7.7 Linda

- Approche assez particulière qui combine des élément de la programmation par variables partagées et par échange de messages
- Pas un langage de programmation . . . mais plutôt un *langage de coordination* : composé uniquement de six (6) primitives utilisées pour accéder un *espace de tuples* (*tuple space*)
- N'importe quel langage séquentiel peut être augmenté des primitives Linda, par ex. : C, FORTRAN, Java (`JavaSpace`), Modula-3, . . . , Threaded-C

Slide 1

L'espace de tuples agit comme un canal global de communication, à la différence que les tuples ne sont pas ordonnés.

Opérations de base :

- `OUT` : Dépose un tuple dans l'espace de tuples ($\approx$ `send`)
- `IN` : Retire un tuple de l'espace de tuples ($\approx$ `receive`)
- `RD` : Lit un tuple de l'espace de tuples, mais sans le retirer

7.7.1 Espace de tuples et interaction de processus

L'espace de tuples est une collection de tuples de données passifs et de tuples actifs (processus). Lorsqu'un tuple actif termine son exécution, il se transforme en tuple de données passif.

Un tuple est un enregistrement *étiqueté* de la forme générale suivante :

$$(\texttt{"tag"}, valeur_1, \ldots, valeur_n)$$

Slide 2

Les deux (2) principales primitives sont :

- `OUT("tag", expr_1, ..., expr_n)` : évalue les expressions et dépose le tuple résultant dans l'espace de tuples.
- `IN("tag", champ_1, ..., champ_n)` : chaque champ est soit une expression, soit un paramètre formel de la forme "`?var`". Le processus exécutant `IN` bloque jusqu'à ce que l'espace de tuples contienne au moins un tuple qui correspond au *gabarit* spécifié. Le tuple correspondant est ensuite *retiré* de l'espace de tuples.

Slide 3

Autres opérations :

- `RD("tag", champ_1, ..., champ_n)` : comme `IN`, à la différence que le tuple obtenu *n'est pas retiré* de l'espace de tuples.
- `INP/RDP` : semblable à `IN/RD`, à la différence que ce sont des versions non-bloquantes, qui retournent immédiatement `false` si aucun tuple ne correspond au gabarit spécifié.
- `EVAL("tag", expr_1, ..., expr_n);` : crée un tuple actif (processus). Même forme que `OUT`, à la différence que l'une des expressions est généralement un appel de fonction ou procédure et que tous les champs du tuples sont évalués de façon concurrente. De plus, le processus qui exécute `EVAL` n'attend pas que l'évaluation soit terminée. C'est cette primitive qui permet d'introduire de la concurrence et du parallélisme.

Slide 4

Exemples :

Soit `sem` un nom de sémaphore. Alors, les opérations P et V peuvent être obtenues comme suit :

```
IN ( "sem" );
...
OUT( "sem" );
```

Un tableau de sémaphores pourrait être simulé comme suit :

```
IN ( "sems", i );      # P(sems[i]);
...
OUT( "sems", i );      # V(sems[i]);
```

Slide 5

Une barrière, qui doit être initialisé par `OUT("barrier", 0);` :

```
int nb;
IN ( "barrier", ?nb  );  # Signaler l'arrivee a la barriere.
OUT( "barrier", nb+1 );
RD ( "barrier", N    );  # Attendre que les N processus soient arrives.
```

Un canal `ch` :

```
OUT( "ch", e_1, ..., e_n );   # send ch(e_1, ..., e_n);

...

IN ( "ch", ?v_1, ..., ?v_n ); # receive ch(v_1, ..., v_n);
```

Slide 6

7.7.2 Exemple : calcul des nombres premiers avec un sac de tâches

Voir Fig. 7.16 (pp. 338–339) :

- Les processus `worker` partagent un sac de tâches (tuples avec l'étiquette `"candidate"`).
- Les nombres ayant été identifiés comme premiers sont conservés sous formes de tuples étiquetés par `"prime"`.
- Les tuples `"result"` servent à communiquer les résultats au processus maître.
- Le tuple `"stop"` permet au processus maître de signaler la fin des traitements (limite maximum atteinte).