Formalisation et implémentation de techniques sûres de passage de destinations pour les langages de programmation fonctionnels purs

Thomas Bagrel

Résumé de thèse de doctorat, soutenue le 14 novembre 2025

1 Introduction

Tout comme la pléthore de langues humaines, il existe un nombre impressionnant de langages de programmation différents, chacun avec des choix de conception reflétant des compromis entre expressivité, performance, sûreté et facilité d'utilisation.

Un des aspects par lesquels les langages se distinguent est leur approche de la gestion mémoire. Ce choix influence profondément la manière dont les programmes sont écrits. Les langages de haut niveau, comme Python ou Java, délèguent la gestion mémoire à un ramasse-miettes, simplifiant l'écriture de code mais introduisant une latence imprévisible. À l'opposé, les langages bas niveau, tels que C ou Zig, confient la gestion mémoire au programmeur, offrant un contrôle fin mais exposant à des erreurs très coûteuses. Des approches plus récentes, comme les pointeurs intelligents en C++ ou le modèle d'*ownership* en Rust, tentent de concilier sûreté et performance sans recourir à un système de ramasse-miettes, mais ces dernières ne s'appliquent pas trop aux langages fonctionnels, qui sont l'object de notre étude.

Langages de programmation fonctionnels Les langages de programmation fonctionnels se caractérisent d'abord par l'accès à des fonctions pouvant être manipulées, stockées, ou passés en paramètres comme n'importe quelle autre valeur. Ils affichent également une préférence pour les expressions (à la place des instructions ayant des effets de bords) et les structures immuables. Inspirés et proches de modèles mathématiques, leur formalisation est facilitée, et il présentent de nombreuses propriétés intéressantes pour leur analyse statique (permettant d'éviter de nombreuses classes d'erreurs). C'est d'autant plus le cas pour les langages fonctionnels *purs*, où les effets de bords sont clairement circonscrits et explicitement représentés au niveau du système de type, et où toute fonction est transparente par substitution. La pureté impose alors usuellement l'utilisation d'un système de gestion automatique de la mémoire, typiquement un ramasse-miettes, limitant les possibilités de contrôle explicite (qui peuvent sembler antithétiques avec l'idée de pureté).

Structures à trous Cependant, Minamide a présenté en 1998 un premier système permettant de représenter des structures incomplètes, ou « avec trous », c'est à dire des structures de données non encore finalisés, au sein d'un langage fonctionnel pur. Ces structures peuvent être progressivement complétées, tout en garantissant, via un usage ingénieux d'un système de

types linéaires, qu'elles ne sont lisibles qu'une fois entièrement initialisées. Cette technique permet de dépasser certaines contraintes habituelles des langages fonctionnels purs (par exemple l'immuabilité des structures de données, qui force souvent à créer des copies intermédiaires) sans introduire de risques d'erreurs.

Ma thèse poursuit cette idée en proposant un langage fonctionnel où les structures avec trous sont dotées de pointeurs d'écriture explicites – les destinations – pointant vers leurs trous. Cette approche n'est pas sans rappeler le passage de paramètres par adresse dans les langages impératifs, mais elle est ici adaptée à un contexte fonctionnel pur, en réutilisant les idées pionnières de Minamide.

Grâce à la manipulation explicite de *destinations*, il est possible à la fois d'exprimer certains algorithmes – dont l'implémentation fonctionnelle était peu ergonomique – d'une manière simple se rapprochant des approches impérative ; mais également de permettre une gestion mémoire semi-manuelle afin d'obtenir des gains de performances quand la gestion automatique du ramasse-miette n'est pas optimale. Je développe d'abord un langage formel, servant de cadre théorique pour raisonner sur la sûreté et la correction des approches de passage de destinations, avant de concrétiser mon approche par un prototype en Haskell.

L'ensemble de mon approche repose, comme pour Minamide, sur un système de types incorporant les types linéaires. Grâce aux types linéaires, il est possible d'imposer et d'assurer que chaque destination est bien utilisée une seule et unique fois, garantissant à la fois que chaque champ d'une structure est correctement initialisé avant toute lecture, mais également qu'une valeur ne sera pas écrasée par une autre une fois écrite dans un champ d'une structure (permettant de respecter l'immuabilité chère aux langages fonctionnels).