Université de Montréal

Diffusion de modules compilés pour le langage distribué Termite Scheme

par

Frédéric Hamel

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

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

7 mars 2020

© Frédéric Hamel, 2020

Université de Montréal

Faculté des études supérieures et postdoctorales

Ce mémoire intitulé

Diffusion de modules compilés pour le langage distribué Termite Scheme

présenté par

Frédéric Hamel

a été évalué par un jury composé des personnes suivantes :

Michalis Famelis
(président-rapporteur)

Marc Feeley
(directeur de recherche)

Stefan Monnier
(membre du jury)

Résumé

Ce mémoire décrit et évalue un système de module qui améliore la migration de code dans le langage de programmation distribuée Termite Scheme. Ce système de module a la possibilité d'être utilisé dans les applications qu'elles soient distribuées ou pas. Il a pour but de faciliter la conception des programmes dans une structure modulaire et faciliter la migration de code entre les nœuds d'un système distribué. Le système de module est conçu pour le système Gambit Scheme, un compilateur et interprète du langage Scheme utilisé pour implanter Termite. Le système Termite Scheme est utilisé pour implémenter les systèmes distribués.

Le problème qui est résolu est la diffusion de code compilé entre les nœuds d'un système distribué quand le nœud destination n'a aucune connaissance préalable du code qu'il reçoit. Ce problème est difficile car les nœuds sont hétérogènes, ils ont différentes architectures (x86, ARM).

Notre approche permet d'identifier les modules de façon unique dans un contexte distribué. La facilité d'utilisation et la portabilité ont été des facteurs importants dans la conception du système de module.

Le mémoire décrit la structure des modules, leur implémentation dans Gambit et leur application. Les qualités du système de module sont démontrées par des exemples et la performance est évaluée expérimentallement.

Mots clés: Programmation fonctionnelle, Scheme, Erlang, Système de module, Système distribué, Agent mobile.

Summary

This thesis presents a module system for Termite Scheme that supports distributed computing. This module system facilitates application modularity and eases code migration between the nodes of a distributed system. This module system also works for developing non-distributed applications. The Gambit Scheme system is used to implement the distributed Termite and the Module system.

The problem that is solved is the migration of compiled code between nodes of a distributed system when the receiving node has no prior knowledge of the code. This is a challenging problem because the nodes are not homogenous, they have different architectures (ARM, x86).

Our approach uses a naming model for the modules that uniquely identifies them in a distributed context. Both ease of use and portability were important factors in the design of the module system.

The thesis describes a module system and how it was integrated into Gambit. The system allows developing distributed modular systems. The features of this system are shown through application examples and the performance is evaluated experimentally.

Keywords: Functional programming, Scheme, Erlang, Module System, Distributed System, Mobile Agent.

Table des matières

Liste des figures

Liste des sigles et des abréviations

FFI	Interface de fonction étrangère (*Foreign Function Interface*)
REPL	Boucle de lecture et d'évaluation (*Read Eval Print Loop*)
RPC	Appel de procédure à distance (*Remote Procedure Call*)

Remerciements

Je voudrais remercier mon directeur Marc Feeley pour l'inspiration, les précieux conseils et le support qu'il m'a fournis.

Merci à ma famille, qui a toujours été là pour moi tant dans les moments faciles et difficiles. Mon père Gérald et ma mère Sylvie mon supporté et encouragé durant mon parcours académique. Mon frère Alex qui m'a aussi encouragé. Merci à ma tante Jocelyne, qui m'a appris le piano et appris la confiance en soi. Merci à mes neveux Thomas et William pour leur présence et amour. J'ai la chance exceptionnelle de les côtoyer et de pouvoir enrichir et m'inspirer d'eux.

Mes amis ont une place importante dans ma vie, ils m'ont aidé à avancer dans la vie. Entre autres, je remercie Aldo Lamarre, Sébastien Richer, Abdel et les gens de l'association d'informatique pour les discussions intéressantes. Merci aux personnes que j'ai pu rencontrer au piano publique, qui mon données l'énergie de continuer.

Chapitre 1

Modularisation des systèmes distribués

Les programmes modernes ont une structure *modulaire*, c'est-à-dire que leur code se décompose logiquement en différentes parties relativement indépendantes, les *modules*. Cette structure a de nombreux avantages, entre autres sur les plans du développement, la maintenance et le déploiement des programmes. Ce mémoire porte sur la modularité dans le contexte des systèmes distribués où le calcul est réparti sur de multiples ordinateurs reliés en réseau. Ce contexte pose des défis particuliers de déploiement.

1.1. Déploiement

Le *déploiement* d'un programme c'est sa mise en service pour qu'il soit utilisable. Ça consiste à installer une forme exécutable du programme sur le(s) ordinateur(s) de sorte à pouvoir l'exécuter. La façon de diffuser les modules, les stocker sur disque, les charger en mémoire, etc. peut prendre plusieurs formes.

Un programme sous forme *monolithique* contient dans son code exécutable toutes les instructions exécutées par l'ordinateur. Cette forme était la norme dans les premiers systèmes informatiques, et l'est toujours pour les systèmes embarqués qui n'ont pas de système d'exploitation indépendant. Lorsqu'un système d'exploitation est disponible sur l'ordinateur on peut le considérer comme étant un module puisqu'il offre des services précis avec une interface standardisée. Dans ce cas, un programme peut prendre la forme d'un seul fichier de code qui, à son exécution, communiquera avec le système d'exploitation pour accéder à ses services. Ce genre de fichier exécutable est obtenu par une *édition de liens statique* qui combine en un seul fichier tous les modules (à l'exception du système d'exploitation). Par rapport à la forme monolithique, cette organisation simplifie le développement, car le

programmeur n'a pas à se soucier du développement des services de base comme l'accès aux fichiers, la gestion des processus et de la mémoire, etc. Le programme peut être diffusé à d'autres ordinateurs ayant le même système d'exploitation simplement en y transférant le fichier exécutable.

L'édition de lien statique a un certain nombre de défauts. L'état des modules utilisés au moment de l'édition de liens est figé au sein du programme, ce qui empêche la mise à jour individuelle des modules à de nouvelles versions. Il faut recompiler tous les modules qui ont subi une mise à jour et refaire l'édition de liens du programme principal. Le coût en temps et l'effort pour un petit changement sont importants. Le chapitre 4 va détailler plus en profondeur ces problèmes.

L'édition de liens peut se faire paresseusement par le système d'exploitation à l'exécution du programme, ce qu'on appelle l'*édition de liens dynamiques*. Cela permet de garder la structure modulaire au *déploiement*. Chaque module est une composante séparée du programme principal. Ces modules sont lus du disque et chargés en mémoire durant l'exécution du programme. Ce chargement est effectué par l'*éditeur de liens dynamique du système d'exploitation* [**2, 18**] qui s'occupe de lier les fonctionnalités des modules au programme principal. L'avantage principal du chargement dynamique de module est la possibilité de mise à jour individuelle d'un module sans avoir à lier le programme principal; dans le modèle statique, le programme principal doit être lié à nouveau avec les modules. Les modules chargés dynamiquement par le système d'exploitation peuvent être partagés entre différents programmes.

1.2. Systèmes distribués

Un *système distribué* est un groupe d'ordinateurs, les *nœuds* de calcul, reliés en réseau afin qu'ils puissent échanger des messages et coordonner leurs activités. Chaque nœud peut exécuter le même code que les autres nœuds ou bien un code spécialisé au rôle qu'il joue au sein du système (e.g. serveur vs client). D'une façon ou de l'autre on parlera de l'ensemble du code comme étant le *programme distribué* qu'ils exécutent.

Les programmes distribués posent de nouveaux problèmes de déploiement, car les nœuds sont rarement identiques. Ils peuvent avoir des architectures matérielles différentes, et/ou des ressources et périphériques différents, et/ou des systèmes d'exploitation différents. Il

est à noter que ces caractéristiques peuvent changer pendant la période d'exploitation du programme, par exemple lors de mise à jour du matériel et du système d'exploitation. Ce problème est exacerbé par le *code mobile*, c'est-à-dire un calcul en exécution sur un nœud qui se déplace, ou *migre*, sur un autre nœud pour poursuivre son exécution. Cette *migration de tâche* est utile pour améliorer la performance lorsque le nœud destinataire est plus puissant ou possède localement les données utilisées par la tâche migrée. Finalement, dans le cas de programmes distribués offrant un service à l'externe, on veut minimiser les interruptions de service causées par les mises à jour du système ou d'une de ses parties, que ce soit 1) au niveau matériel et/ou le système d'exploitation ou 2) le programme distribué lui-même. Dans le premier cas, la migration de tâche peut être utilisée pour migrer le service à un autre nœud le temps que se fasse la mise à jour. Dans le deuxième cas, on aimerait utiliser les nouveaux modules modifiés sans avoir à arrêter le programme et le redémarrer.

Il apparait donc avantageux que le code du programme distribué prenne une forme exécutable *portable* qui peut s'adapter aux particularités de chaque nœud sans demander au développeur de modifier ou recompiler le programme distribué. Idéalement, cette portabilité ne devrait pas causer des pertes de performance. D'autre part, la possibilité de substituer une version d'un module par une nouvelle version sans interruption du programme est attrayante.

1.3. Modules versionnés

Nous définissons le terme *version de module* comme étant tout simplement un état de son code. Ainsi, lorsqu'on fait une modification au code d'un module pour corriger un problème ou pour étendre ses fonctionnalités on obtient une nouvelle version du module. La version d'un module est donc essentielle pour identifier son code, et donc sa fonctionnalité, de façon précise. Pour permettre plusieurs versions d'un module au sein d'un programme, il est important que chaque version des modules soit distinguable. Lors de la migration de code, il est possible que plusieurs versions d'un module soient chargées pour accéder des fonctionnalités.

Le concept de version de module n'est pas intégré à la sémantique de plusieurs langages de programmation. En Python et Java les modules sont référés dans le code avec un nom qui n'inclut pas la version. Go est un exemple de langage de programmation qui permet d'inclure la version lorsqu'on réfère à un module.

Dans les systèmes qui considèrent chaque version d'un module comme un module différent, il devient possible d'écrire des programmes qui utilisent de multiples versions d'un module simultanément. Cela peut amener certains problèmes de conflit entre les versions. Ces conflits peuvent être observés dans les langages interprétés comme JavaScript ou dans les langages compilés comme C/C++. Les conflits peuvent être dans les noms des fonctions du module ou dans des variables globales partagées entre plusieurs versions du module.

1.4. Code mobile

Dans un système distribué, il est utile d'amorcer des calculs sur un nœud à partir d'un autre. Cela peut se faire avec un appel RPC (*Remote Procedure Call*) ou par *migration de tâches*. Un appel RPC correspond à une requête d'exécution d'un calcul sur un nœud distant. Une migration de tâche copie l'état courant de la tâche et l'envoie sur le nœud distant où son exécution est continuée.

Les appels RPC sont présents dans plusieurs systèmes et langages de programmation. Le protocole `ssh` permet une forme d'appel RPC par invocation de processus sur un nœud distant. Java offre les appels distants par un mécanisme similaire au RPC nommé RMI (*Remote Method Invocation*). Le langage C offre des bibliothèques qui implémentent un protocole de RPC. Ces variantes de RPC sont limitées aux fonctionnalités fournies par le nœud distant. Pour le bon fonctionnement de cette approche, il est nécessaire que le service demandé soit déjà déployé sur la machine distante.

La migration de tâche permet de transférer une tâche d'un nœud à un autre. Ce mécanisme a été implémenté dans différents langages comme Erlang [**25**], Java [**5**], Scheme [**32, 16**], JavaScript [**24**]. Ce mécanisme requiert typiquement que l'ensemble du code soit disponible sur le nœud distant.

Stefan Fünfrocken a conçu un système d'agents mobiles écrit en Java [**5**]. Il capture l'état présent du programme qui inclut les valeurs et les types de toutes les variables de chaque objet. La pile des appels de méthode avec les valeurs de toutes les variables est aussi incluse dans l'état du programme. Ces informations sont transmises au nœud distant qui les utilise pour reconstruire le programme initial. L'ensemble des méthodes sur le nœud de départ est présent sur le nœud d'arrivée.

Le système implémenté dans le langage de programmation Erlang [**25**] qui implémente des agents mobiles requiert que le code soit disponible sur le nœud destination.

Le système de migration en JavaScript [**24**] utilise des nœuds pour conserver le code. Il est basé sur le fait que le code des agents est connu par les nœuds spécialisés. Puisque le code des agents est connu, il suffit de transmettre l'état de l'agent pour effectuer une migration. Le scénario de migration de code présenté dans le chapitre 2 nous a inspiré dans la conception d'un système de module. Notre motivation est la migration de code entre des nœuds d'un système distribué comme présenté à la fin du chapitre 2. Nous avons développé un système de module pour le langage de programmation Scheme qui offre la possibilité de migrer un agent sur un nœud qui ne connait pas à priori le code de l'agent, et qui permet à des versions multiples d'un module de coexister.

Notre point de vue est que la migration de code ne devrait pas demander une préconfiguration des nœuds, car une gestion coordonnée de tous les nœuds d'un système distribué est difficile ou impossible lorsqu'il y a plusieurs nœuds et/ou des nœuds de nature différente et/ou il n'y a pas un gestionnaire unique de l'ensemble des nœuds. Même dans le cas où les nœuds sont sous le contrôle d'un unique gestionnaire, il serait avantageux de ne pas interrompre le fonctionnement du système distribué lorsque les modules sont améliorés.

Le problème qui nous attaquons est la transmission de code compilé entre les nœuds d'un système distribué hétérogène. Chaque nœud a possiblement une architecture différente (x86, ARM) et est sur un système d'exploitation différent (Windows, Mac, Linux). Les architectures différentes ne sont pas compatibles, donc le code compilé ne peut pas juste être transmis tel quel. Le format des exécutables natifs varie d'un système d'exploitation à un autre. Le code transmis a des dépendances qui doivent aussi être transmises pour son bon fonctionnement.

Les contributions de ce travail sont la conception d'un système de module pour les systèmes distribués et son intégration dans Gambit Scheme. Ce système permet la diffusion des modules compilés entre les nœuds d'un système distribué sans préconfiguration des modules avec le code et permettant de multiples versions d'un module de coexister.

1.5. Survol

Le chapitre 2 explique les approches existantes de modularisation en Scheme. Le chapitre 3 traite de l'implémentation du système de module et son intégration dans Gambit, un compilateur/interprète performant pour Scheme. Le chapitre 4 explique les mécanismes de chargement des modules. Le chapitre 5 traite de l'organisation générale des modules. Le chapitre 6 traite de la diffusion des modules et de la migration de tâche. Le chapitre 7 présente une évaluation du système de modules avec des résultats d'expériences effectuées entre des systèmes d'exploitation et architectures différentes.

Chapitre 2

Bibliothèques Scheme

Pour développer notre approche de modularisation dans le contexte de systèmes distribués nous avons choisi un langage de programmation nous permettant de facilement expérimenter avec le code mobile. Le langage de programmation Scheme a plusieurs implémentations, dont Gambit, Racket [**27**], Chez Scheme [**11**], Guile [**15**], Chicken [**7**], Bigloo [**29**], Gerbil [**33**] et JazzScheme [**6**]. Le langage Termite Scheme [**16**], une variante de Scheme étendue avec des fonctionnalités de programmation distribuée inspirées du langage Erlang, nous est apparu comme une option intéressante vue sa performance, sa portabilité et son support pour la méta programmation facilitée par son homoiconicité. Termite est implémenté en tant que module de Gambit. Notre travail a donc pu se concentrer sur la conception d'un système de module spécialisé aux besoins du code mobile. Le système Termite Scheme est expliqué sommairement à la section 2.4.

Ce chapitre introduit la syntaxe de Scheme et les différentes approches de modularisation existantes. Afin de faciliter son adoption, le système de module que nous avons développé se base sur ces approches de modularisation.

2.1. Scheme et sa syntaxe

Le langage Scheme [**9**], conçu en 1975 par Guy L. Steele et Gerald Jay Sussman, est une variante minimaliste de Lisp. Depuis sa conception, plusieurs normes ont vu le jour; les plus connues étant le R4RS (1991), R5RS (1998), R6RS (2007) et R7RS (2013). Notre implémentation utilise R7RS, mais pourrait aussi s'appliquer à la syntaxe de R6RS. Il suffit que la structure des bibliothèques permette les noms uniques pour les modules.

Scheme est un langage de programmation avec un système de type dynamique. Les types sont associés aux valeurs plutôt qu'aux variables et une variable donnée n'est pas contrainte à contenir un type particulier. Il permet la programmation fonctionnelle, la programmation impérative et la méta programmation.

Sur le plan syntaxique, Scheme hérite la syntaxe préfixe parenthésée de Lisp. Un programme Scheme est une séquence de *s-expressions* (expressions symboliques). Chaque *s-expression* correspond soit à une constante, une variable ou une forme `(<op> <arg>...)` qui dénote un appel de procédure, un appel de macro, ou une forme spéciale.

Les formes syntaxiques spéciales de base en Scheme sont `define`, `lambda`, `let`, `if` et `set!`.

- La forme `(define <name> <val>)` associe le nom *<name>* avec la valeur de l'expression *<val>*. Elle est utilisée pour définir les variables globales.
- La forme `(lambda <params> <body>)` permet la définition de procédures anonymes. Les paramètres sont *<params>* et le corps de la procédure est l'expression *<body>*.
- La forme `(let <bindings> <body>)` permet de créer des liaisons (*<bindings>*) visibles seulement dans le contexte de l'expression *<body>* dont la valeur de la forme `let`. Les liaisons sont sous la forme d'une liste associative nom et valeur.
- L'évaluation conditionnelle est obtenue par la forme `(if <e1> <e2> <e3>)`. L'expression *<e3>* est exécutée si la valeur de l'expression *<e1>* est fausse sinon l'expression *<e2>* est exécutée.
- La forme `(set! <name> <val>)` modifie le contenu de la variable *<name>* avec la valeur *<val>*.

À titre d'exemple, la figure 2.1 montre un programme Scheme simple avec deux définitions de fonctions.

Les différents types primitifs disponibles dans la norme Scheme sont `boolean`, `pair`, `symbol`, `number`, `char`, `string`, `vector`, `port` et `procedure`. Plusieurs systèmes Scheme offrent des extensions à ces types, tels les dictionnaires (*hash tables*), les structures (*records*) et tableaux numériques (par exemple Gambit offre le type `u8vector` qui est un tableau d'octets).

```
(define sq (lambda (x) (* x x)))


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


(println (fact (sq 10)))
```

Fig. 2.1. Programme Scheme qui imprime la factorielle de 100.

2.2. Procédures et macros

Les procédures sont des objets de première classe, c'est-à-dire pouvant être manipulées comme n'importe quel type de donnée. Elles peuvent être passées en tant que paramètre à une procédure et retournées en tant que résultat. Certaines procédures, comme `for-each`, `map` et `fold`, en tirent profit.

Les procédures sont définies par la forme `lambda` qui a une liste de paramètres et une séquence d'au moins une *s-expression* comme corps. Lors de l'appel d'une procédure, chacun des paramètres actuels est évalué puis propagé à la procédure. C'est un mode de passage de paramètre par valeur. Les boucles sont normalement exprimées sous la forme de récursion. Cela est facilité par l'existence de l'appel terminal [**20**] garanti, qui permet des récursions sans coût en espace. L'appel terminal de procédure est simplement le dernier appel effectué dans une procédure. Ces appels n'ont pas besoin de retourner, ils permettent d'exprimer les boucles très longues souvent utilisées dans le contexte d'un serveur qui boucle sur les évènements. Par exemple, l'appel récursif de `repeat` dans la figure 2.2 est terminal, car l'appelant n'a pas besoin du résultat pour terminer, donc l'appelant peut donner le contrôle à la procédure. Sans les appels terminaux, les boucles infinies mèneraient à un débordement de la pile d'appels. Il est assez simple d'ajouter une syntaxe pour les boucles à l'aide de la méta programmation.

```
(define (repeat msg n)
  (if (> n 0)
      (begin
        (display msg)
        (repeat msg (- n 1))) ;; appel terminal
      (newline))) ;; appel terminal
```

Fig. 2.2. Exemple de boucle affiche `n` fois le message `msg` qui utilise des appels terminaux lors de l'appel de `repeat` et `newline`.

La méta programmation en Scheme est basée sur la capacité d'un programme de manipuler d'autres programmes comme des données. Cela implique qu'il est possible de générer, analyser et modifier le code d'un autre programme. Les constructions utilisées pour manipuler le code du programme et ajouter des extensions au langage sont les macros.

En C et C++, les macros sont basées sur un modèle de remplacement textuel simple et ne permettent pas de récursion. Un appel à la macro est remplacé par le corps de celle-ci. Les macros de style Lisp permettent des transformations qui se font avec l'ensemble des procédures du langage, ce qui leur donne plus de flexibilité. La différence entre les procédures et les macros est le mode de passage de paramètres. Les paramètres sont passés à la macro sans être évalués. Certaines implémentations de Scheme offrent la forme `define-macro` pour définir les macros. Cette forme est équivalente au `defmacro` de Lisp. Elle accepte en paramètre des *s-expression*s et retourne une *s-expression*.

```
(define-macro (include filename)
  (call-with-input-file
    filename
    (lambda (port)
      `(begin
        ,@(read-all port)))))
```

Fig. 2.3. Implémentation de la macro `include` qui permet l'inclusion d'un fichier dans un autre fichier avec la forme `define-macro`.

Un exemple qui montre les capacités des macros Scheme est la macro `include`. Cette macro permet l'inclusion du contenu d'un fichier au point d'appel. Pour inclure un fichier dans un autre, il faut tout d'abord lire le contenu du fichier à inclure. Ensuite, il suffit de retourner le code lu. La figure 2.3 montre une implémentation de cette macro avec `define-macro`. Pour les systèmes Scheme n'offrant pas la forme `define-macro`, il est assez simple d'implémenter la forme `include` avec `define-syntax` qui est une forme de définition de macro alternative à `define-macro`. L'implémentation de cette macro est donnée à la figure 2.4. Les particularités de `define-syntax` ne sont pas expliquées puisqu'elles ne sont pas pertinentes à ce mémoire.

```
(define-syntax include
  (lambda (stx)
    (syntax-case stx ()
      ((include filename)
       (let ((content
              (call-with-input-file
                 (syntax->datum (syntax filename))
                 (lambda (port)
                   `(begin ,@(read-all port))))))
         (datum->syntax stx content))))))
```

Fig. 2.4. Implémentation de la macro `include` qui permet l'inclusion d'un fichier dans un autre fichier avec la forme `define-syntax`.

2.3. Structure des bibliothèques

Les bibliothèques, aussi appelées modules, facilitent le partage de fonctionnalités entre plusieurs programmes. Dans le standard R4RS [**8**] et R5RS [**22**] les modules consistent en des fichiers Scheme qui contiennent des définitions de procédures et de macros. Ils sont chargés dans l'environnement global par la procédure `load`. Certaines implémentations de Scheme ont la forme spéciale `include` qui permet de séparer un module Scheme en plusieurs

parties. Cette forme peut s'ajouter facilement au langage (tel que montré à la figure 2.4). Le modèle de bibliothèque basé sur `load` et `include` possède plusieurs lacunes.

- Ce modèle de chargement n'est pas à l'abri des chargements multiples d'un module qui mène soit à de la duplication de code (dans le cas de la forme `include`) ou à de la réévaluation d'un code (dans le cas de `load`).
- Toutes les déclarations dans un module sont ajoutées à l'environnement global lors du chargement par `load`. Cela mène à des conflits de nom entre les identifiants du module principal et des modules importés.
- L'importation d'un module par `load` ou `include` nécessite la connaissance de son emplacement dans le système de fichier.

Le chargement d'un module dans Gambit Scheme par `load` se fait en plusieurs phases: l'analyse lexicale, l'analyse syntaxique, l'expansion de macro et l'évaluation. L'analyse lexicale brise l'expression en lexèmes. La séquence de lexèmes est associée à un contexte par l'analyse syntaxique dans laquelle il y a aussi une expansion des macros. L'inclusion d'un fichier avec `include` n'effectue qu'une analyse lexicale qui est effectuée par la procédure `read-all`. L'évaluation est effectuée après l'analyse syntaxique et l'expansion des macros. D'autres systèmes Scheme permettent le chargement des macros par `load`.

Dans un module, il y a du code qui est exécuté lors de l'expansion (les macros) et à l'évaluation. La procédure `load` donne accès aux procédures définies dans le module, mais pas aux macros, car elles sont expansées. Après un `load`, il ne reste que les procédures et variables globales qui résultent de l'expansion des macros. Pour avoir accès aux macros, il faut utiliser la forme spéciale `include` qui est expansée par le contenu du fichier. L'expression `(include "foo.scm")` est remplacée par le contenu du fichier `foo.scm`. L'exemple 2.5 montre un exemple de module simple n'utilisant que la procédure `load`.

Le standard R6RS [**31**], qui est une option possible pour concevoir le système de module, renforce le concept de module avec la forme syntaxique `library` qui a 4 parties: le nom, une sous-forme `export`, une sous-forme `import` et le corps du module. Le nom du module l'identifie de façon unique et il peut contenir une spécification de version. La version est spécifiée par une liste d'entiers positifs. Une liste vide `()` correspond à ne pas spécifier la version. Ensuite, il y a la liste des exportations qui est spécifiée par la sous-forme `export`. Chaque

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

```
;; main.scm
(load "fact.scm")
(display (fact 5))
```

Fig. 2.5. Le fichier `fact.scm` est un exemple de module R4RS exposant la fonction mathématique `fact`. Le fichier `main.scm` est un programme principal qui utilise le module `fact.scm`.

élément de cette liste est soit un identifiant ou une sous-forme `rename` qui renomme l'identifiant exporté. L'`import` donne la liste des dépendances du module. Chaque dépendance spécifie:

- le nom du module importé et de façon optionnelle, une contrainte sur la version;
- le niveau d'import (temps d'expansion de macros ou évaluation);
- un sous-ensemble de l'`export` du module et le nom local utilisé au sein du module présent pour chaque exportation du module.

Le corps du module contient une séquence de définitions suivie par une séquence d'expressions. Une définition peut être locale ou exportée. Les expressions initialisent le module lors de l'exécution. Le R6RS ajoute aussi la forme `import` pour l'importation d'un module et enlève la procédure `load`. Contrairement au `load`, la forme `import` garantit le chargement unique d'un module. La syntaxe du `import` permet de manipuler les noms des symboles importés et exportés.

```
(library <library name>
  (export <export spec> ...)
  (import <import spec> ...)
  <library body>)
```

Fig. 2.6. Structure globale d'un module R6RS

La forme `import` de R6RS permet l'importation d'un ensemble de modules. Chaque spécification d'import `<import spec>` peut être une simple importation ou une importation avec un niveau. Les différents niveaux d'importation sont: `run`, `expand` ou `(meta <level>)`. Le niveau méta donné par `<level>` est un entier exact. Un niveau d'importation 0 correspond à `run` et un niveau d'importation de 1 correspond à `expand`.

La syntaxe pour l'importation avec les niveaux méta est décrite dans la spécification R6RS [**31**]. Le `import` R6RS permet un grand contrôle lors de l'importation des modules au prix d'une sémantique plus complexe.

Les déclarations d'un module R6RS sont dans un espace distinct de l'espace global et des autres modules. Les déclarations d'un module ne peuvent pas être en conflit avec des déclarations globales ou d'autres modules. L'élément qui permet de distinguer deux modules est leurs identifications (nom avec version). Le nom du module correspond à l'espace de nom du module. C'est ce qui unit les déclarations au module et empêche les conflits de nom entre les modules.

Le standard R7RS [**30**] simplifie la syntaxe des modules R6RS [**31**]. L'importation d'un module fait abstraction du niveau d'importation. Les macros et les procédures sont importées de façon transparente. La procédure `load` est conservée. La forme spéciale pour définir un module est `define-library`. Il n'y a pas d'ordre spécifique dans les déclarations de la bibliothèque comme en R6RS. Un module R7RS commence par un nom suivi de plusieurs déclarations.

```
(define-library <library name>
  <library declaration>*)
```

Fig. 2.7. Structure globale d'un module R7RS

Une déclaration dans un module est soit un `import`, un `export`, un `include`, un `include-ci`, un `cond-expand` ou un bloc `begin`.

- La déclaration `import` est équivalente au R6RS sans le concept de niveau d'importation.
- La déclaration `export` est idem au R6RS.
- Les déclarations `include` et `include-ci` permettent l'inclusion d'un fichier en tant qu'un bloc de code. La seconde forme non sensible à la casse des caractères.

- La déclaration `cond-expand` est une extension du `SRFI-0` [**12**] dans le contexte d'une bibliothèque permettant l'inclusion conditionnelle de code.

La forme `import` en R7RS permet d'importer un ensemble d'identifiants qui sont exportés par un module. Chaque ensemble importé spécifie le nom des identifiants du module et parfois même associe un nom local aux identifiants. L'`import` peut prendre l'une des formes suivantes:

- *<library-name>*
- `(only` *<import-set>* *<id1>* `...)`
- `(except` *<import-set>* *<id1>* `...)`
- `(prefix` *<import-set>* *<id>*`)`
- `(rename` *<import-set>* `(`*<id1>* *<id2>*`) ...)`

Dans la première forme, tous les identifiants exportés par la bibliothèque *<library-name>* sont importés. Les autres types de *<import-set>* modifient l'ensemble comme suit:

- **only** inclut seulement les identifiants spécifiés. C'est une erreur d'importer un identifiant non exporté par la bibliothèque.
- **except** permet d'exclure des identifiants de l'ensemble.
- **rename** renomme les identifiants importés.
- **prefix** ajoute un préfixe à l'ensemble des identifiants importés.

2.4. La syntaxe de Termite

Ce langage, conçu par Guillaume Germain en 2006, est l'implémentation du style de programmation par message d'Erlang dans Gambit Scheme. Les processus sont représentés par les threads de Scheme. La communication entre ces processus est effectuée par un système de boîte de message présent dans Gambit. Chaque thread possède une file d'attente de messages entrants.

Termite expose plusieurs procédures pour gérer les processus et la transmission de message entre chacun des nœuds.

- La procédure `(spawn` *<thunk>* `#!key` *<name>*`)` permet de créer des processus Termite. Le paramètre *<thunk>* est la procédure principale du thread. Le paramètre *<name>* est un paramètre associatif optionnel.

- La procédure (! *<node>* *<msg>*) envoie le message msg au nœud node.
- La procédure (?) permet de recevoir un message envoyé par un autre processus au nœud courant.
- La procédure (!? *<node>* *<msg>*) envoie un message au nœud node et attend la réponse.
- La macro (recv (*<pattern>* *<expr1>* ...) ...) permet un filtrage par motif des messages reçus du nœud courant.

La figure 2.8 est un exemple d'application Termite. Cet exemple a un bogue volontaire, le serveur pong répond à un message *ping* par le message *gnop* au lieu de *pong*.

Dans cet exemple, il y a un scénario de mise à jour de code à la volée, c'est-à-dire sans interruption de service. Il y a deux nœuds, un serveur qui exécute le code de buggy-pong.scm et un client exécute le code ping.scm.

```
;; ping.scm (running on node1)
(declare (block))
(import (termite))

(define pong-server
  (remote-service 'pong-server node2))

(define new-server
  (letrec ((server-loop
            (lambda () ;; code that will be migrated
              (recv
                ((from tag 'clone)
                 (call/cc
                   (lambda (k)
                     (! from (list tag k)))))
                ((from tag 'ping)
                 (! from (list tag 'pong)))
                (('update k)
                 (k #t)))
              (server-loop))))
  (spawn server-loop)))

(node-init node1)

(!? pong-server 'ping) ; => gnop
(! pong-server (list 'update (!? new-server 'clone)))
(!? pong-server 'ping) ; => pong
```

```
;; buggy-pong.scm (running on node2)
(declare (block))
(import (termite))

(define pong-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'ping)
           (! from (list tag 'gnop))) ;; BUG!
          (('update k)
           (k #t)))
        (loop)))))

(node-init node2)

;; publish the pong server
(publish-service 'pong-server pong-server)
```

Fig. 2.8. Un serveur Ping-Pong en Termite Scheme

(1) Le client envoie le message ping au serveur buggy-pong.

(2) Le serveur répond par le message gnop, c'est un bogue.

(3) Le client envoie le message `clone` au service local `new-server` contenant le code sans le bogue.

(4) Le service local clone son état courant en capturant sa continuation par `call/cc`.

(5) Le client envoie la l'état du service local au serveur `buggy-pong` par le message `update`.

(6) Le serveur exécute la continuation, ce qui met à jour son code.

(7) Le client envoie un autre message `ping` au serveur `buggy-pong`.

(8) Le serveur répond par le message `pong`.

Dans ce scénario, la mise à jour de code est effectuée dans les étapes 4, 5 et 6. Cela permet de réparer le bogue qui est dans le serveur. Termite permet déjà la migration de code, mais sous certaines conditions.

Les nœuds doivent être interprétés, pour que le code source des procédures soit transmis. Ceci permet d'exécuter du code arbitraire sur le nœud distant. Lorsque le code est compilé, l'information transmise ne contient pas le code source.

2.5. Conclusion

Pour notre objectif de modularisation dans les systèmes distribués, les révisions de Scheme qui offrent les fonctionnalités requises sont R6RS et R7RS. Elles permettent toutes les deux la modularisation du code et la création de modules avec des noms uniques, ce qui est requis pour le bon fonctionnement du code mobile. Nous avons choisi l'approche avec `define-library`, puisque Gambit se veut compatible avec le R7RS. Cela a permis d'expérimenter avec le code mobile. En plus, cela facilite son adoption et sa portabilité entre les implémentations de Scheme qui sont de plus en plus compatibles avec R7RS. Les modules existent dans le langage de programmation Scheme. L'implémentation Gambit Scheme n'avait pas le support des modules.

Chapitre 3

Implémentation des modules

Ce chapitre traite de notre implémentation des modules dans Gambit qui se veut le plus portable entre les différentes implémentations de Scheme. Nous avons ajouté une syntaxe pour les modules à Gambit parce qu'elle était manquante. Notre implémentation des modules se base sur la syntaxe de R7RS et utilise des formes déjà existantes et aussi de nouvelles formes qui ont qui ont été ajoutées au besoin. Ces nouvelles formes nous ont permis d'intégrer la syntaxe des modules R7RS avec les modules primitifs de Gambit. Il y a deux façons d'écrire des modules. Les modules primitifs sont utilisés pour implémenter des fonctionnalités système dans Gambit, par exemple, un module qui implémente une nouvelle syntaxe. La syntaxe des modules R7RS est utilisée lorsqu'un module se veut compatible avec d'autres implémentations du langage de programmation Scheme supportant R7RS.

Les formats des modules R7RS, construits avec `define-library`, contiennent plusieurs composantes.

- L'espace de nom du module qui regroupe toutes les fonctionnalités.
- Une liste des modules qui sont utilisés par le module courant.
- Une liste des symboles exportés par le module.

Il est possible de modifier les identifiants d'un module lors de l'importation et de l'exportation. L'importation multiple d'un module doit correspondre à un seul chargement. Pour exprimer les relations entre les modules certaines formes spéciales ont été ajoutées dans Gambit. Si les concepteurs de bibliothèques respectent la syntaxe R7RS, alors il est possible de l'importer dans Gambit. C'est indépendant du système Scheme pour lequel les bibliothèques ont été écrites.

3.1. La notation des fichiers dans Gambit

Gambit utilise une notation qui est compatible avec l'environnement du système hôte. Il permet de référer au répertoire maison de l'utilisateur et aux répertoires d'installation.

Un chemin est une chaîne de caractère qui dénote une ressource. Le séparateur des composantes d'un chemin dépend du système d'exploitation, Linux et macOS utilisent '/' alors que MSDOS et Microsoft Windows utilisent '\'.

La notation qui permet de référer au répertoire maison de l'utilisateur courant est par la notation '`~`' qui n'est pas suivit d'un autre '`~`'. Le nom qui suit '`~`' est le nom de l'utilisateur. Par exemple, `~USER` représente le répertoire de l'utilisateur `USER`.

Un répertoire symbolique commence par les caractères '`~~`'. S'il n'y a rien qui suit ces caractères alors ça réfère au répertoire d'installation central de Gambit. Sinon ce qui suit le '`~~`' est le nom d'un répertoire symbolique qui sera cherché dans un dictionnaire de répertoires symboliques. Par exemple, '`~~lib`' réfère au répertoire d'installation '`lib`' des bibliothèques système de Gambit. Il est a noté que Gambit permet de redéfinir l'emplacement des répertoires symboliques par l'utilisation de l'option d'exécution '`-:~~NAME=DIRECTORY`'.

3.2. La forme `##namespace`

Un identifiant est un symbole, utilisé dans le code, qui n'est pas sous un `quote`. Par exemple `pair?` est l'identifiant de la procédure qui teste que le type de donnée est une paire. Il est généralement lié à une valeur ou une macro. Un symbole est un type de donnée primitif de Scheme.

Les espaces de nom sont gérés avec une forme spéciale propre à Gambit. Cette forme se nomme `##namespace` et permet de lier des identifiants à d'autres identifiants dans le code. Cela affecte l'ensemble des identifiants spécifiés dans la forme `##namespace`, l'ensemble vide correspond à tous les identifiants. Cette forme s'applique uniquement aux identifiants de variable et de macro. Elle ne s'applique pas aux valeurs. Cette forme primitive est présente dans Gambit depuis longtemps. Un espace de nom se compose de n'importe quelle séquence de caractère terminé par un `#`. Il y a seulement l'espace de nom vide qui est une exception à cette règle et c'est l'espace de nom par défaut. Les associations de symboles données par la forme `##namespace` respectent la portée lexicale. Il y a trois types d'opérations avec les espaces de nom.

Il y a la déclaration d'espaces de nom global qui s'applique à tous les symboles qui ne contiennent pas de `#`.

```
(##namespace ("<ns>"))
;; <symbol-name> => <ns><symbol-name>
```

Fig. 3.1. Namespace Global

Il est possible de spécifier la liste des symboles qui sont affectés par la déclaration d'espace de nom. À partir de la syntaxe de la figure 3.1, il suffit d'ajouter les symboles après le nom de l'espace de nom.

```
(##namespace ("<ns>" A B ...))
;; A => <ns>A
;; B => <ns>B
;; ...
```

Fig. 3.2. Namespace Set

La forme `##namespace` permet aussi de lier un identifiant à un autre identifiant dans un espace de nom donné. Chaque association est marquée par une paire qui crée un alias entre le premier élément et le second. Par exemple, la paire `(<old> <new>)` remplace `<old>` par `<ns><new>`.

```
(##namespace ("<ns>" (<old> <new>) ...))
;; <old> => <ns><new>
;; ...
```

Fig. 3.3. Namespace Rename

Cette forme est utilisée pour créer un espace distinct pour chaque module. Cela permet d'éviter les conflits de nom entre les identifiants des modules. Chaque module commence par déclarer son espace de nom suivi des définitions des procédures du module. Les différentes formes d'espace de noms sont données par les figures 3.1, 3.2 et 3.3.

L'exemple 3.4 est un exemple d'utilisation de la forme `##namespace` pour créer un espace pour le module `hello`. La procédure `hi` est dans l'espace de nom `hello#`.

```
;; hello.scm
(##namespace ("hello#" hi))
(define (hi)
  (display "Hello, world!\n"))
(hi)
```

Fig. 3.4. Module Hello

3.3. La forme ##demand-module et ##supply-module

Le mécanisme de chargement des modules est géré par la forme spéciale `##demand-module`. Cette forme indique au système de charger un module s'il n'est pas déjà chargé. Cette forme gère le chargement multiple d'un module. Elle est utilisée pour importer la liste des modules requis par le module courant. Le fonctionnement de cette forme est similaire à la procédure `load` avec quelques différences. La forme spécial `##demand-module` génère une expression vide. L'effet de cette forme agit après la phase d'expansion des macros. Le paramètre passé à `##demand-module` doit être un symbole qui correspond au nom du module. La procédure `load` requiert le chemin complet vers le fichier à charger.

Il est à noter que l'ordre dans lequel les `##demand-module` apparaissent correspond à l'ordre dont les modules sont visités. Cette forme est permise partout où une définition de macro est permise.

Nous avons ajouté un forme spéciale conjointe au `##demand-module` pour indique le nom symbolique des modules des modules exportés. Cette forme spéciale est `##supply-module`, elle accepte comme paramètre le nom du module exporté par l'entité courant. La syntaxe de ces deux formes dans la figure 3.5.

```
(##demand-module <module-ref>)
(##supply-module <module-ref>)
```

Fig. 3.5. Syntaxe des formes `demand-module` et `supply-module`

3.3.1. Les méta informations

Il est utile d'attacher à un module des informations qui sont accessibles lors de l'expansion et même la compilation. Ces informations sont spécifiées par la forme `##meta-info`. Cette

forme accepte au moins un paramètre qui correspond au nom de la méta information, le reste des paramètres est la valeur associée à la méta donnée.

```
(##meta-info <name> <value>)
(##meta-info <name> <value> ...)
```

Les méta informations sont utilisées pour donner des paramètres de compilation du module. Les différentes méta informations sont `cc-options`, `ld-options`, `ld-options-prelude`, `pkg-config` et `pkg-config-path`. Ces méta informations ne sont utiles que pour les modules compilés.

- Les `cc-options` sont ajoutés aux options de la commande qui invoque le compilateur C.
- Les méta informations `ld-options` et `ld-options-prelude` composent les paramètres de la commande qui invoque l'éditeur de lien. Les paramètres dans `ld-options-prelude` précèdent ceux qui sont dans `ld-options`.
- `pkg-config` contient le nom des bibliothèques C à être liées au module Scheme. Les options nécessaires pour le compilateur C sont déterminées automatiquement par l'utilitaire `pkg-config`.
- `pkg-config-path` ajoute des répertoires à la variable d'environnement `PKG_CONFIG_PATH` qui est utilisée par l'utilitaire `pkg-config`.

3.4. Implémentation des modules primitifs

Un module primitif est généralement constitué d'un fichier d'entête avec la déclaration de l'espace de nom et les définitions de macros et un fichier contenant les procédures. Dans Gambit les fichiers d'entête sont marqués par un `#` juste avant l'extension, tel que `angle2/angle2#.scm`.

- *<name>*`#.scm` est la structure du nom fichier d'entête. Ce fichier contient des déclarations d'espace de nom et des définitions de macros.
- *<name>*`.scm` est la structure du nom du fichier qui contient les procédures du module.

Le nom des fichiers doit correspondre à la dernière partie du nom de module. Par exemple, le module primitif `angle2` doit inclure les fichiers `angle2/angle2.scm` et généralement `angle2/angle2#.scm`.

```
;; angle2/angle2#.scm
(##namespace ("angle2#" deg->rad rad->deg))
```

```
;; angle2/angle2.scm
(include "angle2#.scm")
(##namespace ("angle2#" factor))
(##supply-module angle2)
(define factor (/ (atan 1) 45))
(define (deg->rad x) (* x factor))
(define (rad->deg x) (/ x factor))
```

Fig. 3.6. Écriture d'un module qui implémente des fonctions de conversions entre les angles en degrés et en radian. Ce module est séparé en 2 fichiers. Le fichier `angle2/angle2#.scm` contient les exportations et `angle2/angle2.scm` contient l'implémentation des fonctions.

Dans l'exemple de la figure 3.6, il y a dans `angle2/angle2.scm` l'inclusion du fichier d'entête `angle2/angle2#.scm` qui ajoute une déclaration redondante de l'espace de nom dans ce cas. La déclaration `(##namespace ("angle2#"))` implique l'espace de nom ajouté par l'inclusion du fichier d'entête. Il est possible que l'espace de nom déclaré dans `angle2/angle2#.scm` soit différent de celui utilisé dans `angle2/angle2.scm`.

La forme `##namespace` dans l'exemple 3.6 s'applique aux identifiants suivants:

```
factor    --> angle2#factor
deg->rad  --> angle2#deg->rad
rad->deg  --> angle2#rad->deg
```

3.4.1. La forme `##import`

L'importation des modules est effectuée par la forme `##import` qui effectue deux actions, l'inclusion du fichier `<name>#.scm` et un chargement des définitions. La forme `##import`, comme `##demand-module` s'occupe de trouver l'emplacement du fichier d'entête à partir du nom du module. Elle génère le `##include` du fichier d'entête s'il existe et un

`##demand-module` du module. L'importation `(##import angle2)` est équivalente à:

```
(##include "/un/chemin/angle2/angle2#.scm")
(##demand-module angle2)
```

Fig. 3.7. Expansion de `(##import angle2)`

3.5. Implémentation des modules R7RS

Pour que le système de module soit compatible avec d'autres implémentations de Scheme, les modules de haut niveau sont définis dans le standard R7RS Small [**30**]. Les modules sont définis par la forme `define-library` dont la syntaxe est donnée par la figure 2.7. Les formes `define-library` et `import` sont expansées dans les formes spéciales utilisées par les modules primitifs. L'élément qui distingue un module primitif et un module R7RS est donc l'utilisation de la forme `define-library`.

3.5.1. Expansion du `import`

L'expansion de la forme `import` dépend du type de module qui est importé. L'importation d'un module primitif est différente de l'importation d'un module R7RS. Gambit permet l'importation d'un module primitif en utilisant la même forme que pour les modules R7RS. Les capacités du `import` dépendent de sa provenance, s'il est dans un `define-library` ou dans un programme principal. Dans le cas d'un `define-library` le `import` supporte l'importation relative, qui est une extension de Gambit.

3.5.1.1. *Importation d'un module primitif*

L'importation d'un module primitif limite la syntaxe du `import`. Il n'est pas possible d'utiliser les extensions `only`, `except` et `rename` sur un module primitif. Pour utiliser ces extensions, il faut les métadonnées que les modules R7RS offrent. La structure modules primitifs ne contient pas de déclaration qui indique les identifiants uniformes. Pour avoir l'ensemble des identifiants qu'un module primitif il faudrait analyser l'ensemble des fichiers du module primitif pour y extraire les identifiants qui sont exportés. Nous avons choisi la simplicité.

Le `import` R7RS se rabat sur le `##import` des modules primitifs qui ne supporte pas les extensions R7RS. C'est une extension que nous avons ajouté dans notre implémentation des modules R7RS pour permet l'utilisation des modules primitifs dans un contexte R7RS.

Les modules R7RS ont tous la déclaration `export` qui donne l'ensemble des identifiants qui sont exportés. Les modules primitifs sont plus proches de R5RS avec un mécanisme de chargement sophistiqué. Le `import` requiert la méta information fournit par la déclaration `export` dans le `define-library`.

Les modules primitifs font le pont entre R5RS et R7RS.

```
;; expansion of (import (termite))
(##import termite)
```

Fig. 3.8. Expansion du `import` d'un module primitif

3.5.1.2. *Importation d'un module R7RS*

L'importation d'un module R7RS est expansée en au plus trois parties.

- Un `##demand-module` qui s'occupe de charger l'implémentation des procédures du module.
- Une déclaration d'espace de nom qui donne accès aux identifiants que le module exporte.
- L'implémentation des macros qui sont exportées par le module.

L'instruction de chargement du module est générée dans tous les cas qu'un module définit des procédures. Un module qui ne définit que des macros ne nécessite pas d'être chargé durant l'exécution seulement dans le contexte d'expansion des macros. L'importation d'un module R7RS qui ne contient qu'une déclaration `export` ne nécessite pas d'être chargé durant l'exécution. Ce type de module est utilisé pour exporter les fonctionnalités déjà implémentées dans Gambit dans un contexte R7RS.

La forme utilisée pour rendre disponible l'ensemble des identifiants importés est `##namespace`. L'ensemble des identifiants importés dépend de la forme du `import`. Par défaut, tous les identifiants exportés par le module sont importés. Les opérateurs `only` et `except` affectent le nombre d'identifiants importés. Les opérateurs `prefix` et `rename` affectent le nom des identifiants. Dans l'exemple 3.9, l'importation inclut l'ensemble des

identifiants exportés par le module. L'ensemble des formes `##namespace` générées par un `import` est donné par la figure 3.10.

```
;; expansion of (import (github.com/gambit/hello))
(##demand-module github.com/gambit/hello)
(##namespace ("github.com/gambit/hello#" hi salut))
;; macros
```

Fig. 3.9. L'exemple de l'expansion du `import` du module R7RS `github.com/gambit/hello` qui exporte les procédures `hi` et `salut`.

```
;; (import (only (github.com/gambit/hello) hi))
(##namespace ("github.com/gambit/hello#" hi))

;; (import (except (github.com/gambit/hello) hi))
(##namespace ("github.com/gambit/hello#" salut))

;; (import (prefix (github.com/gambit/hello) m-))
(##demand-module github.com/gambit/hello)
(##namespace ("github.com/gambit/hello#" (m-hi hi) (m-salut salut)))

;; (import (rename (github.com/gambit/hello) (hi howdy)))
(##namespace ("github.com/gambit/hello#" (howdy hi) salut))
```

Fig. 3.10. Différents `##namespace` générés par l'expansion du `import` d'un module R7RS.

,

3.5.2. Expansion du `define-library`

La forme `define-library` est expansée dans les formes qui composent un module primitif. Chacune des déclarations de la bibliothèque est utilisée dans l'expansion du `define-library`. La déclaration d'exportation est valide si tous les identifiants exportés sont distincts. Une déclaration `export` qui exporte un identifiant plusieurs fois cause une erreur de syntaxe. Les informations sur les identifiants exportés ne sont pas utilisés lors de l'expansion du `define-library` (figure 3.11), mais lors de l'importation de cette bibliothèque. Les déclarations `import` sont expansées de la même façon que dans des programmes principaux.

```
(define-library (hello)
  (import (scheme base) (scheme write))
  (export hi salut)
  (begin
    (define (exclaim msg1 msg2)
      (display msg1)
      (display msg2)
      (display "!\n"))
    (define (hi name) (exclaim "hello " name))
    (define (salut name) (exclaim "bonjour " name))
    ;; it is best for a library to not have side-effects...
    #;(salut "le monde")))
```

```
;; expansion of (define-library (hello) ...)
(##declare (block))
(##supply-module github.com/gambit/hello)
(##namespace ("github.com/gambit/hello#"))
(##namespace ("" define ...))
(##namespace ("" write-shared write display write-simple))
(define (exclaim msg1 msg2)
    (display msg1) (display msg2) (display "!\n"))
(define (hi name) (exclaim "hello " name))
(define (salut name) (exclaim "bonjour " name))
(##namespace (""))
```

Fig. 3.11. Le code source du module `github.com/gambit/hello` et son expansion.

3.5.2.1. *Extensions de Gambit*

Gambit offre des extensions au `define-library` et au `import`. L'importation dans le contexte d'une bibliothèque peut être relative au module courant. Plusieurs déclarations supplémentaires ont été ajoutées dans la forme `define-library`.

- `namespace`
- `cc-options`
- `ld-options` et `ld-options-prelude`
- `pkg-config` et `pkg-config-path`

La figure 3.12 est un exemple d'importation relative. L'importation relative part du `module-ref` du module courant. Un `import` de `(.. C)` à partir du module `(A B)`

correspond à l'importation de `(A C)`. Cela permet au sous-module de tests unitaires de référer au module principal en préservant le `module-ref` avec la version.

```
(define-library (A B)
  (import (.. C))  ;; => (import (A C))
  (import (..))) ;; => (import (A))
```

Fig. 3.12. Exemple d'importation relative du module

La déclaration `namespace` permet de forcer l'espace de nom d'un module. L'utilisation primaire de cette déclaration est l'implémentation de modules qui exporte les fonctionnalités déjà implémentées dans Gambit.

```
(define-library (scheme case-lambda)
  (namespace "")
  (export
case-lambda
))
```

Fig. 3.13. Implémentation de la bibliothèque système `(scheme case-lambda)`.

Les déclarations `cc-options`, `ld-options`, `ld-options-prelude`, `pkg-config` et `pkg-config-path` permet d'ajouter des éléments dans les méta informations respectifs.

3.6. Conclusion

Nous avons ajouté la notion de module dans Gambit pour résoudre le manque de modularité dans le langage Termite. Les formes que nous avons ajoutées sont: `##demand-module`, `##supply-module`, `##import`, `import` et `define-library`. Notre implémentation des modules utilise une forme compatible avec les autres implémentations de Scheme R7RS. En plus, les extensions que nous avons ajoutées offrent une interface avec les modules système de Gambit. L'implémentation actuelle des modules est suffisante pour permettre la diffusion de code, car elle offre des identifiants uniques pour les modules.

Notre approche supporte la création de modules qui dépendent de bibliothèques du système d'exploitation. Cela est réalisé par les extensions de la forme `define-library` et par la FFI de Gambit. Le système de module permet de regrouper de façon logique les fonctionnalités dans un module.

Chapitre 4

Modèle de chargement

Ce chapitre traite des mécanismes de chargement que nous avons implémenté pour le système de module. Dans notre implémentation, les modules sont référés par un nom symbolique plutôt que leur emplacement sur le système de fichier. Elle garantit l'ordre dans lequel ils sont chargés et aussi le chargement unique de chaque module. Pour permettre la diffusion des modules, nous avons ajouté un mécanisme d'installation automatique et de chargement des modules à la volée.

Un système est composé d'un ensemble de modules qui interagissent entre eux. L'interaction entre les modules est dans un contexte statique ou dynamique. Dans le contexte statique, les modules sont incorporés au sein de l'application, ils n'ont pas besoin d'être chargés. Dans le contexte dynamique, les modules sont externes à l'application et requièrent un chargement dynamique. Le chargement dynamique de modules est une partie importante dans un système de modules, il permet l'utilisation de modules externes.

La diffusion des modules requiert que le chargement de modules qui ne sont pas encore installés soit effectué durant l'exécution.

4.1. Chargement des bibliothèques

Le chargement d'une bibliothèque Scheme (ou module) dans Gambit est séparé en plusieurs niveaux. Il y a la phase de recherche du module et de ses dépendances qui valide la présence, sur le système de fichier ou en mémoire, de tous les modules nécessaires. Ensuite, le module et ses dépendances sont chargés dans un ordre qui respecte les relations de dépendance. Un module est soit sur le système de fichier ou déjà en mémoire. L'emplacement

des bibliothèques sur le système de fichier est lié par défaut aux chemins spécifiés par le `##module-search-order` qui a comme défaut `~~userlib` et `~~lib`.

La procédure exacte de chargement des bibliothèques par `import` n'est pas spécifiée par le standard R7RS. Le standard spécifie seulement une syntaxe de base et le comportement principal qui est requis. L'importation d'une bibliothèque doit charger la bibliothèque et rendre ses fonctionnalités disponibles dans le contexte où l'importation a eu lieu (qui peut soit venir d'un programme principal ou d'une bibliothèque).

Le chargement d'une bibliothèque peut-être effectué à l'exécution par l'utilisation de `eval` (par `load`) pour les fichiers source et `load-object-file` pour les bibliothèques compilées. Cette recherche peut aussi avoir lieu durant l'édition des liens en utilisant les méta-infos contenues dans les fichiers **.c** qui sont chacun compilés par le compilateur C en **.o** et lié par l'éditeur de lien.

4.2. Modèle statique

Le modèle de bibliothèque statique consiste à intégrer la bibliothèque dans le fichier exécutable de l'application finale. La bibliothèque n'a donc pas besoin d'être chargée durant l'exécution. L'avantage du modèle statique est sur le plan du déploiement. Puisque toutes les dépendances sont dans l'application finale, il suffit de distribuer celui-ci. Le compilateur Gambit supporte un modèle statique pour des programmes simples. Compiler une application liée statiquement est possible de façon manuelle. Pour lier statiquement deux fichiers Scheme simples, il suffit d'invoquer le `gsc` comme suit:

```
$ gsc -exe file1.scm file2.scm
```

Un des problèmes du modèle statique est le coût lié à la maintenance. Les programmes qui utilisent des bibliothèques statiques ne permettent pas une construction modulaire. La mise à niveau d'une des bibliothèques statiques nécessite la recompilation du programme au complet. En plus, cela n'est pas adapté pour des applications évolutives qui peuvent être étendues par l'utilisateur. Une solution qui a été adoptée est le modèle dynamique, présenté dans la prochaine section. Cela offre une plus grande flexibilité dans le déploiement des programmes.

4.3. Modèle dynamique

Dans le modèle dynamique, chaque module externe est séparé durant l'exécution. L'application contient les informations pour accéder aux fonctionnalités des modules durant l'exécution. Le déploiement d'une application nécessite la distribution de toutes les dépendances directes et indirectes. Les bibliothèques partagées offrent plusieurs avantages par rapport aux bibliothèques statiques.

Dans ce modèle les bibliothèques sont liées au programme durant l'exécution. Cela nécessite que les bibliothèques soient organisées sur le système de fichier d'une façon distinguable. Chaque module doit posséder un nom unique qui permet d'y référer. Ce nom unique va être utilisé lors de la collection des dépendances.

La recherche des bibliothèques est effectuée dans un ordre spécifique indépendant de la spécification. L'algorithme de recherche des bibliothèques prend en paramètre le nom de la bibliothèque et retourne le chemin absolu correspondant à son emplacement dans l'arborescence du système de fichier. Les bibliothèques sont situées dans différents répertoires: le répertoire des bibliothèques système (`~~lib`) et le répertoire des bibliothèques utilisateur (`~~userlib`) ou des répertoires explicitement ajoutés au «module search order».

Chaque module possède trois niveaux d'initialisation dans le système numérotés de 0 à 2. Le niveau 0 indique que le module est collecté en mémoire, mais non initialisé. Le niveau 1 indique que l'initialisation de bas-niveau a été complétée et que le descripteur Scheme du module a été récupéré. Le niveau 2 marque les modules chargés dont toutes les définitions et expressions au niveau principales («top level») ont été evaluées et donc le module est prêt à être utilisé.

Dans l'exemple à la figure 4.2, l'exécution du module principal **main.scm** déclenche la collection des modules X et Y, qui récursivement déclenche la collection de W et Z. L'algorithme de collection des modules gère les modules qui apparaissent plusieurs fois au sein du graphe et les cycles. Après la collection de tous ces modules, le descripteur de module est récupéré par un appel aux primitives du système d'exploitation si compilé.

```
> (define mods (##collect-modules '(_zlib)))
> mods
(#(_digest 0 #(#(_digest) #() () 1 #<procedure #2> ...))
 #(_zlib 0 #(#(_zlib) #(_digest) () 1 #<procedure #4> ...)))
> (##init-modules mods)
#t
> ##registered-modules
(#(_digest 2 #(#(_digest) #() () 1 #<procedure #2 _digest#> ...))
 #(_zlib 2 #(#(_zlib) #(_digest) () 1 #<procedure #4 _zlib#> ...))
 ...)
```

Fig. 4.1. Un exemple qui montre la collection des modules à partir du module `_zlib` suivit de l'initialisation des modules collectés. La collection des modules est effectuée par la procédure `##collect-modules`. L'ensemble des modules retournés sont initialisés par la procédure `##init-modules`. Les enregistrements des modules `_zlib` et `_digest` ont été ajoutés à la liste des modules enregistrés (variable `##registered-modules`).

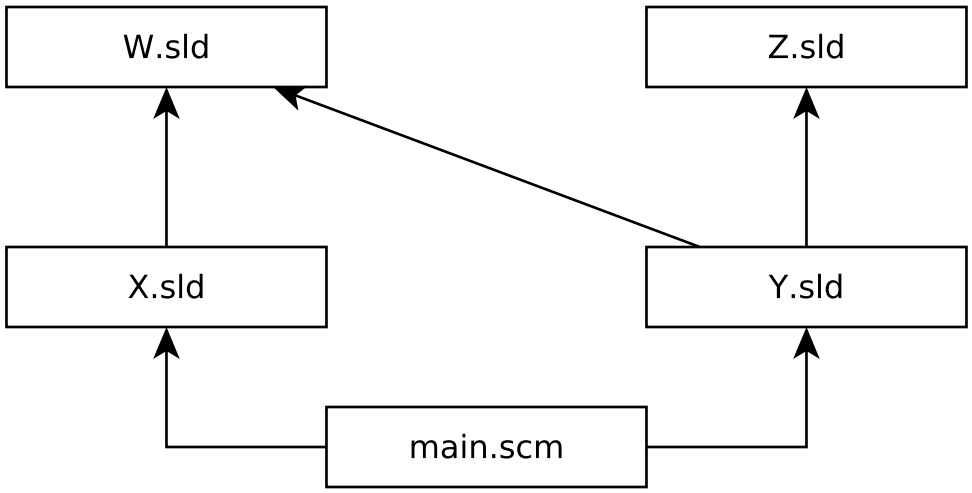


Fig. 4.2. Un exemple d'un système fictif composé de différents modules. Le module principal se nomme **main.scm** avec l'extension **.scm** et les bibliothèques ont l'extension **.sld**

4.3.1. Module compilé dans Gambit

Les modules dans Gambit peuvent être compilés pour plus de performance en bibliothèque dynamique. Les programmes principaux peuvent être compilés en exécutable. Gambit permet l'utilisation de bibliothèques statiques et dynamiques.

```
;; fib.scm
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Fig. 4.3. Un module qui implémente la fonction mathématique `fib` au niveau principal («top level»).

La construction d'une bibliothèque dynamique à partir du fichier `fib.scm` de la figure 4.3 s'effectue par le compilateur de Gambit qui se nomme `gsc`. Cela produit un fichier avec l'extension `.oN` où le `N` correspond au numéro de séquence générée de la bibliothèque, qui commence à 1. Gambit supporte déjà une certaine gestion des versions des modules compilés avant l'intégration du système de module.

4.4. Module hébergé

Un module qui est hébergé est un module dont le code source est stocké sur un serveur de versionnement de source accessible sur le réseau au nœud d'un nom de domaine comme `github.com`. La syntaxe des noms de domaine est inspirée du RFC-2396 [**3**]. La différence avec la spécification du *hostname* dans le RFC-2396 est que le *hostname* ne peut pas se terminer par un point et doit contenir au moins un `domainlabel`. C'est pour permettre de distinguer un module local et un module hébergé.

```
hostname      = ( domainlabel "." )+ toplabel
domainlabel   = alphanum | alphanum ( alphanum | "-" )* alphanum
toplabel      = alpha | alpha ( alphanum | "-" )* alphanum
alphanum      = alpha | digit
alpha         = [a-zA-Z]
digit         = [0-9]
```

Fig. 4.4. Grammaire EBNF représentant un hostname selon un sous-ensemble du RFC-2396.

4.4.1. Installation automatique

Gambit permet l'installation automatique des modules. Pour contrôler les provenances des modules qui sont installés automatiquement, nous avons ajouté une liste des emplacements de confiance à l'environnement d'exécution qui indique quels modules peuvent être installés inconditionnellement. Nous avons nommé cette liste *whitelist*, elle valide les *<module-ref>*. La validation consiste à trouver un élément dans la *whitelist* qui correspond à un préfixe du *<module-ref>*. Chaque processus Gambit possède une *whitelist* différente, cela implique que chaque nœud Termite peut avoir une *whitelist* distinct.

Un élément est considéré un préfixe, si chacune des composantes de l'élément est incluse dans le *<module-ref>* dans le même ordre. L'élément `github.com/gambit` est un préfixe de `github.com/gambit/hello`, car les parties `github.com` et `gambit` font partie du *<module-ref>*. Le module `github.com/gambitXYZ/hello` ne contient pas le préfixe `github.com/gambit`, car `gambit` est différent de `gambitXYZ`.

Par défaut l'emplacement `github.com/gambit` est inclus dans la *whitelist*, puisque cela correspond à la emplacement qui de des sources de Gambit. Cela implique que tous les modules sous `github.com/gambit` peuvent être installés automatiquement. Par exemple, le module `github.com/gambit/hello` peut être installé inconditionnellement. La *whitelist* peut être modifiée par les variables d'environnement et par les arguments de ligne de commande.

En plus de la *whitelist* nous avons aussi ajouté un mode d'installation. qui indique si l'installation demande une confirmation à l'usager pour l'installation d'un module qui n'est pas dans la *whitelist*. Il y a trois modes d'installation.

```
$ gsi -e '(import (github.com/feeley/pyffi))'
*** ERROR IN (stdin)@1.9 -- Cannot find library (github.com/feeley/pyffi)

$ gsi -:whitelist=github.com/feeley -e '(import (github.com/feeley/pyffi))'

$ gsi -:whitelist= -e '(import (github.com/gambit/hello))'
*** ERROR IN (stdin)@1.9 -- Cannot find library (github.com/gambit/hello)
```

Fig. 4.5. Exemples de manipulation de la *whitelist* par les arguments de la ligne de commande. Le premier exemple montre le rejet de l'installation, car le module n'est pas dans la *whitelist*. Le second exemple permet les modules sous `github.com/feeley` d'être installé automatiquement. Le dernier exemple montre comment désactiver l'installation automatique en vidant la whitelist.

- **never**: il y a seulement les modules qui sont dans la *whitelist* qui peuvent être installés.
- **repl** (*Read Eval Print Loop*): les modules qui ne sont pas dans la *whitelist* peuvent être installés avec la confirmation textuelle de l'usager s'il y a une *repl* disponible. C'est le mode par défaut.
- **ask-always**: les modules qui ne sont pas dans la *whitelist* peuvent être installés avec la confirmation textuelle de l'usager.

Le mode d'installation est spécifié par le *runtime option* nommée *ask-install*.

```
$ gsi -:ask-install=always
Gambit v4.9.3

> (import (github.com/frederichamel/semver))
Hosted module github.com/frederichamel/semver is required but is not installed.
Download and install (y/n)? n
*** ERROR IN (stdin)@1.9 -- Cannot find library (github.com/frederichamel/semver)
>
```

Fig. 4.6. L'exemple montre le résultat d'une réponse négative lors de l'installation du module qui n'est pas dans la *whitelist*. Il ne se fait pas installer.

4.5. Conclusion

Ce chapitre a traité des méthodes de chargement de module et des mécanismes d'installation automatique qui sont requis dans la diffusion des modules. Notre implémentation

du chargement des modules garantit le chargement unique des modules dans un ordre qui respecte les dépendances. Ceci est demandé dans l'implémentation des modules Scheme R7RS.

Notre approche nous permet l'installation automatique de modules qui sont inclus dans la *whitelist* lorsque demandé. Cela offre une forme simple de contrôle de sécurité sur les modules qui peuvent s'installer automatiquement sur un nœud du système distribué. Cela offre la diffusion des modules entre les nœuds d'un tel système pour permettre des appels RPC sur un nœud ne connaissant pas le code demandé.

Chapitre 5

Gestion des modules

Ce chapitre décrit l'organisation des modules sur le système de fichier que nous avons choisi pour permettre plusieurs versions d'un module. Notre approche est d'associer à chaque version d'un module un répertoire différent. Cela permet de stocker les différentes versions d'un module dans le système de fichier. Nous avons comparé les différentes façons d'organiser les modules dans plusieurs langages comme Python, JavaScript, Go, OCaml. Il est important que l'installation d'un module n'altère pas le bon fonctionnement des autres modules. Le modèle de gestion des modules choisi garantit que les dépendances d'un module sont fixes.

5.1. Sommaire

La gestion des modules inclut généralement l'installation, la mise à jour et la désinstallation. L'organisation des modules est un élément important dans la gestion des modules. Les gestionnaires de modules sont présents dans beaucoup de langages tels que Python, Ruby, JavaScript, Common Lisp, Go, etc. Un gestionnaire de module est inclus dans le système Gambit Scheme pour organiser les modules.

Le gestionnaire de module de Gambit Scheme fournit les opérations d'installation, de désinstallation, de mise à jour, de compilation d'un module et l'exécution des tests unitaires du module. Les modules sont versionnés par Git. L'emplacement des modules est spécifié par une liste de répertoires qui inclut les modules système et les modules utilisateur. La gestion des modules est effectuée par le module `_pkg` qui offre les procédures d'installation et de désinstallation.

5.2. Organisation des modules

Les modules sont organisés dans des répertoires donnés par le système Gambit. Ils contiennent l'ensemble des modules internes et actuellement installés sur le système. Il y a trois principaux répertoires spéciaux qui contiennent des modules.

- `~~userlib`: c'est le répertoire qui contient les modules de l'utilisateur, par défaut `.gambit_userlib` dans le répertoire maison de l'utilisateur.
- `~~lib`: c'est le répertoire d'installation de Gambit qui contient les modules système. Les modules dans ce répertoire sont communs à tous les utilisateurs.
- `~~instlib`: c'est le répertoire d'installation des modules. Par défaut, il correspond au répertoire `~~userlib`.

Un module local ou hébergé est identifié de façon unique par un `module-ref`. Un `module-ref` est séparé en trois parties: le `hostname`, le `path` et l'étiquette qui donne la version. La différence entre une référence à un module local et hébergé est la première partie. Dans le cas hébergé, le champ `hostname` contient l'information du nom de domaine qui, dans le cas local, est vide. Le `<tag>` spécifie la version du module avec une référence à un commit du système de révision Git. Un `<tag>` vide réfère à la version de développement du module (à éviter pour le déploiement final). La syntaxe du nom de domaine est donnée par la grammaire à la figure 4.4. La grammaire formelle à la figure 5.1 décrit la syntaxe du `<module-ref>` dans le cas hébergé et local.

```
<module-ref>       := <local> | <hosted>
<local-package>    := <id>
<hosted-package>   := <hostname>/<id>/<id>
<local-module>     := <local-package>(/<id>)*(@<tag>)?
<hosted-module>    := <hosted-package>(/<id>)*(@<tag>)?
```

Fig. 5.1. Grammaire formelle d'un `module-ref`

La règle `<id>` est liée à un élément de l'URL vers le dépôt de code. Par exemple, le `module-ref github.com/gambit/hello` réfère au module `hello` sur le serveur de révision `github.com` dans l'espace `gambit`, les `<id>` sont respectivement `gambit` et `hello`. Le champ `host` est dans ce cas `github.com/gambit` qui correspond au nom de domaine avec le nom de l'espace sur le serveur. Les deux `<id>` dans la syntaxe d'un module hébergé correspond au

nom utilisateur ou de l'organisation (par exemple `gambit`) et au nom du dépôt sur le serveur de version.

5.2.1. Installation des packages et des modules

Les modules sont versionnés par un système de gestion de version pour permettre de différencier et conserver chaque version du module. Un *package* est un regroupement de modules. L'installation d'un package est faite à partir d'un serveur de révision ou un répertoire local qui est versionné par un système de gestion de versions (git, hg, svn). La syntaxe d'un *package* local est donné par la règle `<local-package>` de la figure 5.1 et la syntaxe d'un *package* hébergé est donné par la règle `<hosted-module>`. Chaque *package* est associé à un répertoire unique.

Les *packages* déployés sont généralement hébergés sur des serveurs de version tels que github, gitlab, bitbucket, etc. Une version d'un module est installée à partir d'un package dans un répertoire distinct. Il est donc possible d'avoir plusieurs versions d'un module installées sur le système de fichier.

L'installation d'un *package* est faite par le module `_vcs` qui offre une interface uniforme aux outils de gestion de versions tels que `git`, `hg` et `svn`. Le processus d'installation est séparé en plusieurs étapes.

(1) Le *package* est téléchargé dans un répertoire temporaire.

(2) Le répertoire temporaire est renommé au répertoire prévu de façon atomique.

Cette procédure d'installation garantit qu'un seul processus d'installation peut installer ce package.

Les modules sont installés en même temps que le *package* qui les contient. La version des modules correspond à la version du *package*. Chacun des modules est contenu dans un sous répertoire du *package*.

Tout d'abord, la branche principale du dépôt du *package* est clonée dans un répertoire unique dans le préfixe d'installation qui est par défaut `~~userlib`. L'installation d'un module se fait par la copie de ses fichiers dans le *package* dans un dossier temporaire qui est renommé dans un dossier `@<version>` dans le répertoire du *package*. Le répertoire `@` est utilisé comme version de développement qui change avec les mises à jour du *package*. Plusieurs versions d'un module peuvent coexister dans un même préfixe d'installation, puisque chaque version

est associée à un répertoire différent. La version d'un module spécifié par une étiquette au sein du système de gestion de version.

Il est possible d'installer un module par une procédure dans le langage. La procédure `install` du module interne `_pkg` permet l'installation de module, elle accepte deux paramètres: le nom du module et de façon optionnelle le préfixe d'installation. Le module est installé par l'installation du *package* englobant. Elle retourne la valeur de vérité vraie (`#t`) si l'installation réussit, sinon elle retourne la valeur de vérité fausse (`#f`).

```
(install mod #!optional dir)
```

L'installation des *package* peut aussi s'effectuer par l'invocation de l'interprète Gambit avec l'option `-install`. Cette option requiert le nom du package. Le répertoire d'installation optionnel est spécifié par l'option `-dir`. Il est possible de spécifier un répertoire local où le *package* est situé. Un paramètre est reconnu comme un répertoire local, s'il termine soit par une barre oblique (`/`) ou par un point.

```
$ gsi -install [-dir <path>] [local-dir] package
```

Le répertoire d'installation `<path>` est la racine utilisée pour installer les modules et est spécifié par l'option `-dir`. La racine par défaut est `~~userlib`. Voici un exemple d'installation d'une version spécifique du module `semver` qui implémente la logique du *semantic versioning* [**28**] qui est une façon d'organiser les versions. Cette organisation est recommandée pour étiqueter les versions. La présente version de système de module ne requiert pas l'utilisation du *semantic versioning*. La figure 5.2 montrent les différentes façons d'installer d'un *package* en invoquant de l'interprète Gambit.

```
$ gsi -install github.com/frederichamel/semver

$ gsi -install -dir /tmp/exemple github.com/frederichamel/semver

$ gsi -install -dir /tmp/exemple /local/dir/ github.com/frederichamel/semver
```

Fig. 5.2. Exemples d'installation d'un *package* hébergé dans le préfixe d'installation par défaut, dans un répertoire spécifique et dans un répertoire spécifique à partir d'un répertoire local.

```
$ gsi -install -dir /tmp/exemple github.com/frederichamel/semver
```

Fig. 5.3. Exemple d'installation d'un *package* hébergé dans le préfixe d'installation `/tmp/exemple`

```
$ gsi -install -dir /tmp/exemple /local/dir/ github.com/frederichamel/semver
```

Fig. 5.4. Exemple d'installation d'un *package* dans le préfixe d'installation `/tmp/exemple` à partir du répertoire local `/local/dir/`.

5.3. Désinstallation

Il est possible de désinstaller une version d'un module spécifique où le *package* au complet. La désinstallation d'un module consiste à supprimer les fichiers de ce module. Le module `_pkg` offre la procédure `uninstall` qui accepte deux arguments: le nom du module, et de façon optionnelle, le répertoire dans lequel les modules sont situés. Les valeurs retournées par cette procédure sont similaires à la procédure `install`.

```
(uninstall mod #!optional dir)
```

La désinstallation peut être faite en passant l'option `-uninstall` à l'interprète Gambit. Cette option requiert le nom du module et le répertoire des modules à désinstaller.

```
$ gsi -uninstall [-dir <path>] (<module>|<package>)
```

Le répertoire `<path>` est l'emplacement des modules à désinstaller. Le format des arguments pour la désinstallation est le même que pour l'installation. Le répertoire par défaut est le répertoire `~~userlib`.

La désinstallation du module `semver` qui est installé dans le répertoire `/tmp/exemple` est fait par la commande suivante:

```
$ gsi -uninstall -dir /tmp/exemple github.com/frederichamel/semver
```

5.4. Mise à jour

Cette opération actualise le *package* courant. Cela donne accès aux nouvelles publications d'un module. Le répertoire `@` est effacé et remplacé par le dernier état de la branche de développement. Pour installer une nouvelle version d'un module, il suffit de faire la mise à jour de la branche master et d'installer la nouvelle version. La mise à jour d'un module

n'affecte pas les versions spécifiques, comme la version 1.0. Cette opération est utilisée pour synchroniser le dépôt local avec le dépôt web. Sans cette opération l'installation de versions plus récente échouerait quand ils ne sont pas dans la cache locale du dépôt. La mise à jour d'un module est l'équivalent d'un `pull` dans git.

Notre système de module permet l'installation d'une branche. L'opération de mise à jour sur cette branche n'est pas encore supportée. C'est un aspect qui est à explorer.

```
$ gsi -update [-dir <path>] <package>
```

5.5. Tests unitaires

Les tests unitaires exécutés sont dans un fichier conjoint au module. Gambit offre un module de test unitaire nommé `_test`. Il contient plusieurs procédures et macros pour tester le bon fonctionnement d'un module. Les tests unitaires pour un module nommé `A` sont par convention dans le sous-module `A/test`.

```
$ gsi module/test
*** all tests passed out of a total of N tests
```

La commande affiche le résultat des tests unitaires contenus dans le sous-module test.

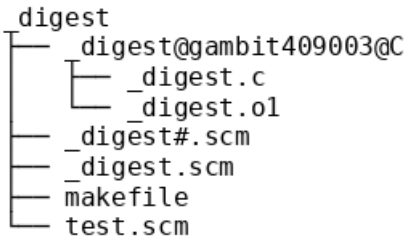
5.6. Compilation d'un module

La compilation d'un module est faite en passant le nom du module (*<module-ref>*) en paramètre. Le compilateur cherche le module dans les répertoires du `##module-search-order`. Le module est installé au besoin et ensuite compilé dans un sous répertoire. Ce dossier associe la compilation de ce module à la version de Gambit et à la cible (C, JavaScript, ...). La compilation d'un module se fait par la commande suivante:

```
$ gsc <module-ref>
```

L'arborescence du répertoire du module après la compilation du module `_digest` pour le *backend* C est:

Il est possible de forcer la compilation d'un module par la présence d'un fichier *module-name.*`_must-build_`. Cette fonctionnalité permet le chargement d'un module qui doit être compilé pour fonctionner correctement.



5.7. Comparaison avec d'autres systèmes

L'organisation des modules sur le système de fichier dans Gambit diffère de celui de OCaml, Python et NodeJS et des autres implémentations de Scheme. Akku [**34**], un gestionnaire de module pour Scheme compatible avec plusieurs implémentations de Scheme comme Chez Scheme, Chibi Scheme et GNU Guile. Il gère les modules pour les projets. Une seule version de chaque module peut être installé dans un projet.

Ceux-ci ne permettent pas l'installation de plus d'une version d'un module directement. Le système de module qui est utilisé dans le langage de programmation Go est le plus similaire à celui dans Gambit.

Notre système offre l'identification unique des modules, ce qui permet d'installer plusieurs versions d'un module. Un système de module permet la coexistence de plusieurs versions du même module sur le système de fichier s'ils sont considérés comme des modules différents. L'installation d'une version différente d'un module ne remplace pas la version déjà installée. L'organisation des bibliothèques est importante pour permettre cette coexistence.

La caractéristique que le système de bibliothèque doit avoir pour permettre plusieurs versions d'une bibliothèque est une organisation qui permet de distinguer les différentes versions de la bibliothèque par un chemin unique. La plupart des systèmes de module ne distinguent pas les versions d'un même module et ne permettent l'installation que d'une seule version. Les systèmes de module permettent la gestion de différents préfixes dans lesquels les modules sont installés. Chaque préfixe peut contenir une version différente d'un même module. Pour avoir une nouvelle version d'un module, il faut créer un nouveau préfixe.

La gestion de module de notre système ressemble à Nix [**10**], un gestionnaire de *package* utilisé par le système d'exploitation NixOS. Les modules dans Nix sont traités de façon fonctionnelle. Chaque *package* possède ces dépendances. Plusieurs versions d'un *package* peuvent être installées simultanément pour permettre le fonctionnement de l'ensemble des *packages*. Une mise à jour du système dans Nix ne brise pas les dépendances d'un *package*.

	Multiple versions
OCaml	✗
Python	✗
NodeJS	✗
Java	✗
Go	✓
Akku	✗
Racket	✗
Chez Scheme	✗
Gambit	✓
Nix	✓

Fig. 5.5. Une table qui compare différents systèmes de module sur la capacité d'installer plusieurs versions d'un module. Le système Go permet plusieurs versions d'un module pour des versions incompatibles selon le sémantique de version. La version `1.0.0` coexiste avec la version `2.0.0`. La version récente `1.2.0` remplace la vieille version `1.0.1`.

5.7.1. Organisation de OCaml

Le système de gestion de bibliothèques d'OCaml se nomme OPAM. Ce système permet d'avoir plusieurs environnements distincts contenant chacun un ensemble de versions des bibliothèques. Chaque environnement permet l'installation d'une version spécifique de chaque bibliothèque et est étiqueté avec un nom choisi par l'utilisateur. Un changement d'environnement est effectué par une requête de l'utilisateur `opam switch <envname>`. Il utilise le projet *mancoosi* pour gérer les contraintes de version, les dépendances optionnelles et la gestion des conflits. L'environnement par défaut est lié aux dépôts standard d'OCaml.

5.7.2. Organisation de Python

L'organisation des bibliothèques Python ne permet de stocker qu'un (typiquement la dernière) version d'une bibliothèque. Les emplacements des bibliothèques sont modifiés par la variable d'environnement `PYTHONPATH` qui correspond dans Python à la variable `path` de la bibliothèque interne `sys`. Le système de bibliothèque de Python ne permet pas la coexistence

de plusieurs versions de la même bibliothèque. Le *package manager* principal de Python est *pip*. L'installation d'une autre version d'une bibliothèque désinstalle ou masque la version déjà installée. Le système de module ne permet pas de référer à deux versions de la même bibliothèque.

```
>>> import sys
>>> print('\n'.join(sys.path))
/usr/lib/python37.zip
/usr/lib/python3.7
/usr/lib/python3.7/lib-dynload
/home/username/.local/lib/python3.7/site-packages
/usr/lib/python3.7/site-packages
```

Fig. 5.6. L'ensemble des répertoires qui est utilisé par Python version 3.7 pour organiser les bibliothèques sur un système de type Linux.

Python a le concept équivalent à OCaml de `virtualenv` qui permet d'avoir plusieurs versions installées sur la même machine. Cela permet d'installer des bibliothèques dans un environnement isolé des autres. L'avantage est qu'il est possible d'avoir une compatibilité avec des logiciels qui utilisent des versions de bibliothèques antérieures. Un inconvénient est qu'il n'y a pas un partage des versions de bibliothèques communes entre les différents environnements, cela a comme effet d'avoir plus d'un exemplaire d'une version de la bibliothèque installée sur le système de fichier. Chaque `virtualenv` ne permet qu'une seule version de chaque bibliothèque d'être installé.

5.7.3. Organisation de NodeJS

NodeJS est un interprète JavaScript qui a été conçu pour être exécuté du côté serveur dans un modèle client-serveur. Les bibliothèques sont installées au niveau du projet. Cela implique que plusieurs projets qui utilisent la même version de la bibliothèque vont référer au même exemplaire de la bibliothèque.

La structure d'une bibliothèque dans NodeJS est décrite par un fichier `package.json` qui contient plusieurs méta données comme le nom, la version, le nom des dépendances, la version des dépendances, la licence sous laquelle la bibliothèque est publiée et plusieurs autres méta données liées à la bibliothèque. Sous NodeJS, les bibliothèques sont gérées par

projet plutôt que globalement cela a comme avantage que chaque projet fonctionne avec ses versions des bibliothèques.

5.7.4. Organisation de Java

Les modules en Java sont nommés *package*. Le nom des modules utilise généralement l'inverse d'une URL comme espace de nom. Par exemple, les noms des modules liés aux services Google vont débuter par `com.google`. Cette convention a pour but d'unifier les noms de module. La version des modules n'est pas liée au nom du module dans le cas de Java. Il n'est pas possible de charger deux modules qui utilisent le même espace de nom, comme deux versions d'un même module.

Les projets Java utilise générallement Gradle ou Maven des système de *build* automatique. Ces système gère l'installation et la compilation des dépendances.

5.7.5. Organisation de Go

L'organisation des bibliothèques dans Go [**4**] est effectuée dans environnement dont la racine est spécifiée par la variable d'environnement `GOPATH` avec un répertoire pour les exécutables compilés (`bin`), un répertoire contenant le code source des différents projets (`src`) et un répertoire pour les objets des modules installés (`pkg`). Chaque paire de systèmes d'exploitation et d'architecture a son propre répertoire dans `pkg`.

Le système Go permet l'installation de plusieurs versions d'un même module dans le même environnement en utilisant le service *gopkg.in*. Il y a deux syntaxes utilisées pour l'URL des modules Go avec *gopkg.in*. Il est possible de spécifier une version spécifique du module lors de l'importation.

```
gopkg.in/pkg.v3      -> github.com/go-pkg/pkg (branch/tag v3, v3.N, or v3.N.M)
gopkg.in/user/pkg.v3 -> github.com/user/pkg   (branch/tag v3, v3.N, or v3.N.M)
```

Fig. 5.7. Exemple d'importation de la version v3 du module pkg en utilisant le service *gopkg.in*.

```go
package main

import (
  hellov1 "gopkg.in/FredericHamel/go-hello.v1"
  hellov2 "gopkg.in/FredericHamel/go-hello.v2"
)

func main() {
  // use hello version 1
  hellov1.Hello("Bob")

  // use hello version 2
  hellov2.Salut("Alice")

  // hellov1.Salut("Eve")
}
```

Fig. 5.8. Un exemple qui montre l'importation de deux versions d'un même module en Go. Le module **go-hello** version 2 exporte la fonction `Salut` qui n'existe pas dans la version `1`.

```
$GOPATH/
  - bin
    - ... binaries
  - src
    - github.com
      - UserName1
        - project1
        - project2
      - UserName2
        - projectA
        - projectB
  - pkg
    - linux_amd64
      - pkglist
        - objets
```

Fig. 5.9. Arborescence des fichiers dans le gopath qui contient l'URL complet vers le dépôt de chaque module.

5.8. Conclusion

Ce chapitre a traité de la gestion des modules. Les opérations du système de modules présentés dans ce chapitre sont l'installation, la désinstallation et la mise à jour. Notre approche de gestion des modules offre correctement la possibilité d'avoir plus d'une version d'un module. Cela empêche les bris de dépendances lors de l'installation de nouveaux modules. L'installation des modules peut-être déclenchés procéduralement ce qui est nécessaire pour diffuser les modules. Notre approche de gestion, qui utilise le nom du module et sa version, permet d'identifier de façon unique les modules et de les conserver dans le système de fichier. L'installation automatique des modules avec le chargement dynamique permet l'évolution d'une application sans qu'il soit redémarré. C'est une partie important de la diffusion des modules.

Chapitre 6

Migration de code

Ce chapitre traite du mécanisme de diffusion de modules qui est utilisé dans la migration de code. Notre approche exploite la sérialisation des objets Scheme, le langage Termite et l'installation automatique des modules durant l'exécution.

La migration de code entre les nœuds d'un système distribué n'est possible que si chaque nœud utilise la même version de Gambit configuré avec les mêmes paramètres. Des versions différentes de Gambit vont compiler le code différemment, ce qui peut rendre la sérialisation/désérialisation possiblement incompatible. La compilation des modules doit utiliser les mêmes déclarations et optimisations Scheme. Le compilateur C, le système d'exploitation, la taille des mots mémoires n'ont pas d'importance.

6.1. Systèmes distribués

Les systèmes distribués sont constitués d'un ensemble de nœuds interconnectés de calculs. Les nœuds interagissent par l'envoi et la réception de message au sein d'un réseau de communication. Chaque nœud a un but spécifique. Le *Web* est un exemple notable représentatif. Il est composé de clients et de serveurs qui exécutent des applications clients et serveurs différentes.

L'implémentation d'un système distribué inclut le développement des applications installées sur les nœuds et la logique d'interaction entre les nœuds. Il est possible de voir l'ensemble des programmes sur les nœuds comme un programme global. Les problèmes discutés dans ce chapitre sont les suivants:

- **RPC:** Comment l'appel distant à une procédure (RPC) implémentée quand l'envoyeur et le receveur ne sont pas conçus ensemble?

- **Mise à jour de code:** Comment la mise à jour du programme d'un nœud est effectuée lors d'un *bugfix* ou lorsqu'une nouvelle version est disponible?
- **Migration de tâche:** Comment déplacer un service sur un nouveau nœud quand le système sous-jacent est sur un système d'exploitation différent, à une architecture différente, etc.?
- **Opération continue:** Comment éviter les interruptions dans les situations précédentes?

Le langage Termite Scheme [**16**] a été conçu pour simplifier l'implémentation de systèmes distribués et fournit certaines solutions aux problèmes de ces systèmes. Le langage Termite Scheme est fortement inspiré des concepts du langage de programmation d'Erlang avec la syntaxe et la sémantique de Scheme. Une fonctionnalité intéressante qui est absente en Erlang est la capacité d'envoyer une continuation en message. Termite Scheme est implémenté sur le système Gambit Scheme qui offre une façon de sérialiser la plupart des objets Scheme incluant les procédures et les continuations.

La sérialisation de procédures est un outil utile pour implémenter un protocole RPC. En plus des procédures, il est possible de sérialiser des continuations. Une continuation est une structure de donnée qui capture l'état d'un processus. Donc il est possible de transmettre l'état d'un processus sur un autre nœud.

L'implémentation originale de Termite avait certaines limitations lors de la sérialisation des procédures et des continuations. Dans le cas interprété, les procédures et les continuations sont transmises sans problème entre les nœuds. Dans le cas compilé, il faut que chaque nœud possède le code compilé des procédures qui sont transmises.

6.2. Communication dans Termite

L'ensemble des données transmises sont sérialisées par la procédure `object->u8vector` qui prend en paramètre l'objet à sérialiser et optionnellement une procédure *transform* qui est appellé sur tous les sous-objets dans l'objet pour personnaliser le processus de sérialisation. Le résultat est un `u8vector` (vector d'octets).

La sérialisation de la plupart des objets utilise les bits de poids fort du premier octet pour indiquer le type de l'objet. Les autres des bits du premier octet sont utilisés pour encoder

des propriétés de base comme la longueur de l'objet (si la valeur peut être encodé avec ces bits).

La sérialization du vecteur `#(1 2 3)` est représenté par les 4 octets `#x23` `#x51` `#x52` `#x53`. Le premier octet indique que c'est un type vector de longueur 3, les autres octets représentent les nombre 1, 2 et 3. La sérialisation d'une procédure contient le nom de celle-ci et un entier qui l'identifie. La figure 6.1 montre un exemple de la sérialisation d'une procédure compilée. Cette représentation est conçue pour être compacte et éviter la redondance dans les données. En plus, elle est indépendante de l'architecture de la machine.

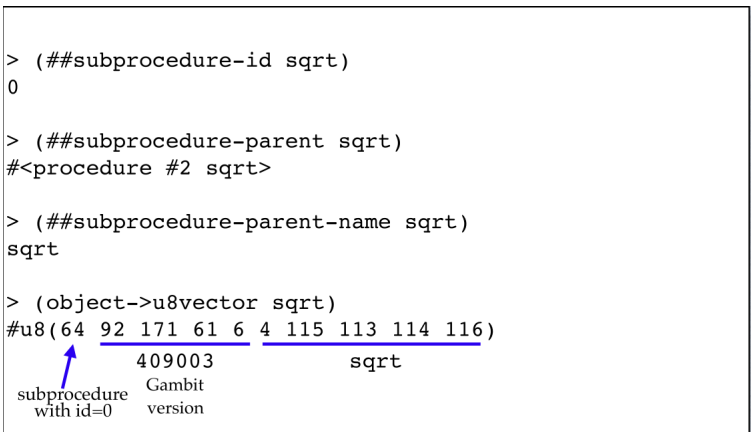


Fig. 6.1. La représentation du résultat de la sérialisation en `u8vector` de la procédure `sqrt`.

6.3. *Hook* des procédures inconnues

Le système Termite permet la migration de code compilé avec les procédures `object->u8vector` et `u8vector->object`. Ces procédures effectuent respectivement la sérialisation et la désérialisation. La sérialisation d'une procédure est encodée par le nom de la procédure et un entier qui l'identifie (`id`).

Le nom qualifié de la procédure (i.e. qui contient un `#`) est composé de l'espace de nom et du nom court. La procédure `_hamt#make-hamt` est dans l'espace de nom `_hamt` et à un nom court `make-hamt`. L'espace de nom dans un module qui est hébergé correspond au `module-ref` qui est dans le cas des modules hébergés l'URL du dépôt contenant le code du module.

Dans le processus de désérialisation, le nom de la procédure et le `id` sont utilisés pour retrouver la procédure. Un *hook* est invoqué si la procédure n'existe pas dans le processus courant. Le but de ce *hook* est de dynamiquement installer et charger le module qui implémente la procédure inconnue.

6.4. Exemple de migration de code

Pour montrer les capacités du système nous avons conçu un scénario de mise à jour de code entre deux nœuds. Un nœud qui exécute l'application compilée et un nœud qui exécute le code qui effectue la mise à jour de l'application sans interrompre le service.

Les deux nœuds ont la même version de Gambit, car la mise à jour de code compilé requiert une version uniforme de Gambit. La représentation des *subprocedure* peut être inconsistante entre les versions de Gambit.

L'application est une horloge programmable écrite en Termite Scheme qui affiche l'heure dans un certain fuseau horaire configurable. Cette application a un bogue d'affichage volontaire qui se produit lorsque l'horloge passe de `12h59` à `1h00`. L'affichage montre `1h009` au lieu de `1h00`.

Cette horloge offre un API simple:

- La modification du fuseau horaire est faite en envoyant un message (nombre entier).
- Le message `timezone-get` permet de récupérer le fuseau horaire courant.
- Le message `update-code` permet la mise à jour dynamique du code du serveur.
- Tous les autres messages sont rejetés par le serveur.

L'horloge programmable est démarrée sur le nœud, par exemple un raspberry pi 4. L'heure affichée n'est pas dans le bon fuseau horaire, donc nous utilisons le programme `timezone-set.scm` de la figure 6.4 pour configurer le bon fuseau horaire. Par exemple, pour assigner le fuseau 4 nous invoquons le programme comme suit: `gsi timezone-set.scm 4`.

L'horloge est maintenant dans le fuseau horaire 4. Une nouvelle version de l'horloge programmable est disponible. Le service ne doit pas être interrompu. Par commodité l'horloge supporte un message qui permet la mise à jour du code qui est (`update-code`).

Le programme `updater.scm` de la figure 6.4 permet d'envoyer la nouvelle version de l'horloge programmable. La mise à jour est faite en envoyant une procédure ou une continuation

qui contient la logique (comme dans la figure 2.8). Dans ce cas, c'est une procédure qui est envoyée pour mettre à jour le horloge programmable.

La procédure envoyée provient du module `github.com/frederichamel/termite-clock@v2` qui est la nouvelle version de l'horloge. L'espace de nom de ce module est `github.com/frederichamel/termite-clock@v2#`. Le nom de la procédure est `clock-thread-loop` dans l'espace de nom de ce module.

La sérialisation de la procédure contient le nom de la procédure incluant l'espace de nom. Le nœud Termite qui roule l'application de l'horloge programmable a configuré l'installation automatique des modules. Même si la procédure n'existe pas sur le nœud Termite de l'horloge programmable le système va l'installer, le compiler et le charger dynamiquement.

```
;; clock-app.scm
(import (termite))
(import (github.com/frederichamel/termite-clock @v1))

;; Should be integrated in the system.
(##unknown-procedure-handler-set!
  (lambda (name id)
    (let* ((name-str (##symbol->string name))
           (proc/ns (##reverse-string-split-at name-str #\#)))
      (and (##pair? (##cdr proc/ns))
           (let ((mod-name (##last proc/ns)))
             (##load-module (##string->symbol mod-name))
             (let ((proc (##get-subprocedure-from-name-and-id name id)))
               proc))))))

(define node (make-node "b.local" 3000))

(define (start)
  (node-init node)
  (display "\033[H\033[J") ;; clear screen
  (clock-start 'clock-app)
  (wait-for (resolve-service 'clock-app)))

(start)
```

Fig. 6.2. Le code du serveur qui configure le *hook* qui résout les références à des procédures inconnues. C'est effectué avec la procédure `##unknown-procedure-handler-set!`.

```
(define (clock-thread-loop timezone)
  (let tick ()
    (let* ((now (time->seconds (current-time)))
           (next (* 0.5 (floor (+ 1 (* 2 now))))) ;; next 1/2 second
           (timeout (seconds->time next)))
      (let wait ()
        (recv
          ((from tag timezone) (where (integer? timezone))
           (! from (list tag 'ok)) ;; send confirmation
           (clock-thread-loop timezone))

          ((from tag ('update-code k))
           (! from (list tag 'ok)) ;; send confirmation
           (k timezone))

          ((from tag 'timezone-get)
           (! from (list tag timezone))
           (wait))

          (after timeout
           (clock-update next timezone)
           (tick)))))))
```

Fig. 6.3. Extrait du code de l'horloge programmable.

6.5. Conclusion

Ce chapitre présente le mécanisme de diffusion des modules utilisés dans la migration de code. L'exemple de l'horloge montre une application de mise à jour dynamique du code qui est possible grâce au mécanisme de diffusion du code. Notre implémentation de ce système permet l'installation de la nouvelle version d'un module qui répare un bug dans l'application `termite-clock` dans arrêter l'application.

Ce mécanisme est utile pour mettre à jour un serveur évolutif sans l'interrompre. Notre approche permet la diffusion de modules entre les nœuds d'un système distribué. Nous évaluons les performances de notre système dans le prochain chapitre.

```
;; config.scm
(define local    (make-node "a.local" 3000)) ;; x86 arch
(define remote   (make-node "b.local" 3000)) ;; ARM arch
```

```
;; timezone-set.scm
(import (termite))

;; Configure the current node.
(include "config.scm")

;;; Update timezone
(node-init local)
(let* ((args (command-line))
       (rest (cdr args)))
  (if (or (null? rest)
          (pair? (cdr rest)))
      (!? (remote-service 'clock-app remote) 0) ;; => 'ok
      (let ((timezone (string->number (car rest))))
        (!? (remote-service 'clock-app remote) (or timezone 0))))) ;; => 'ok
```

```
;; updater.scm
(import (termite))
(import (github.com/frederichamel/termite-clock @v2))

;; Configure the current node.
(include "config.scm")

(node-init local)
(!? (remote-service 'clock-app remote)
    (list 'update-code clock-thread-loop)) ;; => 'ok
```

Fig. 6.4. Les applications clients qui interragissent avec l'horloge programmable. Il y a le programme qui modifie le timezone de l'horloge programme dans `timezone-set.scm`. Il y a le programme de mise à jour de l'horloge programmable dans `updater.scm`.

Chapitre 7

Évaluation

L'implémentation originale de Termite était capable d'envoyer des messages qui contient du code, qui est absent sur le nœud destination, à condition que le code soit interprété. Notre système permet cette transmission dans le contexte compilé. Ce chapitre évalue les performances du système de modules. Notre approche d'évaluation utilise trois programmes de test (benchmarks). Nous avons fait des expérimentations pour comparer les performances de la diffusion de module, que nous avons conçu, dans un contexte compilé avec la diffusion de code interprété déjà présent dans Termite Scheme.

Le but est de prouver que l'utilisation de modules compilés, rendu possible par notre approche, est plus avantageuse au niveau de la performance par rapport à la diffusion de code interprété. Nous avons aussi testé la diffusion des modules entre des nœuds de nature différente (ARM, x86).

7.1. Description des expériences

Les expériences exploitent l'installation automatique des modules pour la diffusion. Les performances du système de module sont mesurées en utilisant le temps de téléchargement, de compilation et de chargement des modules. Les modules pour tester les capacités de diffusion de code sont basés sur certains *benchmarks* standard de Scheme présent dans Gambit:

- Puzzle (4K)
- Scheme (40K)
- Compiler (400K)

Nous référons à ces *benchmarks* par leur taille du code source en octets (4K, 40K et 400K).

Les expériences sont effectuées dans trois scénarios distincts:

- INTERPRETED: Les deux nœuds sont interprétés.
- STEADY-STATE: Les modules sont déjà installés et compilés sur le nœud destination, mais non chargé.
- FIRST-INSTALL: Les modules ne sont pas installés sur le nœud destination.

Les tests de performance sont effectués sur deux nœuds. Le test de RPC consiste à diffuser une tâche sur le nœud générique qui ne connait aucun code qui est diffusé dessus.

La diffusion de code est effectuée 100 fois pour chaque benchmark. Le temps d'exécution du benchmark est paramétré de telle sorte que le temps de la version interprétée du benchmark soit proportionnel à sa taille. Le temps respectif de la version interprétée des benchmarks: Puzzle, Scheme et Compiler sont respectivement dans les ordres de 0.1 seconde, 1 seconde et 10 secondes. La première ligne des figures 7.1 montre cette tendance dans les temps d'exécutions. Notre but est de montrer un lien entre la taille du programme et le temps d'exécution.

7.2. Spécification des machines

Les machines sur lesquelles les tests ont été effectués sont:

- $M_{x86/Linux}$ est une machine `x86_64` roulant Linux;
- $M_{x86/macOS}$ est une machine `x86_64` roulant macOS;
- $M_{ARM/Linux}$ est un Raspberry Pi `ARM` roulant Linux.

Les spécifications détaillées de ces machines sont présentes à l'annexe A. La diversité des machines utilisées montre que la diffusion de code compilé est indépendante de la plateforme et de l'architecture de la machine.

La machine $M_{x86/Linux}$ est la machine la plus stable numériquement de toutes celles utilisées, car elle a un système de refroidissement au liquide. Les résultats pris sur $M_{x86/Linux}$ ont une variance plus petite et sont reproductibles.

La machine $M_{ARM/Linux}$ est la machine la moins performante. Elle est d'une architecture ARM, ce qui permet de tester la transmission de tâche entre des architectures différentes.

7.3. Résultats

Les temps dans les tests sont mesurés en millisecondes (ms). Le programme est exécuté 20 fois, les valeurs extrêmes (la plus grande est la plus petite) sont éliminées pour tenir compte des variations de la latence du réseau. Les tables des résultats contiennent la moyenne et l'écart-type de chaque mesure.

La première observation est que pour tous les scénarios le temps du nœud de départ n'a pas d'impact sur le temps RPC total. Cela est attendu puisque la majorité du travail est effectué sur le nœud destination. Les temps les plus grands sont généralement pour $M_{ARM/Linux}$. Cela peut être expliqué par le temps de transfert qui est légèrement plus grand.

Le temps de transfert (la latence réseau et de sérialisation/désérialisation) varie proportionnellement avec la taille des messages. Dans le scénario STEADY-STATE, les messages sont les paramètres de la procédure et le résultat. Dans ce scénario, le temps associé aux transmissions est de 13-15ms pour $M_{x86/macOS}$ et de 25-30ms pour $M_{ARM/Linux}$ (un processeur et une interface réseau moins performants). Les temps mesurés sur le nœud destination sont essentiellement les mêmes.

Dans le scénario INTERPRETED le message contient une représentation du code à exécuter, ce qui est gros pour le 400K. Dans ce scénario, le temps de transmission est de 15-83ms pour $M_{x86/macOS}$ et de 47-411ms pour $M_{ARM/Linux}$ (encore une fois le temps est affecté par la performance du processeur et l'interface réseau). Le temps de transmission est négligeable par rapport au temps total du RPC. Pour le scénario STEADY-STATE, le temps de transmission est relativement plus important pour le 4K qui est le programme le plus petit. La transmission demande plus de travail par rapport au calcul sur le nœud destination.

7.3.1. Comparaison entre les scénarios INTERPRETED et STEADY-STATE

Pour évaluer les performances de notre système, nous pouvons comparer le temps des scénarios INTERPRETED et STEADY-STATE. Le temps RPC total du scénario STEADY-STATE est 22x plus plus petit pour $M_{x86/macOS}$ et $M_{ARM/Linux}$ que dans le scénario INTERPRETED pour le 400K. Pour le 4K, les temps d'exécution du RPC total sont de 6x-9x

$M_{x86/macOS}$

Temps (ms)	4K	40K	400K
Total pour RPC	146.4 ± 0.6	966.9 ± 6.1	10463.8 ± 3.3
Sur la destination	131.6 ± 0.2	948.7 ± 5.9	10381.0 ± 2.7

$M_{ARM/Linux}$

Temps (ms)	4K	40K	400K
Total pour RPC	179.2 ± 1.6	1002.7 ± 8.1	10801.1 ± 11.0
Sur la destination	132.6 ± 0.7	954.6 ± 0.0	10390.5 ± 2.6

Fig. 7.1. Les temps dans le scénario INTERPRETED avec $M_{x86/macOS}$ ou $M_{ARM/Linux}$ comme le nœud de départ

$M_{x86/macOS}$

Temps (ms)	4K	40K	400K
Total pour RPC	15.5 ± 0.7	48.5 ± 0.6	478.7 ± 0.6
Sur la destination	2.2 ± 0.0	35.2 ± 0.1	463.3 ± 0.3

$M_{ARM/Linux}$

Temps (ms)	4K	40K	400K
Total pour RPC	26.9 ± 1.2	60.4 ± 0.6	492.5 ± 0.7
Sur la destination	2.2 ± 0.0	35.2 ± 0.0	462.8 ± 0.3

Fig. 7.2. Les temps dans le scénario STEADY-STATE avec $M_{x86/macOS}$ ou $M_{ARM/Linux}$ comme le nœud de départ

plus petits que dans le cas INTERPRETED. Dans le cas moyen 40K, l'accélération est de 16x-20x plus rapide que le scénario interprété.

7.3.2. Comparaison entre les scénarios INTERPRETED et FIRST-INSTALL

Le scénario FIRST-INSTALL inclut le temps d'installation et de compilation des modules. En comparaison avec le scénario INTERPRETED, une première observation indique que le temps total RPC pour $M_{x86/macOS}$ et $M_{ARM/Linux}$ est 6x-8x fois plus grand pour le

$M_{x86/macOS}$

Temps (ms)	4K	40K	400K
Total pour RPC	1208.6	2460.3	148536.2
Sur la destination	2.2	36.1	464.7

$M_{ARM/Linux}$

Temps (ms)	4K	40K	400K
Total pour RPC	1159.8	2502.0	153272.6
Sur la destination	2.2	37.1	464.1

Fig. 7.3. Les temps dans les scénario FIRST-INSTALL avec $M_{x86/macOS}$ ou $M_{ARM/Linux}$ comme le nœud de départ

bench 4K (le module le plus petit). Cela s'explique par le temps de compilation et d'installation. Le temps d'exécution sur le nœud de destination est 60x plus rapide que dans le cas INTERPRETED, mais le temps de computation ajoute un coût substanciel qui fait que le RPC est plus lent.

7.3.3. Comparaison entre les scénario FIRST-INSTALL et STEADY-STATE

Le temps d'exécution total des programmes de tests exploitant la diffusion de code compilé, nécessitant l'installation et la compilation du module, est plus court que la version interprétée. Le facteur qui prend le plus de temps dans le contexte compilé est l'installation et la compilation des modules diffusés. Suite à une première exécution qui a installé les modules compilés, l'exécution est plus rapide que dans le cas interprété.

Dans le cas où les modules ont déjà été installés et compilés, l'exécution complète des programmes de tests est plus rapide dans le cas compilé, jusqu'à 22x. Dans ce cas, les modules sont chargés dynamiquement dans le programme de test lors de la diffusion de code. Les figures 7.2 montrent les temps dans le cas où les modules ont été compilés et les figures 7.1 montrent les temps des modules interprétés. Cela s'explique par le fait qu'il y a moins de données transmises et que l'exécution de code compilé est plus rapide que l'interprétation.

Le temps de transmission des données dans les programmes de test compilés était plus court que dans le cas interprété pour certaines des transmissions, malgré le fait qu'il y a

moins de données à transmettre. Cela affecte environ 10% des transmissions et est lié à des facteurs aléatoires du réseau.

7.4. Conclusion

Le temps de diffusion des modules est proportionnel à leurs tailles. Plus un module est gros plus son temps de diffusion est grand. La diffusion de modules compilés est avantagée lorsque le temps d'exécution de l'appel RPC interprété est plus grand que les temps de compilation et d'installation. Le temps d'exécution compilé lorsque les modules sont déjà installés, mais non chargés est plus rapide que la transmission des modules interprétés. Notre approche de diffusion offre une exécution qui est amortie après la première diffusion de module, c'est-à-dire après que les modules aient été installés et compilés.

Chapitre 8

Conclusion

Ce mémoire a présenté un système de module spécialisé pour les systèmes distribués. Il permet la conception d'applications qui exploitent la diffusion transparente de modules entre les nœuds. Les appels RPC et la migration de code mobile entre des nœuds de nature différente sont possibles. Chaque module diffusé a un nom unique basé sur l'URL du dépôt de code qui permet de le récupérer. La diffusion se fait au moyen du nom des identifiants qui contient l'URL du module. La façon de nommer les modules dans ce système est similaire à celui du langage de programmation Go.

Nous avons commencé par exposer les limitations du système Termite Scheme existant par des expérimentations dans plusieurs situations. Dans le contexte purement interprété, la migration fonctionnait correctement même dans le cas ou le nœud de destination ne connait pas le code de l'agent mobile. C'est dans le contexte où les applications de chaque nœud sont compilées que la migration de tâche est un défi, car elle requiert la présence du code compilé de l'agent mobile sur l'ensemble des nœuds qui peuvent avoir des architectures et un système d'exploitation différents.

Nous avons exploré plusieurs méthodes de chargement automatique des modules. Le langage Scheme (que ce soit R5RS, R6RS ou R7RS) permet le chargement de module à la demande, mais pas simultanément de multiples versions d'un module ce qui limite énormément la possibilité de mise à niveau d'un service sans interruption. Ceci nous a menés à un constat que le nom des identifiants de module transmis devait être unique au sein du système distribué.

Nous avons ajouté une forme spéciale à Gambit pour permettre la définition de modules. Ce projet a mené au système de modules spécialisés pour les systèmes distribués présenté

dans ce mémoire. Le résultat est prometteur, il permet le déploiement de serveurs sur plusieurs machines d'architecture et de systèmes d'exploitation différents, tel que démontré par nos expériences.

Le système Gambit a été amélioré par l'ajout des modules primitifs et aussi des modules compatible avec la syntaxe de R7RS. De nouveaux mécanismes de chargement de modules ont été ajoutés pour garantir l'ordre et le chargement unique de chaque module.

L'installation et la compilation des modules sont effectuées automatiquement à la demande. Les coûts du chargement de module sont amortis après la première installation et compilation d'un module. L'utilisation interprétée de Termite n'utilise pas la compilation et l'installation automatique. La vitesse d'exécution après la compilation surpasse la version interprétée. Le facteur d'accélération observé après la première exécution qui installe les modules compilés est approximativement de 75x pour le test de 4K, 459x pour le test de 40K et 33x pour le test de 400K, des programmes de taille grandissante. Le facteur d'accélération est principalement lié au fait que le code compilé est plus rapide que celui interprété. Puisqu'il y a moins de transmission dans le cas compilé, car les données sont plus compactes que lorsqu'interprété, le temps de transmission est généralement plus petit. Dans nos tests, la taille de la sérialisation d'une procédure compilée est environ 400 fois plus petite que sa version interprétée.

L'approche de diffusion des modules est applicable à d'autres langages. La coexistence de plusieurs versions d'un module est indépendante du chargement dynamique et de la sérialisation des objets. Le chargement dynamique est nécessaire pour développer des applications extensibles, car cela permet le chargement de bibliothèque durant l'exécution (sans interruption de service). Les procédures doivent être manipulables comme une donnée et sérialisable pour pouvoir être transmis à un autre nœud. Une sérialisation et déserialisation qui est indépendante de la machine est nécessaire pour la transmission des objets (nombre, booléen, liste, vecteur, procédures) entre les nœuds d'un système distribué hétérogène. La sérialisation doit être similaire à celle qui est utilisée dans Gambit, par son uniformité sur toutes les plateformes. L'installation automatique avec le chargement dynamique offre la possibilité de charger un module qui n'est pas présent sur le nœud courant. La sérialisation des procédures contient l'information qui est utilisée dans l'installation automatique du module qui contient cette procédure.

Notre approche de gestion des modules diffère des autres gestionnaires par le traitement des versions des modules. Plusieurs versions d'un module peuvent être installées. La gestion des modules dans Racket (*raco*) ne permet pas d'installer plusieurs versions des modules. Akku, un autre gestionnaire de modules compatible avec plusieurs implémentations de Scheme a des caractéristiques similaires à *raco* (le gestionnaire de *package* de Racket [**27**]).

L'implémentation actuelle des modules a quelques aspects mineurs à améliorer. Les macros définies dans le contexte d'un module ont des problèmes d'hygiènes lors de l'expansion dans un autre module. Le temps d'installation des modules peut être optimisé. L'opération de mise à jour ne permet pas de mettre à jour les branches qui sont installées. Les messages d'avertissement (*warnings*) lors de la compilation des modules sont manquants. Il n'y a pas de message indiquant l'utilisation d'une liaison non définie.

En résumé, notre approche de diffusion des modules compilés offre une plus grande flexibilité et performance à l'exécution. Cela permet la mise à jour de code d'un nœud distant sans interrompre son service et sans intervention manuelle de déploiement du nouveau code. Cette approche est applicable à d'autres langages. La transmission permet d'avoir une meilleure performance d'exécution.

Références bibliographiques

[1] *TCLTK'96: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, Berkeley, CA, USA, 1996. USENIX Association.

[2] David M. Beazley, Brian D. Ward et Ian R. Cooke : The inside story on shared libraries and dynamic loading. *Computing in Science and Engineering*, 3(5):90–97, 2001.

[3] T. Berners-Lee, MIT/LCS, R. Fielding, U.C. Irvine, L. Masinter et Xerox Corporation : Uniform resource identifiers (uri): Generic syntax. `https://tools.ietf.org/html/rfc2396`, 1998.

[4] Tyler Bui-Palsulich et Eno Compton : Go reference manual 1.13.5. `https://blog.golang.org/using-go-modules`, 2019.

[5] Capturing, et Stefan Funfrocken : Transparent migration of Java-based mobile agents. *In Mobile Agents*, pages 26–37. Springer-Verlag, 1998.

[6] Guillaume Cartier et Louis-Julien Guillemette : Jazzscheme: Evolution of a lisp-based development system. *In 2010 Workshop on Scheme and Functional Programming*, page 50. Citeseer, 2010.

[7] Scheme CHICKEN : A practical and portable scheme system. *URL http://www. cal l-cc. org*, 131:11–20.

[8] W. CLINGER, J. REES et *et al* : Revised^4 report on the Algorithmic Language Scheme. `http://people.csail.mit.edu/jaffer/r4rs.pdf`, novembre 1991.

[9] William D Clinger : Scheme@33. *In Celebrating the 50th Anniversary of Lisp*, LISP50, pages 7:1–7:5, New York, NY, USA, 2008. ACM.

[10] Eelco Dolstra, Merijn Jonge et Eelco Visser : Nix: A Safe and Policy-Free System for Software Deployment. pages 79–92, 01 2004.

[11] R Kent Dybvig *et al.* : Chez scheme, 2011.

[12] Marc Feeley : Feature-based conditional expansion construct. `https://srfi.schemers.org/srfi-0/srfi-0.html`, 1999.

[13] Marc Feeley et Philip W. Trinder, éditeurs. *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*. ACM, 2006.

[14] Adrian Francalanza et Tyron Zerafa : Code management automation for Erlang remote actors. *In* Jamali *et al.* [**19**], pages 13–18.

[15] Mark Galassi : Guile user manual. *Los Alamos National Laboratory and Cygnus Support*, 1996.

[16] Guillaume Germain : Concurrency oriented programming in termite scheme. *In* Feeley et Trinder [**13**], page 20.

[17] Robert S. Gray : Agent tcl: A flexible and secure mobile-agent system. *In Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 9–23, Berkeley, CA, USA, 1996. USENIX Association.

[18] W. Wilson Ho et Ronald A. Olsson : An approach to genuine dynamic linking. *Softw., Pract. Exper.*, 21(4):375–390, 1991.

[19] Nadeem Jamali, Alessandro Ricci, Gera Weiss et Akinori Yonezawa, éditeurs. *Proceedings of the 2013 Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013*. ACM, 2013.

[20] Guy L. Steele Jr. : Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: the ultimate GOTO. *In* Ketchel *et al.* [**23**], pages 153–162.

[21] Takashi Kato : Implementing r7rs on an r6rs Scheme system. *In Scheme and Functional Programming Workshop, SFPW 2014*, 2014.

[22] R. KELSEY, W. CLINGER, J. REES et *et al* : Revised^5 report on the Algorithmic Language Scheme. `https://schemers.org/Documents/Standards/R5RS/r5rs.pdf`, février 1998.

[23] James S. Ketchel, Harvey Z. Kriloff, H. Blair Burner, Patricia E. Crockett, Robert G. Herriot, George B. Houston et Cathy S. Kitto, éditeurs. *Proceedings of the 1977 annual conference, ACM '77, Seattle, Washington, USA, October 16-19, 1977*. ACM, 1977.

[24] A. Lukić, N. Luburić, M. Vidaković et M. Holbl : Development of multi-agent framework in JavaScript. *In ICIST 2017 Proceedings Vol.1*, pages 261–265, 2017.

[25] Stefan M, Raymond Bimazubute et Herbert Stoyan : Mobile intelligent Agents in Erlang. *In Fourth International ICSC Symposium on ENGINEERING OF INTELLIGENT SYSTEMS (EIS 2004)*, 2004.

[26] MichałPiotrowski et Wojciech Turek : Software agents mobility using process migration mechanism in distributed erlang. *In Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 43–50, New York, NY, USA, 2013. ACM.

[27] PLT Inc. : Racket scheme. `https://racket-lang.org/`, 2020. Accessed: 2020-02-24.

[28] Tom Preston-Werner : Semantic versioning 2.0.0. `https://semver.org/spec/v2.0.0.html`, 2013. Accessed: 2020-02-27.

[29] Manuel Serrano et Pierre Weis : Bigloo: a portable and optimizing compiler for strict functional languages. *In* Alan Mycroft, éditeur : *Static Analysis*, pages 366–381, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[30] A. SHIN, J. COWAN, ARTHUR A. GLECKLER et *et al* : Revised^7 report on the algorithmic language scheme. `https://small.r7rs.org/attachment/r7rs.pdf`, juillet 2013.

[31] MICHAEL SPERBER, R. KENT DYBVIG et *et al* : Revised^6 report on the Algorithmic Language Scheme. `http://www.r6rs.org/final/r6rs.pdf`, septembre 2007.

[32] Eijiro Sumii : An implementation of transparent migration on standard scheme. *In Department of Computer Science, Rice University*, 2000.

[33] Dimitris Vyzovitis : Gerbil scheme. `https://cons.io/`, 2020. Accessed: 2020-02-21.

[34] Göran Weinholt : Akku package management made easy. `https://akkuscm.org/`, 2019. Accessed: 2020-02-21.

Annexe A

Information des environnement de test

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
CPU(s):	4
Thread(s) per core:	1
Core(s) per socket:	4
Vendor ID:	GenuineIntel
Model name:	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
CPU MHz:	4200.000
CPU min MHz:	800.0000
Storage:	216GB (NVME)
RAM:	16GB
Swap:	16GB
C Compiler	gcc (Debian 6.3.0-18+deb9u1) 6.3.0 20170516
Ethernet Speed	1Gbps

Fig. A.1. La spécification de la machine arctic qui est utilisé comme nœud de destination dans l'ensemble des tests. Cette machine, nommé $M_{x86/Linux}$ est refroidit au liquide.

```
Architecture:        armv7l
Byte Order:          Little Endian
CPU(s):              4
Thread(s) per core:  1
Core(s) per socket:  4
Vendor ID:           ARM
Model name:          Cortex-A72
CPU max MHz:         1500.0000
CPU min MHz:         600.0000
OS:                  Linux tictoc 4.19.66-v7l+ #1253 SMP
Hostname:            tictoc.iro.umontreal.ca
Storage:             26GB (sdcard)
RAM:                 2GB
Swap:                2GB
C Compiler           gcc (Raspbian 8.3.0-6+rpi1) 8.3.0
Ethernet Speed       1Gbps
```

Fig. A.2. La spécification du CPU du Raspberry Pi $M_{ARM/Linux}$ utilisé dans les tests.

```
Architecture:        x86_64
CPU(s):              12
Thread(s) per core:  2
Core(s) per socket:  6
Model name:          Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz
OS:                  Darwin Kernel Version 19.2.0
Hostname:            gambit.iro.umontreal.ca
Storage:             1TB (SSD)
RAM:                 32GB
Network Speed:       1Gbps
```

Fig. A.3. La spécification de la machine $M_{x86/macOS}$ qui roule mac OS.