République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mentouri Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

THESE

Présentée par

Elkamel MERAH

Pour obtenir le titre de

Docteur en Sciences

Spécialité **Informatique**

Thème

Vers un Nouveau Langage Mobile

Devant le Jury composé de :

Pr Mohamed BENMOHAMMED	Président	Université de Constantine
Dr Allaoua CHAOUI	Rapporteur	Université de Constantine
Dr Brahim BELATTAR	Examinateur	Université de Batna
Dr Okba KAZAR	Examinateur	Université de Biskra
Dr Djameleddine SAIDOUNI	Examinateur	Université de Constantine

Soutenue le 06 Juillet 2009

نحـــو لغـــة برمجـــة حركيـــة جديـــدة

الكـامـل مراح

مختصـــــــر

في هذه الرسالة، نقترح امتدادا للغة Caml Light هدفه جمع نماذج البرمجة الدالية، الآمرة، التنافسية و الموزعة في لغة برمجة واحدة. هذا الامتداد مكيف مع الأنظمة ذات الذاكرات الموزعة. هذا الامتداد موجه أيضا لدعم الحراك. على غرار الكثير من حلول أخرى، ذهبنا لنظام غير زمني كما هو الحال في لغة ERLANG، حيث إرسال و استقبال الرسائل لا يكون متزامنا بالضرورة. بالنسبة للامتداد ألتزامني ل Caml Light، نقترح مجموعة من الدوال ذات معنى بسيط. إن دعامة الإنشاء ألحراكي للبرامج و الاتصالات الغير الزمنية ما بين البرامج يعتبران من أهم الامتدادات ل لغة Caml Light. إن إعداد برامج موزعة في هذه اللغة الممتدة ينجز عن طريق تكييف الدوال المستعملة في التنافس. استعمال هذه الدوال مدعم ببعض الأمثلة.

Towards A New Mobile Language

Elkamel MERAH

Abstract

In this thesis, we propose an extension of Caml Light language which aims to encompass functional, imperative, concurrent and distributed programming paradigms in a single programming language. This extension is tailored for distributed memory systems. It is also aimed towards mobility. In contrast to other solutions, we chose to have an asynchronous system like in ERLANG, where sending and receiving do not to have to occur simultaneously. For the concurrent extension of Caml Light, we propose a set of primitives with very simple semantics. Supports for dynamic process creation and inter-process asynchronous communication are the two of the essential extensions to Caml Light language. Writing distributed programs in Distributed Caml Light is made by the adaptation of primitives used for concurrency. The use of these constructions added to Caml Light is illustrated through some examples.

Vers Un Nouveau Langage Mobile

Elkamel MERAH

Résumé

Dans cette thèse, nous proposons une extension du langage Caml Light qui vise à rassembler les paradigmes de programmation fonctionnelle, impérative, concurrente et distribuée dans un seul langage de programmation. Cette extension est adaptée aux systèmes à mémoires distribuées. Elle est aussi destinée au support de la mobilité. A l'opposé de beaucoup d'autres solutions, nous avons opté pour un système asynchrone comme dans le langage ERLANG, où l'envoi et la réception de messages n'ont pas à se produire simultanément. Pour l'extension concurrente de Caml Light, nous proposons un ensemble de primitives avec une sémantique très simple. Le support pour la création dynamique et la communication asynchrone entre processus sont les deux plus importantes extensions du langage Caml Light. Ecrire des programmes distribués dans ce langage étendu est faite par l'adaptation des fonctions utilisées pour la concurrence. L'utilisation des constructions ajoutées à Caml Light est illustrée par quelques exemples.

Remerciements

Je suis très reconnaissant envers Monsieur Allaoua Chaoui, maître de conférences à l'université Mentouri de Constantine, pour avoir proposé ce thème et avoir accepté de diriger cette recherche, et aussi pour sa disponibilité et ses nombreux conseils. Je suis très profondément touché par la confiance qu'il m'a accordée tout au long de ce travail.

J'exprime ma profonde gratitude à M. M. Benmohammed, Professeur à l'université Mentouri de Constantine, pour l'honneur qu'il me fait de présider mon jury de thèse.

Je suis particulièrement reconnaissant envers M. B. Belattar, maître de conférences à l'université de Batna, et un de mes anciens professeurs à l'université de Constantine, pour avoir accepté d'être parmi les jurés de cette thèse.

Je tiens à remercier tout particulièrement M. O. Kazar, maître de conférences à l'université de Biskra, d'être parmi les jurés de cette thèse.

Je remercie M. D. Saidouni, maître de conférences à l'université Mentouri de Constantine, pour avoir accepté la lourde charge d'être aussi examinateur.

Mes remerciements vont naturellement à l'ensemble des gens du centre universitaire de Khenchela que j'ai côtoyé et qui m'ont encouragé. En particulier je voudrais remercier M. B. Houha et M. N. Benalicherif qui n'ont cessé de m'encourager.

Je remercie finalement, mes amis et tous les collègues du département d'informatique et de l'institut SET, pour leur soutien et leurs encouragements, en particulier Hakim Sahour.

Je ne saurais oublier, l'aide de mon fils Djaber, de 14 ans, pour sa maintenance impeccable de mon PC, aucun virus n'a eu la chance de se trouver quelque part dans mon disque ! Merci Djaber.

Je remercie aussi mes anciens amis et mes anciens collègues qui se regroupent souvent dans la boutique de Saihi.

Et puis je remercie tous ceux que je n'aurais bien sûr pas dû oublier …

Je dédie cette Thèse

- *A la mémoire de ma mère, qui m'a quitté durant la préparation de cette thèse et qui me manque énormément,*
- *A mon père qui m'a toujours soutenu,*
- *A Hayet, mon épouse, pour m'avoir supporté,*
- *A mes enfants Raouf, Faiza, Rayane et Djaber.*

Table des Matières

Introduction générale

Le paradigme de la mobilité

Les applications distribuées Internet se démocratisent, et la plupart des applications sont aujourd'hui distribuées, même si la distribution se réduit des fois aux simples échanges de fichiers. Les équipements utilisés par ces applications se sont diversifiés que ce soit au niveau matériel (stations de travail, ordinateurs portables …) ou au niveau systèmes.

L'hétérogénéité des architectures d'ordinateurs et des systèmes informatiques, ont défié depuis les années 50 la communauté des chercheurs. Plusieurs tentatives ont été faites afin de fournir un modèle de calcul homogène, parmi ces tentatives le langage universel UNCOL (*UNiversal Computer Language*) **[Conw58]**. Malheureusement, l'idée d'un langage universel unifiant tous les langages et supportant toutes les architectures s'est révélée très peu pratique malgré les développements ultérieurs dans ce domaine.

Mais, il est utile de rappeler quand même que bien avant l'apparition du langage Java et son bytecode, le P-code de Pascal **[Nori81]** (code intermédiaire destinée à une machine virtuelle) fût introduit pour diffuser, sans Internet qui n'existait pas encore bien sûr, avec un grand succès le premier compilateur Pascal. Grâce à ce P-code (repris par la suite par beaucoup, les développeurs de Java aussi !) produit par le compilateur Pascal et la technique du *bootstrapping* **[Nori75]**, il était facile et pratique d'utiliser le langage Pascal sur n'importe quelle machine. Nous avons eu la chance un jour d'utiliser ce compilateur et l'interpréteur de cette machine virtuelle (la machine *P*).

Ce sujet hiberna beaucoup jusqu'à l'avènement de la programmation Internet (obligée !). Le développement d'Internet et du Web a changé ou plutôt a imposé ses conditions, en proposant un environnement de connexion et un environnement dynamique où la mobilité du code peut convenir à la conception d'applications distribuées. Ce développement a donc donné un nouvel essor aux travaux sur la mobilité. D'autre part, le langage Java a fourni un support et des outils pour écrire des

programmes qui peuvent se déplacer à travers des réseaux, comme c'est le cas très répandu avec les applets.

La programmation Internet est basée sur un calcul qui doit être mobile et donc doit traiter l'hétérogénéité et tous les problèmes de sécurité. Un calcul mobile est une unité d'exécution qui démarre son exécution sur un site et peut le cas échéant se déplacer vers un autre site pour continuer son exécution. Un tel transfert peut être initié à tout moment de l'exécution. L'état courant de l'exécution est transféré avec le code de manière que l'exécution puisse reprendre au point où elle a été arrêtée.

Les codes mobiles ou agents mobiles comme on les appelle des fois, sont essentiellement utilisés dans les applications distribuées dans des réseaux, car ils permettent de minimiser les coûts de communication en se déplaçant vers les sites disposant des ressources nécessaires. La mobilité est aussi exploitée dans un contexte de calcul important à des fins d'équilibrage de charge ou de tolérance aux pannes

Actuellement, le paradigme le plus populaire pour concevoir des applications distribuées est le modèle client/serveur. Il est notamment utilisé comme base dans les environnements CORBA **[OMG95]** et Java **[Sun95]** à travers un mécanisme d'invocation d'objet à distance. Mais il ne suffit pas simplement de déplacer des objets comme dans RMI ou CORBA, ou de déplacer uniquement du code comme avec les applets Java. La mobilité, qui consiste à déplacer le code d'une application et son contexte d'exécution d'une machine à une autre afin d'effectuer ses tâches, est difficile à implanter. Elle pose des problèmes délicats d'hétérogénéité de plateformes, de captures et de restaurations de l'état d'exécution, et des problèmes de sécurité. Par exemple, il est très difficile de contrôler le comportement des codes mobiles à cause de leur migration une fois sur les sites de destination.

Les langages du code mobile utilisent la programmation distante et chaque langage doit fournir un ensemble de primitives afin de supporter la mobilité.

Plusieurs systèmes ou langages mobiles ont été développés durant ces dernières années comme : Telescript, Aglets, AgentTcl … La plupart d'entre eux ont été

implémentés au dessus de Java, à cause de sa grande diffusion et de quelques avantages techniques : indépendance de la machine, et le typage fort pour des raisons de sécurité.

Cependant même Java n'apporte pas de réponse complète aux besoins de mobilité des applications : une application Java – avec son code, ses données et l'état de son exécution – ne peut pas être rendue mobile. La prise en compte du contexte d'exécution des applications Java pour des besoins de mobilité a fait l'objet de travaux de recherches **[Bouc01]**. Mais ses langages ou d'autres, souffrent de beaucoup d'insuffisances comme :

- Langage très privé, donc peu connu, alors très peu d'utilisation
- Langage étendu pour la mobilité mais dès le départ non prédisposé
- Modifications majeurs dans le langage, ce qui nie aux objectifs initiaux …

C'est donc dans ce contexte que s'inscrit notre travail, afin de proposer la conception d'un nouveau langage – en réalité un langage au dessus d'un langage existant – visant le support de la mobilité.

Objectifs de la thèse

Nos objectifs sont donc de démarrer d'un langage « prédisposé » pour la mobilité, c'est-à-dire qui possède des propriétés que nous jugeons *favorables* pour la mobilité. L'idéal serait de recommencer à zéro, c'est-à-dire établir un cahier de charges de mobilité très complet et définir un nouveau langage de programmation qui supporte entre autres la mobilité. Mais les inconvénients de cette démarche ne manquent pas :

- Le temps nécessaire à la conception d'un nouveau langage et comment assurer son utilisation par la communauté des programmeurs.
- Il n'existe pas de définition précise ou du moins une définition de la mobilité acceptée partout.
- Est-ce que la mobilité est une nécessité absolue pour lui consacrer tout cet effort et tout ce temps, surtout si l'on consulte le fameux article de Giovanni Vigna **[Vign04]** un des fondateurs de la mobilité où il avance dix raisons pour l'échec des agents mobiles !
- …

Nous avons pensé, après un bout de temps assez important, qu'il serait judicieux de supporter la mobilité au dessus d'un langage existant et qui va faciliter le support de la mobilité, bien sûr, ce langage doit bénéficier d'une large utilisation. Nous avons choisi le langage Caml Light, une version légère et portable du langage Caml, pour les raisons qui seront justifiées tout au long de cette thèse. Caml Light est donc notre candidat pour une extension vers la mobilité. Alors notre extension devra d'abord supporter ces deux paradigmes ou modèles nécessaires à la mobilité et qui sont la *concurrence* et la *distribution*. Le langage Caml light étant typiquement séquentiel.

Tous nos efforts seront donc consacrés à 3 points : le choix du bon langage, le choix du bon modèle de concurrence et le choix du bon modèle de distribution tout en ayant à l'esprit la mobilité comme objectif ultime. La démarche que nous avons adoptée pour aborder le problème de la mobilité du code est composée de cinq parties :

1. L'étude des travaux antérieurs concernant la mobilité du code.
2. Choix du langage (et donc aussi d'un paradigme) adapté et à étendre vers une mobilité
3. Proposition d'un modèle de concurrence et donc d'un Caml Light concurrent.
4. Proposition d'un modèle de distribution proche de celui de la concurrence pour garder une certaine homogénéité dans le langage, et donc d'un Caml Light distribué.
5. Spécification de mécanismes pour la migration de processus pour aller vers une mobilité dans Caml light.

C'est à dire nous allons opérer par phases successives. Il faut quand même dire que ceci n'est qu'une étape préliminaire, car l'implémentation réelle de tous ces mécanismes justifiera les décisions prises.

Le résultat de notre contribution se présente donc sous la forme d'extensions du langage Caml light, à travers de nouvelles primitives – primitives de haut niveau, c'est-à-dire un support direct au niveau langage – d'abord pour la concurrence et la distribution. C'est aussi une contribution au développement, hors INRIA **[INRIA]**, du langage Caml choisi pour l'extension citée, et qui devient de plus en plus populaire dans

les milieux universitaires, et commence à attirer de manière sérieuse la communauté industrielle, en Europe et aussi aux Etats-Unis.

Structure de la thèse

Le reste de la thèse est organisé comme suit. Le chapitre 1 est consacré à la description du langage Caml Light. Il ne s'agit pas d'un manuel de langage bien sûr, mais il s'agit juste de présenter les caractéristiques essentielles et importantes et qui serviront à la justification d'une extension de ce langage.

Le chapitre 2 décrit le modèle choisi pour la concurrence et montre comment les opérations asynchrones peuvent être utilisées pour implémenter les mécanismes de communication entre processus. Un Caml Light concurrent sera alors proposé **[Mera09c]**. Dans ce chapitre aussi, nous présentons de manière brève les caractéristiques essentielles du langage ERLANG, duquel nous avons emprunté quelques concepts. Nous avons vu nécessaire de lui donner de l'importance par cette présentation.

Le chapitre 3 est dédié à la distribution où nous allons montrer que le choix du modèle de concurrence peut être étendu et élargi pour avoir un langage Caml Light distribué **[Mera09a]**.

Dans le chapitre 4, et le dernier, nous discutons la mobilité, et le support qui manque à Caml Light pour se rapprocher de la mobilité **[Mera09b]**.

Nous terminons bien sûr par une conclusion qui résume l'apport essentiel de nos travaux. Cette conclusion ouvre également de nouveaux éléments de réflexion et quelques perspectives de recherche.

Chapitre 1

Le Langage Caml Light

Ce chapitre est consacré au langage Caml, plus exactement au langage Caml Light, langage choisi pour une extension vers la mobilité. Bien sûr, il ne s'agit pas d'un manuel d'utilisation du langage, mais plutôt d'une petite introduction aux concepts fondamentaux du langage Caml Light, qui quand même permet de se faire une idée des traits saillants de ce langage. A la fin de ce chapitre, on essayera de donner une justification du choix de Caml Light comme candidat à notre extension.

1.1 Présentation compacte

Le langage Caml Light a hérité d'une dizaine d'années de la recherche dans la théorie des types, la conception de langages et l'implémentation de langages fonctionnels.

Il est utilisé dans l'éducation, dans l'apprentissage de la programmation, dans l'industrie et dans le monde de la recherche pour le développement de projets académiques. Notre projet se situe aussi dans ce contexte.

Caml Light est un langage de programmation de haut niveau essentiellement fonctionnel et à usage général. Il possède aussi quelques traits impératifs.

Caml Light adopte le typage statique qui assure l'écriture d'applications sûres et possède un système d'inférence de types. Sa stratégie d'évaluation est stricte et il est polymorphe. Il offre une gestion automatique de la mémoire.

Caml Light dispose d'un système de gestion des exceptions pour la signalisation et la manipulation des conditions exceptionnelles. Il permet le filtrage.

Son système de modules puissant et très expressif permet l'écriture d'applications importantes.

Caml Light est léger et portable. Il fonctionne en deux modes : interprété et compilé. Le langage est mature et son compilateur produit du code efficace pour différentes architectures.

Caml Light est actuellement en open source et est disponible gratuitement **[INRIA]**.

1.2 Une petite histoire de Caml

Caml **[Chât03]** est à l'origine un acronyme pour « ***C****ategorical* ***A****bstract* ***M****achine* ***L****anguage* » qui est un jeu de mots sur :

- CAM : **M**achine **A**bstraite **C**atégorique : une machine virtuelle basée sur la théorie des catégories et sur le lambda-calcul.
- ML : *Meta language* : la famille de langages de programmation à laquelle Caml appartient. A l'origine de cette famille est Robin Milner de l'Université d'Edinburgh avec sa proposition du langage ML dans les années 70.

Caml est donc né dans les années 80 dans les laboratoires de l'INRIA (Institut National de Recherche en Informatique et en Automatique) en France. Il fût d'abord conçu par l'équipe *Formel* **[Form]** et poursuit actuellement son développement par l'équipe *Cristal* **[Cris]**.

La première véritable implémentation de Caml arriva en 1987. En 1991, Xavier Leroy, actuellement Directeur de recherche à l'INRIA et que j'ai contacté pour discuter du projet de cette thèse, créa une nouvelle implémentation de Caml basé sur un interpréteur de bytecode écrit en C et connue maintenant sous le nom de Caml Light **[Lero97]**. Elle fût aussi dotée d'un bon système d'allocation mémoire.

Le terme Light (léger) est pour version hautement portable avec un interpréteur léger et efficace, ce qui a permis une large diffusion de cette version dans les domaines de l'enseignement et de la recherche. C'est l'une des raisons principales, justifiée par la suite, de notre choix pour cette version *légère* de Caml, c'est-à-dire Caml Light.

Xavier Leroy, une grande figure à l'INRIA, ne s'arrêta pas là, et ajouta à Caml Light un système de modules puissant et à partir de 1996, tous ses travaux, avec ses collaborateurs, ont abouti au langage Ocaml (*Objective* Caml) **[Lero07]**. Le O pour *objective* de programmation orienté objet, par l'adjonction d'une couche objet au langage Caml. Ce langage gagne régulièrement en popularité et attire une communauté significative d'utilisateurs. Le langage Caml continu toujours a évolué, essentiellement

dans les laboratoires de l'INRIA, par de nouveaux concepts et de nouvelles implémentations encore plus efficaces.

Enfin et pour terminer, en 1996 Guy Cousineau, aussi l'un des fondateurs de Caml, écrivait « L'histoire de Caml aurait certainement pu être plus linéaire. Néanmoins, à travers une série de *tâtonnements*, une capacité à produire des implantations de haute performance, portables et flexibles, pour les langages de programmation fonctionnels a émergé en France ».

1.3 Types de données

Avant de parler des différents types dans Caml Light, il est important d'indiquer que ce langage est statiquement typé, ce qui veut dire que le contrôle de type se fait à la compilation, ce qui conduit très tôt à la détection d'erreurs, ajoute de la sécurité et améliore les performances des programmes. Encore nul besoin d'indiquer les types dans le code, un moteur d'inférence de types se charge de la détection des types.

Il existe en Caml Light plusieurs types de données prédéfinis :

- Types de base :
  - **`int`** pour les entiers
  - **`float`** pour les réels
  - **`bool`** pour les booléens
  - **`char`** pour les caractères
  - **`string`** pour les chaînes de caractères
  - **`unit`** pour le type vide avec **()** comme unique valeur, l'équivalent de **void** en C.
- Types composés :
  - tuples : `(1, true, 'ab')` est un tuple de type `int*bool*string`. Les éléments sont de types différents mais leur arité est fixe.
  - listes : `[], [1 ; 2 ; 3]` les éléments doivent être de même type.
  - …

En plus de ces types prédéfinis, le langage propose de puissants moyens pour construire de nouveaux types ou types utilisateurs :

- Types produit : produit mathématique de plusieurs types (enregistrements).
- Types somme : union disjointe de plusieurs types.

1.4 Valeurs et déclarations

Il y a deux types de déclarations :

- Les déclarations globales : connues par toutes les expressions qui suivent la déclaration, en utilisant la construction `let name = expr ;;`
- Les déclarations locales : connues par une seule expression avec la construction `let name = expr1 in expr2 ;;`

Le mot clé **`let`** ne fait pas une affectation, il permet d'introduire un identificateur dans un environnement.

1.5 Fonctions

Caml Light est un langage fonctionnel : les fonctions sont des valeurs de première classe, au même titre que les entiers, les chaînes, etc., elles peuvent être retournées en résultats et passées en arguments à d'autres fonctions. Elles se manipulent comme en mathématiques.

Une valeur de type *fonction*, encore appelée *valeur fonctionnelle*, est une règle de correspondance qui associe à chaque valeur d'un type *t*1 une valeur d'un type *t*2. Une telle valeur exprime donc la notion *mathématique* de fonction. En mathématique on écrira, par exemple :

$$\begin{aligned} successeur : \quad & N \rightarrow N \\ & n \rightarrow n + 1 \end{aligned}$$

et on dira : "successeur est la fonction qui, à tout nombre entier désigné par n, fait correspondre le nombre entier calculé par l'expression `n+1`". "La valeur" de cette

fonction est donc la règle de correspondance exprimée par l'écriture `n → n + 1`. L'identificateur `successeur` constitue le nom de cette fonction.

En CAML la définition d'une fonction est composée d'un nom (identificateur) suivi d'un ou de plusieurs arguments (paramètres formels), et d'une expression de définition. On utilise pour cela la construction **`let`** comme pour la définition des autres valeurs.

```
# let successeur n = n+1 ;;  /* notation la plus simplifiée
```

`successeur` est donc le nom d'une fonction avec l'argument `n` (entier), et qui retourne `n + 1` comme valeur. Pour appliquer cette fonction il suffit de faire :

`successeur 2009` qui produira `2010.`

La fonction suivante est une fonction récursive :

```
let rec fact n =
    if n = 0 then 1
    else n * fact (n-1)
```

Comme les fonctions sont des valeurs d'ordre supérieur, alors :

- Une fonction peut avoir un argument de type fonctionnel :

  ```
  let accroissement_un f = (f 1) - (f 0) ;;
  ```

  La fonction `accroissement_un` fait correspondre, à une fonction d'entiers vers entiers, la différence entre les deux valeurs f(1) et f(0). On peut l'appliquer avec la fonction `successeur` comme suit :

  `accroissement_un successeur` qui produira la valeur 1.

- De même une fonction peut retourner un résultat de type fonctionnel.
- Une fonction peut aussi avoir un argument de type fonctionnel et retourner un résultat de type fonctionnel. La fonction `bigger` qui prend comme argument une fonction (`f`) et retourne une autre fonction comme valeur:

  ```
  let bigger f x y = f x y;
  ```

L'*application* d'une fonction s'écrit tout simplement `f x`
L'*application partielle* d'une fonction est l'application d'une fonction à quelques uns et non à tous ses arguments.

1.6 Filtrage de motifs

Le filtrage (*pattern matching*) de Caml Light est un moyen de tester les cas de la structure d'un objet et de choisir des actions à effectuer selon chaque cas.

Le filtrage est appliqué aux valeurs. Il est utilisé pour reconnaître la forme d'une valeur et on utilise souvent la construction **`match`** suivante :

```
match  expr  with
    motif1 -> expr1
    motif2 -> expr2
    ...
    motifn -> exprn
```

L'expression `expr` est confrontée (*matching*) séquentiellement aux motifs. Si `expr` satisfait le motif, alors l'expression correspondante est évaluée. Toute expression satisfait le motif _ qui est le motif universel (tous les autres cas).

Le filtrage est aussi souvent utilisé pour définir une fonction par cas, comme dans :

```
let rec fib = function
    0   -> 1
    | 1 -> 1
    | n -> fib(n -1)+ fib(n -2);;
```

Pour la suite de Fibonacci.

Ou bien dans:

```
let implique v = match v with
   (true,true)     ! true
|  (true,false)    ! false
|  (false,true)    ! true
|  (false,false)   ! true ;;
```

Pour définir la fonction d'implication logique *implique* avec v comme argument (deux booléens) et retournant un booléen.

1.7 Les programmes

Un programme est une expression. Son exécution est l'évaluation de cette expression. En Caml Light, on n'écrit pas une suite d'instructions, mais on nomme des expressions avec **`let`**, comme avec :

```
let rec fact  n =
if n = 0 then 1
else n * fact (n–1)
```

Un programme est donc une suite de phrases :

- définition de valeur : **`let`** `x = e`
- définition de fonction : **`let`** `f x1 ... xn = e`
- définition de fonctions : **`let`** `[` **`rec`** `] f1 x1 ... = e1` ... mutuellement récursives `[` **`and`** `fn xn ... = en ]`
- définition de type(s) **`type`** `q1 = t1... [` **`and`** `qn = tn ]`
- expression e

Les phrases se terminent par `;;` qui sont optionnels entre des déclarations, mais obligatoires sinon.

1.8 Les modules

Les modules permettent :

- de découper un programme en parties compilables séparément
- de structurer un programme pour le rendre compréhensible
- de spécifier des liens (interfaces) entre les modules pour la maintenance

Dans ce contexte, Caml Light offre un système de modules très riche qui permet de contrôler la visibilité des champs en cachant certaines définitions de valeurs ou de types.

Un module **M** se compose de deux fichiers :

- un fichier d'implémentation (code) **m.ml** : une suite de phrases
- un fichier d'interface (optionnel) **m.mli** : une suite de spécifications (valeurs, types …)

Un autre module **`N`** peut faire référence à **`M`** en utilisant la notation `M.x` (notation pointée : `Module.Composante`), ou bien en faisant `open` **`M`**. Les modules peuvent aussi être paramétrés (généricité).

1.9 Aspects impératifs dans Caml Light

1.9.1 Références

Avec Caml light, une variable liée à une valeur est condamnée à garder cette valeur le long de sa vie. C'est le cas bien sûr des variables mathématiques. Une possibilité est quand même offerte par ce langage pour modifier les valeurs, c'est le cas des références. Une référence est vue comme un pointeur vers une valeur quelconque. On construit une référence à une valeur en utilisant la fonction ***ref***. L'opérateur ***!*** (déréférencement) permet d'atteindre la valeur référencée et **:=** (affectation) permet la modification de la valeur de la référence, comme dans l'exemple suivant :

```
let x = ref  0 ;;
x := !x + 1 ;;
```

1.9.2 Les structures de contrôle

```
for nom = expr to expr do expr done
for nom = expr downto expr do expr done
while expr do expr done
```

1.9.3 Les entrées sorties

Caml Light dispose d'un modèle impératif d'entrées sorties. Une opération d'entrée sortie provoque donc un effet de bord. Les entrées sorties sont effectuées sur des canaux avec comme type *in_channel* et *out_channel*. *std_in*, *std_out* et *std_err* sont les canaux standards pour les entrées, les sorties et les sorties erreurs connectés aux clavier et écran.

1.10 Autres traits de Caml Light

1.10.1 Exceptions

Caml Light possède un mécanisme de traitement des exceptions pour traiter les situations exceptionnelles ou des situations qui normalement ne devraient pas arriver. Quelques exceptions sont prédéfinies comme **`Failure`**, d'autres peuvent être définies par la construction **`exception`**.

Elles sont soulevées par la fonction **`raise`** et on les traite avec la construction syntaxique **`try...with`**. Dans le bloc **`with`** on peut filtrer les différentes exceptions qui ont pu être soulevées.

1.10.2 Stratégie d'évaluation

Caml Light est un langage "strict", par opposition aux langages paresseux. Cependant la pleine fonctionnalité permet de créer des suspensions et donc de coder l'évaluation paresseuse de données potentiellement infinies.

1.10.3 Gestion mémoire

Pour des raisons de sûreté, Caml light offre une gestion automatique de la mémoire: l'allocation et la libération des structures de données est implicite. Il n'y a pas de primitives de manipulation explicite de la mémoire comme "new" ou "free" ou "dispose" qu'on retrouve dans d'autres langages comme Pascal.

1.10.4 Polymorphisme

Caml Light est doté d'un typage puissant "polymorphe": certains types peuvent rester indéterminés, représentant alors "n'importe quel type" (**`type 'a list`**). Ainsi, les fonctions qui sont d'usage général s'appliquent à n'importe quel type de données, sans exception (par exemple les routines de tri s'appliquent à tout type de tableaux). Ce qui aide à la réutilisabilité du code.

1.11 Un petit exemple en Caml Light

```
(* racines2nd.ml *)

(* resolution d'une equation du second degre *)

let liste_des_racines (a,b,c) =
    if  a = 0.  then  if  b = 0. then [] else  [-. c /. b]
    else
        let delta = b ** 2. -. 4. *. a *. c in
        if delta < 0. then []
    else if delta = 0. then [-. b /. 2. *. a]
    else [(-. b +. sqrt(delta)) /. 2. *. a ;
          (-. b +. sqrt(delta)) /. 2. *. a]
```

```
(* [ liste_des_racines (a,b,c) ] renvoie la liste des
racines du polynome ax² + bx +c *)
```

Le résultat de la fonction `liste_des_racines` est une liste.

1.12 Conclusion

Voici les caractéristiques essentielles du langage Caml light et qui nous ont guidé à le choisir comme candidat à notre extension :

- langage assez répandu et devient de plus en plus populaire que ce soit dans l'enseignement ou dans les entreprises industrielles.

- Langage fonctionnel : en plus de la compacité du code des programmes écrits dans les langages fonctionnels, Les avantages de la programmation fonctionnelle sont de plus en plus évidents, surtout dans le domaine de la validation des programmes **[Gilm97]**, encore lorsque les programmes sont exécutés dans d'autres environnements comme c'est le cas des sites de destination du code mobile.

- Caml Light est statiquement typé donc très sûr : de nombreuses erreurs de programmation deviennent ainsi impossibles.

- Il offre une gestion automatique de la mémoire : on obtient ainsi une programmation bien plus sûre, puisqu'il n'y a jamais de corruption inattendue des structures de données manipulées.

- Caml Light est portable : son compilateur produit du bytecode indépendant de toute plateforme, une propriété essentielle et nécessaire à la distribution et à la mobilité à cause de l'hétérogénéité des plateformes. Dans ce sens Caml Light est doté d'une machine virtuelle qui exécute les programmes générés par le compilateur. L'interpréteur de la machine virtuelle est écrit en langage C.

- Caml Light est léger : le code (bytecode) généré par le compilateur est très compact, ce qui est très important pour la migration du code et qui permet de réduire le temps de chargement. Aussi sa machine virtuelle occupe très peu d'espace, 200 Ko pour un interpréteur minimal, ce qui permet de lancer plusieurs machines virtuelles (nœuds) dans le cas d'applications réparties ou distribuées.

- Il est gratuit ! et distribué par l'INRIA **[INRIA]**.

Chapitre 2

Caml Light concurrent

Nous allons décrire, dans ce chapitre, une extension du langage Caml Light par des primitives de création de processus et des primitives de communication inter processus.

Cette extension se basera sur un modèle de concurrence à échange asynchrone de messages inspiré du langage ERLANG que nous présenterons de manière très brève.

Pour illustrer la capacité à notre extension de Caml light d'exprimer des programmes concurrents, quelques solutions à des problèmes concurrents simples et utilisant les primitives ajoutées, seront exposées.

2.1 Introduction

L'informatique a débuté avec une programmation complètement séquentielle qui repose sur l'indépendance des programmes. Des domaines sont apparus par la suite, où les programmes deviennent dépendants et désirent coopérer. D'autre part, la technologie actuelle des semi-conducteurs approche ses limites physiques par rapport à la vitesse de transfert du signal électrique, et des solutions sont recherchées pour atteindre un maximum de performance. Parmi ces solutions, l'exploitation de la « concurrence » pour prendre avantage d'un parallélisme potentiel dans les programmes, même dans un contexte monoprocesseur.

La concurrence est au moins souhaitable, pour ne pas dire nécessaire, dans au moins deux cas, d'une part certaines applications sont implicitement concurrentes, et d'autre part, l'utilisation croissante de machines avec plusieurs processeurs pour augmenter les performances des programmes, ce qui exige le découpage d'un programme en plusieurs parties qui peuvent s'exécuter de manière concurrente.

Un programme, dans un langage de programmation, est qualifié de « concurrent » lorsqu'il contient des séquences d'instructions qui peuvent être exécutées de manière parallèle, comme les deux simples séquences suivantes :

```
X := A*B + C    et   Y := D + E ;
```

La programmation concurrente engage donc, d'une part des notations pour exprimer un parallélisme potentiel de sorte que des opérations peuvent être exécutées en parallèle, et d'autre part des techniques pour résoudre les problèmes de *synchronisation* et de *communication*. Il est à noter qu'on s'intéresse ici à un parallélisme potentiel dans un programme qui lui peut tout à fait s'exécuter de manière séquentielle sur une machine monoprocesseur, à l'opposé du vrai *parallélisme* physique, qui ne fait pas partie du cadre de notre travail. Il s'agit d'apporter un support langage à la programmation concurrente.

Le langage Caml light inclue toutes les caractéristiques requises pour la programmation séquentielle, mais un domaine non traité dans la définition du langage c'est qu'aucun mécanisme n'est prévu pour la concurrence. C'est dans ce contexte que nous allons agir.

Un concept fondamental de la programmation concurrente est la notion de *thread*.

2.2 Processus et Thread

Pour créer un nouveau processus, un système d'exploitation moderne doit pouvoir lancer l'exécution d'un programme. Un processus est donc un programme en cours d'exécution. Chaque processus possède au moins un thread qui s'exécute en lui.

Les systèmes d'exploitation modernes permettent à plusieurs threads de s'exécuter de façon concurrente dans un seul processus. C'est le cas par exemple du langage Java, lorsque la machine virtuelle est lancée par le système d'exploitation, un nouveau processus est crée, et dans ce processus plusieurs threads peuvent aussi être créés. On parle donc de programmation concurrente ou multithreading.

Un des grands atouts du langage Java est qu'il est doté d'un support intégré pour l'écriture de programmes concurrents. Ses concepteurs d'origine ont bien estimé l'importance et la valeur du multithreading en décidant d'inclure directement le support des threads dans le langage même.

Même si un système d'exploitation ne supporte pas le concept des threads, la machine virtuelle du langage Java saura simuler un environnement multithread. C'est aussi dans cet axe que nous allons étendre le langage fonctionnel Caml Light, mais avec un autre modèle de concurrence autre que celui de Java, que nous pensons est relativement de bas niveau.

Les threads sont souvent appelés processus légers. Un thread est un chemin d'exécution dans le code d'un programme, et chaque thread possède ses propres variables locales, son compteur de programme (PC) et sa durée de vie.

Les mêmes concepts et les mêmes mécanismes ont été repris pour permettre à un processus de comporter plus d'un fil (thread) d'exécution, en quelque sorte avoir plusieurs processus « légers » au sein d'un processus hôte, « lourd ». Comme exemple concret, l'*explorateur* sous Windows utilise un thread pour chaque fenêtre.

Le terme « léger » provient du fait que dans un processus, les threads se partagent le même espace d'adressage, et qu'il n'y a pas de commutation de contexte. Uniquement une pile (pour les variables locales et les paramètres des fonctions) est recréée pour chaque thread. Il ne faut pas donc se priver d'avoir plusieurs threads si l'application est naturellement constituée d'actions qui peuvent être menées "simultanément".

On trouve souvent deux types de threads :

- threads pris en charge par le système : leurs créations et leurs gestions est à la charge du système d'exploitation
- threads inconnus du système : c'est le processus qui héberge les threads qui se charge de la création et de la gestion, via une librairie ou un support natif par le langage de programmation.

Dans notre contexte d'extension d'un langage de programmation pour le support de la concurrence, on va se restreindre uniquement aux concepts nécessaires.

2.3 Le modèle de concurrence proposé

Actuellement, il y a beaucoup de langages de programmation qui supportent la concurrence. Ce support est appelé le *modèle de concurrence*. Il dépend essentiellement de la manière d'échanger de l'information entre les différents composants d'un programme.
Comme exemples de modèles de concurrence, on trouve:

- Les structures de données partagées et synchronisées, comme en Java et en C#.
- L'échange synchrone de messages comme en Concurrent ML **[Matt97]** et en Concurrent Haskell **[Jone96]**.
- La technique du rendez-vous comme en Ada **[Ichb83]** et en Concurrent C **[Geha92]**.
- L'échange asynchrone de messages comme en ERLANG **[Arms96]**.

La technique la plus commune actuellement est la synchronisation à travers des structures de données partagées. Cependant, un certain nombre de techniques basées sur l'échange de messages, sont apparues ces dernières années, après l'avènement des réseaux de communication. Et ce modèle de concurrence est devenu convenable pour ce type d'applications. C'est cette tendance que nous allons suivre pour l'extension de Caml light.

Les caractéristiques les plus importantes d'un langage de programmation concurrent sont la synchronisation et les primitives de communication. Celles-ci peuvent être divisées en deux grandes catégories: les primitives à mémoire partagée et les primitives à mémoire distribuée (ou échange de messages).

Les langages à mémoire partagée utilisent un état partagé mutable pour implémenter la communication entre processus, de telle manière qu'il est nécessaire de verrouiller la mémoire alors qu'elle est utilisée. Lorsqu'il y a des verrous, il y a des clés qui peuvent se perdre. Nous considérons que les verrous sont un mécanisme de bas niveau qui n'a pas sa place dans un langage de haut niveau comme Caml Light. Ce n'est pas le cas dans Java qui use de ces verrous. Aussi, les langages concurrents à mémoire partagée reposent sur un état mutable pour la communication interprocessus. Cela

conduit à un style impératif, qui va à l'encontre du style essentiellement fonctionnel des programmes Caml Light.

Pour ces raisons, nous pensons que les primitives à mémoire partagée ne sont pas adaptées à une extension concurrente et surtout distribuée de Caml Light, et que nous n'allons pas utiliser.

L'autre classe des primitives de concurrence est la mémoire distribuée (ou échange de messages). Les primitives ou opérations de base dans la communication par échange de messages sont "envoyer un message" et "recevoir un message", et sont toutes les deux utilisées pour la communication et la synchronisation. Il existe deux grandes classes pour la sémantique de l'échange de messages: l'envoi de messages non bloquant (ou échange asynchrone de messages), et l'envoi de messages bloquant (ou échange synchrone de messages).

L'échange asynchrone de messages dans les systèmes distribués, est aussi le plus souvent asynchrone, et la communication asynchrone maximise le parallélisme **[Repp92]**. Les communications asynchrones sont simples et plus primitives que celles synchrones, même si chacune peut simuler l'autre. L'asynchronisme est le choix fait par plusieurs approches théoriques mais moins par des plateformes réelles, par exemple le modèle client-serveur utilise généralement les communications synchrones. Aussi, plusieurs systèmes ou langages possèdent des communications synchrones : EJB, CORBA, RMI **[Emme01]**. Mais, ces toutes dernières années des intérêts commencent à se manifester par les communications asynchrones, et c'est le cas du langage ProActive **[Proa]** en cours d'élaboration à l'INRIA, qui est un langage asynchrone et basé sur Java, et surtout le cas du langage ERLANG **[Arms07]**.

C'est pourquoi nous avons opté pour l'échange asynchrone de messages – c'est-à-dire "envoyer ne-pas-attendre " – en Caml Light, et d'ici, notre extension sera un exemple de langage de programmation, comme ERLANG, basé sur l'échange asynchrone de messages, dans la classe des langages à mémoire distribuée.

Les communications asynchrones sont moins contraignantes d'un point de vue concurrence mais l'émetteur ne sait pas si un message sera reçu par un récepteur. Un autre fait important avec l'asynchronisme, c'est l'impossibilité de différencier un composant lent d'un composant arrêté et son impact sur la gestion des erreurs. D'un autre côté les communications synchrones consomment plus de temps.

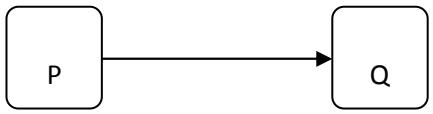
ERLANG est un des rares langages de programmation ayant utilisé l'envoi asynchrone de messages et ayant prouvé son efficacité dans beaucoup d'applications. Le langage ERLANG commence à influencer la communauté de conception de langages de programmation, surtout concurrente et distribuée. Huch **[Huch99]** a aussi proposé un Haskell au style d'ERLANG. Nous avons aussi été, dans notre cas, influencé par le modèle de concurrence et de distribution d'ERLANG, c'est pour cela que nous avons vu digne de notre part de présenter ce langage même partiellement, par un bref survol dans la section 2.7.

Avec cette décision d'échange asynchrone de messages, les problèmes de programmation seront résolus en envoyant et en recevant des messages au lieu d'écrire et de lire dans des variables partagées protégées par des sémaphores ou des moniteurs. Ceci nous offre un nouveau style pour résoudre les problèmes habituels. Bien sûr, nous avons besoin d'implémenter un buffer – du fait que l'envoi de messages est non bloquant – que nous allons appeler *mailbox* (boite à messages). Ce buffer agit comme un canal mais il fait partie du récepteur.

Donc les deux nouveaux ingrédients ajoutés pour une extension concurrente du langage fonctionnel Caml Light sont:

- Les processus, et un mécanisme pour la création et le lancement de processus
- La communication interprocessus

La création de processus et de la communication interprocessus sont explicites.
La forme de désignation de processus est directe et asymétrique:



Envoi de message à Q Toute réception

2.3.1 Architecture des threads (indication d'implémentation)

L'architecture que nous proposons pour la gestion de la mémoire des threads est une architecture à mémoire privée c'est-à-dire chaque thread possède sa mémoire privée essentiellement découpée en deux parties : un tas (*heap*) privé et une pile (*stack*) privée. Le tas est utilisé pour les arguments de fonctions, les variables locales et les adresses de retour. Le tas contient les listes, les tuples …

Les deux zones mémoires, tas et pile, sont co-localisées, c'est à dire que le tas et la pile évoluent chacun vers l'autre. L'avantage est qu'un débordement est facilement détecté, juste par la comparaison du pointeur de tas avec le pointeur de pile qu'on peut garder dans des registres pour des raisons de performance. Un inconvénient, bien sûr, est qu'une expansion du tas ou de la pile engage les deux zones.

Tout ceci doit être pris en considération au niveau de l'interpréteur du bytecode généré par le compilateur Caml Light. Une implémentation est en cours afin de rendre Caml Light concurrent. Le langage Caml Light est typiquement séquentiel.

2.4 Création de processus

Un processus est une fonction évaluée dans un *thread* (fil d'exécution) séparé, qui encapsule un calcul concurrent. A tout moment, un nombre arbitraire de processus peuvent être exécutés simultanément. A chaque fois qu'un thread est créé, une nouvelle mémoire privée est créée et associée avec ce thread, et il n'y a aucun moyen qu'un thread accède à la mémoire privée d'un autre thread.

Un processus n'aura pas de notation spéciale, nous utilisons la même notation pour les modules. **spawn_process** est la nouvelle primitive proposée, qui crée et lance un processus concurrent:

spawn_process (*mod__func , arg*) où:

• __ sont deux caractères de soulignement

• *mod*: est le module qui contient la fonction *func*. *mod* peut être omis si *func* est locale

• *func*: est le nom d'une fonction

• *arg*: est l'argument de la fonction *func*

La primitive **spawn_process**, avec la signature:

```
value spawn_process: ('a → 'b) → 'a → int ;;
```

prend deux arguments, le premier argument est une fonction qui est exécutée dans le nouveau thread, et le second argument est l'argument de la fonction. Les arguments de la fonction doivent être passés en tant que structure de données. **spawn_process** crée un processus concurrent qui effectue ou évalue la fonction *func* avec son argument *arg* contenue dans le module *mod*. **spawn_process** est asymétrique: si un processus exécute un **spawn_process**, le processus fils correspondant est exécuté de manière concurrente avec le processus père. **spawn_process** retourne immédiatement, un *PId* (Identificateur de Processus), sans attendre la fin de l'évaluation de la fonction *func*. *PId* est utilisé pour communiquer avec le processus créé, et il est connu uniquement par celui qui l'a créé.

Un processus se termine une fois sa fonction évaluée et ne renvoie aucun résultat. Pour le moment, le *PId* est de type entier, et il peut être manipulé (comparé à d'autres PIds, stocké dans une liste, ou envoyé comme message à d'autres processus).

La primitive **spawn_process** doit être précédée par une directive Caml Light # `open ("mod");;` si la fonction *func* est définie dans un module séparé. Dans ce cas, le module *mod* peut être omis dans la primitive **spawn_process**.

Voici un exemple :

```
(* fichier  processus.ml *)
let  p = ref 0;;

let  start() =
   p := spawn_process(run, ());
   print_string("Spawned ");
   print_int(p);
   print_newline();
   flush(std_out);;

let run() = print_string("Hello world !\n");;

(* fichier processus.mli *)
value start: unit → unit;;
```

Dans un autre module, test par exemple, on peut utiliser:

```
(* fichier test.ml *)
#open “processus”;;
start();;
```

2.5 Communication interprocessus

Dans ce modèle proposé, chaque processus possède une boîte à messages, et tous les messages qui sont envoyés au processus sont stockés dans la boîte à messages dans l'ordre de leur arrivée. Cette boite à messages sera implémentée soit :

- Par un buffer de taille limitée : s’il y a débordement du buffer, il sera difficile de détecter l’erreur (débordement ou autre)

- Buffer de taille illimitée : il faut prévoir une allocation dynamique

Nous introduisons aussi un nouveau type appelé **`messages`** comme un type de données algébrique pour définir les messages que le processus peut recevoir:

```
type messages
                Cons1 t11…t1n1
               |Cons2 t21…t2n2
               ...
               |Consk tk1…tknk ;;
```

Les $Cons_i$ sont des constructeurs de type:
$t_{i1} \rightarrow t_{i2} \rightarrow \ldots \rightarrow t_{ini}$

La primitive **send_message**, que nous verrons par la suite, n'envoie pas directement un message à un processus, mais dépose un message dans la boîte à messages du processus. La primitive **receive_message** aussi, ne reçoit pas directement un message à partir d'un processus, mais tente de lire et de supprimer un message dans la boîte à messages. Dans une certaine mesure, l'émetteur et le récepteur du message ne se synchronisent pas, c'est la nature même de l'échange asynchrone de messages.

Les primitives **send_message** et **receive_message**, avec la même sémantique que dans ERLANG, fonctionnent comme suit:

2.5.1 La Primitive send_message

La primitive send_message possède la signature:

```
value send_message: Process_Id → 'a →'a ;;
```

est utilisée pour envoyer des messages et a la syntaxe et la sémantique suivantes :

```
send_message (PId, Message)
```

Le message *Message* est envoyé au processus *PId*. Plus exactement, la primitive **send_message** évalue ses arguments *PId* et *Message* ; *Message* est placé dans la boîte à messages du processus avec identificateur *PId*, et renvoie *Message* comme valeur. L'émetteur ne mémorise donc pas les messages. Un processus

qui désire envoyer un message n'est pas suspendu jusqu'à ce qu'un processus désirant recevoir ce message soit trouvé, et vice versa.

On suppose que les messages arrivent toujours au récepteur, en d'autres termes on suppose que nous avons des communications fiables ou sûres. Les messages perdus sont hors portée de cette étude, on peut consulter **[Raja02]** sur ce sujet.

Afin de recevoir les messages d'un processus spécifique, le langage Caml Light étendu permet à un émetteur d'inclure son propre *PId* dans le message, qui peut être obtenu avec la primitive **self ()**:

```
value: self: unit → int;;
```

Cela peut être fait par: **send_message** (**self ()**, *PId*, *Message*)

2.5.2 La Primitive `receive_message`

Les messages sont reçus dans le *mailbox* du récepteur et lus de manière asynchrone par le récepteur.

La primitive **receive_message** possède la signature:

```
value  receive_message: 'a mailbox → 'e;;
```

est utilisée pour recevoir ou accepter des messages.

La syntaxe et la sémantique de la primitive **receive_message** est:

```
receive_message
with  pattern1   →   expression1
      |    pattern2   →   expression2
 ...
      |    patternn    →   expressionn
end
```

Le caractère "_" à la fin de la primitive **receive_message**, s'il est utilisé, doit être lu «autre» comme en Caml Light. Quand un programme évalue une primitive **receive_message**, sa boîte à messages est examinée et le premier message dans la boîte à messages est comparé avec les motifs $pattern_1$ jusqu'à $pattern_n$. Si la correspondance de $pattern_i$ réussit, l'expression associée $expression_i$ est évaluée, et sa valeur devient la valeur de la primitive **receive_message**. Si aucun des motifs ne correspond à ce premier message dans la boîte à messages, il est enregistré pour traitement ultérieur. Si la correspondance réussit ou échoue, ce premier message est supprimé de la boîte à messages du processus qui exécute la primitive **receive_message**. Cette procédure est répétée jusqu'à ce qu'une correspondance soit trouvée ou jusqu'à ce que tous les messages dans la boîte à messages soient examinés. Si aucun des messages dans la boîte à messages ne correspond, le processus qui évalue **receive_message** est suspendu jusqu'à ce qu'un message soit enregistré dans la boîte à messages. Tous les messages sauvegardés pour traitement ultérieur, sont de nouveau remis dans le même ordre dans la boîte à messages, quand un nouveau message arrive et sa correspondance réussit.

Nous avons choisi la même syntaxe de la constrution **match** du langage Caml Light pour la primitive **receive_message**.

Afin de recevoir des messages uniquement d'un émetteur particulier avec *PId* comme identifiant, nous pouvons faire:

```
receive_message
…     {PId, pattern} →      …
```

On peut aussi ajouter un délai (temps), exprimé en millisecondes, comme en ERLANG, à la primitive **receive_message** afin d'éviter à un processus d'attendre un message qui n'arrivera jamais. La syntaxe est:

```
receive_message
    with  pattern1   →   expression1
     ...
          patternn   →   expressionn
          after time →   expressiont
    end
```

2.6 Exemples en Caml Light étendu

2.6.1 Une calculatrice de bureau concurrente

Pour illustrer l'utilisation des primitives de concurrence proposées, voici un exemple plus complet de programme concurrent écrit dans le langage Caml Light étendu.

Nous proposons ici une petite application concurrente client-serveur. Le serveur calcule la somme, la négation ou le produit de deux entiers fournis par un client. Le client et le serveur sont des processus séparés, et l'échange asynchrone de messages proposé est utilisé pour la communication entre eux. Nous supposons que le client et le serveur fonctionnent sur la même machine. On désire (un client) envoyer une requête à un processus (le serveur), puis attendre une réponse du serveur. Pour envoyer une requête, un client doit inclure une adresse à laquelle le serveur peut répondre. Nous écrivons une fonction appelée ***rpc*** (pour ***r****emote* ***p****rocedure* ***c****all* : appel de procédure à distance) qui encapsule l'envoi d'une requête à un serveur et attend une réponse de celui-ci.

Nous allons mettre cela dans un module appelé **`serveur_cal`** pour serveur de calculatrice et stocker le module dans le fichier appelé `serveur_cal.ml`.
L'ensemble du module ressemble à ceci:

```
(* fichier serveur_cal.ml *)
let lancer() =  spawn_process(loop, ());;

let evaluer PId expr =  rpc(PId,expr);;

let rpc p requete =
   send_message(self(),p,requete);
   receive_message
   with
      {p,reponse} → reponse
   end;;

let loop() =
   receive_message
   with
      (de,(somme,a,b))   →  send_message(self(),de,a+b);
     loop();
      (de,(moins,a,b))   →  send_message(self(),de,a-b);
          loop();
      (de,(produit,a,b)) →  send_message(self(),de,a*b);
          loop();
               (de,_) →  send_message(self(),de,erreur_);
     loop();
   end;;

(* fichier serveur_cal.mli *)
value lancer: unit → int;;
value evaluer: int → int → int;;
```

Les fonctions *rpc* et *loop* sont cachées à l'intérieur du module **serveur_cal**. Un module client n'a besoin que de deux fonctions ; la fonction *lancer* pour créer le serveur et la fonction *evaluer* pour effectuer ses calculs.

Ce module se présente comme suit:

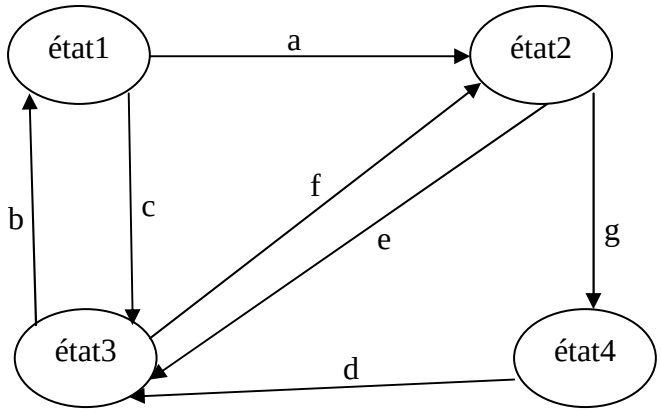
```
(* fichier un_client.ml *)
#open “serveur_cal”;;

let  p = ref 0;;
p := lancer(); evaluer(p,(somme,5,3));
evaluer(p,(produit,4,9));;
```

2.6.2 Un automate à états finis

Dans cet exemple, on veut montrer comment les primitives de concurrence peuvent être exploitées à d’autres fins, comme la primitive **`receive_message`**.

La figure suivante montre un automate à états finis (EAF) avec quatre états (état1, état2, état3, état4) et plusieurs transitions et les évènements qui les produisent (a,b,c,d,e,f,g) :



On va représenter chaque état par une fonction Caml Light, et chaque évènement par un message. La primitive **`receive_message`** sera utilisée pour accepter les évènements ou les messages. Le code, en Caml Light étendu, qui simule le comportement d’un automate à états finis, est le suivant :

```
(* fichier auto_etats_finis.ml *)
let lancer_aef() =  spawn_process(etat1, ());;
let etat1() =
     receive_message
     with
          a    →    etat2();
          c    →    etat3();
     end;;
let etat2() =
     receive_message
     with
          e    →    etat3();
          g    →    etat4();
     end;;
let etat3() =
     receive_message
     with
          b    →    etat1();
          f    →    etat2();
     end;;
let etat4() =
     receive_message
     with
          d    →    etat3();
     end;;
(* fichier auto_etats_finis.mli *)
value lancer_aef(): unit → int;;
```

Une utilisation :

```
(* fichier utilisation.ml *)
#open “auto_etats_finis”;;
let  p = ref 0;;
p := lancer_aef();
send_message(p,a);
send_message(p,f); ...
```

2.7 ERLANG

ERLANG, du nom d'un mathématicien Danois, Agner Krarup Erlang (1878-1929), est un langage de programmation fonctionnelle développé par Joe Armstrong, Robert Virding, Claes Wikström et Mike Williams dans le laboratoire d'informatique d'Ericsson [**Bark99**]. Le langage ERLANG est dédié à la programmation d'applications industrielles, surtout dans le domaine des télécommunications, dépendantes de caractéristiques comme : la tolérance aux pannes, la concurrence, la distribution et le temps réel [**Arms07**].

ERLANG est d'abord un langage de programmation fonctionnelle où un programme est composé de plusieurs fonctions stockées dans des modules. Les fonctions peuvent s'exécuter de manière concurrente dans des processus légers (threads), et communiquent entre elles par échange asynchrone de messages. La création et la suppression de processus exigent peu de mémoire et de temps de calcul.

En ERLANG la mémoire est allouée dynamiquement. Un gestionnaire de mémoire libère la mémoire qui n'est plus référencée. Le partage mémoire n'existe pas, les processus sont obligés d'échanger les données par envoi de messages.

Un mécanisme de gestion des erreurs, une fonctionnalité phare du langage, permet à un processus de traiter des exceptions créées par l'évaluation d'expressions incorrectes.

Un autre aspect important du langage ERLANG, est son mécanisme de distribution basé sur TCP/IP. Les processus peuvent inter agir entre eux sur plusieurs machines physiques, ou sur la même machine en exécutant différents systèmes ERLANG.

Le système ERLANG offre plusieurs fonctions prédéfinies appelées les BIF (Built-In Functions) qui permettent d'étendre le langage pour des opérations non disponibles directement dans le langage.

Dans un environnement de développement, et à la manière du langage LISP ou PROLOG, les fonctions sont évaluées dans le shell du système ERLANG. C'est un interpréteur de commandes interactif qui permet à un utilisateur de charger, compiler et évaluer des expressions. Il permet aussi d'inspecter l'état du système.

Les programmes peuvent aussi être compilés pour obtenir des exécutables qui peuvent être lancés sur des environnements depuis les PC jusqu'aux systèmes embarqués.

2.7.1 Types de données

Les variables ERLANG sont dynamiquement typées, et n'ont pas besoin d'être déclarées. Une variable est initialisée lors de sa création et ne peut plus être modifiée, c'est un langage à affectation unique.

ERLANG possède les types de données suivants :

- Les constantes :
  - Numériques (entiers et réels)
  - Atomes (abc, bonjour_tout_le_monde …)
- Les types de données composés :
  - Les tuples : similaires aux enregistrements et structures dans d'autres langages conventionnels : {a, 12, b}, {}, {1, 2, 3} …
  - Les listes : [a, 12, b], [], [1, [2,0], 3] … comme les listes de LISP

Une valeur peut être stockée dans une variable :

```
X = {livre, préface, remerciements, contenu, {chapitres [
                                                {Chapitre, 1, 'introduction'},
                                               {Chapitre, 2, …}
                                               ] } }
```

2.7.2 Les fonctions et les modules

Un programme ERLANG consiste en un ensemble de modules, où chaque module défini un certain nombre de fonctions. Un module est identifié par son nom (atome), alors qu'une fonction dans un module est identifiée par son nom (atome) et son arité.

Pour chaque module, seulement les fonctions explicitement déclarées exportables peuvent être accessibles depuis d'autres modules, les autres sont implicitement locales. Voici un exemple d'un module ERLANG [**Arms96**]:

```
-module (math).          % math: le nom du module
-export ([double/1]).    % double: fonction à exporter avec
                         un argument
double (X)  →
          fois (X, 2).
fois (X, N) →            % fois : fonction locale
          X * N.
```

Les appels aux fonctions peuvent être qualifiés avec le nom du module, en utilisant la syntaxe **module : fonction (…)** comme avec l'évaluation de **math:double(8)** qui donnera 16.

2.7.3 Le filtrage

Le filtrage (pattern matching) ou l'unification, est un outil clé dans le langage ERLANG. Le filtrage est utilisé pour faire correspondre des motifs avec des termes. Si la correspondance entre un motif et un terme se produit, toute variable qui apparait dans le motif sera liée à la structure de données qui apparait dans le terme à la position correspondante. Par exemple, Il est utilisé pour affecter des valeurs à des variables, par la primitive **=** de filtrage : motif **=** terme

Exemple :

```
>  N = {12, Bananes}.
```

```
{12,  Bananes}
>  {A, B} = N.
{12, Bananes}
>  A.
12
>  B.
Bananes
```

Le filtrage est aussi utilisé lors de l'appel d'une fonction ou pour déterminer le flux de contrôle dans un programme.

2.7.4 La concurrence dans ERLANG

ERLANG [**Arms07**] est un langage où la concurrence fait partie du langage de programmation et non pas du système d'exploitation. Les mécanismes de la programmation concurrente sont intégrés dans le langage même.

Un programme ERLANG est constitué de beaucoup de processus qui fonctionnent de façon indépendante. Ils communiquent les uns avec les autres en échangent des messages par l'échange asynchrone de messages, la seule méthode par laquelle les processus peuvent échanger des données.

Un processus est initié à l'aide d'une fonction dont le rôle est de déterminer la nature et le comportement du processus, il est créé lors de l'appel de cette fonction et disparait lorsque l'exécution de la fonction se termine.

Un processus dispose d'une boite de messages (mémoire) dans laquelle il peut récupérer ou attendre un message vérifiant certaines propriétés. Le système, en utilisant la technique du filtrage, tente de faire correspondre les messages dans la boîte de messages avec les motifs, à l'aide de l'instruction *receive*.

Un processus peut, bien entendu, envoyer des messages à un autre processus à l'aide de l'opérateur d'envoi de message (**!**). Pour répondre à ce besoin, chaque processus est identifié par un *pid* (*process identifier*).

Dans l'exemple qui suit, un processus attend des messages, et à leur réception il affiche leur contenu :

```
-module (exemple).
-export ([processus/1]).
processus (Compteur) ->
   receive
      stop ->
          io:format("Le processus se termine après avoir
          reçu ~p messages~n",[Compteur]);
      Message ->
         io:format("Message numéro ~p = ~p~n",[Compteur+1,
       Message]),processus(Compteur+1)
   end.
```

Dans le module `exemple`, la fonction `processus` utilise l'instruction **`receive`** avec deux clauses pour recevoir ou accepter des messages. Si le message reçu est 'stop', le processus affiche le message 'le processus se termine …' et s'arrête. Si le message reçu n'est pas 'stop', alors il est affecté à la variable `Message` et le contenu et le numéro de ce message sont affichés. Dans ce cas, la fonction `processus` se réappelle en incrémentant le compteur, pour se mettre en attente d'autres messages provenant d'autres processus.

2.7.5 Conclusion sur ERLANG

On peut dire que le langage ERLANG a été développé dès le départ comme étant langage orientée concurrence. Les primitives de concurrence font partie intégrante du langage. Il a été conçu pour gérer un nombre important de processus simultanés. Actuellement, ERLANG (on peut se permettre aussi de dire **ER**icsson **LANG**uage) est

utilisé dans plusieurs types d'applications comme les télécommunications et les applications temps-réel. ERLANG est en open source depuis quelque temps, il fonctionne sur UNIX, Microsoft Windows et sur les systèmes embarqués. Il peut être utilisé en mode interprété ou en mode natif.

2.8 Conclusion

Nous avons décrit la conception d'une extension du langage Caml Light, avec le support de la programmation concurrente. A cette fin, quelques primitives simples de concurrence ont été ajoutées à Caml Light en utilisant le style de ERLANG qui supporte la concurrence et l'échange de messages asynchrone. Quelques exemples sont donnés, pour montrer l'utilisation des primitives que nous avons ajoutées à Caml Light. Ce modèle, que nous pensons très naturel, sera aussi utilisé pour la distribution.

Chapitre 3

Caml Light distribué

Dans ce chapitre, nous allons introduire les primitives proposées pour une extension du langage Caml Light pour la distribution, que nous allons utiliser pour écrire des programmes distribués. D'abord il est nécessaire de définir ce que c'est qu'un système distribué. Nous ne donnerons ici qu'une vue globale de la distribution, le lecteur intéressé par ce sujet pourra consulter **[Tane92]**.

3.1 Définitions

On parle souvent de :

- systèmes concurrents
- systèmes parallèles
- systèmes distribués

Nous allons tenter d'expliquer très brièvement chacun des termes suivants, malgré l'inexistence d'un véritable consensus. Néanmoins il y a quelques points communs entre ces différents systèmes qui se partagent aussi souvent le même genre de problèmes (partage de ressources, coordination …). Des fois ils peuvent même être entremêlés, comme avoir une programmation concurrente (système concurrent) sur une machine avec plusieurs processeurs (système parallèle).

Généralement lorsque nous avons des éléments (processus) qui s'exécutent dans un même espace de ressources comme dans les systèmes d'exploitation multitâches actuels (UNIX ou Microsoft Windows sur un PC) : il s'agit de *systèmes concurrents*.

L'existence des *systèmes parallèles* est souvent due lorsqu'il s'agit de découper un problème de taille en plusieurs sous problèmes moins complexes et que chacun des sous problèmes soit traité par un processeur, nous avons donc un système multiprocesseur (super calculateurs …). Les processeurs fonctionnent simultanément.

Pour ce qui est des *systèmes distribués* et selon Tanenbaum **[Tane02]**, « Un système distribué est une collection d'ordinateurs indépendants qui apparait comme un seul système pour ses utilisateurs ». Ce qui induit implicitement un environnement

d'éléments répartis dans l'espace, et communiquant les uns avec les autres, mais qui apparait du point de vue de l'utilisateur comme une entité unique, par opposition à un système centralisé où tout est localisé sur la même machine.

3.2 Intérêts des systèmes distribués

Les systèmes distribués sont généralement utilisés (s'ils ne sont pas imposés !) pour des raisons de :

- Partage de ressources distantes entre utilisateurs :
  - Bases de données : système de réservation aérienne
  - Serveur de fichiers : utiliser ses fichiers à partir de n'importe quelle machine
  - Périphériques : imprimante laser couleur
- Performance : puissance de calcul et de stockage
- Fiabilité: la défaillance d'une machine n'affecte pas la disponibilité des autres
- Distribution naturelle de certaines applications
- …

On peut trouver comme exemples de systèmes distribués :

- Calcul distribué
- Serveur Web (World Wide Web) : Un serveur Web auquel se connecte un nombre quelconque de navigateurs Web (clients)
- Système de réservation de place d'avions
- Système de gestion d'agences bancaires
- Applications Clients/Serveur

Les problèmes qu'on peut rencontrer dans les systèmes distribués :

- Désignation et communication

- Hétérogénéité des réseaux, des machines, des systèmes d'exploitation et des logiciels
- Sécurité : authentification, cryptographie
- Logiciels : moins de logiciels disponibles et peu d'expérience
- Réseaux : saturations, délais, pertes de messages …

3.3 Le modèle de distribution

Le langage Caml Light a été étendu pour servir à la conception de programmes ou systèmes distribués. Les programmes distribués sont les programmes conçus pour être exécutés sur des réseaux d'ordinateurs et qui peuvent coordonner leurs activités, par échange asynchrone de messages, dans notre cas.

Dans notre extension de Caml Light, les programmes sont écrits pour être exécutés sur des réseaux de *nœuds*. On peut créer et lancer un processus sur n'importe quel nœud, le mécanisme d'échange asynchrone de messages employé pour la concurrence fonctionnera comme dans le cas d'une seule machine ou d'un seul nœud.

Toutes les primitives proposées pour la concurrence auront les mêmes propriétés dans un système distribué que dans une seule machine. Ce sera un grand avantage que de conserver la même philosophie de conception, d'une part le programmeur ne se dépaysera pas en allant depuis la concurrence vers la distribution afin d'écrire des applications distribuées, et d'autre part le langage gardera une certaine homogénéité.

Ecrire un programme distribué dans le langage étendu est relativement facile ; il suffit de créer et de lancer les processus sur les bonnes machines, et le tout fonctionnera comme dans le cas de la concurrence. C'est une extension directe des primitives de concurrence vers la distribution en ajoutant le concept de nœud.

On aura besoin donc de savoir deux choses :

- Comment lancer un nœud ?
- Et comment effectuer un appel distant de procédure (mécanisme largement accepté pour supporter la conception d'applications distribuées) ?

3.3.1 Nœuds et connexions

Un nœud est un concept central dans notre extension. Un nœud représente une instance d'une machine virtuelle Caml Light avec son propre espace d'adressage et son propre jeu de processus. On peut considérer un nœud comme un microsystème d'exploitation pour les processus dans le langage Caml light étendu. Chaque processus s'exécute dans un nœud particulier et une fois qu'il est créé il reste sur ce nœud.

Plusieurs nœuds peuvent être situés sur un même ordinateur ou sur plusieurs ordinateurs dans un réseau. Le nom d'un nœud est de la forme *Name@host*, un nom (*Name*) et un hôte (*host*) c'est-à-dire le nom de la machine sur laquelle s'exécute le nœud (typiquement un nom de domaine Internet), séparés par le caractère '@'. La fonction `node()` permet de retourner le nom d'un nœud. Supposons que le nœud *nd1* est en cours d'exécution sur la machine *kamel.univ-khenchela.dz*, le nom complet du nœud *nd1* sera *nd1@kamel.univ-khenchela.dz* sera retourné par `node()`. Le programmeur n'a pas à initialiser les connexions entre les nœuds. Elles sont faites par le système exécutif, quand un nœud, pour la première fois, est référencé par un autre nœud.

3.3.2 Authentication (indication d'implémentation)

La création de nœuds engage quelques aspects de sécurité qui ne sont que partiellement gérés par l'environnement d'exécution.

L'accès à un nœud ou à un ensemble de nœuds est sécurisé par un système de cookies **[Yang06]**. Chaque nœud possède un seul cookie, et ce cookie doit être identique aux cookies des nœuds avec lesquels ce nœud communique. Donc, tous les nœuds dans un système Caml Light distribué, qui veulent communiquer entre eux, doivent être lancés avec le même *cookie*. L'ensemble des nœuds connectés possédant le même cookie est appelé *cluster*.

Pour initialiser un même cookie pour deux nœuds qui veulent communiquer, on utilise la méthode d'implémentation suivante :

- Enregistrement d'un même cookie dans le fichier $HOME/.camllight.cookie. Ce fichier peut être copié, par son propriétaire, sur les différentes machines qui vont entrer en communication. Il peut être créé, avec les commandes Unix suivantes :
  - $ cd
  - $ cat > .camllight.cookie
  - TYRU75654476YHG
  - chmod 400 .camllight.cookie

La dernière commande permet de protéger le fichier : accès exclusif au propriétaire.

3.3.3 Les primitives de distribution

Les primitives ajoutées à Caml Light pour l'écriture de programmes distribués sont comme suit:

- La primitive **`rspawn_process`** (**`r`** pour *remote* : à distance), un **spawn** étendue, est: **`rspawn_process`** (*`nœud, mod__func, arg`*)**`.`**
  Elle engendre un nouveau processus sur *nœud*, en appliquant la fonction *`func`*, localisée dans le module *`mod`*, avec comme argument *`arg`*. **`rspawn_process`** retourne l'identificateur du processus créé.

- Envoyer un message à un processus distant:
  `rsend_message` (*`nœud, PID, Message`*)**`.`**
  Elle envoie le message *`Message`* au processus *`PID`* sur le nœud *`nœud`*.
- **`register_process`**(*`nom`*,`p`)**`;;`**
  Permet de publier le PId p d'un processus connu sous le nom *nom*, afin de permettre à d'autres processus de communiquer avec celui-ci.

3.4 Un exemple de serveur de noms

Voici un exemple d'un serveur de noms écrit dans cette extension distribuée de Caml Light.

Ce serveur de noms, appelé serveur_noms, est un programme, qui à partir d'un *nom*:

- Retourne une valeur associée à ce *nom*
- Ou stocke une valeur associée à ce *nom*

Nous supposons l'existence d'un dictionnaire global qui peut être manipulé par deux fonctions **put** et **get**. La fonction **put** ajoute une nouvelle valeur, un entier différent de 0, au dictionnaire et associe cette valeur à *nom* (une chaîne de caractères) et retourne la valeur *true*. La fonction **get** retourne la valeur associée à *nom* dans le dictionnaire, autrement elle retourne 0 si aucune valeur n'est associée à *nom*.

```
(* fichier serveur_noms.ml *)
let p = ref 0;;

let lancer () =   p := spawn_process(loop, ());
register_process(serveur_noms,p);;

let enregistrer nom valeur =
          rpc((enregistrer,nom,valeur));;

let rechercher nom = rpc((rechercher,nom));;

let rpc arg =   send_message(self(),p,arg));
   receive_message
   with
      (p,reponse) → reponse
   end;;

let loop() =
   receive_message
   with
     (from,(enregistrer,nom,valeur)) → put(nom,valeur);
          send_message(self(),from, true); loop();
     (from,(rechercher,nom)) →
          send_message(self(),from, get(nom)); loop();
   end;;
```

```
(* fichier serveur_noms.mli *)
value lancer: unit → int;;
value enregistrer: string → int -> bool;;
value rechercher: string → int;;
```

Au niveau du *shell*, on fait un test local en lançant l'exécution d'une instance de la machine virtuelle de Caml Light étendu appelée **cle** (en cours de développement) :

$ **cle**
> `serveur_noms__lancer()` donne - : le PId du serveur
> `serveur_noms__enregistrer("nom1",valeur1)` donne - : true
> `serveur_noms__enregistrer("nom2",valeur2)` donne - : true
> `serveur_noms__rechercher("nom2")` donne - : valeur2
> `serveur_noms__rechercher("nom3")` donne - : indéfini

Maintenant, pour avoir un client sur une machine (*hote1*) et un serveur sur une machine séparée (*hote2*), dans un réseau comme *Internet*, nous avons à créer deux nœuds ou machines virtuelles, un pour le client (*noeud1*) et un pour le serveur (*noeud2*), sur les deux machines différentes, avec l'interpréteur. Alors, on peut envoyer quelques commandes de *noeud1* sur *hote1* à *noeud2* sur *hote2*. Sur la machine *hote2*, on lance l'exécution d'une instance de la machine virtuelle. Cette instance aura pour nom *noeud2*. Sous le contrôle de cette machine virtuelle, on lance le serveur de noms :

hote2 $ **cle** *noeud2* cookie1
cle > serveur_noms__lancer()
PId
cle > **…**

Sur la machine *hote1* distante, on lance l'exécution d'une autre instance de la machine virtuelle (un client). Cette instance aura pour nom *noeud1*. Sous le contrôle de *noeud1* on lance deux requêtes adressées au serveur qui tourne sur *hote2*, en envoyant deux messages distants :

hote1 $ **cle** *noeud1* cookie1
cle > rsend_message(**self()**, *noeud2@hote2*, serveur_noms,(enregistrer, "nom1", valeur1))
true
cle > rsend_message(**self()**, *noeud2@hote2*, serveur_noms,(rechercher, "nom1"))
valeur1
cle > **…**

cookie1 représente le nom d'un fichier contenant la valeur d'un même cookie utilisé par *noeud1* et *noeud2* pour former un cluster. Avec le système UNIX, on peut mettre cette valeur dans $HOME/.camllight.cookie1

3.5 Conclusion

Le modèle concurrence a été étendue aisément, vue la simplicité des primitives et leur adaptation à un environnement distant, afin de supporter la distribution, c'est-à-

dire le fait de permettre à plusieurs programmes Caml Light de communiquer uniquement par envoi asynchrone de messages dans un réseau local ou un réseau de réseaux comme Internet. La difficulté, peut être, interviendra au niveau de l'implémentation des nœuds.

Chapitre 4

Vers un Caml Light mobile

Ce chapitre présente une vue d'ensemble de la technologie du code mobile, en situant ce modèle d'exécution distribuée par rapport à d'autres modèles classiques comme l'évaluation à distance, tout en présentant leurs caractéristiques. Le modèle du code mobile est présenté d'une manière assez détaillée. Quelques langages mobiles seront aussi évoqués ainsi que les propriétés minimales et souhaitables qu'un langage mobile devra avoir. Une étude et une proposition est faite pour se rapprocher d'un Caml Light mobile.

4.1 Introduction

4.1.1 Définition

Grâce à l'arrivée des ordinateurs, il a été possible de construire un nouveau type d'applications : les applications distribuées.
Ces applications reposent sur différents modèles appelés *modèles d'exécution distribuée* (passage de messages, exécution à distance …). Parmi ces modèles, il en est un relativement nouveau: le *code mobile* ou des fois appelé :

- Agent mobile
- Calcul mobile
- Composant mobile
- Application mobile

dont une définition :

Un code mobile (CM) est une entité logicielle, qui a la capacité de migrer à partir d'une machine (site source) vers une autre machine (site de destination) pour accéder à des données ou à des ressources. Une fois arrivée, cette entité peut devenir autonome, elle peut encore migrer (en cours d'exécution) sur un autre site. Le moment de migration et le site cible sont décidés par ce code mobile.

Durant cette mobilité, ce code mobile peut accumuler des résultats sur un quelconque site hôte et peut communiquer avec d'autres codes mobiles. Une communication avec le site initial n'est pas nécessaire.

4.1.2 Motivations

Voici quelques applications qui peuvent exploiter ce concept de mobilité :

- Diminution du trafic réseau (*data access locality*)
  - déplacer l'application vers le site qui héberge les informations (données ou services) fréquemment accédées.
- Équilibrage de charge (*load balancing*)
  - pour des besoins (mémoire ou processeur) les applications migrent à partir des nœuds les plus chargés vers des nœuds moins chargés.
- Tolérance aux pannes (*fault tolerance*)
  - En cas de panne d'une machine, une application peut migrer vers un autre site pour être reprise à partir de sa dernière sauvegarde.
- Administration système (*improved system administration*)
  - Faire migrer les applications des sites à réinitialiser ou à rendre indisponibles, vers d'autres.
- Reconfigurer une application dynamiquement
  - ajouter une nouvelle fonctionnalité à une quelconque application de manière dynamique.
- Recherche d'information répartie (moteur de recherche …)
- Commerce électronique

4.2 Les différents modèles classiques d'exécution distribuée

Les paradigmes de programmation des applications distribuées sont :

- Echange de messages

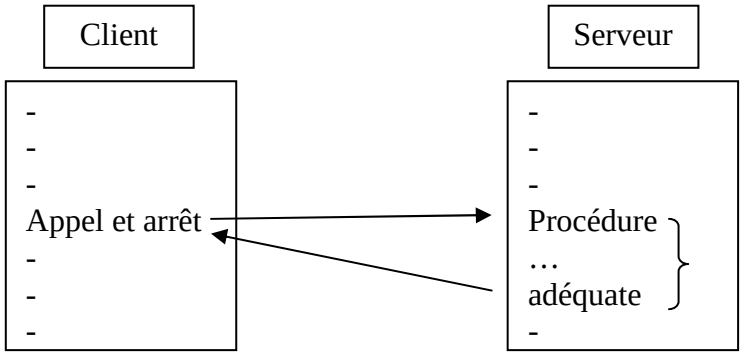
- Appel de procédure distante
- Evaluation à distance
- Code à la demande

4.2.1 Echange de message (*MS* : pour *Message Sending*)

C'est le paradigme le plus ancien. Il permet à différents processus de communiquer explicitement par envoi de messages. Des concepts ont été proposés pour une communication synchrone ou asynchrone. Les aspects (communication et synchronisation) sont à la charge du programmeur. C'est un paradigme non mobile, avec un degré de mobilité nulle. Il n'y a aucune migration de code, sauf un transport de données (les messages).

4.2.2 Appel de procédure distante *(RPC* : pour *Remote Procedure Call*)

Afin de communiquer, les processus appellent des procédures distantes. Ce modèle très familier à la communauté des programmeurs est souvent appelé modèle *Client-serveur*: le site source (client) transmet les données d'entrée au site de destination (le serveur) qui détient le savoir-faire (code = procédure) en plus des ressources nécessaires à l'exécution de ce savoir faire. Les services (les procédures) disponibles sur le site de destination sont fixés d'avance. La communication est synchrone. C'est aussi un paradigme non mobile avec un degré de mobilité nulle. Aucun code n'est transmis.

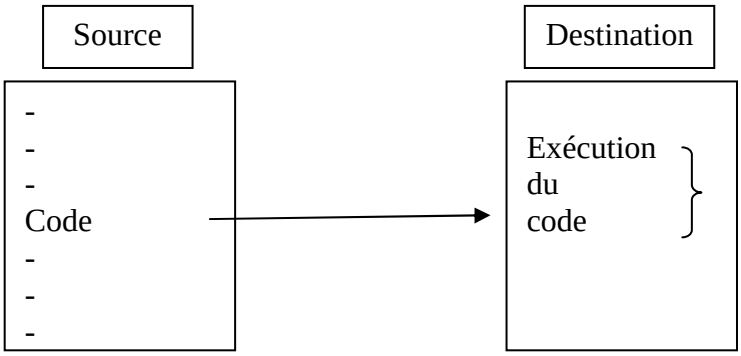


4.2.3 Evaluation à distance (*REV* : pour *Remote EValuation*)

L'évaluation ou l'exécution à distance **[Stam90]** est une généralisation du *RPC*, en plus des données d'entrées fournies, le code est aussi transmis. Il y a donc transfert de code (savoir faire) d'un site source initiateur à un autre site destination où ce code sera exécuté. Le site de destination détient les ressources nécessaires à l'exécution du code, généralement un pouvoir de calcul et non des services spécifiques. L'évaluation à distance peut porter sur une procédure ou tout un programme. On parle de migration de code. Il y a donc un certain degré de mobilité.

Exemples de REV :

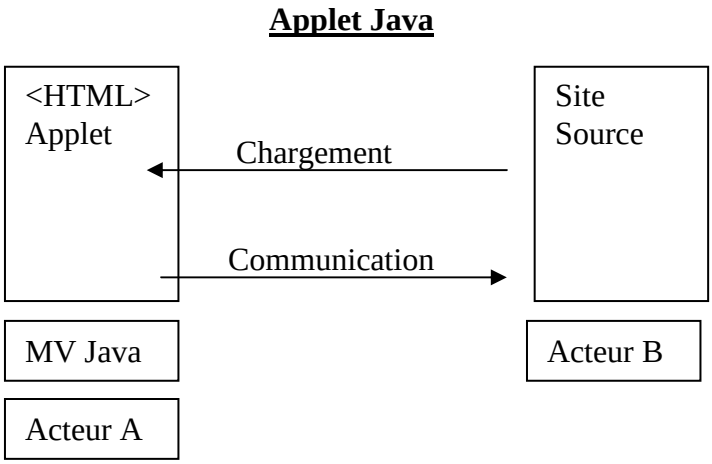
- ✓ *rsh* (*remote shell*): commande UNIX permettant d'exécuter un script, code que fournit le site source, sur une machine distante spécifiée en argument.
- ✓ Impression d'un document Postscript. Le code à transmettre est le document lui-même écrit dans le langage Postscript. Le site de destination comprend l'interpréteur du Postscript (situé dans l'imprimante elle même) et l'imprimante pour imprimer.
- ✓ Un site source désire transférer une application (un filtre), vers un site distant disposant d'une base de données, pour accéder à cette base de données et compresser des données locales. Ainsi on évite d'engager l'administrateur de cette base de données dans cette tâche.



4.2.4 Code à la demande (*COD* : *Code On Demand*)

C'est la situation symétrique à *REV*, encore une généralisation de RPC !
Un acteur A sur un site de destination (initiateur) dispose de ressources mais pas de code, alors qu'un autre acteur B sur un site source détient le code. Le code est alors demandé à B pour être amené localement, migré ou téléchargé, et exécuté sur le site de destination c'est-à-dire le site de l'acteur A.

Exemple: les Applets Java



Comme dans le cas de *REV*, on parle aussi de migration de code pour *COD*, mais dans un autre sens. Il y a donc quand même un certain degré de mobilité.

4.3 Le modèle du code mobile

4.3.1 Définition du code mobile (définition plus concrète)

Le code mobile est surtout apparu non pas pour remplacer les autres modèles d'exécution distribuée qui ont quand même leur domaine d'application, mais pour les compléter à d'autres fins.

Un code mobile ou agent mobile est vue comme:

- une unité d'exécution (programme en cours d'exécution)
  - Un *processus* UNIX
  - Un *Thread* dans un environnement multithread

avec l'adjectif « mobile » qui indique que cette unité d'exécution s'exécutant sur un site source peut se déplacer vers un autre site (machine distante) où l'exécution *reprendra* ou *continuera* : ces deux termes vont donner naissance à deux types de mobilité qu'on discutera par la suite. Une fois sur le site de destination, ce code mobile peut accomplir quelques tâches prescrites.

Le modèle du code mobile est proche de celui de *REV* dans le sens où l'initiateur détient le code des opérations à effectuer et l'envoie pour exécution sur un site distant. La différence réside dans l'asynchronisme des interactions entre les deux sites. Une fois migré sur un site distant, un code mobile devient une entité *autonome*, qui peut suivant l'exécution du code, se déplacer encore vers un autre site, accumuler des résultats, communiquer avec d'autres codes mobiles, signaler un évènement, sans qu'aucune connexion permanente soit maintenue avec le site initial.

Donc, la différence fondamentale avec les autres types de migration, comme avec le cas de *REV*, est :

- L'objectif de déplacement
- L'information déplacée
- L'initiateur du déplacement

Dans les autres types de migration, l'objectif était surtout d'optimiser l'utilisation des ressources réparties en cachant la distribution aux applications. La décision (initiateur) est prise par le système d'exploitation, la migration est ainsi transparente c'est le système qui décide du processus à migrer et de sa destination.

Alors que dans le code mobile, l'objectif est de naviguer en fonction de l'application, d'étendre des fonctionnalités ou de supporter des opérations déconnectées. La décision est prise par l'application elle-même, c'est-à-dire l'agent ou le code mobile,

décide du *moment* et du *lieu* de destination. C'est un mécanisme puissant mais de mise en œuvre très complexe en raison des problèmes d'hétérogénéité matérielle et logicielle et de difficulté de capture de l'état d'exécution, lorsqu'il s'agit surtout d'une mobilité forte (plus loin).

Enfin, on peut dire qu'un code mobile est constitué :

- d'un code exécutable : l'algorithme ou le savoir faire.
- et d'un état.

Cet état est lui aussi divisé en deux parties :

- un état des données : données initiales (variables globales) qui accompagnent le code mobile, spécifiées par le programmeur.
- et un état ou contexte d'exécution : contenu des variables locales des entités actives. En plus, l'état d'exécution renferme des informations de contrôle, en particulier :
    - la valeur du pointeur d'instruction
    - la valeur des points de retour aux procédures appelantes

On distingue deux types de mobilité :

- Code *faiblement* mobile
- Code *fortement* mobile

selon la nature de l'information déplacée.

4.3.2 Mobilité faible et mobilité forte

4.3.2.1 Mobilité faible

Lorsqu'il y a migration :

- du code de l'unité d'exécution
- de l'état des données

Simulation de la migration :

- interruption de l'exécution de l'unité d'exécution sur le site source

- transfert du code et de l'état des données du site source vers le site de destination
- arrivée, l'application mobile, reprend son exécution **depuis le début**, tout en possédant les valeurs mises à jours des données de l'état des données, c'est-à-dire que les données sont conservées et non réinitialisées.

4.3.2.2 Mobilité forte

Lorsqu'il y a migration :

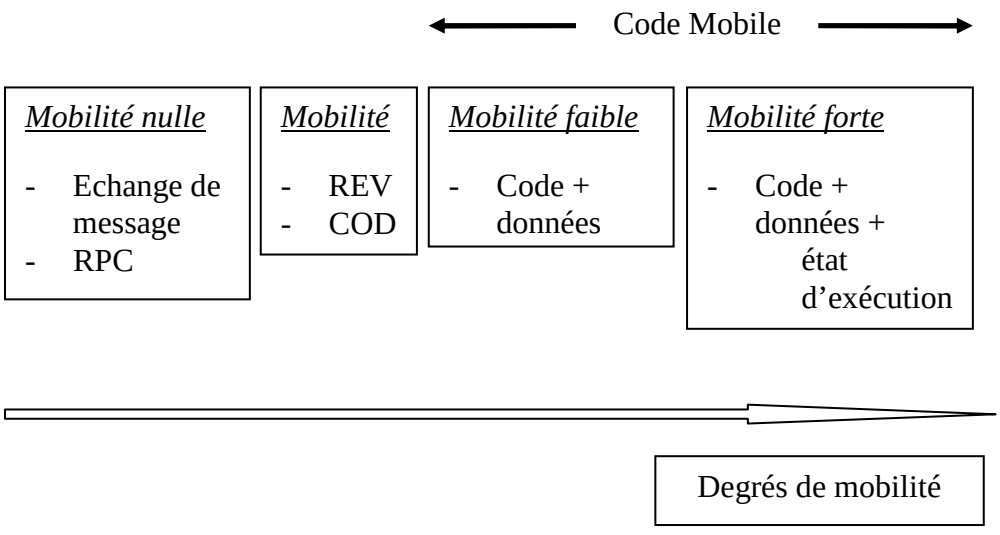
- du code de l'unité d'exécution
- de l'état des données
- et de l'état d'exécution

Simulation de la migration :

- interruption de l'exécution de l'unité d'exécution sur le site source
- transfert du code, de l'état des données et de l'état d'exécution du site source vers le site de destination
- arrivée, l'application mobile, reprend son exécution **au point où elle a été interrompue** sur le site de départ

4.3.3 Résumé des différents modèles (modèles classiques et code mobile)

Voici une schématisation qui nous renseigne sur le degré de mobilité de chacun des paradigmes évoqués précédemment :



L'objectif idéal est la mobilité forte.

4.3.4 Quelques problèmes du code mobile

La mobilité se heurte à de nombreux problèmes, dont beaucoup restent ouverts. En plus de l'hétérogénéité du matériel et du logiciel, la capture et la restauration du contexte d'exécution, dans le cas de la mobilité forte, est un point critique. Un frein majeur aussi, à une réelle utilisation du code mobile est celui de la sécurité. Le fait qu'un code mobile est un « code étranger » et peut être malicieux, alors comment protéger les ressources et les données des machines hôtes contre des attaques par ce dernier ? Le problème devient encore plus complexe lorsqu'une machine hôte peut être à son tour malveillante, intentionnelle ou non, contre un code mobile. Des solutions à ces différents problèmes ont été envisagées mais sont loin de satisfaire.

4.4 Langages mobiles

Les langages mobiles représentent une nouvelle tendance dans les langages de programmation pour les systèmes distribués.

Un langage de programmation mobile est un langage à usage général et qui permet le développement d'applications mobiles. Il doit être capable d'exprimer des codes mobiles, d'exprimer leur transmission, leur réception, et leur exécution ultérieure. Son implémentation doit gérer l'hétérogénéité architecturale entre les machines communicantes.

En plus des constructions classiques que l'on retrouve dans un langage de programmation, un langage mobile doit avoir les propriétés *indispensables* **[Knab96]** à la réalisation de la mobilité. En plus de ces propriétés indispensables, un langage mobile peut avoir à supporter d'autres propriétés dites *souhaitables* ou *désirables* **[Knab96]**. D'autres propriétés plus ou moins proches de celles étudiées par Knabe sont présentées dans **[Thor97]** et **[Cugo98]**.
Nous allons d'abord discuter les propriétés indispensables et qui sont :

- Support pour la manipulation de code
- Support de l'hétérogénéité des architectures des ordinateurs
- Une performance acceptable

Puis nous allons aussi discuter quelques propriétés souhaitables dans les langages mobiles et qui sont :

- Typage fort
- Gestion automatique de la mémoire
- Sécurité

Par la suite, dans la section suivante, on va essayer de satisfaire ces exigences, indispensables et souhaitables, dans le langage Caml Light afin de montrer que ce langage peut être un bon point de départ pour le support du paradigme de mobilité.

4.4.1 Propriétés indispensables

4.4.1.1 Manipulation de code

Le langage mobile doit offrir un support de base pour la manipulation de code, c'est à dire il doit offrir des primitives ou des fonctions de la librairie avec les quelles le programmeur exprime la transmission ou la migration d'un code. L'environnement exécutif du langage doit être capable d'exécuter les codes mobiles reçus.

4.4.1.2 Hétérogénéité

Les systèmes distribués sont caractérisés par l'hétérogénéité des architectures des machines et des systèmes d'exploitation. Il est relativement facile d'exécuter des codes mobiles sur des plateformes homogènes, mais ils ne seront ni générales ni réalistes pour des applications distribuées. Aussi, il n'est pas pratique de créer une version spécifique du code pour chaque type d'architecture. L'exécution du code mobile doit donc être possible sur différentes plateformes avec différents systèmes d'exploitation et différentes architectures.

4.4.1.3 Performance

L'implémentation d'un langage mobile doit prendre en considération la taille du code mobile, son temps de migration, le temps de sa conversion en une forme pour la migration … tous ces coûts doivent être réduits au minimum. Mais ces exigences dépendent du type d'application et il n'est pas possible de juger la performance juste par la technique employée par l'implémentation. Il faut donc des codes mobiles benchmarks (programmes de tests) pour évaluer réellement les performances d'un langage mobile.

4.4.2 Propriétés souhaitables

4.4.2.1 Typage fort

Rappelons que le typage fort indique que tous les types sont connus à la compilation, afin d'éviter la majorité des erreurs ayant traits aux types.
Les avantages du typage fort ou statique sont déjà bien connus dans les langages conventionnels. Ils prennent encore plus de l'importance avec la programmation du code mobile.

4.4.2.2 Gestion automatique de la mémoire

Avec une gestion automatique de la mémoire, le problème des pointeurs pendants peut être éliminé. Les pertes de mémoire sont aussi réduites.

4.4.2.3 Sécurité

Il est nécessaire d'assurer que le code mobile s'exécutant sur une machine cible n'endommage pas les ressources systèmes.

4.4.3 Quelques langages mobiles

Il ne s'agit pas bien sûr d'une étude comparative de langages mobiles, mais juste de citer quelques tentatives d'extensions de langages existants vers la mobilité. On peut trouver une telle étude, par exemple portant sur trois langages, dans **[Fiel06]**.

A notre connaissance il n'y a pas eu de travaux sur la mobilité dans le langage Caml et qui se rapportent entièrement à l'échange asynchrone de messages, à l'exception d'une tentative très récente de **[Grun07]**, mais qui est sans suite.

Il existe d'autres extensions de langages fonctionnels pour supporter la concurrence, la distribution et la mobilité. ERLANG, JoCaml et Distributed Haskell sont des exemples.

ERLANG est un langage fonctionnel qui permet la concurrence et la distribution d'une manière similaire à l'extension présentée dans cet article. Mais ERLANG reste non typé, en dépit d'une tentative de typage **[Arts98]** et plus récemment **[Sago08]**.

JoCaml **[Lefe99]**, un langage expérimental aussi, est un système basé sur les agents mobiles développé à l'INRIA. JoCaml est une extension du langage Objective-Caml **[Lero07]** par un calcul distribué, le Join-Calculus **[Four96]**.

Un autre travail qui est étroitement lié à notre extension est Distributed Haskell **[Huch99]** avec un style ERLANG. Distributed Haskell étend le langage purement fonctionnel et paresseux Haskell pour supporter la concurrence et la distribution. Ce Distributed Haskell est un langage paresseux et utilise les monades pour l'implémentation de la distribution et de la mobilité.

Pour ce qui est des langages non fonctionnels, plusieurs ont été étendus dans le sens de la mobilité. Java, vue sa popularité avec Internet, étant le langage qui a subi le plus des expériences sur des extensions mobiles, malheureusement aucune d'elles n'a pu séduire la communauté des programmeurs. Un des inconvénients de l'implémentation de Java, est que l'état d'exécution ne peut ni être capturé ni être restauré. Un travail dans ce sens a été réalisé par Bouchenak **[Bouc01]** au niveau de la

machine virtuelle, mais il faudrait convaincre la société **Sun** pour tenir compte de cette modification. Un autre inconvénient, à notre avis, sont les mécanismes relativement de bas niveau qu'utilise le langage Java pour réaliser la concurrence et la distribution. L'INRIA propose une autre extension au dessus de Java, mais cette fois-ci avec d'autres mécanismes pour la concurrence et la distribution et il ne s'agit que de l'envoi asynchrone de messages **[Proa]**.

Le langage Telescript de General Magic **[Whit96]** est un langage spécialement conçu, dès le départ, pour la programmation d'applications mobiles. Il est orienté objet et interprété, donc portable, malgré que sa machine virtuelle est un peu exigeante en mémoire. Telescript possède des constructions incorporées pour manipuler et transmettre du code mobile. La primitive ***go*** permet à un code mobile de se déplacer de manière autonome. ***go*** se charge de tout le transfert du code mobile, et l'instruction qui suit le **go** sera exécutée sur le site hôte. Mais le langage Telescript est d'une utilisation très limitée: il est resté longtemps une propriété privée.

4.5 Caml Light vers la mobilité

Cette extension de Caml Light à la concurrence et à la distribution fournit un bon point de départ pour développer un langage mobile.

Au lieu de concevoir et d'implémenter un langage mobile à partir de zéro, nous avons choisi, pour quelques raisons que nous avons déjà évoqués, le langage Caml Light qui répond même partiellement aux exigences des propriétés indispensables et désirables de la mobilité, et que nous allons maintenant discuter et situer par rapport au langage Caml Light. Donc il s'agit dans cette section, de comparer Caml Light avec les propriétés, indispensables et souhaitables, d'un langage mobile et voir quelles sont celles qu'il supporte et celles qui restent à fournir.

4.5.1 Caml Light et les propriétés indispensables

4.5.1.1 Manipulation de code

La manipulation de code dans le langage même est non disponible dans le langage Caml Light. C'est-à-dire que notre langage mobile doit pouvoir offrir un support de base pour la manipulation de code. Des primitives ou des fonctions de la librairie doivent être disponibles pour exprimer la transmission d'un code mobile. Et bien sûr, l'environnement exécutif du langage doit être capable d'exécuter les codes mobiles reçus.

Nous allons représenter un code mobile par une fonction **[Kirl01]**. Un code mobile peut alors être structuré par plusieurs fonctions avec l'une d'elles désignée comme point d'entrée. Comme indiqué auparavant, les fonctions de Caml Light sont de première classe, c'est-à-dire elles peuvent être passées en arguments à d'autres fonctions et retournées comme résultats d'autres fonctions. Elles peuvent donc aussi être transmissibles (leur code) par la primitive `send_message`. En plus du code et pour atteindre une mobilité forte, il est nécessaire de capturer l'état d'exécution d'un code mobile. Une approche courante et intéressante **[Marz07]**, appliquée à Java, pourra être utilisée, dans le cas de Caml Light, et qui consiste en la transformation d'un code mobile en un objet sérialisable. Cette approche a été conçue pour être totalement transparente au programmeur et indépendante de toute plateforme.

4.5.1.2 Hétérogénéité

Une approche interprétée peut supporter l'exécution dans un environnement de machines hétérogènes en fournissant un interpréteur au niveau de chaque plateforme engagée. Caml Light était conçu dès le départ avec cette notion de portabilité. Son compilateur, à partir de programmes sources écrite en Caml Light, génère un code intermédiaire (bytecode), avec une bonne utilisation mémoire, indépendant de toute plateforme et destiné à une machine virtuelle (machine à pile). Donc il est possible de diffuser l'interpréteur écrit en langage C de cette machine virtuelle pour chaque type de

plateforme disponible dans le réseau. On peut considérer que cette propriété *indispensable* (l'hétérogénéité) est satisfaite avant même de réfléchir à l'idée de rendre Caml light mobile. Certes l'interpréteur de la machine virtuelle devra subir des modifications pour accepter des codes mobiles.

4.5.1.3 Performance

L'approche précédente, c'est à dire le code source de haut niveau est compilé en code source intermédiaire de bas niveau, peut être exploitée pour améliorer les performances et donc générer un code intermédiaire compacte d'une part efficace pour la transmission lors de la migration du code mobile et d'autre part efficace à l'exécution de ce code sur la machine cible. Le langage Caml Light est allé dans ce sens dès ses premières implémentations. On peut dire que cette propriété est assez satisfaite, mais nous l'avons fait savoir précédemment, l'évaluation réelle des performances doit passer par les benchmarks.

4.5.2 Caml Light et les propriétés souhaitables

4.5.2.1 Typage fort

Le langage Caml Light supporte déjà le typage fort. Son avantage est certain, la détection des erreurs est faite dès la compilation. Les programmes sont encore plus sûrs. En plus, il n'est pas nécessaire de déclarer les types, le compilateur infère tous les types, c'est-à-dire il calcule automatiquement le type des objets. Le traitement des exceptions qu'offre Caml Light favorise encore la création de programmes robustes. Les langages typés statiquement sont actuellement largement acceptés comme nécessaires pour une exécution sûre des programmes, surtout dans un contexte mobile, vérifiant à la compilation que les structures de données sont correctement utilisées et apportent de l'information sur l'utilisation des données afin de les représenter d'une manière optimale. Le langage Caml Light améliore la flexibilité du typage statique en ajoutant le polymorphisme, qui permet un contrôle de type statique de composants software

réutilisables qui opèrent généralement sur des structures de données telles que les listes, les ensembles, les arbres …

Dans Caml Light, le typage statique, le polymorphisme paramétrique et l'inférence de types sont combinés afin améliorer la productivité du programmeur et la construction de composants software fiables et réutilisables.

4.5.2.2 Gestion automatique de la mémoire

Caml Light dispose d'un ramasse miettes (*garbage collector*) très efficace. Celui-ci récupère automatiquement la mémoire, donc pas de références pendantes ni de *segmentation default* dues à des manipulations manuelles de mémoire, alors des erreurs en moins.

4.5.2.2 Sécurité

Le langage Caml light, comme décrit dans [**[Lero97]**, est un langage moderne 'sûr', garantissant que les règles d'accès et de types sont toujours respectées. Le code produit par le compilateur Caml light n'est pas directement interprété par la machine virtuelle. Il peut être analysé par un module supplémentaire, le vérificateur de bytecode, pour vérifier l'absence de constructions potentiellement dangereuses.

Pour aller plus loin, Ailleret **[Aill02]** proposa le typage du bytecode afin d'assurer à un utilisateur que le code compilé qu'il possède s'exécutera sans erreur, et ceci sans aucune connaissance du code source du programme ni de la façon dont il a été compilé.

Dans les langages de programmation fortement typés, comme dans Caml light, la sûreté de l'exécution d'un programme est garantie par les propriétés de typage du langage source. Toutes les vérifications étant entièrement statiques, les informations de type sont logiquement effacées à la compilation car inutiles ensuite. En revanche, en présence de code mobile, il est impératif de garantir la correction du programme lors de sa réception, car on ne dispose plus du code source dont la validité assurait la sûreté.

Pour ce faire, l'approche d'Ailleret **[Aill02]** consiste à ajouter au bytecode quelques informations de type, qui, sans rien dévoiler du code source du programme, permettent de vérifier rapidement qu'un code compilé possède les mêmes garanties de sûreté que le langage source.

4.6 Conclusion

Des trois propriétés *indispensables*, deux sont supportées, à savoir l'hétérogénéité et la performance. Pour ce qui est de la manipulation du code, c'est là qu'il faudrait faire une extension. Les trois propriétés désirables citées auparavant sont aussi supportées. On peut dire d'une manière générale que le langage Caml Light était vraiment prédisposé à la mobilité, comme nous l'avons fait savoir dès le début, et peut être un bon point de départ pour le développement d'un langage mobile. Malgré ceci, cette extension proposée de Caml Light concurrent et distribué pour la mobilité ne représente pas un langage complet, un travail assez important reste à réaliser.

Conclusion

La mobilité est un défi. Elle pose des problèmes très épineux, pour disposer d'applications mobiles. Ce qui a poussé récemment même Giovanni Vigna **[Vign04]**, l'un des fondateurs de ce modèle distribué à écrire « Agents Mobiles : Dix Raisons Pour L'échec » ! Parmi ces raisons (la raison n°9) et une qui nous intéresse particulièrement, est que la mobilité ne dispose pas encore d'un langage mobile acceptable pour la diffusion d'applications mobiles sur les réseaux. C'est aussi, pour nous, une déclaration qui nous a poussé à travailler sur ce thème – malgré sa diversité, sa portée et son ampleur – non pas bien sûr pour contredire cette raison n°9 de Giovanni Vigna, mais de donner une petite lueur à la mobilité. Cette thèse s'efforce de montrer que le chemin emprunté, pour la conception d'un nouveau langage mobile, en particulier le choix d'un langage candidat à la mobilité, est très important.

Nous avons décrit la conception d'un langage concurrent et distribué et qui se prête à la mobilité à base d'un langage existant Caml light qui prend de l'ampleur, que ce soit dans le milieu de la recherche ou dans celui de l'industrie. Les primitives proposées pour la concurrence et la distribution sont de haut niveau, à mémoire distribué et d'une sémantique très simple, mais très claires et présentent une structure naturelle au programmeur, surtout au programmeur non trop impliqué dans une programmation relative de bas niveau – sémaphores, verrous …– Nous avons été influencé en particulier par le langage ERLANG et l'échange de message asynchrone. Nous nous sommes aussi efforcés de montrer que le langage Caml light dispose de plusieurs des propriétés indispensables et désirables pour la mobilité.

Le fruit de notre apport a été aussi la publication de cinq papiers [Mera08a], [Mera08b], [Mera09a], [Mera09b] et [Mera09c] qui décrivent l'extension de Caml Light pour la concurrence, la distribution, et une vision pour la mobilité.

Mais la faisabilité et l'efficacité de notre proposition, ne se fera qu'à travers une implémentation réelle et une comparaison avec d'autres langages mobiles. Il reste donc des choses à faire.

Certes nous avons acquis une certaine expérience et une vue assez large de ce thème de mobilité, mais nos efforts ont été aussi éparpillés par la diversité des concepts de concurrence, de distribution et de mobilité.

Dans l'idée de se pencher sur des points très précis pour concentrer le travail et être plus efficace, nous envisageons de poursuivre nos travaux, dans une équipe pluridisciplinaire depuis les langages jusqu'à la mobilité en passant par les paradigmes de concurrence et de distribution.

Et pour conclure, on peut avancer que les codes mobiles vont certainement donner une autre importance aux applications distribuées et nous croyons que le potentiel de cette technologie est en cours de reconnaissance et que ces codes mobiles seront de plus en plus disponibles sur Internet dans les années à venir.

D'autre part aussi, Les langages utilisant les communications asynchrones commencent à prendre aussi de l'ampleur et connaissent aujourd'hui un regain d'intérêt dans le contexte des applications réparties sur Internet. En effet, on s'accorde à penser que les modèles de communication asynchrones sont mieux adaptés que les modèles synchrones (de type client-serveur) pour gérer les interactions entre systèmes faiblement couplés. Même les langages orientés objet ne font pas exception, comme c'est le cas de Proactive **[Proa]** et de Joram (Java Open Reliable Asynchronous Messaging) **[Balt07]** tous les deux à base de Java. Nous pensons que cette utilisation va augmenter avec l'émergence croissante d'applications distribuées.

La contribution de cette thèse s'efforce aussi pour une compréhension de ce thème.

Bibliographie

Les références bibliographiques sont ordonnées par ordre alphabétique des quatre premiers caractères du nom de l'auteur principal.

[Aill02] Sebastien Ailleret, "Typage de bytecode Caml", Congrès JFLA 2002: journées francophones des langages applicatifs: (Anglet, 28-29 janvier 2002).

[Arms07] J. Armstrong, "Programming ERLANG", ISBN-13:978-1-934356-00-5, The Pragmatic Bookshelf, Raleigh North Carolina, Dallas Texas, 2007.

[Arms96] J. Armstrong, R. Virding, C. Wikström and M. Williams, "Concurrent programming in ERLANG", ISBN 0-13-508301-X, London: Prentice Hall Europe, 1996.

[Arts98] T. Arts and J. Armstrong, "A practical type system for ERLANG", Technical Report, ERLANG User Conference, September 1998.

[Balt07] R. Balter, A. Freyssinet "Joram, un intergiciel de communication asynchrone" Licence Creative Commons 8 janvier 2007. http://creativecommons.org/licenses/by-nc-nd/2.0/fr/deed.fr

[Bark99] J. Barklund, R. Virding, "ERLANG 4.7.3 Reference Manual", February 9,1999.

[Bouc01] Sara Bouchenak, "Mobilité et Persistance des Applications dans l'environnement Java", Thèse de Doctorat. Institut National Polytechnique de Grenoble, 2001.

[Carz97] A. Carzaniga, G. Picco, and G. Vigna. "Designing Distributed Applications with Mobile Code Paradigms", In Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston, MA, April 1997.

[Chât03] Pierre Châtel et Carine Pivoteau, "Caml, Historique, Concepts et Etude Comparative", Caml, Rapport de T.E., 2003.

[Conw58] Melvin E. Conway, "Proposal for an UNCOL", *Communications of the ACM* **1**:3:5 (1958).

[Cris] Projet Cristal. Arrêté en 2005. Projet Gallium lui succède. INRIA.

[Cugo98] "Analyzing mobile code Languages", Gianpaolo Cugola, Carlo Ghezzi Gian Pietro Picco and Giovanni Vigna. Springer Berlin / Heidelberg ISSN 0302-9743 (Print) 1611-3349 (Online) Volume 1222/1997, 1997.

[Emme01] W. Emmerich, N. Kaveh. "F2: Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model", In Volker Gruhn, editor, Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01), volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 311–312. ACM Press, 2001.

[Erla03] ERLANG. http://www.erlang.org, page, May 2003.

[Fiel06] Zara Field, P.W. Trinder and André Rauber Du Bois, "A comparative Evaluation of Three Mobile Languages", Proceedings of the 3rd International Conference on Mobile Technology, applications and systems, Bangkok, Thailand 2006.

[Form] Projet FORMEL. INRIA.

[Four96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget and D. Rémy, "A calculus of mobile agents", In Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96), pages 406–421. Springer-Verlag, 1996.

[Fugg98] A. Fuggetta, G. Picco, and G. Vigna. "Understanding Code Mobility", IEEE Transactions on Software Engineering, 24(5):342-361, May 1998.

[Geha92] N.H. Gehan1, W. D. Roome "Implementing Concurrent C ", Software Practice And Experience. Vol. 22(3). 265–285 (March 1992)

[Gilm97] S. Gilmore, "Programming in Standard ML'97: A tutorial introduction", Technical Report ECS-LFCS-97-364, Laboratory for Foundations of Computer Science, 1997.

[Gray95] R.S. Gray. " Agent Tcl: A Transportable Agent System ", In Proceedings of the CIKM'95 Workshop on Intelligent Information Agents, 1995.

[Grun07] B. Grundmann, "eConcurrency: Concurrent and Distributed Programming in OCaml", http://osp.janestreet.com/files/econcurrency.pdf August 2007.

[Huch99] F. Huch, "Erlang-Style Distributed Haskell", In Draft Proceedings of the 11th International Workshop on Implementation of functional Languages, September 7th-10th 1999.

[Ichb83] J. Ichbiah (ed.): "*Ada Programming Language*", ANSI-MIL-STD-1815A, Ada Joint Program Office, Department of Defense, Washington DC, 1983.

[INRIA] **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique http://caml.inria.fr/download.fr.html

[Jone96] S. P. Jones, A. Gordon, S. Finne. " Concurrent Haskell ", In Conference Record of POPL '96 : The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 295-308, St. Petersbourg Beach, Florida, 21-24 January 1996.

[Kirl01] Zeliha Dilsun Kirli, "Mobile Computation with Functions", PhD Thesis, UK, 2001.

[Knab96] Frederick Knabe, " An Overview of Mobile Agent Programming ", P. Universidad Cat´olica de Chile, Casilla 306, Santiago 22, Chile

[Lefe99] F. LeFessant, "JoCaml : Distributed programming and mobile Agents", Première Conférence Française sur les Systèmes d'exploitation. Rennes, 8 juin-11 juin 1999.

[Lero07] Xavier Leroy, "The Objective Caml system release 3.10", **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique, May 16, 2007.

[Lero97] Xavier Leroy, "The Caml Light system release 0.74", **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique, 1997.

[Marz07] S. Marzouk, M. Ben Jemaa, M. Jmaiel, "Serialisation based approach for processes strong mobility", ICTA'07, April 12-14 2007, Hammamet, Tunisia.

[Matt97] D. Matthews. "ML with Concurrency : Design, Analysis, Implementation, and Application", Chapter3, pages 31-58. Springer, 1997.

[Mera08a] Elkamel Merah, Allaoua Chaoui, "Distributed Caml Light", International Review On Computers and Software (IRECOS) ISSN 1828-6003 Vol.3 N.5 IRECOS Journal Italia, September 2008

[Mera08b] Elkamel Merah, Allaoua Chaoui, "TOWARDS A MOBILE CAML LIGHT: mCamllight", Ubiquitous (UBICC) Special issue on New Technologies, Mobility and Security. (NTMS) ISSN Online 1992-8424 ISSN Print 1994-4608 UBICC Journal South Korea October 2008.

[Mera09a] Elkamel Merah, Allaoua Chaoui, "Un Caml Light Distribué ", Conférence Internationale sur l'Informatique et ses Applications Saïda-Algérie, 03-04 Mai 2009.

[Mera09b] Elkamel Merah, Allaoua Chaoui, "Towards Mobility For Caml Light ", First International Conference on 'Networked Digital Technologies' (NDT 2009) Technical co-sponsored by IEEE. VSB- Technical University of Ostrava, Czech Republic. July 28-31, 2009.

[Mera09c] Elkamel Merah, Allaoua Chaoui, "Design of A Concurrent Caml Light : ACCL", To appear in Second International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2009) Technical co-sponsorship by IEEE UK&RI Selection. London Metropolitan University, United Kingdom August 4-6, 2009.

[Nori75] Nori, K.V, Ammann, U., Jensen, K. Nageli, H. (1975). "The Pascal P Compiler Implementation Notes", Zurich: Eidgen. Tech. Hochschule.

[Nori81] Nori, K.V, Ammann, U., Jensen, K., Nageli, H., and Jacobi, C., " Pascal : The language and its implementation ", ch. Pascal-P Implementation Notes, pp.113-214. Wiley, 1981.

[OMG95] Object Management Group, " Corba : Architecture and specification ", August 1995.

[Proa] ProActive. http://www-sop.inria.fr/oasis/ProActive/.

[Raja02] S. K. Rajamani, J. Rehof, "Conformance checking for models of asynchronous message passing software ". Lecture Notes in Computer Science, 2404:166–179, 2002.

[Repp92] J. Reppy, "Higher-order Concurrency", PhD Thesis, Department of Computer Science, Cornell University Ithaca, NY 14853, 1992.

[Sago08] K. Sagonas, D. Luna, "Gradual Typing of Erlang Programs: A Wrangler Experience", in Seventh ACM SIGPLAN Erlang Workshop, Victoria, British Colombia, Canada, September 27, 2008.

[Scho95] E. Scholz, "Four Concurrency Primitives for Haskell", Haskell Workshop, June 25, 1995, La Jolla, California.

[Stam90] J. W. Stamos and D.K. Gifford, "Remote Evaluation", ACM Transactions on Programming Languages and Systems, 12(4) 537-565, October 1990.

[Sun95] Sun Microsystems. "The Java Language Specification", October 1995.

[Tane02] Andrew S. Tanenbaum and Marten van Steen, "Distributed Systems: Principles and Paradigms", Prentice-Hall, 2002.

[Thor97] Tommy Thorn, "Programming languages for mobile code", ACM Computing Surveys, 29(3):213–239, Sept., 1997.

[Vign04] Giovanni Vigna, "Mobile Agents: Ten Reasons For Failure", Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM'04).

[Whit96] J. White, "Telescript Technology: Mobile Agents", In J. Bradshaw, editor, Software Agents. AAAI Press/MIT Press, 1996.

[Yang06] J. Yang, K. H. Rhee, "A New Design for a Practical Secure Cookies System", Journal Of Information Science And Engineering 22, 559-571 (2006).