Notes de Lecture: Linear Regions Are All You Need

Hector Suzanne

Introduction

La plupart des langages de avec garanties de sûreté utilisent un ramasse-miette dont le comportement dynamique a un coût parfois inacceptable en latence (*le temps que le GC tourne*) ou fuites mémoires (*le GC n'est pas idéal*). Les auteurs ont développés le langage *Cyclone*, un dialecte de C avec typage statique et gestion mémoire manuelle mais sûre. Le modèle mémoire de Cyclone est basé sur un mécanisme de *région* basé sur Tofte et Talpin, où les régions qui possèdent la mémoire allouée dynamiquement sont LIFO. Hélas, avec ce modèle, il est en pratique difficile de coder des programmes aussi efficaces que ceux avec GC à cause des fuites mémoires dans les régions en bas de pile.

Pour palier à cela, Cyclone incorpore des *régions dynamiques* et des *pointeurs uniques* qui permettent un contrôle plus fin de la mémoire, utile notamment pour le code itératif ou à base de continuations, où le système LIFO n'est pas assez expressif. Les pointeurs uniques sont des régions dynamiques contenant un unique objet. La sûreté des régions dynamique est assurée par les *capacités linéaires*, des témoins légers réifiant les droits de libérations des régions dynamiques.

Dans cet article, nous prouvons la sûreté de ces mécanismes dans un langage simple,λ^{rngUL}, dans lequel un théorème de cohérence d'un système de type sous-structurel permet est aisé. Nous procédons en deux temps: D'abord, nous traduisons notre langage à types et effets dans une variante monadique du système F nommée F^{RNG} avec des primitives sûres de gestion de régions; Nous l'avions introduit dans des travaux précédents. Ensuite, nous traduisons ce langage dans λ^{rngUL} en dépliant les calculs monadiques. La preuve de cohérence est établie simplement dans le langage cible. Nous avons créés λ^{rngUL} dans l'optique de servir d'outil mental et de représentation intermédiaire dans les compilateurs.

Langage source: F^{RNG}

Notre langage cible est inspiré de la monade ST de Launchbury & Peyton-Jones. ST (*courant en Haskell*) représente les calculs avec état comme des transformateurs sur un conteneur d'état opaque et modifié en place. Muni de polymorphisme paramétrique, on peut alors encapsuler et faire tourner des calculs avec état mutable. Nous étendons cette monade avec comme conteneur une *pile de région* dans lesquelles ont peux allouer des valeurs mutables accessibles par références.

Notre langage se compose du système F muni des types suivants:

$$\tau = ... \mid rgn\ s\tau \mid ref\ s\tau \mid hnd\ s \mid pf(s_1 \leq s_2)$$

Les régions sont représentée par des indices s de *kind* différent des variables de types classique de *kind* $\star$. Les calculs avec régions renvoyant des τ sont $rgn\ s\tau$, et peuvent utiliser des références de type $ref\ s'\tau'$. On peut allouer dans une région en donnant son *témoin* $hnd\ s$. Les régions sont munies d'un pré-ordre avec les témoins associés $pf(s_1 \leq s_2)$. Les termes du système F sont enrichis des opérations monadiques sur $rgn\ s$, de l'accès en lecture/écriture aux références, des opérations de pré-ordre des témoins d'inclusion et de coercion le long des chaînes du pré-ordre. Enfin, les primitives d'allocations sont:

$$\texttt{new} : \forall\sigma.\forall\alpha.\ hnd\ \sigma \to \alpha \to rgn\ \sigma\ (ref\ \sigma\ \alpha)$$

$$\texttt{letRgn} : \forall\sigma_1.\forall\alpha.\ (\forall\sigma_2.\ pf(\sigma_1 \leq \sigma_2) \to hnd\ \sigma_2 \to rgn\ \sigma_2\ \alpha) \to rgn\ \sigma_1\ \alpha$$

`letRng` permet un usage non-strictement lexical des régions: on peut allouer une valeur mutable dans une région qui n'est pas au sommet de la pile, du moment qu'on peut prouver qu'elle vivra plus longtemps que la région dans laquelle cette valeur sera utilisé. Le

Le typage dans F^{RGN} est exactement celui de F avec un environnement de base enrichi des opérations sur les régions. On a donné dans des travaux précédent une sémantique opérationelle de F^{RGN} et un Théorème de cohérence. Mais la sémantique doit maladroitement entremêler les calculs purs et les mutations dans les région. On donne ici une sémantique opérationelle via une traduction dans λ^{rngUL}.

Langage cible: λ^{rngUL}

Dans des travaux précédents, nous avions introduit un λ-calcul polymorphique sous-structurel muni de références avec accès modaux. Les modes (linéaire, affine, illimité, pertinent) permettaient d'y encoder et d'étudier les différentes sortes d'accès unique dans les langages de haut-niveaux et comment ils interagissent. Pour cet article, nous adaptons cet outils dans λ^{rngUL}, qui possède en

plus de modalités d'accès "illimitée" (U) et linéaire (L), contient des primitives de création et destructions manuelle de régions modalisées par des *capacités linéaires* (*N.D.L. à la Rust, avant l'heure*). Dans ce langage, les accès au régions via les références exigent la possession (*prêt en Rust*) d'une capacité linéaire créée par `newrgn` et consommé définitivement par `freergn`.

Les types du langages suivent ceux d'un -calcul polymorphique. Ils sont formés d'un pré-type classique muni d'un modal U ou L. Les pré-types contiennent le produit et l'unité, la flèche, les quantificateurs universels et existentiels et les variables associés. On ajoute à ces pré-types classiques les références $ref\ r\ t$, les témoins $hnd\ r$ similaires à ceux de F^{RNG}, et les capacités d'accès $cap\ r$ à une région r. Le passage des pré-types au types induit une notion de sous-typage: comme $U \prec L$ (ce qu'on peut faire avec les types linéaires, on peut le faire avec les types illimités), on peut définir un sous-typage entre les types/contextes et les modalités $\tau \prec q$ et $\Gamma \prec q$. Enfin, dans l'esprit d'un typage sous structurel, on dispose d'une opération de séparation des contexte qui évite la duplication des types linéaire (*que l'on note dans ce document* $\Gamma = \text{split}(\Gamma_1, \Gamma_2)$).

La syntaxe des termes est celle d'un λ-calcul polymorphique classique, où les constructeurs de données sont modaux (c.à.d. annotés par un q), et où les primitives de création et libération de régions, de création, lecture et écriture de références sont disponibles. Ces primitives sont munis de types garantissant la cohérence de λ^{rngUL} par rapport à une sémantique opérationelle à petit-pas, où l'état (ψ, e) contient une mémoire ψ liant les noms de régions aux régions, et dans les régions, les pointeurs aux valeurs. La cohérence est établie formellement en garantissant que les primitives d'accès aux références soient bien prêtés les témoins des témoins des régions qu'elles accèdent.

L'expressivité du modèle mémoire de ce langage est suffisante pour encoder la gestion monadique des régions de F^{RGN}, mais aussi les fonctionnalités plus dynamiques de Cyclone citées en introduction. On décrit maintenant la traduction de cette gestion monadique, puis de ces fonctionnalités, dans λ^{rngUL}.

Traduction

Premières remarques:F^{RGN} n'est pas syntaxiquement ou sémantiquement linéaire, on présage donc que ses types seront annotés U. Mais, les opérations monadiques permettent de mettre à jour l'état en place, ce qui est parfaitement adapté au typage sous-structurel du langage cible. La pile de régions devrait donc pouvoir être traduite avec une modalité L. On va la représenter comme un pile imbriquée de tuples contenant des capacités linéaires. Enfin, il convient de remarquer que si les objets de F^{RNG} indicent la pile de région, ils seront traduits par des objets mentionnant explicitement la région qu'ils impliquent.

La partie la plus intéressante de la traduction concerne $pf(\sigma_1 \leq \sigma_2)$, qui est traduit par un isomorphisme entre la traduction de σ_2 est un couple de σ_1 et d'information supplémentaire β. On peut reconstruire Les types de σ_i et β sont

linéaire, mais l'isomorphisme de l'est pas: bien que les objets qu'ils types soient éphémères, l'isomorphisme tient dès qu'ils existent tout les deux !

La traduction de `letRgn` est la plus complexe, mais suit un schéma suffisamment simple pour que l'on puisse garantir sa cohérence. On procède en créant une région pour l'évaluation interne, en l'empilant sur la pile de capacité et en créant l'isomorphisme adéquat. A la fin de l'exécution interne, on peut décomposer la pile et libérer la région allouée en début de calcul.

Extensions

Utiliser la gestion mémoire de bas-niveau proposée par λ^{rngUL} permet de s'affranchir de la discipline de pile, et donc de modéliser plus de paradigme de gestion mémoire réels. Nous avons encodé les régions dynamiques dans λ^{rngUL}, en les représentant comme des *clés*: des couples linéaires de témoins de région et de capacité dont la capacité peut être prêté aux calculs sur la région. Le témoin nécessaire à la libération de la région est gardé en sécurité par la primitive d'usage des régions dynamiques. Dans un exemple réel en Cyclone, un serveur web utilise une région dynamique car connexion, permettant ainsi une programmation concurrente qui s'arrangerait mal des contraintes LIFO de l'interface monadique. Le prix à payer est qu'il faut manuellement garantir la linéarité des clés, ce que le système de type de Cyclone ne fait pas.

On peut aussi implémenter les pointeurs uniques comme des paquets existentiel sur une région qui contient la valeur pointée. Le paquet offre accès à la capacité linéaire d'accès *via* le pointeur et une référence vers la valeur pointée à qui on peut prêter la capacité pour accéder à la ressource. La paquet contient enfin une fonction consommant pour de bon la capacité et libérant la région afférente.

Enfin, comme les capacités et les isomorphismes ne sont pas utilisés au runtime, il est possible d'introduire un système de phase séparant les valeurs de la compilation de celle de l'exécution. L'introduction d'une nouvelle modalité *statique/dynamique* garantirait que les calculs statiques n'utilisent pas de valeurs dynamiques. (*Ça ressemble à du typage dépendant tout ça*)

Questions Ouvertes et Travaux Annexes

Des travaux précédents étendent la discipline de région de Tofte et Talpin pour implémenter des régions à durée de vie non-lexicale (*voir Rust*), nous pensons que ces systèmes admettent un encodage dans le notre. Plus généralement, notre langage peux servir de représentation intermédiaire pour nombre d'outils basés sur la discipline de région, pour des langages comme Cyclone, ou pour écrire des ramasse-miettes sûrs basés sur une discipline de type de haut-niveau.

Enfin, le langage Vault a des fonctionnalités idoine de celles présentées dans cet article, mais support aussi la mutabilité de ressources linéaires et la coercion

temporaire de U à L. Il n'existe par (*en 2006*) de justification formelle de la sûreté de ces mécanismes. Nous pensons que λ^{rngUL} et nos travaux précedent peuvent fournir une base à ce type de justification.

Le lecteur note que Rust supporte ce modèle mémoire, avec quantifié de fioritures, et qu'il a été formellement prouvé sûr via le projet RustBelt en 2017, soit dix ans après la publication du présent article. RustBelt utilise néanmoins un système à base de logique de séparation.