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 à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

décembre 2019

Sommaire

Le présent mémoire décrit et évalue un système de module qui améliore la migration de code dans le langage distribué Scheme. Ce système de module a la possibilité d'être utilisé dans les applications qu'elle soit distribués 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 Gambit, un compilateur et interprète de Scheme.

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.

Mots clés : Langage de programmation fonctionnel, Scheme, Erlang, Système de module, Système distribué, Agent mobile.

Summary

This thesis presents a module system for Gambit Scheme that supports distributed computing. This module system facilitates application modularity and ease code migration between the nodes of a distributed system. The Termite Scheme language is used to implement the distributed applications.

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

The thesis will describe the module structure and how it was implemented into Gambit. The features of this system are shown through application examples.

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

Table des matières

Liste des tableaux

Table des figures

Remerciements

remerciements

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 5 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 dynamique*. 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* qui s'occupe de lier les fonctionnalités des modules au programme principal. Le chapitre 3 explique plus en profondeur le fonctionnement de l'éditeur de liens dynamique. 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. Ce type de module, les *bibliothèques partagées*, est décrit dans le chapitre 3.

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.

Le concept de version de module n'est pas intégré à la sémantique de plusieurs langages de programmation. Ainsi en Python et Java les modules sont référés dans le code avec un nom qui n'inclut pas la version. Go est un rare 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.

Nous reviendrons sur les modules versionnés au chapitre 3, car ce concept est essentiel dans notre approche de modularisation.

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 de systèmes et langages de programmation. Le protocole `ssh` permet une forme d'appel RPC par invocation de processus sur un système 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 [**17**], Java [**4**], Scheme [**21**][**10**], JavaScript [**16**]. Ce mécanisme requiert typiquement que l'ensemble du code soit disponible sur le système distant.

Le système Java présenté dans l'article [**4**] est un cas de migration de tâche. Il capture l'état présent du programme qui consiste aux valeurs et aux 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 Erlang [**17**] qui implémente des agents mobiles requiert que le code soit disponible sur le nœud destination.

Le système de migration en JavaScript 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.

Un exemple concret de migration de code est présenté au chapitre 7.

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.

1.5. Survol du mémoire

Pour ces raisons, 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.

Le chapitre 2 explique les approches existantes de modularisation en Scheme. Le chapitre 3 explique les problématiques des dépendances entre des modules natifs au sein du système de module. Le chapitre 4 traite de l'implémentation du système de module et son intégration dans Gambit, un compilateur/interprète performant pour Scheme. Le chapitre 5 explique les méchanismes de chargement 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'expérience effectuée entre des systèmes d'exploitation (SE) 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 Termite Scheme, 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. Notre travail a donc pu se concentrer sur la conception d'un système de module spécialisé aux besoins du code mobile.

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.

2.1. Scheme et sa syntaxe

Le langage Scheme[**6**], 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).

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*. 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>*. Il est utilisé 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 associations (*<bindings>*) visibles seulement dans le contexte de l'expression *<body>* dont la valeur de la forme `let`. Les associations 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 *<e1>* est exécutée si la valeur de l'expression *<e2>* est vraie sinon l'expression *<e3>* 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.

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

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

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

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

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).

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 garanti. Il est assez simple d'ajouter une syntaxe pour les boucles à l'aide de la méta programmation.

La méta programmation en Scheme est basée sur la capacité d'un programme de manipuler d'autre programme 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 se référant à elle-même. 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 entrée des *s-expression*s et retourne une *s-expression*.

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

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

Figure 2.2. Implémentation de la macro `include` qui permet l'inclusion d'un fichier dans un autre fichier avec la forme `define-macro`.

un autre, il faut tout d'abord lire le contenu du fichier à inclure. Ensuite, il suffit de retourner le code lu. La figure 2.2 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 2.3. Les particularités de `define-syntax` ne sont expliquées puisqu'elles ne sont pas pertinentes à ce mémoire.

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

Figure 2.3. 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[**5**] et R5RS[**15**] les modules consistent en des fichiers Scheme qui contiennent des définitions de procédures et de macros. Ils sont chargés dans le module courant 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 2.3). 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. Le `include` d'un fichier 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 définit 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 2.4 montre un exemple de module simple n'utilisant que la procédure `load`.

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

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

Figure 2.4. 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`.

Le standard R6RS[**20**] renforce le concept de modules avec la syntaxe `library`. Un module R6RS est séparé en 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, il peut contenir une spécification de version. La version est spécifiée par une liste d'entiers positifs. Une liste vide `()` est l'équivalent de ne pas spécifier la version. Ensuite, il y a la liste des exportations spécifier par la sous-forme `export`. Chaque 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` empêche le chargement multiple d'un module. La syntaxe du `import` permet de manipuler les noms des symboles importés et exportés.

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

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

Figure 2.5. Structure globale d'un module R6RS

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 [**20**]. 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'autre module. 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[**19**] simplifie la syntaxe des modules R6RS[**20**]. 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>*)
```

Figure 2.6. 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 version non sensible à la casse des caractères.
— La déclaration `cond-expand` est une extension du `SRFI-0` [7] dans le contexte d'une bibliothèque permettent 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. 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

```
;; Bibliotheque R6RS
(library (math)
  (export fact)
  (import (rnrs base))
  (define (fact n)
    (if (< n 2)
      1
      (* n (fact (- n 1))))))
```

```
;; Bibliotheque R7RS
(define-library (math)
  (export fact)
  (import (scheme base))
  (begin
    (define (fact n)
      (if (< n 2)
        1
        (* n (fact (- n 1)))))))
```

Figure 2.7. Comparaison entre la syntaxe des modules R6RS et R7RS

`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.

Chapitre 3

Coexistence de bibliothèques dynamiques

Ce chapitre traite du fonctionnement de l'éditeur de liens du système d'exploitation (*dynamique linker*) et des bibliothèques dynamiques. Le système de modules, présenté dans ce mémoire, supporte le chargement de plusieurs versions d'un module. Cela a le potentiel de causer des problèmes d'exécution du programme. Puisque ce système de module utilise des bibliothèques dynamiques du système d'exploitation, il est donc nécessaire de déterminer la nature de ces situations problématiques. Nous avons procédé à des expériences pour déterminer sous quelles conditions des bibliothèques dynamiques peuvent coexister.

3.1. Format des modules

La modularisation est utile pour séparer un programme en plusieurs composantes relativement indépendantes. Ici nous traitons des modules au niveau du système d'exploitation ("system libraries"). La forme dans laquelle un module est stocké dépend du contexte. Dans les langages interprétés, les modules ont la forme de fichier textuel, contenant du code, qui est compréhensible par un humain. Cela facilite les modifications du code du module. Par contre, l'interprétation de module en format textuel est beaucoup plus lente. Les modules compilés sont l'alternative aux modules textuels.

Un module qui est compilé en format binaire est plus difficile à comprendre du point de vue d'un humain, mais est plus rapide à s'exécuter. Le compromis est la rapidité contre la lisibilité. Le format binaire peut varier d'un système à l'autre. Les systèmes Linux utilisent le format ELF (*Extensible Linking Format*), Microsoft Windows le format PE (*Portable Executable*) et macOS le format Mach-O (*Mach object*). Ces formats binaires sont utilisés pour les exécutables, les bibliothèques statiques et les bibliothèques dynamiques.

Les bibliothèques dynamiques ont des problèmes propres comparativement aux bibliothèques statiques.

3.2. Édition de liens dynamique

Une application qui utilise une bibliothèque partagée ne contient pas le code de la bibliothèque, mais plutôt le nom des bibliothèques utilisées. La routine qui récupère le code à partir du nom est la résolution qui est effectuée par le *dynamic loader*. Lorsqu'un programme lié dynamiquement à plusieurs bibliothèques partagées exécute invoque une procédure externe, la routine de résolution est démarrée pour résoudre le code de cette procédure.

Par exemple, sous Linux l'utilitaire «yes», qui est écrit en C, est lié aux bibliothèques système suivantes :

```
linux-vdso.so.1 (0x00007ffeef7f9000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007ff68161c000)
/lib64/ld-linux-x86-64.so.2 => ...
```

La bibliothèque *libc.so.6* contient un grand nombre des fonctions standards du système sous Linux. Le chargement des bibliothèques s'effectue au début de l'application, avant l'exécution de la fonction principale souvent nommée **main**. Plusieurs bibliothèques peuvent coexister simultanément au sein d'un même processus sans que l'exécution du programme en soit affectée.

La résolution des fonctionnalités de ces bibliothèques est effectuée par un programme appelé le *program interpreter* du système qui correspond à */lib64/ld-linux-x86-64.so.2*. Il est possible de forcer la résolution d'une fonctionnalité d'une bibliothèque de façon manuelle. Ce genre d'interaction est possible sur les trois principales plateformes utilisées sur le marché (Windows, macOS et Linux).

Sur Linux, l'API qui permet d'interagir avec les bibliothèques partagées provient de *libdl.so*. Elle contient les fonctions *dlopen*, *dlsym*, *dlerror* et *dlclose* pour gérer des bibliothèques de code supplémentaire chargé manuellement à l'exécution. Pour charger la fonction *foo*, qui ne prend pas d'argument et ne retourne rien de la bibliothèque *libFoo.so* en C, il faut exécuter les deux appels de la figure 3.1 :

```
...
void *handle = dlopen("./libFoo.so", RTLD_LAZY);
void (*foo)() = dlsym(handle, "foo");
...
```

Figure 3.1. Chargement dynamique de la bibliothèque *libFoo.so* et résolution de la fonction *foo* sans gestion d'erreur sous Linux

L'équivalent des bibliothèques partagées sous Windows est les DLLs, qui peuvent être chargées de façon similaire dans un programme en utilisant les fonctions *LoadLibrary*, *LoadLibraryEx* et *GetProcAddress*. Ils fonctionnent de la même façon que leur équivalent Linux. Pour macOS, il faut passer par les routines :

— *NSCreateObjectFileImageFromFile*
— *NSLinkModule*
— *NSLookupSymbolInModule*
— *NSAddressOfSymbol*

La majorité des langages interprétés permettent l'importation de bibliothèques de code natives, via une interface nommée *foreign function interface* (FFI). Cette interface offre une couche d'abstraction de ces procédures. Prenons comme exemple les langages Python, Ruby, Lua et Scheme. Python possède le module **ctypes** qui permet de charger des bibliothèques natives dynamiques, et Ruby possède le module **ffi**.

Certains langages ont un mécanisme pour charger des bibliothèques natives s'ils ont été conçus spécialement. Dans le langage de programmation Lua, il est possible de charger directement une bibliothèque dynamique si elle contient une fonction **luaopen_*libname*** où *libname* est le nom de la bibliothèque. Gambit Scheme utilise un mécanisme équivalent. Il permet le chargement de ces modules qui ont été compilés en bibliothèque partagée (DLL) avec la fonction `(load "libname")`.

La résolution des fonctionnalités effectuée par le *dynamic linker* utilise un ordre de recherche définit. Cet ordre de recherche inclut l'exécutable courant, les dépendances de l'exécutable, la bibliothèque passée à *dlsym*. Une résolution d'une fonctionnalité est soit directe

```
require 'ffi'
# Chargement d'une bibliotheque native.
module LibFoo
    extend FFI::Library
    ffi_lib './libFoo.so'
    attach_function :foo, [], :void
end
# Appel de la fonction foo.
LibFoo.foo
```

Figure 3.2. Code d'importation de la fonction **foo** de la bibliothèque *libFoo.so* en Ruby

ou indirecte. La résolution d'une fonctionnalité par *dlsym* qui n'engendre pas la résolution d'un autre fonctionnalité externe est directe. Une résolution directe n'utilise pas le programme principal dans l'ordre de recherche. Les résolutions de fonctionnalité provenant d'appels indirects au *dynamic linker* inclut le programme principal et ses dépendances avant la bibliothèque passée à `dlsym`.

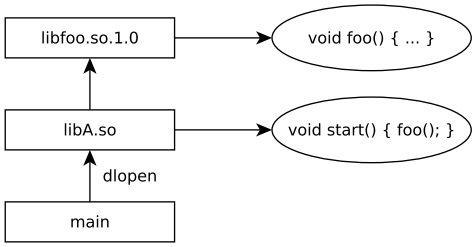

Figure 3.3. Un exemple de dépendance de bibliothèques au sein d'un application simple fictive. La bibliothèque *libA.so* est chargée dans l'application `main` via les appels aux procédures *dlopen* et *dlsym*. Les fonctionnalités utilisées dans l'exemple sont marquées dans des ellipses.

Dans la situation présentée dans la figure 3.3, qu'elles sont les étapes incluses dans l'exécution de ce programme qui invoque la fonctionnalité `start` de la bibliothèque *libA.so*. La fonctionnalité externe `start` est résolu de façon directe par un appel à `dlsym(libA, "start")`,

qui commence la recherche de la procédure `start` dans la bibliothèque spécifiée dans `dlsym`. Le programme l'invoque une fois la procédure trouvée. L'appel à une procédure non résolue (e.g. la procédure `foo` invoqué dans `start`) déclenche une procédure automatique de résolution des fonctionnalités. Cette procédure de résolution commence sa recherche à partir de l'exécutable, puis itère la liste des dépendances directe. Si la fonctionnalité n'est pas encore trouvée, la recherche continuera à partir de la bibliothèque passée à `dlsym`.

Il est facile d'exploiter l'ordre de recherche du *dynamic linker* pour causer un masquage de fonctionnalité dans une application. Ce masquage est utilisé dans certains contextes pour déboguer un programme, mais il peut aussi nuire à l'exécution du programme. Il y a au moins deux structures de programme qui causent un masquage. L'application principale est construite comme une bibliothèque dynamique exécutable et fournit une fonctionnalité qui porte le même nom qu'une fonctionnalité exportée par une bibliothèque externe chargée manuellement. L'application principale est liée avec une bibliothèque qui exporte une fonctionnalité qui porte le même nom que celle utilisée dans la bibliothèque externe.

La figure 3.4 est un exemple de masquage qui utilise la seconde méthode. La première méthode requiert des bibliothèques exécutables qui ne sont pas disponibles sur toutes les plateformes. Sous Linux, il est possible de créer une bibliothèque exécutable en passant le paramètre `-rdynamic` à *gcc* lors de la construction de la bibliothèque.

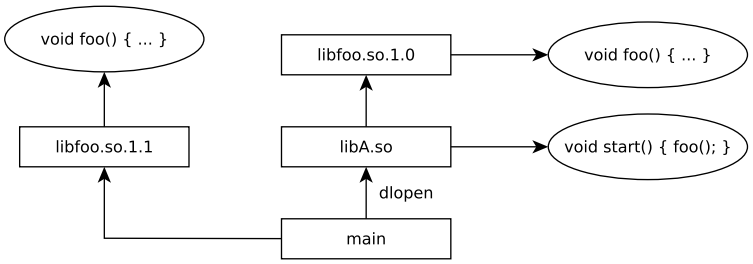

Figure 3.4. Exemple de dépendance dans une application qui cause le masquage de la fonctionnalité `foo` de la bibliothèque *libfoo.so.1.0* par la bibliothèque *libfoo.so.1.1*

L'analyse des interactions entre des bibliothèques au sein d'un même programme permet de mieux comprendre quelles sont les circonstances qui peuvent conduire à des comportements non désirés, comme le masquage de fonctionnalité présent dans l'exemple de la

figure 3.4. Cela permet aussi d'établir les conditions qui permettent d'éviter ces comportements non désirés.

3.2.1. Coexistence entre les bibliothèques

Les bibliothèques coexistent dans deux contextes principaux, de façon passive dans un système de fichier et de façon active durant l'exécution d'un programme. Un système de fichier contient une arborescence hiérarchique de répertoires et de fichiers avec une seule racine. Un fichier est l'entité dans un système de fichier qui contient le code des bibliothèques. Les répertoires sont les entités qui permettent de regrouper plusieurs fichiers de façon logique. La racine correspond au sommet de la hiérarchie du système de fichier. La façon de référer à un fichier dans un système de fichier est d'utiliser le chemin absolu. Cela correspond à la liste des répertoires à parcourir de la racine jusqu'au fichier. L'outil responsable d'organiser les bibliothèques sur un système est le *package manager*. Il a plusieurs objectifs incluant l'installation de bibliothèque, la mise à jour des bibliothèques installées, la désinstallation de bibliothèque et la résolution des dépendances.

Un cas intéressant de coexistence entre bibliothèques est celui qui inclut plusieurs versions d'une bibliothèque. Les problèmes peuvent être liés à d'organisation sur le système de fichier et au masquage de fonctionnalité durant l'exécution d'un processus. Il y a aussi plusieurs utilités de conserver plusieurs versions d'une bibliothèque, cela permet de supporter des applications qui dépendent de bibliothèques antérieures.

Une autre application est de convertir un vieux format de fichier vers un format plus récent. Dans le cas où il n'est pas possible d'avoir en même temps plusieurs versions d'une bibliothèque, il faut alors écrire un parseur pour lire le vieux format ensuite utiliser les fonctions de la version cible de la bibliothèque pour générer le nouveau format du fichier. Cette solution est clairement laborieuse et susceptible à des erreurs de codage.

3.3. Conditions de coexistence

Des conditions suffisantes pour que deux bibliothèques puissent coexister ensemble au sein d'un processus incluant deux versions de la même bibliothèque sont l'absence d'état global ou local, et l'unicité des noms des fonctionnalités. L'absence d'état garantit que chaque fonction de la bibliothèque retourne toujours un résultat prévisible dans un contexte identique. Un cas

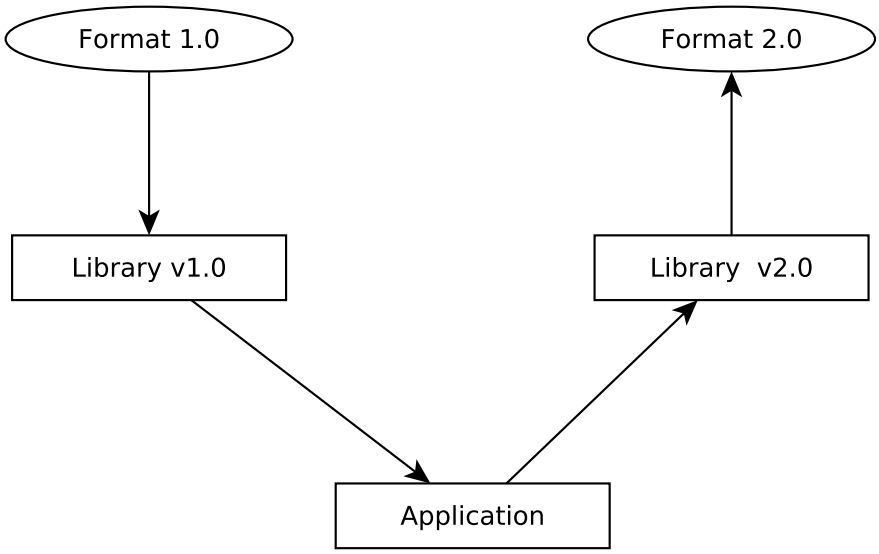

Figure 3.5. Un exemple d'application de conversion entre deux versions d'un format de fichier comme sqlite2 et sqlite3 exploitant la possibilité de charger plusieurs versions d'une bibliothèque.

extrême est la pureté fonctionnelle, qui est d'autre part avantageuse dans un contexte *multi-threadé*, car il n'est pas possible d'avoir une condition de course sur une donnée partagée. La pureté fonctionnelle garantit qu'il n'y a pas d'affectation et donc pas d'état. Le fait que chaque fonctionnalité soit associée à un nom unique qui empêche une bibliothèque de masquer une fonctionnalité d'une autre bibliothèque.

Ces conditions sont suffisantes pour que deux bibliothèques coexistent sans problème, mais elles ne sont pas nécessaires. Il existe des bibliothèques qui ne respectent pas ces conditions, mais coexistent avec d'autres versions de la bibliothèque. Pour tester la coexistence entre plusieurs bibliothèques, des expériences ont été effectuées dans plusieurs systèmes de modules existants. Le but de l'expérience est d'observer le bon fonctionnement des bibliothèques au sein du processus.

Les expériences qui suivent permettent d'observer et d'identifier les cas de cohabitation de bibliothèques au sein d'une application qui cause des problèmes. Ils permettent aussi d'identifier les capacités d'un langage à charger plusieurs versions d'une bibliothèque.

3.3.1. Bibliothèque C

Les bibliothèques dans le langage C sont compilées dans un format natif pour la plateforme courante (e.g. Window, macOS, Linux). Leur format diffère d'un système d'exploitation à un autre, Window utilise le format *dll* (*dynamic loading library*), macOS utilise le format *dylib* (*Mach-O dynamic library*) et Linux utilise le format *so* (*shared object*).

Les bibliothèques C consistent de symboles qui correspondent aux fonctions et variables globales exportées. Une bibliothèque est généralement liée à un programme C lors de la création du programme. Les symboles non définis sont résolus durant l'exécution du programme.

Les collisions de symboles entre deux bibliothèques causent un masquage de fonctionnalité d'une des deux bibliothèques. Un problème est de savoir si cela est possible de charger deux bibliothèques avec des collisions de symboles et accéder aux fonctionnalités distinctes de ces bibliothèques.

Pour charger deux versions d'une même bibliothèque en C, il faut utiliser un moyen qui prend en compte des symboles communs. La résolution des symboles se fait par un parcourt en largeur dans la liste des symboles des bibliothèques. Le résultat est le premier objet qui correspond au nom de symbole recherché. L'ordre des bibliothèques utilisé dans la résolution des symboles correspond à l'ordre spécifié lors de la construction de l'exécutable.

```
> gcc -lSDL -lSDL2 exemple.c -o exemple
> ldd ./exemple
libSDL.so => /usr/lib/libSDL.so
libSDL2.so => /usr/lib/libSDL2.so
```

Figure 3.6. Création d'un exécutable lié aux deux bibliothèques SDL et SDL2 dans cette ordre.

La figure 3.6 donne la procédure de création d'un exécutable lié avec deux versions de SDL. Les tests ont été effectués sur deux versions de SDL, 1.2 et 2. Ces versions peuvent utiliser des ressources communes comme les évènements du clavier, la mémoire et le GPU en utilisant OpenGL pour faire de l'affichage 2D ou 3D. Plusieurs situations sont testées, chacune sans thread et avec des threads :

(1) Une utilisation de SDL minimaliste qui n'écoute pas les événements utilisateurs.

(2) Une utilisation de SDL avec une boucle d'événements de base.

(3) Une utilisation de SDL avec une utilisation d'un contexte OpenGL.

Puisque SDL1.2 et SDL2 ont des collisions de symboles (e.g. `SDL_Init`, `SDL_FillRect`, `SDL_BlitSurface`, ...), l'une des premières observations à effectuer est la bonne répartition des appels de fonctions des deux bibliothèques dans le contexte d'une même application. Le problème à identifier est un appel invalide à une procédure de SDL2 qui était destiné à SDL1.2. Le facteur qui influence laquelle des deux bibliothèques va masquer l'autre est l'ordre dans laquelle elles sont liées au programme à la création, qui est montrée à la figure 3.6.

Le masquage est un problème qui peut mener à une défaillance du programme, car cela peut provoquer la transmission d'une structure incompatible de la première bibliothèque à la deuxième. Dans le cas de SDL, la structure `SDL_Surface` à une disposition différente des champs, donc incompatible. Le premier champ qui décale l'alignement de ces deux structures est le *pitch*, qui dans SDL1.2 est déclaré en Uint16 alors qu'en SDL2 il est un int qui sont des types de taille différente. Donc, l'accès au prochain champ `pixels` est différent entre SDL1.2 et SDL2, ce qui peut causer un accès non désiré en mémoire si une structure de SDL1.2 est accédée comme une structure de SDL2.

La cohabitation entre deux bibliothèques conflictuelles au sein d'une application n'est pas possible en lien l'application aux bibliothèques durant sa construction 3.6. Il est peu probable d'avoir une cohabitation entre deux bibliothèques avec des noms de fonctionnalités similaires au sein.

La façon d'avoir plusieurs bibliothèques conflictuelles au sein d'un programme

Le cas qui pourrait permettre plusieurs bibliothèques avec des collisions de symboles au sein d'une même est avec l'API du *dynamic linker*. Un test simple permet de démontrer la capacité de répartir les appels d'une fonction identifiés par le même nom dans différentes bibliothèques.

La structure générale des tests est organisée comme suit. Deux bibliothèques implémentant une interaction valide avec l'une des versions de la bibliothèque (e.g. SDL1.2, SDL2). Une application qui unit ces deux bibliothèques en utilisant l'API du *dynamic linker* pour exécuter ces deux bibliothèques séquentiellement ou parallèlement.

```
typedef struct SDL_Surface {
    Uint32 flags;
    SDL_PixelFormat *format;
    int w, h;
    Uint16 pitch;
    // Different offset here
    void *pixels;
    ...
} SDL_Surface;
```

```
typedef struct SDL_Surface {
    Uint32 flags;
    SDL_PixelFormat *format;
    int w, h;
    int pitch;
    // Different offset here
    void *pixels;
    ...
} SDL_Surface;
```

Figure 3.7. C'est la comparaison entre la structure **SDL_Surface** de SDL1.2 et SDL2 respectivement.

Dans le test minimaliste sans gestion d'évènements, l'exécution des bibliothèques fonctionne dans les deux cas, séquentiellement et en parallèle. La raison qui explique ce bon fonctionnement est la répartition des appels aux fonctions des deux bibliothèques et le fait qu'il n'utilise pas de structure commune, qui pourrait causer des conditions de course dans le test en parallèle.

Dans le test incluant des évènements, l'hypothèse supposait que les évènements du clavier auraient causé des problèmes de conditions de course en parallèle. Le test a aussi fonctionné séquentiellement et en parallèle, malgré la dépendance commune du clavier.

Le test qui incluait OpenGL, n'aurait pas dû fonctionner par hypothèse. Les deux versions de SDL référaient à une unique version de OpenGL. Le résultat a été surprenant, car le test séquentiel et parallèle à fonctionné. Ce qui amène à conclure qu'il existe des cas de coexistence entre des bibliothèques avec des collisions de symbole qui fonctionne sans causer des problèmes d'exécution.

```
#include <SDL/SDL.h>

int main(int argc, char **argv) {
  SDL_Init(SDL_INIT_VIDEO);
  SDL_WM_SetCaption("sdl1_2", NULL);
  SDL_Surface *win =
    SDL_SetVideoMode(200, 200, 24, SDL_HWSURFACE);

  Uint32 color = SDL_MapRGB(win->format, 255, 0, 0);

  SDL_FillRect(win, NULL, color);
  SDL_Flip(win);
  SDL_Delay(1000);

  SDL_FreeSurface(win);
  SDL_Quit();
  return 0;
}
```

Figure 3.8. Programme qui utilise la bibliothèque SDL1.2 sans la gestion des évènements Cette application génère une fenêtre rouge qui se ferme après 1 seconde de délais.

```
#include <SDL2/SDL.h>


int main(int argc, char **argv) {
  SDL_Init(SDL_INIT_VIDEO);
  SDL_Window *win = SDL_CreateWindow("title",
      SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
      200, 200, SDL_WINDOW_SHOWN);
  SDL_Surface *screen = SDL_GetWindowSurface(win);


  Uint32 color = SDL_MapRGB(screen->format, 0, 255, 0);
  SDL_FillRect(screen, NULL, color);
  SDL_UpdateWindowSurface(win);
  SDL_Delay(1000);
  SDL_DestroyWindow(win);
  SDL_Quit();
  return 0;
}
```

Figure 3.9. Programme qui utilise la bibliothèque SDL2 sans la gestion des évènements. Cette application génère fenêtre verte qui se ferme aussi après 1 seconde de délais.

3.3.2. Bibliothèque Scheme avec FFI

La compatibilité entre versions a été testée sur la partie cryptographique RSA de OpenSSL, SDL et un générateur aléatoire qui possède un état global. Pour observer le comportement de bibliothèque Scheme liant des fonctionnalités implémentées dans un autre langage, dans ces exemples C. Ces tests montrent le problème de masquage décrit précédemment dans des bibliothèques réelles. Pour tester les bibliothèques natives en Scheme, il a fallut écrire du code spécial pour lier les types et les fonctions à des procédure Scheme. Cela a été fait par la FFI de Gambit.

Par hypothèse, la partie cryptographique des deux versions de OpenSSL peut coexister. Chacune des fonctions cryptographiques de OpenSSL1.0 et OpenSSL1.1 exécute du code disjoint qui n'interfère pas avec une zone mémoire commune. Cela implique qu'un appelle à une fonction de OpenSSL1.0 n'affectera pas les résultats des appelles futures de OpenSSL1.1. Les fonctions nécessaires pour tester la partie cryptographique RSA sont :

— `PEM_read_bio_RSA_PUBKEY` et `PEM_read_bio_RSAPrivateKey` : Ce sont les fonctions OpenSSL qui permettent de lire la clef publique et la clef privée.

— `RSA_public_encrypt` : La fonction de cryptage qui crypte un message en utilisant une clef publique.

— `RSA_private_decrypt` : La fonction qui permet de décrypter un message en utilisant la clef privée.

— `RSA_free` : La fonction qui libère la mémoire allouée pour les clefs publiques et privées. Cette fonction n'est pas nécessaire pour les tests, mais utile pour une bonne gestion mémoire.

La structure de donnée utilisée par les fonctions d'OpenSSL pour la cryptographie RSA est un pointeur vers un enregistrement RSA. La liaison avec le type C est effectuée avec la forme spéciale `c-define-type`.

Le test consiste à crypter un message en utilisant une clé publique, décrypter le cryptogramme avec la clé privée et comparer le message original avec le message décrypter pour les deux versions de OpenSSL chargé dans le même processus. Pour vérifier la bonne invocation des fonctions de OpenSSL1.0 et OpenSSL1.1 respectifs, le débogueur **gdb** a été utilisé. Les bibliothèques d'OpenSSL consistent en deux fichiers `libssl.so` et `libcrypto.so`. En déboguant la répartition des appels avec *gdb*, il y a eu l'observation que la bibliothèque

`libcrypto.so` de OpenSSL1.1 masquait la version de OpenSSL1.0 s'il n'est pas spécifié explicitement lors de la création de la bibliothèque Scheme *rsa.scm* comme à la figure-3.10. Les raisons sont liées à plusieurs facteurs qui étaient présents durant l'expérience. L'ordre de recherche des symboles qui inclue les symboles des bibliothèques liés à l'exécutable qui sont prioritaire qui contenait `libssl.so` et `libcrypto.so` de OpenSSL1.1. Dans ce cas, il était possible de résoudre ce problème en ajoutant une dépendance directe à *libcrypto* lors de la construction des bibliothèques Scheme `RSA` comme montrée dans la figure 3.11.

```
gsc -o rsa-1-0-0.o1 \
  -cc-options '-I /usr/include/openssl-1.0' \
  -ld-options '-L /usr/lib/openssl-1.0 -lssl' \
  -prelude '(define-cond-expand-feature|openssl-v10|)' rsa.scm

gsc -o rsa-1-1-0.o1 \
  -ld-options "-lssl" rsa.scm
```

Figure 3.10. Création des bibliothèques *rsa.scm* pour OpenSSL1.0 et OpenSSL1.1 sans la spécification de *libcrypto.so*

```
gsc-script -o rsa-1-0-0.o1 \
  -cc-options '-I /usr/include/openssl-1.0' \
  -ld-options '-L /usr/lib/openssl-1.0 -lssl -lcrypto' \
  -prelude '(define-cond-expand-feature|openssl-v10|)' rsa.scm

gsc-script -o rsa-1-1-0.o1 \
  -ld-options "-lssl -lcrypto" rsa.scm
```

Figure 3.11. Création des bibliothèques *rsa.scm* pour OpenSSL1.0 et OpenSSL1.1 avec la spécification de *libcrypto.so*

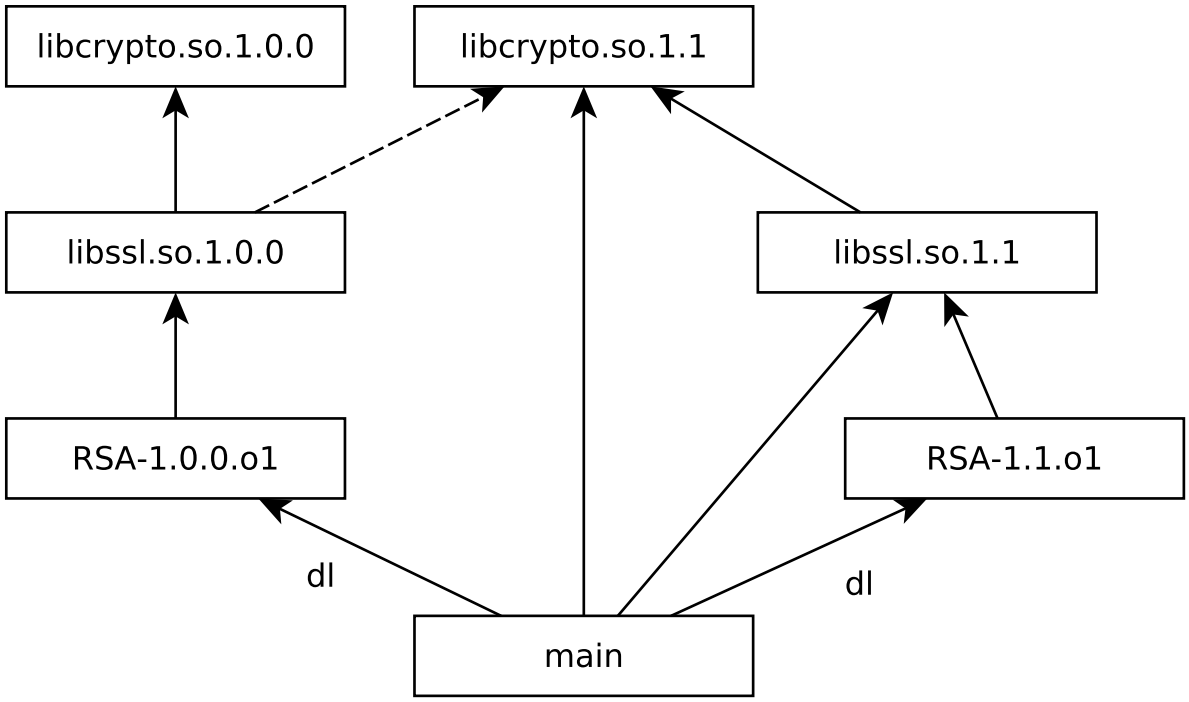

Figure 3.12. C'est un schéma des dépendances entre les bibliothèques au sein d'un processus. Les dépendances qui ont été liées dynamiquement lors la création de l'application sont représentés par une flèche avec trait plein sans annotation. Ceux qui représentent les chargements de bibliothèque dynamique via `dlopen` ont l'annotation *dl*. La flèche en pointillé indique un masquage des symboles de la bibliothèque source par une ligne pointée.

3.3.3. Bibliothèque JavaScript (NodeJS)

Pour effectuer des tests sur la coexistence de différente version d'une bibliothèque JavaScript dans NodeJS, il faut tout d'abord permettre l'importation de plusieurs versions d'une même bibliothèque. Dans NodeJS, l'importation de modules s'effectue avec la fonction `require(module-name)`. Puisque l'information de version de la bibliothèque n'est pas fournie en paramètre à la fonction, il faut donc utilisé une autre méthode de forcer plusieurs versions des bibliothèques. La configuration d'un module dans NodeJS utilise le format JSON pour spécifier le nom, la version, les dépendances ....

Les dépendances sont conservées sous la forme d'un arbre, chaque module à ses dépendances directes qui ont aussi des dépendances indirectes. Lors de l'écriture d'un module

JavaScript, il est possible de spécifier la version de chaque dépendance dans le fichier *package.json*.

```
{
  ...
  "dependencies": {
    "express": "4.16.3"
  },
  ...
}
```

En utilisant cette fonctionnalité du système de module de NodeJS, deux bibliothèques *wrapper* sont écrites pour interfacer les deux versions de express. Puisque l'API public d'express n'a pas changé entre les versions 3.21.1 et 4.16.3, il est possible de recycler le code de la bibliothèque qui encapsule une version d'express (figure-3.3.3).

```
const express = require('express');

function start() {
  const app = express();
  const port = Math.floor(Math.random() * 64535 + 1000);

  app.get('/', (req, res) => {
    res.send('Hello␣world!\n');
  });

  app.listen(port, 'localhost', () => {
    console.log('Listen␣on␣port␣' + port);
  });
}
exports.start = start;
```

Le programme principal ne fait qu'importer les deux encapsulations de bibliothèque et invoque la fonction *start*.

Le résultat attendu dans cette expérience est que ces deux versions de la bibliothèque express coexistent sans problème, sauf dans le cas où le port tcp utilisé par les deux versions est le même. Dans ce cas, c'est la bibliothèque dont la fonction *start* a été invoqué en premier qui va monopoliser le tcp port. Dans ce cas la ressource qui inhibe la coexistence de ces modules au sein d'un même processus est liée au *socket*.

3.3.4. Variables globales communs

Puisque qu'il n'existe qu'une seule instance de chaque bibliothèque en mémoire, cela implique que les variables globales d'une bibliothèque.

Définissons 3 bibliothèques B, C et D telle que D a une variable globale nommée *value*. B et C ont chacun une dépendance directe vers D, et exportent une référence de la variable globale *value* de D.

Le programme principal A commence par charger B et C. Ensuite lit la valeur de *B.D.value* puis modifie *C.D.value* et relit *B.D.value*. Le test réussi si la valeur de *B.D.value* reste inchangée par la modification de *C.D.value*. Cela implique que les références vers la bibliothèque D est différentes de via B et via C.

Dans NodeJS, un module peut être installé via un dossier, une archive tarball, un dépôt de code source git ou directement via Npmjs. Puisqu'un module publié dans Npmjs ne peut pas être retiré facilement étant donné la politique liée au module (`https://docs.npmjs.com/cli/unpublish`).

3.4. Conclusion

TODO!!!

Chapitre 4

Implémentation des modules

La syntaxe des modules se veut portable entre les différentes implémentations de Scheme. Gambit est un système qui se veut conforme R7RS. La syntaxe choisit pour les modules, est celle qui utilise `define-library` dans R7RS. Cette forme offre plus de flexibilité que la forme `library` dans R6RS. Le standard R7RS est une simplification du standard R6RS qui offre des fonctionnalités équivalentes.

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 dans lequel les bibliothèques ont été écrites.

4.1. La forme `##namespace`

Les espaces de nom sont gérés avec une forme spéciale propre à Gambit. Cette forme se nomme `##namespace` et permet d'associer des identifiants à d'autres identifiants. 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 ne respecte pas cette règle, c'est l'espace de nom par défaut. Les associations de

symbole données par la forme `##namespace` respectent la portée lexicale. Il y a trois types d'opérations avec les espaces de nom.

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

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

Figure 4.1. 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 du 4.1 il suffit d'ajouter les symboles après le nom de l'espace de nom.

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

Figure 4.2. Namespace Set

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

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

Figure 4.3. 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 4.1, 4.2 et 4.3.

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

Figure 4.4. Module Hello

L'exemple 4.4 est un exemple d'utilisation de la forme `##namespace` pour créer un espace pour le module `hello`. La procédure `hello` est dans l'espace de nom `hello#`.

4.2. 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 `##demand-module` est une macro qui 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 des `##demand-module` correspond à l'ordre dont les modules sont visités. Cette forme agit, peu importe son emplacement où l'expansion de macros est appliquée.

Une forme spéciale conjointe au `##demand-module`, qui indique le nom symbolique des modules importés 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 4.5.

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

Figure 4.5. Syntaxe `demand-module` et `supply-module`

4.2.1. Les méta informations

Un module a des informations qui sont utilisées 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 associer 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 lié au module Scheme. Les options nécessaires pour au compilateur C sont déterminés 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`.

4.3. 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.

— *<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))
```

Figure 4.6. Écriture d'un module qui implémente une pile. Ce module est séparé en 2 fichiers. Le fichier `stk#.scm` qui contient les exportations et `stk.scm` qui contient les implémentations des fonctions.

Dans l'exemple 4.6, il y a dans `angle2/angle2.scm` l'inclusion de 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` ne corresponde pas à celui utilisé dans `angle2/angle2.scm`.

La forme `##namespace` dans l'exemple 4.6 s'applique aux identifiants suivants :

```
factor    --> stk#factor
deg->rad  --> stk#deg->rad
rad->deg  --> stk#rad->deg
```

4.3.1. 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 stk)
```

Figure 4.7. Expansion de `(##import angle2)`

4.4. Implémentation des modules R7RS

Pour que le système de module soit compatible avec d'autres implémentations Scheme, les modules haut niveau sont définis dans le standard R7RS Small [**19**]. Les modules sont définis par la forme `define-library` la syntaxe est donnée par la figure 2.6. Les formes `define-library` et `import` utilise les formes spéciales utilisées par les modules primitifs. L'élément qui distingue un module primitif et un module R7RS est l'utilisation de la forme `define-library`.

4.4.1. Expansion du `import`

La façon que la forme `import` est expansée 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 s'il est d'un `define-library` d'un programme principal. Dans le cas d'un `define-library` le `import` supporte l'importation relative, qui est une extension de Gambit.

4.4.1.1. *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 présentement. Le

`import` R7RS se rabat sur le `##import` des modules primitifs qui ne supporte pas les extensions R7RS. Cela permet d'utiliser des modules primitifs dans un contexte R7RS.

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

Figure 4.8. Expansion du `import` d'un module primitif

4.4.1.2. *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 du 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` affecte le nom des identifiants. Dans l'exemple 4.9, 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 4.10.

4.4.2. Expansion du `define-library`

La forme `define-library` est expansé 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

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

Figure 4.9. L'exemple de l'expansion du `import` du module R7RS `github.com/gambit/hello` qui exporte les procédures `hello` et `hi`

```
;; (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))

;; expansion of (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 bonjour)))
(##namespace ("github.com/gambit/hello#" (howdy hi) salut))
```

Figure 4.10. Différent `##namespace` généré par l'expansion du `import` d'un module R7RS.

,

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 utilisé lors l'expansion du `define-library`, mais lors de l'importation de cette bibliothèque. Les déclarations `import` sont expansées de la même façon que dans contexte 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")))
```

Figure 4.11. C'est le code source du module `github.com/gambit/hello` avant l'expansion.

```
;; 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 (""))
```

Figure 4.12. Expansion de la forme `define-library` du module `github.com/gambit/hello`.

4.4.2.1. *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`.

— `namepase`

— `cc-options`

— `ld-options` et `ld-options-prelude`

— `pkg-config` et `pkg-config-path`

La figure 4.13 est un exemple d'importation relatif. L'importation relative part :w du `module-ref` du module courant. Un `import` de `(.. C)` à partir du module `(A B)` correspond à l'importation de `(A C)`. C'est pour permettre 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))
```

Figure 4.13. Importation relatif du module `(A C)`

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.

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.

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

Figure 4.14. Implémentation de la bibliothèque système `(scheme case-lambda)`.

4.5. Conclusion

TODO!!!

Chapitre 5

Modèle de chargement

Un système est composé d'un ensemble d'éléments (modules) qui interagissent entre eux. L'interaction entre les module est dans un contexte statique ou dynamique. Dans le contexte statique les modules sont incorporés au sein de l'application. Dans le contexte dynamique les modules sont externe à l'application.

5.1. Chargement des bibliothèques

Le chargement d'une bibliothèque Scheme (ou module) dans Gambit est séparé en plusieurs niveaux. Il y a la phase recherche du module et de ces dépendances qui valide la présence de toutes les modules nécessaire. Ensuite le module et ses dépendances sont chargés dans un ordre qui suit les relations de dépendance. Un module est soit sur le disque ou déjà en mémoire. L'emplacement des bibliothèques sur le système de fichier est lié par défaut aux chemin spécifié par le `##module-search-order` a comme défaut `~~lib` et `~~userlib`.

La procédure exacte de chargement des bibliothèques par `import` n'est pas spécifier par le standard R7RS. Le standard spécifie seulement une syntaxe de base et le de comportement principal qui est requis. L'importation d'une bibliothèque doit chargé la bibliothèque et rendre c'est fonctionnalité disponible dans le contexte l'importation a eu lieu qui peut soit ovenir d'un programme principale ou d'une bibliothèque.

Le chargement d'une bibliothèque peut-être effectuée 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 lien en utilisant les méta-infos contenus dans les **.c** qui sont chacun compilé par le compilateur C en **.o** et lié par le *linker*.

5.2. Modèle statique

Le modèle de bibliothèque statique consiste à intégrer la bibliothèques dans l'application finale. La bibliothèque n'a pas besoin d'être chargé durant l'exécution. L'avantage du modèle statique est le déploiement. Puisque tout les dépendances sont dans l'application finale¸ il suffit de distribué celle-ci. Le compilateur Gambit support un modèle statique pour des programme simple. Compiler un application lié statiquement est possible de façon manuelle. Pour lié statiquement deux fichiers Scheme simple il suffit d'invoquer le `gsc` comme suit :

```
$ gsc file1.scm file2.scm
```

Un des problèmes du modèle statique est le coût lié à la maintenance. Les programmes qui utilise 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 évolutifs qui peuvent être étendu par l'utilisateur. Une solution qui a été adopté est le modèle dynamique 5.3. Cela offre une plus grande liberté dans la conception des programmes.

5.3. Modèle dynamique

Dans le modèle dynamique chaque module est séparé durant l'exécution. L'application contient les informations pour extraire les fonctionnalités des modules durant l'exécution. Le déploiement d'un application nécessite la distribution de toutes les dépendance. Les bibliothèques partagés offre plusieurs avantages par rapport aux bibliothèques statique.

Dans ce modèle les bibliothèques sont lié au programme durant l'exécution. Cela nécessite que les bibliothèques soit organisé 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 effectué dans un ordre spécifique indépendant de la spécification. L'algorithme de recherche les bibliothèques prend entré le nom de la bibliothèque et retourne le chemin absolu correspondant à sont emplacement dans l'arborescence du système de fichier. Les bibliothèques sont situées dans différents répertoires l'origine du programme, le répertoire des bibliothèques système (`~~lib`) et le répertoire de bibliothèque utilisateur (`~~userlib`).

Chaque module possède trois niveaux d'initialisation dans le système numérotés de 0 à 2. Le niveau 0 indique que le module est collecté, mais non initialisé. Le niveau 1 indique que le descripteur du module à été récupéré. Le niveau 2 marque les module chargé qui sont près à être utilisé.

```
> (define mods (##collect-modules '(_zlib)))
> mods
(#(_digest
   0
   #(#(_digest) #() () 1 #<procedure #2> #<foreign #3 0x7f448797ec00>))
 #(_zlib
   0
   #(#(_zlib) #(_digest) () 1 #<procedure #4> #<foreign #5 0x7f44879f6740>))
 )
> (##init-modules mods)
#t
> ##registered-modules
(#(_digest 2 #(#(_digest) #() () 1 #<procedure #2 _digest#> ...))
 #(_zlib 2 #(#(_zlib) #(_digest) () 1 #<procedure #4 _zlib#> ...))
 ...)
```

Figure 5.1. 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 éffectuée par la procédure `##collect-module`. L'ensemble des modules retournés sont initialisés par la procédure `##init-modules`. Les enregistrements des modules `_zlib` et `_digest` sont montrer par la variable `##registered-modules`.

Dans l'exemple 5.2, 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 apparaisse plusieurs fois au sein du graphe. Une fois la collection de tous ces modules est complété le descripteur de module est récupéré par un appels aux primitives du système d'exploitation si compilé.

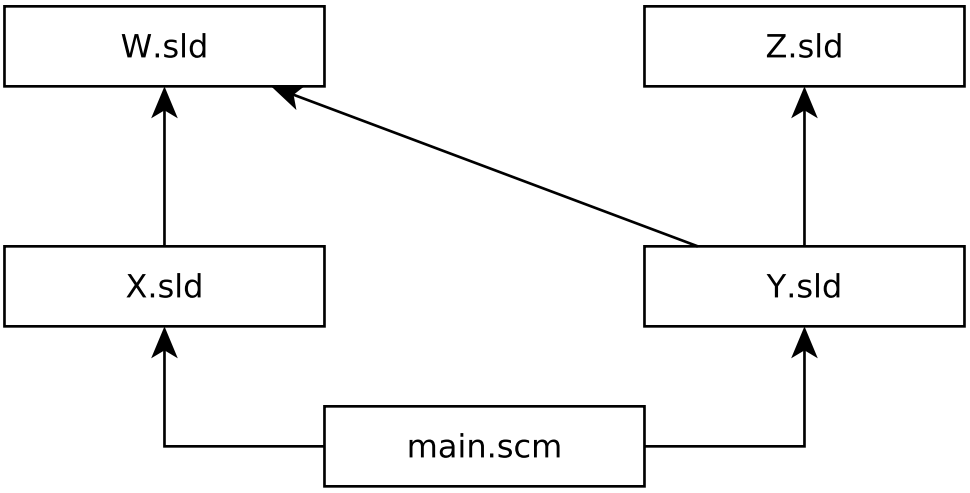

Figure 5.2. Un exemple d'un système fictif composé de différents modules. Le module principale se nomme **main.scm** avec l'extension **.scm** et les bibliothèques ont l'extension **.sld**

5.3.1. Module compilé dans Gambit

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

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

Figure 5.3. Un module qui implémente la fonction mathématique `fib`.

La construction d'un bibliothèque dynamique à partir du fichier `fib.scm` de la figure 5.3 s'effectue par le compilateur de Gambit qui nomme `gsc`. Cela produit un fichier avec l'extension `.oN` où le `N` correspond à la version généré de la bibliothèque qui commence à 1.

5.4. Module hébergé

Un module qui est hébergé est un module qui dont son contenu se retrouve sur un domaine comme `github.com`. La syntaxe des noms de domaine est inspiré du RFC-2396 [**3**]. La différence avec la spécification du *hostname* dans le RFC-2396 est que le *hostname* ne peut pas finir par un point et doit contenir au moins un `domainlabel`. C'est pour permettre de distingué 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]
```

Listing 5.1. Grammaire BNF représentant un hostname selon un sous ensemble du RFC-2396.

5.4.1. Installation automatique

L'installation automatique d'un module dépend du contenu de la *whitelist*. La *whitelist* contient une liste de préfixe utilisé pour valider les *<module-ref>*. La validation consiste a trouvé un élément dans la *whitelist* qui correspond à un préfixe du *<module-ref>*.

Un élément est considéré un préfixe si chaque composants sont inclus 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 *gihtub.com/gambit*, car *gambit* est différent de *gambitXYZ*.

Par défaut `github.com/gambit` est inclus dans cette liste. Cela implique que tous les modules sous `github.com/gambit` peuvent être installé automatiquement. La *whitelist* peut être modifié par un option de l'interprète :

```
$ gsi -:whitelist=github.com/foo
> (##os-module-whitelist)
("github.com/gambit" "github.com/foo")
```

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

- **never** : il y a seulement les modules qui sont dans la *whitelist* qui peuvent être installé.
- **repl** : les modules qui ne sont pas dans la *whitelist* peuvent être installé avec la confirmation textuelle de l'usager s'il y a une *repl*.
- **ask-always** : les modules qui ne sont pas dans la *whitelist* peuvent être installé avec la confirmation textuelle de l'usager.

Le mode d'installation est spécifié par le *runtime option* nommé *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)
>
```

Figure 5.4. 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 installé.

Chapitre 6

Gestion des modules

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 présent dans beaucoup de langage telle 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 inclus les modules systèmes et les modules utilisateurs. La gestion des modules est effectuée par le module `_pkg` qui offre les procédure d'installation d'installation et de désinstallation.

6.1. Organisation des modules

Les modules sont organisés dans des répertoires données 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 courant.

— `~~lib` : c'est le répertoire d'installation de Gambit qui contient les modules système. Les modules dans ce répertoire sont commun 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 champs `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. La syntaxe du nom de domaine est donnée par la grammaire[5.4]. La grammaire formelle6.1 décrit la syntaxe du `<module-ref>` dans le cas hébergé et local.

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

Listing 6.1. Grammaire formelle

Le `module-ref github.com/gambit/hello` réfère au module `hello` sur le serveur de révision `github.com` dans l'espace `gambit`. Le champs `host` de est dans ce cas `github.com/gambit` qui correspond au nom de domaine avec le nom de l'espace sur le le serveur.

6.1.1. Installation de module

L'installation des modules peut être soit à partir d'un serveur de révision ou d'un répertoire local versionné par git. Dans les deux cas d'installation, le `module-ref` est utilisé pour déterminer l'emplacement du module dans le répertoire des modules. Chaque `module-ref` est associé à un réperoire unique dans le répertoire des modules.

Les modules sont hébergés sur des serveurs de version tel que github, gitlab, bitbucket, etc. Chaque version d'un module est installé dans un répertoire distinct. Il est donc possible d'avoir plus d'une version d'un module installé. L'installation des modules s'effectue par l'intermédiaire du module `_git` qui offre un interface à l'utilitaire `git`. Le processus d'installation est séparé en plusieurs étapes. Le contenu du module est installé dans un répertoire temporaire qui est ensuite renommé au répertoire de destination. Cela permet d'installer le module de façon atomique.

Tout d'abord, la branche `master` dépôt du module est cloné. Ensuite un archive de la version est faite et extraite dans le préfixe des modules. Le préfixe est par défaut `~~userlib`. Il est possible de spécifier un préfixe d'installation dans lequel installer les modules. Plusieurs versions d'un même module coexistent dans le même préfixe d'installation.

La branche `master` est utilisé comme version de développement et comme cache pour installer les autres versions. Une version d'un module est soit un commit une branche ou un étiquette. L'installation d'une version spécifique utilise la cache pour récupérer l'archive de la version demandé et l'extraire dans l'espace des module.

La procédure `install` de `_pkg` accepte deux paramètres : le nom du module et de façon optionnelle le préfixe d'installation. Elle retourne la valeur de vérité vrai (`#t`) si l'installation réussi, sinon faux (`#f`).

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

L'installation peut être effectué en passant l'option `-install` à l'interprète gambit. Cette option requière le nom du module et de façon optionnelle le préfixe d'installation.

```
$ gsi -install [-to <path>] module [...]
```

Le préfixe `<path>` est la racine utilisée pour installer les modules et est spécifier par l'option `-to`. 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*.

```
$ gsi -install -to /tmp/exemple github.com/frederichamel/semver@1.0.1
```

6.2. Désinstallation

La désinstallation d'un module consiste à supprimer les fichier de ce module. Le module `_pkg` offre la procédure `uninsall` qui accepte deux arguments : le nom du module et de façon optionnelle le répertoire dans lequel les modules sont situé. Les valeurs retournées par cette procédure sont similaire à la procédure.

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

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

```
$ gsi -uninstall [-to <path>] module
```

Le répertoire <path> est l'emplacement des module à désinstallé. 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 dans installé dans le répertoire /tmp/exemple est fait par la commande suivante :

```
$ gsi -uninstall -to /tmp/exemple github.com/frederichamel/semver@1.0.1
```

6.3. Mise à jour

Cette opération actualise la branche master du module. Cela donne accès au nouvelle publication d'un module. 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.

```
$ gsi -update [-to <path>] module
```

6.4. Tests unitaires

Les tests unitaires exécutés sont dans dans un fichier conjoint au module. Gambit offre un module de test unitaire nommé gambit/test. Il contient plusieurs procédure pour tester le bon fonctionnement d'un module. Les tests unitaires pour un module nommé A est un fichier A-test.scm dans le répertoire du module.

```
$ 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.

6.5. Compilation d'un module

La compilation d'un module est fait 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 ressemble à :

```
|-- _digest@gambit409003@C
|   |-- _digest.c
|   `-- _digest.o1
|-- _digest#.scm
|-- _digest.scm
`-- test.scm

1 directory, 5 files
```

6.6. Comparaison avec d'autre système

L'organisation des module sur le système de fichier dans Gambit diffère de celui de OCaml, Python et NodeJS. Ils ne permettent pas l'installation de plus d'une version d'un module directement. Le système utilisé dans Go ressemble beaucoup au système implémenté dans Gambit.

Un système de module permet la coexistence de plusieurs version du même module sur le système de fichier s'ils sont considérés comme des modules différents. L'installation d'un version différente d'un module ne remplace pas la version déjà installée. L'organisation des bibliothèques est importante pour permettre cette coexistance.

La caractéristique que le système de bibliothèque doit avoir pour permettent plusieurs version d'une bibliothèque est un organisation qui permet de distinguer les différentes versions de la bibliothèque par un chemin unique. La plupart des système de module ne distinguer pas les version d'un même module et ne permette l'installation que d'une seul version. Les systèmes de module permettent la gestion de différent préfixe dans lequel 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.

	Multiple versions
OCaml	✗
Python	✗
NodeJS	✗
Java	✗
Go	✓

Figure 6.1. Une table qui compare différent système de module sur la capacité d'installer plusieurs version d'un module. Le système Go permet plusieurs version d'un module pour des version incompatible selon le sémantique de version. La version `1.0.0` coexiste avec la version `2.0.0`. La version récent `1.2.0` remplace la vielle version `1.0.1`.

6.6.1. Organisation de OCaml

Le système de gestion de bibliothèques d'OCaml se nomme OPAM. Ce système permet d'avoir plusieurs environnement distinct contenant chacun un ensemble de version des bibliothèques. Chaque environnement permet l'installation d'une version spécifique de chaque bibliothèque et est étiqueté avec un nom choisit par l'utilisateur. Un changement d'environnement est effectué par une requête de l'utilisateur `opam switch <envname>`. Il utilise le projet *mancoosi*, un projet Européen de recherche dans le 7e cadre de recherche (FP7) de la commission Européenne, pour gérer les contrainte de version, les dépendances optionnelles et la gestion des conflits. L'environnement par défaut est lié aux dépôts standard d'OCaml.

6.6.2. Organisation de Python

L'organisation des bibliothèques Python ne permet de stocker que la dernière version d'une bibliothèques. 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 version 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 installé. Le système de module ne permet pas de référer à deux version de la même bibliothèques.

```
>>> 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
```

Figure 6.2. 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 de d'installer des bibliothèques dans un environnement isolé des autre. L'avantage est qu'il est possible d'avoir une compatibilité avec des logiciel qui utilise des version de bibliothèques antérieur. Un inconvénient est qu'il n'y a pas un partage des versions de bibliothèques communes entre les différent environnement, 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.

6.6.3. 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 installé au niveau du projet. Cela implique que plusieurs projet qui utilise la même version de la bibliothèque vont avoir la même exemplaire de la bibliothèque.

La structure d'une bibliothèque dans NodeJS est décrit 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 publié et plusieurs autres méta données lié à la bibliothèque. Sous NodeJS les bibliothèques sont gérer par projet plutôt que globalement cela à comme avantage que chaque projet fonctionne avec ses versions des bibliothèques.

6.6.4. Organisation de Java

Les modules en Java sont nommé *package*. Le nom des modules utilise généralement l'inverse d'un url comme espace de nom. Par exemple, les noms des modules liés aux services Google vont débuté par `com.google`. Cette convention à pour but d'unifier les noms de module. La version des module n'est pas lié au nom du module dans le cas de Java. Il n'est pas possible de chargé deux module qui utilise le même espace de nom, comme deux version d'un même module.

6.6.5. Organisation de Go

L'organisation des bibliothèques dans Go est plus dans un contexte environnement dont la racine est spécifier par la variable d'environnement `GOPATH` avec un répertoire pour les exécutable compilé (`bin`), un répertoire contenant le code source des différent projets (`src`) et un répertoire pour les objets des modules installé (`pkg`). Chaque paire de système d'exploitation et d'architecture a son propre répertoire dans `pkg`.

Le système Go permet l'installation de plusieurs version d'un même module dans le même environnement en utilisant le service *gopkg.in*. Il y a deux syntaxes utilisées pour l'url des module 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)
```

Refer to go help golang

```
$GOPATH/
  - bin
    - ... binaries
  - src
    - github.com
      - UserName1
        - project1
        - project2
      - UserName2
```

```
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")
}
```

Figure 6.3. Un exemple qui montre l'importation de deux version d'un même module en Go. Le module **go-hello** version 2 exporte la fonction `Salut` qui n'existe pas dans la version 1.

```
        - projectA
        - projectB
  - pkg
    - linux_amd64
      - pkglist
        - objets
```

Chapitre 7

Migration de code

Les systèmes distribués sont constitué d'un ensemble de nœuds interconnecter de calculs. Les nœuds interagissent par l'envoie et la réception de message au sein d'un réseau de communication. Chaque nœud a but spécifique. Le *World Wide Web* est un exemple notable qui permet d'avoir un aperçu. Ils est composé de clients et de serveurs qui roulent des application clients et serveurs différent.

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

- **RPC :** Comment l'appelle distant à une procédure (RPC) implémenté quand l'envoyeur et le receveur ne sont pas conçu ensemble ?
- **Mise à jour de code :** Comment la mise à jour du programme d'un nœud est effectué lors d'un *bugfix* ou 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, a un architecture différente, . . . ?
- **Opération continue :** Comment éviter les interruptions dans les situations précédentes ?

Le langage Termite Scheme[**10**] a été conçu pour simplifier l'implémentation de systèmes distribués et fournit certaine solution au problèmes de ces système. 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 absent en Erlang

est la capacité d'envoyer une continuation en message. Termite Scheme est implémenté sur le système Gambit Scheme qui offre la une façon de sérialiser la plupart des objets Scheme incluant les procédures et les continuations.

La sérialisation de procédures est un outil utile dans 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 original de Termite avait certaines limitations lors de la sérialisation des procédures et des continuations. Dans le cas interprété les procédures et les continuations sont transmis sans problème entre les nœuds. Dans le cas compilé, il faut que chaque nœud possède la définition des procédures qui sont transmis.

7.1. Le langage 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 effectué par un système de boîte de message présent dans Gambit. Chaque thread possède une queue de message entrant.

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.

— 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 *pattern-matching* sur les message reçu du nœud courant.

7.2. *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 encodé par le nom de la procédure et un 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 correspond dans le cas des modules hébergés à l'url du dépôt content le code du module.

Dans le processus de désérialisation le nom de la procédure et le id est utilisé pour retrouvé 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.

7.3. Exemple de migration de code

L'exemple d'application est un horloge programmable implémenté en Scheme avec Termite. L'exemple montre un exemple de migration de tâche qui contient du code nom présent sur le nœud destination. Cette horloge offre un API simple :

— La modification du fuseau horaire est fait en envoyant un entier.

— Le message `timezone-get` permet de récupérer le fuseau horaire courant.

— Le message `update-code` permet la mise à jour du code du serveur à l'exécution.

— Tous les autres messages sont rejetés par le serveur.

Le code exécuté par la boucle principale est présent dans la figure 7.2. L'affichage de l'heure est actualisé à une fréquence constante. Le serveur démarre un nœud Termite qui exécute la boucle de la figure 7.2.

```
;; 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 "0.0.0.0" 3000))
(define (start)
  (node-init node)
  (display "\033[H\033[J")
  (clock-start 'clock-app)
  (wait-for (resolve-service 'clock-app)))

(start)
```

Figure 7.1. Le code du serveur qui configure la *hook* qui résout les références à 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
           (to (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))

          (msg
           (pp msg)
           (wait))

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

Figure 7.2. C'est le code de la boucle principal de l'horloge programmable.

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

(node-init local)
```

Figure 7.3. Configuration des nœuds qui est utilisé sur les clients.

```
(import (termite))

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

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

Figure 7.4. Un programme qui change le fuseau horaire utilise par l'horloge.

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

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

(!? (remote-service 'clock-app raspi4)
    (list 'update-code clock-thread-loop))
```

Figure 7.5. Un exemple de mise à jour du code du serveur en par le message `update-code`.

Chapitre 8

Évaluation

La gestion des modules permet plusieurs versions d'un module d'être installé. Au sein d'un application, il est possible de chargé plus d'une version d'un module dans le système.

Les performances du système de module est mesuré en utilisant le temps de téléchargement, de compilation et de chargement des modules. Les modules pour tester les capacités de migration de code sont basé sur certains *benchmark* standard de Scheme présent dans Gambit.

— Compiler ($\tilde{4}$00K)

— Scheme ($\tilde{4}$0K)

— Puzzle ($\tilde{4}$K)

Les tests de performance sont effectués sur du nœuds **A** et **B**. Le tests de migration consiste à migrer une tâche sur le nœud générique **B**. Le nœud générique ne connait aucun code qui est migré dessus.

8.1. Spécification des machines

L'ensemble des expériences ont été faites sur trois machine : arctic, gambit et tictoc.

8.2. Résultats

Les résultat lié au transfert est affecté par les pertes de paquets. La vitesse de transfert est affecté dans l'ordre des millisecondes. Cela affect 1/10 des transfert effectué dans les testes compilé.

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

Figure 8.1. La spécification de la machine arctic qui est utilisé comme nœud de destination dans l'ensemble des tests. Cette machine, nommé **Arctic** 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
Storage:             26GB (sdcard)
RAM:                 2GB
Swap:                2GB
C Compiler           gcc (Raspbian 8.3.0-6+rpi1) 8.3.0
Ethernet Speed       1Gbps
```

Figure 8.2. La spécification du CPU du Raspberry Pi utilisé dans les tests. Cette machine se nomme **tictoc**.

```
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
Storage:             1TB (SSD)
RAM:                 8GB
Network Speed:       1Gbps
```

	4K	40K	400K
RPC	1761.6(σ = 84.9)	4816.6(σ = 136.5)	48609.4(σ = 112.4)
Scheme à C	122.7(σ = 4.9)	589.6(σ = 4.9)	9335.7(σ = 15.7)
Compilation C	686.3(σ = 0.0)	3102.1(σ = 99.7)	36443.8(σ = 53.8)
Install	850.4(σ = 17.6)	918.6(σ = 82.3)	1192.3(σ = 67.4)
Transfert de module	58.1(σ = 63.8)	172.0(σ = 60.0)	133.5(σ = 77.4)
Exécution	9.3(σ = 4.8)	0.2(σ = 0.0)	1473.0(σ = 9.3)

Figure 8.3. Ce sont les temps d'exécution et transmission de module de différente taille entre les machines Gambit et Arctic. Les modules sont installés automatiquement sur Arctic lors de l'exécution.

	4K	40K	400K
RPC	23.6(σ = 0.9)	10.5(σ = 0.3)	1492.8(σ = 1.9)
Transfert de module	4.8(σ = 0.1)	5.7(σ = 0.1)	14.2(σ = 0.5)
Exécution	14.0(σ = 0.7)	0.2(σ = 0.0)	1473.5(σ = 1.6)

Figure 8.4. Ce sont les temps d'exécution et de transmission de module de différente taille entre Gambit et Arctic dans le cas ou les modules sont présents sur chaque nœud.

	4K	40K	400K
RPC	1851.8(σ = 93.6)	4885.8(σ = 167.9)	48656.7(σ = 112.8)
Scheme à C	123.3(σ = 5.0)	590.1(σ = 5.2)	9336.8(σ = 16.3)
Compilation C	686.3(σ = 0.1)	3068.8(σ = 98.5)	36444.8(σ = 54.6)
Install	869.8(σ = 56.6)	948.1(σ = 98.7)	1213.5(σ = 29.4)
Transfert de module	89.1(σ = 64.8)	194.0(σ = 66.1)	150.7(σ = 79.2)
Execution	10.2(σ = 5.5)	0.2(σ = 0.1)	1467.6(σ = 9.8)

Figure 8.5. Cette expérience est le même que 8.3, sauf que le transfert de module est fait entre un machine ARM (tictoc) et x86 (arctic).

	4K	40K	400K
RPC	92.6(σ = 15.2)	92.9(σ = 5.2)	1554.1(σ = 17.3)
Transfert de module	31.7(σ = 1.5)	32.9(σ = 1.4)	49.0(σ = 1.4)
Execution	19.8(σ = 0.0)	0.2(σ = 0.0)	1486.6(σ = 2.1)

Figure 8.6. Cette expérience est le même que 8.4, sauf que le transfert de module est fait entre un système ARM (tictoc) et x86 (arctic).

	4K	40K	400K
RPC	2123.0(σ = 152.4)	5219.7(σ = 152.9)	49377.5(σ = 154.4)
Scheme à C	123.3(σ = 5.2)	590.9(σ = 5.5)	9345.3(σ = 20.7)
Compilation C	686.3(σ = 0.1)	3094.8(σ = 100.4)	36440.8(σ = 47.8)
Install	895.6(σ = 133.8)	999.1(σ = 71.2)	1614.4(σ = 95.2)
Transfert de module	335.0(σ = 71.6)	449.2(σ = 61.9)	468.0(σ = 80.7)
Execution	10.4(σ = 5.6)	0.2(σ = 0.0)	1467.7(σ = 10.5)

Figure 8.7. Ce test est dans identique à 8.5 avec un réseau de $10Mbit/s$.

	4K	40K	400K
RPC	150.9(σ = 1.4)	978.2(σ = 1.5)	12397.7(σ = 5.2)
Serialization	0.2(σ = 0.0)	0.2(σ = 0.0)	0.1(σ = 0.0)
Deserialization	5.2(σ = 0.0)	10.6(σ = 0.0)	46.7(σ = 1.0)
Transmission	0.4(σ = 0.0)	0.7(σ = 0.0)	7.7(σ = 0.1)
Full-Transmission	5.8(σ = 0.0)	11.6(σ = 0.0)	54.6(σ = 1.0)
Execution	139.0(σ = 1.3)	957.3(σ = 1.3)	12274.7(σ = 5.0)

Figure 8.8. Cette expérience permet de comparer la performance RPC de Termite avant les modules dans un contexte interprété.

Chapitre 9

Conclusion

Ce mémoire a présenté un système de module spécialisé pour les systèmes distribués. Il permet la conception d'applications qui exploite la diffusion de modules entre les nœuds. Ce qui permet des d'effectuer des appels RPC et de la migration de code mobile vers un nœud destination générique. Chaque module diffusé a un nom unique basé sur l'URL du dépôt de code qui permet de le récupérer. La diffusion se fait par le nom des identifiants qui contient l'URL du module. Le nommage des modules ce système ressemble à celui du langage de programmation Go.

Nous avons commencé par exposer les limitations du système Termite Scheme par des expérimentations dans plusieurs situations. Dans le contexte purement interpréter, la migration fonctionnait parfaitement même dans le cas ou le nœud 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 de l'agent mobile sur l'ensemble des nœuds.

Nous avons exploré plusieurs méthodes de chargement automatique des modules. Le langage Scheme permet le chargement de module à la demande.

Ce qui nous a menés à un constat que le nom des identifiants transmit manquait d'information et devait être unique au sein du système distribué. Les modules présents, basés sur R5RS, n'offraient pas la possibilité d'avoir des identifiants uniques.

Nous avons ajouté une forme spéciale à Gambit pour permettre la création de modules.

Le projet a mené au système de modules spécialisés pour les systèmes distribués à commencer par des expérimentations avec Termite. Le résultat présent est prometteur, il permet

le déploiement de serveurs sur plusieurs machines d'architecture et de systèmes d'exploitation différents.

Le système Gambit-C a été amélioré par l'ajout des modules primitifs et aussi des modules 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 lorsque demandé. Les performances du chargement de module sont amorties 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 l'amortissement surpasse la version interprétée.

La majorité du temps de module permet des temps de diffusion et d'exécution plus courts.

Bibliographie

[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, 1998.

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

[5] W. CLINGER, J. REES et *et al* : Revised^4 report on the algorithmic language scheme, 11 1991.

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

[7] Marc Feeley : Feature-based conditional expansion construct, 1999.

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

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

[10] Guillaume Germain : Concurrency oriented programming in termite scheme. *In* Feeley et Trinder [**8**], page 20.

[11] 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 2–2, Berkeley, CA, USA, 1996. USENIX Association.

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

[13] 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.

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

[15] R. KELSEY, W. CLINGER, J. REES et *et al* : Revised^5 report on the algorithmic language scheme, 2 1998.

[16] 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.

[17] Stefan M, Raymond BIMAZUBUTE et Herbert STOYAN : Mobile intelligent agents in erlang.

[18] 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.

[19] A. SHIN, J. COWAN, ARTHUR A. GLECKLER et *et al* : Revised^7 report on the algorithmic language scheme, 7 2013.

[20] MICHAEL SPERBER, R. KENT DYBVIG et *et al* : Revised^6 report on the algorithmic language scheme, 9 2007.

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