Extension OPTIM

Spécification détaillée

Architecture et choix des algorithmes

Projet Génie Logiciel — ENSIMAG

EL GOUIJ Faical

Tuteur : M. NICOLLIN Xavier

Date : January 7, 2026

Contents

1 Introduction

Ce document présente une **spécification préliminaire** de l'extension `OPTIM` du compilateur `Deca`.
Il ne s'agit pas d'un rapport d'implémentation finale, mais d'un **document d'architecture et de réflexion**, servant à :

- introduire les **principes théoriques** d'optimisation envisagés ;
- justifier les **choix algorithmiques** retenus ;
- expliciter les **limitations imposées par le cadre du projet Deca** ;
- servir de **base bibliographique et conceptuelle** pour l'implémentation.

Les techniques décrites ci-dessous (CFG, SSA, optimisations locales) sont issues des compilateurs classiques, mais les versions proposées dans ce document sont partielles, pragmatiques et évolutives. Ainsi, bien que notre stratégie initiale ne traite pas ces techniques dans toute leur généralité, nous insistons sur le fait qu'il s'agit d'une réflexion préliminaire, susceptible d'être enrichie ultérieurement afin de couvrir un spectre plus large de cas et d'exploiter davantage le potentiel de ces approches.

2 Positionnement de l'extension OPTIM dans le flot de compilation

L'extension OPTIM est intégrée après la vérification contextuelle et avant la génération de code IMA.

```
Code source Deca
  -> Analyse lexicale et syntaxique
  -> Construction de 'lAST
  -> Vérification contextuelle
  -> OPTIM (extension)
  -> Génération de code IMA
```

Ce choix garantit que :

- le programme est syntaxiquement et sémantiquement valide ;
- les optimisations n'introduisent aucune ambiguïté de typage ;
- le comportement observable du programme est préservé.

De manière volontaire, l'extension OPTIM privilégie des optimisations **sûres et localement justifiables**.
Les analyses globales avancées (dominance, SSA complet, analyse interprocédurale) ne sont pas retenues ici, car elles exigent une infrastructure (CFG complet, calculs de points fixes) plus coûteuse et plus risquée dans le cadre du projet Deca. Mais par nature évolutive de notre stratégie un ou plusieurs aspects de ces derniers peut être implémenté si une discussion preuve ça pertinence ou sa simplicité.

3 Techniques d'optimisation

3.1 Analyse du flot de contrôle — Control Flow Graph (CFG)

3.1.1 Principe général

Un **Control Flow Graph (CFG)** est une représentation graphique des chemins d'exécution possibles d'un programme.

- les nœuds représentent des **blocs basiques** ;
- les arêtes représentent les **transferts de contrôle**.

Un **bloc basique** est une séquence d'instructions :

- exécutée séquentiellement ;
- sans saut interne ;
- avec une seule entrée et une seule sortie.

L'intérêt principal du CFG est de raisonner sur l'**atteignabilité** du code : si un bloc ne peut être atteint depuis l'entrée, alors toutes ses instructions sont du **code mort structurel** (éliminable sans changer la sémantique).

3.1.2 Construction partielle du CFG dans Deca

Dans le cadre du projet Deca, la construction d'un CFG complet (dominance, analyses globales, boucles complexes imbriquées) est hors périmètre.
La construction envisagée est donc **partielle** et repose sur :

- les instructions `if / else` ;
- les boucles `while` ;
- les instructions `return`.

3.1.2.1 Exemple 1 — `if(false)` : bloc inatteignable

```
if (false) {
    a = 1;
}
b = 2;
```

Le CFG associé contient un bloc correspondant au corps du `if`, mais l'arête conditionnelle est **impossible** (`false`).
Le bloc `a = 1` est donc **inatteignable**.

3.1.2.2 Exemple 2 — `while(false)` : boucle triviale

```
while (false) {
    println(42);
}
println(1);
```

Le CFG contient une structure de boucle, mais le test initial est toujours faux : le corps de boucle est inatteignable.
L'optimisation peut supprimer entièrement la boucle (remplacer par une instruction vide) et garder uniquement `println(1)`.

3.1.2.3 Exemple 3 — `return` : code après retour

```
int f() {
    return 3;
    println(99);  // code mort
}
```

Dès qu'un `return` est rencontré, aucun chemin d'exécution ne peut atteindre les instructions suivantes dans le même bloc.
Le CFG (ou un raisonnement séquentiel local) permet donc de marquer `println(99)` comme inatteignable.

3.1.3 Détection de code mort et simplification de l'AST à partir du CFG

Le CFG est utilisé dans OPTIM pour :

- détecter du **code mort structurel** (blocs inatteignables) ;
- éliminer des constructions triviales comme `if(false)`, `if(true)`, `while(false)` ;
- simplifier l'AST avant génération de code.

3.1.3.1 Comment le CFG aide concrètement (approche pragmatique) Dans une version partielle, on peut implémenter une logique simple :

- construire des blocs pour les séquences linéaires ;
- relier les blocs via des arêtes conditionnelles (if/else, while) ;
- faire un parcours depuis l'entrée (DFS/BFS) ;
- marquer les blocs non visités comme **inatteignables**.

3.1.3.2 Pourquoi on ne planifie pas à faire un CFG complet Un CFG complet devient rapidement coûteux dès qu'on ajoute :

- des informations de dominance ;
- une prise en compte exhaustive des cas (boucles imbriquées, retours multiples, etc.).

Dans le cadre du projet Deca, l'objectif est plutôt d'obtenir des gains **sûrs** sur des patterns fréquents, tout en conservant une architecture évolutive.

3.2 Static Single Assignment (SSA)

3.2.1 Présentation et intuition

La forme **Static Single Assignment (SSA)** est une représentation où chaque variable logique est **définie une seule fois**.
L'idée n'est pas d'interdire les réaffectations au niveau du langage source, mais de **renommer** les variables en interne pour rendre chaque dépendance explicite.

Dans un code impératif classique, une variable peut être réassignée plusieurs fois ; le nom de variable mélange alors plusieurs valeurs au cours du temps.
En SSA, chaque affectation crée une nouvelle version, ce qui simplifie plusieurs optimisations (propagation de constantes, détection d'affectations inutiles, analyse de dépendances).

3.2.2 Motivation

```
x = 1;
x = x + 2;
x = x * 3;
```

Sans transformation, le compilateur doit raisonner sur le "temps' ' : quel `x` est utilisé où ?
En SSA, on sépare les valeurs :

```
x1 = 1
x2 = x1 + 2
x3 = x2 * 3
```

Chaque dépendance de données devient explicite.

3.2.3 Comment SSA aide la propagation de constantes

Considérons :

```
x = 3;
y = x + 2;
z = y + 1;
```

En SSA :

```
x1 = 3
y1 = x1 + 2
z1 = y1 + 1
```

On voit immédiatement que :

- `x1` est une constante ;
- donc `y1` devient une constante (`5`) ;
- donc `z1` devient une constante (`6`).

La propagation de constantes se ramène alors à une substitution locale :
si une définition est constante, tous ses usages peuvent être remplacés par cette constante, ce qui déclenche souvent du **constant folding**.

3.2.4 Détection d'affectations inutiles (dead assignments) via SSA

```
x = 1;
x = 2;
println(x);
```

SSA :

```
x1 = 1
x2 = 2
println(x2)
```

La définition `x1` n'a **aucun usage** : c'est une affectation inutile, supprimable.
Ce raisonnement est particulièrement simple en SSA car chaque définition correspond à une valeur unique.

3.2.5 Limitation et choix : SSA-lite

Dans une première réflexion, Notre extension OPTIM ne va pas mettre en œuvre un SSA complet au sens classique :

- pas de calcul de dominance ;
- pas de renommage global sur l'ensemble du CFG ;
- pas de fonctions ϕ (fusion explicite de valeurs à la jonction des chemins).

3.2.5.1 Pourquoi on ne fait pas de SSA complet

Un SSA complet exige :

- un CFG précis et exhaustif ;
- la dominance ce qui nécissite des parcours de graphe CFG plutôt complexes ;
- l'insertion de ϕ aux points de jonction (`if/else`, boucles) ;
- un renommage global cohérent.

Ces éléments sont coûteux à implémenter correctement et augmentent fortement le risque de bugs sémantiques dans un projet pédagogique.

3.2.5.2 Ce que signifie SSA-lite dans OPTIM

Notre OPTIM va adopte une approche **locale et conservatrice** :

- on exploite le principe SSA *là où il est applicable sans ambiguïté* ;
- typiquement dans des blocs linéaires et des patterns simples ;
- dès qu'une variable peut provenir de plusieurs chemins, on s'abstient (approche conservative).

Ainsi, SSA-lite permet néanmoins :

- la propagation de constantes sur des chemins simples ;
- la détection d'affectations inutiles dans des séquences linéaires ;
- un raisonnement local sur les dépendances de données.

Mais nous re-insistons sur le fait que ça peut changer selon notre avancement et compréhension de plus en plus approfondie de ces outils.

3.3 Optimisations sur l'AST

3.3.1 Constant Folding

Le **constant folding** consiste à évaluer à la compilation les expressions composées uniquement de constantes.

```
3 + 4         // devient 7
2 * (5 - 1)   // devient 8
```

3.3.1.1 Pourquoi c'est pertinent

Cette optimisation :

- réduit le nombre d'instructions générées (moins d'opérations arithmétiques) ;
- diminue le coût à l'exécution ;
- est sémantiquement sûre (aucun effet de bord dans l'expression).

3.3.1.2 Comment l'appliquer dans l'architecture Deca

Dans Deca, les expressions sont représentées par des classes dérivées de `AbstractExpr` (par exemple `AbstractOpArith` et donc `Plus`, `Minus`, `Multiply`, etc.).
L'approche pragmatique consiste à ajouter une méthode d'optimisation sur les expressions, par exemple :

- une méthode `optimizeExpr()` dans `AbstractExpr` ou dans ces sous-classes les plus pertinenetes ;
- chaque sous-classe applique récursivement l'optimisation à ses opérandes ;
- si les deux opérandes sont des littéraux (`IntLiteral`, `FloatLiteral`, `BooleanLiteral`), on calcule le résultat et on remplace le nœud par un littéral.

3.3.1.3 Lien avec SSA / propagation de constantes

La propagation de constantes (via SSA-lite ou une table locale des valeurs) transforme des variables en constantes dans l'AST.
Cela rend davantage d'expressions éligibles au **constant folding**.

3.3.2 Simplification algébrique

La simplification algébrique repose sur des identités valides pour tout `x` (sans nécessiter que `x` soit une constante).
Elle permet de simplifier l'AST et donc de générer du code plus court.

```
x + 0  -> x
0 + x  -> x
x * 1  -> x
1 * x  -> x
x - 0  -> x
x * 0  -> 0
0 * x  -> 0
```

3.3.2.1 Principe On détecte des motifs locaux dans l'AST et on remplace l'expression par une forme équivalente plus simple.
Cette optimisation est également sûre tant qu'on reste sur des expressions sans effets de bord (ce qui est le cas ici pour l'arithmétique pure).

3.3.2.2 Pourquoi on reste prudent Dans un langage plus riche, certaines expressions peuvent avoir des effets de bord (appels de méthodes, accès mémoire complexe).
Dans Deca, l'arithmétique est simple, mais il reste prudent de ne simplifier que des patterns clairement sûrs (littéraux et opérations arithmétiques standards).

3.3.3 Suppression de code mort

La suppression de code mort vise à supprimer du code qui n'a aucun impact sur le résultat observable du programme.
On distingue deux cas complémentaires :

- **code mort structurel** : instructions/blocs inatteignables (via CFG) ;
- **code mort par données** : affectations dont la valeur n'est jamais utilisée.

3.3.3.1 Exemples (structurels via CFG)

```
if (false) {
    a = 1;     // mort (bloc inatteignable)
}
println(2);
```

```
while (false) {
    println(99); // mort (corps inatteignable)
}
println(1);
```

```
int f() {
    return 3;
    println(99);  // mort (apres return)
}
```

3.3.3.2 Exemples (par données / affectations inutiles)

```
int x;
x = 1;    // mort si x n'est jamais lu ensuite
x = 2;
println(x);
```

```
int y;
y = 5;    // mort si y n'est jamais utilisé
println(1);
```

3.3.3.3 Pourquoi on ne fait pas une analyse globale complète

Une élimination de code mort parfaite exige généralement :

- une analyse de “live variables” (analyse de flot de données) ;
- un CFG complet ;
- parfois une analyse interprocédurale (appels de fonctions).

Dans Notre OPTIM (à ce moment de l’écriture de cette spéc), on se limite à des cas sûrs :

- code mort structurel évident (if/while constants, après return) ;
- affectations inutiles détectables localement (SSA-lite / suivi local des usages).

3.4 Peephole Optimization

3.4.1 Principe

La peephole optimization est une optimisation locale appliquée directement au code IMA, typiquement en analysant une fenêtre de 1 à quelques instructions consécutives.

```
LOAD #0 R1
ADD R2 R1
```

Après optimisation (exemple illustratif) :

```
LOAD R2 R1
```

3.4.1.1 Autres patterns classiques (illustratifs)

- suppression de `LOAD` redondants ;
- élimination de `ADD #0`, `MUL #1`, etc. ;
- simplification de séquences `PUSH/POP` inutiles.

3.4.2 Intégration

Cette optimisation est appliquée :

- après la génération du `IMAProgram` ;
- par parcours séquentiel des instructions ;
- indépendamment de l'AST et du CFG.

3.4.2.1 Pourquoi on l'applique après la génération de code Certaines inefficiences n'apparaissent qu'après la sélection d'instructions (codegen).
Le peephole permet donc de "nettoyer' ' le code final, même si l'AST a déjà été simplifié.

4 Architecture logicielle - Fichiers et packages impactés

Création d'un nouveau package :

```
1 fr.ensimag.deca.optim
```

Incluant notamment :

- `Optimizer` (orchestrateur : applique les passes dans l'ordre)
- `ConstantFolder` (réécritures de constantes dans l'AST)
- `AlgebraicSimplifier` (règles algébriques locales)
- `DeadCodeEliminator` (code mort structurel + cas simples par données)
- `PeepholeOptimizer` (réécriture locale du `IMAProgram`)

Des modifications mineures sont prévues dans :

- `AbstractExpr` et ses sous-classes (par ex. `AbstractOpArith`, comparaisons, littéraux) pour offrir un point d'entrée `optimizeExpr()` ;
- certaines instructions de contrôle (if/while/return) pour permettre des simplifications de structure ;
- la phase de génération de code, afin d'insérer l'appel à OPTIM entre la vérification contextuelle et la génération IMA.

L'architecture est conçue pour rester **évolutive** : de nouvelles passes pourront être ajoutées (ou rendues plus globales) sans modifier l'API principale du compilateur.

5 Limitations et perspectives

Les optimisations décrites sont volontairement limitées par :

- les contraintes pédagogiques du projet ;
- l'absence d'analyses globales complètes (dominance, SSA complet) ;
- la volonté de conserver des transformations **sûres** et **justifiables localement**.

Perspectives d'évolution possibles :

- enrichir le CFG (blocs basiques plus précis, retours multiples, etc.) ;
- ajouter une analyse de "variables vivantes" pour un dead code plus complet ;
- implémenter un SSA plus formel (dominance, ϕ), si le périmètre le permet.

6 Conclusion

L'extension OPTIM propose une première approche structurée de l'optimisation dans le compilateur Deca.
Elle s'appuie sur des concepts fondamentaux des compilateurs tout en adoptant une mise en œuvre **pragmatique et progressive**.

Ce document constitue une **référence conceptuelle et bibliographique**, destinée à guider l'implémentation de l'extension.