Extension OPTIM — Retour de Mise en Fonction (RMF)

État d'avancement, écarts avec la spécification initiale et perspectives

Faical EL GOUIJ

2026-01-15

Contents

1 Introduction

Ce document constitue un **Retour de Mise en Fonction (RMF)** de l'extension **OPTIM** du compilateur Deca. Il vise à présenter l'état réel des optimisations implémentées à ce stade du projet, à analyser les écarts entre la **spécification initiale** et les **choix techniques effectifs**, et à proposer une vision structurée des perspectives d'évolution.

Contrairement à la spécification initiale, qui se voulait volontairement ambitieuse et conceptuelle, ce RMF reflète une **approche pragmatique**, guidée par : - les contraintes réelles du compilateur Deca, - la nécessité de préserver strictement la sémantique, - les limites raisonnables d'un projet pédagogique.

---

2 Positionnement actuel de l'extension OPTIM

L'extension OPTIM est désormais structurée autour de **passes distinctes**, intervenant à deux niveaux :

2.1 Optimisations sur l'AST (avant génération IMA)

- Constant Folding
- Élimination de code mort

2.2 Optimisations sur le code IMA généré

- Peephole optimization avancée
- Simplifications arithmétiques locales
- Propagation de copies (SSA-lite linéaire)

Cette séparation permet de distinguer clairement : - les optimisations **sémantiques et structurelles**, - des optimisations **machine-dépendantes et locales**.

---

3 Analyse du flot de contrôle (CFG) — Retour critique

3.1 Intention initiale

La spécification initiale proposait la construction d'un **CFG partiel**, afin de détecter : - des blocs inatteignables, - des structures triviales (`if(false)`, `while(false)`), - du code mort après `return`.

3.2 Difficultés rencontrées

L'implémentation a mis en évidence plusieurs limites majeures :

- gestion délicate des labels et des branchements implicites ;
- nécessité d'un calcul de dominance même pour des cas simples ;
- risque élevé d'erreurs sémantiques sans CFG global fiable.

Une implémentation complète, même partielle, du CFG s'est révélée **disproportionnée** au regard des gains attendus.

3.3 Choix final

L'extension OPTIM ne construit pas de CFG explicite. À la place, elle adopte une stratégie **structurelle et conservative** intégrée à l'AST :

- suppression du code après `return`,
- élimination de blocs triviaux déjà simplifiés,
- suppression d'affectations et déclarations inutiles.

Ces transformations sont centralisées dans la passe **DeadCodeEliminator**, appliquée jusqu'au point fixe.

---

4 SSA : de la théorie à une SSA-lite linéaire

4.1 Rappel de la spécification initiale

La SSA avait été envisagée comme un outil central pour : - la propagation de constantes, - la détection d'affectations inutiles, - la clarification des dépendances de données.

Un SSA complet (dominance, fonctions ϕ) avait cependant été explicitement exclu.

4.2 Implémentation effective : propagation de copies

L'optimisation mise en œuvre correspond à une **propagation de copies par parcours linéaire** :

- parcours séquentiel du code IMA ;
- suivi des équivalences entre variables (`b = a`, `c = b`, etc.) ;
- propagation transitive dans les séquences droites ;
- invalidation immédiate dès qu'un point d'incertitude apparaît (branchement, label, écrasement de registre).

Cette passe ne repose sur : - aucun CFG, - aucune analyse de dominance, - aucun renommage global.

Elle constitue ainsi une **approximation SSA-lite strictement linéaire**, volontairement conservative.

---

5 Constant Folding : stabilisation et convergence

La passe de **Constant Folding** est désormais pleinement intégrée :

- elle opère exclusivement sur l'AST ;
- elle applique les réécritures récursivement ;
- elle itère jusqu'à convergence avec une borne de sécurité.

Le Constant Folding joue un rôle fondamental : - simplification précoce de l'AST, - déclenchement d'optimisations ultérieures, - réduction du code généré.

Cette implémentation est conforme à la spécification initiale, avec l'ajout d'une gestion explicite du point fixe.

---

6 Peephole Optimization : implémentation avancée

6.1 Écart avec la spécification initiale

La spécification décrivait un peephole classique à fenêtre courte. L'implémentation actuelle adopte une approche plus agressive :

- exploration vers l'avant tant qu'aucune barrière n'est rencontrée ;
- prise en compte explicite des registres lus et écrits ;
- gestion prudente des instructions de contrôle, appels et labels.

6.2 Optimisations effectivement implémentées

Parmi les optimisations réalisées :

- suppression de `ADD #0`, `MUL #1`, `DIV #1` ;
- transformation de `MUL #0` en `LOAD #0` ;
- propagation de `LOAD` entre registres ;
- suppression de `LOAD` immédiats redondants ;
- élimination de `STORE` écrasés ;
- suppression conditionnelle des `BOV` devenus inutiles.

Le peephole agit ainsi comme une **analyse de flot locale**, tout en restant strictement séquentielle.

---

7 Interactions entre les passes OPTIM

Les différentes passes OPTIM forment un **pipeline coopératif** :

1. Constant Folding simplifie l'AST.
2. Propagation de copies clarifie les dépendances.
3. Dead Code Elimination supprime l'inutile.
4. Peephole nettoie le code IMA final.

Chaque passe renforce l'efficacité des suivantes, sans dépendances circulaires dangereuses.

---

8 Perspectives et évolutions possibles

Plusieurs axes d'évolution sont identifiés :

- enrichissement progressif de l'analyse de flot sans CFG global ;
- extension de la propagation de copies à des motifs plus complexes ;
- amélioration de la détection des barrières dans le peephole ;
- exploration d'un SSA plus formel si une infrastructure CFG fiable est introduite.

---

9 Conclusion

L'extension OPTIM a évolué d'une **spécification conceptuelle ambitieuse** vers une **implémentation pragmatique, sûre et convergente**.

Les choix réalisés privilégient : - la correction sémantique, - la lisibilité des transformations, - des gains d'optimisation mesurables et justifiables.

Ce RMF constitue une base solide pour une évolution progressive et maîtrisée de l'extension.