Un Caml Light Distribué

Elkamel Merah[1,2], Allaoua Chaoui[2]

[1] Centre Universitaire de Khenchela
B.P 1252 Elhouria 40004 Khenchela, Algérie
kmerah@yahoo.fr

[2] Université Mentouri de Constantine, Algérie
a_chaoui2001@yahoo.com

Résumé. Dans cet article nous proposons une extension du langage fonctionnel Caml Light appelé ACCL (pour *A Concurrent Caml Light*) dont le but est de combiner les paradigmes de programmation impérative, fonctionnelle, de concurrence et de distribution dans un seul langage de programmation. Pour l'extension concurrente de Caml Light nous proposons quelques primitives avec une sémantique très simple en utilisant le modèle du langage de programmation ERLANG. Le support de la création dynamique de processus et de la communication asynchrone entre processus sont les deux principales extensions du langage Caml Light. Une adaptation de ces concepts à la distribution est aussi faite. Ce langage proposé sera développé par la suite pour supporter la mobilité.

Mots-clés. Programmation fonctionnelle, Concurrence, Distribution, Communication asynchrone.

1 Introduction

Les systèmes distribués sont ceux qui contiennent plusieurs processeurs reliés par un réseau de communication, qui peut être un réseau local comme Ethernet, ou un réseau très large comme Internet. Ces systèmes sont de plus en plus disponibles en raison de la baisse des prix des processeurs d'ordinateurs.
Voici quelques bonnes raisons d'utilisation des applications distribuées:
• La **tolérance aux pannes** est améliorée en distribuant le calcul entre plusieurs ordinateurs de sorte que, en présence de défaillance d'un ou plusieurs ordinateurs, on peut continuer sur un autre ordinateur, c'est-à-dire le système dans son ensemble peut encore survivre.
• Le **partage des ressources** est permis par l'accès aux ordinateurs qui offrent des ressources spéciales comme les imprimantes. Le **partage des données** est également possible.

• La performance est atteinte lorsqu'une application est parallélisée et exécutée sur des machines différentes.
• Beaucoup d'applications sont par nature, distribuées, et engagent des machines séparées dans l'espace, par exemple, un jeu multiutilisateurs ou un système de *chat*.

Le nombre de systèmes distribués a augmenté à l'échelle mondiale. Le World Wide Web est un exemple d'un service d'information global. Les systèmes distribués sont également utilisés dans le e-commerce, les jeux d'ordinateur, les appareils mobiles ...

Dans ce contexte, nous nous proposons d'étendre le langage Caml Light avec un petit nombre de primitives simples liées à la concurrence [1] et à la distribution, ce langage portera le nom de ACCL [1] (***A C****oncurrent* ***C****aml* ***L****ight*). Réduire le nombre de primitives de concurrence et simplifier leur complexité peut avoir quelques effets bénéfiques dans la vérification et la sémantique [2]. D'autre part, les programmes fonctionnels sont concis, sûrs et élégants [3]. Ces qualités découlent des points forts des langages de programmation fonctionnelle telle que les fonctions d'ordre supérieur, leur système de type robuste et la gestion automatique de la mémoire. Les avantages de la programmation fonctionnelle sont de plus en plus évidents.

Malgré ces atouts, et dans certains cas, un autre paradigme de programmation, comme le paradigme impératif, semble plus apte pour des utilisations directes des ressources machine. Caml Light [4], une version légère du langage Caml, est un langage fonctionnel parmi autres qui possède des traits impératifs. Alors, notre objectif est d'étendre le langage de programmation fonctionnelle Caml Light [4] avec un concept de haut niveau pour la concurrence et la communication entre processus et la distribution. Nous trouvons ceci dans le langage ERLANG (mais sans typage statique) en raison de sa simplicité et de sa grande utilisation dans le développement de systèmes distribués ; ce qui est l'objet de la section 2.

Caml Light est brièvement décrit dans la section 3. Les sections 4 et 5 présentent l'extension de Caml Light, dans le style de la conception de la concurrence et de la distribution du langage ERLANG. Cette approche a également été utilisée avec le langage Haskell [5]. Nous présentons des idées sur la mise en œuvre dans la Section 6, et nous la comparons avec d'autres approches dans la section 7. Nous concluons et nous présentons les futurs travaux dans la section 8.

2 Le Langage ERLANG

ERLANG [6] est un langage de programmation fonctionnelle pour les systèmes concurrents, temps réel et distribués, tolérants aux pannes. Ce langage a été conçu par le laboratoire d'informatique chez Ericsson pour des applications de télécommunications [7]. ERLANG est un langage non typé et possède des types de base et des types de données structurées. Les types de données de base sont les atomes et les nombres. Les types de données structurées sont les tuples et les listes. Les variables sont utilisées pour le stockage de valeurs de n'importe quel type de données et peuvent être affectées une seule fois. Le filtrage de motif est utilisé pour affecter une valeur à une variable.

Les programmes sont entièrement écrits en termes de fonctions: la sélection de fonction est faite par la technique du filtrage de motif qui conduit à des programmes très compacts. Le filtrage apparait aussi dans d'autres instructions du langage. Toutes les fonctions ERLANG appartiennent à quelque module. Le module le plus simple contient une déclaration de module, les déclarations d'exportation et le code correspondant aux fonctions qui sont exportées par le module. Les fonctions exportées peuvent être exécutées à l'extérieur de ce module. Toutes les autres fonctions ne peuvent être exécutées qu'à l'intérieur de ce module.

ERLANG [8] est un langage où la concurrence fait partie du langage de programmation et non pas du système d'exploitation. Le langage possède des mécanismes intégrés pour la programmation concurrente.

3 Un Aperçu du Langage Caml Light

Caml Light (***Ca****tegorical Abstract* ***M****achine* ***L****anguage*) est un langage de programmation de haut niveau à usage général statiquement typé dans la famille ML, qui combine les caractéristiques fonctionnelles et impératives. Caml Light a été publié en France en 1995 par Xavier Leroy [4]. Caml Light est une implémentation légère et portable du langage Caml, avec un compilateur bytecode. Le système Caml Light est un logiciel libre (*open source*). En raison de son état stable, il est très utilisé dans l'éducation et l'expérimentation.

Caml Light adopte le typage statique pour améliorer la sécurité, alors que ERLANG utilise le typage dynamique. Caml Light est un langage strict et, par conséquent, est meilleure, en termes de performance, que les langages paresseux comme Haskell.

Caml Light est un langage sûr qui garantit l'intégrité des données manipulées par un programme. Il dispose d'un système d'inférence de type qui permet d'éviter les déclarations explicites dans la plupart des cas, l'information de type est automatiquement déduit par le compilateur.

Caml Light est un langage de programmation fonctionnelle: les fonctions sont des valeurs de première classe, c'est-à-dire les fonctions peuvent être passées comme arguments à d'autres fonctions ou retournées comme résultats. Il supporte également le style impératif, ce qui inclut les boucles, ainsi que des structures de données mutables, tels que des tableaux. Il dispose aussi du traitement des exceptions pour la gestion des erreurs.

Caml Light possède les types de données de base habituels: entiers, nombres flottants, booléens, caractères et chaînes de caractères. Les structures de données prédéfinies incluent les n-uplets, les tableaux et les listes. Des mécanismes généraux de définition d'autres structures de données sont également fournis.

Le langage peut aussi être utilisé pour le traitement symbolique: manipulation formelle des expressions arithmétiques contenant des variables. Il dispose d'une bonne capacité de calcul symbolique par le biais du filtrage de motif qui est une généralisation de l'analyse par cas traditionnelle. Le filtrage fournit un moyen concis et élégant pour analyser et nommer les données simultanément. Le compilateur Caml Light tire avantage de ce trait pour soumettre le code à plusieurs vérifications :

branches superflues et branches manquantes sont détectées et signalées, ce qui permet souvent d'éliminer des subtiles. Lorsqu'aucune erreur n'est signalée, on peut avoir la certitude qu'aucun cas n'a été oublié. Le filtrage apporte un confort inégalé et un niveau de sécurité dans le traitement des données de nature symbolique.

Caml Light connait le polymorphisme, ce qui signifie que nous pouvons formuler des fonctions qui prennent des valeurs d'entrée ou de produire des résultats de type quelconque.

Un programme dans ce langage est formé par des unités de compilation que le compilateur traite séparément. Son système de modules est puissant et sûr: toutes les interactions entre modules sont vérifiées statiquement lors du contrôle de type.

Enfin Caml Light dispose d'un gestionnaire automatique de mémoire, c'est-à-dire qu'on n'a pas besoin d'allouer ou de libérer de la mémoire, ceci est laissé à la charge du compilateur.

4 Un Caml Light Concurrent : ACCL

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 ya 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. 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 de Caml Light, et que nous n'allons pas utiliser.

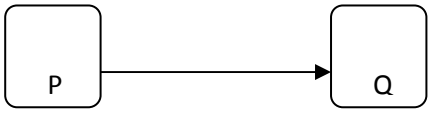
L'autre classe des primitives de concurrence est la mémoire distribuée (ou échange de messages). Les primitives ou opérations de base dans la communication par échange de messages sont "envoyer un message" et "recevoir un message", et sont toutes les deux utilisées pour la communication et la synchronisation. 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 ne limite pas le parallélisme [9]. 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, ACCL [1] sera un exemple de langage de programmation, comme ERLANG, basé sur l'échange asynchrone de messages, dans la classe des langages à mémoire distribuée.

ACCL [1] sera alors une extension concurrente du langage fonctionnel Caml Light, avec deux nouveaux ingrédients ajoutés à Caml Light:

- 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

4.1 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. Un processus, dans ACCL, n'a pas de notation spéciale, nous utilisons la même notation pour les modules. ACCL propose une nouvelle primitive **spawn_process**, 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 simple de ACCL

```
(* 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();;
```

4.2 Communication Interprocessus

Dans ACCL, chaque processus possède une boîte aux lettres, et tous les messages qui sont envoyés au processus sont stockés dans la boîte aux lettres dans l'ordre de leur arrivée. Nous introduisons un nouveau type appelé **messages** [5] comme un type de données algébrique pour définir les messages que le processus peut communiquer avec:

type messages $Cons_1\ t_{11}\ldots t_{1n1}$
$| Cons_2\ t_{21}\ldots t_{2n2}$
...
$| Cons_k\ t_{k1}\ldots t_{knk}$;;

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 aux lettres 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 aux lettres. 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:

4.2.1 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 aux lettres du processus avec identificateur *PId*, et renvoie *Message* comme

valeur. 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.

Afin de recevoir les messages d'un processus spécifique, le langage ACCL 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*)

4.2.2 La Primitive receive_message

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 $pattern_1$ → e$xpression_1$

| $pattern_2$ → e$xpression_2$

...

| $pattern_n$ → $expression_n$

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 instruction **receive_message**, sa boîte aux lettres est examinée et le premier message dans la boîte aux lettres 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 aux lettres, il est enregistré pour traitement ultérieur. Si la correspondance réussit ou échoue, ce premier message est supprimé de la boîte aux lettres 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 aux lettres soient examinés. Si aucun des messages dans la boîte aux lettres ne correspond, le processus qui évalue **receive_message** est suspendu jusqu'à ce qu'un message soit enregistré dans la boîte aux lettres. Tous les messages sauvegardés pour traitement ultérieur, sont de nouveau remis dans le même ordre dans la boîte aux lettres, quand un nouveau message arrive et sa correspondance réussit.

Nous avons choisi la même syntaxe de l'expression *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*} → …

ACCL ajoute aussi 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 *pattern*$_1$ → *expression*$_1$
...
| *pattern*$_n$ → *expression*$_n$
after *time* → *expression*$_t$
end

4.2.3 Un Exemple de Calculatrice de Bureau Concurrente

Pour illustrer l'utilisation des primitives de concurrence proposées, nous proposons une petite application concurrente client-serveur, écrit en ACCL. 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 de ACCL 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 en ACCL 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(p,(self(),requete));
  receive-message
  with
    {p,reponse} → reponse
  end;;
let loop() =
  receive_message
  with
    (de,(somme,a,b)) → send_message(de,(self(),a+b)); loop();
    (de,{moins,a,b}) → send_message(de,(self(),a-b)); loop();
    (de,{produit,a,b}) → send_message(de,(self(),a*b)); loop();
    (de,_) → send_message(de,(self(),(erreur,_)); loop();
  end;;
 (* fichier serveur_cal.mli *)
value lancer: unit → unit;;
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 en ACCL, 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))**;;**

5. Adaptation de ACCL pour la Distribution

Dans cette section, nous allons décrire les primitives proposées pour Caml Light et que nous allons utiliser pour écrire des programmes distribués. Toutes les primitives proposées pour la concurrence auront les mêmes propriétés dans un système distribué que dans une seule machine.

Dans ACCL les programmes distribués s'exécutent sur un réseau de nœuds. Un nœud est un concept central dans le langage ACCL. Un nœud représente une instance d'une machine virtuelle avec son propre espace d'adressage et son propre jeu de processus. Plusieurs nœuds peuvent être situés sur un même ordinateur ou sur plusieurs ordinateurs dans un réseau. Chaque nœud est composé de deux parties, un nom et un hôte, séparées par le caractère '@'. Par conséquent, le PID doit indiquer le nœud sur lequel un processus est localisé. Aussi, ACCL publie un identificateur de processus de sorte que tout processus dans le système peut communiquer avec ce processus, ce processus est appelé un processus enregistré. Cela se fait par la primitive **register_process** (*nom*, *pid*) où le *pid* est enregistré globalement comme *nom* sur le nœud où il est exécuté. La primitive **unregister_process**(nom), supprime cette association.

Les autres primitives utilisées 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 args*) et engendre un nouveau processus sur *nœud*, en appliquant la fonction *func*, localisée dans le module *mod*, avec comme argument *args*.

Envoyer un message à un processus distant est syntaxiquement et sémantiquement identique à l'envoi d'un message à un processus local, et cela peut être fait avec: **rsend_message** (*nœud*, *PID*, *Message*).

5.1 Un serveur de noms en ACCL

Voici un exemple d'un serveur de noms écrit dans le langage ACCL proposé.
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(p,(self(),arg));
  receive_message
  with
    (p,reponse) →reponse
  end;;
let loop() =
  receive_message
  with
    (from,(enregistrer,nom,valeur)) → put(nom,valeur);
                                      send_message(from,(nv_serveur_noms,true));
                                      loop();
    (from,(rechercher,valeur)) → send_message(from,(serveur_noms,get(nom)));
                                 loop();
  end;;
 (* fichier serveur_noms.mli *)
value lancer: unit → unit;;
value enregistrer: string → int -> bool;;
value rechercher: string → int;;
```

Au niveau du shell, on peut faire localement, après chargement du système interactif de ACCL, en cours de développement:

```
# serveur_noms__lancer()                          donne - : true
# 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 - : 0
```

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 de ACCL. Alors, on peut envoyer quelques commandes de noeud1 sur hote1 à noeud2 sur hote2, en utilisant le système interactif :

```
# rpc(noeud2,serveur_noms,enregistrer,("nom3",valeur3))   donne - : true
# rpc(noeud2,serveur_noms,rechercher,("nom3"))            donne - : valeur3
```

rpc(noeud, mod, func, (arg1, arg2, ..., argN)) avec quatre arguments, effectue un appel de procédure à distance sur nœud. La fonction à appeler est mod__func(arg1, arg2, ...,argN).

6. Implémentation

Un prototype d'implémentation du langage ACCL, qui est essentiellement un outil pédagogique pour décrire des caractéristiques de concurrence et de distribution, au dessus de Caml Light est actuellement en cours d'investigation, ce qui n'est pas facile vu que le langage Caml Light est typiquement séquentiel. Cette implémentation servira comme environnement de test pour le développement de programmes ACCL. Des expérimentations seront menées et reportées dans l'avenir.

7. Travaux Connexes

A notre connaissance il n'y a pas eu de travaux sur la concurrence dans le langage Caml qui se rapportent entièrement à l'échange asynchrone de messages, à l'exception d'une tentative de Grundmann [10], mais qui est sans suite.

Il existe d'autres extensions de langages fonctionnels pour supporter la concurrence et la distribution. 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 [11] et plus récemment [12]. JoCaml [13], un langage expérimental aussi, est un système basé sur les agents mobiles développé à l'INRIA. JoCaml est une extension du langage Objective-Caml [14] par un calcul distribué, le Join-Calculus [15]. Un autre travail qui est étroitement lié à notre extension est Distributed Haskell [5] 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.

8. Conclusion et Perspectives

Nous avons décrit la conception d'un langage expérimental et non une proposition pour un langage complet, mais une petite et simple extension du langage Caml Light, que nous avons appelé ACCL, avec le support de la programmation concurrente et distribuée. À 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. Ces mêmes primitives ont été adaptées pour rester valides dans système distribué. Un petit exemple complet est donné, pour montrer l'utilisation des primitives que nous avons ajoutées à Caml Light.

L'utilisation du paradigme de l'échange de messages asynchrone, pour mettre en œuvre la concurrence et la distribution est, à notre avis, plus naturelle que celui des monades utilisées avec Haskell ou celui du Join-Calculus avec le langage JoCaml, ce qui est tout de même un aspect important pour la communauté des programmeurs. Nous pouvons considérer cette proposition du langage ACCL, comme un langage à

échange de messages et, par conséquent, il est très facile de comprendre un programme ACCL.

Une question évidente pour la poursuite des travaux de ACCL est une autre extension pour supporter la mobilité et il est suggéré que le fait d'avoir des fonctions à la base d'un code mobile est particulièrement un bon choix [16] et [17]. Une première investigation sera consacrée à la communication de fonctions dans un système distribué.

9. Références

1. Les memes auteurs, "A Design of A Concurrent Caml Light: ACCL" Accepted, to appear in The Second International Conference on the Applications of the Digital Information and Web Technologies (ICADIWT 2009) August 4-6 2009, London Metropolitan Business School, London Metropolitan University, UK.
2. E. Scholz, "Four Concurrency Primitives for Haskell" Haskell Workshop, June 25, 1995, La Jolla, California.
3. S. Gilmore, "Programming in Standard ML'97: A tutorial introduction", Technical Report ECS-LFCS-97-364, Laboratory for Foundations of Computer Science, 1997.
4. Xavier Leroy, The Caml Light system release 0.74. **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique, 1997.
5. F. Huch, "Erlang-Style Distributed Haskell" In Draft Proceedings of the 11th International Workshop on Implementation of functional Languages, September 7th-10th 1999.
6. ERLANG. http://www.erlang.org, page, May 2003.
7. J. Armstrong, R. Virding, C. Wikström and M. Williams, Concurrent programming in ERLANG. ISBN 0-13-508301-X, London: Prentice Hall Europe, 1996.
8. J. Armstrong, Programming ERLANG. ISBN-13:978-1-934356-00-5, The Pragmatic Bookshelf, Raleigh North Carolina, Dallas Texas, 2007.
9. J. Reppy, Higher-order Concurrency. PhD Thesis, Department of Computer Science, Cornell University Ithaca, NY 14853, 1992.
10. B. Grundmann, "eConcurrency: Concurrent and Distributed Programming in OCaml", http://osp.janestreet.com/files/econcurrency.pdf August 2007.
11. T. Arts and J. Armstrong, "A practical type system for ERLANG", Technical Report, ERLANG User Conference, September 1998.
12. 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.
13. F. LeFessant, "JoCaml : Distributed programming and mobile Agents" Première Conférence Française sur les Systèmes d'exploitation. Rennes, 8 juin-11 juin 1999.
14. 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.
15. 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.
16. Zeliha Dilsun Kirli, Mobile Computation with Functions, PhD Thesis, UK, 2001.
17. 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.