héritage de types pour la lecture de types complexes, auquel cas une modification importante dans la couche de bas niveau serait nécessaire. Il serait donc judicieux de bien connaître les besoins qui doivent figurer dans l'API de la couche Nit avant d'entreprendre une amélioration dans la couche C de bas niveau.

CONCLUSION

L'ajout de mécanismes qui permettent de générer du parallélisme dans un langage existant n'est pas un problème simple à résoudre. Lorsqu'on modifie un langage tel que Nit, qui ne possède aucune instruction permettant de générer du traitement parallèle, on a l'opportunité de choisir la voie à emprunter pour générer du parallélisme. Cette latitude peut davantage nuire qu'aider, puisqu'il n'existe rien au départ, donc tout est à faire. De plus, le simple ajout d'une instruction peut avoir des répercussions sur des mécanismes internes à Nit déjà existants. Par exemple, la génération d'un nouveau fil d'exécution (*thread*) avec variables partagées exigerait une modification dans l'exécution du *garbage collector* de Nit. Certains mécanismes qui ont été explorés dans d'autres langages semblent intéressants au niveau de leur facilité d'utilisation, mais amènent une complexité d'implémentation puisqu'ils sont fortement couplés avec les mécanismes internes de Nit. Il faut donc considérer que lors d'ajout d'extensions parallèles dans un langage au départ séquentiel, plusieurs facteurs doivent être considérés et certains compromis doivent être faits entre la performance d'exécution, la facilité d'utilisation, la complexité d'implémentation et la nature du langage.

En choisissant d'intégrer une solution sous forme de bibliothèque, nous avons fourni un nouvel objet, appelé NitSpace, qui permet de partager de l'information, sans avoir à modifier le noyau d'instructions du langage. Cependant, l'utilisation d'un NitSpace ne génère aucun parallélisme dans un programme; l'instruction `fork` a donc été ajoutée afin de permettre de dupliquer des processus et ainsi générer des processus concurrents. L'ajout de l'instruction `fork` a été réalisé à l'aide d'une redéfinition de la classe `Object` dans le module NitSpace. Aucune modification dans le compilateur Nit n'a été nécessaire. Les modifications apportées dans le langage Nit sont donc minimales et, conséquemment, la complexité d'implémentation a été réduite. La complexité d'implémentation est sans doute le facteur ayant influencé le plus la solution que nous avons retenue.

Le modèle Linda reste un moyen simple d'échanger de l'information entre processus concurrents. Un espace partagé avec quelques opérations pour l'ajout, l'extraction et la lecture reste un moyen de communication facile à comprendre. On a vu au chapitre 1 qu'il est possible, avec

ce modèle, de représenter différentes structures distribuées en fonction du ou des paradigmes que l'on désire utiliser. Une amélioration possible serait de bonifier la bibliothèque NitSpace afin d'encapsuler la manipulation de ces structures. Ainsi, l'utilisation d'une barrière de synchronisation, ou l'utilisation d'un tableau distribué, pourrait être simplifiée davantage pour le développeur. Mais même sans cette bonification, le modèle Linda reste simple et accessible, et c'est une des raisons pour lesquelles cette solution a été retenue, la simplicité d'utilisation étant le deuxième facteur ayant le plus influencé le choix de l'implémentation.

Bien que ce ne soit pas le facteur le plus important, la cohésion du langage n'a pas été négligée pour autant. Idéalement, si on ne veut pas égarer les développeurs, il faut que les nouvelles instructions qu'on ajoute au langage s'apparentent aux instructions du noyau du langage. Bien que le module NitSpace n'ait pas dénaturé le langage à l'aide de nouvelles instructions, il reste que la manipulation de valeurs via des tuples n'est pas particulièrement «orientée objet». Sachant que dans Nit tout est objet, une solution alternative devait être proposée afin de permettre d'échanger des objets plus complexes que les trois types de base. L'introduction des types complexes vient justement pallier ce problème. Puisque Nit ne possède aucun mécanisme de réflexion, il était impossible de faire comme avec Java et de déterminer durant l'exécution le type de l'objet ainsi que ses attributs. L'implémentation d'une interface qui exige la définition des méthodes de sérialisation et instanciation de l'objet via des tuples fonctionne, mais elle est un peu exigeante. Une amélioration possible serait d'intégrer, dans le compilateur Nit, la génération de ces méthodes afin que les développeurs ne soient pas obligés de les définir constamment.

Pour ce qui est des performances, quelques améliorations ont été apportées afin d'augmenter la vitesse d'exécution du processus qui gère le NitSpace. Les insertions en début de liste, l'ajout de valeurs de hachage pour les chaînes de caractères et la gestion des listes de tuples à l'aide d'une table de hachage sont des exemples de ce qui a été fait pour améliorer les performances d'exécution. Fait important à souligner, puisque la gestion du NitSpace est réalisée directement en C, cela permet d'optimiser le code sans se soucier du compilateur et des mécanismes qui fonctionnent durant l'exécution comme le *garbage collector*. En se libérant des contraintes imposées par Nit, on pourrait apporter d'autres types d'améliorations, par exemple, améliorer le NitSpace en permettant une exécution concurrente : actuellement, le NitSpace gère les demandes de façon séquentielle et une congestion pourrait ralentir le traitement en présence d'un nombre trop élevé de processus qui interagissent avec le NitSpace. Il serait également possible d'améliorer la recherche de tuples à l'aide de structures plus appropriées comme des arbres de

recherche.

Malgré toutes les optimisations qui pourraient être apportées, il reste que le type d'algorithme choisi influence directement le temps d'exécution. Avec une approche comme Linda, il est préférable de travailler, lorsque c'est possible, avec des variables locales. Le module NitSpace permet justement de partager des variables locales lors de la commande `fork`. Il est donc préférable d'initialiser l'ensemble des variables communes avant de générer les autres processus. Il ne reste plus qu'à voir comment exploiter ce type d'approches plus efficacement.

APPENDICE A

CALCUL DES NOMBRES PREMIERS AVEC NITSPACE

L'algorithme du crible d'Ératosthène permet de trouver tous les nombres premiers inférieurs à un entier donné. Dans cette approche, l'algorithme démarre avec un tableau contenant tous les entiers devant être vérifiés. Ensuite, on procède par élimination, en supprimant du tableau tous les multiples de chaque nombre, l'un après l'autre. L'algorithme peut se terminer lorsque le carré du prochain nombre dont on doit supprimer ses multiples est supérieur au plus grand nombre premier du tableau. À la fin, tous les nombres qui n'ont pas été supprimés du tableau sont premiers.

Dans notre exemple, le programme se sert d'un algorithme de style *sac de tâches*, tel que présenté à la section 1.4.2. Il utilise une variante de l'approche du crible d'Ératosthène pour trouver les 1000 premiers nombres premiers. Dans cet algorithme, au lieu de définir un tableau fixe et de supprimer les nombres qui ne sont pas premiers, on vérifie un à un chaque nombre afin de s'assurer qu'il n'est pas un multiple d'un des nombres premiers déjà trouvés. Chaque nombre à vérifier est ajouté dans le NitSpace en tant que tâche à traiter pour les processus travailleurs.

Dans le programme de notre exemple, quatre processus s'exécutent en parallèle pour vérifier les entiers. Une fois le nombre vérifié, le processus travailleur communique le résultat au processus coordonnateur en utilisant le NitSpace. C'est le coordonnateur qui transfère les nombres premiers trouvés aux autres processus et c'est également le coordonnateur qui ordonnera l'arrêt aux processus travailleurs lorsque le nombre requis de nombres premiers aura été trouvé.

```
import nitspace


# Classe qui définit le traitement d'un processus travailleur
class Worker
    # Référence sur le NitSpace.
    var nitSpace: NitSpace

    # Liste des nombres premiers trouvés
    var nombresPremiers: List[Int]

    # Constructeur d'un travailleur
    init (nitSpace: NitSpace)
    do
        self.nitSpace = nitSpace

        # Initialise la liste des nombres premiers avec les deux nombres premiers, 2 et 3
        nombresPremiers = new List[Int]
        nombresPremiers.push(2)
        nombresPremiers.push(3)
    end

    # Fonction qui vérifie si le coordonnateur a demandé l'arrêt
    fun termine: Bool
    do
        # Vérifie la commande d'arrêt dans le NitSpace
        var resultat = nitSpace.read_no_wait ( "Control", "STOP" )

        # Retourne faux tant que la commande de contrôle d'arrêt n'est pas rencontrée
        return resultat != null
    end
```

```
# Fonction qui exécute le traitement du travailleur
fun executer
do
    # Tant que l'arrêt n'est pas demandé, on vérifie le prochain nombre
    while not termine do
        # Récupère dans le NitSpace le prochain nombre à vérifier
        var resultat = nitSpace.take ( "Candidat", nsInt )
        var v: Int = resultat[1].as(Int)

        # Ajoute dans le NitSpace le prochain nombre à vérifier,
        # v+2 car on ne vérifie que les nombres impairs
        nitSpace.write( "Candidat", v+2 )

        var i: Int = 0
        var estPremier: Bool = true

        # On vérifie si le nombre peut être divisé par un des nombres premiers déjà trouvés
        while nombresPremiers[i] * nombresPremiers[i] <= v do
            # S'il est divisible, c'est que ce n'est pas un nombre premier
            if v % nombresPremiers[i] == 0
            then
                estPremier = false
                break
            end

            # S'il n'y a plus de nombres premiers dans la liste,
            # on met la liste à jour à partir du NitSpace
            i += 1
            if i >= nombresPremiers.length
            then
                resultat = nitSpace.read ( "Premier", nombresPremiers.length+1, nsInt )
                nombresPremiers.push ( resultat[2].as(Int) )
            end
        end
```

```
            # On communique le résultat au coordonnateur
            nitSpace.write ( "Resultat", v, estPremier.to_s )
        end

        # Avertit le coordonnateur qu'il a terminé
        nitSpace.write ( "Control", "TERMINE" )
    end
end


# Classe qui définit le traitement du processus coordonnateur
class Master
    # Référence sur le NitSpace
    var nitSpace: NitSpace

    # Liste des nombres premiers trouvés
    var nombresPremiers: List[Int]

    # Quantité de nombres premiers recherchés
    var nbRecherches: Int

    # Constructeur du coordonnateur
    init( nitSpace: NitSpace, nbRecherches: Int )
    do
        self.nitSpace = nitSpace
        self.nbRecherches = nbRecherches

        # Initialise la liste des nombres premiers avec les deux nombres premiers 2 et 3
        nombresPremiers = new List[Int]
        nombresPremiers.push(2)
        nombresPremiers.push(3)
    end
```

```
    # Fonction qui exécute le traitement du coordonnateur
    fun executer
    do
        # Ajoute dans le NitSpace le prochain nombre à vérifier
        var prochain: Int = 5
        nitSpace.write ( "Candidat", prochain )

        # Compile tous les résultats obtenus des processus travailleurs
        var resultat: Tuple
        while nombresPremiers.length < nbRecherches do
            # Lit dans le NitSpace le résultat de la vérification du nombre
            resultat = nitSpace.take ( "Resultat", prochain, nsString )

            # Si le résultat est vrai, c'est un nombre premier
            if resultat[2].to_s == "true"
            then
                # On l'ajoute dans la liste des nombres premiers
                nombresPremiers.push( prochain )

                # On diffuse le nombre premier dans le NitSpace
                # afin d'avertir les autres travailleurs
                nitSpace.write ( "Premier", nombresPremiers.length, prochain )
            end
            prochain += 2
        end

        # Le traitement est terminé, on communique le signal d'arrêt aux travailleurs
        nitSpace.write ( "Control", "STOP" )

        # On affiche la liste des nombres premiers, séparés par des virgules
        print nombresPremiers.join (", ")
    end
end
```

```
################################
# Bloc d'instructions de départ #
################################


# Nombre de processus travailleurs désiré
var nbWorkers: Int = 4

# Génère un nouveau NitSpace
var nitSpace = new NitSpace

# Duplique le processus en cours nbWorkers fois pour chaque nouveau processus,
# et on démarre le traitement de la classe travailleur
if fork(nbWorkers) != 0 then
    (new Worker(nitSpace)).executer

# Si fork == 0, on est toujours dans le processus original,
# donc on démarre le traitement de la classe coordonnateur
else
    (new Master(nitSpace, 1000)).executer

    # Barrière de synchronisation.
    # On s'assure que tous les processus travailleurs ont terminé
    for i in [1..nbWorkers] do
        nitSpace.take ( "Control", "TERMINE" )
    end

    # Termine le processus et libère le NitSpace
    nitSpace.stop
end
```

BIBLIOGRAPHIE

(1) N. Carriero et D. Gelernter : Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

(2) N. Carriero et D. Gelernter : *How To Write Parallel Programs: A First Course.* The MIT Press, Cambridge, Massachusetts, 1990.

(3) E. Freeman et S. Hupfer : Make room for JavaSpaces. `http://www.javaworld.com/jw-11-1999/jw-11-jiniology.html`, 1999.

(4) E. Freeman, S. Hupfer et K. Arnold : *JavaSpaces: Principles, Patterns, and Practice.* Addison-Wesley Professional, 1999.

(5) M. M. Glybovets, S. S. Gorohovskiy et M. S. Stukalo : Extension of Scala language by distributed and parallel computing tools with Linda coordination system. *Cybernetics and Systems Analysis*, 46(4):624–629, 2010.

(6) GRÉSIL : *Développez en Nit.* Université du Québec à Montréal, 2011.

(7) GRÉSIL : A concise reference of the Nit language. `http://nitlanguage.org/doc/`, 2012.

(8) S. Hupfer : The nuts and bolts of compiling and running JavaSpaces programs. `http://java.sun.com/developer/technicalArticles/jini/javaspaces/`, 2000.

(9) Cray Inc : *Chapel Language Specification 0.796.* Cray Inc., 2010.

(10) A. Laferrière : *L'interface native de Nit, un langage de programmation à objets.* Université du Québec à Montréal, Maîtrise en informatique, 2012.

(11) ruby doc.org : rinda: Ruby standard library documentation. `http://ruby-doc.org/stdlib-1.9.3/libdoc/rinda/rdoc/index.html`, 2012.

(12) V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu et D. Grove : *Report on the Programming Language X10, version 2.1*, 2010.

(13) G. C. Wells, A. G. Chalmers et P. G. Clayton : Linda implementations in Java for concurrent systems. *Concurrency — Practice and experience*, 00:1–7, 2003.

(14) A. Wilkinson : PyLinda v0.6. `http://www-users.cs.york.ac.uk/~aw/pylinda/`, 2012.