Université de Montréal

Conception d'un langage de programmation pour applications distribuées

par
Guillaume Germain

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Conception d'un langage de programmation pour applications distribuées

présenté par:

Guillaume Germain

a été évalué par un jury composé des personnes suivantes:

Mostapha Aboulhamid
président-rapporteur

Marc Feeley
directeur de recherche

Stefan Monnier
codirecteur

Yann-Gaël Guéhéneuc
membre du jury

Mémoire accepté le 28 juin 2006

RÉSUMÉ

Le présent mémoire décrit un langage de programmation spécialisé pour le développement d'applications distribuées. Le langage, nommé Termite, est basé sur le langage Scheme et utilise un modèle de concurrence par passage de message inspiré du langage Erlang. Le système a pour but de fournir un ensemble minimal mais complet de fonctionnalités nécessaires à l'expérimentation avec la création d'abstractions adaptées à la programmation d'applications distribuées.

Le modèle de connexion ouvert permet la création d'un système distribué non-centralisé. La possibilité de pannes est prise en compte dans le modèle, et des manières de les gérer sont définies. La présence de continuation de première classe dans le langage est exploitée afin de permettre l'expression de concepts de haut-niveau, tel que la migration de processus et la mise à jour dynamique de code. La présence des macros permet la création de nouvelle formes syntaxiques afin d'expérimenter avec la création d'abstractions spécialisée dans l'expression de concepts de programmation distribuée.

Le mémoire décrit le langage Termite, son implantation et son utilisation. La validité du modèle est démontrée à l'aide d'exemples d'applications distribuées.

Mots clés: Language de programmation fonctionnels, Scheme, Erlang, programmation d'applications distribuées.

ABSTRACT

This thesis describes a programming language specialized for distributed applications development. The language, named Termite, is based on the Scheme programming language and uses a message-passing concurrency model inspired by the Erlang programming language. The design goal for the system is to have a minimal but complete set of features necessary to experiment with the creation of abstractions specific to the programmation of distributed applications.

The open connection model allows the creation of a non-centralized distributed system. The possibility of failures is taken into account in the model, and ways to handle failures are defined. The presence of first-class continuations in the language is exploited in order to express high-level concepts such as process migration and dynamic code update. The presence of macros provides the capability of creating new syntactic constructs in order to experiment with the creation of abstractions specialized in the expression of concepts related to distributed programming.

This thesis describes the Termite programming language, its implementation and its utilization. We show that Termite meets its original goal usefully and effectively through examples of distributed applications.

Keywords: Functional programming languages, Scheme, Erlang, distributed applications development.

TABLE DES MATIÈRES

LISTE DES FIGURES

REMERCIEMENTS

Je voudrais premièrement remercier mon directeur Marc Feeley pour l'inspiration, les précieux conseils et le support qu'il m'a fournis. Merci également à mon co-directeur Stefan Monnier pour son soutien et ses excellentes suggestions lors de mes travaux.

Merci à ma famille, qui a toujours été là pour moi. Mon père Yvon et ma mère Lucie n'ont jamais tari d'encouragements lors de l'ensemble de ma vie et de mon cheminement académique. Je leur dois tout. Merci à mon frère François-Xavier et à mes soeurs Geneviève et Adéline pour leur présence et leur amour. J'ai la chance exceptionnelle de les côtoyer et de pouvoir m'enrichir et m'inspirer de la variété des intérêts et réalisations de chacun d'eux.

Mes amis ont une place importante dans ma vie, et ceux-ci m'ont aidé autant à me divertir qu'à avancer dans la réalisation de ma maîtrise. Entre autres, je remercie Pierre-Alexandre Fournier, Jean-François Roy, Olivier Duval et Mathieu Plante pour les discussions intéressantes et pour le plaisir que j'ai à les côtoyer.

Finalement, merci à Danielle, ma conjointe et âme-soeur qui est toujours là pour m'encourager et me motiver. Elle m'appuie dans tous mes projets en s'assurant que ma vie conserve un équilibre sain et humain. La vie est tellement plaisante à ses côtés.

CHAPITRE 1

INTRODUCTION

Nous présentons dans ce mémoire un modèle simple et puissant pour exprimer des programmes distribués qui se base sur la concurrence par passage de message et sur le langage fonctionnel Scheme.

Avec l'omniprésence des réseaux et de l'Internet, les applications distribuées sont de plus en plus en demande. En effet, ce type d'application se retrouve sous plusieurs formes : courriels, pages web, discussions, jeux vidéos, téléphonie, partage de fichiers, partage de périphériques et de services, etc.

Toutes ces applications ont la particularité commune d'être constituées d'un ensemble de processus s'exécutant concurremment sur différents ordinateurs distants et communiquant entre eux afin d'échanger des données et de coordonner leurs activités, ainsi que gérer les possibilités de pannes de communication pouvant survenir entre les ordinateurs.

La pratique courante dans l'industrie est d'utiliser une forme de solution particulière à chaque tâche. Une grande partie du travail doit être refaite à chaque fois : comment représenter les données, synchroniser le calcul, exprimer des conditions d'erreur, etc. Ceci entraîne un surplus de travail lors du développement d'une application, ainsi que la création d'un nombre d'opportunités d'introduire des erreurs de programmation.

Nous croyons qu'il est possible d'abstraire ces problèmes avec un modèle simple de concurrence. Nous choisissons de représenter chaque processus d'un système distribué comme un processus évoluant dans un espace isolé, comme s'il était seul sur son propre processeur. Un processus voulant partager des données avec un autre processus le fera par envoi de message. Ce modèle est appelé *concurrence par passage de messages.*

1.1 La concurrence par passage de messages

Le modèle de concurrence par passage de messages s'applique bien à la programmation distribuée car il modélise la situation réelle de plusieurs processus s'exécutant sur des ordinateurs différents. Ceci permet d'abstraire le lieu d'exécution d'un processus. Du point de vue du programmeur il n'y aura pas de différence de notation pour envoyer un message à un processus s'exécutant présentement sur le même ordinateur ou sur un autre ailleurs dans le réseau.

Ce modèle permet également de faire abstraction des différences matérielles entre les machines exécutant des parties d'un travail distribué. Les composantes n'ont qu'à parler un langage commun, c'est-à-dire d'avoir une représentation uniforme des données. La notion de noeud est utilisée afin de représenter un système faisant partie de l'ensemble des composantes du système distribué. En général, un noeud représente un ordinateur dans un système distribué. Un système distribué va être composé de plusieurs noeuds. La communication entre les noeuds est ce qui

permet de faire le traitement distribué en soit.

De plus, le modèle fournit une forme claire d'encapsulation des erreurs pouvant survenir dans un processus : une erreur est localisée au processus, et elle ne peut pas affecter le bon fonctionnement d'un autre processus par accident (par une corruption de la mémoire, par exemple). Ceci évite une importante classe de problèmes se retrouvant dans d'autre systèmes. Les erreurs peuvent quand même être propagées d'une manière contrôlée définie par le programmeur. Ceci est accompli à l'aide de la création de *liens* entre les processus. Lorsqu'un processus possède un lien vers un autre processus, une erreur entraînant l'arrêt du processus sera communiquée au processus lié. Cela permet d'exprimer diverses relations entre des processus, par exemple lorsque le bon fonctionnement mutuel de deux processus est crucial à leurs exécutions respectives ou lorsque l'on veut qu'un processus puisse être averti d'une erreur causant l'arrêt d'un autre processus. Les fautes matérielles comme une perte de courrant ou la destruction de matériel sont encapsulées par les noeuds. Ceci permet de cerner de manière claire les effets des pannes sur le système.

Le modèle de réseau ouvert est adopté pour le système. Tous les noeuds ont un status égal dans le système. Il n'y a pas de notion de noeud central. On peut donc qualifier le modèle de *pair-à-pair*. Ceci permet une grande flexibilité dans la conception de systèmes ainsi d'une plus grande capacité à résister aux fautes.

1.2 Le langage de programmation Scheme

Nous avons choisi le langage de programmation Scheme [1] afin de construire notre système. Celui-ci est simple, flexible et possède certaines caractéristiques qui font de lui qu'il est particulièrement bien adapté au modèle de programmation distribuée que nous voulons implanter. Ces caractéristiques sont sa syntaxe, ses mécanismes d'entrée/sortie, les fonctions de première classe et la possibilité de manipuler les continuations explicitement.

1.2.1 Les macros

Scheme a une syntaxe extrêmement simple. En effet, le code a la même représentation que les données du langage. Cette représentation du code sous forme de listes facilite grandement la manipulation de celui-ci.

Les macros sont utilisées pour faire ces manipulations de code. Ce sont des fonctions qui s'exécutent avant l'exécution du code (c'est-à-dire au moment de la compilation).

Les macros peuvent être utilisées pour développer de nouvelles formes syntaxiques. Ceci permettra de mieux intégrer au langage Scheme les extensions que nous y feront, donc de créer un nouveau langage adapté aux problèmes auxquels nous nous attaquons.

1.2.2 Les mécanismes d'entrée/sortie

Scheme a comme philosophie d'avoir un mécanisme d'entrée/sortie qui tente de préserver la structure des données. Ceci signifie qu'une donnée écrite puis relue devrait conserver la même forme et le même contenu.

Ce mécanisme est représenté par la procédure de lecture `read` et la procédure d'écriture `write`. La représentation de données sous une forme externe (donc l'écriture) est aussi appelée *sérialisation*. L'opération de lecture inverse s'appellera la *désérialisation*.

Certains types sont simples à sérialiser, comme par exemple un entier, une chaîne de caractères ou une liste. Ils sont simple à sérialiser car ils ont une représentation directe sous forme de texte. D'autres types sont beaucoup plus problématiques, comme les fermetures et les continuations, parce qu'ils n'ont pas de représentation directe sous forme de texte.

Cette forme de sérialisation sera utilisée pour l'échange des données entre les processus faisant parti du système distribué global. Nous désirons avoir le plus de latitude possible sur les valeurs que nous désirons transférer dans un message. Il deviendra alors important de résoudre le problème de sérialisation des objets complexes.

1.2.3 Les fonctions de première classe

Les fonctions et continuations en Scheme sont dites de *première classe* ou *d'ordre supérieur* car il est possible de les manipuler comme n'importe quelle autre valeur dans le langage et de les passer en paramètre à d'autres fonctions. Un cas particulier de fonction en Scheme est nommé un *tronc*, qui est simplement une fonction qui n'attend aucun argument.

La présence des fonctions de première classe dans un langage permet entre autre la définition de fonctions créant de nouvelles fonctions lors de l'exécution et l'envoi de fonctions dans les messages du système. Il est possible d'envoyer une fonction à un autre ordinateur afin de démarrer une tâche à distance sans que celui-ci ne connaisse au préalable la définition de cette fonction.

1.2.4 La manipulation des continuations

La possibilité en Scheme de capturer des continuations et de les manipuler explicitement est l'une des importantes particularités de ce langage. Nous croyons qu'il est possible d'exploiter leur puissance dans le cadre d'un système distribué. Étant donné qu'une continuation capturée représente le futur d'un calcul (un calcul suspendu), celles-ci permettent de réaliser des tâches au niveau du langage comme la migration d'un processus.

CHAPITRE 2

LA PROGRAMMATION D'APPLICATIONS DISTRIBUÉES

Les applications distribuées sont des applications composées de processus s'exécutant simultanément sur plusieurs ordinateurs et communiquant entre eux à l'aide d'un réseau. Cette communication est souvent ce qui fait l'intérêt de l'application. En effet, c'est cette possibilité de transporter et d'échanger des données qui est la raison d'être de ces applications. De plus, l'omniprésence actuelle de l'Internet fait que les applications distribuées sont utilisées dans une multitude de situations en tout temps.

Il est intéressant d'examiner les applications distribuées actuelles afin de déterminer les caractéristiques importantes de celles-ci et de constater en quoi un langage de programmation spécialisé pour la programmation d'applications distribuées peut être utile. Nous examinerons également un exemple d'application simple implanté en premier lieu avec un système populaire puis avec notre langage.

2.1 Exemples d'applications distribuées

Cette section du mémoire examine diverses applications distribuées, dans le but de cerner le travail concret effectué par ces applications. Ceci est intéressant afin de déterminer quelles fonctionnalités sont nécessaires afin de faciliter le développement d'applications distribuées.

Pour chaque type d'application nous expliquons le but de l'application, nous examinons un scénario d'utilisation typique et nous déterminons le genre de technologie de programmation distribuée utilisée.

2.1.1 Courrier électronique

Les systèmes de courrier électronique sont un exemple d'application distribuée : ils impliquent différents acteurs exécutant une transaction au travers du réseau.

Voici un scénario décrivant une interaction typique ayant lieu lors de l'envoi d'un courrier électronique : en premier, un usager utilise son client de courriel pour rédiger et adresser un message. Lors de l'envoi de ce message, le client contacte le serveur de courriel de l'utilisateur, qui prendra en charge la tâche d'assumer le transport du courriel vers son destinataire. Le serveur identifie le serveur distant auquel il doit envoyer le message et lui communique ce message. Puis le serveur distant accepte ce message et le garde en mémoire jusqu'au moment où le client de courriel du destinataire interrogera le serveur distant afin de consulter ses messages.

Différents protocoles entrent en jeu dans cette interaction. Entre le client final et le serveur, les protocoles POP [2] ou IMAP [3] peuvent être utilisés. Ceux-ci diffèrent principalement au niveau des fonctionnalités offertes pour la gestion de la boîte de messages de l'utilisateur. Pour le transport des messages entre deux serveurs de courriel, ainsi qu'entre le premier client et le serveur, le protocole SMTP [4] ("Simple Mail Transport Protocol") est utilisé. Le protocole MIME [5] est uti-

lisé afin de permettre le transport d'informations autres que le simple texte, en spécifiant de quelle manière ces informations peuvent être représentées.

Le modèle d'interaction est celui de clients et de serveurs qui créent de nouvelles connexions sporadiques, dans le but d'échanger des données. La latence de l'interaction n'est pas très importante mais la fiabilité de la transmission l'est. Un ensemble de protocoles est nécessaire pour la définition de contenu, afin que celui-ci puisse être compris par le destinataire d'un message.

2.1.2 IRC

IRC [6] est un acronyme signifiant "Internet Relay Chat". Cette application sert à créer des groupes de discussion en *temps réel*, c'est-à-dire où les gens peuvent communiquer en groupe et s'échanger des messages instantanés.

Plusieurs réseaux IRC existent. Ces réseaux sont composés d'un ensemble de serveurs. Chaque réseau possède un certain nombre de salles de discussion. Dans chaque salle de discussion un groupe peut échanger des messages. Ces messages doivent donc être dirigés par le serveur au client de chaque membre présent dans la salle. Il est également possible d'établir des salles de discussions privées où seul deux utilisateurs peuvent se trouver. De plus, ce médium peut servir par la suite à établir une connexion directe entre les clients, qui n'auront alors plus besoin du serveur pour communiquer entre eux.

L'interaction se fait donc entre plusieurs serveurs inter-connectés, distribuant

l'information à plusieurs clients par serveur. La latence des interactions est un aspect relativement important, sans que celle-ci ne soit vraiment exigeante à satisfaire (un délai de 1 seconde serait acceptable alors que 10 secondes le serait beaucoup moins). Les connexions entre les participants sont continues.

2.1.3 Messagerie instantanée

Les services de messagerie instantanée sont nombreux et populaires. On peut penser à *ICQ*, *MSN Messenger*, *Jabber*, *Skype*, etc. Ceux-ci permettent à un usager d'être disponible à tout moment afin d'échanger des messages avec des connaissances. Le programme client utilisé par l'usager l'informe de la présence en ligne des autres usagers connus (aussi appelés *contacts*).

Le service fait penser en quelque sorte au courrier électronique, mais est conçu de manière à ce que le destinataire reçoive instantanément le message, de façon à créer un échange similaire à un dialogue parlé plutôt qu'à une communication par lettres.

Un usage typique de ce genre de service se déroule comme suit : un usager s'authentifie auprès d'un serveur central, en utilisant le programme client. Ensuite, le serveur central fait parvenir au client la liste de ses *contacts*, soit ses connaissances sur le réseau. Le client peut demander l'état actuel de ses contacts (par exemple *disponible*, *hors-ligne*, *occupé*, etc.). Lorsqu'un usager envoie un message à un autre usager, celui-ci est relayé à sa destination par un serveur central qui

connaît l'adresse IP courante des deux usagers. Il est généralement possible de faire parvenir des données autre que du simple texte à l'aide d'un transfert de fichier effectué directement d'un usager à un autre.

Ce genre d'application nécessite une connexion continue des clients au serveur central et possiblement des connexions sporadiques aux autres clients avec lesquels on veut communiquer. La latence relativement basse et la fiabilité du service sont souhaitables afin d'offrir une interaction intéressante. Le transfert de différents types de données doit pouvoir être effectué.

2.1.4 Jeux multijoueurs

À tout moment, des centaines de milliers de personnes participent à des jeux en ligne sur l'Internet. La nature de ces jeux varie grandement, ainsi que le type d'interconnexion nécessaire entre les participants.

Par exemple, certains jeux d'action nécessitent un échange de données avec très peu de délai entre chaque mise-à-jour de l'état global partagé par les clients. Un délai trop grand nuit au plaisir de l'interaction. Pour ce genre de jeu, chaque client a une connexion à un serveur central, qui s'occupe de gérer la partie.

D'autres jeux, comme certains jeux de stratégie (échecs, go, etc.) sont moins demandants au niveau de l'interaction nécessaire entre les clients. Les messages sont moins nombreux et le délai de transfert d'un message est moins important. Dans ce genre de jeu, un serveur central est généralement utilisé afin d'établir une

communication entre deux joueurs. Le client peut informer le serveur central de la création d'une partie ou demander au serveur central la liste des parties créées qui sont en attente de joueurs pour démarrer la partie. Une fois la partie démarrée, les clients communiquent entre eux pour faire leur travail, sans avoir besoin du serveur central.

2.1.5 Applications collaboratives

Les applications collaboratives sont un type d'application qui permet conceptuellement à plusieurs personnes de partager un espace de travail afin d'accomplir un travail commun. Un exemple d'application collaborative serait un traitement de texte dans lequel plusieurs usagers peuvent écrire en même temps sur le même document depuis plusieurs ordinateurs reliés par le réseau.

Certaines fonctionnalités seront nécessaire afin de garantir la cohérence du résultat obtenu, ainsi que d'aider les participants à différencier le travail effectué par chaque participant. De plus, une communication à un niveau supérieur sera probablement établie entre les participants afin de les aider à coordonner leurs activités, faire part à leurs pairs de leurs commentaires, etc.

Le type d'interaction entre chaque participant va dépendre de l'application exacte qui est faite, mais on peut s'imaginer que la fiabilité de la connexion, une latence raisonnable et la possibilité de transporter toute sorte d'information sont des caractéristiques importantes de telles applications.

2.1.6 Services Web

Les services web sont des programmes généralement ouvertement accessibles qui peuvent être consultés par des programmes quelconques à l'aide de protocoles ouverts. Par exemple, Amazon offre la possibilité de consulter sa banque d'information sur les items de leur catalogue et Google offre la possibilité de faire des recherches directement à partir d'une interface programmatique.

Certains protocoles tels que REST et SOAP sont généralement populaire pour l'utilisation de ce genre de service. Ceux-ci utilisent généralement le protocole de transport HTTP afin de communiquer.

Typiquement, un programme utilisera ce genre de protocole et fournira au serveur qui offre le service un nom d'usager et la requête voulue, et le serveur fournira la réponse correspondante à la requête. Il n'y a pas vraiment de limitation au genre de service qui peut être offert de cette manière, mais l'usage commun est d'offrir un accès programmatique à une banque d'informations.

2.2 Exemple d'application simple

Nous exposons dans cette section un exemple d'application simple, écrit avec la technologie RMI du langage Java, puis reprenons le même exemple avec notre langage Termite. Le but est de donner un aperçu concret du potentiel du langage Termite.

L'exemple que nous prenons ici est celui d'un compte de banque, accessible à

distance, avec une interface simple. Il est possible de déposer de l'argent dans le compte, d'en retirer, et de faire une demande pour connaître le solde du compte. Un registre des accès et opérations effectuées sur le compte est conservé. Pour simplifier l'exemple, nous ignorons les aspects de sécurité d'accès à la ressource.

La conception imposée par le langage Java et RMI est le suivant : le compte lui-même est implanté dans une classe `Account`, qui implante une interface nommée `RemoteAccount`, qui permet de rendre le compte disponible à distance. L'accessibilité à distance est offerte par le serveur `AccountServer`, alors que cet accès est réalisé par le client `AccountClient`. Une classe séparée, `LogRecord`, est utilisée afin de définir la forme des enregistrements dans le registre des opérations effectuées. Le code source de cet exemple se retrouve en annexe, à la section A.1.1. Il comporte 5 fichiers et 127 lignes de code.

Le code équivalent, programmé avec Termite, notre langage dérivé de Scheme, se trouve dans un seul fichier. La technique utilisée afin de représenter un compte de banque est d'utiliser un processus. Le processus conserve l'état courant du compte de banque, soit le solde et l'historique des transactions effectuées sur celui-ci. Une procédure pour démarrer un serveur est définie, avec trois procédures qui servent d'interface à ce serveur. L'interface fonctionne automatiquement tant en accès local qu'en accès distant puisque le compte de banque est implanté à l'aide d'un processus. Voici le code du code de banque, version Termite :

```
(define (make-account)
  (spawn
    (lambda ()
      (let loop ((balance 0)
                 (log '()))
        (recv
          ;; DEPOSIT
          (('deposit amount)
           (loop
            (+ balance amount)
            (cons (list (now) 'deposit amount) log)))

          ;; WITHDRAWAL
          ((from token (list 'withdraw amount))
           (let ((amount (if (> amount balance)
                             amount
                             balance)))
             (! from (list token amount))
             (loop (- balance amount)
                   (cons (list (now) 'withdraw amount)))))

          ;; QUERY
          ((from token 'query)
           (! from (list token balance))
           (loop balance
                 (cons (list (now) 'query balance) log))))))))

(define (deposit account amount)
  (! account (list 'deposit amount)))

(define (withdraw account amount)
  (!? account (list 'withdraw amount)))

(define (query account amount)
  (!? account 'query))
```

Le langage Java a tendance à être verbeux et à nécessiter la création d'un ensemble de classes réparties dans plusieurs fichiers, même lorsqu'une tâche simple est à accomplir. Le langage Termite laisse beaucoup plus de liberté au program-

meur pour la conception de l'application. Il est intéressant de noter que diverses techniques sont disponibles afin de rendre le code Termite encore plus concis grâce à l'utilisation de macros ou de fonction d'ordre supérieur, alors que le code Java ne pourrait pas vraiment être condensé.

On a vu dans ce chapitre que les interactions entre des participants d'une application distribuée consistent généralement en la connexion de systèmes distants dans le but de synchroniser un travail et de transférer des données de tout genre, et cela en ayant une certaine fiabilité dans la réalisation du travail. Termite vise à offrir une solution à ces problèmes en intégrant la notion de connexion distante, d'échange de données et de gestion des erreurs comme des concepts fondamentaux du langage. Un exemple a permis d'illustrer que cette approche est potentiellement plus simple que celle de Java, fréquemment utilisée en pratique.

CHAPITRE 3

LE LANGAGE SCHEME

Le langage Scheme est un langage de programmation fonctionnelle typé dynamiquement avec portée lexicale et fonctions de première classe. C'est un membre de la famille des langages Lisp. Nous présentons dans ce chapitre le langage avec suffisamment de détails pour qu'un lecteur qui ne connaisse pas Scheme puisse avoir un bon aperçu du langage. Nous décrirons la forme de base de la syntaxe du langage ainsi que les particularités qui font de lui un langage particulièrement intéressant pour nos travaux de recherche.

3.1 Les langages Lisp

Les langages Lisp forment une famille informelle à laquelle l'appartenance d'un langage se détermine par certaines caractéristiques générales. La syntaxe parenthésée, le typage dynamique et la présence de procédures anonymes (généralement introduites avec le mot-clé `lambda`) sont probablement les caractéristiques qui permettent d'identifier un langage comme faisant partie de la famille Lisp.

3.1.1 La syntaxe des expressions

La syntaxe des expressions des langages Lisp est composées d'*atomes* et de *listes*. Les atomes peuvent être des symboles (par exemple `foo` ou `+`), des nombres

(par exemple `42` ou `3.1416`) ou des chaînes de caractères (par exemple `"Hello world!"`). Les symboles sont des identificateurs étant généralement des variables et possiblement des mots-clés. Les listes, délimitées par des parenthèses, comprennent des atomes et/ou d'autre listes. Par exemple, le code suivant est composé d'une liste avec le symbole `+`, suivi d'un `2` et d'une sous-liste, elle-même composée du symbole `*` suivi des nombres entiers `3` et `4` :

```
(+ 2 (* 3 4))
```

Les parenthèses autour d'une expression signifient généralement que cette expression est un appel de fonction. La syntaxe générale d'un appel de fonction est la suivante :

```
(fun arg1 arg2 ...)
```

Les parenthèses sont significatives dans les langages Lisp, contrairement à la plupart des autres langages de programmation. Dans l'exemple précédent, `fun` est soit une référence à une variable dont la valeur devrait être une procédure, soit une macro (voir la section 3.1.2), soit une sous-expression, dont la valeur devra éventuellement être une procédure. Les éléments `arg1`, `arg2`, etc. sont les paramètres actuels.

3.1.2 Les macros

Une des particularités importantes des langages Lisp est le fait que le code aie la même syntaxe que la représentation des données : le code est écrit sous forme de

listes imbriquées. Ceci entraîne la possibilité de manipuler le code dans le langage lui-même comme si il n'était qu'une donnée comme une autre. Le mécanisme pour faire ce genre de manipulation s'appelle les *macros*.

Les macros font leur travail au moment de la compilation du code plutôt que lors de l'exécution. Elles permettent donc au programmeur d'étendre le langage et le compilateur afin de créer de nouvelles forme syntaxiques, par exemple une nouvelle structure de contrôle pour exprimer des boucles de type "FOR".

Les macros Lisp, de par le fait qu'elles peuvent manipuler le code d'une façon arbitraire sont beaucoup plus puissantes que le genre de macro que l'on retrouve pour C. Dans le cas de C, les macros ne peuvent faire que de simples substitutions textuelles.

3.2 Scheme

Nous avons jusqu'à maintenant parlé des langages Lisp en général, mais désormais nous nous intéresserons davantage à un des langages membre de cette famille : le langage Scheme. La version actuelle de celui-ci est définie par le standard R5RS [1].

3.2.1 Les fonctions d'ordre supérieur

Scheme a un mot clé particulier pour la définition de fonctions, c'est la forme `lambda`. Celle-ci permet la création de fonctions anonymes qui sont d'ordre supérieur

en Scheme et peuvent ainsi être passées en paramètre à d'autres fonctions. Voici un exemple de création de fonction anonyme et son utilisation dans le cas d'un appel à une fonction d'ordre supérieur :

```
(map
  (lambda (x) (* x 2))
  (list 1 2 3))
                                    ⟹ (2 4 6)
```

La fonction anonyme (`(lambda (x) (* x 2))`) est un fonction qui attend un paramètre et en retourne le double de la valeur. La procédure `map` attend deux paramètres, soit une procédure et une liste. La valeur retournée est une nouvelle liste contenant le résultat de l'application de la fonction sur chaque élément de la liste passée en paramètre.

3.2.2 Définition de variables

La définition de nouvelles variables globales se fait avec la forme spéciale `define` :

```
(define pi 3.1416)
(define double (lambda (x) (* 2 x)))
```

Les variables ainsi définies sont visibles globalement, donc dans l'ensemble du programme.

3.2.3 Portée lexicale

La définition de variables locales se fait avec la forme spéciale `let` :

```
(let ((x 21))
  (double x))                    ⟹ 42
```

La résolution des variables se fait en Scheme en utilisant la portée lexicale. Cela signifie que pour trouver la valeur d'une variable dans une expression il faudra trouver le contexte textuel englobant le plus directement l'expression afin d'identifier la source de la définition de la variable.

Les fonctions anonymes créées avec `lambda` peuvent contenir des références à des variables définies dans le contexte de l'expression. Une fonction qui capture son environnement est appelée une *fermeture*. Les variables capturées de l'environnement englobant sont nommées *variables libres*.

Voici un exemple d'une fonction qui utilise la capture de variable afin de construire de nouvelles fonctions qui peuvent faire un travail particulier :

```
(define make-multiplier
  (lambda (n)
    (lambda (x)
      (* n x))))
```

Dans cet exemple on a une fonction, `make-multiplier`, qui attend un argument `n`. La valeur de retour d'un appel à la fonction `make-multiplier` est une nouvelle fonction qui attend un argument `x`, qui sera multiplié avec le `n` qui avait été donné en paramètre lors de la création de cette fonction. Exemple :

```
(define f (make-multiplier 2))
(f 21)                           ⟹ 42
```

3.2.4 Mutations

Il est possible de changer la valeur d'une variable avec la forme spéciale de mutation nommée `set!`. Cet opérateur permet d'assigner une nouvelle valeur à une variable :

```
(define x 42)
(= x 42)                    ⟹ #t
(set! x 21)
(= x 42)                    ⟹ #f
(* x 2)                     ⟹ 42
```

`set!` peut également servir afin de modifier la valeur d'une variable capturée dans une fermeture :

```
(define make-adder
  (let ((sum 0))
    (lambda (n)
      (set! sum (+ sum n))
      sum)))

(define adder (make-adder))
adder                       ⟹ procedure
(adder 10)                  ⟹ 10
(adder 5)                   ⟹ 15

(define adder2 (make-adder))
(adder2 42)                 ⟹ 42
(adder 0)                   ⟹ 15
```

Comme on peut le voir, la création d'une fermeture permet de conserver un état interne dans cette fermeture. On peut ensuite envoyer un *message* à cette fermeture afin de lui demander de changer son état interne.

3.2.5 Effets de bord

Les mutations brisent le modèle d'évaluation purement fonctionnelle : ce sont des effets de bord. Ceci signifie qu'un effet durable est induit dans le système qui affecte l'état global du système.

Un autre exemple d'effet de bord dans le langage Scheme sont les entrées/sorties. L'affichage d'une valeur est une opération généralement réalisée pour son effet (l'affichage) plutôt que pour la valeur de l'appel à la fonction d'affichage (qui sera généralement indéterminée, puisque sans importance).

Tel qu'indiqué précédemment, c'est la présence des effets de bord comme les mutations qui font que Scheme n'est pas un langage purement fonctionnel. Celui-ci encourage tout de même le style fonctionnel.

3.2.6 Types

Scheme est un langage fortement et dynamiquement typé. Le fait que le langage soit fortement typé signifie qu'il n'est pas possible de "tromper" le système et de lui faire interpréter une donnée comme étant d'un type qu'elle n'est pas. Ceci contraste avec le typage faible de C, par exemple, où il est possible de faire réinterpréter le type d'une valeur à l'aide d'un transtypage ou en dépassant les bornes d'un tableau. Le typage fort est présent dans d'autres langages comme Java et Erlang.

Le typage dynamique du langage signifie que les types sont attachés aux valeurs plutôt que d'être attachés aux variables (comme c'est le cas dans les langages typés

statiquement). Donc au lieu de décider et de vérifier au moment de la compilation si les types des variables sont cohérents (grâce aux déclarations ou à l'inférence de type), les vérifications sont faites durant l'exécution du programme. Au moment de manipuler une valeur une vérification est faite afin de déterminer si le type de cette donnée supporte réellement l'opération que nous désirons réaliser sur elle. Une incohérence générera une erreur, comme par exemple l'addition d'une chaîne de caractères et d'un entier.

3.2.7 Optimisation des appels en position terminale

Une caractéristique importante du langage Scheme est l'obligation pour l'implantation du langage d'optimiser un appel de fonction en position terminale.

Un appel de fonction se trouve en position terminale si le contexte de cet appel n'attend pas la valeur de retour de l'appel afin de faire un travail.

Dans les implantation des langages de programmation en général les appels de fonctions sont effectués en sauvegardant le contexte courant du système sur une pile afin de pouvoir poursuivre le traitement lors du retour de l'appel de fonction. Par contre, si aucun travail ne sera à faire lors du retour de l'appel autre que de faire suivre la valeur de retour de cet appel au prochain bloc d'activation sur la pile, il est inutile de sauvegarder le contexte local. Cela même à une économie d'espace et permet d'exprimer des traitements itératifs (boucles) avec la récursivité sans consommer d'espace sur la pile.

Voici un exemple d'utilisation de la récursion pour calculer la valeur de la fonction factorielle appliquée sur un nombre entier :

```
(define (fact n)
  (if (< n 2)
      n
      (* n (fact (- n 1)))))
```

Cette récursion demandera l'allocation d'un bloc d'activation sur la pile pour chaque appel récursif, puisqu'il est nécessaire de conserver le contexte d'un sous-appel à `fact` afin de multiplier la valeur de retour par `n`. Voici un exemple d'utilisation de la présence obligatoire de l'optimisation d'appels en position terminale de Scheme qui permet d'exprimer le calcul précédent, mais en consommant une quantité constante d'espace sur la pile :

```
(define (fact n)
  (define (helper n product)
    (if (< n 2)
        product
        (helper (- n 1)
                (* n product))))
  (helper n 1))
```

On remarque l'utilisation d'une procédure interne, qui a deux paramètres : celui de l'argument `n`, ainsi que `product`, qui sert d'accumulateur pour le produit calculé jusqu'à présent. Comme un appel récursif à `helper` est en position terminale (aucun travail n'est à effectuer au retour de l'appel de `helper`), celui-ci n'entraînera pas l'allocation d'un nouveau bloc d'activation sur la pile. Ceci permet d'exprimer des

processus itératifs de manière efficace à l'aide de procédures récursives. La lecture de [7] est recommandée sur ce sujet.

3.2.8 Continuations

Un concept présent dans tous les langages mais rarement exploité directement est le concept de *continuation*.

La continuation d'une expression est le calcul futur qui reste à réaliser avec la valeur de retour de cette expression. Par exemple, si une expression fait la somme du nombre 2 avec le produit de 3 et 4 :

```
(+ 2 (* 3 4))
```

La continuation du calcul de l'expression `(* 3 4)` est d'additionner la valeur de ce calcul au nombre 2. Cette continuation peut être vue comme étant en quelque sorte l'adresse de retour à laquelle il faut fournir la valeur retournée par l'expression évaluée.

Les continuations sont en général implicites dans les langages de programmation. Par contre, Scheme possède une fonction spéciale nommée `call/cc` qui permet de capturer la continuation d'un calcul et de la réifier sous forme de fonction. Un appel à `call/cc` attend comme argument une fonction avec un seul argument formel. Cette fonction sera appelée avec un paramètre qui sera la continuation de l'appel à `call/cc` réifié sous forme de fonction. Voici un exemple qui montre la capture d'une continuation qui ajoute 2 à la valeur retournée à cette continuation

(la continuation est assignée à la variable globale `kont`) :

```
(define kont #f)

(+ 2 (call/cc
      (lambda (k)
        (set! kont k)
        (* 3 4))))
                                    ⟹ 14

(kont 40)
                                    ⟹ 42
```

Il est également possible de changer le flot normal d'exécution d'une procédure en invoquant une continuation afin d'échapper à l'évaluation courante. L'exemple suivant consiste en la définition d'une procédure `map*`, qui est une variante de `map`. Normalement, `map` applique une fonction sur chaque élément d'une liste et retourne une nouvelle liste. La variante `map*` retourne la valeur `#f` dès que l'application de la procédure passée en paramètre retourne `#f` lors de son application sur un des éléments de la liste originale, ceci étant fait en capturant une *continuation d'échappement* que nous nommons `return` :

```
(define (even+1 x)
  (and (even? x) (+ x 1)))

(map even+1 (list 2 3 4))
                                    ⟹ (3 #f 5)

(define (map* f lst)
  (call/cc
   (lambda (return)
     (map (lambda (x)
            (or (f x) (return #f)))
```

```
          lst))))

(map* even+1 (list 2 3 4))
                                    ⟹ #f

(map* even+1 (list 2 4 6))
                                    ⟹ (3 5 7)
```

Il est intéressant de remarquer que la procédure `map*` peut utiliser la procédure standard `map` sans même que cette dernière n'ait à appliquer la fonction sur chaque élément de la liste, même si cette dernière est programmée pour le faire. `call/cc` est donc un opérateur de contrôle du flot d'évaluation extrêmement puissant, mais qu'il faut utiliser avec soin.

3.2.9 Entrées/Sorties

Les entrées et sorties en Scheme sont faites généralement avec deux fonctions : `read` (lecture) et `write` (écriture).

La fonction de lecture `read` est en fait un parseur qui comprend la syntaxe des expressions Scheme et qui est généralement capable de transformer la représentation textuelle des objets Scheme en leur représentation interne. La fonction d'écriture `write` *sérialise* les objets Scheme afin de leur donner une représentation textuelle (qui sera idéalement possible de traiter avec `read`).

Voici un exemple qui illustre cette possibilité d'utiliser la représentation externe d'une valeur afin de sérialiser (avec `write`) et de désérialiser (avec `read`) des données en les écrivant dans un ficher `dump.data`, puis en les récupérant et en testant la

structure récupérée :

```
(define (dump data)
  (call-with-output-file "dump.data"
    (lambda (port)
      (write data port))))

(define (restore)
  (call-with-input-file "dump.data"
    (lambda (port)
      (read port))))

(dump (list 1 'foo "hello"))

(let ((data (restore)))
  (and (list? data)
       (number? (first data))
       (symbol? (second data))
       (string? (third data))))
                                  ⟹ #t
```

Notons à ce point la présence de différentes procédures dans Scheme pour tester l'égalité entre des objets. Les deux procédures les plus intéressantes dans le cadre de cette discussion sont `eq?` et `equal?`. La première fait un test d'égalité de la valeur immédiate de l'objet, alors que la deuxième fait un test d'égalité structurelle. Cela signifie qu'une valeur sérialisée puis désérialisée ne sera pas nécessaire égale en terme de `eq?` à la valeur originale, mais devrait être égale en terme de `equal?`. Par exemple, une structure allouée en mémoire comme une liste ne restera pas égale au sens de `eq?` :

```
(define x (list 1 2 3))
(eq? x (begin (dump x)
              (restore)))        ⟹ #f
```

```
(equal? x (begin (dump x)
                 (restore)))     ⟹ #t
```

3.2.10 Évaluation de code à l'exécution

Scheme permet de faire la construction de code et l'évaluation de celui-ci au moment même de l'exécution. Cela signifie qu'il est possible pour un programme Scheme de construire ou de recevoir des données et de les interpréter comme des fragments de programmes. Cette évaluation se fait avec la fonction `eval`. Par exemple :

```
'(+ 1 2)                         ⟹ (+ 1 2)
(eval '(+ 1 2))                  ⟹ 3
```

Le fait que Scheme possède cette fonction `eval` signifie que l'interprète du langage est inclus dans celui-ci.

Cet aperçu du langage Scheme devrait permettre de comprendre les notions utilisées dans ce mémoire ainsi que les exemples de code. Le langage Scheme est concis et flexible et, ainsi, bien adapté à la recherche sur les langages de programmation. Pour un apprentissage plus poussé du langage, le lecteur est référé au document standard du langage [1] ainsi qu'aux livres suivants : [8] et [9].

CHAPITRE 4

TERMITE : UNE ADAPTATION DE SCHEME À LA PROGRAMMATION DISTRIBUÉE

Dans ce chapitre nous décrivons le langage Termite. Nous exposerons d'abord la philosophie, les influences et les objectifs du langage. Ensuite nous expliquerons comment celui-ci se dérive de Scheme.

4.1 Présentation du langage Termite

Termite est un langage de programmation qui a pour but de faciliter le développement d'applications distribuées et l'expérimentation de formes de contrôle et de synchronisation dans ces applications. La conception a été guidée par le principe qu'il devrait être possible d'identifier un nombre minimal de concepts les plus généraux possible afin de pouvoir faire de la programmation distribuée.

Termite a pour principales influences deux langages de programmation : Erlang et Scheme. Dans cette section nous décrirons les fonctionnalités de Termite, mais en partant de la source de chacune.

4.1.1 Influence de Erlang

Du langage Erlang [10], Termite retire le concept de concurrence par passage de messages, d'isolation d'erreurs, d'absence de mutations et de distribution. Ces

concepts, adoptés dans Termite et inspirés de Erlang, sont expliqués ici.

4.1.1.1 Concurrence par passage de message

La concurrence par passage de message est un type de concurrence où chaque processus évolue dans son espace mémoire propre et où chaque échange de donné entre plusieurs processus doit être fait par un envoi de message.

Un modèle similaire de concurrence est utilisé au niveau des processus d'un système d'exploitation comme Linux. Chaque processus opère dans son espace mémoire et est incapable d'accéder directement au contenu mémoire d'un autre processus. Les processus peuvent tout de même communiquer entre eux à l'aide de canaux de communication.

Ce modèle de concurrence dans le cas de Termite et de Erlang est conçu comme suit : chaque processus a une *boîte aux lettres* où les messages qui lui sont envoyés sont emmagasinés. Un processus peut envoyer des messages aux processus dont il a l'adresse. L'envoi de message se fait de manière *asynchrone*. Le processus qui fait l'envoi de message ne bloque pas lors de cet envoi. La consultation de message dans la boîte aux lettres se fait de manière *synchrone*. Le processus sera suspendu jusqu'à la réception d'un message si la boîte aux lettres est vide.

De plus, la réception de message peut se faire de manière sélective. Bien qu'ils soient emmagasinés dans une file d'attente, il n'est pas obligatoire de traiter le message le plus vieux de la file en premier. Il est possible de désigner un prédicat

ou d'assigner un filtre qui choisira quels messages on veut traiter. Ceci permet, par exemple, d'ignorer certains types de messages à un moment pour se concentrer sur un autre type.

La livraison de messages est une opération *incertaine*. Il est possible qu'un message ne se rende pas à sa destination, car des erreurs peuvent survenir dans un environnement distribué : une machine peut tomber en panne, un lien réseau peut être coupé, etc. Ces possibilités de panne sont donc reflétées dans le langage par la non-fiabilité de la livraison de message. Afin de pouvoir gérer les pannes, il est possible de spécifier un délai maximal aux opérations de récupération de message. Cela permet de réagir aux erreurs et d'adapter le traitement en présence de pannes et d'erreurs.

4.1.1.2 Isolation des erreurs

Un concept important adopté dans Termite provenant du langage Erlang est l'isolation des erreurs. Cela signifie qu'une erreur qui survient dans un processus ne devrait jamais affecter un processus qui ne lui est pas relié.

Lorsque l'on fait de la programmation avec le modèle de concurrence avec mémoire partagée, il peut arriver qu'une erreur de traitement se produisant dans un processus entraîne la corruption d'un espace mémoire appartenant à un autre processus. Afin de pouvoir créer un système concurrent fiable, ceci devrait être impossible.

L'isolation complète des processus dans le modèle par passage de message permet de n'avoir qu'à considérer le processus lui-même afin de déterminer quelles erreurs peuvent survenir et comment les traiter.

Parfois, on veut quand même que les erreurs survenant dans un processus puissent être signalées et traitées par un autre processus. On utilise alors le concept de *liaison* des processus. Dans Erlang, lorsque deux processus sont liés une erreur survenant dans un des deux processus et n'étant pas traitée se propage à l'autre processus.

Ceci permet par exemple de construire des hiérarchies de processus, où des processus *superviseurs* sont responsables du bon fonctionnement d'un ensemble de processus *travailleurs* qui effectuent une tâche. Si une erreur irrécupérable survient dans un travailleur, le superviseur se charge, par exemple, de démarrer un nouveau travailleur. Ce mode de construction d'application permet d'atteindre un grand niveau de robustesse. Cette technique est un point central de la thèse de Joe Armstrong [11].

Termite hérite de Erlang ce mécanisme de gestion d'erreur avec une légère nuance. Alors que les liens de propagation d'erreurs dans Erlang sont toujours bidirectionnels, ils sont unidirectionnels dans Termite. Il est possible d'accomplir la même tâche qu'un lien bidirectionnel en utilisant deux liens unidirectionnels. Il semble que la possibilité de contrôler le flot de la propagation des erreurs entraîne une plus grande généralité du design. Un exemple utile de propagation dans un seul

sens des erreurs serait le cas où un observateur est associé à un processus, dans le but de réagir si le processus plante et signale une erreur. Dans une autre situation, il serait aussi possible que le code de l'observateur contienne une erreur qui le fasse planter : à ce moment on ne veut pas nécessairement que le processus plante lui aussi : cela dépend de la logique de l'application.

4.1.1.3 Absence de mutations

Une particularité de Erlang qui le fait ressembler à certains langages purement fonctionnels est l'absence de mutation au niveau du langage. Une mutation est un changement d'état d'une variable, ou le changement du contenu d'une structure contenue en mémoire.

Le fait qu'il ne soit pas possible de faire de mutation permet d'échanger une donnée entre deux processus en ne passant qu'un simple pointeur vers l'endroit de la mémoire où est contenu une structure.

Il est important de noter que même si les mutations ne sont pas disponibles dans le langage en tant que tel, il est quand même possible de créer l'équivalent de structures de données mutables. Celles-ci doivent être créées à l'aide de processus. L'état du processus représente l'état de la structure de donnée mutable.

Les opérations de démarrage de processus et d'envoi et de réception de messages sont toutes des opérations ayant un effet sur l'état global du système. Elles sont donc équivalentes en terme de puissance expressive aux mutation dans d'autre

langages, comme en Scheme par exemple. Cela signifie que le même genre d'objet mutable qui est utilisé en Scheme peut être utilisé dans Erlang, mais en utilisant une technique différente : il faut utiliser les processus pour représenter un objet avec un état particulier.

4.1.1.4 La programmation distribuée

Un langage supporte la programmation distribuée s'il permet d'exprimer les opérations nécessaires à ce genre de programmation. Parmi les opérations essentielles, on retrouve la possibilité de créer des processus, de les faire communiquer ensemble, de gérer une communication réseau, etc. Plusieurs approches sont possibles, selon ce qu'offre la plate-forme ou le langage.

Kali est un système Scheme offrant la possibilité de créer un système distribué dans lequel les objets sont mobiles, y compris les fermetures et les continuations. Malheureusement, le système dépend d'un contrôleur central et ne tolère pas les fautes matérielles et les pertes de connectivité réseau. Le modèle de mémoire partagée distribuée est la cause de ces problèmes.

Erlang est un langage spécialisé pour ce genre de traitement. Son modèle de concurrence par passage de message s'applique bien à des processus qui ne s'exécutent pas sur la même machine.

En effet, le modèle de processus isolé dans son espace mémoire et devant envoyer des messages afin de communiquer est représentatif de la situation du monde réel :

les ordinateurs reliés par le réseau ne partagent pas leur mémoire physique, le transfert d'information doit donc se faire par un envoi du message. De plus, l'envoi de message est incertain, ce qui modélise la possibilité de panne dans un réseau. Ces aspects sont exploités pour refléter et modéliser une situation réelle et sont exposés afin de faciliter le développement d'application ayant à évoluer dans ce genre d'environnement.

4.1.2 Influence de Scheme

Scheme [1] est un langage de programmation avec une spécification concise et cohérente. La syntaxe de ce langage, la présence des macros et l'absence de mots-clés réservés font de lui un langage particulièrement flexible et adapté à l'expérimentation.

4.1.2.1 Les continuations

Les continuations en Scheme sont de *première classe*. Il est possible de capturer la continuation courante à un point quelconque du programme et de réifier cette continuation sous forme de fonction. Cette possibilité de manipuler les continuations dans Scheme permet de manipuler arbitrairement le flot de contrôle d'un programme. Ceci fourni un outil puissant, qui nous permet dans un environnement distribué d'exprimer des concepts comme la migration de processus, le clonage d'un processus, la mise à jour dynamique du code d'un processus, etc. Des exemples

d'applications manipulant des continuations sont donnés à la section 6.1.1.

4.1.2.2 Les macros

Les macros en Scheme permettent de développer de nouvelles formes syntaxiques. Par exemple, Erlang supporte le filtrage, mais ce n'est pas quelque chose qui est offert dans le standard Scheme. Implanter le filtrage d'une manière efficace nécessite de contrôler l'ordre d'évaluation des expressions. Pour que ce soit pratique pour l'utilisateur, la syntaxe doit pouvoir permettre d'exprimer d'une manière claire les patrons de filtrage. Ces caractéristiques demandent l'utilisation de macros.

Les macros permettent donc de créer des *langages spécifiques au domaine.* Ceci est très important étant donné que nous souhaitons que le langage que nous développons soit flexible et qu'il permette bien d'expérimenter avec de nouvelles formes d'expression de programmes distribués. Un bon exemple de création de langage spécifique au domaine est donné dans [12].

4.1.3 Influences communes aux deux langages

Le langage Scheme et le langage Erlang partagent des particularités communes. Ces particularités en font des langages "compatibles", c'est-à-dire où les concepts sont propices à être mariés. Ces caractéristiques communes sont le typage dynamique, la portée lexicale et les fonctions de première classe.

4.1.3.1 Typage dynamique fort

Le langage Scheme et le langage Erlang sont *dynamiquement typés* car les types sont associés aux valeurs plutôt que d'être associés aux variables. Il n'y a donc pas de vérification statique de type. Chaque variable peut contenir une valeur d'un type quelconque, et les vérifications sur la cohérence des types lors d'opérations sur ces données est faite durant l'exécution du programme.

4.1.3.2 Portée lexicale

Comme pour la plupart des langages, Scheme et Erlang supportent la portée lexicale. Cela signifie que pour trouver la valeur d'une variable, l'environnement lexical est consulté. Il est donc possible de trouver l'expression définissant la valeur de la variable en consultant le texte.

4.1.3.3 Fonctions de première classe

Dans Erlang comme dans Scheme, les fonctions sont de première classe. La forme `fun` en Erlang et `lambda` en Scheme permettent de créer de nouvelles fonctions. Ces nouvelles fonction seront *fermées* sur leur environnement lexical : elles possèdent des références vers les variables de leur environnement.

La possibilité d'avoir des fonctions qui peuvent construire de nouvelles fonctions et passer celles-ci en argument et dans les messages envoyés entre les processus est extrêmement important. Ceci permet la construction d'une grande variété d'abs-

tractions.

4.1.3.4 Optimisation d'appel en position terminale

Les deux langages ont une caractéristique particulière qui est l'obligation d'implanter un appel en position terminale d'une manière à ce que l'utilisation de la pile d'appels soit constante. Cela signifie qu'il est possible d'implanter une boucle à l'aide d'un appel récursif sans que l'espace mémoire de la pile soit consommé. Donc, il est possible d'exprimer l'itération avec une forme récursive, ce qui est important dans un contexte où les mutations ne sont pas permise.

4.2 De Scheme à Termite

Au cours des sections précédente dans ce chapitre nous avons vu les influences majeures du langage en construction et déterminé les caractéristiques importantes pour notre système. Nous verrons dans cette section comment partir d'un système Scheme et l'augmenter de manière à obtenir un système de programmation distribuée basée sur Scheme et Erlang.

4.2.1 Les modifications apportées à Scheme

Certaines modifications et spécialisations doivent être faites à Scheme afin de lui permettre de s'adapter aux besoins de la programmation d'applications distribuées.

4.2.1.1 Les mutations

Termite est par choix un langage sans mutations explicites. Il n'est pas possible de changer le contenu mémoire d'une valeur : cela permet donc de partager ce contenu mémoire entre plusieurs processus sans avoir à s'inquiéter des accès concurrents. Cela fait qu'il est plus facile d'obtenir un comportement cohérent lors d'accès à ces endroits mémoire. Ceci se fait au coût d'avoir à allouer l'espace mémoire pour un processus supplémentaire ainsi que d'avoir à gérer la durée de vie de celui-ci.

Le langage Scheme possède 5 fonctions qui posent un problème avec le modèle que nous désirons car elles effectuent la mutation explicite d'une structure en mémoire. Ces fonctions sont : `set-car!`, `set-cdr!`, `vector-set!` et `string-set!`, qui sont des fonctions de mutation, ainsi que `eq?`, qui est une fonction qui fait un test d'égalité en faisant essentiellement une comparaison de pointeurs. Ces fonctions de mutation peuvent tout de même être compatibles avec le modèle de Termite, mais en autant que l'on change l'implantation des structures de données sur lesquelles elles opèrent : elles doivent être implantées à l'aide de processus : l'état du processus suspendu en attente de message est donc utilisé pour représenter l'état d'une structure de donnée. Un exemple est donné à la section 6.1.3.3. Ceci a également pour effet de permettre d'une manière cohérente l'échange de structures de données mutables au travers du réseau. Dans le cas de la fonction `eq?`, le problème est que l'égalité de pointeur n'est pas nécessairement conservée entre la sérialisation et la désérialisation d'un objet. Il faudra alors remplacer `eq?` par

`equal?`, qui fait un test d'égalité structurelle.

La forme spéciale `set!` pose un problème intéressant. À première vue, celle-ci n'est pas exprimable directement dans Termite. Le fait de faire un `set!` sur une variable change sa valeur. Il faut alors penser qu'une variable peut être une partie intégrante d'une structure de donnée qu'est une fermeture. On voudrait donc que la mutation d'une variable dans une fermeture soit visible par un autre processus possédant une copie de cette fermeture. Puisque qu'une affectation faite à une variable doit également être visible dans une autre fermeture, on doit partager l'endroit où est stocké le contenu de la variable. En Scheme, la technique classique (telle qu'expliquée dans [13]) est de créer une cellule contenant la valeur de la variable pour chaque variable qui peut subir une mutation avec `set!` et d'avoir une référence à cette variable dans chaque fermeture. Il serait possible dans Termite de remplacer cette cellule par une implantation basée sur un processus. Un bénéfice de cette approche est que le concept fonctionne encore lorsqu'une variable est référée dans une fermeture sur un noeud distant. Cette modification à Scheme est moins superficielle que celles faites pour les procédures de mutation, mais serait réalisable en changeant le code source de l'implantation Scheme.

4.2.1.2 Les ports

Les ports tels que décrits dans le standard Scheme ne peuvent pas être utilisés d'une manière directe dans Termite. Les ports servent généralement à représenter

une communication avec l'extérieur du système. Cela implique d'avoir des références à des ressources extérieures, comme la console (le terminal), un fichier ou une ressource du système d'exploitation.

Dans le système Termite, il est nécessaire que les valeurs envoyées dans des messages conservent leur sens lorsqu'elles sont consultées sur un autre noeud que le noeud d'où origine de cette valeur. Il est donc nécessaire qu'une référence locale puisse conserver son sens lorsqu'elle est consultée à distance.

La solution à ce problème est d'abstraire les ports en utilisant des processus. Au lieu d'avoir des objets "port", nous avons des processus qui connaissent ces ressources locale et sont capable de les utiliser. Une référence à un port sera donc un identificateur de processus, ce qui permettra d'envoyer la référence à un noeud distant et celui-ci pourra utiliser le port sans distinction sur la localité de celui-ci. Ceci implique que les fonctions d'entrée/sortie doivent être adaptées à l'utilisation de ce genre de port.

La solution d'abstraire les ports du système par des processus diffère de celle d'abstraire les structures de données mutables. Dans le cas des structures de données mutable on utilise l'état d'un processus suspendu afin de représenter l'état courant de la structure. Dans le cas des ports l'idée est d'abstraire une ressource qui ne peut pas être sérialisée dans un message et de la rendre tout de même accessible au travers du système distribué. Les bénéfices sont toutefois similaires dans les deux cas : les ressources sont accessibles universellement et un modèle cohérent d'accès

concurrent en résulte.

4.2.1.3 call-with-current-continuation

La procédure `call/cc` en Scheme conserve le même comportement dans Termite. Celle-ci capture la continuation courante du processus mais ne capture pas le contenu de la boîte aux lettres et autre informations locales au processus.

4.2.2 Ajouts au langage Scheme

Le langage Scheme nécessite certains ajouts afin de lui permettre de répondre aux besoins de la programmation d'applications distribuées. Notamment, un système de processus doit être ajouté ainsi qu'une manière d'échanger des données entre les processus.

4.2.2.1 Système de processus

Il est nécessaire d'ajouter à Scheme un système de processus afin de pouvoir exprimer la concurrence dans le langage. Le document SRFI-18 [14] décrit un système de processus à mémoire partagée qui peut être ajouté à Scheme et qui est supporté par plusieurs implantations de Scheme. Le système décrit par ce document n'est pas exactement celui souhaité (car il décrit certaines fonctionnalités incompatibles avec notre modèle comme les mutex et les variables de conditions) mais il fournit la base nécessaire pour que les processus soient isolés.

En effet, la notion d'exception dans le modèle est locale à un processus (donc une erreur n'entraîne pas l'arrêt de tout le système). Avec l'absence de mutation dans le modèle, il n'y a pas de partage observable entre les processus. Donc, même si les processus tels que décrits par le SRFI-18 partagent leur espace mémoire, ceci n'est pas un problème.

4.2.2.2 Passage de messages

La manière d'échanger des données entre des processus lors d'absence de zone de mémoire partagée est d'avoir un mécanisme de passage de message entre les processus. Pour ce faire, il est nécessaire d'avoir un mécanisme d'adressage et une structure pour garder les messages envoyés dans le système.

Pour pouvoir livrer un message à un processus de destination, un mécanisme d'adressage des processus doit être créé. Ceci est réalisé avec le *pid* des processus, que l'on obtient lorsque l'on démarre un processus. Le *pid* contient assez d'information pour localiser le processus et sa *boîte aux lettres*, que ce processus soit local ou distant. Chaque processus possède une boîte aux lettres dans laquelle les messages qui sont destinés à un processus sont stockés.

4.2.2.3 La sérialisation

Afin de pouvoir réaliser l'envoi de message entre deux processus qui ne s'exécutent pas sur le même ordinateur, il est nécessaire de pouvoir formatter les informa-

tions envoyées entre les deux ordinateurs d'une manière représentable sous forme "sérielle", donc sous une forme qui pourrait être inclue dans un fichier ou une communication réseau. Cette forme doit contenir assez d'information afin d'être capable de recréer la structure originale.

Le langage Scheme ne force pas une implantation à être capable de faire ce genre d'opération, mais nous croyons que ceci reste dans l'esprit du standard, notamment des procédures `read` et `write`.

La procédure `write` peut être vue comme une procédure de sérialisation. Le standard dit qu'elle "affiche une représentation écrite de son argument sur le port spécifié" [1]. La procédure `read` agit comme une procédure de désérialisation, et sa spécification stipule qu'elle : "converti la représentation externe d'objets Scheme en les objets eux-mêmes".

Un mécanisme général de sérialisation s'intègre donc bien à l'esprit du langage, et en est une extension qui est cruciale dans la possibilité de créer une plate-forme de programmation d'applications distribuées, puisqu'il permet l'échange de valeurs arbitraires entre deux systèmes.

4.2.2.4 Gestion des exceptions

Aucune procédure n'est spécifiée dans le standard du langage Scheme afin de faire la gestion des exceptions. Par contre, deux procédures de base sont disponibles afin de faire la gestion des exceptions dans Gambit-C [15] et celles-ci sont adoptées

dans Termite : `with-exception-handler` et `with-exception-catcher`. Celles-ci permettent de spécifier une fonction qui sera appelée avec en argument l'objet de l'exception si une erreur survient ou si une exception est levée. Le comportement exact de ces fonctions est décrit dans la section 5.1.5.

4.2.2.5 Filtrage

Le filtrage est un outil utile qui permet de spécifier d'une manière compacte et expressive les tests sur la forme d'une donnée et la déstructuration de celle-ci. Le filtrage est en fait un sous-langage ajouté à Scheme grâce à l'utilisation de macros. L'utilisation de celle-ci est nécessaire puisque les règles d'évaluation des clauses de filtrage sont différentes des clauses normales d'évaluation en Scheme. Les formes utilisant le filtrage dans Termite sont `match` et `recv`, décrites à la section 5.1.4. L'implantation du filtrage est décrite à la section 7.1.4.

Ce chapitre discute de la philosophie et de la forme du langage distribué Termite. Celui-ci repose sur le concept, exploité dans Erlang, de la concurrence par passage de message. Un grand nombre de processus isolés peuvent communiquer entre eux pour effectuer un travail. Un modèle simple des fautes par l'échec des communications sert de base à la construction d'abstractions robustes. La syntaxe du langage et sa forme générale provient de Scheme. Le mélange des deux langages, qui s'effectue d'une manière harmonieuse, nous donne de nouvelles capacités par rapport à Erlang ou à Scheme pris indépendamment. Dans le cas de Erlang, Termite

ajoute principalement les continuations et les macros. Dans le cas de Scheme, Termite ajoute le modèle de concurrence par passage de message et la programmation distribuée.

CHAPITRE 5

DOCUMENTATION DU SYSTÈME TERMITE

Cette section documente le système Termite qui a été implanté avec Gambit-C [15]. Elle peut servir de référence lors de la programmation dans le langage et de l'utilisation du système.

5.1 Le langage de base

Les éléments de base du système sont décrits dans cette section. Ces éléments sont fondamentaux et le reste du système est généralement construit en utilisant ceux-ci. Ces éléments sont les types de donnés, les opérateurs sur les processus et les opérations d'envoi et de réception de message.

5.1.1 Types de donnée

Trois types de donnée supplémentaires à ceux trouvés dans Scheme se retrouvent dans Termite : les *pid*s (identificateurs de processus), les *node*s (identificateurs de noeuds) et les *tag*s (symboles dits *universellement uniques*, donc uniques au travers du système distribué complet).

5.1.1.1 pid

Un *pid* est un identificateur de processus. Les identificateurs de processus sont universellement uniques. Ils contiennent l'information nécessaire à la localisation d'un processus, afin de le retrouver lors de l'envoi d'un message.

`(spawn` *thunk*`)` $\Longrightarrow$ *pid*

5.1.1.2 node

Un *node* est un identificateur de noeud. La structure contient des informations pour identifier un système et sur la manière de le contacter dans un environnement distribué.

`(current-node)` $\Longrightarrow$ *node*

5.1.1.3 tag

Un *tag* est un identificateur universellement unique. C'est l'analogue aux identificateurs uniques décrits dans le RFC 4122 [16].

Les *tag*s sont associés à d'autre objets dans le but de les rendre uniques. Par exemple, ils sont utilisés pour distinguer les processus. Ils sont également utilisés d'une manière importante afin d'identifier une transaction entre deux processus.

`(make-tag)` $\Longrightarrow$ *tag*

5.1.2 Opérateurs sur les processus

Les formes et procédures qui servent à créer et à manipuler des processus sont données dans cette section.

5.1.2.1 spawn

La forme `spawn` est la forme de base qui permet de créer un nouveau processus. Le *tronc* donné en argument à l'intérieur de la forme est exécuté dans un nouveau processus.

```
(spawn (lambda ()
         (fib 20)
         (print 'done)))
```

$\Longrightarrow$ *pid*

5.1.2.2 self

La procédure `self` permet d'obtenir le *pid* du processus courant.

```
(self)
```

$\Longrightarrow$ *pid*

5.1.2.3 halt!

La procédure `halt!` permet d'interrompre immédiatement l'exécution du processus courant. Un message informant de l'arrêt du processus est envoyé à ses processus liés, comme si le processus terminait normalement son exécution.

5.1.3 Envoi et réception de messages

Afin de pouvoir échanger des données, les processus doivent procéder à l'envoi de messages. Chaque processus possède une boîte de réception de messages qui est une file dans laquelle les messages en attente de récupération sont stockés. Il n'y a qu'une façon d'envoyer un message, mais plusieurs façons de les récupérer.

5.1.3.1 !

L'opérateur ! sert à envoyer un message. Celui-ci attend deux paramètres : le *pid* du processus auquel le message est destiné et le message en tant que tel. L'opération réussi toujours et est non-bloquante (l'envoi de message est asynchrone) mais il est possible que le message ne se rende pas à sa destination.

Par exemple le code suivant envoie un message constitué d'une liste de nombres au processus dont le *pid* est contenu dans la variable `foo` :

```
(! foo (list 1 2 3))
```

5.1.3.2 ?

L'opérateur ? sert à récupérer le premier message disponible de la boîte de réception de message. Il est possible de spécifier le délai d'attente maximal pour un message ainsi que la valeur qui devrait être retournée lorsqu'aucun message n'est disponible avant l'expiration du délai. Dans le cas où un délai est spécifié,

et qu'aucune valeur par défaut n'est donnée, une exception signalant l'atteinte du délai maximal est levée.

```
(! (self) 42)
(?)                                    ⟹ 42
(? 2) ;; bloque un maximum de deux secondes,
      ;; puis signale une exception
```

5.1.3.3 ??

L'opérateur ?? sert à récupérer le premier message de la boîte de réception respectant un certain prédicat. La boîte de réception de messages est parcourue jusqu'à ce qu'un message satisfaisant le prédicat puisse être récupéré. Il est également possible de spécifier un délai maximal d'attente pour obtenir un message qui satisfait le prédicat.

```
(! (self) 21)
(! (self) 42)
(?? even?)                             ⟹ 42

;; definition equivalente de '?'
(define (?) (?? (lambda (x) #t)))
```

5.1.4 Filtrage

Deux formes spéciales permettent de faire du filtrage ("pattern matching") dans Termite. Celles-ci sont `match` et `recv`. Elles implantent un sous-langage qui ajoute de la puissance expressive à Termite. La première forme spéciale permet de filtrer

une donnée quelconque alors que la deuxième sert à faire la récupération sélective de message dans la boîte aux lettres du processus.

5.1.4.1 match

La forme `match` permet de faire du filtrage sur une donnée quelconque. Un filtre est composé de clauses. Chaque clause spécifie un patron. Celui-ci est associé avec la donnée et permet à la fois de "déconstruire" une valeur en ses composantes et d'introduire de nouvelles variables dans l'environnement lexical.

La forme générale d'un appel à `match` est la suivante :

```
(match value
  (<pattern1> <code1>)
  (<pattern2> (where <condition>) <code2>)
  ...)
```

Un patron peut être constitué à l'aide de plusieurs éléments :

- une valeur *littérale* ;
- une valeur composée ;
- une variable qui sera liée à la valeur correspondant sa position ;
- une référence à une variable.

Les clauses sont essayées séquentiellement, dans l'ordre spécifié dans le code. Un appel à `match` qui ne peut satisfaire aucune des clauses du filtre lance une exception.

Les valeurs littérales comprennent les nombres, les chaînes de caractères et les symboles. Voici un exemple d'utilisation de `quote` pour dénoter un symbole littéral :

```
(match 'allo
 ("allo" 1)
 ('allo  2)
 ( 4110  3))                    ⟹ 2
```

Les valeurs composées permettent de filtrer des listes et des vecteurs :

```
(match (list 1 2 3)
 ((1 2) #f)
 (#(1 2 3) #f)
 ((1 2 3) #t)))                 ⟹ #t
```

Afin de travailler avec la valeur passée en paramètre, il est possible d'utiliser des variables dans un patron :

```
(match (list 21 21)
  ((a b) (+ a b)))              ⟹ 42
```

Il est possible d'obtenir le *reste* des éléments d'une liste en utilisant une notation avec une liste pointée :

```
(match (list 1 2 3 4 5)
  ((first second . rest) rest)) ⟹ (3 4 5)
```

Une variable se présentant plus d'une fois dans le même patron doit prendre la même valeur chaque fois sinon le patron échoue. Dans l'exemple suivant, l'utilisation de la variable _ est spéciale car elle ne sera pas liée dans l'environnement :

```
(match (list 2 3)
  ((x x) 'equal)
  ((_ _) 'not-equal))           ⟹ not-equal
```

Il est également possible de demander à un élément d'un patron d'être égal à une valeur se trouvant dans l'environnement lexical :

```
(let ((x 42))
  (match 42
    (,x 'ok)
    (_ 'wrong)))                    ⟹ ok
```

Il est enfin possible de spécifier des *clauses gardées* avec l'aide du mot clé `where`. Une expression arbitraire peut servir de clause gardée :

```
(match 42
  (x (where (number? x) (odd? x)) 'odd-number)
  (x (where (number? x))           'even-number)
  (_                               'something-else))
                                  ⟹ even-number
```

5.1.4.2 recv

La macro `recv` utilise le même sous-langage de filtrage que `match` afin d'effectuer la récupération sélective d'un message dans la boîte aux lettres du processus courant. La syntaxe est légèrement différente, puisqu'il n'y a pas de valeur à spécifier sur laquelle opérer. Les clauses sont, comme pour `match`, testées dans l'ordre textuel, mais un échec de filtrage a pour effet de réessayer le filtrage avec le message suivant de la boîte aux lettres. Si aucun autre message n'est disponible, le processus se met en attente. Si le message courant satisfait le filtre, il est retiré de la boîte aux lettres et le prochain appel à une procédure de récupération sélective de message examinera la boîte aux lettres à partir du début.

Voici la forme générale d'un appel à `recv` :

```
(recv
  (<pattern1> <code1>)
  (<pattern2> (where <condition>) <code2>)
  ...
  (after <timeout> <code3>))
```

On remarque la présence du mot-clé spécial `after`. Celui-ci, qui doit apparaître en dernier dans la liste des clauses, permet de spécifier un délai d'attente maximal lorsqu'aucun message de la boîte aux lettres ne peut satisfaire le filtre. Lorsque le délai est échu, le code associé à la clause `after` est exécuté. Le cas avec un délai de 0 est utile pour tenter la réception de message sans faire bloquer le processus.

5.1.5 Gestion d'erreurs

Des fonctions sont disponibles afin de signaler et de manipuler des conditions d'erreur et des situations exceptionnelles. La procédure `raise` est utilisée pour signaler une erreur ou une exception. Intercepter et gérer les conditions d'erreur se fait avec les procédures `with-exception-catcher` et `with-exception-handler`. Finalement, une série de fonctions permet de contrôler les liens entre les processus et, ainsi, la propagation des erreurs.

5.1.5.1 raise

La procedure `raise` permet de signaler une condition exceptionnelle. Un appel à `raise` va lever l'exception qui est la valeur passée en paramètre. Ceci va entraîner

l'exécution du gestionnaire d'exception englobant.

Dans le cas où on exécute du code en ligne de commande et que l'on laisse le gestionnaire d'exception de Gambit gérer l'exception, on obtient :

```
> (raise 'error)
** ERROR IN (console)@1.1 -- This object was raised: error
1>
```

Le 1> signifie que l'on est dans une sous-boucle d'interaction qui nous permet de réagir à l'erreur. Il faudra référer à la documentation de Gambit au sujet de l'interaction en ligne de commande.

5.1.5.2 with-exception-catcher

La procedure `with-exception-catcher` est celle décrite dans le manuel de Gambit-C 4. Celle-ci permet d'installer un gestionnaire d'exception pour l'exécution d'un *tronc*. La continuation du gestionnaire d'exception est celle de l'appel à `with-exception-catcher`.

Dans l'exemple suivant, on signale une exception à l'aide de `(raise 'error)` ; l'exception est prise en charge par le gestionnaire d'exception `(lambda (exception) 42)`, qui retourne 42 :

```
(with-exception-catcher
  (lambda (exception) 42)
  (lambda () (raise 'error) 123))
                                  ⟹ 42
```

On voit que le contrôle sort de la forme `with-exception-catcher` après l'invocation du gestionnaire d'exception et que c'est la valeur résultante de l'application de celui-ci qui est le résultat final.

5.1.5.3 with-exception-handler

La procedure `with-exception-handler` est celle décrite dans le SRFI-18. Celle-ci permet d'installer un gestionnaire d'exception pour l'exécution d'un *tronc*. La continuation du gestionnaire d'exception est celle de l'appel qui aura soulevé l'exception entraînant l'invocation de `with-exception-handler`.

Dans l'exemple suivant, on signale une exception qui est prise en charge par le gestionnaire d'exception :

```
(with-exception-handler
  (lambda (exception) 42)
  (lambda () (raise 'error) 123))
                                ⟹ 123
```

On remarque que dans le cas de `with-exception-handler`, après l'invocation du gestionnaire d'exception, le contrôle est retourné à la continuation du code qui a causé la levée de l'exception.

5.1.5.4 full-link

La procédure `full-link` crée un lien bidirectionnel entre le processus courant et un autre processus. Une exception soulevée dans l'un ou l'autre des processus et non gérée par un gestionnaire d'exception installé avec `with-exception-handler`

ou `with-exception-catcher` est propagée à l'autre processus. L'exception est soulevée dans l'autre processus dès que celui-ci tente de récupérer un message.

Dans l'exemple suivant, le processus `pid` se met en attente d'un message. Quoiqu'aucun message ne lui est envoyé explicitement, l'erreur survenant dans le processus principal (suite à la division par zéro) lui est communiquée et cause l'arrêt de son exécution :

```
(define pid
  (spawn
    (lambda ()
      (?)))) ;; crash

(full-link pid)
(/ 1 0) ;; crash
```

5.1.5.5 outbound-link

La procédure `outbound-link` crée un lien unidirectionnel entre le processus courant et un autre processus. Une exception soulevée dans le processus courant est propagée à l'autre processus

L'exemple suivant aura le même comportement que l'exemple précédent (le cas où le lien bidirectionnel avait été établi), car le lien de propagation est *outbound*, c'est à dire vers l'extérieur du processus principal et dirigé vers le processus `pid` :

```
(define pid
  (spawn
    (lambda ()
      (?)))) ;; crash
```

```
(outbound-link pid)
(/ 1 0) ;; crash
```

5.1.5.6 inbound-link

La procédure `inbound-link` crée un lien unidirectionnel entre le processus courant et un autre processus. Une exception soulevée dans l'autre processus sera propagée au processus courant.

Dans l'exemple suivant, contrairement aux deux exemples précédents, le processus désigné par la variable `pid` ne reçoit pas l'exception, puisque le sens de propagation de l'exception est *inbound*, soit de `pid` vers le processus principal :

```
(define pid
  (spawn
    (lambda ()
      (?))))

(inbound-link pid)
(/ 1 0) ;; crash
```

5.1.5.7 spawn-link

La procédure `spawn-link` permet de démarrer un processus et de lier celui-ci bidirectionnellement, et ce de manière atomique. Si l'on effectuait un appel à `spawn` suivi d'un appel à `full-link`, il serait possible qu'une exception survienne dans le processus nouvellement créé et qu'elle ne soit pas propagée parce que le lien ne serait pas encore établi. `spawn-link` évite ce problème.

5.1.6 Programmation distribuée

Afin de pouvoir faire de la programmation distribuée, il est nécessaire d'avoir une manière de spécifier l'exécution d'un calcul sur un ordinateur distant. Deux fonctions de base sont fournies, soit `remote-spawn` et sa variante `remote-spawn-link`.

5.1.6.1 remote-spawn

La procédure `remote-spawn` prend un noeud et un *tronc* en paramètre et démarre un nouveau processus avec ce tronc sur le noeud spécifié. Le *pid* du processus nouvellement démarré est retourné. Voici un exemple où un processus distant est créé pour répondre à des requêtes *ping* :

```
(define remote-pong-server
  (remote-spawn remote-node
    (lambda ()
      (let loop ()
        (recv ((from 'ping)
               (! from 'pong)))
        (loop)))))
```

5.1.6.2 remote-spawn-link

La procédure `remote-spawn-link` effectue la même tâche que `remote-spawn`, mais en plus crée atomiquement un lien bidirectionnel entre le processus créateur et le processus créé. `remote-spawn-link` est à `remote-spawn` ce que `spawn-link` est à `spawn`.

```
(remote-spawn-link remote-node
  (lambda ()
    (/ 1 0)))

(?) ;; error, division by zero
```

5.2 Les bibliothèques

Certaines fonctionnalités sont fréquemment nécessaires lors de l'écriture d'applications distribuées. Ces outils sont construits à partir des fonctionnalités de base et tentent de capturer l'usage habituel.

5.2.1 Abstraction de patrons de synchronisation

Cette section décrit certaines fonctions permettant de faire l'abstraction de patrons de synchronisation fréquents.

5.2.1.1 !?

L'opérateur `!?` est l'opérateur d'appel de procédure distante classique (RPC). Il sert à faire une requête synchrone à un autre processus. Il effectue un envoi de message et bloque le processus qui fait l'envoi jusqu'à la réception du message correspondant à la requête. Pour un appel `(!? to term)` la forme d'un message de requête envoyé est `(from tag term)`, où `from` est l'originateur de la requête, `tag` sert à identifier uniquement la requête et `term` est le contenu de la requête. Il est important de connaître la représentation des messages envoyés par `!?` car

ceux-ci doivent être interprétés par le serveur.

Il est possible de spécifier un délai maximal pour l'attente d'une réponse ainsi que de spécifier une valeur à être retournée si aucune n'arrive à temps, comme pour les opérations de récupération de message.

Voici un exemple simple de scénario d'utilisation de la procédure `!?` :

```
(define (square-server)
  (recv
    ((from tag x) (! from (list tag (* x x)))))
  (square-server))

(!? square-server 12)           ⟹ 144
```

5.2.1.2 on

L'opérateur on est un opérateur qui sert à exécuter un *tronc* sur un noeud distant et à retourner la valeur obtenue de l'évaluation de celui-ci. L'opérateur est utile lorsque l'on veut obtenir une valeur venant du contexte d'un autre noeud. L'évaluation du *tronc* est faite de manière synchrone donc le processus qui fait appel à on bloque jusqu'au retour de la valeur. Voici un exemple simple qui montre l'évaluation de la procédure `current-node` dans le contexte d'un autre noeud :

```
(equal? some-node
        (on some-node
            (lambda () (current-node))))
                                ⟹ #t
```

5.2.1.3 future

La forme `future` permet de spécifier que l'exécution d'un bout de code devrait se produire en parallèle. La valeur retournée par `future` est une *promesse* et la valeur de l'évaluation de l'expression passée à `future` est obtenue en appelant `touch` sur la promesse. Optionnellement, il est possible de spécifier un délai maximal d'attente à `touch`, ainsi qu'une valeur par défaut.

```
(define promise (future (long-computation)))
;; ... some code
(touch promise)                    ⟹ result
```

5.2.2 Migration

Deux procédures de migration sont fournies avec Termite. Leur implantation est discuté à la section 6.1.1.1. La migration permet de déplacer un processus sur un autre noeud afin qu'il y poursuive son exécution. Les deux variantes de migration se distinguent par le support qu'ils offrent pour que la migration soit transparente. La première forme n'offre aucun support alors que la seconde offre un support à l'aide d'un *proxy*.

5.2.2.1 migrate-task

La procédure `migrate-task` doit être appelée avec une référence à un noeud en argument. Cette procédure entraîne la migration du processus vers un autre noeud : son exécution se poursuivra à cet endroit. Les messages en attente dans la boîte

aux lettres du processus sont *abandonnés* et les liens du processus sont *brisés*. Tout message envoyé au processus dans le futur *ne sera pas* automatiquement relayé jusqu'au processus à sa nouvelle destination.

Dans l'exemple suivant, un processus est migré afin qu'il effectue un calcul coûteux sur le noeud `foreign-node`, puis ensuite qu'il revienne sur le noeud original afin d'imprimer le résultat du calcul :

```
(spawn
  (lambda ()
    (migrate-task foreign-node)
    (let ((result
           (long-computation)))
      (migrate-task original-node)
      (print (list result: result)))))
```

5.2.2.2 migrate-with-proxy

La procédure `migrate-with-proxy` doit être appelée avec une référence à un noeud en argument. Cette procédure entraîne la migration du processus vers un autre noeud : son exécution se poursuivra à cet endroit. Les messages en attente dans la boîte aux lettres du processus sont tous envoyés à la nouvelle copie du processus et les liens du processus sont conservés. Tout message envoyé au processus dans le futur est automatiquement relayé jusqu'au processus à sa nouvelle destination par un proxy restant sur le noeud original.

L'exemple suivant illustre un processus qui répond à des messages `ping` sa position courante et migre sur demande :

```
(define position-reporter
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'ping)
           (! from (list tag (current-node))))
          (('migrate node)
           (migrate-with-proxy node)))
        (loop)))))

(!? position-reporter 'ping)     ⟹ node1
(! position-reporter (list 'migrate node2))
(!? position-reporter 'ping)     ⟹ node2
(! position-reporter (list 'migrate node3))
(!? position-reporter 'ping)     ⟹ node3
```

5.2.3 Interface avec l'extérieur

Quelque fonctionnalités sont offertes afin d'effectuer des entrées et sorties pour permettre l'interaction avec des éléments extérieurs au système Termite.

5.2.3.1 Les ports d'entrée/sortie

Les ports d'entrée/sortie et les opérateurs associés (`write`, `read`, etc.) que l'on retrouve en Scheme sont surdéfinis dans Termite afin de permettre leur utilisation dans un contexte distribué. Les types définissant les ports d'entrée/sortie sont redéfinis (en utilisant la technique illustrée dans l'exemple à la section 6.1.3.3). Les procédures sont redéfinies afin de fonctionner avec ces nouveaux types. Ceci permet l'utilisation des ports dans un contexte distribué.

L'exemple suivant illustre comment un transfert de fichier peut être effectué à

l'aide des opérations d'entrée/sortie de Termite. Une lecture du contenu entier d'un fichier est effectuée puis celui-ci est écrit directement sur le port de sortie :

```
(define (upload filename node)
  (let ((output (on node (lambda ()
                           (open-output-file filename)))))
    (call-with-input-file filename
      (lambda (input)
        (write (read-line port #f) output)
        (close output)))))
```

5.3 Utilisation du système

Afin d'utiliser le système, il est nécessaire de faire un appel à la fonction d'initialisation `node-init` de Termite. La fonction d'initialisation attend en argument un objet *node* qui définit sur quel port TCP le noeud actuel écoute pour des demandes de connexions venant d'autres noeuds Termite. Un exemple de programme Termite minimaliste est donné dans le fichier `start.scm` livré avec les sources de Termite [17].

CHAPITRE 6

EXEMPLES D'APPLICATIONS DISTRIBUÉES

Dans ce chapitre nous donnons plusieurs exemples de programmation avec Termite qui permettent d'expliquer, de démontrer l'utilisation et de motiver le modèle. Les exemples sont séparés en deux sections : dans la première section nous examinons des techniques de programmation d'applications distribuées. Dans la deuxième section nous donnons quelques programmes complets qui ont été réalisés avec Termite et qui exposent les forces du langage.

6.1 Techniques

Cette section expose plusieurs techniques de programmation. D'abord il est question de l'utilité des continuations de première classe pour exprimer certains concepts. Ensuite le modèle proposé par la plate-forme Erlang/OTP [18] pour développer des applications robustes est présentée, avec des exemple de code Termite.

6.1.1 Utilisation des continuations

Les continuations de première classe permettent de contrôler d'une manière arbitraire le flot de contrôle d'un processus. Ceci peut être exploité pour exprimer des concepts comme la migration de tâche, le clonage de processus et la mise à jour

dynamique de code.

6.1.1.1 Migration

La migration de tâche signifie que le travail accompli par un processus est transféré à un autre processus. Cela implique donc que le processus de départ doit capturer la continuation courante puis l'envoyer dans un message. Le processus qui va poursuivre le traitement résume le contrôle de cette tâche.

Un processus qui fait un appel à la fonction `migrate-task` va entraîner la migration du flot de traitement vers le processus désigné par l'argument `to`, puis terminer l'exécution du processus courant :

```
(define (migrate-task node)
  (call/cc
   (lambda (k)
     (remote-spawn node (lambda () (k #t)))
     (halt!))))
```

La migration avec proxy signifie également qu'un processus suspend son exécution pour la reprendre sur un autre noeud, mais lors de la migration un processus est laissé derrière afin de faire suivre les messages qui sont envoyés au processus maintenant migré. Une forme simple de migration avec proxy s'implémente comme suit :

```
(define (proxy pid)
  (let loop ()
    (! pid (?))
    (loop)))

(define (migrate/proxy node)
```

```
  (call/cc
   (lambda (k)
     (proxy
      (remote-spawn-link node (lambda () (k #t)))))))
```

6.1.1.2 Clonage de processus

On appelle le *clonage de processus* la "copie" du comportement d'un processus par un autre. Avec la coopération du processus original, il est possible de créer un nouveau processus qui réagit de la même manière que ce premier processus.

Dans cet exemple, le processus original va créer un clone de lui-même :

```
(define original
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'clone)
           (call/cc
            (lambda (clone)
              (! from (list tag (lambda ()
                                  (clone (void))))))))
          (loop))))))

(define clone (spawn (!? original 'clone)))
```

6.1.1.3 Mise à jour dynamique de code

Il est également possible d'utiliser la capture de continuation afin de remplacer dynamiquement le code d'un processus sans en interrompre l'exécution. Un processus devra accepter d'exécuter le code reçu dans un message.

Ici, on voit qu'un serveur `server` a été démarré. Son code contient un bogue : on peut donc définir un nouveau serveur qui répond aux mêmes messages, mais avec le code corrigé :

```
(define server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((’update k)
           (k #t))

          ((from tag ’ping)
           (! from (list tag ’gnop))))     ; bug
        (loop)))))

(define new-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((’update k)
           (k #t))

          ((from tag ’clone)
           (call/cc
            (lambda (k)
              (! from (list tag k)))))

          ((from tag ’ping)
           (! from (list tag ’pong))))     ; fixed
        (loop)))))

(!? server ’ping)                ⟹ gnop

(let ((replacement (!? new-server ’clone)))
  (! server (list ’update replacement)))

(!? server ’ping)                ⟹ pong
```

6.1.2 Abstractions robustes

Afin de pouvoir construire efficacement des applications distribuées robustes (qui résistent aux fautes, comme les déconnexions de noeuds par exemple), il est utilise d'abstraire les problèmes fréquemment rencontrés dans ce genre d'application. Deux techniques permettent d'atteindre ce but : l'abstraction de composantes génériques et la création d'arbres de supervision. Ces techniques sont décrites dans la thèse de Joe Armstrong [11], et nous en donnons ici une implantation partielle comme exemple. Nous exposons une relation travailleur/superviseur, le travailleur étant un composant générique.

6.1.2.1 Travailleur

La première technique propose l'utilisation de composants déjà partiellement construits. Ceux-ci abstraient le travail habituel fait par certains types génériques de processus, y compris la gestion de l'interaction concurrente avec les autre processus. Il ne reste par la suite qu'à les paramétrer (avec un "plugin") afin de spécifier leur comportement.

Beaucoup de processus dans un système distribués agissent comme "serveur" : ils vont généralement rester à l'écoute afin de répondre à des requêtes et conserver un certain état local. L'abstraction de serveur générique permet d'éviter d'avoir à écrire du code qui est dépendant de la concurrence ; le code qui sert à paramétrer

le serveur est purement séquentiel.

Ce "plugin" est composé de 4 fermetures, dont chacune sert à gérer une situation particulière. Le serveur conserve un état. Cet état est passé à chaque appel et retourné, potentiellement modifié, par chaque appel à une fermeture.

La première fermeture (*init*) est appelée lors du démarrage du serveur. Celle-ci reçoit des argument pour configurer le comportement du serveur, et renvoie l'état initial.

La deuxième fermeture (*call*) est celle qui implante les fonctionnalités utilisées par les clients qui consultent ce serveur. Cette fermeture peut renvoyer un nouvel état pour le serveur. Les appels au serveur implantés par la fermeture *call* sont synchrones (une réponse est toujours attendue).

La troisième fermeture (*cast*) est utilisée pour les fonctionnalités reliées à la gestion du serveur, par exemple pour redémarrer ou arrêter l'exécution du serveur. Les appels implantés par cette fermeture sont asynchrones (une réponse n'est pas demandée).

La quatrième fermeture (*terminate*) est invoquée lors de l'arrêt et la destruction du serveur. Celle-ci permet d'exécuter du code afin de faire un "nettoyage" à la fin de l'exécution du serveur.

L'exemple donné ici est celui d'un serveur qui implante une table d'association clé/valeur. Celui-ci supporte deux opération pour l'usager : l'enregistrement d'une nouvelle association (`kv:store`) et la recherche de la valeur associée à une clé

(kv:lookup). Il y a aussi deux fonctions servant au démarrage et à l'arrêt du serveur, soient kv:start et kv:stop.

```
(define key/value-server-plugin
  (make-server-plugin
   ;; INIT
   (lambda (args)
     (print "Key-Value server starting")
     (make-dict))

   ;; CALL
   (lambda (term from state)
     (match term
       (('store key val)
        (dict-set! state key val)
        (values (void) state))

       (('lookup key)
        (values (dict-ref state key) state))))

   ;; CAST
   (lambda (term state)
     state)

   ;; TERMINATE
   (lambda (reason state)
     (info "Key-Value server terminating"))))

(define (kv:start)
  (server:start key/value-server-plugin))

(define (kv:stop server)
  (server:cast server 'stop))

(define (kv:store server key val)
  (server:call server (list 'store key val)))

(define (kv:lookup server key)
  (server:call server (list 'lookup key)))
```

6.1.3 Abstractions de contrôle et d'état

Il est possible d'utiliser les processus et l'envoi de message afin d'exprimer des abstraction de flot de contrôle et d'état. Nous donnons ici deux exemples d'abstraction de contrôle, soit la forme `future` et le balancement de charge et un exemple d'abstraction de l'état (une paire mutable).

6.1.3.1 La forme future

La forme `future` permet d'avoir une manière simple et concise d'exprimer un calcul parallèle. Cette forme est documentée à la section 5.2.1.3.

On remarque ici l'utilisation de deux techniques utiles. Premièrement `future` est une macro car on désire contrôler comment sera évalué son argument. Deuxièmement, un nouveau type de donnée est créé pour les promesses. Même si la promesse est implantée par un processus, l'abstraire avec un nouveau type permet d'avoir un prédicat de type disjoint (`promise?`). On peut également remarquer l'utilisation de `match` pour gérer les arguments optionnels à `touch`.

```
(define-type promise
  id: 20c86f4a-de87-470f-b974-0b6fd7979a7a
  unprintable:
  pid)

(define-macro (future . body)
  `(future* (lambda () ,@body)))

(define (future* thunk)
  (make-promise
   (spawn-link
```

```
    (lambda ()
      (let ((value (thunk)))
        (let loop ()
          (recv
            ((from tag 'ref)
             (! from (list tag value))))
          (loop)))))))

(define (touch promise . opt)
  (let ((pid (promise-pid promise)))
    (match opt
      ('()
       (!? pid 'ref))
      ((timeout)
       (!? pid 'ref timeout))
      ((timeout default)
       (!? pid 'ref timeout default)))))
```

6.1.3.2 Balancement de charge

Il peut être profitable de répartir la charge d'un serveur sur plusieurs ordinateur. L'exemple de cette section montre un mécanisme simple de balancement de charge. Celui-ci se sépare en deux partie : la partie de mesure de charge de l'ensemble des noeuds qui peuvent recevoir des tâches et la partie de répartition des tâches en tant que telle.

La partie de mesure de la charge des noeuds est composé d'un processus *superviseur* et de plusieurs processus *travailleurs*. Un processus travailleur va s'exécuter sur chaque noeud de la grappe et rapporter périodiquement au processus superviseur la charge du noeud sur lequel il s'exécute. Deux procédure permettent d'obtenir de l'information sur l'état de la grappe : `less-loaded` et `average-load`.

```
(define (start-meter supervisor)
  (let loop ()
    (! supervisor
       (list 'load-report
             (self)
             (local-loadavg)))
    (recv (after 1 'ok)) ; pause for a second
    (loop)))

(define (meter-supervisor meter-list)
  (let loop ((meters (make-dict)))
    (recv
      ((='load-report from load)
       (loop (dict-set meters from load)))
      ((from tag 'less-loaded)
       (let ((min (find-min (dict->list meters))))
         (! from (list tag (pid-node (car min)))))
       (loop dict))
      ((from tag 'average-load)
       (! from (list tag
                     (list-average
                      (map cdr (dict-list meters)))))
       (loop dict)))))

(define (less-loaded supervisor)
  (!? supervisor 'less-loaded))

(define (average-load supervisor)
  (!? supervisor 'average-load))
```

La deuxième partie du système de balancement de charge reçoit les demandes de répartition et les envoie sur le noeud qui est le plus légèrement chargé à ce moment.

```
(define (start-work-dispatcher load-server)
  (spawn
    (lambda ()
      (let loop ()
        (recv
```

```
          ((from tag ('dispatch thunk))
           (let ((less-loaded-node
                  (less-loaded load-server)))
             (spawn
               (migrate less-loaded-node)
               (! from (list tag (thunk)))))))
        (loop)))))

(define (dispatch dispatcher thunk)
  (!? dispatcher (list 'dispatch thunk)))
```

6.1.3.3 Structure de donnée mutable

L'exemple donné dans cette section est simple mais fondamental. Celui-ci illustre qu'il n'est pas nécessaire d'avoir la présence de mutation explicite de structure de donnée dans Termite pour avoir des structures de donnée mutable : il est possible d'utiliser l'état d'un processus suspendu pour ce faire. Cet exemple utilise la technique décrite à la section 6.1.3.1 afin d'avoir un type de donné disjoint et testable avec un prédicat.

```
(define-type cell
  id: 713cb0a4-16ea-4b18-a18e-7a9e33e7b92b
  unprintable:
  pid)

(define (cell obj)
  (make-cell
   (spawn
     (lambda ()
       (cell-loop obj)))))

(define (cell-loop obj)
  (recv
    ((from tag 'ref)
```

```
     (! from (list tag obj))
     (cell-loop obj))

    (('set! obj)
     (cell-loop obj)))))

(define (cell-ref c) (!? (cell-pid c) 'ref))
(define (cell-set! c obj) (! (cell-pid c) (list 'set! obj)))

(define c (cell 42))
(cell-ref c)                    ⟹ 42
(cell-set! c 123)
(cell-ref c)                    ⟹ 123
```

6.1.3.4 Utilisation de macros

On a vu que le volume de code à écrire pour la création d'un type d'enregistrement mutable tel que donné dans l'exemple précédent pouvait être volumineux. Heureusement, il est possible d'utiliser une macro pour créer une forme spéciale pour permettre la création d'enregistrement mutable mais avec une syntaxe plus compacte. La forme `define-termite-type`, dont le code est donné à la section A.3, permet d'exprimer une structure mutable un peu de la même façon que la forme `define-type` dans Gambit-C permet de le faire :

```
(define-termite-type point
  id: 308a52f6-fc90-46ff-bc77-325959b402c3
  x
  y)
```

Une telle déclaration crée la procédure `make-point`, qui permet de créer un enregistrement comprenant les coordonnées x et y d'un point. Un processus est

utilisé pour implanter l'enregistrement. Les autres procédures créées sont `point?` pour le test de type disjoint, `point-x` et `point-y` pour accéder aux champs de la structure et `point-x-set!` et `point-y-set!` pour changer le contenu d'un champ de la structure. Cet exemple démontre bien l'utilité des macros afin d'abstraire et de simplifier un patron d'usage fréquent.

6.2 Programmes complets

Cette section comprend deux exemples de programmes complets et utiles. Le premier exemple est le programme *Dynamite* (une compression des mots *dynamique* et *Termite*), qui est un cadre d'applications pour le développement de programmes avec une interface *web* plus dynamique que ce que le modèle traditionnel propose. Le deuxième programme, nommé *Schack*, est un jeux multijoueurs interactif, implanté en utilisant l'infrastructure Dynamite.

6.2.1 Dynamite

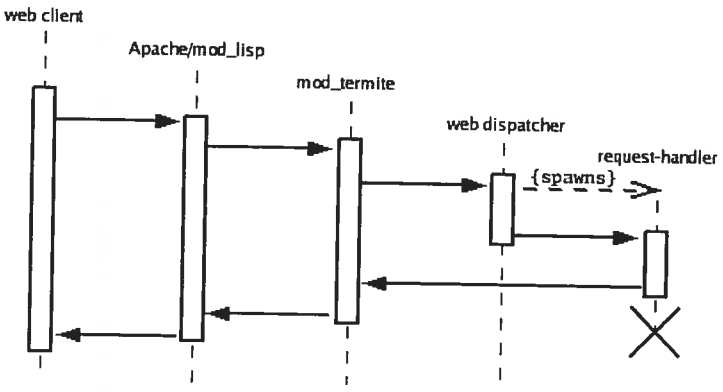

FIG. 6.1 – Diagramme d'interaction lors du service d'une page dynamique.

Le protocole *HTTP*, typiquement utilisé pour les communications entre un navigateur et un serveur web, est essentiellement sans état. Chaque interaction entre le client et le serveur est indépendante. Ceci contraint le genre d'interaction faite lors d'usage d'applications construites en utilisant ce protocole. Ces applications sont généralement formées d'une suite de formulaires et de pages présentant de l'information. Typiquement, à chaque interaction de l'usager une nouvelle page est chargée qui contiendra la "suite" de l'application. Il est tout de même possible d'avoir des applications plus dynamiques dans un navigateur web. Les techniques pour y arriver peuvent se classer dans deux catégories : celles qui s'exécutent à l'aide d'un *plugin* afin d'incorporer une applications à l'intérieur d'une page et celles qui étendent l'infrastructure de base.

Nous nous intéressons particulièrement au paradigme où l'infrastructure habituelle est étendue afin de permettre un plus haut degré d'interactivité. Alors que le modèle de base suppose qu'une page est une entité immuable qui est affiché dans un navigateur, le fait est qu'il est possible d'interagir programmatiquement avec la structure et le contenu d'une page. La plupart des navigateurs supportent à cette fin Javascript [19] et le modèle DOM [20] ("Document Object Model"), qui permettent de contrôler le contenu d'une page. Il est possible de faire des requêtes aux serveur à partir de code Javascript qui est exécuté côté client et de récupérer les informations que le serveur renvoie. Ceci permet donc de faire des requêtes au serveur "à l'intérieur" d'une page web. Ce genre de technique actuellement populaire

est surnommée "AJAX" [21] ("Asynchronous Javascript And XML).

Dynamite sert de cadre à la programmation d'applications de type *AJAX*. Cet exemple est approprié car il démontre l'utilité de Termite à la solution de problèmes actuels pour lesquels une approche particulière n'est pas largement acceptée. *Dynamite* se base sur l'idée qu'il devrait être possible de programmer des applications web avec des interfaces graphiques un peu de la même manière que ceci serait fait dans un contexte normal de programmation d'applications avec interfaces graphiques. À cette fin, on utilise la notion de composante graphique (une sorte de gadget logiciel [1]).

6.2.1.1 Architecture générale de Dynamite

Les requêtes HTTP se font normalement du client au serveur, mais nous désirons avoir une interaction complète entre le client et le serveur. La communication se doit alors d'être bidirectionnelle afin que l'application et le serveur puissent interagir activement. Les communications se font à l'aide d'un objet Javascript nommé `XMLHttpRequest` ; c'est cet objet qui permet de faire des requêtes HTTP à l'intérieur du code d'une page. Ceci permet de faire des requêtes au serveur sans avoir à rafraîchir la page au complet. Afin de pouvoir faire une requête du serveur au client, nous utilisons une connexion persistante, que nous nommons *pompe d'évènements*, entre le client et le serveur. Le client fait une demande et attend

[1] "widget" en anglais.

une réponse du serveur durant un certain moment. Si au bout d'un certain temps le serveur n'a pas de message à faire au client, le serveur répond tout de même un message qui indique que le délai maximal a été atteint. Sur réception de ce message, le client recrée une nouvelle connexion permanente avec le serveur. Cette manière de faire évite qu'un délai trop grand sans réponse soit signalé par le client et cause l'arrêt de l'exécution du code Javascript dans la page. Les messages qui sont envoyés du serveur au client sont toujours des fragments de code Javascript valides. Ceci créée une interaction qui rappelle une *read-eval-print-loop* classique, mais où la lecture se fait depuis les évènements provenant du serveur et l'évaluation est celle de code Javascript du côté client.

Dynamite ne parle pas le protocole HTTP directement. Afin de communiquer avec un client web, le serveur *Apache* [22] est utilisé, avec le module *mod_lisp* [23]. Apache traite les demandes du serveur et invoque le module *mod_lisp* qui transforme la requête dans un format facile à traiter en Scheme. Lors de la réception d'une requête, *mod_lisp* va établir une connexion TCP avec un noeud Termite. Le programme qui s'exécute à l'autre bout du protocole *mod_lisp* est nommé *mod_termite*. Ce programme traduit les requêtes du serveur en message Termite et les fait suivre à un processus enregistré pour traiter ce message. Toute application web doit donc s'enregistrer auprès de *mod_termite* afin de recevoir un certain URL et *Dynamite* n'y fait pas exception.

C'est le processus de base de *Dynamite* qui s'occupe de traiter la requête et de

fournir une réponse à *mod_termite*. Ce processus de base comprend une liste qui associe des URLs à des fermetures et des processus. Lorsqu'un client consulte un URL associé à une fermeture, celle-ci est invoquée et sa valeur de retour est envoyée au client. Lorsque l'URL correspond à un processus, celui-ci reçoit un message contenant les paramètres de la requête du client. Le fait d'avoir deux types de ressource qui peuvent être attachés à un URL permet d'être flexible sur le genre de service fourni. Les fermetures sont généralement utilisées pour la génération de pages alors que les processus peuvent être utilisé pour représenter une activité continue côté client, comme lors de la gestion d'une page dynamique.

6.2.1.2 Les composantes de base

Deux types de composantes de base sont utilisées dans *Dynamite*. Le premier type est un élément de formulaire (que nous nommerons simplement *gadget*) comme par exemple un bouton ou une boîte de texte. Le deuxième type est un canevas, qui sert à afficher des données et à grouper d'autre composantes.

Une composante d'interface est représentée du côté client par un élément HTML. On peut penser aux gadgets comme étant des sous-éléments d'un élément HTML `form` et les canevas sont des éléments `div`. Du côté serveur, les composantes sont représentées par des processus qui conservent l'état courant des composantes, comme le texte contenu dans un champ ou l'étiquette actuelle d'un bouton, ainsi qu'une procédure permettant de créer la représentation HTML de cette composante. Il

est possible de changer la valeur ou le contenu d'une composante en lui envoyant un message et cette valeur est communiquée à l'interface afin que celle-ci reflète la valeur actuelle de la composante. Les événements subis par les composantes du côté client (un clic sur un bouton ou un changement de valeur d'une boîte de texte) sont envoyés au processus représentant du côté serveur la composante web. Une procédure de rappel associée à chaque composante permet de réagir à ces évènements. La procédure de rappel reçoit en argument une référence au processus correspondant et une valeur décrivant l'évènement survenu. À l'aide de la référence au processus il est possible d'obtenir la valeur actuelle de la composante et de la changer.

Chaque composante créée du côté serveur est associée à un processus nommé *processus d'interface* qui gère la session courante de l'application. Ce processus est associé dans une table à un URL unique sur le serveur, donc toutes les requêtes faites à cet URL sont dirigés par *mod_termite* à ce processus. Celui-ci est en fait responsable d'agir comme relais pour les messages envoyés du client vers les composantes sur le serveur. Le processus d'interface sait à quel processus local il doit associer une composante car un identificateur unique (un *tag*) est associé à chacune. L'identificateur unique est conservé dans le champ *ID* de la composante HTML et une table est conservée dans le processus d'interface qui associe l'identificateur unique avec un processus. On peut voir le processus d'interface comme étant le côté serveur de la *pompe d'évènements*. D'ailleurs, lorsque cette dernière

signale une erreur (parce que la connexion entre le client et le serveur est terminée, généralement), le processus d'interface en est informé et peut réaliser une action. Comme le processus d'interface connaît tous les gadgets relatifs à la page web, il lui est possible de réagir, comme de terminer l'exécution des processus représentant les gadgets lorsque l'application termine.

Dynamite est une application intéressante de Termite car le résultat est un cadre de programmation dans lequel il est possible d'écrire des applications web avec des interfaces dynamiques qui ont des caractéristiques semblables aux applications graphiques classiques, tant au niveau des techniques de programmation (procédures de rappel sur les gadgets, etc.) qu'au niveau des fonctionnalités résultantes. Ceci est dû à la possibilité d'abstraire le traitement concurrent qui survient et les communications réseau nécessaires.

6.2.2 Schack

Schack (une compression de Scheme et *Hack*[2]) est un jeu multijoueurs interactif avec interface web. Celui-ci est développé en utilisant *Dynamite* et exploite certains aspects de Termite, comme la notion de composantes génériques et l'utilisation de la concurrence afin de décrire de manière directe les interactions possibles entre les joueurs. L'aspect intéressant du développement de *Schack* est de voir comment il est possible de modéliser un problème à l'aide de processus concurrents. Pour

[2] *Hack* et ses descendants forment une famille de jeux avec la notion simple d'un personnage capable d'utiliser armes et magie qui descend dans un donjon peuplé de montres.

ce faire, il faut identifier les diverses parties du modèle et voir comment celles-ci interagissent.

6.2.2.1 Modélisation du problème

Schack est un jeu multijoueurs avec une interface web dynamique. Les événements qui se produisent dans le jeu sont reflétées dans l'interface des participants sans que ceux-ci aient à recharger la page. Les acteurs principaux du jeu sont des *créatures* qui parcourent un donjon dans lequel se trouve d'autres créatures et des objets. Les créatures peuvent être des héros (habituellement contrôlés par un joueur) ou des monstres (contrôlés par un joueur ou par une intelligence artificielle). Les créatures peuvent s'attaquer entre elles et ramasser et utiliser les objets qu'elles trouvent dans le donjon. Chaque créature possède donc un certain ensemble d'objets, que l'on nomme *inventaire*. Une créature possède également certains attributs comme par exemple un nombre de points de vie et un nombre de points d'expérience. Les créatures sont modélisées par une structure de donnée mutable (en utilisant `define-termite-type`) avec un champ pour chacun de leurs attributs, et un champ contenant une référence vers un objet représentant l'inventaire de cette créature. Un *inventaire* est programmé avec un serveur générique qui permet d'énumérer les éléments d'un ensemble ainsi que l'ajout et le retrait d'éléments de cet ensemble. Les créatures évoluent à l'intérieur d'un donjon qui est composé de *pièces* interconnectées. Chaque *pièce* doit garder trace des créatures qui y sont présentement et

des objets qui se trouvent sur son sol. Les pièces sont implémentées comme des serveurs génériques qui contiennent dans leur état un référence vers un inventaire pour le sol, un inventaire pour les créatures se trouvant dans la pièce et des références vers les pièces voisines. Les *objets* (armes, potions, etc.) sont chacun implanté à l'aide d'une structure qui contient des informations telles que le nom et la descriptions de l'objet ainsi qu'une procédure de rappel qui permet d'appliquer l'effet de cet objet sur une créature.

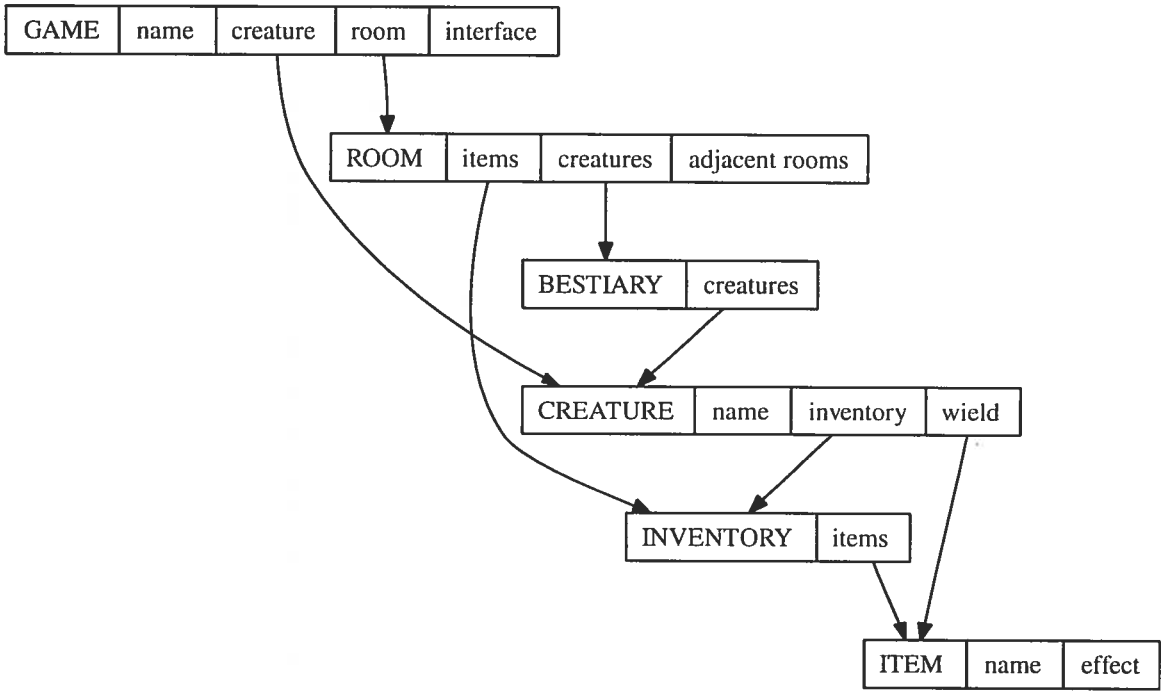

FIG. 6.2 – Relations entre les données de Schack

La gestion de l'affichage de chacun des éléments du jeu est faite en associant à ces éléments des composantes génériques de *Dynamite*. L'interface de jeu a 5 panneaux (voir la figure 6.3) : le *bestiaire* qui comprend les créatures se trouvant dans la pièce actuelle, l'*inventaire* qui comprend les objets détenus par le joueur, le

sol qui comprend les objets qui sont sur le sol de la pièce actuelle, la *navigation* qui comprend des boutons qui permettent au joueur de voyager vers une autre pièce du donjon, un panneau *action* qui permet d'exécuter des actions telles que de parler aux autres joueurs et d'enregistrer la partie courante et finalement un *historique de messages* qui liste les évènements qui surviennent dans la partie.

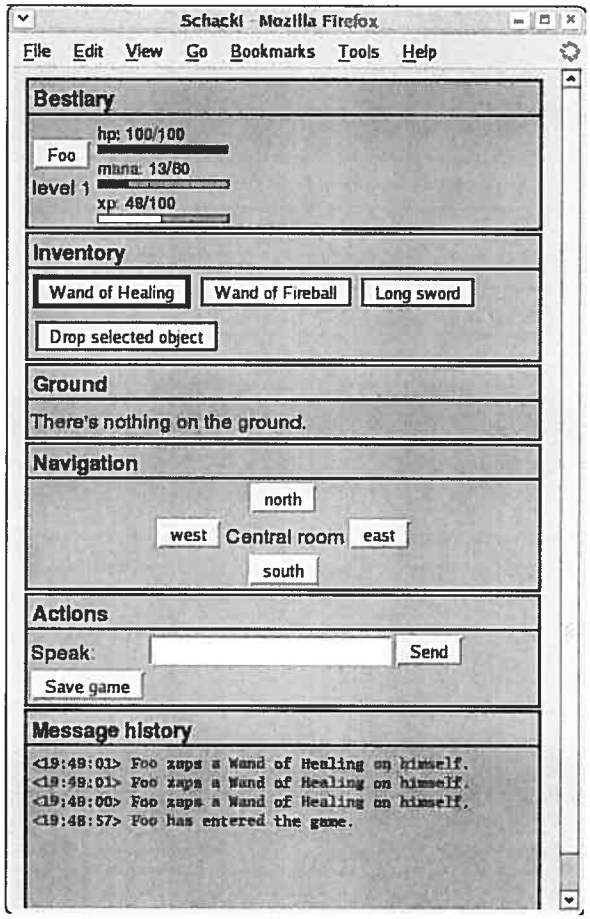

FIG. 6.3 – Le jeu Schack en action

Le panneau *action* ne change pas lors de la partie, mais les quatre autre panneaux changent au fur et à la mesure que l'état des éléments de jeu qu'ils représentent change. Afin que l'affichage de ces panneaux puisse être rafraîchi chaque fois qu'il y

a un changement de l'élément correspondant, ceux-ci connaissent la liste de toutes les composantes d'interfaces qui les représentent.

Prenons pour exemple le cas où un joueur envoie un message aux autres joueurs dans une salle. Ceci doit entraîner la mise à jour de la valeur des boîtes de texte du client de chaque joueur qui se trouve dans la pièce. Il se produit alors les choses suivantes :

1. Le joueur entre un message et clique le bouton *envoyer*.
2. Une requête est faite à partir du client qui contient l'événement indiquant que ce bouton a été cliqué et le contenu de la boîte de message à ce moment.
3. La requête est passée au processus qui gère la partie du joueur sur le serveur à l'aide de l'URL enregistré pour cette session.
4. Le code associé à la diffusion d'un message est invoqué : celui-ci récupère l'information de la pièce courante du joueur.
5. Le processus responsable de la pièce possède une référence sur chaque processus qui gère la boîte de texte d'une interface et fait suivre le message à ceux-ci.
6. Puisque la valeur de la composante change, un message est envoyé à l'interface afin de l'informer de la nécessiter de rafraîchir l'information.
7. Le processus de gestion d'interface fait une demande au processus qui gère la boîte de texte afin que celui-ci génère sa représentation HTML.

8. La nouvelle représentation de la composante est renvoyée au client, qui la met à jour.

On voit qu'un patron assez complexe de traitement concurrent est en cours, mais celui-ci s'implante directement à l'aide des abstractions de concurrence de Termite telles que les serveurs génériques.

On a pu voir dans ce chapitre les applications du langage Termite lors d'expérimentation avec des concepts de programmation distribuée, comme lors de la création d'abstractions comme des serveurs génériques, de l'utilisation des macros pour faire de la concurrence déclarative ou de l'utilisation des continuations pour manipuler le code d'un processus. Le langage se prête également bien à des applications pratiques comme la programmation d'applications web dynamiques. Ces exemples démontrent que le système est intéressant et atteint ses buts en tant qu'outil d'expérimentation et de développement d'applications distribuées.

CHAPITRE 7

IMPLANTATION DU SYSTÈME TERMITE

Une implantation d'un système Termite a été réalisée. Elle est construite au-dessus du système Gambit-C 4 [15]. Le système Gambit-C est particulièrement bien adapté à cette tâche puisqu'il présente des caractéristiques essentielles à l'implantation de Termite, soit des processus légers et la sérialisation d'une grande partie des objets du système.

Il a quand même fallu créer un certain nombre de fonctionnalités, et certaines de celles-ci ont, lors du parcours, été intégrées à Gambit-C. Ces intégrations ont contribué à simplifier l'implantation de systèmes similaires à Termite et a permis d'améliorer la performance du prototype.

Nous discutons dans ce chapitre des détails importants de l'implantation du système Termite, en discutant des différentes approches essayées et adoptées. Ensuite nous abordons certaines particularités de Gambit-C qui ont causé des problèmes affectant potentiellement d'une manière négative la qualité finale du système.

7.1 Détails des parties importantes

Dans cette section nous discutons de l'implantation des parties essentielles du système : le système de processus, le transport des messages et leur stockage, la récupération de message, les ports, la gestion des condition d'erreurs et la propa-

gation de celles-ci.

7.1.1 Système de processus

Le système de gestion des processus inclut la création des processus et la gestion de leurs identités. Tel que mentionné précédemment, les processus Termite sont en fait des processus tels qu'implantés par Gambit-C. Les processus de Gambit-C sont au niveau du langage (c'est à dire qu'ils ne sont pas des processus du système d'exploitation). Ceux-ci sont donc moins coûteux à créer. Le système Gambit-C peut créer et exécuter des millions de processus concurremment. Le système de processus respecte les spécifications SRFI-18 [14] et SRFI-21 [24].

Les références à des processus, c'est-à-dire les *pid*s, sont gérées comme suit : une structure contient un pointeur vers un objet qui peut être soit le processus (quand le processus est local), soit une référence à un enregistrement décrivant l'identité d'un processus distant.

Les références aux processus locaux sont simples à gérer. Il devient plus compliqué de gérer les références aux processus distants. L'enregistrement qui décrit l'adresse du processus distant contient un identificateur universellement unique pour ce processus ainsi que le noeud sur lequel s'exécute ce processus. Un processus ne se voit assigner un identificateur unique que lorsqu'une référence vers celui-ci est exportée à l'extérieur de son noeud de création. Un mutex global au noeud contrôle l'assignation d'une identité aux processus afin d'éviter les incohérences

possibles lors de conditions de course.

7.1.2 Le transport des messages

Lors de l'envoi de message, deux cas sont possibles : soit le *pid* donné en argument à l'opérateur d'envoi de message est une référence à un processus local, soit c'est une référence à un enregistrement décrivant l'adresse d'un processus distant. Le cas de l'envoi à un processus local est simple : il s'agit de suivre une référence locale et d'ajouter le message à la boîte aux lettres du processus.

Pour l'envoi d'un message à un processus distant le message est envoyé à un processus nommé *répartiteur*. Celui-ci inspecte l'enregistrement du processus distant et détermine le noeud vers lequel le message doit être envoyé. Le répartiteur conserve une table où sont inscrits les noeuds distants connus avec un *messager* associé.

Si aucun messager n'est associé présentement au noeud distant, un nouveau messager est créé avec comme tâche d'établir et de maintenir une communication avec le noeud distant. Par la suite, lorsque le répartiteur recevra une communication qui est destinée à un noeud pour lequel un messager existe, le message est relayé au messager directement au lieu d'avoir à établir une nouvelle connexion.

Un noeud écoute sur un port TCP afin d'être en mesure d'accepter des requêtes d'établissement de connexion à partir d'un processus distant. Lors d'une demande de connexion un nouveau messager est démarré afin de servir de relais et ce messager

est enregistré auprès du répartiteur local. Ceci est un système de cache de connexion TCP, ce qui évite d'avoir à ouvrir une nouvelle connexion chaque fois qu'un échange de message survient avec un noeud distant.

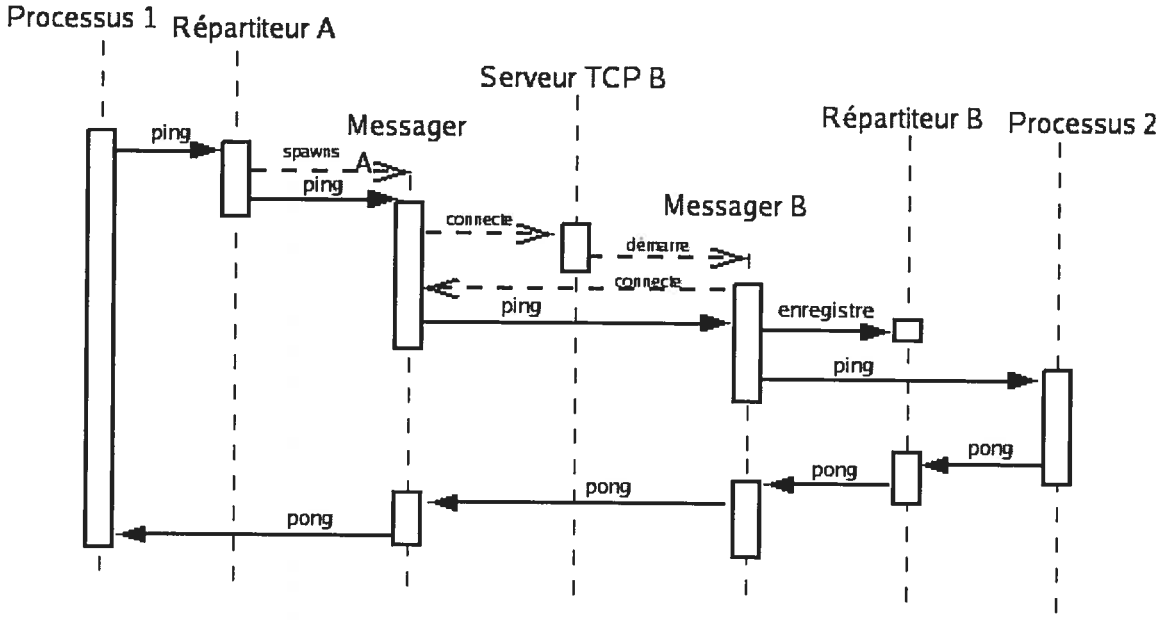

FIG. 7.1 – Diagramme d'interaction de noeuds

Les processus participants à la communication avec un autre noeud, soit le *messager* et les deux processus (entrant et sortant) qui gèrent la connexion TCP, sont liés entre eux afin de permettre la réclamation des ressources à la fin de la connexion. Une erreur de connexion ou la lecture de l'objet de fin de fichier entraîne une exception dans le processus d'entrée qui cause la fin de son exécution et la fin de l'exécution du processus de sortie. L'exception communiquée au *messager* lui fait invoquer son gestionnaire d'exception. Le gestionnaire signale au *répartiteur* la fin de son exécution et ce dernier retire le *messager* de sa table d'association.

7.1.3 Boîtes aux lettres

Chaque processus possède sa propre boîte aux lettres, dans laquelle sont emmagasiné les messages qui lui sont envoyés. Cette boîte aux lettres peut contenir un nombre quelconque de messages. Les messages doivent être accessibles en ordre chronologique, c'est-à-dire que le plus vieux message doit être le premier à être consulté lorsque le processus veut récupérer un message dans sa boîte aux lettres. Il doit également être possible pour le processus de parcourir séquentiellement sa boîte aux lettres afin de rechercher un message qui satisfait des conditions de récupération (dans le cas où la procédure de récupération est celle qui spécifie un prédicat à respecter pour le message ou lorsque le filtrage est utilisé pour choisir un message).

Une version préliminaire de Termite utilisait une file où les messages sont simplement chaînés. Une sonde est attachée à la file afin de pouvoir garder un pointeur vers l'endroit où le processus est rendu à parcourir dans sa file. La sonde est réinitialisée à la tête de la file lorsqu'un message est récupéré. Les nouveaux messages reçus sont ajoutés à la fin de la file.

Les opérations de récupération de message offrent la possibilité de spécifier un délai maximal pour la récupération de message. Quand aucun message satisfaisant n'est disponible, il doit être possible de se mettre en attente pour un certain temps. Si cela est souhaité, le processus suspendra son exécution sur une variable de condition attachée à la boîte aux lettres. L'arrivée d'un message envoie un signal

au processus suspendu afin que celui-ci examine le message qui vient d'arriver. Si aucun message n'arrive dans le délai proscrit, le processus peut faire deux choses ; si une valeur a été spécifiée comme étant celle à retourner si le délai maximal a été atteint, elle est retournée. Dans le cas où aucune valeur n'a été spécifiée à retourner dans cette situation, une exception est levée, signalant que le délai a été atteint.

Une ancienne version de Termite implantait les boîtes aux lettres en utilisant strictement les standards *R5RS* et *SRFI-18*, mais la performance n'était pas optimale. La révision *beta 15* de *Gambit-C 4* intègre désormais un mécanisme de boîte de courrier à chaque processus *SRFI-18*, qui est désormais exploité par l'implantation de Termite.

7.1.4 Filtrage

La forme `recv` de Termite permet la récupération sélective de messages en utilisant le filtrage. Le filtrage est implanté à l'aide de macros. Ceci permet la création d'un "sous langage" spécifique au domaine d'expression de filtres.

Les filtres sont implantés à l'aide du système de macro `define-macro`. Ce genre de macro permet l'exécution de code Scheme au moment de la compilation. En effet, le langage Scheme au complet peut être utilisé afin de faire la génération de code au moment de la compilation. Dans le cas de l'implantation du filtrage par-dessus Gambit, ceci permet la compilation de filtres. Cette compilation permet d'optimiser le code généré pour implanter le filtrage.

La compilation de filtres utilise quelques techniques d'optimisation, la technique de base utilisée est l'élimination de tests redondants, telle que décrite dans [25]. Pour ce faire, un arbre de décision est créé. Les branches de l'arbre représentent chaque choix possible lors de la réalisation des tests de filtrage. Les tests ayant été effectués sont donc connus lors du parcours de l'arbre et on peut ainsi éviter de faire des tests redondants. Par contre, cette approche peut dans certains cas créer un volume de code important.

Un certain nombre de branches mènent au succès de la même clause. Si le code associé au succès de chaque clause n'est pas trivial (comme une expression littérale, par exemple) et que le code de la clause est utilisé un certain nombre de fois, alors le code généré pour ces clauses ne l'est qu'une seule fois sous forme d'une lambda expression, et celle-ci sera référée lorsque nécessaire.

7.1.5 Les services

Plusieurs *services* sont offerts sur un noeud Termite. Ces services font partie intégrante du système, mais ne sont pas implantés au "coeur" du système mais en utilisant les fonctionnalités de base de Termite (les processus, l'envoi de messages, etc.).

Trois services sont essentiels au bon fonctionnement du système. Le premier service gère l'écoute pour les connexions TCP externes. Celui-ci permet aux autres noeud d'établir une connexion avec le noeud courant. Un deuxième service gère l'en-

voi de message aux processus distants (c'est le répartiteur mentionné précédemment). Finalement, un troisième service est responsable de la gestion de l'établissement de liens d'un processus local avec un processus distant.

Une autre catégorie de services offrent des fonctionnalités pratiques mais non-essentielles. Parmi ces services on retrouve un mécanisme de *dictionnaire local* qui sert à associer un processus à un nom sur le noeud courant. Un autre processus répond aux demandes de démarrage de processus venant des autres noeuds.

7.1.6 Les ports

Afin de pouvoir effectuer des entrées/sorties à l'aide des ports du système Gambit, il a fallu abstraire les ports derrière les processus. Cela permet de rendre valide les références aux ports lors de la sérialisation.

Les processus qui servent à abstraire les ports de sorties sont implantés d'une manière simple. Ils attendent des messages composés d'une fonction à un argument. Cette fonction est appelée avec le port Gambit en argument. Par exemple, un message `(lambda (port) (display "hello" port))` reçu par le processus entraînerait l'affichage de la chaîne de caractères `"hello"` sur la sortie désignée.

Alors que les ports de sortie sont simplement en attente de message pour agir, les ports d'entrée peuvent fonctionner de deux manières différente. La première manière, qu'on dit passive, est similaire à la technique utilisée pour les ports de sortie. On attend un message contenant une fonction pour effectuer la lecture sur

le port ainsi que les informations nécessaires pour répondre à la requête. L'autre manière d'implanter les ports de sortie est "active" : certains processus intéressés par l'arrivée de données sur ce port peuvent "s'abonner" à la réception des messages sur ce port. À ce moment, lorsque quelque chose devient disponible en lecture, les processus intéressés reçoivent ces informations.

Les fonctions qui manipulent les ports (`read`, `write`, etc.) sont donc remplacées dans l'implantation actuelle de Termite et les ports par défaut sont abstraits par des processus.

7.1.7 Gestions des erreurs

La gestion des exceptions et des erreurs est un aspect important de la plate-forme. Celle-ci reste tout de même plutôt simple à implanter. Deux fonctions de base servent à la gestion des exceptions : la fonction `with-exception-handler`, et la fonction `with-exception-catcher`. La différence entre les deux repose sur la continuation qui est invoquée lors du retour du code qui gère l'exception : avec `with-exception-catcher` la continuation est celle de l'appel à cette fonction, alors que pour `with-exception-handler` la continuation est celle du code qui a généré une exception. Signaler une exception se fait avec la procédure `raise`. À la base, le système d'exceptions implanté dans Termite est le même que le système d'exception de Gambit-C. La différence repose dans le gestionnaire d'exception de base installé pour chaque processus. Alors que le gestionnaire d'exception de base par défaut de

Gambit-C va simplement interrompre le traitement et possiblement le signaler à l'utilisateur avec un message sur la console, dans Termite le gestionnaire d'exception de base va communiquer cette exception à tous les processus qui sont *liés* avec le processus courant.

Une exception peut donc être levée explicitement ou être reçue d'un autre processus. Une exception reçue d'un autre processus est levée dès la prochaine tentative de récupération de message. Cela signifie que les exceptions sont traitées de manière synchrones. Ceci diffère avec Erlang, où les exceptions sont traitées de manière asynchrone. Signaler les exceptions de manière synchrone fait qu'il est facilement possible d'identifier les sites dans le code où l'exécution peut être interrompue : il n'y a que les opérations de récupération de message qui doivent être tenues en compte. Dans le cas du signalement d'exceptions asynchrones, le code peut être interrompu à tout moment. Ceci peut potentiellement compliquer la programmation de *zones critiques*, par exemple lorsque deux messages doivent être envoyés afin d'accomplir une certaine tâche, on ne veut peut-être pas que le deuxième envoi puisse échouer à cause d'une erreur dans un autre processus. D'un autre côté, la propagation asynchrone permet d'interrompre un processus, même si celui-ci ne fait pas d'opération de réception de message.

7.1.8 Gestion des liens entre les processus

Les exceptions se propagent d'un processus à un autre lorsque les processus sont liés. Les liens entre les processus sont directionnels. Chaque processus comprend dans sa structure le décrivant une liste de processus auxquels il est lié. Le gestionnaire d'exception de base de chaque processus va envoyer un message à chacun de ces processus s'il est invoqué.

Lors de l'établissement d'un lien d'un processus distant vers un processus local, il est nécessaire de communiquer la requête de lien. Celle-ci se fait en communiquant avec un service spécial (le *lieur*) sur le noeud distant. Ce lieur s'occupe d'enregistrer le lien dans la structure du processus concerné.

7.2 Particularités du système Gambit-C 4

Le système Gambit-C a été conçu à la base de manière à bien interagir avec C. Bien que la programmation d'applications distribuées n'a jamais fait parti des usages du système, celui-ci s'est quand même bien adapté à ce genre d'utilisation. Les deux fonctionnalités particulière à Gambit-C ayant facilité l'implantation du système sont les processus légers et la sérialisation. Les fonctionnalités de sérialisation par contre n'étaient pas optimales et certains problèmes ont dus être résolu en cours de développement, alors que d'autres restent.

Afin de pouvoir communiquer des données entre des processus s'exécutant sur différents ordinateurs, il est nécessaire de transformer les données sous une repré-

sentation qui peut voyager. Cette transformation, la sérialisation, doit pouvoir représenter les objets d'une manière à ce qu'ils puissent être reconstitués et conserver les même caractéristiques.

Le système Gambit-C est composé d'un interprète et d'un compilateur. Les deux utilisent une représentation interne différente du code, ce qui entraîne certains effets au niveau de la sérialisation des fermetures et des continuations.

L'interprète conserve l'ensemble de l'environnement dans la représentation des fermetures à l'exécution. Il n'y a pas d'analyse afin de déterminer quelles variables ne sont pas capturées par une fermeture. Ceci est délibéré, car la présence de ces informations facilite le déboguage. Le code en mémoire est représenté comme un graphe de fermetures qui s'appellent entre elles. Les fermetures contiennent un pointeur vers le contexte lexical englobant. Ceci fait que même une simple fermeture ne capturant pas de variable de l'environnement lexical peut quand même faire référence en mémoire à un grand ensemble de données. Un autre aspect est que l'environnement global n'est pas capturé dans la continuation. Cette représentation permet de transporter le code d'un ordinateur à un autre : celui-ci est un objet mémoire qui peut être sérialisé. Par contre, la grosseur de la structure générée peut possiblement être néfaste à la performance du système.

Le compilateur fait une analyse afin de ne retenir dans l'environnement que ce qui est nécessaire à l'exécution du code. Cela veut dire que les fermetures sont représentées d'une manière plus compacte. Le code, par contre, n'est pas représenté

comme un graphe de fermetures mais sous forme de code machine compilé. La représentation des fermetures comprend donc un pointeur vers une zone mémoire qui contient le code à exécuter lors de l'application de cette fermeture. Cela signifie qu'il faut s'assurer que le même code ait été chargé lorsque l'on veut migrer des fermetures de code compilé entre deux noeuds. Cela rend moins pratique l'utilisation de code compilé mais offre une performance supérieure que lorsque le code est interprété.

L'approche préférable serait d'avoir un format de code portable mais compilé et optimisé, tel que du code octet pour une machine virtuelle. Ceci permettrait d'avoir un code plus compact et n'incluant que les informations nécessaires à son exécution. D'autres techniques pourraient être utilisées, comme l'utilisation de *cache* de code entre les noeuds.

L'implantation du système Termite par dessus un système Scheme, en l'occurrence Gambit, s'est révélé être une tâche relativement simple. La plate-forme possédait déjà quelques caractéristiques cruciales qui, avec quelques ajustement, répondent particulièrement bien aux besoins de la programmation d'un système pour applications distribuées. Le langage Scheme se prête bien à l'expérimentation, et sa flexibilité permet d'éviter des obstacles qui se seraient dressés dans le cas d'autres langages. Quelques points seraient à améliorer, comme la facilité générale d'utilisation du système et les outils périphériques, mais le résultat actuel est suffisant pour expérimenter avec le langage et développer des applications distribuées.

CHAPITRE 8

MESURES DE PERFORMANCE

Afin d'évaluer la performance du système Termite, quelques mesures de performance on été effectuées en utilisant la version 0.8 de Termite. Une partie des tests de performance réalisés ici sont les mêmes que ceux réalisé dans [26]. Lorsque cela était possible, des programmes Erlang équivalents ont été mesurés en utilisant Erlang/OTP version 5.4.8 afin de comparer les performances relatives des deux systèmes. De plus, certain tests de performance ont également été écrits directement dans Gambit Scheme et exécutés avec la version 4.0 beta 17 afin d'évaluer le surplus de travail entraîné par l'implantation. Plusieurs des tests avec Termite ont étés réalisés dans deux contextes différents. Dans le premier contexte, Termite est compilé avec le programme. Dans le deuxième contexte, Termite est compilé au coeur de Gambit (soit le mode normal d'utilisation), afin d'essayer d'éliminer les appels inter-modules lors de l'utilisation des fonctions d'envoi et de réception de message. Le compilateur GCC version 4.0.2 a été utilisé pour compiler Gambit, et l'option de configuration "–enable-single-host" a été fournie lors de la compilation. Tous les tests ont été exécutés sur des ordinateurs basés sur le microprocesseur Athlon 64 d'AMD cadencé à 2.2 GHz avec 2Go de mémoire vive reliés par un réseau Éthernet à 100Mb/s roulant le système d'exploitation GNU/Linux version 2.6.10.

8.1 Tests de performance de base

Des tests de performance simples ont été réalisés afin d'estimer la performance générale des systèmes sur du code ne faisant pas appel aux fonctionnalités de programmation concurrente et distribuée. Les tests ne correspondent pas nécessairement aux programmes typiquement écrits dans ces systèmes, mais permettent tout de même d'évaluer la performance de mécanismes comme les appels de fonction ou l'allocation mémoire.

Les tests suivants sont utilisés :

- la fonction de Fibonacci (`fib`) ainsi que la fonction de Takeuchi (`tak`), pour estimer la performance des appels de fonctions ;
- l'inversion de l'ordre des éléments d'une liste de manière naïve (`nrev`), afin d'exercer l'allocation et la réclamation de mémoire,
- le tri avec l'algorithme *quicksort* (`qsort`) d'une liste de nombres aléatoires ;
- lne implantation de l'algorithme Smith Waterman (`smith`) pour la recherche de correspondances entre des chaînes de caractères.

Les résultats de ces tests sont donnés dans la figure 8.1. Ceux-ci démontrent que le système Gambit est en général plus rapide pour du code qui fait beaucoup d'appels de fonctions (*fib* et *tab*), mais est un peu plus lent pour des tests qui font beaucoup d'allocation mémoire (*nrev* et *smith*). En général, la performance des deux systèmes est assez similaire.

Test de performance	Erlang temps (s)	Termite temps (s)
fib (34)	1.671	0.493
tak (27, 18, 9)	0.779	0.467
nrev (5000)	0.297	0.425
qsort (250000)	1.28	0.91
smith (600)	0.468	0.473

FIG. 8.1 – Tests de performance de base.

8.2 Tests de performance pour les primitives de concurrence

Les tests de cette section ont pour objectif de mesurer la performance relative entre Gambit, Termite et Erlang des opérations primitives de concurrence, soit la création de processus ainsi que l'envoi/réception de message.

Les deux premiers tests effectués sont simples. Le premier (*spawn*) effectue une boucle qui crée un grand nombre de processus qui ne font aucun travail réel, et le temps de création par processus est donné. Dans le deuxième test (*self*), un processus s'envoie un message à lui-même puis le récupère, et ce à répétition. Le temps nécessaire à l'opération d'envoi/récupération d'un message est donné. Les résultats sont donnée à la figure 8.2. On remarque que la performance de Gambit et Termite est inférieure à celle de Erlang pour ces tests, mais que la différence n'est pas majeure.

8.2.1 Anneau de processus

Ce test de performance crée un anneau de 250,000 processus sur un noeud unique. Ensuite, un nombre est envoyé sur l'anneau, et celui-ci est décrémenté par

Test de performance	Erlang temps (μs)	Gambit temps (μs)	Termite [a] temps (μs)	Termite [b] temps (μs)
Spawn	1.47	3.19	4.16	3.95
Self	0.069	0.221	0.349	0.289

[a]Compilé avec le programme
[b]Compilé avec Gambit

FIG. 8.2 – Tests de performance pour les primitives de concurrence.

chaque processus successivement. Lorsque le nombre passé autour de l'anneau est 0, le processus relaie le 0 au processus suivant et arrête ensuite son exécution.

Ce test est exécutée deux fois avec un nombre initial différent. Le premier coup, un nombre initial de 0 est envoyé. Le test mesure alors principalement le temps de création de l'anneau ainsi que le temps nécessaire à la destruction de celui-ci. Ensuite, le test est effectué avec un nombre initial de $1,000,000$, le temps pris pour le passage de message devient alors plus significatif. Les résultats de ce test sont donnés à la figure 8.3. La performance est donnée en nombre de millièmes de secondes nécessaire à l'exécution, plus la valeur est basse, meilleure est la performance.

K [c]	Erlang temps (ms)	Gambit temps (ms)	Termite [a] temps (ms)	Termite [b] temps (ms)
0	913	1452	1577	1556
1000000	1593	3780	4347	4271

[a]Compilé avec le programme
[b]Compilé avec Gambit
[c]Nombre initial passé à l'anneau

FIG. 8.3 – Test de performance : anneau de 250,000 processus

On remarque que la création de processus dans Gambit (et donc dans Termite)

est moins rapide que celle de Erlang. L'envoi de messages est plus performant dans le cas de Erlang. Pour ce qui est de Termite, la pénalité supplémentaire du système par rapport à Gambit reste plutôt minime. On peut inclure une pénalité lors de la création de processus afin d'initialiser les données supplémentaires à chaque processus (comme la liste de liens), et il y a une pénalité également au moment de l'envoi d'un message car le message doit être examiné pour déterminer si celui-ci est vraiment destiné à un processus local ou s'il est destiné à un processus s'exécutant sur un autre noeud.

8.3 Tests de performance d'application distribuées

8.3.1 Échanges de "Ping-Pong"

Ce test de performance mesure le temps nécessaire à l'envoi d'un message dans le cadre d'échanges de *ping-pong* fait entre deux processus, sous trois conditions différentes : lorsque les processus s'exécutent sur le même noeud (figure 8.4), lorsque les processus s'exécutent sur deux noeuds différents mais sur le même ordinateur (figure 8.5), et finalement lorsque deux processus s'exécutent sur deux noeuds différents, étant chacun sur un ordinateur différent (figure 8.6). Dans chaque situation, nous varions le volume de la donnée envoyée comme message entre les deux processus, en utilisant une liste d'un certain nombre d'éléments. La performance est donnée en temps nécessaire pour l'envoi du message entre deux processus en microsecondes. Plus la valeur est basse, meilleure est la performance.

Longueur de la liste	Erlang temps (μs)	Gambit temps (μs)	Termite [a] temps (μs)	Termite [b] temps (μs)
10	0.73	1.25	1.55	1.42
20	0.96	1.25	1.53	1.42
50	1.60	1.25	1.54	1.43
100	2.46	1.25	1.55	1.43
200	4.05	1.29	1.58	1.51
500	9.21	1.26	1.55	1.45
1000	17.71	1.26	1.55	1.44

[a]Compilé avec le programme
[b]Compilé avec Gambit

FIG. 8.4 – Mesure du temps nécessaire à l'envoi d'un message de longueur variable entre deux processus s'exécutant sur le même noeud.

Le test de performance de *ping-pong local* illustre un point intéressant : lorsque le volume des messages augmente, la performance du système Erlang diminue grandement alors que celle du système Termite reste pratiquement la même. Ceci est dû au fait que le système Erlang utilise un *tas* (zone d'allocation mémoire) par processus, alors que le système Gambit, sur lequel Termite repose, utilise une approche avec un tas partagé par tous les processus. Puisque les objets ne sont pas mutables dans le système Termite, l'approche partagée est tout aussi valable. Les avantages de cette approche par rapport à celle utilisée par Erlang sont décrits dans [27].

On remarque que la performance de Termite par rapport à Erlang est significativement moindre dans les cas distribués. Alors que la conception d'Erlang a voulu que la sérialisation des objets soit une opération extrêmement efficace, la sérialisation dans Gambit est une fonctionnalité relativement nouvelle. Il est possible que celle-ci puisse être améliorée ou que l'implantation de Termite n'exploite pas

Longueur de la liste	Erlang temps (μs)	Termite [a] temps (μs)	Termite [b] temps (μs)
10	69.75	379.08	392.16
20	70.93	381.83	391.08
50	71.94	383.29	392.46
100	73.00	384.17	393.55
200	77.03	387.60	396.04
500	177.75	390.02	398.88
1000	259.47	389.56	399.68

[a]Compilé avec le programme
[b]Compilé avec Gambit

FIG. 8.5 – Mesure du temps nécessaire à l'envoi d'un message de longueur variable entre deux processus s'exécutant sur deux noeuds différents sur le même ordinateur.

celle-ci de manière efficace. Une investigation plus profonde serait de mise afin de déterminer comment il serait possible d'améliorer la performance de Termite dans un contexte distribué.

8.3.2 Migration de processus

Ce test de performance a été exécuté avec Termite uniquement, puisqu'il mesure le temps nécessaire à la migration d'un processus entre deux noeuds et que cette fonctionnalités n'est pas supportée par Erlang. Le test a été effectué sous trois circonstances : lorsque le processus "migre" à l'intérieur d'un même noeud, lorsque les deux noeuds s'exécutent sur le même ordinateur, et lorsque les deux noeuds s'exécutent sur deux ordinateurs différents et communiquent au travers du réseau. Les résultats sont donnés à la figure 8.7. La mesure de performance est donnée en nombre de secondes nécessaire à une migration. Plus la valeur est basse, meilleure

Longueur de la liste	Erlang temps (μs)	Termite [a] temps (μs)	Termite [b] temps (μs)
10	129.10	820.34	838.22
20	137.36	821.69	839.63
50	150.08	821.69	838.22
100	177.68	817.66	836.12
200	211.95	821.69	838.22
500	911.58	818.33	836.82
1000	1479.29	817.66	837.52

[a]Compilé avec le programme
[b]Compilé avec Gambit

FIG. 8.6 – Mesure du temps nécessaire à l'envoi d'un message de longueur variable entre deux processus s'exécutant sur deux noeuds différents sur deux ordinateurs reliés par un réseau.

est la performance.

Test de performance	Termite [a] temps (ms)	Termite [b] temps (ms)
Migration à l'intérieur d'un noeud	0.003	0.003
Migration entre deux noeuds locaux	2.16	2.19
Migration entre deux noeuds par le réseau	1.79	1.81

[a]Compilé avec le programme
[b]Compilé avec Gambit

FIG. 8.7 – Test de performance : mesure du temps nécessaire à la migration d'un processus.

Ce test démontre que le coût majeur d'une migration est dans la sérialisation d'une continuation ainsi que dans la transmission de celle-ci. Comparativement, capturer une continuation sur le noeud courant et démarrer un nouveau processus avec celle-ci est très peu coûteux.

CHAPITRE 9

TRAVAUX RELIÉS

9.1 Technologies de programmation d'applications distribuées existantes

Plusieurs technologies de programmation d'applications distribuées sont déjà existantes. Celles-ci varient en fonction de leurs objectifs (pour quel type d'application elles ont été conçues) et de la philosophie générale derrière le système et le langage d'implantation.

Nous séparons en deux catégories les technologies servant à créer des applications distribuées : la première catégorie porte sur les technologies dites "d'appel de procédure distantes". Les technologies de cette catégorie ne sont pas nécessairement attachées à des langages particuliers. La deuxième catégorie traite des technologies reliées à des langages, tels que des langages de programmation conçus expressément pour la programmation distribuée.

9.1.1 Appel de procédures distantes

Les technologies d'appel de procédure distantes sont souvent décrites par l'acronyme RPC ("Remote Procedure Call"). L'idée générale est d'avoir un protocole qui permette à des logiciels d'exécuter des appels de procédure sur un autre ordinateur.

Deux caractéristiques générales de ce genre de protocole sont d'avoir une manière

d'adresser des requêtes (un système d'adresse pour les services) et une forme standard de représentation des données pour que celles-ci puissent être transportées au travers du réseau et manipulées par les systèmes participant à la communication. Des exemples de cette manière de faire sont CORBA, DCOM, RMI/Java et les "services Web".

9.1.1.1 CORBA

CORBA (*Common Object Request Broker Architecture*) [28] est un standard du consortium OMG (*Object Management Group*), qui est formé de grandes entreprises du marché de développement de logiciels.

Le standard spécifie une manière de représenter les objets sous une forme binaire pouvant être partagé par plusieurs ordinateurs. Un langage de définition d'interface nommé *IDL* est utilisé afin de spécifier les capacités de l'objet, c'est à dire ses méthodes et comment les appeler. CORBA permet à un programme d'exporter cette interface afin de rendre possible l'appel de procédure par un autre programme à distance. CORBA permet également de spécifier un cadre pour la gestion de transactions et des droits d'accès (sécurité).

Alors que CORBA tente de faire interopérer des applications écrites dans des langages différents au travers d'une interface standard, Termite est spécifique à un langage particulier. CORBA permet de définir des droits d'accès aux ressources mais aucune construction du genre n'existe pour Termite, bien qu'il soit pos-

sible pour le programmeur de développer cette fonctionnalité sur mesure pour son programme. Il serait possible de construire une interface CORBA pour Termite. D'ailleurs une implantation de référence existe déjà pour Lisp [29].

9.1.1.2 DCOM

DCOM (*Distributed Component Object Model*) [30] est une technologie développée par Microsoft qui sert à permettre à des composants indépendants de communiquer entre eux, potentiellement entre différents ordinateurs. Un glaneur de cellule distribué est implanté pour le système. La technologie n'est disponible que pour le système d'exploitation Windows, mais est généralement indépendante du langage de programmation utilisé pour programmer une composante. C'est en quelque sorte la version Microsoft de CORBA. Tout comme pour CORBA, il serait possible de faire interopérer Termite avec DCOM.

9.1.1.3 RMI/Java

Java [31] [32] est un langage de programmation orienté objet développé par la compagnie Sun qui a pour objectif de permettre la portabilité de code entre plusieurs plate-forme afin que celles-ci puissent s'échanger du code et communiquer plus facilement entre elles. La technologie repose sur une machine virtuelle qui est porté sur chaque architecture et système d'exploitation où l'on désire exécuter des programmes Java.

Afin de permettre les programmes d'invoquer des méthodes sur des objets distants, la technologie RMI [33] a été développée. Celle-ci permet de rendre transparente la position d'objets et de faire de la programmation distribuée. La technologie RMI est liée au langage Java. Microsoft a reproduit une technologie semblable à Java et RMI avec sa plate-forme *.NET* [34]. *.NET* est également le successeur des technologies COM.

Tout comme pour Termite, la technologie RMI est liée à un langage de programmation. Dans RMI, les interfaces entre les objets qui communiquent doivent être statiques (elles sont fixées au moment de la compilation). Ceci est dû à l'abstraction utilisée, soit l'appel de méthode à distance, à la nature statique du langage Java ainsi qu'aux choix réalisés lors du design du protocole RMI. Termite permet d'envoyer un message avec n'importe quel contenu à un autre processus connu, mais il n'est pas possible d'avoir le même genre de vérification statique obtenu par Java au moment de la compilation pour déterminer si le genre de message envoyé sera compris. Le compromis entre les deux design est analogue à celui fait entre les langages typés dynamiquement et les langages typés statiquement.

9.1.1.4 Services Web

Les services Web sont une technologie dont on parle depuis longtemps mais dont on ne commence qu'à voir les applications intéressantes. Les services web sont en fait des procédures qu'il est possible d'appeler avec un protocole comme XML-

RPC [35] ou SOAP [36]. Généralement, ceux-ci sont accessible par le protocole HTTP [37]. L'information transmise entre le client utilisateur du service web et le serveur est sérialisée sous forme de XML [38].

L'espoir de cette technologie est qu'en se servant de standards bien établi, il sera facile de faire interopérer différents programmes et de les rendre facilement accessible par l'Internet. Des compagnies comme Amazon, Yahoo et Google offrent des accès à leurs services par l'entremise de services Web, permettant le développement de nouvelles applications utilisant leur service.

Termite est en position d'offrir des fonctionnalités semblable. Un noeud Termite peut être accessible sur l'Internet afin d'offrir un service (par exemple l'accès à une base de donnée) à des programmes externes. La représentation des données dans Termite est faite sous une forme *opaque*, qui n'est pas destinée à pouvoir être interprétée directement par un autre système. Ceci permet d'échanger des données de type beaucoup plus complexe entre des noeuds, et aussi de le faire d'une façon plus efficace qu'en utilisant une représentation binaire au lieu de XML. Il serait possible de construire une interface de type *service web* à un noeud Termite et ainsi de rendre accessible des fonctionnalités du noeud à des outils utilisant ce genre de requête.

9.1.2 Systèmes et langages de programmation distribuée

Les technologies d'appel de procédure distante sont généralement indépendantes du langage de programmation, puisque celles-ci sont des ajouts fait à des langages de programmation existants. Une autre perspective possible pour la création d'applications distribuées est d'utiliser un système conçu spécialement pour ce type d'application. Nous présentons dans cette section une sélection de systèmes distribués qui sont ceux ayant le plus inspiré Termite.

9.1.2.1 Kali Scheme

Kali [39] est une implantation de Scheme supportant la programmation distribuée. L'implantation permet la migration d'objets, y compris les fermetures et les continuations, entre des ordinateurs. Un modèle de mémoire partagé est utilisé, et un GC distribué est nécessaire. Le modèle de distribution est centralisé : un noeud a l'autorité sur l'ensemble des autres noeuds. Par contraste, Termite utilise un modèle où les espaces mémoire de chaque processus sont isolés et propose un système où les noeuds sont chacun indépendants et égaux (modèle *pair-à-pair*). Kali n'offre pas de moyen de gérer les fautes : une faute sur un noeud peut provoquer l'arrêt de l'ensemble du système distribué. L'implantation est intéressante car elle permet la communication efficace entre les noeud en conservant une cache d'objets et en transmettant de manière paresseuse le code des fermetures.

9.1.2.2 Oz/Mozart

Oz/Mozart [40] regroupe un langage (Oz) et un environnement d'exécution (Mozart) qui tente de rassembler le plus grand nombre de paradigmes de programmation dans un système cohérent. Beaucoup de fonctionnalités, abstractions, etc. sont fournies. Le système permet d'exprimer à la fois la concurrence déclarative, la concurrence avec passage de message et la concurrence avec mémoire partagée. Le système supporte également la programmation distribuée et possède un GC distribué.

La démarche et l'approche de Termite et celle de Mozart sont presque opposées. Alors que dans le cadre de Termite nous essayons de simplifier un problème et de donner un petit nombre de fonctionnalités avec lesquelles peuvent être construites de nouvelles abstractions, Oz/Mozart tente de tout offrir dans un même ensemble. L'approche de Termite au point de vue de la conception du langage est plus près de celle utilisée pour le langage Scheme, où l'on tente de minimiser le nombre de particularités.

9.1.2.3 Erlang

Erlang [10] [11] est un langage de programmation qui est dit "orienté concurrence". Le modèle de concurrence de Erlang est essentiellement celui qui a été repris pour Termite, tel qu'expliqué à la section 4.1.1. Le langage séquentiel de Erlang est un langage purement fonctionnel typé dynamiquement.

Les fonctionnalités principales étant ajoutée par Termite à Erlang sont la présence de continuations de première classe et la présence de macros. Aussi, Termite permet d'établir des liens unidirectionnels entre deux processus pour contrôler la propagation d'erreur, alors que Erlang établi toujours des liens bidirectionnels. Une particularité de Erlang qui n'est pas présente dans Termite est la capacité de pouvoir traiter des exceptions propagées depuis un autre processus d'une manière asynchrone. Dans Termite, le traitement d'une exception ne se fait qu'au moment où celle-ci est reçue dans un message.

CHAPITRE 10

CONCLUSION

Ce mémoire a présenté Termite, un système et langage pour la programmation d'applications distribuées. Termite permet d'expérimenter avec la conception d'abstractions spécialisées ainsi que de réaliser la programmation d'applications distribuées. Le système s'inspire de Scheme et de Erlang et repose ainsi sur le principe d'un langage de programmation fonctionnel avec un modèle de concurrence par passage de message.

Nous avons dans un premier temps déterminé les applications cibles de ce genre de système. Les applications nécessitant une interaction entre plusieurs parties distribuées sur internet sont d'un intérêt particulier. Ensuite, nous avons décrit les sources principales d'inspiration de la conception. Le langage Scheme sert de base à la construction du nouveau langage. À ce langage est ajouté le modèle de concurrence par passage de message de Erlang. Puis le système Termite a été décrit par la documentation permettant son utilisation et par une discussion des techniques utilisées pour l'implantation du système, ainsi que la mesure de la performance de celui-ci. Des exemples d'utilisation du système ont ensuite été donnés, comme des abstractions utiles, des techniques de programmation de composantes d'application distribuées robustes et un système de programmation d'applications web.

Le projet qui a mené au design de Termite a été un travail d'exploration et

d'expérimentation. Le résultat final est prometteur : il permet d'exprimer de façon concise des programmes qui sont volumineux et complexes à écrire dans d'autres langages. Lorsque celui-ci a semblé n'offrir qu'un coeur de fonctionnalités trop minimaliste, la présence des continuations et des macros a montré qu'il était possible de faire grandir le langage afin de lui permettre d'être plus puissant. La puissance des langages Lisp est bien reflétée dans cette constatation. L'extension du langage Scheme avec le modèle de concurrence de Erlang permet d'avoir, en quelque sorte, le meilleur de deux monde : un langage de programmation séquentiel extrêmement flexible avec un modèle de concurrence permettant de s'écarter d'une bonne partie des problèmes de la concurrence avec mémoire partagée telle que normalement utilisée dans le contexte de Scheme. Ce modèle de concurrence s'étend naturellement à la programmation d'applications distribuées.

Une autre contribution du projet est l'amélioration résultante du système Gambit-C. Le système a été étendu afin de permettre la sérialisation binaire flexible et efficace des objets et les processus ont maintenant un mécanisme performant de passage de messages. Les tests de performance ont également permis de détecter et de corriger certaines anomalies dans les mécanismes de communications par le réseau.

Le système actuel offre un point de départ intéressant pour des travaux futurs. Plusieurs aspects seraient pertinents à explorer, tels qu'une implantation avec des caractéristiques différentes (plus compacte ou plus performante, par exemple), la

création d'outils évolués pour le développement d'applications et d'interfaces avec une variété de protocoles et de systèmes existants. Termite offre un modèle de programmation bien adapté aux tâches d'exploration et de développement d'applications pour lesquelles il a été développé.

BIBLIOGRAPHIE

[1] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9) :26–76, 1998.

[2] Network Working Group, IETF. Post Office Protocol - Version 3. `http://www.ietf.org/rfc/rfc1939.txt`.

[3] Network Working Group, IETF. Internet Message Access Protocol - Version 4rev1. `http://www.ietf.org/rfc/rfc2060.txt`.

[4] Network Working Group, IETF. Simple Mail Transfer Protocol. `http://www.ietf.org/rfc/rfc2821.txt`.

[5] Network Working Group, IETF. Multipurpose Internet Mail Extensions (MIME). `http://www.ietf.org/rfc/rfc2045.txt`.

[6] Network Working Group, IETF. Internet Relay Chat Protocol. `http://www.ietf.org/rfc/rfc1459.txt`.

[7] Jr. Guy Lewis Steele. Lambda : The ultimate declarative. Technical Report AI Lab Memo AIM-379, MIT AI Lab, November 1976.

[8] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs.* MIT Press, Cambridge, Mass., 1996.

[9] Kent R. Dybvig. *The Scheme Programming Language.* Englewood Cliffs, 1987.

[10] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 1996.

[11] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology, Department of Microelectronics and Information Technology, Stockholm, Sweden, December 2003.

[12] Shriram Krishnamurthy. Automata as macros. *Journal of Functional Programming, to appear*, 2006.

[13] David A. Kranz, Richard A. Kelsey, Jonathan A. Rees, Paul Hudak, and James Philbin. Orbit : an optimizing compiler for scheme. In *Proceedings of the SIGPLAN symposium on Compiler contruction*, pages 219–233. Association of Computer Machinery, June 1986.

[14] Marc Feeley. SRFI 18 : Multithreading support. `http://srfi.schemers.org/srfi-18/srfi-18.html`.

[15] Marc Feeley. Gambit-C version 4. `http://www.iro.umontreal.ca/~gambit`.

[16] Network Working Group, IETF. A Universally Unique IDentifier (UUID) URN Namespace.

`http://www.ietf.org/rfc/rfc4122.txt`.

[17] Guillaume Germain. Sources du système termite. `http://toute.ca/termite`.

[18] Erricsson Computer Science Laboratory. Erlang/otp. http://www.erlang.org.

[19] Robert Ginda. JavaScript page at mozilla.org. http://www.mozilla.org/js/.

[20] W3C DOM Interest Group. Document Object Model (DOM). http://www.w3.org/DOM/.

[21] Jesse James Garrett. Ajax : A New Approach to Web Applications. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[22] The Apache Software Foundation. Apache HTTP Server Project. http://httpd.apache.org/.

[23] Marc Battyani. Mod_lisp Home Page. http://www.fractalconcept.com/asp/2aM/sdataQI7jn05lCw92DM==/asdataQuvY9x3g\$ecX.

[24] Marc Feeley. SRFI 21 : Real-time multithreading support. http://srfi.schemers.org/srfi-21/srfi-21.html.

[25] Christian Quiennec. Compilation of Non-Linear, Second Order Patterns on S-Expressions. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 340–357. Springer, August 1990.

[26] Marc Feeley Guillaume Germain and Stefan Monnier. Termite : a Lisp for Distributed Computing. 2nd European LISP and Scheme Workshop. http://lisp-ecoop05.bknr.net/submission/19654.html, July 2005.

[27] Marc Feeley. A case for the unified heap approach to erlang memory management, August 17 2001.

[28] Object Management Group, Inc. Catalog of OMG CORBA/IIOP Specifications. http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[29] Object Management Group, Inc. Lisp Language Mapping. http://www.omg.org/technology/documents/formal/lisp_language_mapping.htm.

[30] Microsoft Corporation. COM : Component Object Model Technologies. http://www.microsoft.com/com.

[31] Guy Steele James Gosling, Bill Joy and Gilad Bracha. The java language specification, third edition. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[32] Sun Microsystems, Inc. Java Technology. http://java.sun.com.

[33] Sun Microsystems, Inc. Java Remote Method Invocation - Distributed Computing for Java.

http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html.

[34] Microsoft Corporation. Microsoft .NET Homepage. http://www.microsoft.com/net.

[35] Dave Winer. XML-RPC Specification. http://www.xmlrpc.com/spec.

[36] World Wide Web Consortium. SOAP Version 1.2 Part 1 : Messaging Framework. http://www.w3.org/TR/soap12-part1/.

[37] World Wide Web Consortium. HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[38] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

[39] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5) :704–739, 1995.

[40] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

ANNEXE A

CODE DES EXEMPLES

A.1 Compte de banque

A.1.1 Compte de banque en Java

A.1.1.1 Classe Account

```
import java.util.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class Account extends UnicastRemoteObject implements RemoteAccount
{
    int balance;
    LinkedList<LogRecord> log;

    public Account () throws RemoteException
    {
        super ();
        balance = 0;
        log = new LinkedList <LogRecord>();
    }

    public void deposit (int amount)
    {
        balance += amount;
        log.add (new LogRecord ("deposit", amount));
    }

    public int withdraw (int amount)
    {
        if (amount > balance)
            amount = balance;
        balance -= amount;
        log.add (new LogRecord ("withdraw", amount));
```

```
        return amount;
    }

    public int query ()
    {
        log.add (new LogRecord ("query", balance));
        return balance;
    }
}
```

A.1.1.2 Interface RemoteAccount

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteAccount extends Remote {
    public void deposit (int amount) throws RemoteException;
    public int withdraw (int amount) throws RemoteException;
    public int query ()              throws RemoteException;
}
```

A.1.1.3 Classe LogRecord

```
import java.util.*;

class LogRecord
{
    Calendar time;
    String message;
    int amount;

    LogRecord(String message, int Amount)
    {
        this.time = Calendar.getInstance();
        this.message = message;
        this.amount = amount;
    }
```

```
    public String toString()
    {
        return new String ("<time> " + message + " " + amount);
    }

    public static void main (String [] args)
    {
        LogRecord lr = new LogRecord ("foo", 42);
        System.out.println(lr);
    }
}
```

A.1.1.4 Classe AccountServer

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class AccountServer
{
    public static void main (String[] argv)
    {
        System.setSecurityManager (new RMISecurityManager());
        try
            {
                Naming.rebind("AccountInstance", new Account ());
            }
        catch (Exception e)
            {
                System.out.println("AccountServer error: " + e.getMessage());
                e.printStackTrace();
                System.exit(1);
            }
    }
}
```

A.1.1.5 Classe AccountClient

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class AccountClient
{
    public static void main (String[] argv)
    {
        System.setSecurityManager(new RMISecurityManager());

        try
            {
                // bind server object to object in client
                RemoteAccount account =
                    (RemoteAccount)
                    Naming.lookup ("rmi://" + argv[0] + "/AccountInstance");

                // invoke methods on server object
                account.deposit (50);
                account.withdraw (30);
                System.out.println(Integer.toString
                                    (new Integer (account.query())) +
                                    " dollars in the account");
            }
        catch(Exception e)
            {
                System.out.println ("Exception occured");
                e.printStackTrace();
                System.exit (1);
            }
        System.out.println("RMI connection successful");
    }
}
```

A.2 Serveur générique : genserver.scm

```
;;; "Types" for the functions in a SERVER plugin
;;;
;;; INIT      :: args         -> state
;;; CALL      :: term   state -> reply state
;;; CAST      :: term   state -> state
;;; TERMINATE :: reason state -> void

(include "../termite.scm")

(define-type server-plugin
  id: 2ca2d07c-5d6a-44a8-98eb-422b2b8e7296
  read-only:
  init
  call
  cast
  terminate)

(define *server-timeout* 1)

(define (server plugin)
  (let loop ((state (void)))
    (recv
      ((from tag ('init args))
       (let ((state ((server-plugin-init plugin) args)))
         (! from (list tag state))
         (loop state)))

      ((from tag ('call term))
       (call-with-values
         (lambda ()
           ((server-plugin-call plugin) term state))
         (lambda (reply state)
           (! from (list tag reply))
           (loop state))))

      (('cast term)
       (loop ((server-plugin-cast plugin) term state)))

      (('stop reason)
       ((server-plugin-terminate plugin) reason state)))))
```

```
(define (internal-server-start spawner plugin args)
  (let ((server (spawner (lambda () (server plugin)))))
    (!? server (list 'init args) *server-timeout*)
    server))

(define (server:start plugin args)
  (internal-server-start spawn plugin args))

(define (server:start-link plugin args)
  (internal-server-start spawn-link plugin args))

(define (server:call server term)
  (!? server (list 'call term) *server-timeout*))

(define (server:cast server term)
  (! server (list 'cast term)))

(define (server:stop server reason)
  (! server (list 'stop reason)))


;; build a trivial server, give a single callback to be invoked on
;; calls, should expect two values and return two
(define (make-simple-server-plugin initial-state callback)
  (make-server-plugin
   ;; INIT
   (lambda (args)
     initial-state)
   ;; CALL
   (lambda (term state)
     (callback term state))
   ;; CAST
   (lambda (args state)
     state)
   ;; TERMINATE
   (lambda (reason state)
     (void))))
```

A.3 Définition de type

```
(define-macro (define-termite-type type id tag . fields)

  (define (symbol-append . symbols)
    (string->symbol
     (apply
      string-append
      (map symbol->string symbols))))

  (define (make-maker type)
    (symbol-append 'make '- type))

  (define (make-getter type field)
    (symbol-append type '- field))

  (define (make-setter type field)
    (symbol-append type '- field '-set!))

  (if (not (eq? id id:))
      (error "id: is mandatory in define-termite-type"))

  (let* ((maker (make-maker type))
         (getters (map (lambda (field)
                         (make-getter type field))
                       fields))
         (setters (map (lambda (field)
                         (make-setter type field))
                       fields))

         (internal-type (gensym type))
         (internal-maker (make-maker internal-type))
         (internal-getters (map (lambda (field)
                                  (make-getter internal-type field))
                                fields))
         (internal-setters (map (lambda (field)
                                  (make-setter internal-type field))
                                fields))

         (facade-maker (gensym maker))
         (plugin (gensym (symbol-append type '-plugin)))
```

```
      (pid (gensym 'pid)))

  `(begin
     (define-type ,type
       id: ,tag
       constructor: ,facade-maker
       unprintable:
       ,pid)

     (define-type ,internal-type
       ,@fields)

     (define ,plugin
       (make-server-plugin
        ;; init
        (lambda (args)
          (apply ,internal-maker args))
        ;; call
        (lambda (term state)
          (match term
            ,@(map (lambda (getter internal-getter)
                     `(',getter (values (,internal-getter state) state)))
                   getters
                   internal-getters)))
        ;; cast
        (lambda (term state)
          (match term
            ,@(map (lambda (setter internal-setter)
                     `((',setter x) (,internal-setter state x) state))
                   setters
                   internal-setters)))
        ;; terminate
        (lambda (reason state)
          (void))))

     (define (,maker ,@fields)
       (,facade-maker (server:start-link ,plugin (list ,@fields))))

     ,@(map (lambda (getter)
              `(define (,getter x)
                 (server:call (,(make-getter type pid) x)
```

```
                              ',getter)))
        getters)

,@(map (lambda (setter)
         `(define (,setter x value)
            (server:cast (,(make-getter type pid) x)
                         (list ',setter value))))
        setters))))
```

ANNEXE B

CODE DES TESTS DE PERFORMANCE

B.1 Fibonacci

B.1.1 Scheme

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(define (main n)
  (let ((n (string->number n)))
    (write `(fib termite ,n ,(time* (fib n))))
    (newline)))
```

B.1.2 Erlang

```
-module(fib).

-export([run/1, fib/1]).


fib(X) when X < 2 -> X;
fib(X) ->
    fib(X-1)+fib(X-2).

run([Arg]) ->
    N = list_to_integer(Arg),
    {Time, _} = timer:tc(fib,fib,[N]),
    io:format("(fib erlang ~w ~w)~n", [N, round(Time / 1000)]),
```

```
halt(0).
```

B.2 Takeuchi

B.2.1 Scheme

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (tak x y z)
  (if (<= x y)
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))

(define (main x y z)
  (let ((x (string->number x))
        (y (string->number y))
        (z (string->number z)))
    (write `(tak termite (,x ,y ,z) ,(time* (tak x y z))))
    (newline)))
```

B.2.2 Erlang

```
-module(tak).

-export([run/1, tak/3]).

tak(X, Y, Z) when X =< Y ->
   Z;
tak(X, Y, Z) ->
   tak(tak(X-1, Y, Z), tak(Y-1, Z, X), tak(Z-1, X, Y)).


run([Arg1, Arg2, Arg3]) ->
    X = list_to_integer(Arg1),
    Y = list_to_integer(Arg2),
    Z = list_to_integer(Arg3),
```

```
{Time, _} = timer:tc(tak,tak,[X, Y, Z]),
io:format("(tak erlang (~w ~w ~w) ~w)~n", [X, Y, Z, round(Time / 1000)]),
halt(0).
```

B.3 Inversion naïve

B.3.1 Scheme

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (iota n)
  (define (i n acc)
    (if (= n 0)
        acc
        (i (- n 1)
           (cons n acc))))
  (i n '()))

(define (nrev lst)
  (if (null? lst)
      lst
      (append (nrev (cdr lst))
              (list (car lst)))))

(define (main n)
  (let* ((n (string->number n))
         (lst (iota n)))
    (write `(nrev termite ,n ,(time* (nrev lst))))
    (newline)))
```

B.3.2 Erlang

```
-module(nrev).

-export([run/1, nrev/1, iota/1]).


iota(X) ->
    iota (X-1, []).
```

```
iota(0, L) ->
    [0|L];

iota(X, L) ->
    iota(X-1, [X|L]).

nrev([]) ->
    [];

nrev([A|B]) ->
    nrev(B) ++ [A].


run([Arg]) ->
    N = list_to_integer(Arg),
    L = iota(N),
    {Time, _} = timer:tc(nrev,nrev,[L]),
    io:format("(nrev erlang ~w ~w)~n", [N, round(Time / 1000)]),
    halt(0).
```

B.4 Quick Sort

B.4.1 Scheme

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (make-random-list n)
  (if (= n 0)
      '()
      (cons (random-integer 1000000)
            (make-random-list (- n 1)))))

(define (qsort lst)
  (define (partition lst pivot k)
    (let loop ((lst lst)
               (smaller '())
               (greater '()))
      (cond
       ((null? lst)
        (k smaller greater))
       ((< (car lst) pivot)
        (loop (cdr lst)
              (cons (car lst) smaller)
              greater))
       (else
        (loop (cdr lst)
              smaller
              (cons (car lst) greater))))))

  (define (qs lst sorted)
    (if (null? lst)
        sorted
        (let ((pivot (car lst))
              (rest (cdr lst)))
          (partition rest
                     pivot
                     (lambda (smaller greater)
                       (qs smaller
```

```
                              (cons pivot
                                    (qs greater sorted))))))))

  (qs lst '()))

(random-source-randomize! default-random-source)

(define (main n)
  (let ((n (string->number n)))
    (let ((lst (make-random-list n)))
      (write `(qsort termite ,n ,(time* (qsort lst))))
      (newline))))
```

B.4.2 Erlang

```
-module(qsort).

-export([run/1, qsort/1]).

qsort(L) ->
   qsort(L, []).

qsort([X|L], ReallyBigs) ->
   {S, B} = partition(L, X),
   SB = qsort(B, ReallyBigs),
   qsort(S, [X|SB]);
qsort([], Bigs) ->
   Bigs.


partition([X|Rest], Y) when X =< Y ->
   {S, B} = partition(Rest, Y),
   {[X|S], B};
partition([X|Rest], Y) ->
   {S, B} = partition(Rest, Y),
   {S, [X|B]};
partition([],_) ->
   {[], []}.
```

```
mkrandlist(X) ->
   random:seed(),
   mkrandlist(X, []).

mkrandlist(0, L) ->
   L;
mkrandlist(X, L) ->
   mkrandlist(X-1, [random:uniform(1000000) | L]).


run([Arg]) ->
    N = list_to_integer(Arg),
    L = mkrandlist(N),
    {Time, _} = timer:tc(qsort,qsort,[L]),
    io:format("(qsort erlang ~w ~w)~n", [N, round(Time / 1000)]),
    halt(0).
```

B.5 Smith Waterman

B.5.1 Scheme

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

;; Smith Waterman string matching algorithm Scheme version (direct
;; translation of the Erlang version)
;;
;; Orginal Erlang version by Alexander Jean-Claude Bottema
;; [redacted]

(include "bench.scm")

(define (alpha-beta-penalty a b)
  (max (- a 4)
       (- b 1)))

(define (generate-sequence len r)
  (if (= len 0)
      '()
      (cons (remainder r 10)
            (generate-sequence (- len 1)
                               (remainder
                                (+ (* r 11) 1237501) 10067)))))

(define (generate-sequences n len r)
  (if (= n 0)
      '()
      (cons
       (generate-sequence len r)
       (generate-sequences (- n 1) len (+ r 1)))))

(define (match-entry top side upper-left upper left)
  (let ((me-left (alpha-beta-penalty (vector-ref left 2)
                                     (vector-ref left 0)))
        (me-upper (alpha-beta-penalty (vector-ref upper 2)
                                      (vector-ref upper 1))))
    (let ((me-upper-left
           (max me-left
```

```
                me-upper
                (+ (vector-ref upper-left 2) (if (= top side) 1 0))
                0)))
      (vector me-left
              me-upper
              me-upper-left
              (max me-upper-left
                   (vector-ref left 3)
                   (vector-ref upper 3)
                   (vector-ref upper-left 3))))))

(define (match-zero-entry top side v)
  (let ((left       (vector-ref v 0))
        (upper-left (vector-ref v 2))
        (max*       (vector-ref v 3)))
    (let* ((e-left (alpha-beta-penalty upper-left left))
           ;; (weight (max (- 1 (abs (- side top))) 0))
           (e-upper-left (max e-left (- 1 (abs (- side top))) 0))
           (e-max (max max* e-upper-left 0)))
      (vector e-left -1 e-upper-left e-max))))

(define (match* tops side prev upper-left left)
  (match0 tops side prev upper-left left '() 'none))

(define (match0 tops side prev upper-left left acc last)
  (if (null? tops)
      (cons acc last)
      (let ((top (car tops))
            (tops (cdr tops)))
        (if (eq? prev 'none)
            (let ((e (match-zero-entry top side left)))
              (match0 tops side 'none upper-left e (cons e acc) e))
            (let ((upper (car prev))
                  (prev (cdr prev)))
              (let ((e (match-entry top side upper-left upper left)))
                (match0 tops side prev upper e (cons e acc) e)))))))

(define (match-two-seq side top prev)
  (match-two-seq0 side top prev 'none))

(define (match-two-seq0 side top prev acc)
```

```
  (if (null? side)
      acc
      (let ((s (car side))
            (side (cdr side)))
        (let ((tmp (match* top s prev (vector 0 0 0 0) (vector 0 0 0 0))))
          (let ((row (car tmp))
                (result (cdr tmp)))
            (match-two-seq0 side top row result))))))

(define (match-sequences tops side)
  (match-sequences0 tops side -9999999))

(define (match-sequences0 tops side current-result)
  (if (null? tops)
      current-result
      (let ((top (car tops))
            (tops (cdr tops)))
        (let ((result (vector-ref (match-two-seq top side 'none) 3)))
          (match-sequences0 tops side (max current-result result))))))

(define (main n)
  (let ((n (string->number n)))
    (let ((tops (generate-sequences n 32 1))
          (side (generate-sequence 32 0)))
      (write `(smith termite ,n ,(time* (match-sequences tops side))))
      (newline))))
```

B.5.2 Erlang

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program     : Smith Waterman string matching algorithm Erlang version
% File        : smith.erl
% Author      : Alexander Jean-Claude Bottema (alexb@csd.uu.se)
% Datum       : Sep 19 1995
% Revidated   : Sep 19 1995
% ------------------------------------------------------------------------------
% Changes:
%    Optimization 1: Inline "match_weights"
%    Optimization 2: Guards on all int variables (explicit type information)
% ------------------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-module(smith).
-export([run/1, test/1, generate_sequence/2, match_sequences/2]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% max(A,B) - The maximum value of A and B
%

max(A,B) when integer(A), integer(B) ->
    if A > B -> A
     ; true  -> B
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% alpha_beta_penalty(A,B) - The penalty value of A and B
%
alpha_beta_penalty(A,B) when integer(A), integer(B) -> max(A-4,B-1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% match_weights(A,B) - Weight function of A and B
%
% match_weights(A,B) -> if A == B -> 1 ; true -> 0 end.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% generate_sequence( Length, Seed ) - Generate a random sequence of length
%                                     'Length' using seed value 'Seed'.
%

generate_sequence( Length, R ) when integer(Length) ->
    if  Length == 0 -> []
      ; true        ->  [R rem 10 | generate_sequence( Length - 1,
                                                       (R * 11 + 1237501)
                                                       rem 10067)]
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generate_sequences( No, Length, Seed )
%

generate_sequences( 0, _, _ ) -> [] ;
generate_sequences( N, Length, R ) when integer(N), integer(Length) ->
    [generate_sequence(Length, R) | generate_sequences(N-1,Length,R+1)].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entry representation:
%
% Entry = {Left,Upper,UpperLeft,Max}
%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% match_entry(Top,Side,UpperLeft,Upper,Left) -
%     Match sequence entries with surrounding information

match_entry(Top,Side,UpperLeft,Upper,Left) when integer(Top), integer(Side) ->
    MeLeft = alpha_beta_penalty( element( 3, Left ), element( 1, Left ) ),
    MeUpper = alpha_beta_penalty( element( 3, Upper ), element( 2, Upper ) ),
    %
    % match weight removed
    %
    if Top == Side ->
            MeUpperLeft =
                max(MeLeft,
                    max(MeUpper,
                        max( element( 3, UpperLeft ) + 1, 0 )))
```

```
                ;
        true ->
             MeUpperLeft =
                 max(MeLeft,
                     max(MeUpper,
                         max( element( 3, UpperLeft ), 0 )))
    end,
    {MeLeft, MeUpper, MeUpperLeft,
     max(MeUpperLeft,
          max(element(4, Left),
               max(element(4, Upper), element(4, UpperLeft))))}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% match_zero_entry( Top, Side, Left )
%

match_zero_entry( Top, Side, {Left,_,UpperLeft,Max} ) when integer(Top), integ
    ELeft = alpha_beta_penalty(UpperLeft, Left),
    %Weight = max(1-abs(Side-Top),0),
    EUpperLeft = max(max(ELeft,max(1-abs(Side-Top),0)),0),
    EMax = max(max(Max,EUpperLeft),0),
    {ELeft, -1, EUpperLeft, EMax}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% match(Top, Side, Prev, UpperLeft, Left )
%

match(Tops, Side, Prev, UpperLeft, Left) ->
    match0(Tops, Side, Prev, UpperLeft, Left, [], none).

match0([], _, _, _, _, Acc, Last) -> {Acc,Last} ;
match0([Top|Tops], Side, [Upper|Prev], UpperLeft, Left, Acc, _) when
    integer(Top), integer(Side) ->
    E = match_entry(Top, Side, UpperLeft, Upper, Left),
    match0(Tops, Side, Prev, Upper, E, [E|Acc], E) ;
match0([Top|Tops], Side, none, UpperLeft, Left, Acc, _) when
    integer(Top), integer(Side) ->
    E = match_zero_entry(Top, Side, Left ),
    match0(Tops, Side, none, UpperLeft, E, [E|Acc], E).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% match_two_seq(Side, Top, Prev)
%

match_two_seq(Side, Top, Prev) ->
    match_two_seq0(Side, Top, Prev, none).

match_two_seq0([], _, _, Result) -> Result ;
match_two_seq0([S|Side], Top, Prev, _) when integer(S) ->
    {Row,Result} = match(Top,S,Prev,{0,0,0,0},{0,0,0,0}),
    match_two_seq0(Side, Top, Row, Result).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% match_sequences(Tops, Side)
%

match_sequences(Tops, Side) ->
    match_sequences0(Tops, Side, -9999999).

match_sequences0([], _, MaxResult) -> MaxResult ;
match_sequences0([Top|Tops], Side, CrntResult) ->
    Result = element(4, match_two_seq(Top, Side, none)),
    match_sequences0(Tops, Side, max(CrntResult, Result)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% test(N)
%

test(N) ->
    Tops = generate_sequences(N, 32, 1),
    Side = generate_sequence(32, 0),
    {Time, _} = timer:tc(smith, match_sequences, [Tops, Side]),
    io:format("(smith erlang ~w ~w)~n", [N, round(Time / 1000)]).

run([Arg]) ->
    N = list_to_integer(Arg),
    test(N),
```

```
halt(0).
```

B.6 Self

B.6.1 Termite

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(init)

(include "bench.scm")

(define (main n)
  (let ((n (string->number n)))
    (write
     '(self
       termite
       ,n
       ,(time*
         (let loop ((n n))
           (! (self) n)
           (?)
           (if (> n 0)
               (loop (- n 1)))))))
    (newline)))
```

B.6.2 Gambit

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (main n)
  (let ((n (string->number n)))
    (write
     '(self
       gambit
       ,n
       ,(time*
         (let loop ((n n))
```

```
            (thread-send (current-thread) n)
            (thread-receive)
            (if (> n 0)
                (loop (- n 1)))))))
    (newline)))
```

B.6.3 Erlang

```
-module(self).

-export([run/1, test/1]).

test(0) -> ok;

test(N) ->
    self() ! (N - 1),
    receive N1 -> test(N1) end.

run([Arg]) ->
    N = list_to_integer(Arg),
    {Time, _} = timer:tc(self, test, [N]),
    io:format("(self erlang ~w ~w)~n", [N, round(Time / 1000)]),
    halt(0).
```

B.7 Spawn

B.7.1 Termite

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(init)

(include "bench.scm")

(define (main n)
  (let ((n (string->number n)))
    (write
     `(spawn
        termite
        ,n
        ,(time*
          (let loop ((n n))
            (spawn (lambda () n))
            (if (> n 0)
                (loop (- n 1)))))))
    (newline)))
```

B.7.2 Gambit

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (main n)
  (let ((n (string->number n)))
    (write
     `(spawn
        gambit
        ,n
        ,(time*
          (let loop ((n n))
            (thread-start! (make-thread (lambda () n)))
```

```
            (if (> n 0)
                (loop (- n 1)))))))
    (newline)))
```

B.7.3 Erlang

```
-module(spawn).

-export([run/1, test/1]).

test(0) -> ok;

test(N) ->
    F = fun () -> N end,
    spawn(F),
    test(N - 1).

run([Arg]) ->
    N = list_to_integer(Arg),
    {Time, _} = timer:tc(spawn, test, [N]),
    io:format("(spawn erlang ~w ~w)~n", [N, round(Time / 1000)]),
    halt(0).
```

B.8 Ring

B.8.1 Termite

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(init)

(include "bench.scm")

(define (make-relay next)
  (let loop ()
    (let ((k (?)))
      (cond
       ((> k 0)
        (! next (- k 1))
        (loop))
       (else
        (! next k))))))

(define (ring n k)
  (let loop ((current (self))
             (n n))
    (cond
     ((> n 1)
      (loop (spawn
              (lambda ()
                (make-relay current)))
            (- n 1)))
     (else
      (! (self) k)
      (make-relay current)))))

(define (main n k)
  (let ((n (string->number n))
        (k (string->number k)))
    (write `(ring termite (,n ,k) ,(time* (ring n k))))
    (newline)
    (? 1 'ok)))
```

B.8.2 Gambit

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(include "bench.scm")

(define (make-relay next)
  (let loop ()
    (let ((k (thread-receive)))
      (cond
       ((> k 0)
        (thread-send next (- k 1))
        (loop))
       (else
        (thread-send next k))))))

(define (ring n k)
  (let loop ((current (current-thread))
             (n n))
    (cond
     ((> n 1)
      (loop (thread-start!
             (make-thread
              (lambda ()
                (make-relay current))))
            (- n 1)))
     (else
      (thread-send (current-thread) k)
      (make-relay current)))))


(define (main n k)
  (let ((n (string->number n))
        (k (string->number k)))
    (write `(ring gambit (,n ,k) ,(time* (ring n k))))
    (newline)))
```

B.8.3 Erlang

```
-module(ring).

-export([run/1, ring/2, make_relay/1]).

make_relay(Next) ->
    receive
        K when K > 0 ->
            Next ! K - 1,
            make_relay(Next);

        K ->
            Next ! K
    end.

loop(K, Current, N) when N > 1 ->
    loop(K,
         spawn(ring, make_relay, [Current]),
         N - 1);

loop(K, Current, _) ->
    self() ! K,
    make_relay(Current).

ring(N, K) ->
    loop(K, self(), N).

run([N, K]) ->
    N1 = list_to_integer(N),
    K1 = list_to_integer(K),
    {Time, _} = timer:tc(ring,ring,[N1,K1]),
    io:format("(ring erlang (~w ~w) ~w)~n", [N1, K1, round(Time / 1000)]),
    halt(0).
```

B.9 Ping-pong

B.9.1 Termite

```
#!/usr/local/Gambit-C/bin/gsi -:dar1

(define (iota n)
  (if (= n 0)
      '()
      (cons n (iota (- n 1)))))

(define (ping-pong-player)
  (let loop ((n 0))
    (let ((msg (?)))
      (let ((from (car msg))
            (ball (cdr msg)))
        (if (eq? ball 'done)
            (! from n)
            (begin
              (! from (cons (self) ball))
              (loop (+ n 1))))))))

(define (run player1 player2 duration len)
  (! player1 (cons player2 (iota len)))
  (? duration 'ok) ;; pause

  (! player1 (cons (self) 'done))
  (! player2 (cons (self) 'done))
  (?))

(define (pingpong duration len)
  (write `(pingpong
           termite
           ,len
           ,(round (/ (run
                       (spawn ping-pong-player)
                       (remote-spawn node2 ping-pong-player)
                       duration
                       len)
                      duration)))))
```

```
  (newline))

(define (main #!optional (len "42"))
  (cond
   ((equal? (current-node) node1)
    ;; code for node 1
    (let ((len (string->number len))
          (duration 5))
      (pingpong duration len)
      (remote-spawn node2 (lambda () (exit)))
      (? 1 'done)))

   ;; code for node2
   (else
    (write (?))
    (newline))))
```

B.9.2 Erlang

```
-module(pingpong).

-export([run/1, bench/3, pingpong_player/1, iota/1]).


iota(0) -> [];
iota(N) -> [N] ++ iota(N - 1).

pingpong_player(N) ->
    receive
        {From, done} ->
            From ! N,
            exit(normal);
        {From, Ball} ->
            From ! {self(), Ball}
    end,
    pingpong_player(N + 1).

bench(Duration, Len, Remote) ->
    Player1 = spawn(pingpong, pingpong_player, [0]),
    Player2 = spawn(Remote, pingpong, pingpong_player, [1]),
    Player1 ! {Player2, iota(Len)},
    receive
        after Duration -> ok
    end,
    Player1 ! {self(), done},
    Player2 ! {self(), done},
    receive
        X ->
            io:format("(pingpong erlang ~w ~w)~n",
                      [Len, round(X / (Duration / 1000))])
    end.

run([Len, Node]) ->
    Remote = case Node of
                 "remote" ->
                     receive X -> X end;
                 _ -> list_to_atom(Node)
```

```
        end,
Duration = 5000,
bench(Duration, list_to_integer(Len), Remote),
spawn(Remote, erlang, halt, [0]),
halt(0).
```

B.10 "Migration"

B.10.1 Termite

```
#!/usr/local/Gambit-C/bin/gsi -:dari1

(include "bench.scm")

(define (run n)
  (let ((this (self)))
    (spawn
      (lambda ()
        (let loop ((n n))
          (if (> n 0)
              (begin
                (debug "Node: " (current-node))
                (debug "Test: " 'ok)
                (if (even? n)
                    (migrate-task node2)
                    (migrate-task node1)))
              (begin
                (! this 'done)
                (shutdown!)))
          (loop (- n 1))))))
  (?))

(define (main n)
  (let ((n (string->number n)))
    (cond
     ((equal? (current-node) node1)
      ;; code for node 1
      (write `(migrate
               termite
               ,n
               ,(time* (run n))))
      (newline)
      (remote-spawn node2 (lambda () (exit)))
      (? 1 'done))
     ;; code for node2
     (else (?)))))
```