UNIVERSITÉ DU QUÉBEC À MONTRÉAL

L'INTERFACE NATIVE DE NIT, UN LANGAGE DE PROGRAMMATION À OBJETS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ALEXIS LAFERRIÈRE

MARS 2012

REMERCIEMENTS

Remerciements particuliers à mon directeur de recherche, Jean Privat, professeur au département d'informatique de l'UQAM, pour tout son aide, sa patience, le partage de sa vision et la réalisation du langage Nit. Je remercie également mes collègues du groupe de recherche, le GRÉSIL, avec qui j'ai eu la chance de collaborer ; Jean-Sébastien Gélinas, Alexandre Terrasa, Julien Chevalier, ainsi qu'Étienne Gagnon, directeur du groupe de recherche GRÉSIL et professeur au département d'informatique de l'UQAM.

Je me dois de remercier ma famille et amis qui m'ont supporté au cours de la réalisation de cette étude ; Nilo, Shelsy, Arabelle et surtout Christiane ainsi que Sandrine pour leur aide. Sans oublier François et Jodi qui ont malheureusement tout tenté pour m'aider, je vous en suis redevable.

Merci au mot *langage* pour son apparition à 620 reprises dans ce document, spécialement pour sa complicité avec ma coordination pour qu'il soit toujours écrit sous la forme de *lagnage*.

TABLE DES MATIÈRES

LISTE DES FIGURES

LEXIQUE

Ce document utilise différents termes avec une signification pouvant être ambiguë dans le domaine de recherche. Ci-suit la définition de certains termes tels qu'ils sont utilisés dans ce document.

API Une interface de programmation d'applications, ou *application programming interface* en anglais, est un ensemble de règles ainsi que d'une description des paramètres d'appels et de retour pour interagir avec un logiciel ou une bibliothèque de fonctions.

Analyse statique Une analyse statique est normalement réalisée à la compilation du logiciel en analysant le code source pour en tirer des connaissances. Ces connaissances peuvent servir à des optimisations statiques.

Appel à super Nous utilisons l'expression *appel à super* pour représenter l'exécution explicite des implémentations d'une même méthode dans les classes et les modules parents. En Nit [Privat, 2011], une méthode peut être redéfinie par spécialisation ou par raffinement de classes ; lors d'un appel à super, l'implémentation du langage gère alors l'exécution proprement dite des différentes implémentations de la méthode.

Bibliothèque native Dans ce document, nous considérons qu'une bibliothèque native est une bibliothèque de fonctions partagées. Celles-ci prennent généralement la forme de fichiers *.dll* en Windows ou *.so* en Linux. Elles sont chargées dynamiquement à la demande du logiciel ou au lancement du logiciel. Le programmeur peut en invoquer des fonctions natives. Ces fonctions sont compilées pour l'architecture de la machine.

Classes natives Une classe native est une grande catégorie de classes qui associent un type C à une classe Nit. Il s'agit d'un concept que nous avons développé au cours de cette étude. Une telle classe permet que ses instances soient automatiquement traduites par l'interface native et elle permet de typer statiquement des références vers des structures C dans le code Nit. Ce concept est développé en détail au chapitre 5.

Code d'échafaudage Le code d'échafaudage est produit par le compilateur du langage de haut niveau ou par les outils de l'interface native. Il se différencie du code généré par le fait qu'il sert de référence ou de base de travail au programmeur. Il est créé dans le but d'être étendu ou modifié. Il ne sert généralement que d'aide au programmeur sans que son emploi soit vital à la réalisation du logiciel.

Code de base Le code de base, ou *glue code* en anglais, est le code répétitif nécessaire à l'utilisation de l'interface native. Ce code sert uniquement à interagir avec l'interface native sans réaliser de réelle s d'affaires. La génération de ce code peut être automatisée par un logiciel, par exemple le logiciel SWIG [Beazley et al., 1996].

Code généré Le code généré est celui produit directement par le compilateur du langage de haut niveau ou par les outils utilisés. Par exemple, nous considérons les fichiers objet *.o* générés par GCC comme étant de cette catégorie. Celui-ci ne devrait pas être modifié par le programmeur et il ne sert qu'à être compilé à son tour.

Flot d'appels Le flot d'appels d'un logiciel est une connaissance sur le comportement du logiciel représentant les différents chemins d'exécution possibles du logiciel. Le compilateur peut analyser le code source du logiciel et déterminer le flot d'appels possible du logiciel depuis le ou les points d'entrée. Cette connaissance permet différentes optimisations statiques sur le logiciel, telles que le retrait de code inutilisé.

Greffons Les greffons, ou *plugins* en anglais, sont un système permettant de charger dynamiquement des fonctionnalités dans un logiciel. Ces petits logiciels peuvent être réalisés par un développeur tiers.

IDE Un environnement de développement intégré, ou *integrated developpment environement* en anglais, est un logiciel pour programmeur offrant des services facilitant le développement et la maintenance de logiciels.

Interface de fonctions étrangères Une interface de fonctions étrangères est un mécanisme interconnectant deux langages de programmation à l'intérieur d'un même logiciel. Elle permet l'appel de fonction depuis un langage vers l'autre. Elle gère aussi à différents niveaux la conversion des données entre les langages. Ce mécanisme peut prendre différentes formes dont des ajouts à la grammaire des langages, un API ou un module standard. Il offre généralement différents outils pour faciliter son emploi et réaliser les tâches complexes reliées à son utilisation.

Interface native Une interface native est une sorte d'interface de fonctions étrangères qui interconnecte un langage de haut niveau et un langage natif. Ses fonctionnalités diffèrent de celles d'une interface de fonctions étrangères, car elles sont limitées à l'interaction avec les langages natifs.

Langage de haut niveau Un langage de haut niveau est expressif, proche du mode de pensée de l'humain et éloigné des détails de représentations de la machine. Il offre généralement plusieurs services tels qu'un ramasse-miettes et assure une certaine sûreté d'exécution du logiciel. Il permet également l'utilisation de paradigmes de programmation avancés, tels que le paradigme à objets ou la programmation fonctionnelle.

Nous considérons que Nit [Privat, 2011], Java [Gosling et al., 2005], Python [v. Rossum, 2009b], Ruby [Thomas, Fowler et Hunt, 2004], C# [ECMA, 2006] et Eiffel [Meyer, 1992] sont des langages de programmation de haut niveau.

Dans ce document, ce terme est principalement employé relativement au langage natif utilisé par une interface native. Les caractéristiques du langage de haut niveau sont donc généralement évaluées en fonction du langage natif.

Pour l'interface native de Nit, le langage de haut niveau est le langage Nit.

Langage natif Un langage natif est un langage près de la machine, offrant peu de services avancés au programmeur. Par exemple, les langages C et Assembleur n'offrent pas de ramasse-miettes, ni de systèmes de gestion des exceptions au niveau du langage. Il permet toutefois un contrôle plus précis sur la machine et sur l'exécution du logiciel.

Dans ce document, ce terme est principalement employé relativement au langage de haut niveau utilisé par une interface native. Les caractéristiques du langage natif sont donc généralement évaluées en fonction du langage de haut niveau.

Nous utilisons également l'expression code natif pour représenter du code écrit en langage natif.

Pour l'interface native de Nit, le langage natif est le langage C.

Méthode native Une méthode native est une méthode du langage de haut niveau implémentée en langage natif. Dans ce document, nous y référons également dans un but de précision sous le terme de méthode implémentée nativement. Nous considérons que les constructeurs implémentés nativement sont de cette catégorie.

Module hybride Un module hybride est un module implémenté dans plus d'un langage. Il peut également s'agir d'un module dont certaines parties, dont les méthodes, n'ont de sens que pour le langage de haut niveau, mais sont implémentées en langage natif. Nous avons développé le concept des modules hybrides au cours de cette étude et nous le présentons en détail au chapitre 1.

Optimisation statique Une optimisation statique est normalement réalisée à la compilation du logiciel pour en améliorer la performance à l'exécution. Cette optimisation se base sur les informations rendues disponibles par l'analyse statique, telles que le flot d'appels du logiciel.

POSIX Le standard POSIX, acronyme de *portable operation system interface for Unix*, est une suite de standards qui assurent une compatibilité entre les différents systèmes d'exploitation Unix. Ces standards sont au moins partiellement suivis par d'autres systèmes d'exploitation ou suites logicielles.

Spécialisation Le terme spécialisation est un terme utilisé avec le paradigme à objets pour représenter le concept de catégorisation des classes. Il est donc utilisé pour qualifier lorsqu'une classe est sous catégorie (ou sous-classe) d'une autre. Ce concept est souvent abusivement nommé héritage, mais dans ce document le terme spécialisation y sera préféré. [Privat et Ducournau, 2005]

On utilise alors les expressions suivantes : la classe A spécialise la classe B et la classe B généralise la classe A.

Types nullables Les types nullables sont un concept de certains langages de programmation à objets, tels que Nit [Privat, 2011] et C# [ECMA, 2006], où une distinction est faite entre un type acceptant la valeur nulle et ceux ne contenant que des valeurs significatives. [Gélinas, Gagnon et Privat, 2009]

Types numériques ou d'énumération Le langage Nit distingue les classes qui représentent une valeur immuable, tel qu'un nombre, des autres grands types de classes. Ces classes n'ont pas d'état et ne comportent donc pas de champs, mais peuvent avoir des méthodes.

Receveur implicite Dans le paradigme de programmation à objet, un appel de méthode consiste en un envoi de message destiné à un receveur. Toute méthode comporte alors un receveur implicite qui est accessible par le mot clé `self` en Nit.

RÉSUMÉ

L'interface native permet à un logiciel de profiter des avantages des langages natifs ainsi que de ceux du langage de haut niveau. Elle intervient entre les différents langages pour permettre les appels de méthodes et la conversion des données. Son utilisation amène cependant généralement une perte de sûreté à l'exécution du logiciel et son emploi est souvent complexe.

Dans le cadre de cette recherche, nous développons l'interface native du langage de programmation à objets Nit. Notre recherche vise à résoudre au mieux les problèmes soulevés par l'utilisation d'une interface native, et ce, par une analyse rigoureuse des différents détails de conception d'une interface. Notre intention est donc de concevoir, selon des objectifs précis, l'interface native idéale pour le langage Nit. Pour mettre à l'épreuve notre proposition, nous avons conçu et implémenté l'interface native du compilateur Nit.

La conception de cette interface native s'appuie donc sur des objectifs que nous considérons garants d'une interface native de qualité. Ces objectifs consistent à préserver la sûreté d'exécution du logiciel, maintenir une connaissance du flot d'appels, utiliser le langage Nit de façon expressive et selon ses forces, conserver une syntaxe naturelle en C ainsi qu'offrir une interface native versatile et d'utilisation rapide par tout autre moyen.

Pour atteindre ces objectifs, nous proposons quatre grandes approches clés ; la forme des modules hybrides pour gérer la coexistence de deux langages ; une déclaration explicite des appels de méthodes réalisées depuis le langage C pour conserver la connaissance du flot d'appels ; une représentation symétrique des types et méthodes Nit en C pour en permettre une utilisation naturelle et vérifiée statiquement ; les classes natives qui représentent les types C en Nit et leur appliquent les forces du paradigme de programmation à objets, dont le polymorphisme.

Enfin, pour valider l'interface native proposée et implémentée, nous présentons comment nous avons utilisé cette interface pour réaliser des modules et des logiciels Nit. Nous démontrons également que cette interface peut être utilisée dans le développement d'autres interfaces spécialisées en fonction de besoins spécifiques.

Mots clés : interface native, interface de fonctions étrangères, compilation, langages de programmation à objets

INTRODUCTION

Les langages de programmation de haut niveau, tel que Java [Gosling et al., 2005], Python [v. Rossum, 2009b], Ruby [Carlson et Richardson, 2006], C# [ECMA, 2006] et beaucoup d'autres, sont loin du fonctionnement de la machine physique, simple d'utilisation et offrent aux programmeurs plusieurs services tels qu'un paradigme de programmation avancé, une syntaxe expressive, une gestion automatique de la mémoire ou encore, une assurance de sûreté d'exécution. Le langage de programmation à objets Nit [Privat, 2011] est de cette catégorie, il utilise une syntaxe simple, comporte un ramasse-miettes et offre des optimisations statiques à la compilation. En comparaison, le langage C [ans, 1989] conserve de forts avantages, il permet la programmation près de la machine et l'accès à une grande quantité de code préexistant.

Nous avons développé l'interface native du langage Nit qui permet d'intégrer du code C à un logiciel Nit, de façon à invoquer des fonctions C depuis Nit et manipuler les objets Nit depuis le code C. Ceci permet aux programmeurs de profiter du meilleur des deux langages dans un même logiciel.

Beaucoup de langages de programmation de haut niveau offrent une telle interface native, sa forme est adaptée aux langages qu'elle relie. Cette forme limite les stratégies possibles d'intégration des deux langages. Ainsi selon les langages, technologies et contextes d'utilisation, les interfaces natives et leur emploi varient grandement.

Dans le cadre de cette recherche, nous avons étudié les interfaces natives des langages de programmation à objets dans le but de concevoir l'interface native du langage Nit. Pour ce faire, nous avons établi des objectifs précis pour l'interface native de Nit et avons développé des solutions pour atteindre ces objectifs. Nous avons également implémenté l'interface native de Nit dans le compilateur principal du langage.

0.1 Langage de programmation Nit

Nous avons réalisé cette étude dans le cadre du développement du langage de programmation Nit [Privat, 2011]. Dans cette section, nous présentons les particularités du langage Nit principalement en rapport avec les autres langages de programmation à objets. Ces particularités sont utiles pour comprendre les décisions que nous avons prises au cours de l'étude et nous leur faisons référence à répétition dans ce document.

Le langage Nit applique le paradigme de programmation à objets avec typage statique et spécialisation multiple [Ducournau et Privat, 2010]. La syntaxe du langage est voulue pour être simple d'utilisation comme un langage de script. Le code source d'un logiciel en Nit est découpé en modules, des fichiers avec une extension `.nit`. Chaque module peut dépendre (ou importer) d'autres modules et il regroupe les classes servant à l'implémentation d'une préoccupation précise.

Une différence de l'utilisation des modules en Nit est que la définition de chaque classe n'est pas restreinte à un seul module. En fait, le langage Nit permet le raffinement de classes [Privat, 2006], c'est-à-dire la possibilité de rouvrir une classe définie préalablement dans différents modules. Ceci permet de modifier les méthodes, les attributs et les super classes d'une classe existante. Par exemple, dans un module utilisateur, il est possible de raffiner la classe Int qui est définie dans la bibliothèque standard pour y ajouter une nouvelle méthode *fibonacci*. Tout raffinement de classe modifie le comportement de toutes ses instances au travers du logiciel. Cette technique amène une utilisation différente des modules où chaque module ne sert pas uniquement de contenant à des classes, mais représente une fonctionnalité précise. L'interface native du langage doit supporter le raffinement de classe, car cette fonctionnalité est à la base du langage.

Le langage Nit utilise une politique de typage covariant. [Ducournau, 2002] C'est à dire que pour deux types Object et String où String est une sous classe de Object, le type paramétré Array[String] est considéré un sous-type de Array[Object]. Une variable typée statiquement en tant que Array[Object] peut donc contenir une instance de Array[String]. Le typage covariant amène une expressivité plus naturelle et évite des cas d'utilisation

inévitable de conversions explicites. Il nécessite de vérifier à l'exécution certains cas qui ne peuvent pas être détectés dès la compilation. Par exemple, selon les types précédents, assigner à une variable de type Array[Object] une instance de Array[String], pour ensuite lui ajouter une instance de Object est considéré normal à la compilation. Par contre, à l'exécution, la variable étant dynamiquement typé par Array[String], l'ajout d'une instance de Object soulève une erreur, car le type dynamique n'accepte que les instances de String.

Le langage Nit offre également les types virtuels. Ces types virtuels sont définis dans une classe parmi ses propriétés, de façon analogue aux méthodes et aux attributs, et ils peuvent être redéfinis dans les sous classes. Ils sont utilisés comme tout autre type à l'intérieur de la classe. Ces types sont covariants, d'une façon semblable aux paramètres des classes génériques.

En Nit, lorsqu'un type peut contenir une valeur nulle, il doit être explicitement qualifié de nullable [Gélinas, Gagnon et Privat, 2009]. Le programmeur doit vérifier la valeur contenue par une variable typée nullable comme n'étant pas nulle avant d'utiliser la variable comme non nullable. Pour réaliser cette vérification, le programmeur utilise une structure conditionnelle normale semblable à `if obj != null then ...`. Une analyse du flot d'exécution du logiciel est utilisée pour inférer le type des variables selon le flot du code et ainsi, après la vérification, traiter la variable comme n'étant pas nullable.

En plus des constructeurs anonymes (les constructeurs normaux des autres langages de programmation à objets), le langage Nit permet la définition de constructeurs nommés. De plus, un constructeur automatique est offert pour toute classe sans constructeur explicite. Ce constructeur automatique est anonyme et paramétré de façon à remplir tous les attributs de la classe.

Le langage Nit utilise le changement brusque de contexte pour contrôler le flot d'exécution. Le changement brusque de contexte permet de forcer la sortie d'une boucle avec break et de forcer le passage au prochain tour de boucle avec continue. Il permet également de préciser une étiquette vers où effectuer un saut. Ces étiquettes sont associées à la fin des structures de contrôle.

Le langage Nit permet l'utilisation de fermetures pour qu'une méthode appelée puisse exécuter du code précisé lors de l'appel. Le code d'une fermeture a même accès à son environnement, donc aux variables locales du site d'appel de la méthode. Le changement brusque de contexte est également possible au travers de fermetures et permet de sortir d'une fonction depuis une fermeture. Cette dernière fonctionnalité est originale à Nit et entraîne des difficultés nouvelles au niveau de l'interface native.

Au moment de l'écriture, le langage n'offre pas de système d'exception. Toutefois, certains des cas d'utilisation d'un tel système se règlent à l'aide du changement brusque de contexte et des fermetures. Le langage et son implémentation principale ne supportent pas l'exécution parallèle de code Nit.

0.2 Contexte de l'étude

Le langage Nit est conçu pour être indépendant de son implémentation, toutefois au moment de l'écriture de ce document, une seule implémentation existe. Il s'agit d'un compilateur traduisant le code Nit en code C, pour ensuite compiler le code C de façon standard. Notre expérimentation a été limitée à cette implémentation, nous avons toutefois veillé à ce que l'interface native soit compatible avec d'autres implémentations du langage, comme un interpréteur futur.

Précédemment à l'apport de cette étude, l'interface native de Nit était limitée, non standardisée et surtout improvisée. Elle permettait toutefois de répondre aux besoins vitaux d'un langage de programmation, tel que pour faire appel à des fonctions système. Dans le cadre de cette étude, nous avons remplacé l'interface native préexistante par une nouvelle. Dans ce document, lorsque nous discutons de l'interface native de Nit, il s'agit de la nouvelle interface native que nous avons réalisée pour Nit.

Cette étude est adaptée aux particularités du langage Nit, cependant nous tentons de conserver un point de vue ouvert aux autres langages de programmation à objets. Il demeure que certaines sections s'avèrent principalement pertinentes au langage Nit.

0.3 Portée de l'étude

Cette étude traite le sujet de l'interface native du langage Nit et non une interface de fonctions étrangères. Cette dernière implique un système plus général permettant l'interface avec différents langages alors que les interfaces natives interconnectent un langage de haut niveau avec un langage natif. Plusieurs des sujets étudiés sont applicables aux deux types d'interfaces, mais cette étude se concentre sur l'interface native.

De plus, cette recherche touche aux différents aspects d'une interface native, mais se concentre principalement sur les cas pertinents au langage Nit. Nous nous référons au paradigme de programmation à objets tels qu'il prend forme dans le langage Nit, il en est de même pour tout autre paradigme considéré dans cette étude.

Considérant que le langage Nit est encore en développement, certains aspects n'y sont pas encore définis. Il est prévu d'y ajouter un système de gestion des exceptions, une forme de programmation parallèle ainsi que de compléter le système des énumérations. Dans cette étude, nous étudions ces sujets de façon superficielle et générale, sans chercher à déterminer la solution idéale, ni pouvoir l'implémenter.

Cette étude adopte le point de vue de l'interface native qui est utilisée dans un logiciel dont le point d'entrée est en Nit. Le point de vue inverse, l'utilisation dans un logiciel principal en C, est brièvement discuté sans être approfondi. Cette approche du sujet de l'interface native dépend grandement du format des bibliothèques compilées du langage Nit, concept qui n'a pas encore une forme définitive.

0.4 Rôle et importance des interfaces natives

Les interfaces natives sont inévitablement utilisées lorsqu'un logiciel est réalisé dans plus d'un langage. Même lorsque cette interaction est camouflée par l'utilisation de bibliothèques ou de composants. Les logiciels modernes réalisés dans le langage Nit utiliseront l'interface native pour réaliser les appels système ou l'affichage graphique.

Dans cette étude, nous déterminons certains cas d'utilisation communs pour une interface native. Ceux-ci guident la conception de l'interface native de Nit en établissant le contexte du programmeur lors de son interaction avec l'interface. Ci-suivent les cas relevés.

Appel à une fonction système ou d'une bibliothèque native Dans ce cas, le programmeur utilise l'interface native de Nit pour faire appel à une fonction système ou une fonction provenant d'une bibliothèque native. Ceci est fréquemment dans le but d'utiliser du code spécialisé et éprouvé, tel qu'une fonction de chiffrement. Cette utilisation survient principalement pour répondre à un besoin spontané ne demandant l'appel que d'une fonction, et ce, à un seul endroit à l'intérieur d'un même logiciel.

L'interface native permet alors de réaliser le lien vers la fonction de la bibliothèque et elle assure la conversion des types primitifs.

Enveloppe de bibliothèque de fonctions De façon semblable au cas précédent, mais dans le but d'une utilisation plus vaste et fréquente d'une bibliothèque, le programmeur crée une enveloppe d'une bibliothèque de fonctions natives. Cette enveloppe prend la forme d'un module Nit dont l'emploi est identique aux autres modules. L'enveloppe peut être facilement réutilisée par différents logiciels et le programmeur faisant appel à une telle enveloppe ne décèle aucun indice du code C qu'elle recouvre. Une enveloppe vise normalement à couvrir une grande partie ou toutes les fonctionnalités d'une ou plusieurs bibliothèques natives.

Cette utilisation est fréquente, par exemple le module Python [v. Rossum, 2009b] *pygame*[1] enveloppe les bibliothèques natives de graphisme SDL [Hall, 2001] et OpenGL [Woo et al., 1999]. Un programmeur utilisant ce module ne manipule que du code Python sans voir le code C utilisé par le module et les bibliothèques sous-jacentes. Le module *pygame* est implémenté à l'aide de l'interface native de Python. [v. Rossum, 2009d]

Dans ce cas, l'interface native est utilisée pour réaliser les liens vers les fonctions de la bibliothèque native et dans certains cas, pour réaliser en entier la structure objet de l'enveloppe.

---

1. Pour plus d'informations sur le projet *pygame*, voir `http://pygame.org/wiki/about`.

Optimisation de méthode Le langage Nit ne permet pas un contrôle aussi précis sur la machine que le langage C, le programmeur peut alors tirer profit d'implémenter une méthode en C dans un logiciel en Nit. Ceci peut être dans le but d'optimiser une méthode dont la performance est critique ou pour passer des commandes précises directement au processeur.

Par exemple, lors de la réalisation d'un module Nit, nous avons implémenté l'algorithme de chiffrement MD5 [Rivest, 1992] en C pour qu'il soit plus performant. Cet algorithme applique des opérations mathématiques complexes qui bénéficient à être programmées plus près de la machine.

L'interface native doit être employée dans ce cas, mais la technique à utiliser, et l'aisance d'utilisation varie selon la forme de l'interface.

Système de greffons L'interface native peut être utilisée pour implémenter un système de greffons dans un logiciel. Il est même commun de choisir un langage tel que Python pour implémenter les greffons d'un logiciel écrit dans un autre langage statique. Les langages de script permettent une réalisation rapide d'un greffon et leur manque de sûreté d'exécution peut être partiellement contrôlé par une isolation de leur environnement d'exécution.

Par exemple, le logiciel de lecture vidéo, Totem [2], utilise cette approche. Le logiciel principal est écrit en C [Kernighan, Ritchie et Ejeklint, 1988] et les greffons sont réalisés en C, Python [v. Rossum, 2009b] ou encore en Vala [Madiath, 2010].

Dans ce cas, les greffons doivent pouvoir être chargés et déchargés dynamiquement. Ce qui peut être géré dans le langage de haut niveau autant qu'en C.

Preuve de concept et prototypage Des langages rapides d'utilisation peuvent être employés pour développer une preuve de concept ou une fonctionnalité temporaire. Dans ce cas, la sûreté d'exécution et l'architecture du logiciel ne sont pas des aspects privilégiés. L'interface native permet de relier le logiciel en développement avec des fragments de code qui sont utilisés uniquement comme preuve de concept.

Cette utilisation nécessite une moins grande sûreté d'exécution. Une interface native plus rapide d'utilisation peut alors être privilégiée.

---

2. Pour plus d'informations sur le projet Totem, voir `http://live.gnome.org/Totem`.

Une bonne interface doit être suffisamment polyvalente pour permettre une utilisation rapide dans tous ces cas. C'est-à-dire que son utilisation ne doit pas alourdir inutilement le développement d'un logiciel Nit. Mais de façon pratique, elle doit surtout faciliter les cas les plus communs en limitant minimalement son expressivité pour les autres situations.

De ces différentes possibilités, le cas d'enveloppe de bibliothèque est l'un des plus importants et il est fréquemment employé par les utilisateurs avancés. Pour leur part, les cas d'optimisation de méthode et d'appel à une fonction d'une bibliothèque sont importants pour un langage jeune, lequel n'offre pas tous les services attendus par le programmeur.

0.5 Problème étudié et objectifs adoptés

Le problème général étudié est celui de l'interface native du langage de programmation à objets Nit. La conception même d'une interface qui réalise son but premier d'interconnecter Nit et C soulève de grands défis. Chaque détail de la spécification de l'interface doit être adapté aux deux langages et aux besoins d'utilisation de l'interface.

Cette étude divise en cinq objectifs les problèmes amenés par les interfaces natives : assurer la sûreté à l'exécution du logiciel en conservant le typage sûr, maintenir la connaissance du flot d'appels malgré des rappels depuis le code C, utiliser le langage Nit de façon expressive, conserver une syntaxe naturelle en C et, finalement, réaliser par tout autre moyen une interface native versatile d'utilisation efficace.

0.5.1 Sûreté à l'exécution du logiciel

La compilation statique du langage Nit garantit une certaine sûreté à l'exécution du logiciel. Cette assurance est perdue dès l'utilisation des interfaces natives classiques. [Tan et al., 2006; Lee et al., 2010; Klinkoff et al., 2006] Pour maintenir cette sûreté à l'exécution du logiciel, plusieurs aspects protégés dans le langage Nit doivent aussi être contrôlés dans le code C.

Alors que le langage C permet une manipulation libre de la mémoire du logiciel, il est possible de restreindre le programmeur à des manipulations vérifiables. Par exemple, pour prévenir la corruption de la mémoire du logiciel, tout accès mémoriel doit être contrôlé. Ceci prévient les accès hors bornes aux tableaux ainsi que la corruption involontaire des données. La cohérence des objets Nit peut être assurée par une utilisation efficace des types C et par un typage statique des appels de méthodes de haut niveau. Ceci permet de détecter dès la compilation les erreurs de manipulation pouvant nuire au bon fonctionnement du logiciel.

Le langage Nit offre un typage sûr permettant une vérification statique de l'utilisation des types à la compilation. Précédemment, cette vérification se limite au code Nit, les interfaces natives offrent rarement une vérification statique des types en C. [Lee et al., 2010; Tan et al., 2006; Furr et Foster, 2005] Ceci amène une perte de sûreté d'exécution dès que l'interface native est utilisée par un logiciel. Dans cette étude, nous favorisons toutes façons de maintenir ce typage sûr.

0.5.2 Maintenir la connaissance du flot d'appels

L'analyse statique du langage Nit permet d'obtenir des connaissances supplémentaires sur le logiciel, dont le graphe d'appels de méthodes. Ce dernier est une connaissance de tous les chemins d'exécution possible du logiciel. Cette connaissance permet d'optimiser le programme à la compilation, tel qu'en retirant le code mort et en imbriquant des appels de fonctions lorsque possible. [Privat, 2002] Le langage C ne peut pas être analysé aussi efficacement, cette connaissance doit être maintenue autrement lors de l'utilisation d'une interface native. Dans cette étude, nous favorisons toutes solutions permettant de préserver cette connaissance du flot d'appels et ainsi préserver toutes les optimisations malgré l'utilisation de l'interface native.

0.5.3 Utiliser le langage Nit de façon expressive

Une des caractéristiques du langage Nit est son expressivité et sa syntaxe intuitive. Nous désirons préserver au mieux cette forme expressive et intuitive dans les ajouts nécessaires au langage pour donner accès aux fonctionnalités de l'interface native.

Le langage Nit applique le paradigme de programmation à objets et alors que celui-ci est étranger au langage C. Pour cette raison, ce paradigme sera traité autant que possible en Nit et le moins visible possible en C.

0.5.4 Syntaxe naturelle en C

L'utilisation d'une syntaxe naturelle en C permet que l'interface native soit d'utilisation intuitive tout en abaissant le risque d'erreurs humaines. Une forme intuitive permet un apprentissage plus rapide de l'interface et en général une utilisation plus agréable. De plus, une syntaxe naturelle en C est aisément compréhensible par un programmeur expérimenté en C, et elle permet d'éviter les erreurs qui peuvent survenir lors de l'utilisation de code inutilement complexe.

Il est commun que les interfaces natives utilisent une syntaxe inhabituelle au langage C pour représenter les entités du langage de haut niveau, telles que les méthodes et les objets. Il en est le cas pour Java [Gosling et al., 2005] et Python [v. Rossum, 2009b] qui utilisent des types et fonctions dynamiques dans le langage C. Comme leur emploi est moins naturel pour le programmeur, ils nuisent donc à la rapidité d'apprentissage et amènent plus d'erreurs humaines. En comparaison, nous visons à représenter ces propriétés de la façon la plus naturelle possible en C.

La représentation des services provenant du langage Nit dans le code C peut également adopter des formes qui y sont plus ou moins naturelles. Ces services, tels que la gestion de la mémoire, peuvent également dupliquer ceux qui sont naturels au langage C. Par exemple, un ramasse-miettes comparativement à l'allocation manuelle de la mémoire. Dans cette étude, nous avons favorisé d'offrir ces services sous des formes naturelles au langage C.

L'une des principales motivations de l'interface native est de profiter des forces du langage C, par exemple, le langage C permet l'optimisation de méthodes et l'utilisation de code existant. L'interface doit alors s'assurer de préserver au mieux les possibilités amenées par le langage C, et ce, en le restreignant minimalement. De plus, comme le langage C a des faiblesses par rapport au langage Nit (par exemple, le langage C n'applique pas le paradigme de programmation à objets) nous évitons de manipuler, en C, les concepts qui ne sont pas naturels au langage C.

0.5.5 Interface native versatile et efficace à l'utilisation

Le langage Nit se veut être intuitif et simple de programmation, pour cette raison nous conservons ce principe comme priorité dans le développement de l'interface native de Nit. De plus, nous privilégions l'économie de temps de travail du programmeur même si c'est au coût de plus de travail par le compilateur. La réutilisation de code et la multitude d'architectures demandent une grande portabilité du code écrit et des logiciels générés. Dans cette optique, l'interface native doit être versatile et efficace à l'utilisation. L'interface native est versatile si elle peut être utilisée dans tous les cas d'utilisation et si elle favorise la réutilisation de code. Elle est efficace à l'utilisation si elle ne nécessite pas de travail superflu de la part du programmeur et qu'elle peut être utilisée rapidement dans les cas simples.

Alors que tous les objectifs présentés précédemment visent indirectement la conception d'une interface versatile et efficace à l'utilisation, il est possible d'améliorer ces points de façon plus directe. Une façon de faire consiste à offrir des fonctionnalités et des outils pour éviter le travail répétitif au programmeur. Nous ne nous limitons pas par le temps de travail à la compilation s'il est d'une complexité raisonnable et aide à améliorer l'interface native.

L'utilisation du langage C amène généralement une perte de portabilité entre les différentes plateformes. Ce problème est plus important pour les langages visant à être compatibles pour toute plateforme, tel que le langage Java [Gosling et al., 2005; Liang, 1999]. Le code C doit alors être compilé pour chaque plateforme. Il s'agit d'un

problème inhérent à toutes les interfaces natives, mais l'utilisation de bons outils permet d'en réduire l'importance.

Un processus complexe d'utilisation de l'interface native accapare beaucoup de temps du programmeur pour des tâches qui sont répétitives. Par exemple, l'utilisation de l'interface native du langage Java [Gosling et al., 2005; Liang, 1999] demande du programmeur qu'il implémente le code Java, le compile, implémente le code C, le compile à son tour pour ensuite l'utiliser comme toute autre classe Java. Ceci peut décourager le programmeur à utiliser l'interface ou encore ralentir l'adoption d'un nouveau langage. D'autant qu'une démarche complexe amène les utilisateurs à commettre plus facilement des erreurs de programmation [Lee et al., 2010; Furr et Foster, 2005], ce qui rend le développement d'un logiciel plus coûteux et moins sûr. Pour ces raisons, nous privilégions le développement de fonctionnalités et d'outils permettant de simplifier le travail du programmeur.

Une interface native très bien adaptée à certains cas d'utilisation est généralement trop spécialisée pour permettre un usage versatile ; elle s'avère alors lourde dans les autres situations. Par exemple, l'interface native de C# [ECMA, 2006] est très efficace pour associer directement une méthode à une fonction d'une bibliothèque native, mais elle permet difficilement l'optimisation de méthodes. Ou encore, des interfaces amenées par le milieu scientifique [Klinkoff et al., 2006; Yee et al., 2009] assurent que le fragment natif du logiciel ne réalise pas d'opérations ou d'accès mémoire non autorisés, mais cela au coût d'une perte significative de performance. Nous jugeons alors que l'interface native de base d'un langage doit être versatile et, au besoin, utilisée pour réaliser des interfaces spécialisées.

Certaines interfaces et certains langages sont dotés d'une spécification incomplète. Par exemple, le langage Ruby [Thomas, Fowler et Hunt, 2004] n'a pas de spécification officielle et utilise seulement l'implémentation d'origine comme référence. Les détails du comportement de l'interface, surtout en cas d'erreurs, sont alors indéfinis. Ceci fait en sorte que leur comportement varie entre les différentes implémentations d'un même langage ou interface. [Lee et al., 2010] Nous considérons qu'une interface native versatile doit avoir une spécification précise associée au langage et qui est indépendante de son implémentation.

0.5.6 Garantie de qualité de l'interface native

Les cinq objectifs présentés dans les sections précédentes sont à la base de toutes les décisions que nous avons prises au cours de cette étude. Ils définissent ce que nous considérons être une garantie de la qualité d'une interface native. Nous nous y référons donc constamment dans ce document.

0.6 Interface native du langage Nit

Pour atteindre les objectifs établis, nous avons développé une interface native pour le langage Nit qui implémente quatre principales techniques. Ces techniques consistent en la forme des modules hybrides, la déclaration explicite des méthodes appelées depuis le langage natif, la symétrie des types et des méthodes ainsi que les classes natives. Nous présentons brièvement ces techniques dans cette section pour les développer tout au long du document.

Nous avons basé l'interface native de Nit sur la forme de modules hybrides, cette forme sépare en plusieurs fichiers chacun des modules qui utilisent l'interface native. Chaque langage est alors utilisé dans un fichier séparé. Ceci permet de conserver la pureté des différents langages, soit de préserver l'application du paradigme objet dans le langage Nit et d'utiliser une syntaxe native naturelle dans chaque langage.

Le programmeur a la responsabilité de déclarer les méthodes appelées depuis le code C, ce qui préserve la connaissance du flot d'appels de méthodes. Ceci permet d'utiliser cette information supplémentaire pour optimiser la compilation et générer une documentation d'aide au programmeur.

L'application d'une symétrie des types et des méthodes permet de conserver une utilisation naturelle des types Nit dans le langage C. De plus, elle préserve la sûreté à l'exécution par une vérification statique des types à la compilation, telle qu'une compilation normale de code C.

L'utilisation des classes natives permet d'encapsuler les types C dans le langage Nit et ainsi de leur donner une représentation naturelle. Elle permet également d'appliquer d'autres caractéristiques du paradigme de programmation à objets à ces types C, dont la spécialisation de classes et l'utilisation de constructeurs.

0.7 Structure du document

À partir de ce point dans le document, nous abordons le sujet principal, un aspect à la fois.

Le premier chapitre introduit le concept des modules hybrides ainsi que les autres formes d'interface native que nous avons considérées. De plus, il présente les méthodes natives ainsi que les différents outils supportant le travail du programmeur dans l'utilisation de l'interface native.

Le deuxième chapitre décrit une technique pour maintenir la connaissance du flot d'appels malgré l'utilisation de l'interface native. Nous comparons la forme adoptée avec ses alternatives, examinons ce qui est à déclarer pour préserver cette connaissance, proposons une syntaxe pour la déclaration et en tirons les avantages supplémentaires pour le programmeur.

Le troisième chapitre discute de la représentation symétrique des types dans les langages Nit et C. On y présente également les restrictions dans la composition des identifiants C, ainsi que les particularités de certains types. Ces particularités concernent les types primitifs, les types nullables et les types génériques. Nous y traitons également de la durée de vie des types symétriques en C, des implications du ramasse-miettes Nit dans le code C et des détails d'implémentation des types symétriques dans le compilateur Nit.

Le quatrième chapitre présente la symétrie des méthodes Nit en C. Il y est discuté des fermetures, de la sûreté d'exécution amenée par la symétrie et des collisions entre les noms générés. On y présente aussi les détails d'implémentation des méthodes symétriques dans le compilateur Nit, la gestion des exceptions depuis le langage C et la réalisation d'une bibliothèque native depuis un module Nit.

Le cinquième chapitre concerne les classes natives, leur principale raison d'être, les façons de préciser leur équivalent en C, les constructeurs d'une classe native, leur destruction, leur grande catégorie et les possibilités de spécialisation de ces classes.

Le sixième chapitre présente l'aspect pratique de l'interface native de Nit et sa validation. Il y est discuté de la procédure d'utilisation de l'interface, des interfaces alternatives que nous avons élaborées et des modules réalisés à l'aide de ces interfaces.

Le septième chapitre traite des recherches scientifiques connexes à notre sujet de recherche. Celles-ci sont divisées selon quatre catégories ; les interfaces de langage imbriquées, les analyses dynamiques, les analyses statiques et la surveillance du comportement du logiciel.

Le tout est conclu par un rappel de la recherche, une révision de l'interface native de Nit ainsi que par des suggestions de recherches futures dans le domaine.

CHAPITRE I

MODULES HYBRIDES

Une interface native peut prendre différentes formes, chacune étant mieux adaptée à un langage ou à certains cas d'utilisation. Dans ce chapitre, nous considérons les différentes formes possibles d'interface native pour le langage Nit. Pour ce faire, nous analysons les formes du module hybride, de l'association directe à une fonction native, du module conçu en langage C, du chargement dynamique et des langages imbriqués. Nous revenons ensuite à la forme des modules hybrides pour présenter la façon d'y déclarer une méthode implémentée nativement et approfondissons les outils disponibles.

1.1 Module hybride fragmenté

Une forme possible de l'interface native est de réaliser un module composé de plus d'un fichier, chaque fichier servant à un langage différent. Par exemple, un fichier pour le langage Nit et un pour le langage C. Nous appelons cette approche le module hybride, car un même module est composé de plus d'un langage et de plusieurs fichiers. Cette approche permet de conserver l'utilisation des paradigmes de programmation avancés dans le langage Nit tout en profitant des forces du langage C. Son utilisation implique de définir les méthodes natives, des méthodes Nit implémentées en C, dans le fichier Nit et de les implémenter séparément dans le fichier C.

L'interface native officielle du langage Java [Gosling et al., 2005; Liang, 1999], la Java Native Interface (JNI), prend une forme semblable. Elle nécessite tout de même l'action de compiler le module en une bibliothèque native adaptée pour Java, ce qui se

```
1 class FibonacciEngine {
2     public native int Fibonacci( int i );
3     static {
4         System.loadLibrary("LibFib");
5     }
6 }
```

```
1 JNIEXPORT jint JNICALL Java_FibonacciEngine_Fibonacci(
2     JNIEnv *env, jobject jobj, jint n ) {
3 }
```

Figure 1.1: Exemple de module hybride en Java.
Ce module hybride en Java réalise l'implémentation d'une méthode Java en C. Le premier fragment de code contient la déclaration d'une méthode Java, nommée Fibonacci. Celle-ci est implémentée nativement dans la bibliothèque *LibFib*, dont le chargement est précisé par un appel à System.loadLibrary. Le second fragment de code présente la signature que prend l'implémentation de cette méthode dans la bibliothèque native.

fait séparément de la compilation du code Java. Dans cette interface, les fonctions implémentées en C sont déclarées de façon semblable aux autres fonctions, mais ne comportent pas de corps. La bibliothèque native est identifiée par un appel à une fonction de chargement dynamique de bibliothèque.

Avec la JNI, une méthode native est déclarée dans le langage Java et implémentée dans le langage C. La signature en C est dérivée de celle en Java, elle est composée de macros et de types de la JNI qui servent à assurer une compatibilité avec les différentes machines virtuelles Java. Un exemple de code Java utilisant la JNI est présenté dans la figure 1.1.

Pour le langage Nit nous préférons profiter de la compilation intermédiaire en langage C pour compiler autant le code Nit que le code natif. Le code d'implémentation de méthodes natives écrites par le programmeur sera compilé au même moment que le code C généré par le compilateur Nit. Les fichiers sources C associés à un fichier Nit

sont retrouvés facilement s'ils répondent à une convention de nom. Par exemple, le fichier C mon_module.nit.c pour accompagner le fichier Nit mon_module.nit.

Cependant, au niveau de l'efficacité, cette forme est lourde d'emploi pour le cas d'utilisation d'appel à une fonction d'une bibliothèque native. Le programmeur doit déclarer une méthode dans le langage Nit et l'implémenter dans le langage C, même si l'implémentation ne fait que réaliser un appel à une fonction d'une bibliothèque. Ce qui implique que la signature de la fonction doit être écrite deux fois, en deux langages différents. Ceci a tout de même l'avantage de mettre à la disposition du programmeur toutes les fonctionnalités du langage C pour bien gérer les détails de l'appel.

Enfin, nous avons constaté que cette forme s'avère pratique dans les cas d'optimisation de méthodes existantes ou d'extension d'un programme existant. En fait, elle implique l'ajout des fichiers sources pour le langage C, mais permet de n'y implémenter que les fonctions désirées. Par exemple, pour optimiser une méthode existante dans un module, le programmeur redéfinit la méthode comme étant native, ajoute un fichier C associé au module et y implémente la méthode native.

1.2 Association directe avec une fonction d'une bibliothèque native

Une forme simple d'interface native permet de préciser, depuis le langage Nit, une association directe entre une méthode Nit et une fonction d'une bibliothèque native existante. Un appel à la méthode Nit invoque la fonction C de la bibliothèque qui lui est associée. Lors d'un appel simple d'une fonction native préexistante, l'interface native est utilisée entièrement dans le langage Nit, le programmeur n'a pas à écrire en langage C.

Les informations nécessaires pour gérer les détails de l'appel de fonction doivent être déclarées dans le langage Nit. Ceci peut être fait autant à la déclaration de la fonction externe que lors de son appel. Ces informations permettent au programmeur d'adapter les détails d'appel selon les besoins. Ces détails servent à accéder à une fonction avec un nom non standard, à indiquer le nom de la bibliothèque native, à spécifier la convention d'appel de la fonction et la représentation des données.

```
1 package fib
2
3 // #cgo LDFLAGS: -L../lib/ -lfib
4 // #include "../lib/libfib.h"
5 import "C"
6
7 func main() {
8     var r = C.Fibonacci( 12 )
9 }
```

Figure 1.2: Exemple d'association directe à une fonction native en Go.
Le module fib précise les arguments à passer au compilateur avant d'importer le module C. Par la suite, la fonction native Fibonacci, définie dans l'entête *libfib.h*, est invoquée par un appel à C.Fibonacci.

Pour les langages avec introspection qui permettent de définir dynamiquement les attributs d'un objet, il est possible de camoufler ces détails dans les espaces de noms, de l'identifiant de la fonction ou des paramètres de la fonction ; ce qui permet de conserver une syntaxe naturelle au langage de haut niveau. Le langage Go [The Go Programming Language Specification, 2011; Command cgo, 2011] utilise une telle forme d'interface, le programmeur précise les options à passer au compilateur C en commentaires, puis atteint la fonction via un module spécial nommé C. Un exemple d'utilisation est présenté dans la figure 1.2.

Dans le cas des langages compilés avec une introspection limitée, tels que Nit, la syntaxe nécessaire pour définir les informations supplémentaires est moins naturelle au langage. Il est souvent nécessaire de préciser des détails du langage natif parmi le langage de haut niveau. Nous avons observé cette façon de faire dans les interfaces natives des langages C# [ECMA, 2006] et Eiffel [Meyer, 1992; Meyer, 1987].

En C# [ECMA, 2006], une fonction native est associée à une fonction statique, les détails de lien et d'appel sont précisés avec des attributs de fonction, mais la fonction est appelée normalement.

```
1  public class FibonacciEngine
2  {
3      [DllImport ("libfib.so", EntryPoint = "Fibonacci"),
4          CharSet=CharSet.Unicode]
5      private static extern Int32 CFibonacci( Int32 n );
6
7      static public void Main () {
8          int r = CFibonacci( 12 );
9      }
10 }
```

Figure 1.3: Exemple d'association directe à une fonction native en C#.
La classe FibonacciEngine définit une fonction CFibonacci qui est implémentée nativement. Celle-ci est associée à la fonction Fibonacci de la bibliothèque partagée *libfib.so* et les chaînes de caractères sont converties en encodage Unicode pour le langage natif. La signature de la fonction C# utilise des types primitifs avec un équivalent précis dans le langage natif.

Dans le langage Eiffel [Meyer, 1992], la déclaration de la méthode native se fait de façon semblable aux autres méthodes, mais la signature native équivalente est précisée avec les types C normaux. Il est également possible d'y préciser le nom de la bibliothèque partagée au lieu du fichier d'en-tête. Le langage profite de sa compilation statique, se basant sur les compilateurs ANSI C, pour lier directement le logiciel compilé avec la fonction native référée. Un exemple de son utilisation est présenté à la figure 1.4.

Cette forme d'interface associant directement une méthode du langage Nit à une fonction native est peu appropriée pour optimiser une fonction ou étendre un logiciel existant. Dans ces cas, le programmeur doit implémenter et compiler une bibliothèque native indépendante, laquelle n'a aucune connaissance du logiciel principal.

1.3 Module réalisé en langage natif

Une forme possible est de concevoir un module Nit entièrement en langage C. Cette forme est particulièrement adaptée aux langages interprétés ou dynamiques. Dans ces

```
1  class
2      FIBONACCI_ENGINE
3  feature
4      c_fibonacci (i: INTEGER): INTEGER
5              -- Fonction de Fibonacci implementee en C
6          external
7              "C (int): int | %"fib.h%""
8          alias
9              "Fibonacci"
10         end
11 end
```

Figure 1.4: Exemple d'association directe à une fonction native en Eiffel.
La classe FIBONACCI_ENGINE définit la fonction c_fibonacci qui est implémentée nativement. La signature Eiffel est déclarée normalement, sa signature native et la source de la fonction native sont précisées dans la section external. Dans ce cas, la source est un en-tête standard au langage C, mais peut également être directement une bibliothèque native. La section alias précise le nom de la fonction native telle que présente dans la source.

cas, seuls les modules écrits en langage C sont compilés. Ces modules peuvent ensuite être utilisés comme tout autre module. Cette forme est utilisée par les langages Python [v. Rossum, 2009d; v. Rossum, 2009a] et Ruby [Thomas, Fowler et Hunt, 2004].

Pour réaliser un module Python natif simple, le programmeur doit implémenter les fonctions natives avec des types compatibles au langage Python et implémenter la fonction d'initialisation du module. Cette dernière sert à définir la liste des fonctions du module. Le module natif est compilé à l'aide d'un script Python qui se charge d'invoquer le compilateur C avec les bons arguments. Un exemple de son utilisation est présenté à la figure 1.5.

Cette forme permet de conserver la pureté du langage de haut niveau, mais ramène les paradigmes avancés dans le code de bas niveau, telle que la programmation à objets. Ceci amène donc une utilisation inhabituelle et lourde du langage C avec des concepts qui n'y sont pas naturels.

Cette approche s'avère lourde d'emploi dans le cas d'optimisation de méthodes et d'enveloppes de bibliothèques natives. Dans ces cas, un module entier doit être réalisé, même pour implémenter une seule fonction. Ceci nécessite beaucoup de code et de travail de compilation, même pour les cas simples.

1.4 Chargement dynamique de bibliothèques natives

La majorité des systèmes d'opération modernes offrent des fonctions systèmes pour réaliser un chargement dynamique de bibliothèques natives et appeler ses fonctions. Une interface native dynamique peut être développée autour de ce système et ainsi permettre un accès facile aux fonctions natives depuis le langage Nit, et ce, sans nécessiter l'écriture de code C.

Dans ce cas, le code du langage Nit réalisant le chargement de la bibliothèque est exécuté dynamiquement. Les données manipulées, telles que les références aux bibliothèques et aux fonctions natives, prennent alors la forme d'objets standards du langage.

```
1  static PyObject * Fibonacci(PyObject *self, PyObject *args) {
2      /* ... */
3      /*  construit une valeur Python depuis les donnees natives */
4      return Py_BuildValue("i", 122);
5  }
6
7  /* Liste des fonctions du module */
8  static PyMethodDef modfib_methods[] = {
9      {"Fibonacci", Fibonacci, METH_VARARGS, "Nombre de Fibonacci"},
10     {NULL, NULL, 0, NULL},    /* Sentinelle */
11 };
12
13 /* Fonction d`initialisation du module */
14 PyMODINIT_FUNC initmodfib(void) {
15     Py_InitModule("modfib", modfib_methods);
16 };
```

Figure 1.5: Exemple de définition d'un module Python en C.

Le module Python nommé modfib est réalisé entièrement en C. Il y définit une seule fonction Fibonacci qui est implémentée en C. Celle-ci construit la valeur de retour à l'aide de la fonction PyBuildValue de l'API de Python, cette fonction attend le format des arguments et leur liste. La liste des méthodes du module est représentée par un tableau de PyMethodDef et associé au module lors de son instanciation par un appel à Py_InitModule.

```
1 import ctypes
2 libfib = ctypes.LoadLibrary( "libfib.so.1" ) # charge la bibliotheque
3 result = libfib.Fibonacci( 12 ) # appel une fonction
```

Figure 1.6: Exemple de chargement dynamique de bibliothèque native en Python.
Ce module Python importe le module de chargement dynamique, charge la bibliothèque par un appel à ctypes.LoadLibrary et en invoque une fonction par ce qui prend la forme d'un appel normal de fonction. L'appel de la fonction Fibonacci sur la bibliothèque chargée dynamiquement réalise l'invocation de la fonction native du même nom dans la bibliothèque native.

Cette forme d'objet appelable existe déjà dans certains langages dynamiques tel que Python [v. Rossum, 2009b]. Dans ces langages, il est normal de manipuler une fonction en tant qu'objet et de définir dynamiquement les méthodes d'un objet.

Il existe des bibliothèques ou modules permettant le chargement dynamique de bibliothèques natives pour beaucoup de langages. Dans certains cas, elles font même partie de la bibliothèque standard du langage. C'est le cas pour le module *ctypes* de Python [v. Rossum, 2009c] et le module *dl* de Ruby [Thomas, Fowler et Hunt, 2004].

En Python, le module *ctypes* [v. Rossum, 2009c] gère la conversion des données entre les deux langages et trouve les fonctions dynamiquement selon le nom de la fonction appelée sur la bibliothèque. Une bibliothèque externe est alors traitée comme un objet Python et chacune de ses fonctions est appelée comme une méthode normale sur l'objet. Un appel direct à la méthode getattr() permet l'accès aux fonctions natives de noms illégaux selon la syntaxe Python. Un exemple de son utilisation est présenté à la figure 1.6.

En Ruby [Thomas, Fowler et Hunt, 2004], le module *dl* charge dynamiquement les bibliothèques natives et en extrait les fonctions selon leur nom et signature. Dans ce cas, autant la bibliothèque que ses fonctions sont traitées comme des objets Ruby. L'opérateur [] permet de réaliser l'appel aux fonctions natives d'une façon qui invoque un appel normal. Un exemple de son utilisation est présenté à la figure 1.7.

Pour le langage Nit, qui applique un typage statique, l'utilisation de fonctions natives en tant qu'objets n'est pas naturelle, contrairement au langage Python qui utilise des objets

```
1  require 'dl'
2
3  libfib = DL::dlopen( 'libfib.so.1' ) # charge la bibliotheque
4
5  # trouver la fonction voulue avec sa signature, dans ce cas
6  # elle prend un entier en argument et retourne un entier
7  fun_fib = libfib[ 'Fibonacci', 'II' ]
8
9  print fun_fib.call( 12 )[0] # appelle la fonction
10                             # et affiche son retour
```

Figure 1.7: Exemple de chargement dynamique de bibliothèque native en Ruby.
Ce module Ruby utilise le module *dl*, de chargement dynamique, pour charger la bibliothèque native *libfib.so.1* à l'aide de la fonction DL::dlopen. Ensuite, la fonction native Fibonacci est retrouvée depuis la bibliothèque par un appel à [] en précisant la signature de la fonction. La signature est représentée par 'II', signifiant qu'elle prend un entier en paramètre et retourne un entier. La fonction native est invoquée par un appel à la fonction call sur la fonction chargée dynamiquement et retourne alors deux valeurs : le retour de la fonction native et un code d'erreur, en tel cas.

appelables. Il est alors nécessaire d'appeler une méthode sur l'objet, laquelle invoque la fonction native représentée. De plus, une méthode Nit normale est définie avec une signature statique, ce qui permet de vérifier la validité des types utilisés lors des appels. Par contre, il est impossible de vérifier statiquement le type des données utilisées lors de l'appel à une fonction native chargée dynamiquement.

Les fonctions système utilisées par ce type d'interface sont standards et accessibles normalement depuis le langage C, cette interface peut donc être développée à l'aide d'une autre interface statique.

Cette forme d'interface permet de bien répondre au cas d'utilisation de greffons, de prototypage et d'appel à une fonction d'une bibliothèque native. Son utilisation, dans ce dernier cas, n'est pas aussi fiable qu'avec l'interface native statique ; il n'y a pas de vérifications des types à la compilation. Techniquement, cette interface est moins performante puisqu'un temps de travail est nécessaire pour réaliser chaque appel dynamique. Alors qu'une interface statique, telle que celle que nous avons développée, prépare la conversion de type à la compilation du logiciel, l'interface à chargement dynamique détermine la conversion à appliquer à l'exécution. Donc cette interface ne devrait être utilisée que pour les greffons et le prototypage. Pour les autres cas, où un chargement dynamique de bibliothèques est nécessaire, ou même dans le cas des greffons, il est possible de se servir de toute autre forme d'interface et d'utiliser les fonctions système de chargement dynamique.

1.5 Langages imbriqués

Une forme versatile d'interface est d'imbriquer les différents langages dans un même fichier source. Une variante simple de cette forme permet l'implémentation native d'une méthode du langage Nit à la suite de sa déclaration en langage C. Des variantes plus avancées permettent d'imbriquer les deux langages à répétition.

Selon les implémentations, le langage C offre cette forme d'interface pour y imbriquer du code assembleur [Pappas et Murray, 1995; H., 2002], de même pour le langage C++ [Stroustrup, 1997] qui l'utilise pour intégrer le langage C. Elle est également

```
1  int addition( int a, int b ) {
2      int somme;
3
4      asm volatile (
5          "nop\n\t"
6          "nop\n\t"
7          "addl %2, %0"
8              : "=r" (somme)
9              : "0" (a), "r" (b)
10     );
11
12     return somme;
13 }
```

Figure 1.8: Exemple de langages imbriqués en C.
Ce segment de code C (pour l'implémentation du langage par GCC) utilise bloc de code assembleur à l'intérieur de la fonction addition. Ce bloc commence par des nop qui ne servent qu'à démontrer comment séparer les différentes commandes. De façon utile, le bloc exécute une opération d'addition sur deux variables du langage C et assigne la valeur obtenue à la variable somme.

utilisée par un ajout au langage Ruby [Thomas, Fowler et Hunt, 2004] qui permet d'y imbriquer du code C [Ruby Inline Readme, 2008]. Dans la littérature scientifique, deux interfaces proposées pour le langage Java [Gosling et al., 2005], Jeannie [Hirzel et Grimm, 2007] et Janet [Bubak, Kurzyniec et Luszczek, 2000], adoptent cette forme.

Selon les implémentations du langage C [ans, 1989], le code assembleur est contenu entièrement dans un bloc spécialisé, au milieu du code C. Cela a l'avantage de permettre la manipulation des données C depuis le code assembleur, à l'aide des noms de variables tel que déclarés en C. Un exemple de son utilisation est présenté à la figure 1.8.

Le langage C++ [Stroustrup, 1997] a été conçu pour être relativement compatible avec le langage C. Son interface native avec C est donc à la base du langage. Elle utilise

```
1 extern "C" {
2     void f1();
3 }
```

Figure 1.9: Exemple de langages imbriqués en C++.
Ce segment de code C++ démontre comment est utilisé le mot clé extern pour définir un bloc de code comme utilisant les conventions d'appel de fonction C.
Dans ce cas la fonction f1 est traitée comme une fonction C alors que le programme principal est en C++.

le mot clé extern pour identifier les fonctions ou les blocs de code avec un format de lien différent, dont celui du langage C. Un exemple de son utilisation est présenté à la figure 1.9.

Deux interfaces imbriquées ont été développées pour le langage Java. Elles prennent la forme d'un langage dont la grammaire est une composition de celles de Java et de C. L'interface Janet [Bubak, Kurzyniec et Luszczek, 2000] permet l'implémentation de méthodes en code natif, suite à sa déclaration. Alors que l'interface Jeannie [Hirzel et Grimm, 2007] imbrique les deux langages, l'un à l'intérieur de l'autre, que ce soit pour un appel, l'accès à une variable ou pour un bloc de code entier. Un exemple de son utilisation est présenté dans la figure 1.10.

Cette forme est simple d'utilisation pour le cas d'appel d'une fonction dans une bibliothèque native. Elle permet l'accès à toutes les fonctionnalités du langage C pour réaliser les appels natifs. Elle est également adaptée à l'optimisation d'une méthode du langage Nit, permettant d'implémenter la méthode en C dans le même fichier. Selon les fonctionnalités offertes, elle peut permettre de n'implémenter en C que la section critique d'une méthode en glissant un bloc de code natif à l'intérieur une méthode en Nit ou encore, donner accès aux variables locales et d'instances.

À l'implémentation, cette forme soulève différents défis techniques pour assurer un support complet des langages imbriqués et de leurs différentes implémentations. Pour cette raison, elle est plus aisément appliquée par un outil qui assurera la séparation des langages avant l'appel des compilateurs respectifs. Nous avons d'ailleurs réalisé un tel outil comme preuve de concept, il est présenté dans le chapitre 6.

```
1  \.C {
2      int c_fibonacci(int n) {
3          if (n == 0)
4              return 0;
5          if (n == 1)
6              return 1;
7          else
8              return c_fibonacci(n-1) + c_fibonacci(n-2);
9      }
10 }
11
12 class FibonacciEngine {
13     public native int Fibonacci(int n) {
14         return `c_fibonacci(`n);
15     }
16 }
```

Figure 1.10: Exemple de langages imbriqués avec Jeannie.

L'interface native Jeannie est utilisée à l'intérieur d'un fichier source Java et réalise le changement de contexte entre les deux langages par l'utilisation de l'accent grave (`).

Dans cet exemple, un bloc de code C est définit en début du fichier Java. Ce bloc définit la fonction native c_fibonacci. La classe FibonacciEngine définit la méthode Java Fibonacci qui utilise le changement de contexte pour invoquer la fonction native c_fibonacci et y passer en argument la variable Java n.

Dans cet exemple simple, alors que la base est en Java, il y a trois changements de contexte : le bloc C de la ligne 1 à 10, l'appel à c_fibonacci à la ligne 14 et le retour depuis cet appel pour obtenir la valeur de la variable Java n.

1.6 Modules hybrides en Nit

Selon les objectifs de cette étude et en considération des caractéristiques du langage Nit, nous avons retenu la forme du module hybride. Cette forme permet de séparer efficacement le code des deux langages selon leurs spécialités et ainsi de préserver une utilisation intuitive de chaque langage. Elle amène une utilisation versatile qui permet de répondre à tous les cas.

Nous avons sérieusement considéré la forme des langages imbriqués comme forme principale. Toutefois, des limites techniques au niveau du générateur de grammaire du compilateur Nit compliquaient grandement l'implémentation de cette forme. De plus, après expérimentation nous jugeons que la forme du module hybride est à favoriser comme forme de base, la forme imbriquée peut être implémentée à l'aide d'un outil.

L'interface du langage Nit utilise trois fichiers distincts pour les principaux fragments de code d'un module hybride. Un fichier contient le code Nit, un autre est l'en-tête du code C et le dernier est le corps du code C. Nous utilisons une convention de noms de fichiers pour aisément identifier les fichiers d'un même module. Ils commencent par le nom du module et se terminent respectivement par l'extension *.nit*, *.nit.h* et *.nit.c*. Nous considérons que le fragment en langage Nit se compose uniquement du fichier Nit alors que le fragment C, ou natif, se compose des deux fichiers C.

La forme du module hybride, lorsque couplée avec un outil approprié, permet d'éviter au programmeur de définir les signatures des méthodes plus d'une fois. Cet outil produit les fichiers d'échafaudage pour écrire le code C depuis le fichier Nit. Ceci retire alors l'un des principaux désavantages à l'utilisation de cette approche. Cet outil est discuté dans ce chapitre, à la section 1.8.

La versatilité d'utilisation des modules hybrides permet de s'en servir pour offrir les autres formes d'interfaces natives. Il est possible de simuler l'association directe à une fonction d'une bibliothèque native, de construire un module de chargement dynamique ainsi qu'un outil pour gérer l'utilisation des langages imbriqués.

Pour simuler l'association directe aux fonctions d'une bibliothèque, il est possible de préciser le nom en C d'une méthode Nit implémentée nativement et ainsi, à la compilation, celle-ci est retrouvée directement parmi les bibliothèques liées. Toutefois, la façon propre est d'utiliser les fichiers d'en-tête C du module hybride pour rediriger les appels à la bibliothèque externe, évitant ainsi d'écrire du code superflu et répétitif. De plus, il est alors toujours possible d'y réaliser des opérations supplémentaires et de gérer les détails d'appels dans les fichiers C.

Au cours de cette étude, nous avons combiné le concept des modules hybrides avec un outil pour réaliser une autre interface native avec la forme des langages imbriqués. Ce faisant, la forme des modules hybrides agit toujours comme base et l'outil ne fait que réaliser un traitement simple. Cela ouvre toutefois la possibilité de se servir de cette forme d'interface spécialisée dans les cas où elle est pratique, telle que l'optimisation de méthodes. Cet outil est discuté à la section 6.1.2 du chapitre de validation de l'interface native.

L'interface native basée sur des modules hybrides peut également servir à réaliser un module du langage Nit offrant les services de chargement dynamique de bibliothèques natives. Ce module est alors utilisé comme tout autre module et amène une autre forme d'interface spécialisée pour des cas tels que le prototypage. Cette interface alternative est discutée en détail à la section 6.1.1 du chapitre de validation de l'interface native.

1.7 Méthodes natives

La principale raison d'être d'une interface native pour Nit est l'accès à du code C depuis le langage Nit. Cet accès prend normalement la forme d'un appel de fonction. Selon la forme du module hybride, tout appel à une fonction purement en langage natif se fait via une méthode native. Ce type de méthode est donc à la base de toute communication au travers de l'interface. On se restreint ici à des méthodes natives et non à des fonctions natives dans le langage Nit, car ce dernier est purement à objets et ne permet pas la définition de fonctions.

Étant donné que l'aspect le plus important de l'interface proposée concerne les méthodes natives, ce sujet est traité tout au long de ce document. Cette section n'est que l'introduction au concept.

La méthode native est déclarée en Nit dans un module hybride. Cette déclaration se compose de la signature seulement et ne comporte aucun corps. Le corps de la méthode est implémenté entièrement en C. La déclaration et l'implémentation sont alors deux fichiers sources distincts, mais associés au même module. La séparation de ces fichiers ainsi que leur produit est présenté à la figure 1.11.

L'implémentation de la méthode native est réalisée dans le fragment C du module. Elle prend la forme d'une fonction C normale ou encore, selon la préférence du programmeur, elle peut consister en une macro ou une redéfinition par préprocesseur dans l'en-tête du code source. Toutefois, la forme normale reste une déclaration de la fonction selon la signature attendue dans l'en-tête C et son implémentation dans le corps C.

Dans l'interface native développée pour le langage Nit, une méthode native est déclarée dans le fichier source Nit à l'aide de l'expression clé `is extern`. Au besoin, pour marquer son identifiant dans le langage natif, celui-ci est précisé dans une chaîne de caractères délimitée par des guillemets. Dans l'implémentation d'une méthode native, il est possible de manipuler des objets du langage de haut niveau, d'appeler des méthodes, de faire des appels à super, etc. Un exemple de déclaration et implémentation d'une méthode native est présenté à la figure 1.12.

L'implémentation normale d'une méthode native est réalisée dans le fragment C du module. L'implémentation se fait dans le corps d'une fonction C servant explicitement à implémenter la méthode native. Le nom de cette fonction peut être spécifié dans la déclaration de la méthode native ou sinon, il est déterminé à partir du nom de la classe et de la méthode en Nit.

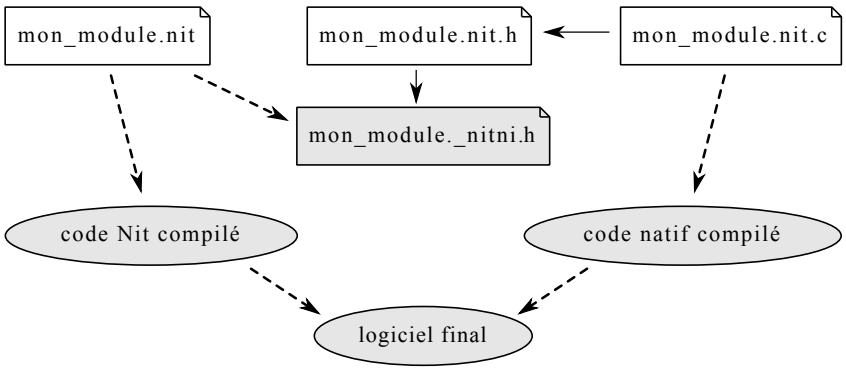

Figure 1.11: Composants d'un module hybride en Nit.

Cette figure représente les différents composants d'un module hybride. Les fichiers en fond blanc sont réalisés par le programmeur. Ceux en fond gris sont générés par le compilateur Nit.

Les flèches pleines représentent des dépendances d'importation entre les fichiers sources en langage C. Les flèches pointillées pointent les fichiers générés depuis les fichiers leur servant de source.

Le fichier *mon_module.nit* est réalisé en Nit, *mon_module.nit.h* consiste en l'en-tête C et *mon_module.nit.c* représente le corps C où les méthodes natives sont implémentées. Le fichier *mon_module._nitni.h* est généré par le compilateur et fournit les fonctions intermédiaires utilisées par le code source natif. Ces fonctions intermédiaires offrent des services au programmeur, nous discutons de ces services dans le reste de ce document.

```
1 module mon_module
2
3 class MaClasse
4     fun foo( i : Int ) : NativeString is extern
5     fun bar( mc : MaClasse ) is extern "c_bar"
6 end
```

```
1 char* impl_MaClasse_foo( int i ) {
2     /* ... */
3 }
4
5 void c_bar( MaClasse mc ) {
6     /* ... */
7 }
```

Figure 1.12: Exemple d'une méthode native dans un module hybride en Nit.

Le premier fragment de code en langage Nit définit une classe MaClasse comportant deux méthodes natives. La première, foo, prend comme argument un entier et retourne une chaîne de caractères dans une forme compatible avec le langage C. La seconde méthode, bar, attend comme argument une instance de MaClasse alors que le nom de son implémentation en C est précisé comme étant c_bar.

Le second fragment de code, celui-ci en C, implémente les deux méthodes déclarées plus tôt. L'implémentation de la méthode foo y est représentée par la fonction impl_MaClasse_foo. La seconde méthode est implémentée par une fonction C au nom précisé dans le langage Nit, c_bar. Les détails de la forme des méthodes et des types passés au travers de l'interface seront discutés dans la suite du document.

1.8 Outils générateurs de code de base

Comparativement aux autres formes considérées, la forme des modules hybrides nécessite une quantité significative de code de base et amène de la duplication d'informations dans plusieurs fichiers. Par exemple, les fichiers sources C doivent référencer les fichiers de l'interface native et la signature des méthodes natives est dupliquée en Nit et en C. De plus, bien que le code C puisse être réalisé manuellement dans son entièreté par un programmeur avancé, il est complexe de se conformer à la spécification et son écriture manuelle est donc vulnérable aux erreurs humaines. Pour ces différentes raisons, il est recommandé de se fier à un outil qui génère le code de base et de réaliser l'implémentation à partir de ce code.

1.8.1 Outil générateur d'échafaudages

Nous avons réalisé un outil, nommé *nits*, pour générer automatiquement le code de base des fichiers C d'un module hybride. Ces fichiers servent de structure à l'implémentation des méthodes natives. Les fichiers créés contiennent l'équivalent C des méthodes natives ainsi que la signature attendue accompagnée d'un corps vide, à implémenter.

Cet outil est utilisé par les autres techniques que nous présentons dans ce document afin de fournir davantage d'informations au programmeur et d'aider à l'implémentation des autres fonctionnalités.

Dans les cas simples comme l'optimisation d'une méthode, suite à l'invocation de l'outil, le programmeur n'a qu'à implémenter en C le corps des fonctions natives dans le fichier corps généré. Pour les cas plus complexes, le programmeur est libre de modifier les deux fichiers C et d'importer d'autres fichiers sources.

1.8.2 Outil générateur d'enveloppes

Un outil spécialisé peut automatiser la majeure partie de la génération de l'enveloppe d'une bibliothèque native et ainsi éviter beaucoup de travail au programmeur. Un tel

outil réalise généralement le travail suite à une analyse statique des en-têtes de la bibliothèque native. Il reste à la responsabilité du programmeur de vérifier le code généré et de corriger les erreurs possibles.

Le danger de ces outils est qu'ils peuvent créer un faux sentiment de sûreté pour le programmeur, car ils génèrent difficilement une enveloppe parfaite et la responsabilité attendue du programmeur, de vérifier le code, est souvent ignorée.

Ce genre d'outils peut être généralisé à plusieurs langages, tel que l'est SWIG [Beazley et al., 1996]. Ce qui permet, après une modification initiale de la bibliothèque native et l'ajout d'informations destinées à l'outil, de générer des enveloppes pour les différents langages, et ce, sans modifications supplémentaires.

Au moment de l'écriture de ce document, aucun outil de ce type n'existe pour le langage Nit. Toutefois, l'adaptation d'un générateur existant faciliterait le développement d'enveloppes pour ainsi offrir l'accès à davantage de code éprouvé aux programmeurs Nit.

1.9 Contribution et atteinte des objectifs

Notre contribution au sujet des formes d'interfaces natives consiste principalement en l'étude de formes possibles, la conception et l'implémentation de trois formes d'interfaces : le module hybride, l'interface imbriquée et l'interface à chargement dynamique. La forme de base du module hybride, la séparation de la déclaration et de l'implémentation, est présente dans au moins un autre langage, Java. Toutefois, le concept du module hybride est nouveau par le fait qu'il associe directement plusieurs fichiers de langages différents dans une unité. Ces fichiers sont considérés comme inséparables et fortement couplés. En comparaison, dans le langage Java, le programmeur doit préciser le nom de la bibliothèque native résultante du code C associé.

Les différentes formes d'interfaces natives que nous avons réalisées pour Nit nous ont aidées à atteindre les objectifs suivants :

Syntaxe naturelle en C La forme du module hybride se base sur une séparation du code Nit et du code C en plusieurs fichiers. Cette séparation aide à préserver une syntaxe naturelle en C.

Utiliser le langage Nit de façon expressive Encore une fois, en séparant les deux langages, la forme du module hybride évite d'alourdir le langage Nit et ainsi favorise son utilisation expressive. Peu de mots clés doivent être ajoutés au langage Nit pour réaliser cette forme.

Interface native versatile et efficace à l'utilisation L'interface native imbriquée, réalisée à l'aide de la forme du module hybride, permet l'atteinte de l'objectif d'offrir une interface native versatile et efficace à l'utilisation. L'implémentation des méthodes natives en C à la suite de leur déclaration permet d'implémenter aussi rapidement une méthode en C qu'en Nit.

CHAPITRE II

MAINTIEN DE LA CONNAISSANCE DU FLOT D'APPELS

Un des objectifs de cette recherche est le maintien de la connaissance du flot d'appels du logiciel et ce, malgré l'utilisation d'une interface native. Le flot d'appels est une forme de connaissance sur le logiciel déterminée par des analyses statiques du code source à la compilation. Cette connaissance représente les chemins d'exécution possibles du logiciel depuis ses points d'entrées. Elle permet donc de prévoir le comportement du logiciel et ainsi le compilateur peut l'utiliser pour optimiser le logiciel. En Nit, cette connaissance permet au compilateur de retirer du code inatteignable et d'éviter les vérifications superflues sur les types nullables. [Privat, 2002; Gélinas, Gagnon et Privat, 2009].

Selon sa forme, l'interface native pourrait ne pas permettre à une analyse statique de savoir quels rappels vers Nit sont possibles depuis le code C. C'est le cas si ces appels sont réalisés dynamiquement, tel qu'en Java [Gosling et al., 2005; Liang, 1999] et en Python [v. Rossum, 2009d] ; dans ce cas, le code C ne peut pas être analysé statiquement pour en extraire un graphe d'appels fiable. Ceci fait en sorte qu'une méthode peut être évoquée depuis le code C sans que cela soit attendu par le système.

Dans ce chapitre, nous considérons différentes approches pour maintenir la connaissance du flot d'appels malgré l'utilisation de l'interface native. Selon la forme de l'interface native de Nit, les méthodes natives sont le seul moyen de passage depuis Nit vers C. Notre étude du problème se concentre sur ces méthodes natives.

2.1 Formes possibles

Nous avons considéré différentes formes générales pour maintenir la connaissance du flot d'appels. Celles-ci consistent en différentes approches pour obtenir l'information voulue.

Considérer toutes les méthodes comme un point d'entrée L'approche la plus simple est de considérer toutes les méthodes Nit comme étant des points d'entrées possibles depuis le code C. Cette approche annule l'utilité du graphe d'appels, car dès qu'une méthode native est invoquée, toutes les autres méthodes sont identifiées comme étant appelables. Comme toutes les méthodes sont considérées comme étant appelables, les possibilités d'optimisation sont réduites.

Cette approche est utilisée par l'interface native de plusieurs langages, tels que celles de Java et de Python. Selon nos objectifs de recherche, cette approche est à éviter, car elle fait en sorte que l'utilisation de l'interface native pour une seule méthode limite les possibilités d'optimisation du logiciel entier.

Déclaration explicite extérieure au langage Le programmeur est responsable de déclarer manuellement les méthodes Nit qui sont appelées depuis le code C. Ceci peut se faire à l'aide de fichiers supplémentaires ou encore directement parmi les arguments du compilateur.

Cette forme d'appel confie toute la responsabilité au programmeur de correctement réaliser la déclaration pour chaque logiciel. Il n'y a aucune vérification qui assure que ces déclarations sont valides.

Analyse du code C Une option est d'appliquer la même technique au langage C que celle utilisée pour le langage Nit soir, d'analyser statiquement le code C et ainsi connaître le graphe du flot d'appels.

Dans le cas où le langage natif n'est le langage C et est analysable, par exemple, pour une implémentation alternative d'un langage tel que Jython [Juneau et al., 2010] et JRuby [Edelson et Liu, 2008] pour lesquels le langage natif est Java, ou IronPython [Hugunin et al., 2004] pour lequel le langage natif est C#, une analyse peut être appliquée. Par contre, cette dernière dépend davantage de l'implémentation

que de la spécification du langage de haut niveau. Ceci s'applique aux interfaces de fonctions étrangères, mais non pas aux interfaces natives. Cette option n'est donc pas viable dans le contexte de notre recherche.

Toutefois, pour l'interface native de Nit le langage natif utilisé est C. Lors d'une utilisation pratique et courante du langage C, le code est difficilement analysable. Le programmeur peut accéder inconditionnellement à toute la mémoire du processus, imbriquer du code assembleur ou encore, faire référence à tout autre langage de haut niveau par son interface native, et ce, en plus d'utiliser différents préprocesseurs, lesquels ajoutent un langage supplémentaire au code source. Le code C est alors trop complexe pour être analysé assez précisément pour obtenir un flot d'appels fiable.

Étendre le langage Nit et déclaration explicite Étendre le langage Nit pour déclarer les informations sur le flot d'appels résout efficacement le problème. Par contre, il faut se fier au programmeur pour déclarer explicitement le flot d'appels depuis chaque méthode native, ce qui a le désavantage de lui confier plus de responsabilités de même que d'exiger une plus grande quantité de travail de sa part, rendant le processus plus susceptible aux erreurs humaines.

Toutefois, cette solution permet une analyse claire du flot d'appels et le tout est connu dès la compilation. Uniquement le code Nit est à analyser, ceci est donc aisément réalisable.

Cette approche peut prendre différentes formes syntaxiques selon la granularité des éléments à déclarer. Ces variantes sont discutées dans ce chapitre, respectivement à la section 2.2 et à la section 2.3.

Analyser un sous-langage de C Pour rendre possible une analyse du code C, ce code C peut être limité à un sous-ensemble analysable du langage C. Dans ce cas, le programmeur n'a accès qu'à une forme analysable de C, mais peut tout de même faire appel à du code C externe qui ne sera pas analysé.

Cette forme est suffisante pour les cas simples, où l'implémentation en C d'une méthode native est restreint à une seule fonction avec des appels directs à Nit ou à du C pur. L'analyse est plus complexe lorsque des instructions de préprocesseur

sont utilisées ou lorsque le code C est réparti dans plusieurs fichiers. Dans ces cas, l'analyse est encore possible, mais nécessite une implémentation plus complexe. Toutefois, cette forme ne permet pas tout ce qui est possible en C, tel que l'appel à des fonctions donc l'adresse est déterminé dynamiquement, ou encore l'ajout de code en assembleur de façon imbriquée.

Nous n'avons pas réalisé d'implémentation de cette forme, mais elle semblerait bien s'intégrer avec une interface native imbriquée. Dans ce cas, seulement le code d'implémentation d'une méthode native serait analysé à la recherche de rappels vers Nit. Les appels de C à Nit non analysables, tels que ceux faits par macro ou indirectement par un appel à une autre fonction, devront être déclarés explicitement.

Pour répondre à l'objectif de maintien de la connaissance du flot d'appels, nous avons retenu l'option d'étendre le langage Nit et de confier au programmeur la déclaration des appels. Les détails de cette forme sont présentés dans les sections suivantes de ce chapitre.

Nous avons considéré que le travail supplémentaire du programmeur en vaut les avantages, même si cela va à l'encontre de l'objectif d'une utilisation simple. Notre implémentation démontre que ceci permet de maintenir la connaissance du flot d'appels et ainsi préserver tous les avantages des analyses statiques du flot.

2.2 Déclaration explicite du flot d'appels

Nous proposons donc que l'interface se base sur la forme où le programmeur est responsable de la déclaration explicite des appels réalisés depuis une méthode native. L'appel de méthode étant un flot directionnel, nous avons évalué les différentes orientations possibles de déclaration des appels. Cette section présente les trois alternatives considérées : marquer une méthode comme étant appelable depuis tout code C, marquer chaque méthode par portée de leur accès et finalement, déclarer explicitement les appels sortants de chaque méthode native.

Librement appelable depuis le code C Une approche facile est de marquer les méthodes appelables depuis le code C. L'implémentation et l'utilisation de cette approche sont simples et ne nécessitent qu'un mot clé identifiant une méthode comme étant appelable à partir de C. Cette approche est également aisément compréhensible par le programmeur, ce mot clé étant utilisé comme toute autre caractéristique d'une méthode.

Une forme semblable est utilisée dans plusieurs langages lors de la création d'une bibliothèque native alors que seulement les fonctions agissant comme points d'entrées sont marquées comme telles. De plus, cette forme est similaire à celle de la déclaration de la visibilité des méthodes.

En ce qui concerne l'analyse du flot, chaque méthode déclarée appelable devra être considérée comme étant parcourue, mais la source exacte des appels ne sera pas connue.

Cette option est intéressante surtout pour les méthodes de base, telles que la manipulation de chaînes de caractères et de types primitifs, ce qui est couramment utilisé avec l'interface native. Par contre, si elle est largement utilisée, elle peut automatiquement marquer beaucoup de méthodes comme étant appelable même si celles-ci ne le sont pas.

Marquer les appels entrants par portée Dans cette forme, le programmeur précise l'accessibilité, depuis le langage C, de toutes méthodes Nit. Ceci peut prendre la forme d'un mot clé dans la déclaration des méthodes Nit, lequel indique si la méthode est potentiellement appelée depuis le code C. Cette forme peut être naturelle au programmeur, car elle est semblable au système de précision de visibilité d'une méthode. D'ailleurs, une implémentation simple serait de se fier à la visibilité de toute méthode pour la considérer comme étant appelable depuis les méthodes natives pour lesquelles elle est visible.

Le principal désavantage de cette forme est que, chaque méthode ayant beaucoup d'appels possibles vers les autres méthodes, elle fait exploser la taille du graphe d'appels, ce qui rend les informations extraites peu utiles.

Déclaration explicite des appels sortants Pour cette approche, le programmeur déclare les méthodes pouvant être appelées depuis l'implémentation native de chaque méthode native, et ce, à la déclaration de la méthode. Cette liste doit prendre une forme analysable par le compilateur, lui permettant de retracer les méthodes visées.

Cette forme a l'avantage de bien compléter l'analyse standard du corps des méthodes qui, elle, y détecterait les appels sortants possibles. De plus, pour le programmeur, il est naturel de déclarer cette liste de méthodes au même moment où il déclare sa signature, où il fait son implémentation et où il en réalise les besoins.

Nous retenons les approches qui consistent à marquer une méthode comme étant librement appelable et à marquer les appels vers Nit. La forme librement appelable est pratique pour les méthodes communes telles que celles sur les types les plus courants, mais son utilisation serait déconseillée au programmeur qui veut écrire du code réutilisable. L'approche avec le flot d'appels sortants a l'avantage de représenter les méthodes appelables ainsi que la source des appels, son utilisation serait donc à favoriser.

Pour l'instant, nous avons implémenté seulement l'option qui consiste à marquer les appels vers Nit. Pour ce faire, le programmeur doit lister les méthodes appelées dans la déclaration de chaque méthode native. Le mot clé `import` est utilisé pour marquer le début de cette liste et chaque élément est séparé d'une virgule. Un exemple d'utilisation est présenté à la figure 2.1.

2.3 Éléments à déclarer explicitement

L'utilisation d'une déclaration explicite des appels sortants d'une méthode native nous amène à considérer ce qui doit être déclaré pour être pris en compte par le graphe du flot d'appels. Dans cette section, nous discutons de la déclaration d'appels de méthodes et de constructeurs, ainsi que des appels à super, des accès aux attributs, des types manipulés et enfin, de l'ordre d'appel.

À noter que cette section introduit les différentes déclarations explicites d'appels et leur particularité syntaxique en Nit. La contrepartie en C de ces appels est discutée dans les chapitres subséquents.

```
1 class MaClasse
2     fun foo is extern import bar, to_s
3     fun bar do end
4 end
```

Figure 2.1: Exemple simple de déclaration explicite d'appel de méthode.
La signature de la méthode foo indique que son implémentation native appelle les méthodes locales bar et to_s. Alors que la méthode bar est définie localement, la méthode to_s provient de la super classe implicite Object.

```
1 class A
2     fun baz : Int do return 4
3     fun foo is abstract
4 end
5
6 class B
7     fun bar is extern import A::baz, Object::to_s
8 end
```

Figure 2.2: Exemple de déclaration explicite d'appel de méthodes avec la forme longue.
La signature de la méthode bar de la classe B, à la ligne 7, utilise la forme longue pour déclarer l'appel de la méthode baz provenant de la classe A ainsi que to_s de la classe Object déclarée dans un autre module.

Méthodes Marquer les méthodes appelées est à la base de cette approche de déclaration explicite des appels depuis le code C. Ceci permet au compilateur de connaître le flot d'appels du logiciel malgré l'appel à des méthodes natives. Le programmeur doit donc déclarer les méthodes appelées à l'aide de leur nom seul. Un exemple d'utilisation est présenté à la figure 2.2.

Il serait possible de préciser également les paramètres utilisés lors des appels de méthodes. Ceci fournit davantage d'informations sur les types utilisés. Par contre, notre étude ne démontre pas le besoin de cette information passée sous cette forme.

```
1  class A
2      init do end
3      init foo do end
4  end
5
6  class B
7      init do end
8      init baz do end
9      fun bar : B is extern import A, B, A::foo, baz
10 end
```

Figure 2.3: Exemple de déclaration explicite d'appel de constructeur

La signature de la méthode bar de la classe B, à la ligne 9, déclare utiliser quatre constructeurs depuis le code C. Il utilise le nom simple de la classe A et B pour représenter les constructeurs anonymes. Il utilise également la forme longue pour déclarer l'utilisation du constructeur foo de la classe A et la forme courte pour le constructeur baz de la même classe.

Constructeurs Les constructeurs doivent également être déclarés comme étant appelables. En Nit, ceux-ci sont déclarés comme les méthodes, à l'exception des constructeurs anonymes qui sont identifiés par le nom de la classe. Un exemple d'utilisation est présenté à la figure 2.3.

Attributs Pour un langage avec des attributs classiques, ceux-ci doivent aussi être déclarés lorsqu'ils sont accédés depuis le code C.

Par contre, dans le langage Nit, les attributs ne peuvent être atteints que par leurs accesseurs. Dans ce cas, les accesseurs sont à déclarer comme méthodes appelées, au même titre que toute autre méthode. Un exemple d'utilisation est présenté à la figure 2.4.

Appel à super Rien n'empêche une méthode native de redéfinir une autre méthode, par spécialisation ou raffinement de classes ; l'interface native de Nit offre cette possibilité. Le programmeur doit tout de même déclarer explicitement l'appel à super pour que celui-ci soit considéré dans la génération du graphe pour le flot d'appels. Un exemple d'utilisation est présenté à la figure 2.5.

```
1 class A
2     var x : Int
3     fun bar is extern import x, x=
4 end
```

Figure 2.4: Exemple de déclaration explicite d'appel aux accesseurs d'un attribut.
La signature de la méthode native bar déclare faire appel à la méthode d'accès et celle d'assignation de l'attribut x. La forme x= est un nom de méthode normal en Nit, dans ce cas, il représente la méthode qui assigne la valeur à l'attribut x.

```
1 class A
2     fun foo : Int do return 4
3 end
4
5 class B
6 super A
7     redef fun foo is extern import super
8 end
```

Figure 2.5: Exemple de déclaration explicite d'appel à super.
La classe B spécialise la classe A et en redéfinit la méthode foo. Celle-ci qui était une méthode normale dans la classe est redéfinie en tant que méthode native dans la classe B. Cette dernière déclare faire un appel à super depuis son implémentation native, à l'aide du mot clé super qui est utilisé comme le serait un nom de méthode locale. Le mot clé super est déjà réservé dans le langage Nit, il n'y a donc pas de conflits de nom possibles avec une méthode.

Types Les types manipulés depuis le langage C pourraient être déclarés explicitement par le programmeur. Par contre, une analyse des méthodes et des autres services explicitement mentionnés s'avère suffisante pour détecter les types utilisés. Cette analyse simplifie le travail du programmeur et lui évite des erreurs et omissions humaines.

Ordre d'appel L'analyse d'une méthode permet de connaître l'ordre dans lequel les appels sortants sont invoqués. Il serait possible de préserver cette information malgré l'utilisation de l'interface en spécifiant que chaque appel explicitement déclaré est dans l'ordre d'utilisation.

Dans certains cas, cette approche nécessiterait la déclaration multiple d'un même appel de méthodes et les boucles d'exécution seraient difficilement représentables. Entrer ces informations complexifie la tâche du programmeur, ces informations n'apportent pas d'avantages clairs à l'analyse du flot d'appels.

De tous ces éléments, nous retenons la déclaration d'appels de méthodes et de constructeurs, ainsi que les appels à super comme les trois formes distinctes utiles. L'accès aux attributs, dans le cas du langage Nit, prend la forme d'un appel de méthodes ; la déclaration explicite des types utilisés et de l'ordre d'appels n'est pas nécessaire pour le graphe d'appels.

2.4 Documentation générée automatiquement

La déclaration explicite des méthodes appelées depuis le langage C amène un autre avantage au programmeur sous la forme de documentation supplémentaire. L'outil générateur d'échafaudages profite de l'information offerte par ces déclarations explicites d'appels pour ajouter de la documentation aux fichiers sources générés. Cette documentation prend la forme de commentaires et sert ainsi de support au programmeur. Ces commentaires exposent la signature des méthodes que le programmeur a précédemment déclarées comme appelables. Ces signatures étant clairement affichées avant toutes sections pouvant les utiliser, le programmeur n'a donc pas à les deviner.

```
1  /*
2  C implementation of mon_module::A::foo
3
4  Imported methods signatures:
5    String new_String_from_cstring( char * str ) for
6        string::String::from_cstring
7    char * String_to_cstring( String recv ) for
8        string::String::to_cstring
9    void A_bar( A recv ) for mon_module::A::bar
10 */
11 String A_foo___impl( A recv, String str ) {
12 }
```

Figure 2.6: Exemple de code d'aide généré par le générateur d'échafaudage de Nit.

Ce code C a été généré par l'outil pour l'implémentation de la méthode foo de la class A. Le programmeur a déclaré que cette méthode appellerait potentiellement le constructeur String::from_cstring ainsi que les méthodes String:: to_cstring et A::bar.

Le code généré laisse le corps de la fonction d'implémentation vide pour que le programmeur la remplisse.

Un exemple de code généré par l'outil générateur d'échafaudages est présenté à la figure 2.6.

Cet avantage supplémentaire facilite le travail du programmeur, ce qui permet de réduire le temps de travail et la possibilité d'erreurs. Cette utilisation est donc très bénéfique à la réalisation de logiciels de qualité, car il est moins susceptible d'y glisser des erreurs humaines.

2.5 Contributions et atteinte des objectifs

Dans le cadre de cette étude, nous apportons quelques contributions au sujet de la préservation du flot d'appels de méthodes lors de l'utilisation d'une interface native. Nous avons réalisé une analyse partielle du problème et des solutions, et surtout, nous avons développé la forme de déclaration explicite du flot d'appels depuis les méthodes natives. Cette dernière est appliquée pour la première fois directement dans un langage de haut niveau et dans son interface native.

La déclaration explicite du flot d'appels répond à différents objectifs de cette étude :

Maintien de la connaissance du flot d'appels du logiciel La déclaration explicite du flot d'appels des méthodes natives répond directement à l'objection du maintien de la connaissance du flot d'appels du logiciel. Alors que le code C n'est pas analysable de façon fiable, les informations entrées par le programmeur sont analysées par le compilateur pour déterminer le flot d'appels des méthodes natives. Ceci permet de connaître le flot d'appels de toutes les méthodes du logiciel.

Interface native versatile et efficace à l'utilisation L'ajout de travail nécessaire pour préciser manuellement le flot d'appels des méthodes natives est contrebalancé par la documentation d'aide générée à l'aide de cette connaissance supplémentaire. Ceci permet de suivre l'objectif de réaliser une interface efficace à l'utilisation.

Sûreté à l'exécution Nous utilisons la connaissance du flot d'appel au niveau du code C pour vérifier statiquement les rappels au code Nit depuis le code C. Ceci améliore la sûreté à l'exécution en détectant les erreurs de nom et de type dès la compilation. Nous développons cette technique au chapitre 4.

CHAPITRE III

TYPES SYMÉTRIQUES

La forme des modules hybrides dans l'interface native de Nit divise la représentation d'un même module entre le langage Nit et C. On l'observe d'abord par la signature de l'implémentation native des méthodes natives puis, par les paramètres qu'elle utilise. Tous les types Nit doivent alors avoir une représentation significative dans le langage C.

Dans ce chapitre, nous discutons des formes que peut prendre cette représentation dans le langage C, nous introduisons la forme des types symétriques et la composition d'un identifiant en C. De plus, nous observons les détails d'utilisation de certains types complexes : les types primitifs, la conversion explicite de types, les types nullables et les types génériques. Finalement, nous discutons de la durée de vie des types symétriques en C, de l'utilisation du ramasse-miettes en C et de l'implémentation réalisée des types symétriques dans le compilateur Nit.

3.1 Transition des objets au travers de l'interface native

Les langages de programmation à objets utilisent les objets pour encapsuler toutes les données. Pour une utilisation pratique de l'interface native, ces données doivent être représentables dans le langage C.

Deux principales formes ont été étudiées pour représenter les objets Nit dans le langage C : les types dynamiques généraux et les types statiques générés par le compilateur.

```
1 PyObject* stackModule = PyImport_ImportModule( "stack_module" );
2
3 /* appel au constructeur de la class Stack */
4 PyObject* stack = PyObject_CallMethod( stackModule, "Stack", "" );
```

Figure 3.1: Exemple d'un type C dynamique et général de Python.
Dans ce segment de code C, le type PyObject∗ sert à représenter autant un module, qu'un objet agissant de pile. Le module et le constructeur sont trouvés via l'API de Python, par leur nom passé sous forme de chaîne de caractère.

3.1.1 Type dynamique général

Dans cette forme, un seul type C représente tous les types Nit. Ce même type C est utilisé pour paramétrer toutes les méthodes et fonctions qui retournent au langage Nit depuis le code C. Lors de l'appel d'une méthode Nit, toutes les vérifications de types Nit sont réalisées dynamiquement.

Entre autres, les langages Python [v. Rossum, 2009b] et Java [Liang, 1999] (pour son interface avec C) utilisent cette approche avec des types plus ou moins généraux. En Python [v. Rossum, 2009d], où tout le typage est dynamique, il n'y a qu'un type C (PyObject∗) pour représenter toutes les entités du langage, autant les objets que les fonctions et les modules. Un exemple d'utilisation est présenté à la figure 3.1.

En Java [Gosling et al., 2005], la JNI [Liang, 1999] offre différents types C pour représenter les objets Java, les classes et les types primitifs. Toutefois, tous les objets Java sont représentés par le même type général. Un exemple d'utilisation est présenté à la figure 3.2.

Pour un langage à typage dynamique, contrairement à Nit, la seule approche cohérente est l'utilisation d'un seul type général. Ce type général évite les problèmes dus aux types complexes du langage de haut niveau, comme le polymorphisme, la généricité et les types nullables. Ces concepts ne sont pas naturels au langage C et les types généraux les camouflent.

```
1 jclass stackClass = (*env)->FindClass( env, "Stack" );
2
3 /* appel au constructeur de la class Stack */
4 jobject stack = (*env)->NewObject( env, stackClass, fInit );
```

Figure 3.2: Exemple de types C généraux de Java.
Dans ce segment de code C, le type jclass sert à contenir une classe qui est retrouvée dynamiquement depuis l'environnement Java. Le type jobject contient une instance créée par un appel à l'environnement Java selon une classe passée en argument.

Cette approche fait perdre de l'expressivité aux signatures de fonctions et aux types des variables, car que toutes les données sont d'un même type. La traduction d'une signature précise de Nit pour le langage C ne peut pas représenter aussi précisément chaque type, la signature en entier perd de la précision. De plus, toutes les vérifications du type des arguments et du retour des fonctions doivent être réalisées à l'exécution, nuisant ainsi à la performance et à la fiabilité du logiciel final.

3.1.2 Types symétriques

Le concept à la base des types symétriques est que pour chaque type Nit, un type C lui est associé. Ces types sont utilisés pour paramétrer toutes les méthodes accessibles du langage Nit ainsi que les fonctions de l'API. De façon pratique, l'ensemble des types symétriques est limité aux types utilisés. Ces derniers peuvent être déterminés depuis la signature des méthodes natives et celles des autres méthodes appelées.

Alors que nous utilisons le terme symétrique pour représenter ces types, ils sont également statiques, précis et opaques. Nous qualifions ces types de symétriques, car pour chaque type, il y a un équivalent dans les deux langages. Ils sont statiques parce qu'ils sont générés à la compilation en tant que type C normal et vérifiable dès la compilation du code C. Ils sont précis, car chaque type généré représente un type Nit distinct. Ils sont opaques dans le sens où leur forme native est inconnue au programmeur et ce dernier doit les manipuler qu'à l'aide de fonctions sans modifier leur contenu.

```
1 Stack stack = new_Stack();
```

Figure 3.3: Exemple de types symétriques avec Nit.
Dans ce segment de code C, le type C Stack sert à contenir une instance de la classe Nit Stack. Cette instance est créée par un appel à la fonction symétrique new_Stack().

Cette forme entraîne une complexité d'utilisation supplémentaire lors de la manipulation de types complexes. Entre autres, le polymorphisme du langage Nit n'est pas naturel au langage C. Ceci limite les possibilités d'automatiser les conversions de type Nit dans le code C. Nous traitons ce cas et les autres cas de types complexes dans le reste de ce chapitre.

Cette approche conserve l'expressivité des types Nit dans le code C, autant dans les signatures de fonctions que dans la déclaration de variables. Elle permet également au programmeur de se fier à la vérification de types du compilateur C, évitant ainsi la majorité des erreurs de types à l'exécution.

La JNI [Liang, 1999] applique une approche semblable pour interfacer avec le langage C++. Dans ce cas, des types précis en C représentent les types de base de Java. Ces types de base consistent en les types primitifs et les tableaux de ces types primitifs. Le polymorphisme naturel à C++ est utilisé pour simuler le polymorphisme de Java. Par exemple, le type jboolean est sous-classe de jobject et ceux-ci peuvent être utilisés pour typer précisément des données Java en C. En comparaison, nos types symétriques sont offerts pour tout type du langage Nit, et non seulement les types de base.

Nous développons cette approche dans le reste de ce document. Un exemple simple de son utilisation est présenté par la figure 3.3.

3.1.3 Types symétriques en Nit

Pour favoriser l'expressivité du code et la sûreté des types, nous avons retenu le concept des types symétriques pour l'interface native de Nit. Pour contrer le problème du polymorphisme en C, nous proposons dans ce chapitre des fonctions de conversion explicites

entre les types permis. Ceci au coût de certaines vérifications à l'exécution et d'une quantité de travail supplémentaire de la part du programmeur.

Cette représentation des types a l'avantage de préserver la sûreté à l'exécution du logiciel de types statiques Nit, ceci en permettant la vérification des types dès la compilation. Par contre, elle soulève différentes difficultés dans le langage C. Celles-ci sont discutées dans le reste de ce chapitre.

3.2 Restrictions générales pour les identifiants en C

La symétrie des types amène l'utilisation de nombreux identifiants de types en C. Ceux-ci sont générés automatiquement par le compilateur et les outils. Leurs noms doivent éviter les collisions avec le code d'implémentation en C, le code existant ainsi que le code généré. Pour assurer une compatibilité optimale, nous appliquons certaines restrictions à tout identifiant généré.

Dans cette section nous discutons des restrictions générales devant être suivies lors de la génération de tout identifiant C, dont la taille de l'identifiant, la présence de caractères spéciaux, l'utilisation d'un préfixe et le style des noms en Nit. Les détails de la génération des types C sont discutés à la section suivante.

Taille En C, le nombre de caractères significatifs dans les identifiants de variables et fonctions est limité. Pour permettre une compatibilité avec le langage Fortan, les identifiants ne peuvent pas dépasser 31 caractères. [Kernighan, Ritchie et Ejeklint, 1988] Sans viser à assurer une compatibilité avec Fortan, nous devons limiter la taille des identifiants générés. De plus, même si la longueur de l'identifiant n'influence pas le code généré, les identifiants inutilement longs compliquent l'utilisation de l'interface. Ceci restreint les possibilités d'ajout d'informations supplémentaires dans les identifiants, tel qu'un nom complet comportant l'espace de nom, le module, la classe et le nom de la propriété.

Caractères ou symboles Le langage C est moins permissif que le langage Nit en ce qui concerne les caractères composant les identifiants. Certains identifiants avancés, tels que les opérateurs, devront alors être traduits en une version texte. Dans ces cas, une association directe est réalisée pour que la traduction soit prévisible. Les équivalents en texte doivent également être formulés de façon à éviter un conflit de nom avec un identifiant existant.

Préfixe pour le système Le langage C utilise des types, des fonctions et des variables globales au système. Ceci fait en sorte qu'il peut y avoir une collision de nom entre deux types, fonctions ou variables provenant de fragments de code indépendant, mais utilisé par un même logiciel. L'interface native de Nit génère beaucoup de fonctions et types statiques, ceux-ci sont alors vulnérables à entrer en collision avec des fonctions et types provenant d'autres fragments de code.

Une façon commune d'éviter ces conflits est de préfixer toutes fonctions et tous types générés par un identifiant spécifique à notre système. L'API offert par l'interface native de Python utilise cette approche en préfixant tout ses fonctions avec Py. Nous considérons d'utiliser cette technique en préfixant les fonctions ou les types générés avec l'identifiant Nit_.

Spécification précise et respect de style de nom dans le langage Nit Une spécification précise du style des noms utilisés en Nit offre une base pour prévenir les conflits. Ce style peut être vérifié par le compilateur ou simplement définir une bonne pratique. Dans le contexte de l'interface native, nous considérons que le style du langage Nit est respecté. Nous nous permettons donc de nous y fier pour prévenir les cas de collision de noms.

En Nit, selon la spécification du langage, les noms de classes commencent par une lettre majuscule et sont en *camelcase*, tel que MaClasse. Tandis que les noms de variables, de méthodes et de modules commencent par une minuscule et les différents mots sont séparés par des caractères de soulignement, tels que ceci_est_une_methode.

Notre interface tient en compte la conversion des caractères spéciaux et se base sur le respect du style de nom en Nit pour générer tous les identifiants.

Il est impossible d'assurer le respect d'une taille maximale d'identifiant, tout en assurant l'absence de collision et que les identifiants aient une forme intuitive. Pour cette raison au moment de l'écriture, aucune restriction sur la taille de l'identifiant n'est prise en compte dans notre implémentation.

Nous considérons que l'utilisation d'un préfixe pour toutes entités générées est une bonne solution au problème de conflits avec d'autres fragments de code C. Au moment de l'écriture, nous n'avons pas implémenté le préfixe dans l'interface native de Nit, le problème peut être contourné par le programmeur. Il est de la responsabilité de celui-ci d'utiliser des types Nit qui ne causent pas de conflits. Une alternative est d'utiliser des directives de préprocesseur pour afficher au programmeur des identifiants simples, mais utiliser à la compilation des identifiants évitant ce genre de conflit. Cette alternative permet de favoriser le développement d'une interface native efficace à l'utilisation en permettant au programmeur d'utiliser des identifiants simples alors que le système assure d'éviter les conflits à la compilation.

3.3 Nom des types symétriques en C

Selon le concept des types symétriques, un type C est créé par le compilateur pour chaque type Nit. Ceci à l'exception des types primitifs, tel qu'il sera discuté à la section 3.4. Chacun de ces types doit avoir un nom intuitif et éviter les collisions avec les autres noms générés. Dans cette section, nous discutons de la forme de ce nom pour les types simples. Les types nullables et les génériques sont discutés dans les sections suivantes.

Pour le langage Nit qui n'utilise pas d'espace de nom tel que le fait Java, nous proposons que les types symétriques utilisent le même nom que celui en Nit, seulement le nom de la classe. Une classe `MaClasse` définie en Nit et utilisée comme type au travers de l'interface, par exemple dans la signature d'une méthode native, est représentée par un type nommé `MaClasse` en C.

Il est possible qu'il y ait une collision de nom entre deux classes différentes dans un même module. Au moment de l'écriture, Nit n'offre pas de système pour résoudre ces collisions au niveau du langage, lorsqu'un tel cas est détecté le compilateur soulève un message d'erreur. Pour cette raison, nous considérons que ce cas n'est pas à être traité par l'interface à ce moment. Celle-ci pourra être adaptée si une solution au problème est apportée en Nit.

3.4 Types primitifs

Nous considérons les types primitifs du langage Nit comme étant ceux qui peuvent être représentés par une valeur native équivalente de façon simple. Les entiers, points flottants, valeurs booléennes et chaînes de caractères sont de cette catégorie et ont alors une équivalence directe entre les deux langages. Les autres types Nit doivent être composés de types primitifs pour être transférables au code C.

Il y a certains cas plus complexes où un type primitif Nit est compatible avec un type C, mais la conversion peut entraîner une perte d'informations. Par exemple, lorsqu'un entier non borné Nit est traduit vers un entier borné C. De même lorsqu'une chaîne de caractères Nit contenant des caractères nuls est convertie en chaîne terminée par un caractère nul, la fin de la chaîne peut alors être perdue. Dans ce dernier cas, une deuxième représentation native existe, elle est en deux parties : un tableau de caractères et un entier représentant sa taille.

Pour permettre tout passage de données au langage C, une façon de convertir ces données doit être adoptée. Deux approches sont considérées, une conversion automatique et une, explicite.

Conversion automatique des types primitifs Une approche considérée est de convertir automatiquement les types primitifs vers leur valeur primitive au travers de l'interface. C'est-à-dire qu'alors que dans la signature d'une méthode Nit un paramètre est de type entier ou chaîne de caractères, son équivalent en C est dans

le format C équivalent. Le code C se retrouve donc à ne jamais manipuler un objet Nit représentant un type C, mais seulement sa forme C.

Les langages Java et C# utilisent une telle technique. Les types primitifs de Java sont convertis en un équivalent C automatiquement. En C#, les objets du langage de haut niveau n'ont pas de représentation valide pour le langage C, alors seuls les types primitifs peuvent être utilisés lors d'associations entre une méthode et une fonction d'une bibliothèque native.

Cette approche peut amener une perte inévitable d'informations dans les cas complexes. Par contre, elle permet une utilisation rapide dans les cas simples et communs tels que l'appel à une fonction native paramétrée d'un entier borné.

Conversion explicite des types primitifs Il est possible de ne pas convertir le type automatiquement lors du passage vers le langage C, de le représenter sous forme d'une structure opaque en C. Dans ce cas, le programmeur doit explicitement obtenir la donnée native depuis l'objet Nit.

Cette approche permet l'ajout de fonctionnalités aux types primitifs plus complexes tels que la chaîne de caractères et l'entier non borné. Par contre, elle complique les cas simples où un entier borné C est attendu.

Nous préférons convertir automatiquement les types primitifs simples tels que les entiers bornés, les points flottants et les booléens. Pour les types primitifs complexes, les entiers non bornés et les chaînes de caractères, des approches différentes sont considérées.

Au moment de l'écriture de ce document, Nit n'offre pas d'entiers non bornés, nous n'avons donc pas expérimenté avec ce concept et nous ne faisons qu'explorer les possibilités sur ce sujet. Les entiers non bornés peuvent représenter des valeurs plus grandes que celles contenues directement dans un type entier C. Les entiers naturels en C sont limités à une représentation dans un certain nombre de bits. Les entiers non bornés de Nit ne doivent donc pas être convertis automatiquement pour éviter les pertes d'informations. Ceux-ci doivent offrir un ensemble de méthodes qui permettent d'en extraire des valeurs compatibles avec le langage C. Ce qui permettrait, depuis le code C, d'en extraire une valeur compréhensible.

Pour l'instant, les entiers en Nit sont bornés à la taille d'un *long* en C moins deux bits. Ces bits sont utilisés par le système Nit pour identifier le type des instances, même pour les valeurs primitives. La taille significative des entiers Nit nous permet de les convertir directement à un *long* C sans perdre d'information. La conversion inverse elle fait perdre les deux bits les plus significatifs du *long* C.

La chaîne de caractères peut prendre différentes formes dans le langage C, soit une chaîne se terminant par une valeur nulle ou un tableau d'une taille connue. Dans ce cas, nous avons choisi d'utiliser deux types de chaînes de caractères en Nit. Une forme considérée comme native est convertie automatiquement en chaîne terminée par une valeur nulle, alors que la forme normale se comporte comme tout autre objet et n'est pas convertie au travers de l'interface. Cette dernière offre des méthodes et constructeurs pour interagir avec la forme alternative du tableau ayant une taille connue. La méthode String :: to_cstring transforme une chaîne de caractères de la forme Nit à la forme native, l'inverse est offert par le constructeur String :: from_cstring. La forme native peut donc être dynamiquement extraite de la forme normale et la taille réelle de la chaîne peut être connue par un appel de méthode.

Au moment de l'écriture de ce document, les chaînes de caractères en Nit utilisent l'encodage ASCII et sont stockées directement dans des tableau de cratères C. Pour cette raison, l'interface native agit directement sur cet encodage sans offrir de transformation d'encodage. Lorsque différents encodages seront offerts par le langage, ceux-ci seront à implémenter au niveau de l'interface native. Nous n'avons pas davantage développé ce problème.

Notre implémentation ne porte pas d'attention particulière à l'encodage des chaînes de caractères. Pour l'instant, en Nit, celles-ci n'ont pas une forme définitive. Il sera toutefois possible de réaliser les différentes conversions à l'aide de méthodes normales, telles que celles utilisées pour convertir une chaîne de caractères en la forme terminée par un caractère nul.

La combinaison de ces différentes approches a l'avantage d'offrir une conversion facultative tout en conservant le style des deux langages et de l'interface.

3.5 Conversion explicite des types en C

Le langage C ne supporte pas directement le polymorphisme, le programmeur ne peut donc pas se fier au langage pour convertir automatiquement les types. Il doit au contraire effectuer la conversion de types explicitement dès qu'une conversion est nécessaire dans le langage C.

Comme pour les types symétriques, ces conversions sont générées sur mesure à la compilation. Pour ce faire, nous profitons de la syntaxe ajoutée à Nit pour maintenir la connaissance du flot d'appels, pour également y déclarer les conversions utilisées par une méthode native. Cette information est utilisée pour générer les fonctions de conversion attendues. De plus, des fonctions de vérification de types sont offertes pour accompagner chaque fonction de conversion lorsqu'elles sont appropriées. Ces fonctions de vérification permettent au programmeur de vérifier dynamiquement le type d'un objet depuis le langage C.

Dans les cas complexes, cette approche nécessite une plus grande quantité de travail de la part du programmeur, car il doit réaliser manuellement toutes les conversions de types. Toutefois, cette manipulation est inévitable pour préserver le type statique qui permet une vérification des types à la compilation et ainsi une plus grande sûreté d'exécution du logiciel. De plus, le polymorphisme est un aspect étranger au langage C, nous considérons que le programmeur doit éviter de toucher au polymorphisme en C et privilégier d'utiliser le langage Nit lorsque le polymorphisme est largement utilisé. Par exemple, lors de l'implémentation d'une méthode native, si le programmeur doit utiliser le polymorphisme pour vérifier le type de différents objets et les convertir à un différent type, il peut implémenter cette section dans une méthode Nit normale et l'appeler depuis le code C.

Comme pour toutes autres méthodes appelées depuis le langage C, l'outil de génération d'échafaudages est utilisé pour présenter au programmeur la signature des fonctions de conversion et de vérification. Ces signatures apparaissent dans le code généré près des fonctions à implémenter.

La figure 3.4 présente un exemple de fonction de conversion explicite tel qu'utilisé par l'interface native de Nit.

```
1 fun foo is extern import A.as( B ), Animal.as( Object ),
2                           Object.as( Animal )
```

```
1 /* conversion */
2 B A_as_B( A v );
3 Object Animal_as_nullable_Object( Animal v );
4 Int Object_as_Animal( Object v );
5
6 /* verification */
7 int A_isa_B( A v );
8 /*int Animal_is_a_Object( Int v ); non generee car toujours vrai */
9 int Object_is_a_Animal( Object v );
```

Figure 3.4: Exemple d'utilisation d'une fonction de conversion.

La déclaration de l'utilisation d'une fonction de conversion se fait parmi les déclarations explicites des méthodes appelées depuis le code C.

Le segment de code C présente les identifiants de ces fonctions, de conversion et de vérification, qui sont générés en combinant le nom du type source et le nom du type de destination. Les vérifications inutiles, celles qui sont nécessairement vraies, ne sont pas générées pour éviter de confondre l'utilisateur.

3.6 Types nullables

Le langage Nit, compte sur les nullables pour assurer une plus grande sûreté à l'exécution et optimiser le programme [Gélinas, Gagnon et Privat, 2009]. Ceux-ci servent à différencier les types pouvant être de valeur nulle de ceux étant assurément instanciés après la construction d'un objet.

Même si, avec les pointeurs, le concept de valeur nulle est présent en C, il n'est pas directement compatible avec les autres types utilisés par l'interface. Dans cette section, nous considérons les différentes alternatives qui permettent de représenter les types nullables en C, tout en visant à préserver la sûreté d'exécution que ces types amènent au langage. Le sujet est approché selon deux points de vue, d'abord à savoir si les types nullables en C sont distincts des non nullables puis, nous nous interrogeons sur la représentation d'une donnée de valeur nulle en C.

Tous les types sont nullables Une façon de représenter les types nullables est de considérer tous les types symétriques comme étant nullables en C. Alors qu'un objet Nit est opaque et sa valeur a peu d'incidence en C, celui-ci peut être de valeur nulle sans que sa forme ne soit en rien changée.

Ceci entraîne une perte d'expressivité et de précision par rapport à sa définition en Nit. Des vérifications dynamiques de types doivent être insérées pour vérifier que les valeurs provenant de C ont bien une valeur significative si un non nullable est attendu. Le besoin de ces vérifications entraîne la perte de la sûreté d'exécution amenée par les types C.

De plus, les types primitifs, qui ont normalement une forme équivalente en C, perdent cette caractéristique lorsqu'ils sont nullables. Si une valeur normalement primitive est à nulle, elle n'a alors plus de formes équivalentes en C. Considérer tous les types en C comme étant nullables est donc difficilement compatible avec l'approche adoptée pour les types primitifs.

Cette approche est suffisamment simple pour être naturelle au langage C. Les types Nit se comportent alors comme les types normaux. Aucune conversion de type n'est nécessaire ; il s'agit de l'option la plus rapide d'utilisation.

Types distincts La représentation des types Nit par des types distincts entre les types nullables et non nullables conserve toute leur précision. Ceci permet également de maintenir la sûreté à l'exécution amenée par les types nullables.

Par exemple, en Nit, le type Object et sa version nullable seraient représentés en C, respectivement par Object et nullable_Object. Cette forme évite toute collision avec d'autres noms générés grâce au style de nom des classes en Nit, lesquelles ne peuvent pas commencer par une lettre minuscule.

Le langage C ne comprenant pas le concept de polymorphisme, la responsabilité de le gérer revient au programmeur. Il doit alors se servir de fonctions de conversion entre les types nullables et non nullables.

Cette approche règle le problème des types primitifs en les offrant sous deux formes. Leur forme nullable, incompatible avec le code C, conserve une forme d'un type statique et opaque, alors que leur forme non nullable est convertie automatiquement. Les fonctions de conversion peuvent être utilisées pour passer d'une forme à l'autre, au besoin.

Représentation par une donnée opaque Pour représenter une valeur nulle du langage Nit dans le langage C, nous considérons d'utiliser un type opaque. Une nouvelle instance de celle-ci est obtenue par l'appel d'une fonction générée automatiquement dès l'utilisation d'un type nullable depuis C.

Cette approche a l'avantage, à l'implémentation, de pouvoir traiter toutes les données passées de la même façon. La donnée opaque contient normalement des données du format naturel au compilateur qui peuvent ainsi être intégrées directement.

Représentation avec NULL Le langage C a une représentation similaire aux types nullables avec les pointeurs qui peuvent être de valeur nulle, il est possible de représenter la valeur nulle du langage Nit par un pointeur de valeur nulle en langage C. Pour ce faire, le type opaque généré doit être de type pointeur.

Dès qu'un type nullable du langage Nit est attendu par une méthode symétrique en C, le programmeur peut alors retourner un NULL standard à C. De même qu'il est naturel qu'il assigne NULL à toutes variables d'un type symétrique.

```
1 fun foo is extern import Object.as( not null ), Object.as( nullable )
```

```
1 /* conversion */
2 Object Object_as_not_null( nullable_Object v );
3 nullable_Object Object_as_nullable( Object v );
4
5  /* verification */
6 int Object_is_null( nullable_Object v );
```

Figure 3.5: Utilisation des types nullables au travers de l'interface native.
Le premier fragment de code est en Nit et déclare explicitement l'utilisation de service de conversion entre la version nullable et non nullable d'un même type. Le second fragment de code est en C et affiche la signature native de ces fonctions de conversion et vérification.

> Cette forme nécessite un peu de travail à l'exécution pour surveiller le retour vers le langage Nit depuis les méthodes natives. Ceci dans le but de détecter les retours de NULL qui devront être traduits en un format compatible avec le système de Nit.

Nous avons retenu l'utilisation des types distincts dans la conception de l'interface native de Nit. Les types distincts permettent d'assurer la sûreté d'exécution du logiciel et de conserver l'expressivité des types au travers de l'interface. Combiné avec l'utilisation de NULL, ceci permet de se servir naturellement de la forme nullable depuis le langage C. Au moment de l'écriture de ce document, nous n'avons pas implémenté l'utilisation de NULL dans l'interface native de Nit, l'approche de la donnée opaque pour représenter une valeur nulle est employée.

Pour déclarer explicitement la conversion entre nullables et non nullables, des formes plus naturelles et rapides sont offertes. Ceci s'applique uniquement entre deux types dont la seule différence est que l'un est nullable. Un exemple de cette forme est présenté à la figure 3.5

3.7 Types génériques

Dans les langages à typage statique, tel que le langage Nit, les types génériques sont largement utilisés, mais ils ne sont pas naturels au langage C. Pour une utilisation naturelle depuis le langage C, ces types doivent donc être accommodés au travers de l'interface.

Les types génériques sont représentables directement par un type dynamique en C, tels qu'en Java et Python. Par contre, en Nit, l'application de la symétrie des types complexifie leur représentation, nous discutons de ces problèmes dans cette section.

Tout en nous basant sur la forme d'un type symétrique en C, nous avons considéré différentes approches pour représenter les types génériques. Le principal défi est d'utiliser la signature native la mieux adaptée pour chaque méthode d'une classe générique ou pour chaque méthode ayant comme paramètre un type générique. Ceci, dans le but de permettre une manipulation naturelle des types génériques depuis le langage C.

Définir le type selon la classe seulement L'approche la plus simple est de préserver la forme que prend tout autre type symétrique en C. C'est-à-dire, d'utiliser que le nom de la classe pour définir le type C qui lui est associé, même si la classe est générique. Ceci implique alors que tous les sous-types paramétrés d'un type générique sont représentés par un seul même type en C. Par exemple, une liste d'entiers et une liste de chaînes de caractères sont identifiées comme étant une simple liste en C. Ceci amène une perte de précision des types Nit en C lors de l'utilisation de ces types au travers de l'interface native. Par exemple, pour une méthode Nit qui attend un argument paramétrable de type précis, la méthode équivalente en C attend un objet nullable.

Il est possible de répondre à ce manque de précision par une vérification dynamique et ainsi préserver la validité des données, mais cela fait perdre la vérification statique des types. De plus, lors d'une utilisation normale, une conversion explicite est nécessaire pour passer une variable du type attendu à la méthode générée. Ceci est non-intuitif et complexifie grandement l'utilisation de tous les types génériques depuis le langage C.

Définir le type selon la classe et ses paramètres Il est possible de profiter de la déclaration explicite des appels de méthode depuis le code C pour obtenir davantage d'informations de la part du programmeur. Nous étudions la possibilité de laisser la liberté au programmeur d'identifier les classes génériques autant par leur nom seul que par leur identifiant paramétré.

Lorsque le paramètre est précisé, il est possible de représenter ce type en C de façon précise. Ceci permet de distinguer une même classe paramétrée différemment, par exemple, de distinguer une liste d'entiers d'une liste de chaînes de caractères, respectivement représentées par List_of_Int et List_of_Object. Ce type précis est donc utilisé pour paramétrer toutes les méthodes où il est précisé comme tel lors de la déclaration explicite des appels.

Cette approche est syntaxiquement moins naturelle, normalement le programmeur s'attend à identifier une méthode à l'aide du nom de sa classe sans qu'elle soit paramétrée. Ceci peut même entraîner de multiples déclarations d'appels de la même méthode, mais selon des paramètres différents.

Le grand avantage de cette approche est qu'à l'utilisation en C, chaque méthode générée attend les arguments du type demandé. Ceci permet, dans le code C, d'éviter au programmeur les conversions explicites de types qui sont non intuitives.

Les collisions de noms sont évitées grâce au style de nom des classes Nit. Celles-ci ne comportent pas de caractères de soulignement ni de mot débutant par une lettre minuscule. Cette forme est alors intuitive et évite les collisions. Les paramètres des classes génériques à plusieurs paramètres peuvent être accumulés et ainsi prendre la forme de Map_of_String_Object pour un dictionnaire associant des chaînes de caractères à des objets.

Cette forme de nom doit prendre en compte les paramètres qui sont eux-mêmes paramétrés. Pour un type tel que A[Int,B[String],Object], les paramètres de la classe B doivent être délimités par rapport à ceux de la classe A. Dans ces cas, alors que le _of_ délimite le début des paramètres lors de tout utilisation de types paramétrés, nous ajoutons un caractère de soulignement après le dernier paramètre, et ce, uniquement lorsque le type est imbriqué. Un type paramétré complexe prend donc la forme de A_of_Int_B_of_String__Object. Cette forme est moins naturelle, mais permet d'éviter efficacement cette rare possibilité de collision.

Nous avons retenu de n'utiliser que la forme paramétrée pour l'interface native de Nit. Celle-ci évite le besoin de conversion manuelle en C et assure une meilleure vérification statique des types. Au moment de l'écriture, nous avons implémenté que la forme plus simple où le type est général à chaque classe générique, dans le compilateur du langage Nit.

Selon l'approche retenue, les types C sont définis selon des paramètres précis, lesquels sont déterminés de différentes façons. Pour les méthodes natives d'une classe générique, le type symétrique du receveur est déterminé selon la borne des paramètres de la classe. Les paramètres des méthodes symétriques de classes non génériques sont tout simplement typés selon le type de la signature. Le receveur des méthodes symétriques des classes génériques est typé selon un paramètre passé suite au nom de la classe lors de la déclaration explicite de leur appel en Nit, ceci nécessite donc une syntaxe alternative pour la déclaration explicite d'utilisation des classes génériques. La figure 3.6 présente des exemples de types symétriques de classes génériques.

3.8 Durée de vie des types symétriques en C

Dans cette étude, nous proposons les types symétriques pour représenter les références à des objets Nit depuis le langage C. Il reste cependant un grand défi, celui d'assurer la validité de ces références dans les différents cas d'utilisation. Celles-ci peuvent être utilisées en tant qu'arguments d'une méthode native, en tant que variables locales en C ou encore comme variables globales.

La validité de ces références est mise en jeu par l'utilisation d'un ramasse-miettes par le langage Nit. Celui-ci peut être invoqué à tout moment pendant l'exécution de code Nit. Dès son invocation, un objet peut être déplacé en mémoire ou simplement effacé s'il est non référencé depuis Nit. Une référence directe, conservée en C serait alors rendue invalide.

Ce problème peut être évité par l'utilisation d'une indirection telle qu'une structure qui est mise à jour par le ramasse-miettes. Celle-ci peut prendre différentes formes selon les cas d'utilisation ou alors, elle peut les combiner.

```
1 class A
2     fun foo( list_a : List[ Int ] ) : List[ String ] is extern
3     fun bar( list_b : Map[ String, Object ] ) is extern
4         import Map[ String, Object ]::length
5 end
6 class B[ Comparable ]
7     fun baz is extern
8 end
```

```
1 List_of_String impl_A_foo( A recv, List_of_Int list_a ) { }
2
3 void impl_A_bar( A recv, List_of_String_Object list_b ) {
4     Map_of_String_Object_length( list_b );
5 }
6
7 void impl_B_baz( B_of_Comparable recv ) { }
```

Figure 3.6: Exemple d'utilisation de types génériques.

Le premier fragment de code en Nit utilise des types génériques parmi la déclaration explicite d'appel de méthodes.

Le second fragment de code présente la représentation en langage C pour ces utilisations. La première fonction est l'implémentation de la méthode native foo de la classe A qui reçoit comme un argument de type List_of_Int . La seconde fonction est l'implémentation de bar de la classe A qui utilise un type générique à plus d'un paramètre. La fonction baz de la classe B, une classe générique, est implémentée nativement à l'aide d'un receveur de type Comparable.

Les détails du format des noms des types symétriques sont discutés à la section 3.3.

Les arguments des méthodes natives peuvent être déterminés à la compilation et ainsi leur validité sera assurée de la même façon que les arguments en Nit. En Nit, tous les arguments de méthodes sur la pile d'appels sont préservés à l'évocation du ramasse-miettes. Il est alors aisé d'étendre cette fonctionnalité aux arguments des méthodes natives.

Il y a quatre façons d'obtenir une référence à un objet Nit depuis le code C. L'objet peut être obtenu depuis un argument d'une méthode native (cas traité précédemment), depuis une variable globale (cas traité ultérieurement), depuis une autre variable locale (ce même cas) ou depuis l'appel à une méthode symétrique. Le cas problème est lorsque l'objet est obtenu depuis une méthode symétrique. Les méthodes symétriques sont générées automatiquement selon leur utilisation dans un module. Il est donc possible d'adapter la méthode symétrique générée pour que les objets qu'elles retournent soient associés à la méthode native en cours d'exécution. De ce point, il est possible de les traiter d'une façon semblable aux arguments des méthodes natives et ainsi d'en assurer la validité pour toute l'exécution de la méthode native. Comparativement à une implémentation semblable en Nit, le danger côté performance est qu'à chaque évocation du ramasse-miettes tous les objets récupérés sont préservés par le ramasse-miettes. L'équivalent en Nit ne préserve que les objets référés. Leur traitement par le ramasse-miettes est équivalent en Nit. Le pire cas est lorsque le code C assigne répétitivement différents objets à une même variable. Ce cas est peu coûteux en Nit pur, mais très couteux à l'évocation du ramasse-miettes dans une méthode native.

Contrairement aux arguments des méthodes natives et des variables locales, les références globales à un objet ont une durée de vie qui dépasse l'exécution d'une méthode native. Ces références peuvent être utilisées par des fonctions natives lors d'un appel subséquent ou pour des rappels asynchrones. Les références globales ne sont pas encouragées avec les langages de programmation à objets, toutefois leur utilisation est inévitable dans certains cas, tel que la gestion des signaux POSIX. Dans ce cas, l'interface nécessite davantage d'informations de la part du programmeur pour connaître la durée de vie d'une telle référence.

Dans le reste de cette section, nous présentons les différentes approches que nous avons considérées pour exposer ces références au programmeur. Ces approches consistent en l'utilisation d'une structure propre aux références et d'étendre les types symétriques en C.

Structure de référence Une approche consiste en l'utilisation d'une structure C conservant une référence vers un objet Nit. Pour ce faire, une fonction de l'API permet d'acquérir une référence depuis un argument ou une variable locale dans le code C. Une autre fonction permet de libérer la référence et ainsi rendre l'objet sujet au ramasse-miettes à sa prochaine exécution. Une troisième fonction retourne l'objet Nit contenu par une référence donnée, ce qui permet alors d'utiliser l'objet pour tout appel de méthodes.

Un avantage à cette solution est l'expressivité de son utilisation qui affiche clairement au programmeur la différence entre une référence globale et un argument ou une variable locale. Ceci peut encourager le programmeur à en faire une utilisation plus propre et ainsi ne pas oublier de libérer des références.

Par contre, cette utilisation amène une duplication de tous les types symétriques en C, l'une, pour les références globales et l'autre, pour les arguments de méthodes natives et les variables locales. De plus, cela oblige le programmeur à faire appel de fonction supplémentaire pour obtenir la valeur réelle conservée par la référence à chacune de ses utilisations.

Étendre la représentation native des types symétriques La seconde solution est d'étendre les types symétriques existants, accessibles comme arguments ou comme variables locales, avec un compteur les protégeant du ramasse-miettes. Cette solution s'avère tout aussi complexe que la précédente, mais en expose peu au programmeur.

Deux fonctions servent à incrémenter et décrémenter le compteur de référence de chaque objet. Celles-ci doivent tout de même être explicitement invoquées par le programmeur au bon moment. Le langage Python utilise une telle forme à l'aide

```
1 Py_INCREF( PyObject *o );
2 Py_DECREF( PyObject *o );
3 Py_CLEAR( PyObject *o );
```

Figure 3.7: Fonctions natives de contrôle du compte de références d'un objet Python.
La fonction Py_INCREF provient de l'API de l'interface native de Python pour incrémenter manuellement le compte de références de l'objet passé en argument. Quant à elles, les fonctions Py_DECREF et Py_CLEAR manipule le compte de références de l'objet respectivement en le décrémentant d'une unité et en lui assignant la valeur zéro.

de fonctions manipulant le compteur de référence d'un objet, les signatures de celles-ci sont présentées par la figure 3.7.

L'avantage principal de cette solution est la clarté à l'utilisation, n'ayant pas plus d'un type pour la même classe d'objets Nit, sans distinction entre les types de références, les arguments de méthodes et les variables locales.

Le désavantage est la complexité des structures nécessaires à l'implémentation. Toutefois, selon l'expérimentation, il est possible d'implémenter ces mêmes systèmes pour que l'ajout et le retrait des références ne soient pas trop coûteux en travail.

Nous avons retenu d'étendre la représentation native des types symétriques pour représenter les références globales à des objets Nit. Celle-ci amène l'utilisation la plus naturelle et permet d'éviter des manipulations superflues. Elle implique toutefois l'utilisation de fonctions d'incrémentation et de décrémentation du compte de références de l'objet.

Techniquement dans l'interface native de Nit, les fonctions manipulant le compte de références sont générées pour chaque type symétrique, nous proposons les fonctions de forme incr_ref_Object(Object o) et decr_ref_Object(Object o). Pour conserver une variable locale ou un argument en tant que référence globale, le compte de références de l'objet doit être incrémenté avant le retour de la méthode native. L'objet peut après être

conservé dans une variable globale, mais son compteur doit être décrémenté avant que la référence ne soit perdue.

Cette approche peut être couplée avec celle permettant le maintien de la connaissance du flot d'appels. Ceci permet alors de préciser quelle méthode native initie une référence globale et quelle autre la libère. Toutefois, dans cette étude et pour le langage Nit, nous jugeons cette connaissance comme étant superflue et le travail supplémentaire de la part du programmeur n'apparaît pas justifié.

3.9 Utilisation du ramasse-miettes en C

Dans le but de profiter de l'optimisation de la gestion de la mémoire offerte par le ramasse-miettes du langage Nit, nous avons considéré de l'utiliser depuis le langage C. Il peut ainsi remplacer les fonctions natives habituelles pour réserver, agrandir, libérer et réaliser toute autre manipulation sur des espaces mémoire. Permettre cette utilisation étend plusieurs des avantages de l'emploi du ramasse-miettes au langage C.

De plus, selon l'implémentation du ramasse-miettes, il peut être plus efficace lorsqu'il possède une meilleure connaissance de l'utilisation globale de la mémoire. Il profiterait alors de connaître l'espace mémoire utilisé par tout le logiciel, même ce qui est uniquement manipulé en C.

Toutefois, cette approche reporte la logique des paradigmes de haut niveau au langage C et s'éloigne de l'utilisation habituelle du langage C. De plus, il existe des bibliothèques natives spécialisées pour ce genre de besoin et nous jugeons qu'il ne relève pas de l'interface native d'offrir un service de ramasse-miettes pour des données non manipulées par Nit. Nous n'avons pas retenu la possibilité d'offres de services donnant accès au ramasse-miettes dans cette étude, autre que les références globales.

3.10 Implémentation des types symétriques dans le compilateur Nit

Nous avons implémenté ce système de types symétriques dans le compilateur du langage Nit. Ces types sont opaques au programmeur et leur implémentation interne peut changer. Cette section présente les détails intéressants de l'implémentation actuelle de ces types.

Chaque type symétrique qui n'est pas primitif ou non nullable est généré sous forme d'un pointeur à une structure. Ces structures sont toutes du même format, mais sont définies individuellement comme types distincts. Elles sont définies dans le fichier d'entête des services intermédiaires et celui-ci est importé par les fichiers en C du module hybride.

Ces structures comportent une référence à l'objet Nit ainsi que les données nécessaires au ramasse-miettes pour que ces structures servent en tant qu'arguments, variables locales ou références globales. Lors du passage d'un objet Nit entre le système Nit et le code utilisateur, une nouvelle instance de cette structure est créée pour envelopper l'objet Nit et seule la référence à cette structure est passée au code utilisateur. Un programmeur bien intentionné ne manipule pas directement l'objet Nit, mais seulement la référence.

3.11 Contribution et atteinte des objectifs

Les types symétriques sont un des apports les plus importants de cette recherche. Notre solution s'étend plus loin que les approches similaires existantes dans les autres interfaces. Par exemple, l'interface native de Java pour C++ offre certains types C++ précis associés aux types Java primitifs. Ces types sont limités aux types primitifs, à l'objet général et aux tableaux de ces types. En comparaison, notre solution est appliquée à tous les types Nit utilisés en C. Dans ce cas, chaque type Nit est associé à un type C et ils sont tous vérifiés statiquement.

L'application du concept des types symétriques dans l'interface native de Nit aide à l'atteinte des objectifs suivants :

Syntaxe naturelle en C Les types symétriques offrent des types C naturels au langage C pour manipuler les types Nit. L'utilisation de ces types permet d'atteindre l'objectif de préserver une syntaxe naturelle en C.

Sûrete à l'exécution du logiciel Les types symétriques sont des types statiques C normaux, leur utilisation peut donc être vérifiée statiquement à la compilation. Cette vérification permet d'assurer une certaine sûreté à l'exécution du logiciel en détectant les erreurs de manipulation de types.

CHAPITRE IV

MÉTHODES SYMÉTRIQUES

Comme nous discutons au chapitre 2, le langage C est utilisé pour appeler des méthodes Nit. Nous proposons d'adopter une forme de méthodes symétriques similaire à celle des types symétriques. Ces derniers sont utilisés pour typer statiquement les méthodes Nit appelées depuis le langage C.

Dans ce chapitre, nous discutons des formes considérées pour représenter les méthodes Nit en C, de l'utilisation des fermetures au travers de l'interface native, de la sûreté d'exécution amenée par les types et méthodes symétriques, des possibilités en collisions de noms des méthodes symétriques, des détails d'implémentation des méthodes symétriques dans le compilateur.

4.1 Méthodes Nit en C

L'appel d'une méthode implémentée dans le langage Nit depuis le code C est une fonctionnalité importante de l'interface. Tout comme pour les types, le choix de la forme suit deux principales possibilités, une forme statique ou une forme dynamique.

Méthodes récupérées dynamiquement Cette approche permet de récupérer dynamiquement une méthode Nit, avec son nom et classe, depuis l'environnement d'exécution. Celle-ci prend alors la forme d'une structure de données opaques qui est référée lors de l'appel d'une fonction de l'API qui réalise l'invocation de la méthode Nit.

```
1 jclass classe = env->FindClass( "FibonacciEngine" );
2 jmethodID fonction = env->GetStaticMethodID( classe ,
3                                   "Fibonacci", "(I)I" );
4 jint r = env->CallStaticIntMethod( classe , fonction , 12 );
```

Figure 4.1: Exemple de méthodes dynamiques avec Java.
La classe FibonacciEngine est retrouvée dynamiquement depuis l'environnement. De même pour la fonction Fibonacci qui est retrouvée depuis sa classe, son nom et sa signature. L'appel de la fonction se fait via un appel à CallStaticIntMethod en précisant la classe, la fonction à invoquer et les arguments à passer.

Cette forme est fréquemment utilisée dans les interfaces natives officielles à un langage, par exemple, en Java [Gosling et al., 2005]. Au besoin, une classe est obtenue à l'aide de l'environnement, les méthodes sont retrouvées dans la classe et invoquées sur l'objet. Le tout est réalisé dynamiquement à l'aide de chaînes de caractères C. La figure 4.1 présente un exemple d'utilisation de cette forme en Java.

Méthodes générées statiquement La technique d'appel découle directement de l'utilisation de types symétriques. Nous pouvons appliquer la même approche que pour les types aux méthodes, c'est-à-dire générer des équivalents C de toutes méthodes Nit appelées depuis le code C.

Cette approche bénéficie de deux concepts introduits précédemment, les types symétriques qui composent la signature et le maintien de la connaissance du flot d'appels. Grâce à ce dernier, les méthodes appelées sont connues précisément. Leur nombre limité peut alors être généré statiquement.

Il y a tout de même certaines différences entre une méthode Nit et une fonction C qui, du premier regard, nuit à une symétrie parfaite. En Nit, il n'y a que des méthodes, alors qu'en C, il n'y a que des fonctions et procédures. Toutefois, il est standard que ces dernières opèrent sur une structure de données et que celle-ci soit alors passée comme premier ou dernier argument à la fonction. On peut en tirer une façon d'assurer la compatibilité entre les deux langages tout en conservant

```
1 /* Pour Object::to_s : String */
2 String Object_to_s( Object recv );
3
4 /* Pour Object::print( text : String ) */
5 void Object_print( Object recv, String text );
```

Figure 4.2: Exemple de méthodes symétriques avec Nit.
Ce segment de code C présente des signatures de méthodes symétriques utilisant aussi les types symétriques. Les deux s'appliquent à toute instance de Object et seulement la première retourne une valeur qui est une instance de String.

leurs styles respectifs. Alors que le receveur d'une méthode en Nit est implicite dans sa signature, il sera passé comme premier argument à la fonction native équivalente.

La méthode accessible depuis le code C a un nom généré selon une nomenclature précise des types et méthodes symétriques. Avec ce nom, ces méthodes peuvent alors être appelées depuis le code C comme toutes autres fonctions.

L'approche statique s'intègre facilement à l'interface native de Nit et elle amène une grande sûreté d'exécution en assurant la vérification statique des appels de méthodes depuis le code C, ainsi qu'en assurant la validité des types passés en arguments.

Le nom de l'équivalent C des méthodes Nit est déterminé à partir du nom de la classe receveur et du nom de la méthode. À l'implémentation, un algorithme est utilisé pour éviter les collisions, nous discutons de cette approche à la section 4.4. Dans leur signature, ces fonctions retournent l'équivalent C du type de retour de la méthode et prennent comme premier argument le receveur.

4.2 Fermetures Nit en C

Au moment de l'appel d'une méthode, les fermetures permettent d'en adapter le comportement en fournissant des fragments de code. Préserver leur utilisation en C s'avère être un grand défi au niveau de l'interface native.

Ces fermetures affectent l'interface native de Nit à deux endroits : les méthodes natives et les méthodes symétriques. Dans le cas des méthodes natives, les fermetures visent à être exécutées depuis le langage C, leur utilisation prend alors une forme semblable à celle de l'appel de méthode depuis le code C. Pour les méthodes symétriques, elles sont à être précisées à l'appel depuis le langage C.

En Nit, les fermetures offrent la possibilité de manipuler le contexte d'exécution de la méthode appelante. Par exemple, une fermeture appelée depuis le corps d'une boucle peut en interrompre l'exécution par l'utilisation d'un break ou sauter au prochain passage avec un continue. De plus, une fermeture peut accéder aux variables locales et appeler les méthodes accessibles depuis la méthode appelante. En ce langage, les fermetures sont nommées, c'est-à-dire qu'à la définition de la méthode chaque fermeture est définie avec un nom et qu'à l'appel de la méthode les fermetures sont précisées à l'aide de ce nom.

Dans cette section, nous traitons les fermetures par la même approche que les types et les méthodes, c'est-à-dire en tentant de leur trouver une forme symétrique. Dans cette optique, nous tentons de trouver une forme qui s'utilise autant avec les méthodes natives qu'avec les méthodes symétriques. Pour chaque forme considérée, nous évaluons sa compatibilité avec ses deux utilisations possibles.

Les ignorer et les interdire lorsque nécessaire L'option la plus simple est d'ignorer les fermetures au travers de l'interface native. Selon cette approche, une méthode native peut être définie avec des fermetures, mais les fermetures ne peuvent pas être appelées depuis le code C. Pour les méthodes symétriques, seules celles sans fermeture et celles avec des fermetures facultatives peuvent être appelées depuis C.

Cette interdiction est vérifiable dès que l'appel à une méthode est déclaré explicitement et qu'une méthode est déclarée native.

Cette approche limite les méthodes appelables depuis le code C et celles qui sont redéfinissables en tant que méthode native. Pour le langage Nit, considérant que les fermetures sont peu utilisées au moment de l'écriture, restreindre leur utilisation d'une telle façon ne limite pas tant de possibilités. De plus, le programmeur peut définir des méthodes en Nit pour servir d'indirection et y traiter les fermetures.

Pointeurs de fonction native Une approche naturelle est d'utiliser ce qui est le plus près d'une fermeture en C, soit le pointeur de fonctions. Il serait alors possible d'étendre les signatures des méthodes natives et des méthodes symétriques pour inclure les fermetures sous cette forme parmi les paramètres.

Le langage C#, dont l'interface native prend la forme d'une association directe avec des fonctions natives associe le concept de délégués avec des pointeurs de fonctions C. Ces délégués agissent de façon semblable à des pointeurs de fonctions en C# et conservent une référence au récepteur, lorsqu'ils pointent vers une méthode. Ils peuvent être utilisés en tant qu'argument lors de l'appel à une fonction native. Dans ce cas, en C, un pointeur de fonction, naturel au langage, représente le délégué.

Cette approche peut être viable pour des fermetures simples, mais dès que la fermeture peut entraîner des changements de contexte brusques, accéder à des variables locales au contexte d'appel et autres, le langage C s'avère trop limité. Ceci ne permet donc que d'utiliser des fermetures simples et non pas d'utiliser les méthodes qui attendent un comportement avancé des fermetures. Il serait possible d'étendre cette solution par l'utilisation de structure complexes offrant toutes les fonctionnalités des fermetures au langage C, mais nous jugeons que ceci ne fait qu'amener une syntaxe moins naturelle en C.

Certaines fonctionnalités apportées aux fermetures par le langage Nit, telles que la manipulation du contexte d'exécution, peuvent être implémentées à l'aide d'ajouts à l'API de l'interface native. Par exemple, une fonction peut servir à marquer les manipulations de contexte en utilisant une variable globale, le tout serait interprété lors du retour dans le système de Nit. Cette variante est toutefois peu naturelle au langage C et ce genre de manipulation est plus adaptée au langage Nit.

Référence à une méthode du langage Nit Dans le cas des méthodes symétriques, passer une référence à une méthode Nit pour remplacer la fermeture est une possibilité un peu complexe. La référence peut prendre la forme d'une nouvelle structure de données identifiant la méthode, ou encore elle peut bénéficier du paradigme à objets. Pour ce faire, une classe abstraite définit une méthode à être invoquée

en tant que fermeture et les sous-classes de celle-ci peuvent être utilisées comme contenants à fermetures pour C.

Cette utilisation est lourde, nécessitant d'implémenter partiellement l'appel de fonction dans les deux langages de l'interface. De plus, elle n'est pas naturelle au langage de bas niveau, lequel se retrouve à manipuler des objets alors que le programmeur s'attend à des fonctions.

Préciser les fermetures lors de la déclaration d'appel Pour les méthodes symétriques, dans l'optique où les fermetures sont naturelles seulement au langage Nit, nous considérons restreindre leur utilisation au langage Nit. Pour ce faire, il est possible de définir les fermetures dès la déclaration explicite de l'appel à une fonction. Celle-ci aurait alors accès aux fonctionnalités avancées telles que l'accès aux propriétés de la classe et la manipulation du flot d'exécution.

Le principal désavantage de cette approche serait qu'elle changerait la raison d'être de ces déclarations. Elles ne serviraient plus seulement pour marquer le flot, mais aussi pour réaliser une partie de l'appel. De plus, il faudrait pouvoir conserver la possibilité d'appeler la même méthode avec différentes fermetures. Si les fermetures sont associées directement à la méthode appelée, un ajout à la syntaxe serait nécessaire pour permettre de dissocier les différentes formes d'appels à la même méthode.

La combinaison d'une syntaxe différente des autres appels de fonctions et le fait d'y déclarer une partie de l'appel rend cette approche lourde et peu naturelle.

Au moment de l'écriture de ce document, nous retenons l'approche d'interdire simplement les appels à des méthodes avec fermetures. Cette interdiction n'est pas si coûteuse dans le cas du langage Nit, ce point sera toutefois à reconsidérer avec l'évolution du langage.

L'approche de l'utilisation d'un pointeur a aussi été jugée très intéressante. Elle permet de conserver la pureté des différents codes tout en ne limitant pas l'utilisation des méthodes existantes depuis le code C.

4.3 Sûreté d'exécution par vérification des types à la compilation

La symétrie des types contribue à répondre à l'un des objectifs de cette étude, soit de préserver la sûreté à l'exécution du logiciel utilisant l'interface native. Alors que d'autres interfaces natives réalisent surtout des vérifications dynamiques [Lee et al., 2010; Tan et al., 2006] ou se fient au respect de la spécification par le programmeur, l'interface native que nous proposons se base sur une vérification statique grâce à la symétrie des types.

La symétrie des types permet d'assurer la validité de leur utilisation au travers de l'interface. Chaque type utilisé étant précis et statiquement vérifiable, le programmeur ne peut pas passer par erreur une donnée du mauvais type à une méthode Nit. Il reste la vérification lors de la conversion explicite de type qui est réalisée à l'exécution, mais cette conversion devant être réalisée explicitement, le programmeur peut difficilement y commettre une erreur d'inattention.

Les méthodes Nit qui sont représentées par une fonction statique en C permettent d'éviter les erreurs reliées à la recherche dynamique de modules, de classes et de méthodes. Ces erreurs à l'exécution peuvent survenir lorsqu'une de ces entités n'est pas trouvée due à une évolution du logiciel ou à une erreur humaine. Dans ce cas, la symétrie des méthodes assure, dès la compilation, que le nom de la fonction invoquée est adéquat et que l'appel sera valide à l'exécution.

À l'exception des types primitifs, le fait que tout objet Nit soit manipulable par ses méthodes uniquement évite les problèmes propres à C, tels que les accès hors bornes à un tableau. Dans ce cas, pour accéder au contenu d'un type tableau ou séquence depuis le code C, le programmeur doit utiliser la même méthode qui serait utilisée en Nit. Cette dernière réalise donc les mêmes vérifications dynamiques et l'intégrité de la mémoire et des données obtenues est assurée.

4.4 Collision entre les noms générés

La symétrie des méthodes et des types amène la génération de nombreux types et fonctions pour le langage C depuis leurs noms en Nit. Pour ce faire, différentes approches sont utilisées pour identifier une méthode et l'associer à sa classe, pour nommer un type paramétré et pour traduire de façon fonctionnelle les caractères spéciaux tels que les opérateurs. Ces transformations supplémentaires amènent des possibilités de collisions de noms en C, même dans les cas où aucun de ceux-ci n'était présent en Nit.

Par exemple, selon une implémentation naïve, la propriété MaClasse::+ et MaClasse::plus pourraient toutes deux être traduites en C pour MaClasse_plus.

Nous traitons l'aspect plus technique des conflits de noms en C ainsi que la composition des identifiants C à la section 3.2. Dans cette section, nous présentons les approches considérées pour résoudre le problème de collision de noms de méthodes issues de la génération automatique, spécifiquement pour le langage Nit. Ces approches consistent en la séparation des composants d'un nom, ajout d'un compteur, interdire les cas de conflits, le traitement des fonctions spéciales et des caractères spéciaux.

Séparation entre les composants du nom Une forme de séparation doit être adoptée pour les noms composés tel que celui d'une méthode, laquelle est identifiée à l'intérieur d'un module par le nom de la classe et le nom de la méthode. Il est possible de les accoler directement ou encore, d'intercaler un ou plusieurs caractères. Dans ce cas, la solution la plus simple est de séparer le nom de la classe et la méthode avec un caractère de soulignement, tel que MaClasse_ceci_est_une_methode.

En Nit, la combinaison de ces deux styles encourage l'emploi d'un séparateur, car à défaut de son utilisation l'accolement des lettres minuscules peut porter à confusion, tel que MaClassececi_est_une_methode. Les noms de classes ne comportant normalement pas de ce caractère de séparation et les noms de méthodes commençant par une lettre minuscule, il ne peut donc pas y avoir de conflits entre deux propriétés différentes.

Ajout d'un compteur Lorsque des cas de collision sont détectés, il est possible d'ajouter un compteur à la fin du nom généré pour différencier deux propriétés du même nom généré. Par exemple, si le nom traduit de deux propriétés est List_of_String , l'une peut être représentée par List_of_String_1 et l'autre par List_of_String_2 .

Cette utilisation va à l'encontre de l'avantage d'utilisation intuitive amenée par les types symétriques. Dans ce cas, le programmeur doit se fier à un outil pour résoudre le cas problème et connaître ce que représente chaque nom.

Interdire les cas de conflits Selon la forme d'utilisation des modules hybrides qui sépare les fichiers C par modules, seuls les cas de conflits à l'intérieur d'un même module sont à gérer. Dans cette optique, il est possible d'annoncer au programmeur les conflits à la compilation. La responsabilité de contourner ces conflits revient donc à la responsabilité du programmeur. En Nit, ces conflits se règlent par un module utilisateur en raffinant une classe existante pour lui ajouter une indirection.

Cette approche est un dernier recours si les conflits ne peuvent pas être évités d'une autre façon. Lors de conflits, elle forcerait le programmeur à utiliser une structure non intuitive pour arriver à ses fins, tel que définir une méthode uniquement pour contourner un cas problème. Ceci nuirait à la clarté du logiciel et complexifierait l'utilisation de l'interface native.

Caractères spéciaux et opérateurs Les opérateurs, tel que +, et ceux combinés à un nom, tel que x=, doivent prendre une certaine forme au travers de l'interface. Pour éviter les cas de conflits avec une méthode du même nom que celui traduit pour l'opérateur, encore une fois, nous pouvons sous baser sur le style des noms spécifié par le langage. Pour ce faire, dans les cas semblables à x= de la classe MaClasse, celui-ci peut-être traduit par MaClasse_x__equal ou encore par MaClasse__assign_x, et ainsi adopter un nom qui diffère du style du langage. Dans le cas de + de la classe MaClasse, il peut être traduit vers MaClasse__plus.

Fonctions spéciales Certaines fonctions natives générées ont des noms dérivés de la fonction symétrique normale, c'est le cas des appels à super, des fonctions d'implémentation de méthodes natives et des constructeurs.

Utilisons comme exemple, la méthode symétrique de la méthode MaClasse::ma_methode prend le nom MaClasse_ma_methode. Pour l'appel à super, la fonction générée prend le nom super_MaClasse_ma_methode ; pour l'implémentation d'une méthode native, il s'agit de impl_MaClasse_ma_methode et, en cas d'un constructeur, nous utilisons new_MaClasse_ma_methode.

Le nom spécial pour l'implémentation d'une méthode native est différent de celui de la méthode symétrique. Ceci permet d'utiliser la méthode native depuis le langage C comme toute autre méthode. Il est recommandé de toujours appeler la méthode via sa forme symétrique, car celle-ci passe par le système Nit et ainsi applique correctement le polymorphisme selon le receveur en plus d'assurer le fonctionnement des services de Nit, tel que le ramasse-miettes.

En se basant encore sur le style de nom utilisé par le langage Nit, où toute classe débute par une lettre majuscule, le placement de l'identifiant spécial devant le nom de la classe prévient tous les cas de collision de noms.

Nous avons retenu d'utiliser une combinaison d'un style précis de noms, de l'utilisation d'un simple caractère de soulignement comme séparateur entre la classe et la méthode pour le nom des méthodes symétriques en natif, du traitement des caractères spéciaux, des opérateurs et des types paramétrés ainsi que le placement des noms spéciaux en suffixe. La combinaison de tous ces moyens prévient entièrement les conflits de noms générés pour l'interface native de Nit. Les cas à problèmes surviennent uniquement si le programmeur ne respecte par le style de noms de Nit ou s'il importe le code généré à des endroits inappropriés. Les cas où les noms générés entrent en conflit avec des noms autres, tels que ceux des fonctions d'une bibliothèque native, sont très rares due à la complexité réelle des noms générés. Pour l'instant, l'interface native de Nit ne permet pas d'éviter ces conflits lorsqu'ils surviennent, il est de la responsabilité du programmeur de s'assurer que son logiciel Nit n'entre pas en conflit avec du code C existant.

4.5 Implémentation des méthodes symétriques et des méthodes natives dans le compilateur Nit

Dans cette section, nous présentons les détails d'implémentation des méthodes natives et des méthodes symétriques tel que nous les avons réalisés dans le compilateur Nit. Il est important de ne pas confondre ces deux sortes de méthodes, car elles ont un fonctionnement similaire, mais un but inverse. Les méthodes natives sont les méthodes Nit implémentées en C, alors que les méthodes symétriques sont la représentation en C des méthodes Nit. Ces deux types de méthodes ne s'appellent pas directement, chaque passage d'une méthode native à une méthode symétrique se doit d'être géré par le système Nit.

Le compilateur Nit génère du code C depuis le code Nit. Ce code généré est ensuite compilé par un compilateur C standard. Le code généré contient toutes les structures nécessaires au système Nit, y compris la pile d'appels de méthodes Nit. Les méthodes symétriques prennent la forme d'une interface standardisée vers le langage Nit depuis C, toutefois leur implémentation dans le compilateur se base sur ce code généré. L'invocation d'une méthode générique depuis le code C appelle en réalité une fonction intermédiaire qui vérifie la validité des arguments et extrait les objets Nit du type symétrique C. Puis, la fonction intermédiaire appelle la bonne méthode dans le système Nit. Cet appel de méthode manipule normalement la pile d'exécution et le polymorphisme.

Pour ce qui est de l'implémentation des méthodes natives, leur appel est géré par le système Nit. Au moment d'exécuter leur implémentation, le système invoque une fonction intermédiaire. Celle-ci gère de transformer les objets Nit en type symétrique C et appelle la fonction d'implémentation en C. À son retour, la valeur de retour est extraite du type symétrique C et retournée au système Nit.

Il est important de noter que toutes les fonctions symétriques sont générées spécifiquement pour chaque module hybride. Elles sont standard au langage Nit, mais indépendantes de son implémentation. De même pour les types symétriques qui ne donnent pas un accès

direct aux valeurs du système Nit. Ceci permet de proprement séparer le code C généré du code C implémenté par l'utilisateur. Tout passage du code C implémenté au code généré est vérifié pour qu'une erreur de programmation ne cause pas de problèmes à retardement dans le système Nit. Une utilisation bien intentionnée de l'interface pourra, au pire, ne causer que des erreurs à la compilation ou des erreurs dynamiques détectées par le système Nit. Un programmeur bien intentionné, suivant la spécification et utilisant les méthodes symétriques, ne peut pas déstabiliser le système Nit.

4.5.1 Méthodes natives

Les méthodes natives sont implémentées de façon à prendre la relève du système Nit immédiatement suite à l'appel de la méthode. Ceci permet de préserver l'application du polymorphisme par le système de Nit ainsi que de laisser le langage correctement mettre à jour la pile d'appels.

Une fonction intermédiaire servant de passage du langage Nit au langage C est alors invoquée. Cette fonction intermédiaire gère la conversion des types primitifs et appelle à son tour la fonction d'implémentation.

La fonction d'implémentation est représentée dans le code C soit par son identifiant spécifié explicitement, soit par la forme par défaut, laquelle consiste en un préfixe `impl_`, suivi du nom de la classe et du nom de la méthode native. À son exécution, cette fonction d'implémentation a accès aux arguments adaptés pour le langage C et peut réaliser toute opération normale en C avant de retourner une donnée, comme il est attendu pour la méthode native.

La valeur de retour est alors traitée par la fonction intermédiaire, s'il s'agit d'un type primitif celui-ci est converti en son équivalent en Nit. Cette fonction vérifie aussi la validité de l'objet à retourner dans le système Nit, par exemple en vérifiant les nullables et les types.

Finalement, si un objet est retourné, celui-ci est renvoyé au système Nit comme avec toute autre méthode Nit.

4.5.2 Méthodes symétriques

Les méthodes symétriques sont générées au besoin, selon la déclaration explicite de leur utilisation par une méthode native. Chacune de ses méthodes symétriques est représentée en C par une fonction attendant un receveur suivi des arguments de la méthode.

Cette fonction, agissant comme méthode symétrique, assure la transition du langage C au langage Nit. Elle est invoquée par le langage C lors de l'exécution d'une méthode native. Cette fonction se charge de convertir les types primitifs en objets Nit et elle assure la validité de tous les arguments. Par la suite, elle invoque la méthode Nit ciblée et en obtient le retour.

La valeur retournée est alors convertie dans sa forme symétrique pour le langage C et retournée par la fonction agissant comme méthode symétrique.

4.5.3 Combinaison de ces fonctions générées

L'utilisation combinée des méthodes natives et des méthodes symétriques permet l'appel de code C, puis le rappel vers Nit. Le nom spécial de l'implémentation évite la collision de nom avec le nom de la méthode symétrique, par exemple la méthode native MaClasse::a est implémentée par la fonction impl_MaClasse_a, mais appelable depuis le code C en tant que méthode symétrique avec MaClasse_a.

Au niveau de l'implémentation, les deux concepts s'appliquent successivement, mais séparément. La figure 4.3 présente un exemple de fonctionnement d'une utilisation combinée.

4.6 Gestion des exceptions

Le langage Nit n'offre pas de système d'exception au moment de l'écriture de ce document. Si cette fonctionnalité est ajoutée au langage, l'interface native devra offrir les services nécessaires pour y accéder depuis C. Ces services permettent au code C, ne

```
1 class MaClasse
2     fun a do b # appel b
3     fun b is extern import c # appel c
4     fun c do print "fin"
5 end
```

```
1 void impl_MaClasse_b( MaClasse recv ) {
2     MaClasse_c( recv ); /* appel a MaClasse::c */
3 }
```

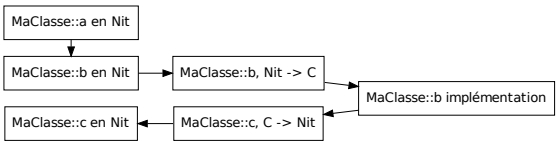

Figure 4.3: Exemple de flot d'appels dans le code généré pour une méthode native.

On observe dans le code Nit une classe MaClasse avec trois méthodes : a, b et c. La méthode a ne fait qu'appeler la b. Cette dernière est implémentée en C, mais son implémentation (présentée dans le code C) réalise un rappel vers le langage Nit pour la méthode c. Finalement, la méthode c ne fait qu'afficher le mot fin sur la sortie standard.

Le graphe représente le flot d'appels de fonctions générées ou implémentées par le programmeur. Chaque nœud constitue une de ces fonctions. Celles de gauche sont directement associées aux méthodes du langage Nit. La fonction sur la droite est une fonction d'implémentation de méthode native, celle-ci est implémentée par le programmeur.

gérant pas naturellement les exceptions, de les manipuler selon deux cas d'utilisation distincts.

La symétrie des méthodes permet l'invocation de méthodes depuis le code C. Cette exécution de code Nit peut lancer des exceptions remontant jusqu'à l'appel depuis C. Celui-ci doit alors pouvoir s'assurer que l'exécution s'est bien déroulée, et en cas inverse, savoir quelle est l'exception soulevée.

De plus, lors de l'implémentation d'une méthode native, le programmeur peut lancer une exception depuis le code C dans le but que celle-ci soit traitée par le code appelant.

Cette section présente les trois principales solutions que nous avons considérées pour ce problème : le retour de tout appel de méthodes Nit qui identifie la présence d'une erreur, l'utilisation d'une variable d'erreur globale et l'utilisation d'un récepteur d'erreurs.

Retour de succès ou d'erreur Il est commun parmi les fonctions système disponibles depuis le langage C que leur retour signale si une erreur est survenue. Une forme semblable peut être appliquée par l'interface native si chaque méthode Nit et chaque fonction de l'API retournent une valeur qui signale la présence et le type d'une exception à traiter. Tous ces appels doivent donc être surveillés pour ne pas ignorer une exception.

Cette approche va à l'encontre du système de symétrie des méthodes et nuit à l'expressivité des méthodes en les empêchant de retourner la valeur attendue directement. Pour contourner cette limitation, la valeur de retour devra être passée par référence via un argument.

Variable d'erreur Une autre forme de représentation des cas d'erreurs naturelles en C est l'utilisation d'une variable globale, comme errorno. Celle-ci permet de savoir si une erreur est à traiter et d'obtenir davantage d'informations à son sujet.

Une façon de reprendre ce concept est l'utilisation d'une variable accessible à l'aide de l'API. Celle-ci indique si une exception est en attente de traitement. Sous cette forme, il est nécessaire de vérifier le statut de cette variable à la suite de chaque appel de méthodes et de fonctions de l'API. Dans le cas où elle indique la présence d'une exception, il faudra la traiter et la marquer comme telle à l'aide d'autres fonctions de l'API.

Fonction réceptrice des erreurs L'utilisation d'une fonction native comme récepteur dynamique des erreurs soulevées en Nit amène une application semblable à celle du traitement des signaux POSIX. Ceci laisse le soin au programmeur de gérer les erreurs comme il le souhaite, par une variable globale ou par des arrêts brusques du logiciel.

La fonction réceptrice doit suivre une signature précise et être assignée comme réceptrice à l'aide d'une fonction de l'API.

Cette approche est lourde d'utilisation et peut rarement répondre par elle-même au problème. Elle force le programmeur à implémenter lui-même une solution supplémentaire. Ceci, dû au fait que la levée d'une exception influence généralement le flot d'exécution du logiciel, alors même si la fonction est invoquée lors d'une exception, elle doit utiliser un autre système pour influencer le flot du code C appelant.

Les exceptions n'étant pas spécifiées dans le langage Nit, aucune sélection précise ne peut être faite dans le cadre de la conception de cette interface. Toutefois, dans le but d'avoir une syntaxe intuitive et constante, l'option de la variable d'erreurs semble à privilégier.

4.7 Bibliothèque Nit utilisable en C

La réalisation d'une bibliothèque native en Nit est une possibilité intéressante, car elle permet la réutilisation de code Nit depuis des logiciels C indépendants. Ce sens d'utilisation n'est pas un des points centraux de cette recherche, nous l'explorons tout de même brièvement. Ce concept représente une utilisation inverse de l'interface native par rapport aux principaux cas d'utilisation de cette recherche.

Cette utilisation inverse de l'interface profite aussi de toutes les techniques développées dans cette étude. Combiné avec l'aisance d'utilisation de la symétrie des types et des méthodes, son emploi permet à un logiciel en C de profiter de façon naturelle de structures de données complexes réalisées en Nit. De même pour ce qui est des classes natives, qui prennent une forme entièrement naturelle en C.

Pour représenter un module Nit en tant que bibliothèque native, différentes formes ont été considérées : la bibliothèque native entière et l'enveloppe native.

Une bibliothèque native entière peut aisément être réalisée pour les langages compilés. Il est alors nécessaire d'ajouter des fonctions d'initialisation pour les services globaux de Nit. Ceux-ci peuvent être appelés automatiquement lors du chargement de la bibliothèque, ou encore, manuellement, pour donner un plus grand contrôle au programmeur ; par exemple, pour qu'il puisse charger et décharger les services au besoin. Une implémentation simple utilise comme point d'entrée possible toutes les méthodes visibles depuis l'extérieur d'un module.

La forme se basant sur une enveloppe native est plus commune aux langages ne se compilant pas en code natif, tel que Java [Gosling et al., 2005], Python [v. Rossum, 2009d] et Ruby [Thomas, Fowler et Hunt, 2004]. Elle implique tout de même la compilation séparée d'une enveloppe en C qui, elle, manipule le langage de haut niveau à l'arrière-plan. Cette enveloppe doit être modifiée et compilée pour chaque bibliothèque, car les fonctions doivent y être statiques.

Dans le cadre de cette étude, nous avons retenu la forme de bibliothèque native entière, car il s'agit de la forme la plus simple d'utilisation et elle concorde bien avec le compilateur du langage Nit. Avec ce dernier, cet ajout se fait simplement et naturellement.

L'implémentation réalisée pour prouver ce concept est incluse directement dans le compilateur. Elle intervient lors de la compilation d'un module Nit sans point d'entrée principal. Dans ce cas, au lieu de générer un exécutable indépendant, le compilateur crée une bibliothèque partagée standard. Cette bibliothèque comporte une fonction d'initialisation et une fonction de déchargement, ces fonctions sont invoquées automatiquement au chargement et au déchargement de la bibliothèque.

Cette implémentation, dans sa forme actuelle, est trop dépendante de la structure du code généré par le compilateur Nit. Il serait préférable d'attendre qu'une spécification pour les modules compilés séparément soit adoptée avant d'établir une forme précise pour les bibliothèques natives Nit.

4.8 Contribution et atteinte des objectifs

Les méthodes symétriques sont une autre contribution importante de notre recherche. Alors que plusieurs autres interfaces natives se fient à une manipulation dynamique des méthodes du langage de haut niveau en C, l'interface native de Nit simplifie leur manipulation et en permet une vérification dès la compilation. Leur manipulation est relativement simple, car dans plusieurs cas un appel de méthode Nit depuis C se fait en une seule ligne, alors qu'en Java un appel similaire se fait en minimum trois lignes de C.

Les méthodes symétriques aident à l'atteinte de certains objectifs de notre étude :

Syntaxe naturelle en C Les méthodes symétriques consistent en un ensemble de fonctions C normales, statiques et générées pour atteindre des méthodes Nit. Tout comme les types symétriques, les méthodes symétriques aident à préserver la syntaxe naturelle en langage C.

Sûreté à l'exécution du logiciel Les méthodes symétriques sont des fonctions statiques normales en C, leur utilisation est donc vérifiée statiquement à la compilation. Ce qui permet de relever les erreurs de type ou dans les identifiants dès la compilation et ainsi poursuivre l'objectif d'assurer la sûreté à l'exécution du logiciel.

CHAPITRE V

CLASSES NATIVES

À l'utilisation de l'interface native, le programmeur doit fréquemment conserver une référence à un espace mémoire ou une donnée C dans le code Nit. Traditionnellement, dans les autres interfaces, cet espace mémoire est opaque au langage de haut niveau et il y est représenté par un type général.

Dans ce chapitre, nous étudions la possibilité de rendre les références aux espaces mémoire C plus expressives. Pour ce faire, nous utilisons un concept général qui associe une classe Nit à un type C. On appelle cette catégorie de classes, les classes natives. Une instance d'une classe native a alors comme seul état sa valeur native.

Nous présentons d'abord la motivation à concevoir ces types, c'est-à-dire faciliter la réutilisation de code C. Ensuite nous explorons la forme que prend les classes natives, la possibilité de leur déclarer des constructeurs, leur destruction, leur grande catégorie de classe et les possibilités de spécialisation.

5.1 Représentation des types C en Nit

Un des principaux cas d'utilisation d'une interface native est la réutilisation de code C existant. Pour être utilisé dans le langage Nit, le code C est généralement enveloppé dans un module Nit. Ce module réalise la manipulation des fonctions et structures C d'origines tout en exposant une interface naturelle au langage Nit. Il doit alors également représenter les types C par une forme naturelle en Nit.

Le langage C se base généralement sur des structures de données bien définies. En général, celles-ci peuvent être référencées à l'aide d'un seul pointeur et sont utilisées par les différentes fonctions du code C.

Certains langages, tels que C#, utilisent comme seule référence à ces structures de données C un type de pointeur. En C#, ce type est le PtrInt et permet de conserver une adresse mémoire.

Pour un langage à objets, l'équivalent des structures de données C sont les objets. En plus d'encapsuler les données et d'associer des méthodes à des données, elles sont à la base de tous les avantages du paradigme objet.

Pour faciliter la gestion des données C en Nit, nous introduisons l'idée d'associer une classe Nit avec un type C. Nous qualifions cette catégorie de classes comme étant les classes natives et leurs instances des objets natifs. Les classes natives permettent de simplifier et naturaliser l'utilisation de structures C depuis Nit, ainsi que de profiter des avantages du paradigme objet sur ces classes natives.

5.2 Précision de l'équivalent C

L'un des grands avantages des classes natives est pour convertir ses instances au travers de l'interface native. Chaque classe native représentant un type C connu, ce type C peut agir en tant qu'équivalent en C (ou l'équivalent natif) pour la classe native.

Pour ce faire, le programmeur doit préciser l'équivalent de chaque classe. Dans cette section, nous discutons de différents endroits et façons qui peuvent être utilisés pour déclarer cet équivalent.

En Nit Le type C associé à une classe native peut être déclaré avec la classe Nit. Le type C équivalent est alors défini à l'aide d'une chaîne de caractères qui ne sera pas interprétée par le compilateur, mais uniquement utilisée comme type par l'interface native. L'utilisation d'une chaîne de caractères relativement opaque au langage Nit est inévitable pour s'assurer de conserver toute l'expressivité du langage C.

Cette approche se mêle bien aux autres caractéristiques d'une classe telles que les superclasses. C'est alors une syntaxe qui est naturelle pour le programmeur. Cette information supplémentaire peut aussi être réutilisée par un outil de génération de documentation.

Sous cette forme, l'équivalent C est connu par l'analyse du langage Nit. Il est alors possible de traduire ce type lors du passage vers C, de la même façon que le sont les types primitifs. Dans les cas de non nullables, en C, seul l'équivalent natif serait utilisé. Le code C est alors plus naturel et n'utilise pas de type Nit.

En C Nous avons considéré de préciser l'équivalent natif uniquement dans le code C. Pour ce faire, le type est défini dans le fichier d'en-tête C. Cette définition peut prendre la forme d'une macro substituant le nom de la classe Nit en un type C.

Cette approche a l'avantage de préserver la séparation des langages tout en conservant toute l'information dans les différents fichiers d'un même module. Elle poursuit également le concept de symétrie des types en manipulant, en C, les classes natives comme toute autre classe Nit.

Le type ne pouvant pas être connu par les analyses du code Nit, il ne peut pas être utilisé par le générateur d'échafaudage. Le code généré utilisera alors le nom de la classe tel que défini en Nit.

Selon les objectifs qui sont de préserver l'expressivité du langage Nit et une syntaxe naturelle en C, nous favorisons l'option de préciser le type C dans le code C. Celle-ci est naturelle pour un utilisateur avancé du langage C et elle permet également de représenter toutes formes de données C utilisant un pointeur.

5.3 Constructeurs natifs

Le concept des classes natives vise à permettre leur manipulation comme toute autre classe. Nous jugeons qu'il est approprié d'y offrir des fonctionnalités pour faciliter leur création, idéalement sous forme de constructeurs. Dans cette section, nous discutons des différentes implications qu'amène l'ajout de constructeurs aux classes natives.

Méthode indirecte À défaut de se servir de constructeurs natifs, le programmeur doit définir une méthode native sur un autre objet pour que celui-ci retourne une nouvelle instance de la classe native, un objet natif. Ceci nécessite d'implémenter la logique de création d'un objet à l'extérieur de la classe à construire.

Dans certains cas, cette approche est naturelle et appropriée, par exemple lors d'une application du patron de conception *Factory*. Mais selon le paradigme objet, cette approche est inhabituelle et ne peut être attendue de tous les cas d'utilisation.

Constructeurs natifs Cette approche consiste à permettre la définition de constructeurs implémentés nativement. Ceux-ci ont accès à toutes les fonctionnalités offertes aux méthodes natives, ces dernières sont également déclarées et implémentées de façon similaire.

L'objet natif serait donc à créer entièrement dans le code C. Le constructeur aurait tout de même la possibilité d'effectuer un rappel vers Nit, au besoin. La fonction native implémentant ce constructeur a l'entière responsabilité de réserver l'espace mémoire et de retourner un pointeur valide.

Constructeurs normaux Cette approche est semblable à la précédente, mais permet également d'y définir des constructeurs normaux, lesquels peuvent réaliser la construction de l'objet en C via une méthode ou un autre constructeur. Cette option offre davantage de possibilités et amène un comportement naturel en Nit.

Ceci est particulièrement pratique si la spécialisation entre les classes natives est permise. Elle donne alors la possibilité à une sous-classe d'appeler le constructeur d'une classe parent.

Un constructeur normal doit invoquer un constructeur natif qui réserve l'espace mémoire et retourne l'adresse de celui-ci qui est à associer à l'objet créé.

Nous avons retenu d'utiliser des constructeurs normaux et des natifs. Offrir ces deux possibilités au programmeur lui permet de se servir de l'interface de façon plus rapide et efficace.

Toutefois, au moment de l'écriture, nous n'avons implémenté seulement le format de constructeurs natifs. Les constructeurs normaux sont moins aisément compatibles avec le système utilisé par le compilateur Nit, et davantage de travail est nécessaire pour en assurer un bon fonctionnement dans tous les cas d'utilisation.

5.4 Invocation du destructeur des classes natives

Les classes natives permettent d'associer une classe Nit à un type C. Généralement, ce type est un pointeur vers une structure de données plus complexe. La mémoire occupée par cette structure doit être gérée et libérée d'une façon ou d'une autre.

Nous discutions précédemment de la possibilité d'utiliser des constructeurs natifs pour gérer la création de cet espace mémoire de façon naturelle depuis Nit. Toutefois, il n'y a aucune garantie qu'un tel espace mémoire a bel et bien été alloué durant l'invocation de ce constructeur. L'objet natif ne peut être qu'un pointeur avec une valeur nulle. Cette approche est tout de même à la discrétion du programmeur. Langage Nit considère tout objet natif créé par un constructeur natif comme étant valide.

Dans tous les cas, lors d'une utilisation pertinente ou valide d'une classe native, si un espace mémoire est réservé celui-ci se doit d'être libéré à un moment où il est encore référencé pour éviter les fuites mémorielles.

Cette section présente les différentes approches que nous avons considérées pour assurer une libération fiable de la mémoire réservée.

Confier la gestion de la mémoire native au programmeur Considérant toutes les difficultés de la gestion de la mémoire lorsqu'elle est gérée par le langage, nous avons considéré de ne pas l'automatiser, d'en laisser la responsabilité au programmeur. Ceci permet de conserver l'intégralité de la logique de la gestion native du côté C et donne au programmeur la possibilité de gérer manuellement et précisément tous les cas de libération de mémoire.

Lors des cas communs, le programmeur conçoit lui-même les méthodes de libération de la mémoire native et s'assure de l'invoquer alors que l'objet natif est encore référencé depuis le langage Nit.

Automatiser l'appel d'un destructeur obligatoire Nous considérons la possibilité de forcer le programmeur à implémenter une méthode de libération pour toute classe native et de l'invoquer automatiquement. Une façon de faire est de définir

une méthode abstraite dans la classe générale à toutes les classes natives. Cette méthode pourrait ensuite être appelée par le ramasse-miettes lors de la libération de l'objet en Nit ou même, lors de toutes sorties de contexte.

La méthode de destruction implémentée par le programmeur serait responsable de déterminer adéquatement si une libération de mémoire est nécessaire et de la réaliser. Pour une utilisation facile de cette forme, le ramasse-miettes doit garder la trace des objets natifs même après que ceux-ci ne soient plus référencés par le logiciel. Ceci permettrait d'appeler le destructeur au bon moment selon le système du langage Nit.

Une implémentation trop dépendante du comportement du ramasse-miettes est limitative autant pour son évolution future que pour la possibilité de réaliser une libération automatique fiable.

La possibilité de libérer automatiquement l'objet lorsqu'il n'est plus référé serait trop dépendante de l'implémentation du ramasse-miette et coûteuse à l'exécution.

Étendre le langage Nit pour inclure le cycle de vie des objets natifs Une approche présente dans les langages C# et Go, est d'étendre la syntaxe du langage Nit pour faciliter la précision du cycle de vie d'un objet natif. Ceci reprend donc l'approche précédente, mais l'utilisateur conserve le contrôle sur la durée de vie de l'objet et ainsi le moment de sa libération.

En C# [ECMA, 2006], cette approche prend la forme de blocs marquant l'utilisation d'un objet natif. Un exemple en ce langage est présenté par la figure 5.1.

Alternativement, dans le langage Go [The Go Programming Language Specification, 2011; Command cgo, 2011], il est possible de spécifier, dès la création de l'objet, une méthode à invoquer en fin de contexte. Il s'agit alors principalement de sucre syntaxique qui pourrait être réalisé autrement. Un exemple avec cette forme est présenté par la figure 5.2.

Cette approche est facile dans les cas simples, mais elle ne s'applique pas à tous les cas. Par exemple, dans le cas où un objet natif est utilisé tout au long de la vie d'un logiciel, tel qu'une ressource globale.

```
1 using ( var image = Display.LoadImage( "foo.png" ) )
2 {
3     Display.draw( image );
4 } // appel implicite de image.dispose();
```

Figure 5.1: Exemple de précision du cycle de vie des objets natifs en C#.
La variable image est instanciée en début du bloc couvrant la ligne 1 à 4 et son cycle de vie est délimité par ce bloc. Elle est utilisée par la méthode draw avant d'être automatiquement libérée en fin de bloc.

```
1 func Print(s string) {
2     s_natif := C.CString( s )
3     defer C.free( unsafe.Pointer(s_natif) )
4     C.fputs( s_natif, (*C.FILE)(C.stdout) )
5 }
```

Figure 5.2: Exemple de précision du cycle de vie des objets natifs en Go.
La variable s_natif sert à représenter une chaîne de caractères compatibles avec le langage C. Cette variable est assignée par le retour de l'appel à C.CString(s) et immédiatement marqué comme étant à libérer en fin de bloc par un appel à defer C.free (...) . Par la suite, dans le code, la variable est utilisée normalement et elle sera libérée automatiquement.

Définir une classe supplémentaire qui est libérable Une approche permettant de rallier différentes approches est de définir une classe abstraite avec, comme unique propriété, une méthode de libération. Chaque classe native serait alors libre de spécialiser cette classe, ce qui encouragerait le programmeur à implémenter correctement cette méthode et l'encouragerait également à l'invoquer au moment approprié.

Cette approche se combine bien avec celle du contrôle du cycle de vie, alors que n'importe quelle sous-classe de la classe libérable serait utilisable dans la structure syntaxique. Elle permet également d'élargir l'utilisation de cette structure à toutes les classes, pas seulement les natives.

Nous avons retenu l'approche consistant à offrir une classe libérable, elle pourra éventuellement être combinée avec l'approche qui permet d'étendre le langage Nit pour y spécifier la durée de vie des objets. Cette dernière offre un aide intéressant au programmeur, mais elle n'est applicable que dans quelques cas pratiques.

Cette option sous-entend que la majeure partie de la gestion de la mémoire native est laissée entre les mains du programmeur, lui seul ayant les connaissances suffisantes pour assurer cette bonne gestion.

5.5 Grande catégorie des classes natives

Certains langages à objet distinguent différentes grandes catégories de classes. Il est commun qu'ils distinguent les interfaces des classes normales, les interfaces ne comportant pas d'état dû à leur absence de variables d'instances. De plus, certains langages distinguent les types primitifs des classes normales.

Dans le langage Nit, un concept en développement est celui des types énumérations. Ceux-ci ont un comportement semblable à celui des types primitifs, leur état dépend uniquement d'une valeur et ne comportent pas de champs. Ils peuvent se spécialiser entre eux et spécialiser des interfaces. Ils offrent des fonctionnalités supplémentaires

```
1 extern A
2     redef fun to_s do return "ceci est une classe native"
3 end
```

Figure 5.3: Exemple de déclaration d'une classe native.
Ce segment de code Nit déclare la classe A comme étant une classe native à l'aide du mot clé extern.

pour gérer les cas d'énumération de valeur et pour appliquer des opérations d'ensemble sur différentes énumérations.

Nous considérons les classes natives comme étant de type semblable avec l'énumération. Leur état dépend entièrement de leur valeur native, laquelle est normalement représentée par un pointeur. Il est donc impossible d'y déclarer des variables d'instance et tout changement d'état doit passer par du code C qui manipule la valeur native associée. Par contre, les classes natives ne peuvent pas être catégorisées en tant qu'énumérations puisque le concept de fond des énumérations est fondamentalement différent.

Nous proposons une grande catégorie spécifique aux classes natives. Nous profitions de cette catégorie pour y établir des règles précises de spécialisation et pour y limiter les champs.

Cette grande catégorie ne permet pas la déclaration de champs parmi ses sous-classes, tout comme les énumérations. Pour les cas plus complexes où un objet doit comporter des données natives et des variables Nit, le programmeur peut employer des classes normales composées de variables normales et de variables de types natifs.

La déclaration d'une classe native prend alors une forme similaire à celle de toute autre classe, à l'exception que le mot clé extern est utilisé comme type de classe. Un exemple de déclaration est présenté par la figure 5.3.

5.6 Spécialisation des classes natives

La spécialisation de toutes classes est une partie importante du paradigme de programmation à objets. Pour en conserver les forces, cette fonctionnalité doit être offerte sous une forme ou une autre avec les classes natives.

Pour ce faire, nous permettons la spécialisation entre les classes natives, mais devons y appliquer certaines restrictions. Celles-ci servent à éviter des cas de conflits entre les équivalents C de chacune des classes se spécialisant. En fait, l'état d'une classe native se définit entièrement par sa valeur native et son équivalent C est utilisé pour déterminer le format de sa conversion vers le code C. Un équivalent C incompatible entre deux classes de la même hiérarchie de spécialisation entraîne des comportements inattendus et de possibles corruptions de mémoire.

Le changement d'équivalent C entre les classes d'une même hiérarchie de spécialisation peut toutefois être utilisé à bon escient. Plusieurs structures de données natives emploient des stratégies pour simuler une spécialisation ; par exemple, le type union permet d'utiliser un même type dans différents buts. Ou encore, différentes structures peuvent commencer par les mêmes types de données pour y stocker leurs caractéristiques communes dans un format pouvant être utilisé par différentes fonctions.

La forme de ces stratégies varie selon le logiciel et le programmeur. Elles sont alors imprévisibles et peuvent difficilement être standardisées par l'interface native. Leur utilisation reste pertinente et lorsque couplée avec l'héritage de classes natives, elle permet de factoriser du code et d'associer efficacement deux concepts semblables de différents langages.

Cette situation soulève différents problèmes et dangers alors nous avons considéré plusieurs approches de la spécialisation des classes natives. Ci-suivent certaines des approches les plus intéressantes, dont quelques-unes touchent même à la spécialisation avec d'autres types de classes.

Conserver le type C de la classe native la plus générale Dans le but de prévenir entièrement les conflits entre les équivalents C dans une même hiérarchie de spécialisation, nous avons considéré que les classes natives au sommet de leur hiérarchie définissent un équivalent C. Toutes les autres classes natives de la hiérarchie sont alors définies par le même équivalent C que la classe au sommet de la hiérarchie.

Cette forme limite la spécialisation multiple de classes natives, sans limiter la spécialisation multiple d'interfaces. Une classe native ne peut avoir qu'un équivalent C, il ne peut provenir que d'une classe native qui est au sommet de la hiérarchie.

Cette approche est aussi limitative pour les cas où le programmeur se sert d'un type C compatible avec le concept de spécialisation. Il doit alors manuellement réaliser les changements de types sur les valeurs natives converties par l'interface.

Cette approche vise à assurer une bonne sûreté d'exécution du logiciel, mais n'empêche aucunement le programmeur de réaliser des manipulations dangereuses. Elle limite donc l'expressivité et n'amène qu'un faible avantage en sûreté.

Utiliser des attributs natifs au lieu de classes natives Une alternative considérée dans cette recherche est la possibilité d'opter pour des attributs natifs au lieu de classes natives. Ceux-ci seraient définis dans une classe Nit normale et seraient traités comme tout autre attribut.

Il serait possible d'étendre la syntaxe du langage Nit pour permettre d'y définir le type C équivalent dès la déclaration de l'attribut, mélangeant les langages, mais permettant ainsi d'indiquer la transformation automatique à appliquer.

Cette approche ne répond pas au cas simple de conserver l'état d'un objet Nit dans sa valeur native. Elle nécessite également des fonctions d'indirection dans plusieurs cas. Par exemple, lorsqu'une classe est associée directement à une valeur native et que ses méthodes natives agissent directement sur cette valeur, une méthode publique doit être offerte pour invoquer la méthode native à l'aide de la variable locale. Cette démarche nécessite une plus grande quantité de travail de la part du programmeur et elle fait perdre le concept à la base même des classes natives.

Ceci peut être simulé à l'aide du concept des classes natives tel que défini dans cette grande section. L'utilisateur peut définir des classes natives de support et s'en servir comme types pour les attributs d'un objet principal.

Autoriser la spécialisation lorsque le format des données natives est compatible

Dans le but de limiter minimalement les possibilités pour spécialiser une classe native, nous avons considéré de permettre la spécialisation (même multiple) par toute classe native. Ces spécialisations sont possibles si tous les équivalents C d'une même hiérarchie sont compatibles. Ceci laisse place à des erreurs indétectables de la part du programmeur qui pourrait utiliser deux équivalents C incompatibles. Par contre, ces restrictions étant limitées aux classes natives uniquement, le programmeur est conscient de ce danger dès qu'il amorce une spécialisation de classes natives.

Assurer l'utilisation de types compatibles peut être difficile lorsque le programmeur n'a pas accès à l'implémentation des classes plus générales, il peut alors difficilement connaître les équivalents C compatibles. Selon où est déclaré l'équivalent C (voir 5.2), l'outil générateur de documentation peut aider. Si l'équivalent est précisé dans le code Nit, l'outil peut l'extraire et l'ajouter à la documentation. Dans le cas où l'équivalent est précisé en code C, il peut difficilement être automatiquement extrait sans ajout de syntaxe supplémentaire.

Encapsuler plus d'une valeur native dans un même objet Nous avons considéré d'autoriser toute spécialisation de classes natives et de stocker à l'interne les différentes valeurs natives qui y sont associées. Au moment de traverser la frontière, une valeur serait sélectionnée comme équivalence selon le type attendu par la méthode native. Les méthodes de chacune des classes natives d'un même objet agiraient alors sur différentes valeurs natives.

Cette approche fait perdre le concept où l'état d'un objet C est défini par sa valeur native. Il serait alors défini partiellement en Nit et par plusieurs valeurs natives, ce qui n'est pas à négliger et qui peut même être pratique pour glisser des classes natives dans de grandes hiérarchies de spécialisation.

Toutefois, ceci ne règle pas les cas où une classe native est spécialisée pour agir sur sa valeur native. Une sous-classe n'aurait alors aucune façon d'introduire une nouvelle opération sur la valeur native d'une classe qu'elle spécialise.

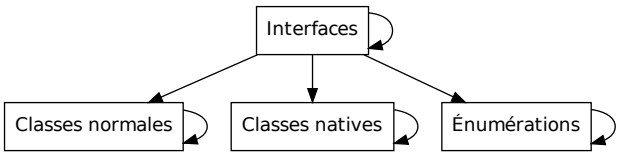

Figure 5.4: Spécialisation autorisée entre les différentes grandes catégories de classes.
La spécialisation possible entre les quatre grandes catégories de classes est identifiée par un flèche depuis la classe plus générale pointant vers la classe pouvant la spécialiser.

Nous avons retenu l'approche permettant toutes formes de spécialisation entre les classes natives et laissant la responsabilité au programmeur d'éviter les conflits. Cette approche permet de conserver le concept de base des classes natives dont l'état est défini par leur valeur native, elle amène une utilisation simple et préserve ce concept comme étant une aide au programmeur. Ce dernier est donc responsable de l'utiliser lorsqu'approprié et de contourner manuellement ses limitations lors de cas plus complexes. La figure 5.4 illustre les possibilités de spécialisation des grandes catégories de classes en Nit suite à l'ajout des classes natives.

5.7 Contribution et atteinte des objectifs

La forme que prennent les classes natives est une de nos contributions importantes. D'autres langages, dont C#, offrent déjà des types C généraux pour conserver une référence à des données C depuis le langage de haut niveau. Notre interface native est la première à permettre d'associer un type Nit à un type C précis. De plus, l'application du polymorphisme à ces types est également nouvelle.

Les classes natives aident à l'atteinte de différents objectifs de notre étude :

Utiliser le langage Nit de façon expressive Les classes natives utilisent le langage Nit de façon expressive en représentant les types C sous une forme naturelle et précise en Nit. Elles se comportent principalement comme les autres classes Nit, permettant ainsi de leur définir des superclasses et des méthodes avec la syntaxe expressive normale au langage Nit.

Syntaxe naturelle en C Les classes natives sont associées directement à un type C, ce qui permet de conserver une syntaxe naturelle en C. Les paramètres typés par une classe native sont représentés par le type C qui y est associé dans le langage C. Le code C utilise donc directement que le type C.

Sûreté à l'exécution du logiciel Le typage précis offert par les classes natives en Nit assure une sureté à l'exécution du logiciel en permettant une vérification statique de l'utilisation leur utilisation. La manipulation de ces types est vérifiée statiquement par le compilateur Nit dans le code Nit et par le compilateur C dans le code C.

CHAPITRE VI

VALIDATION ET UTILISATION DE L'INTERFACE NATIVE DE NIT

Dans ce chapitre, nous discutons du processus d'utilisation de l'interface native de Nit, des autres interfaces qu'elle a servi à développer, des modules qu'elle nous a permis de développer, puis nous approfondissons des logiciels complets qui utilisent l'interface directement ou indirectement. Nous finalisons en exposant des observations personnelles issues de notre utilisation de l'interface.

6.1 Autres interfaces développées

Tel que discuté au moment du choix de la forme du module hybride, cette forme peut servir de base au développement d'interfaces spécialisées. Dans le cadre de cette étude, nous avons réalisé deux autres interfaces natives d'une forme différente. Il s'agit d'un module de chargement dynamique de bibliothèque et d'un outil permettant l'utilisation de langages imbriqués.

6.1.1 Module de chargement dynamique de bibliothèques natives

Nous avons réalisé un module Nit pour donner accès aux services de chargement dynamique de bibliothèques partagées offerts par le système d'exploitation. Ce module permet de charger dynamiquement une bibliothèque native depuis le langage Nit et d'y trouver des fonctions. Elle est particulièrement bien adaptée aux cas d'utilisation des greffons et du prototypage.

L'implémentation de cette interface dynamique est réalisée par module hybride. Chaque fonction native prend alors la forme d'un objet sur lequel une méthode exécutant l'appel est à invoquer.

Nous avons utilisé différentes parties de notre interface native de base pour réaliser ce module. Le module lui-même est hybride et consiste en moins de 500 lignes de code Nit et C. Nous utilisons les méthodes natives pour charger les bibliothèques natives et pour faire appel aux fonctions. Ces méthodes retournent des instances de classes natives que nous utilisons pour représenter les références aux bibliothèques natives et aux fonctions natives en Nit. Nous gérons manuellement le polymorphisme de Nit en C pour extraire les types primitifs lors des appels aux fonctions natives chargées dynamiquement. Ces classes natives et manipulations des types sont une complexité nécessaire pour intégrer ces fonctions natives au langage Nit. Il s'agit d'un langage statique, il nous est alors impossible d'accéder dynamiquement à des fonctions de la même façon que pour les méthodes normales du langage.

Un exemple d'utilisation de ce module est présenté à la figure 6.1. Le code source de ce module est publié avec la bibliothèque standard sur le dépôt de l'interface native implémentée pour Nit. [1]

L'implémentation native de ce module est complexe et nous avons utilisé le préprocesseur du compilateur C pour factoriser le code. De plus, nous avons employé des astuces pour contourner les erreurs de types lors des manipulations de pointeurs de fonctions. Ces techniques, normales avec le langage C, sont un bon exemple de code C difficilement analysable par un outil, mais qui fonctionne correctement avec l'interface native de Nit.

6.1.2 Implémentation imbriquée de méthodes natives

Nous avons développé une interface qui étend le langage Nit pour imbriquer du code C parmi le code Nit et ce, à quatre endroits. Une méthode native peut être implémentée sous sa déclaration dans un même fichier. L'équivalent natif d'une classe native peut

1. Le code source du module *dl* en Nit est disponible à `http://xymus.net/nitni/dl`.

```
1  import dl
2
3  var lib = new DynamicLibrary.open( "libfib.so" )
4                  # charge la bibliotheque
5
6  if not lib.is_loaded then
7      var error = lib.error
8      if error != null then
9          print "DL error: {error}"
10     else
11         print "unknown DL error has occured"
12 else
13     var fun_fib = lib.sym_int( "Fibonacci" ) # trouve la fonc.
14     var r = fun_fib.call( 12 ) # invoque la fonction
15
16     lib.close # decharge la bibliotheque
17 end
```

Figure 6.1: Exemple de chargement dynamique de bibliothèques natives en Nit.
Le module *dl* est utilisé pour charger la bibliothèque native par un appel au constructeur open de la classe DynamicLibrary et retourne ainsi la bibliothèque sous forme d'un objet. Suite à la vérification de succès du chargement, une fonction native retournant un entier est retrouvée par un appel à la méthode sym_int sur la bibliothèque. Cet appel retourne alors une référence à la fonction sous la forme d'un objet de type IntFunction. Un appel à la méthode call sur la référence invoque la fonction native et retourne le type attendu. Dans ce cas, uniquement le type de retour est connu, car il est précisé au chargement de la bibliothèque. Le nombre et les types des arguments sont entièrement dynamiques, aucune vérification statique n'est effectuée.

être précisé suite à la déclaration de la classe. Du code C peut être glissé en en-tête de fichier pour être utilisé dans le corps natif du module. De même que du code peut être inséré parmi les importations de modules pour préciser des include à réaliser depuis l'en-tête natif.

Cette interface alternative diffère de la principale par le fait qu'un module hybride y est implémenté dans un seul fichier au lieu de plusieurs. Un exemple de son utilisation est présenté par la figure 6.2.

Cette interface est spécialisée pour le cas d'optimisation de méthodes, mais simplifie grandement tous les cas d'utilisation pour un programmeur avancé. Le programmeur n'a alors qu'à remplacer le corps d'une méthode normale par un corps en C. Le principal désavantage de cette forme par rapport à l'interface native principale est la perte de la documentation d'aide générée par l'outil générateur d'échafaudages. Le programmeur doit alors deviner la signature des méthodes et des types symétriques. Cette faiblesse pourrait toutefois être corrigée à l'aide d'un IDE adapté, lequel afficherait dynamiquement les signatures natives.

Au moment de l'écriture de ce document, cette interface en est au stade de preuve de concept, mais elle est entièrement fonctionnelle. Elle prend la forme d'un outil qui génère le code pour l'interface native principale. Elle intervient à la génération d'échafaudages et utilise le code C pour combler le corps des méthodes natives ; le compilateur doit être lancé manuellement par la suite. Elle permet au programmeur d'aisément modifier le code Nit d'un module hybride en ajoutant ou en retirant des méthodes natives, et ce, sans avoir à réorganiser manuellement le code C.

6.2 Processus d'utilisation de l'interface

Le programmeur doit utiliser l'interface native de Nit en plusieurs étapes. Dans cette section, nous exposons les étapes selon deux processus d'utilisation. Les processus varient selon les connaissances du programmeur.

Pour les débutants ou les utilisateurs occasionnels, l'emploi du générateur d'échafaudages est fortement encouragé. Voici donc le processus recommandé :

```
1  module mon_module
2
3  import `{ #include <stdio.h>`}
4
5  class MaClasse
6      fun travail( a : Int, b : Float, c : String ) : MaClasse
7              is extern import to_s, String::to_cstring `{
8          char *c_natif;
9          c_natif = String_to_cstring( c );
10         printf( "depuis: %s: %s %f %i",
11                 String_to_cstring( MaClasse_to_s( recv ) ),
12                 c_natif, b, a );
13         return recv;
14     `}
15 end
16
17 var a = new MaClasse
18 a.travail( 123, 5.67, "foo" )
```

Figure 6.2: Exemple de langages imbriqués en Nit.

Cette utilisation démontre un cas simple d'implémentation d'une méthode native avec l'interface native imbriquée. Le module mon_module précise d'abord les importations nécessaires au code C. La méthode native travail prend en argument un entier, un point flottant ainsi qu'une chaîne de caractères et elle retourne une instance de la classe MaClasse. La méthode définit également son flot d'appels, dans ce cas, elle appellera uniquement la méthode locale to_s retournant une chaîne de caractère et la méthode to_cstring de la classe String qui permet d'obtenir le type primitif de la chaîne de caractère se terminant par une valeur nulle.

Suite à la signature de la méthode, l'implémentation en C est contenue dans le bloc délimité par `{ et `}. La symétrie des types est appliquée sur les arguments de la méthode native, l'entier et le point flottant sont alors convertis automatiquement en leurs formes natives. Le receveur implicite, nommé recv, et la chaîne de caractères sont représentés par des types opaques.

1. Créer la structure du module hybride en langage Nit dans un fichier Nit normal. Dans ce fichier, préciser les signatures complètes des méthodes natives, les méthodes appelées depuis le code natif et les classes natives.
2. Générer les échafaudages des fichiers C en exécutant l'outil d'échafaudage sur le fichier Nit.
3. Implémenter le corps des méthodes natives dans les ficher C nouvellement créé.
4. Au besoin, préciser l'équivalent natif des classes natives dans l'entête C.
5. En cas de correction, modifier le code Nit, relancer le générateur d'échafaudages, et extraire les changements du code généré. Le générateur d'échafaudage peut remplacer les fichiers C existants ou les créer séparément.
6. Compiler le programme complet par un appel normal au compilateur Nit.

Pour un programmeur avancé ou expert qui connaît les principes de traduction des noms de méthodes et de types, il est possible de ne pas utiliser le générateur d'échafaudages. Dans ce cas, le programmeur peut implémenter les fichiers Nit et C parallèlement pour finalement compiler le tout :

1. Créer la structure du module hybride en langage Nit dans un fichier Nit normal. Dans ce fichier, préciser les signatures complètes des méthodes natives, les méthodes appelées depuis le code natif et les classes natives.
2. Implémenter le corps des méthodes natives dans les ficher C nouvellement créé.
3. Au besoin, préciser l'équivalent natif des classes natives dans l'entête C.
4. En cas de correction, les apporter directement dans chaque fichier.
5. Compiler le programme complet par un appel normal au compilateur Nit.

6.3 Modules se basant sur l'interface

Dans le but de tester et valider la proposition et son implémentation dans le compilateur, nous avons réalisé plusieurs modules et logiciels à l'aide des interfaces natives développées. Ces modules et logiciels nous ont servi à évaluer l'interface native de Nit selon différents cas d'utilisation, à en assurer sa validité. Dans cette section, nous présentons les détails intéressants de ces utilisations pratiques et quels aspects de l'interface native chacune de ces utilisations a mis à l'épreuve.

6.3.1 Module de chiffrement MD5

Pour répondre à un besoin spontané, nous avons réalisé un module, nommé *md5*, implémentant la fonction de chiffrement MD5 [Rivest, 1992]. Cette fonction est unidirectionnelle et sa plus simple utilisation implique l'implémentation d'une seule fonction. Le code source de ce module est publié avec la bibliothèque standard sur le dépôt de l'interface native de Nit. [2]

Nous avons implémenté cet algorithme entièrement dans le code source d'un module hybride. Aucun appel ou lien à une bibliothèque native n'est nécessaire. Pour ce faire, en langage Nit, nous redéfinissons la classe des chaînes de caractères pour y ajouter une méthode native nommée `md5`. Cette méthode est implémentée en C à l'aide de l'algorithme publié en tant que logiciel libre par L. Peter Deutsch.

Ce module représente le cas d'utilisation de l'interface native qui est d'optimiser une méthode. Nous observons cette optimisation par l'implémentation en C d'une méthode Nit qui nécessite un grand traitement mathématique.

Ce module démontre également que notre interface atteint les objectifs de préserver une utilisation expressive du langage Nit et une utilisation naturelle du langage C. Le code Nit est concis et permet toujours d'appliquer des fonctionnalités avancées du langage tel que le raffinement de classe pour ajouter notre méthode native à une classe existante. De plus, nous avons pu réutiliser du code C existant directement dans notre interface, permettant donc toute utilisation normale du langage C.

6.3.2 Module de communication réseau

Nous avons implémenté un module Nit offrant des services de communication réseau à l'aide de l'interface native et des fonctions système reliées aux *socket* tel que définit par le standard POSIX [Stevens, Fenner et Rudoff, 2003].

---

2. Le code source du module MD5 en Nit est disponible à `http://xymus.net/nitni/md5`.

Ce module vise à être de haut niveau, exposant peu des détails des fonctions système. Par contre, il offre des classes simples d'utilisation et d'autres qui spécialisent les classes appropriées du module *stream* de la bibliothèque standard Nit ; standardisant ainsi les communications réseau avec les autres flux de données. Le code source de ce module est publié avec la bibliothèque standard sur le dépôt de l'interface native implémentée pour Nit. [3]

Les classes natives qui représentent un *socket* réseau sont précisées comme équivalant un pointeur vers un descripteur de fichier dans l'en-tête natif du module hybride. Les méthodes de lecture et d'écriture des classes de flux sont définies en Nit, mais appellent les méthodes des classes natives. Ces dernières méthodes sont implémentées en C et réalisent la majorité du traitement nécessaire à la lecture et à la vérification des erreurs.

Cette utilisation de l'interface native camoufle des appels système de bas niveau à l'aide du paradigme de programmation à objets et de classes natives. Cette représentation spécialise des classes existantes et ainsi peut être utilisée directement par tout logiciel Nit.

Ce module est un exemple du cas d'utilisation d'enveloppe d'une bibliothèque de fonctions. En réalité, ce module enveloppe des fonctions système pour leur donner une forme naturelle au langage Nit. L'interface native de Nit nous a permis d'implémenter efficacement cette enveloppe ainsi que d'intégrer les fonctions système dans les services de la bibliothèque standard du langage.

6.3.3 Module graphique avec SDL

La bibliothèque de fonctions Simple DirectMedia Layer [Hall, 2001], ou simplement SDL, offre différentes fonctions pour gérer le multimédia sur plusieurs plateformes. Plusieurs enveloppes sont disponibles et rendent cette bibliothèque compatible avec 17 langages de programmation, sans compter Nit. [4]

---

3. Le code source du module *sockets* en Nit est disponible à `http://xymus.net/nitni/sockets`.

4. Pour plus d'informations sur le projet Simple DirectMedia Layer, voir `http://www.sdl.org`.

Nous avons réalisé un module Nit nommé *sdl* qui prend la forme d'une enveloppe autour de la bibliothèque SDL et en offre une forme naturelle au langage Nit. Cette utilisation de l'interface native profite principalement du concept des classes natives. Le code source de ce module est publié parmi le dépôt de code du projet de simulateur de forêt, ce projet est discuté plus en détail à la section 6.4.2. [5]

Nous avons implémenté les types principaux de la bibliothèque SDL sous forme de classes natives. Ce qui permet d'associer directement une classe Nit à un type C existant de la bibliothèque. Par exemple, une fois en C, les classes SDLDisplay et Image se traduisent en SDL_Surface* et la classe Rectangle se traduit en SDL_Rect*.

Ces classes sont accompagnées de constructeurs natifs pour en permettre une utilisation plus naturelle en Nit. Par exemple, pour charger une image depuis un fichier, le programmeur utilise l'expression new Image.from_file(path).

La majorité des méthodes du module ont été implémentées en C. Il en est de même pour la méthode de copie d'une image à l'écran ou sur une autre surface, blit , ainsi que pour les méthodes de libération de la mémoire réservée. Toutefois, certaines méthodes, telles que la copie centrée d'une image, sont implémentées partiellement en Nit et appellent la méthode native blit .

Cette implémentation réalise le cas d'utilisation d'enveloppe de bibliothèque en se servant des classes natives, des méthodes implémentées nativement et des constructeurs natifs. Cette enveloppe est très mince et la majorité des méthodes natives sont un appel direct vers une fonction de la bibliothèque. Elle fut réalisée en environ 170 lignes de code Nit et 520 lignes de code C.

---

5. Le code source du module SDL en Nit est disponible à `https://github.com/xymus/nit-forest-sim/tree/master/src/sdl`.

6.3.4 Module pour traiter les signaux ANSI C

En ANSI C, l'en-tête *signal.h* définit différentes fonctions système pour traiter les signaux envoyés à un processus [Stevens, Rago et Ritchie, 1992]. Ceci permet, entre autres, de réagir dans le cas d'une interruption clavier, de la réception d'un signal d'arrêt ou du soulèvement d'un signal d'alarme demandé par le programmeur.

Nous avons réalisé un petit module Nit qui donne accès à ces fonctions. Nous y définissons une classe abstraite qui agit comme réceptrice aux signaux C. Cette classe est composée d'une méthode abstraite de réception et de réaction au signal, ainsi que d'une méthode native qui réalise les appels aux fonctions système pour déclarer le receveur de la méthode comme récepteur des signaux. Les types des signaux à recevoir sont passés en paramètre à la méthode native. La méthode de réception, celle abstraite et pouvant être implémentée en Nit pur, sera invoquée lorsque les signaux attendus sont soulevés.

Pour ce faire, le code C doit préserver une référence globale au récepteur des signaux. Celui-ci est enregistré dans une variable globale par la méthode native qui déclare le receveur. Puis, une indirection est utilisée pour recevoir le signal dans une fonction native, laquelle invoque la méthode voulue sur le récepteur conservé dans la variable globale.

Le code source de ce module est publié dans le dépôt de code de l'implémentation réalisée pour Nit. [6]

Ce module ne doit être utilisé que par un programmeur averti, les signaux système peuvent survenir à tout moment et ainsi interrompre l'exécution du système Nit dans des sections critiques. Pour ces raisons, ce module est destiné pour traiter les signaux d'exceptions fatales, tels que les signaux d'interruption et les erreurs de segmentation.

Cette utilisation de l'interface native consiste en un cas d'appel d'une fonction système et utilise une référence native globale d'un objet Nit. Cette référence globale est utilisée tout au long de l'exécution du logiciel. Ce module démontre le bon fonctionnement des références natives globales à des objets Nit depuis le langage C.

---

6. Le code source du module de traitement de signaux est disponible à `http://xymus.net/nitni/signals`.

6.3.5 Module de sérialisation JSON

Le format JSON [Crockford, 2006] est un standard de sérialisation de dictionnaires, de séquences et de types de bases. Il peut être utilisé par tous les langages, mais sa syntaxe se base sur celle de Javascript.

Nous avons implémenté un module Nit qui consiste en une enveloppe autour d'une bibliothèque native qui implémente le standard Json. Il serait possible de faire l'équivalent en Nit pur, mais utiliser une implémentation existante en C est plus rapide et évite une partie du débogage. Ce module gère indirectement la sérialisation et la désérialisation de différents types d'objets, ce qui implique ainsi l'utilisation du polymorphisme au niveau du module. L'application du polymorphisme depuis le code C étant complexe et simplement à éviter, nous avons implémenté ces conversions en Nit. L'opération principale de désérialisation réalise donc nécessairement des rappels vers les méthodes Nit depuis le code C. Ces méthodes Nit appellent à leur tour les fonctions natives et ainsi amènent une récursivité entre les deux langages. Le code source de ce module est publié avec la bibliothèque standard sur le dépôt de l'interface native implémentée pour Nit. [7]

Cette récursivité a mis à l'épreuve les références locales à des objets Nit depuis le code C. Ces références doivent rester valides malgré le rappel à Nit et l'invocation possible du ramasse-miettes. Notre interface assure correctement le maintien de la validité des références locales.

6.3.6 Module cURL par chargement dynamique

Nous avons réalisé une petite enveloppe de la bibliothèque *cURL* [8] à l'aide du module de chargement dynamique. Celle-ci sert à interagir avec des URL, principalement dans le but de télécharger du contenu.

---

7. Le code source du module JSON en Nit est disponible à `http://xymus.net/nitni/json`.

8. Pour plus d'informations sur les projets cURL et libcurl, voir `http://curl.haxx.se/`.

Nous avons implémenté cette enveloppe entièrement en Nit et elle appelle des fonctions de la bibliothèque native à l'aide du module de chargement dynamique. Son implémentation est donc brève et la bibliothèque n'est chargée que si la classe la représentant est instanciée. Le code source de ce module est publié avec la bibliothèque standard sur le dépôt de l'interface native de Nit. [9]

Cette utilisation est représentative d'un module qui gère le chargement dynamique d'une bibliothèque native en le camouflant aux modules utilisateurs. Elle met à l'épreuve l'interface native à chargement dynamique.

6.3.7 Module GTK+

La bibliothèque GTK+ [10] sert à la création d'interfaces graphiques standardisées. Malgré son implémentation en C, elle simule une hiérarchie de spécialisation des contrôles qu'elle définit. Elle utilise également un système d'événements qui appelle des fonctions personnalisées par le programmeur.

À l'aide de l'interface imbriquée alternative, nous avons commencé le développement d'un module qui sert d'enveloppe à la bibliothèque GTK+. L'aspect imbriqué de cette interface est particulièrement pratique dans ce cas de reproduction de la hiérarchie d'une grosse bibliothèque. Dans ces cas complexes, où nous voulons réaliser le travail complet itérativement par étapes fonctionnelles, l'interface imbriquée évite de gérer manuellement l'évolution des signatures natives des méthodes natives, de même que l'ajout de nouvelles méthodes et l'évolution de la hiérarchie des classes natives. Le code source de ce module est publié dans le dépôt de l'interface native de Nit. [11]

De plus, ce cas d'utilisation bénéficie grandement de la spécialisation des classes natives. Ceci nous a permis de reproduire la hiérarchie simulée dans la bibliothèque, au niveau du module Nit.

---

9. Le code source du module cURL en Nit est disponible à `http://xymus.net/nitni/curl`.

10. Pour plus d'informations sur le projet GTK+, voir `http://www.gtk.org/`.

11. Le code source du module GTK+ en Nit est disponible à `http://xymus.net/nitni/gtk`.

Nous sommes parvenus à représenter adéquatement le système d'événements de GTK+ à l'aide des références globales natives pour les objets Nit. Pour ce faire, nous définissons une classe abstraite qualifiée d'appelable, avec une seule méthode abstraite `call`. Lorsqu'on assigne une action d'un contrôle à une instance de la classe appelable, cette instance est stockée parmi les données personnalisables de l'événement en C. Dans le code C, une seule fonction est définie pour recevoir tous les événements, mais celle-ci retrouve l'instance appelable depuis le code personnalisé et invoque la méthode tel qu'attendu.

De plus, contrairement au cas du module pour les signaux, ces événements sont tous soulevés par la boucle d'exécution de GTK+. Cette dernière est invoquée depuis le code Nit et le tout s'exécute alors correctement pour le système Nit.

Ce cas d'utilisation dévoile une des forces de l'interface alternative de langages imbriqués, soit le travail économisé au programmeur. De plus, elle met à l'épreuve la spécialisation de classes natives sur un grand nombre de cas, l'utilisation d'équivalent natif et les références globales.

6.4 Logiciels entiers

Le fonctionnement d'un module hybride ne garantissant pas par lui-même son fonctionnement dans un vrai logiciel, nous avons créé des logiciels complexes utilisant ces modules. Ces logiciels démontrent que lors d'une utilisation normale de l'interface, il est possible de se servir de plusieurs modules hybrides à l'intérieur d'un même logiciel.

Nous avons implémenté des logiciels ludiques, dont un visionneur non interactif utilisant le réseau, des jeux multijoueurs et une intelligence artificielle de jeu. Ceux-ci utilisent le module SDL pour le graphisme et les contrôles, le module de communication réseau, le module pour le traitement des signaux, le module de chiffrement MD5 et le module de chargement dynamique pour le module cURL.

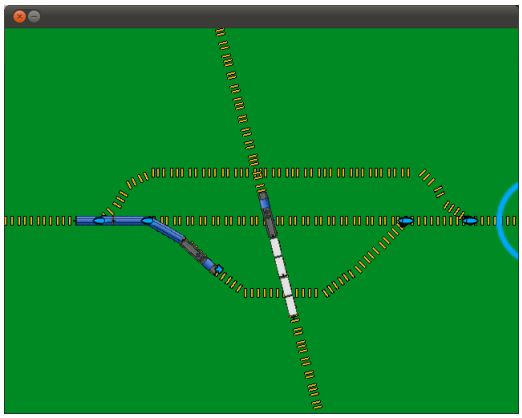

Figure 6.3: Capture d'écran du jeu de simulation de chemins de fer.

6.4.1 Simulation de chemins de fer

Pour tester la bibliothèque graphique SDL nous avons réalisé un jeu simple de simulation de gestion de chemins de fer, en temps réel. Ce logiciel utilise indirectement l'interface native de Nit dans le module SDL. Le jeu permet à son utilisateur de contrôler les embrayages de voies ferrées pour diriger le trafic de trains à travers différents niveaux. Le but est de s'assurer que les trains arrivent à bonne destination et d'éviter les collisions. Une capture d'écran est présentée à la figure 6.3.

Ce jeu, composé de six niveaux, démontre le bon fonctionnement de l'interface native et son efficacité dans le développement d'applications simples. Il s'est avéré que le jeu est parfaitement fluide et toutes les opérations nécessaires pour réaliser l'affichage graphique du jeu sont possibles avec l'interface native de Nit.

6.4.2 Simulateur et visionneur d'évolution d'une forêt

Dans le but de tester d'autres modules Nit utilisant l'interface native ainsi qu'explorer les paradigmes de programmation offerts par le langage Nit, nous avons développé un simulateur de forêt. Celui-ci est réalisé en plusieurs logiciels, un client, un serveur et bien entendu, un logiciel pour les tests. Le logiciel client et le serveur agissent respectivement pour afficher et gérer la simulation.

L'interaction client-serveur est réalisée par communication réseau à l'aide du module de *socket* développé dans ce but. De plus, le client utilise le module graphique SDL pour l'affichage et pour recevoir les entrées du joueur. Une capture d'écran du logiciel client est présentée à la figure 6.4. Le code source de ce logiciel est publié sur son propre dépôt de code. [12]

La logique de jeu est divisée en différents modules qui apportent des fonctionnalités précises. Trois modules gèrent séparément les arbres, les bûcherons et les termites. Ceci permet d'aisément modifier le logiciel, en ne touchant qu'aux importations du fichier source principal et ainsi de simuler la forêt seule ou, avec les bûcherons ou les termites, ou encore, avec tous ces éléments combinés. Un autre module sert à résoudre les conflits lorsque les bûcherons et les termites sont simulés simultanément.

Ce projet nous a permis de constater qu'à l'aide de l'interface native, le langage Nit peut servir au développement d'applications complexes. En ce qui concerne l'interface native elle-même, ce projet a mis à l'épreuve l'utilisation de deux modules hybrides dans un même logiciel ainsi que son mélange avec le raffinement de classes.

6.4.3 Jeu multijoueurs, Lord of Lords

Continuant notre développement de logiciels utilisant l'interface native, nous avons réalisé un jeu plus complexe, intitulé *Lord of Lords*, utilisant l'interface pour plusieurs modules. Celui-ci vise plus loin, un jeu multijoueur de stratégie en temps réel dont la logique de jeu est complexe et persistante. Une capture d'écran de ce jeu est présentée à la figure 6.5.

Ce projet utilise encore les modules de communication réseau et d'affichage graphique SDL, en plus du module de chiffrement MD5 et du traitement de signaux ANSI C. Le module MD5 sert à chiffrer les mots de passe des joueurs avant de les faire parvenir au serveur distant. Les signaux C ne sont utilisés que pour traiter les signaux d'interruption

---

12. Le code source du projet de simulateur de forêt est disponible à `https://github.com/xymus/nit-forest-sim`.

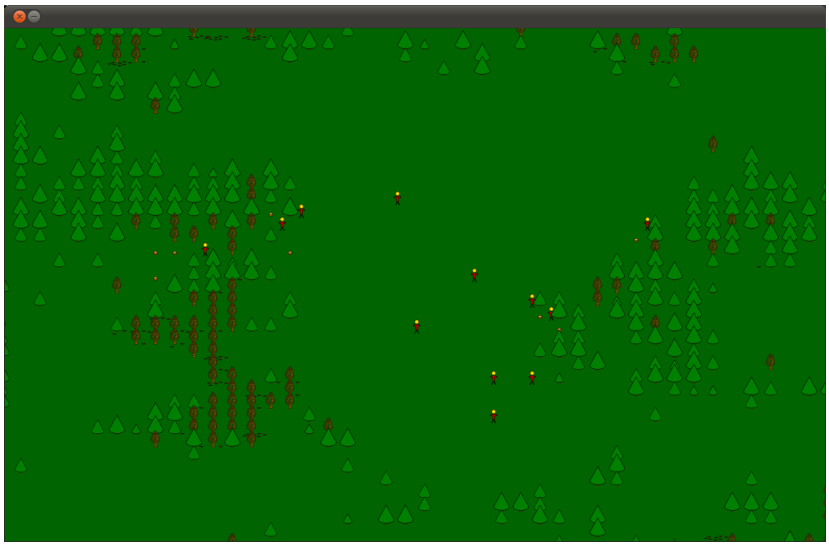

Figure 6.4: Capture d'écran du visionneur de forêt.

côté serveur et ainsi assurer que le jeu se termine proprement, en enregistrant la partie en cours et en fermant adéquatement les ports réseau.

Grâce au langage Nit, la logique de ce jeu est proprement découpée et représentée. Encore une fois, le raffinement de classes permet d'implémenter les différents aspects de la logique de jeu dans des modules peu couplés. Dans ce cas, nous avons un module différent pour gérer l'économie, la population, le système politique féodal, les messages, la carte de jeu, les nations, les échanges commerciaux, les combats et les frontières. Approche qui permet d'ajouter graduellement des fonctionnalités de jeu de façon efficace. De plus, la spécialisation multiple est largement utilisée pour représenter naturellement la logique de jeu.

6.4.4 Intelligence artificielle

Au moment de l'écriture de ce document, nous développons une intelligence artificielle pour le jeu de programmation Berlin [13]. Ce jeu consiste en la réalisation et l'affrontement d'intelligences artificielles gérant des armées sur un graphe de jeu. Les intelligences

13. Pour plus d'informations sur le projet Berlin, voir `http://berlin.thirdside.ca/`.

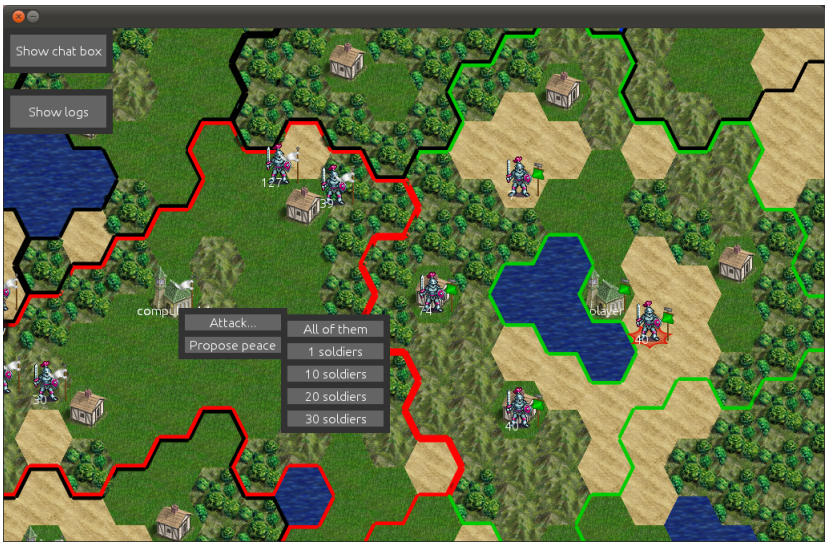

Figure 6.5: Capture d'écran de jeu de Lord of Lords.

artificielles sont développées comme logiciels à part entière qui implémentent un API et interagissent avec le serveur de jeu via le réseau. La communication se fait en format JSON et par des appels HTTP standards.

Pour ce projet, nous avons implémenté le module Json, réalisé l'enveloppe de la bibliothèque cURL (et pour ce faire, le module de chargement dynamique) et entamé le développement d'un serveur web en Nit. Le module Json sert à désérialiser les mises à jour envoyées par le serveur et à sérialiser les réponses de l'intelligence artificielle. Nous n'utilisons que les fonctions de conversion pour le format de chiffrement utilisé par le protocole HTTP de la bibliothèque cURL. Nous nous sommes contentés de développer ce module avec l'interface alternative de chargement dynamique, son utilisation est moins étendue, seules deux fonctions sont utilisées.

Ce projet a profité de la vitesse de développement rendue possible par l'interface à chargement dynamique des bibliothèques natives. Nous avons utilisé cette dernière pour faire appel aux fonctions de conversion de la bibliothèque cURL en Nit pur sans avoir à réaliser un module hybride.

6.5 Observations personnelles

L'un des objectifs de cette étude est la facilité d'utilisation de l'interface développée. Alors que cette aptitude peut difficilement être évaluée expérimentalement, dans cette section nous exposons des observations issues de notre utilisation personnelle de l'interface implémentée.

Une chose importante que nous avons remarquée à l'utilisation de cette interface est la force de la symétrie des types. Les interfaces natives qui utilisent des appels dynamiques pour récupérer les entités de Nit, tel qu'en Java et Python, nécessitent beaucoup de code et de vérification à l'exécution. En comparaison, la symétrie des types et des méthodes introduite dans cette étude permet une manipulation beaucoup plus rapide et naturelle des objets Nit. L'utilisation d'objets en tant que structures de données avancées est même rendue agréable. Cette différence d'utilisation amène le programmeur à employer plus fréquemment les objets Nit dans le code C, non seulement pour assurer le passage des données d'un langage à l'autre, mais aussi pour répondre au besoin du code C comme le ferait toute autre structure bien implémentée.

La déclaration explicite du flot d'appels des méthodes natives semble plus lourde au premier regard, mais un nouvel utilisateur de l'interface bénéficie rapidement de la documentation d'aide produite par le générateur d'échafaudages. Cette documentation affiche la signature des méthodes utilisées par le programmeur et celui-ci n'a donc pas à les deviner.

Le programmeur profite des classes natives surtout grâce aux avantages apportés par le paradigme objet qui permet de développer aisément une structure objet avec spécialisation autour de types natifs existants.

CHAPITRE VII

TRAVAUX CONNEXES

Dans ce chapitre, nous présentons brièvement les travaux d'auteurs que nous n'avons pas encore cités précédemment dans ce document, ces auteurs ont aussi abordé le sujet des interfaces natives au cours des dernières années. Ces recherches traitent généralement du sujet dans l'optique d'y régler des problèmes récurrents, tels que la perte de sûreté à l'exécution et le respect des autorisations d'un logiciel.

Nous discutons de quatre grandes catégories d'apports : les interfaces imbriquant des langages, les analyses dynamiques, les analyses statiques et la sécurité d'exécution des logiciels.

7.1 Interfaces imbriquant le code C

Dans ce document, nous avons précédemment traité le cas de deux études qui présentent des interfaces imbriquant le langage C dans le langage Java. Ces interfaces, Janet [Bubak, Kurzyniec et Luszczek, 2000] et Jeannie [Hirzel et Grimm, 2007], ont été développées pour Java [Gosling et al., 2005] à l'aide de son interface native la JNI [Liang, 1999]. Ces deux interfaces prennent la forme d'un langage dont la grammaire est une composition de celles de Java et de C, dans un même fichier. Elles visent à rendre l'utilisation d'une interface native productive, sûre, portable et efficace.

Dans cette recherche, nous nous sommes inspirés de ces études pour réaliser la version imbriquée de l'interface native. Cette dernière est plus près de l'interface Janet car elle ne permet qu'un seul niveau de code imbriqué.

7.2 Analyse dynamique

Des solutions apportées pour le problème de perte de sûreté à l'exécution des logiciels qui utilisent une interface native prennent la forme d'ajout d'analyses à une interface existante. Deux études [Lee et al., 2010; Tan et al., 2006] ont été publiées avec cette approche. Ces auteurs ajoutent des analyses à la JNI [Liang, 1999] qui peuvent alors être appliquées à tous les logiciels utilisant cette interface.

La bibliothèque Jinn [Lee et al., 2010] se sert d'une machine à état pour observer le flot du logiciel à ses passages par l'interface. Elle assure le suivi de trois grandes catégories de restrictions dictées par la spécification officielle de la JNI. Il s'agit des restrictions sur l'état du logiciel, sur les types utilisés ainsi que sur les ressources. Ces analyses permettent de trouver des bogues qui seraient autrement détectables que par une révision manuelle du code.

Le système SafeJNI [Tan et al., 2006] apporte de multiples composantes pour assurer la sûreté d'exécution d'un logiciel utilisant la JNI. Ces composantes amènent un système de types statiques qui utilise un logiciel externe (CCured [Necula, McPeak et Weimer, 2002]), lequel insère des vérifications dynamiques pour assurer une bonne utilisation des objets Java en natif. Ce système insère aussi d'autres vérifications dynamiques pour vérifier le typage, l'accessibilité des méthodes invoquées, les bornes de tableaux et le traitement des exceptions.

Ces analyses dynamiques assurent le bon fonctionnement d'un logiciel, mais entraînent une perte de performance à l'exécution. Pour cette raison, ils ne sont pas bien adaptés pour les logiciels en production. Les cas vérifiés le sont selon la spécification de l'interface, ceux-ci sont connus d'un utilisateur averti puisqu'ils sont du même type que ceux de tout API C. Quoiqu'il soit bien d'assurer le bon fonctionnement du logiciel, le programmeur est habitué à suivre ce genre de spécifications dans les langages natifs.

Étant donné que notre système de types symétriques et statiques assure déjà une bonne sûreté à l'exécution, il serait possible d'ajouter les fonctionnalités apportées par la bibliothèque Jinn. Celles-ci ne seraient à utiliser qu'en cas de débogage et seraient retirées lors de la compilation pour la production, préservant ainsi une utilisation performante de l'interface native, lorsque nécessaire.

7.3 Analyse statique

Les outils *O-Saffire* et *J-Saffire* [Furr et Foster, 2005], respectivement pour OCaml [Rémy, 2002] et Java [Gosling et al., 2005], utilisent une analyse statique par inférence de type pour valider le code source natif qui réalise l'interconnexion entre les deux langages. Cette analyse identifie les fonctions invoquées en analysant les chaînes de caractères utilisées pour réaliser les appels de fonction depuis le code C. Cette analyse permet de vérifier statiquement la validité de chacun des appels vers le langage de haut niveau, ainsi que la validité des types utilisés.

Cette approche est très intéressante pour assurer le fonctionnement d'un logiciel dès la compilation. Toutefois, elle est conçue pour s'installer par-dessus une interface native existante.

Notre système de types symétriques et statiques amène les mêmes avantages que les systèmes *Saffire*. Ceux-ci s'appliquent bien à une interface existante avec des types dynamiques, mais nous avons observé que nous pouvons atteindre un résultat identique avec les types statiques au niveau de l'interface même.

7.4 Surveillance du comportement du logiciel

L'analyse des langages de haut niveau permet de déterminer leur comportement exact à l'intérieur d'un système. Ceci permet d'assurer la sûreté du système contre des opérations non désirées, par exemple, s'assurer qu'un logiciel ne modifie pas des fichiers du système d'exploitation ou n'accède pas à des données privées. L'utilisation de l'interface native dans un logiciel empêche une analyse complète du code et ainsi une partie de son comportement ne peut être connue, ceci fait alors perdre la garantie de connaissance du comportement du logiciel.

Pour régler ces problèmes, deux recherches [Klinkoff et al., 2006; Yee et al., 2009] proposent une solution similaire, c'est-à-dire séparer les processus pour les fonctions natives de celui du logiciel principal. Pour réaliser les appels, ils utilisent des techniques provenant de la programmation parallèle, avec des appels de fonctions distantes. Ils utilisent

des déclencheurs (*hooks*) sur les fonctions système pour surveiller le comportement du processus natif.

Cette approche permet de protéger l'exécution du code de haut niveau et de surveiller le comportement du code natif. Par contre, l'appel de fonctions distantes est une technique très coûteuse en performance lors de l'exécution du logiciel.

Les solutions apportées sont trop coûteuses en performance pour être applicables dans tous les cas d'utilisation d'une interface. Nous ne les avons donc pas appliquées à l'interface native développée pour Nit. Ces solutions seraient cependant à reconsidérer si une interface native spécialisée est développée pour des applications de source de confiance douteuse.

CONCLUSION

Dans cette étude, nous traitons le sujet de l'interface native du langage de programmation à objets Nit en ayant pour but de concevoir une interface native idéale pour ce langage. Tout logiciel complexe, tel que ceux comportant une interface graphique, réalisant des opérations réseau ou utilisant toute autre fonction système, se base sur une interface native directement ou indirectement. L'interface native d'un langage est donc largement utilisée, sa qualité influence directement celle de tout logiciel réalisé en ce langage.

Dans ce document, nous exposons les solutions apportées par d'autres interfaces natives, qu'elles soient officielles à un langage ou proposées par la littérature scientifique. Nous évaluons ces approches comparativement à de nouvelles solutions dans le but de répondre au mieux aux cas les plus communs dans lesquels l'interface native de Nit sera utilisée.

Pour valider la théorie, nous avons implémenté notre concept d'interface native dans le compilateur du langage Nit. Quoiqu'il s'agisse de la seule implémentation du langage au moment de l'expérimentation, nous croyons que l'interface native développée sera compatible avec les implémentations futures du langage.

Cette interface applique quatre grandes approches innovatrices : les modules hybrides ; la déclaration explicite du flot d'appels des méthodes natives ; la représentation des entités Nit en C sous forme de types symétriques et méthodes symétriques ; ainsi qu'une nouvelle grande catégorie de classes, les classes natives.

La forme générale que prend l'interface est celle de modules hybrides, déclarés en Nit et partiellement implémentés en C. Cette forme permet de conserver la pureté de Nit et C en séparant clairement les fichiers sources, et ce, tout en préservant la cohésion du module. Elle se base sur des méthodes natives, déclarées en Nit mais implémentées

en C. De plus, nous utilisons cette forme comme base à d'autres interfaces natives spécialisées : un outil pour utiliser des langages imbriqués et un module Nit pour le chargement dynamique de bibliothèques natives.

Nous parvenons à maintenir la connaissance du flot d'appels de méthodes par une déclaration explicite des méthodes appelées depuis le code C. Ceci permet de préserver l'optimisation et les vérifications du logiciel final offertes par le compilateur Nit. Enfin, nous réutilisons cette information supplémentaire pour donner une bonne documentation au programmeur lors de la réalisation de l'implémentation des méthodes natives.

L'utilisation d'une représentation des entités Nit en C sous la forme de types et méthodes statiques leur donne une forme native naturelle en C et permet de préserver la vérification statique des types Nit. De plus, nous proposons aussi une approche adaptée à chaque type spécial tels que les types génériques et les nullables.

La représentation en Nit des types C avec les classes natives facilite l'utilisation de types C au travers de l'interface native. Cette approche permet aussi de profiter des avantages du paradigme de programmation à objets, tels que la spécialisation, pour des types C dans le code Nit.

Enfin, ces quatre grandes approches sont mises à l'épreuve par l'implémentation de l'interface native du langage Nit ainsi que par la réalisation de modules et logiciels Nit. Ces modules utilisent directement l'interface native principale ou une de nos alternatives pour offrir des services implémentés en C sous une forme naturelle au langage Nit. Les logiciels réalisés à l'aide de ces interfaces démontrent que l'interface native proposée est viable et même qu'elle facilite la réalisation de logiciels partiellement en code C.

L'interface que nous présentons dans ce document est adaptée aux besoins spécifiques de Nit. Certaines de ses parties sont compatibles directement avec d'autres langages de programmation à objet. Par exemple, la symétrie des types et méthodes serait applicable à tout langage avec typage statique. D'autres parties prennent une forme spécifique à Nit, mais elles pourraient être améliorées avec d'autres langages. Par exemple, le traitement des fermetures est limité par l'interface native de Nit, toutefois, un langage

utilisant des fermetures plus simples ou un système de délégué tel que C# pourrait appliquer une solution plus efficace pour traiter les fermetures en C. Nous considérons que les quatre approches qui sont à la base de l'interface native de Nit pourraient servir de base à de nouvelles interfaces pour tout langage de programmation à objets.

Travaux futurs

Le concept de l'interface native du langage Nit peut être adapté pour tout autre langage de programmation à objets. Une interface native semblable pourrait être réalisée pour d'autres langages permettant ainsi de bénéficier d'une syntaxe native naturelle et de la sûreté d'exécution amenée par nos contributions.

Au moment de l'écriture de ce document, le langage Nit ne spécifie pas encore de système de gestion des exceptions. Alors que nous avons réalisé une analyse sommaire des solutions existantes à ce problème, une étude en profondeur sera nécessaire lorsque les détails du système de gestion des exceptions seront connus.

De plus, à ce jour, le langage Nit n'offre pas encore la compilation séparée des modules. Davantage de recherches devront être effectuées pour adapter correctement l'interface native selon la spécification du langage, lorsque cette option sera disponible. Par une étude rapide du problème et l'analyse des autres interfaces existantes, il nous apparaît que le concept des modules hybrides, qui restreignent bien l'utilisation de l'interface à l'intérieur de chaque module, devrait bien survivre à l'ajout des bibliothèques. Ces modules compilés permettraient en plus d'utiliser des modules hybrides comme tout autre par un futur interpréteur du langage.

L'interface devra être revue pour une utilisation avec parallélisme. Certains détails devront être ajoutés à l'interface pour en assurer le bon fonctionnement dans ce contexte. La première étape serait l'ajout d'une variable d'environnement dans la signature native des méthodes Nit, celle-ci servirait à conserver les références globales propres à un processus.

BIBLIOGRAPHIE

1989. *American National Standard Programming Language C, ANSI X3.159-1989.*

Beazley, D., *et al.* 1996. « Swig : An easy to use tool for integrating scripting languages with c and c++ ». In *Proceedings of the 4th USENIX Tcl/Tk workshop*, p. 129–139.

Bubak, M., D. Kurzyniec et P. Luszczek. 2000. « Creating Java to native code interfaces with Janet extension ». In *Proceedings of the First Worldwide SGI Users' Conference*, p. 283–294.

Carlson, L., et L. Richardson. 2006. *Ruby Cookbook*. O'Reilly, États-Unis.

Command cgo. 2011. « Command cgo ». <`http://golang.org/cmd/cgo/`>.

Crockford, D. 2006. « The application/json media type for javascript object notation (json) ». *`http://www.ietf.org/rfc/rfc4627.txt`*.

Ducournau, R. 2002. « "real world" as an argument for covariant specialization in programming and modeling ». *Advances in Object-Oriented Information Systems*, p. 3–13.

Ducournau, R., et J. Privat. 2010. « Metamodeling semantics of multiple inheritance ». *Science of Computer Programming*, p. 32.

ECMA. 2006. C# language specification. Rapport no. 334, ECMA.

Edelson, J., et H. Liu. 2008. *JRuby cookbook*. O'Reilly Media, États-Unis.

Furr, M., et J. Foster. 2005. « Checking type safety of foreign function calls ». In *ACM SIGPLAN Notices*. T. 40, p. 62–72. ACM, États-Unis.

Gélinas, J., É. Gagnon et J. Privat. 2009. « Prévention de déréférencement de références nulles dans un langage à objets », vol. L-3 de RNTI, p. 5–16.

Gosling, J., B. Joy, G. Steele et G. Bracha. 2005. *The Java Language Specification*. Addison-Wesley, États-Unis, troisième édition.

H., K. 2002. Gcc-avr inline assemble cookbook. Rapport, egnite Software GmbH.

Hall, J. 2001. *Programming linux games*. No Starch Press, États-Unis.

Hirzel, M., et R. Grimm. 2007. « Jeannie : Granting java native interface developers their wishes ». In *Proceedings of the 22nd annual ACM SIGPLAN conference on*

Object-oriented programming systems and applications, p. 19–38. ACM, États-Unis.

Hugunin, J., *et al.* 2004. « Ironpython : A fast python implementation for .net and mono ». *PyCon. Python Software Foundation, March*.

Juneau, J., J. Baker, L. Soto, V. Ng et F. Wierzbicki. 2010. *The definitive guide to Jython : Python for the Java platform*. Springer, Allemagne.

Kernighan, B., D. Ritchie et P. Ejeklint. 1988. *The C programming language*. T. 78. Prentice Hall, États-Unis.

Klinkoff, P., C. Kruegel, E. Kirda et G. Vigna. 2006. « Extending. net security to unmanaged code ». *Information Security*, p. 1–16.

Lee, B., B. Wiedermann, M. Hirzel, R. Grimm et K. McKinley. 2010. « Jinn : synthesizing dynamic bug detectors for foreign language interfaces ». *ACM SIGPLAN Notices*, vol. 45, no. 6, p. 36–49.

Liang, S. 1999. *The Java Native Interface*. Addison-Wesley, États-Unis.

Madiath, V. 2010. The vala guide. Rapport.

Meyer, B. 1987. « Eiffel : programming for reusability and extendibility ». *ACM Sigplan Notices*, vol. 22, no. 2, p. 85–94.

———. 1992. « Eiffel : the language ».

Necula, G., S. McPeak et W. Weimer. 2002. « Ccured : type-safe retrofitting of legacy code ». In *ACM SIGPLAN Notices*. T. 37, p. 128–139. ACM, États-Unis.

Pappas, C., et W. Murray. 1995. *The visual C++ handbook*. McGraw-Hill Osborne Media, États-Unis.

Privat, J. 2002. « Analyse de types et graphe d'appels en compilation séparée ». *Mémoire de DEA, Université Montpellier II, France*.

———. 2006. « De l'expressivité à l'efficacité une approche modulaire des langages à objets ». Thèse de Doctorat, Université Montpellier, France.

———. 2011. A concise reference of the nit langage. Rapport, UQAM, Canada.

Privat, J., et R. Ducournau. 2005. « Raffinement de classes dans les langages à objets statiquement typés ». In *Langages et Modèles à Objets (LMO'05)*. T. 11(1-2) de L'objet, p. 17–32.

Rémy, D. 2002. « Using, understanding, and unraveling the ocaml language from practice to theory and vice versa ». *Applied Semantics*, p. 115–137.

Rivest, R. 1992. « The md5 message-digest algorithm ». *http://www.ietf.org/rfc/rfc1321.txt*.

Ruby Inline Readme. 2008. « Ruby inline readme ». <`http://rubyinline.rubyforge.org/RubyInline/`>.

Stevens, W., B. Fenner et A. Rudoff. 2003. *UNIX network programming : volume 1 : the sockets networking API*. Addison-Wesley, États-Unis.

Stevens, W., S. Rago et D. Ritchie. 1992. *Advanced programming in the UNIX environment*. T. 4. Addison-Wesley, États-Unis.

Stroustrup, B. e. a. 1997. *The C++ programming language*. T. 3. Addison-Wesley Reading, États-Unis.

Tan, G., A. Appel, S. Chakradhar *et al.* 2006. « Safe Java native interface ». In *Proceedings of IEEE International Symposium on Secure Software Engineering*, p. 97–106.

The Go Programming Language Specification. 2011. « The go programming language specification ». <`http://golang.org/doc/go_spec.html`>.

Thomas, D., C. Fowler et A. Hunt. 2004. *Programming Ruby : The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, États-Unis, second édition.

v. Rossum, G. 2009a. *Extending and Embedding the Python Interpreter*. Python Software Foundation, États-Unis, 3.0.1 édition.

———. 2009b. *The Python Language Reference*. Python Software Foundation, États-Unis, 3.0.1 édition.

———. 2009c. *The Python Library Reference*. Python Software Foundation, États-Unis, 3.0.1 édition.

———. 2009d. *The Python/C API*. Python Software Foundation, États-Unis, 3.0.1 édition.

Woo, M., J. Neider, T. Davis et D. Shreiner. 1999. *OpenGL Programming Guide : the official guide to learning OpenGL*. Addison-Wesley Longman Publishing, États-Unis.

Yee, B., D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula et N. Fullagar. 2009. « Native client : A sandbox for portable, untrusted x86 native code ». In *2009 30th IEEE Symposium on Security and Privacy*, p. 79–93. IEEE, États-Unis.