École normale supérieure de Lyon

Internship Report

CFG Patterns: A new tool to formally verify optimisations in Vellvm

Leon Frenot

supervised by
Yannick Zakowski & Gabriel Radanne
at ENS Lyon

February 5th, 2024 - July 5th, 2024

Contents

Abstract

Abstract

1 Introduction

Debut intro: M2, 20 semaines, LIP, CASH. Yannick Zakowski & Gabriel Radanne. Goal.

Compilation certifiée AJD

Importance de la compilation certifiée, et surtout de certifier les optims.

The Contribution of This Work

- Design d'un langage de patterns + Implémentation naive d'un matcher
- Preuve d'un théorème central pour prouver des optims (sur un CFG)
- Utiliser ce langage pour deux optims + preuves de correction

Premier exemple: CCstP

2 Key concepts

2.1 LLVM and Vellvm

llvm (très rapide)

vellvm: but, niveaux d'interprétation (préciser celui auquel on se place)

- denotational proofs, programmes ouverts → utilisera OCFG pour open CFG
- structure en couche, optimisations qui conservent les traces d'interaction

pourquoi travailler sur vellvm

2.2 ITrees

utilité, coinduction, structure, mechanisme de preuve

3 The pattern language

In this section we will:

- Define a Domain Specific Language that can capture optimizable subgraphs in an OCFG.
- Introduce a matcher on this language and the corresponding semantics of each constructor.
- Present the Coq implementation of the language, matcher and semantics.

3.1 Defining the language

Our goal is to define a Domain Specific Language that can characterize optimizable subgraphs in an OCFG. To represent that language, we define an inductive datatype.

```
Inductive Pattern : Type → Type :=
| Graph: Pattern ocfg
| When: ∀ {S}, Pattern S → (S → bool) → Pattern S
| Map: ∀ {S} {T}, Pattern S → (S → T) → Pattern T
| Focus: ∀ {S}, Pattern S → Pattern (ocfg * S)
| Block: ∀ {S}, Pattern S → Pattern (bid * blk * S)
| Head: ∀ {S}, Pattern S → Pattern (bid * blk * S)
| Branch: ∀ {S}, Pattern S → Pattern (bid * blk * S)
.
```

Figure 1: The **Pattern** datatype

Since the goal of a pattern in to capture a subgraph with a certain structure, the **Pattern** datatype has a type argument, which represents the return type of the pattern.
Each constructor adds to the return types of the following constructors, with the base case Graph accepting any graph.

We will now introduce each constructor and their function.

Graph The Graph constructor is the "base" case that matches any graph. It does not take any extra argument, and returns the graph given as argument.

When The When constructor allows adding a boolean condition to a pattern. It takes a pattern and a corresponding boolean function as argument, and returns what the patterns matched if it fulfils the condition.

Map The Map constructor allows mapping a function onto a pattern's return type. It takes a pattern and a function as argument, and returns the image of the function by what the patterns matched.

Focus The Focus constructor matches any subgraph. It takes a pattern as argument to match against the rest of the graph, and returns the matched subgraph and what the pattern matched.

Block The Block constructor matches any single block in the graph. It takes a pattern as argument to match against the rest of the graph, and returns the matched block and what the pattern matched.

Head The Head constructor matches any block of the graph without predecessors. It takes a pattern as argument to match against the rest of the graph, and returns the matched block and what the pattern matched.
Note that this constructor could not be directly implemented as a `When (Block _) _` since it depends on the rest of the graph, which When wouldn't have access to.

Branch The `Branch` constructor matches any block of the graph whose terminator is a conditional jump. It takes a pattern as argument to match against the rest of the graph, and returns the matched block and what the pattern matched. This constructor could be implemented as a `When (Block _) _`, but has been implemented directly because ???.

Pattern examples

With these constructors, we can build patterns that characterize subgraphs.

For example, if we want to capture a block for Conditional Constant Propagation, we can use the pattern `When (Block Graph) f` with `f` a boolean function that takes a block as argument returns true if the block's terminator is a branch with a condition that evaluates to a constant. That pattern has type `Pattern (bid*blk*ocfg)`.

3.2 Matcher functions

To use these patterns, we need to define a matcher function. That is, a function that takes a pattern and an OCFG as argument, and returns a subgraph, or each subgraph, that matches that pattern.

We implemented the `MatchAll` function, which returns all the subgraphs corresponding to a given pattern.

```
Definition flat_map_r {A B C} (f : B → list C) :=
  fix flat_map_r (l : list (A*B)) : list (A*C) :=
    match l with
    | [] ⇒ []
    | (a, b)::q ⇒ (map (λ c ⇒ (a, c)) (f b))++flat_map_r q
end.
```

Figure 2: The `flat_map_r` function

```
Fixpoint MatchAll {S} (P: Pattern S) (g: ocfg) : list S :=
match P with
  | Graph ⇒ [g]
  | When p f ⇒ filter (λ x ⇒ f x = true) (MatchAll p g)
  | Map p f ⇒ map f (MatchAll p g)
  | Focus p ⇒ flat_map_r (MatchAll p) (focus g)
  | Block p ⇒ flat_map_r (MatchAll p) (blocks g)
  | Head p ⇒ flat_map_r (MatchAll p) (heads g)
  | Branch p ⇒ flat_map_r (MatchAll p) (branches g)
end.
```

Figure 3: The `MatchAll` function

`MatchAll` relies heavily on the `flat_map_r` function, of type `∀ A B C, (B → list C) → list (A * B) → list (A * C)`. It applies the rest of the pattern to the right-hand side of what the current constructor's application returns.

`flat_map_r` is characterized by the following lemma:

```
Lemma in_flat_map_r {A B C}:
  ∀ (f:B→ list C) (l:list (A*B)) (a:A) (c:C), (a,c) ∈ (flat_map_r f l) ↔
  ∃ b, (a,b) ∈ l ∧ c ∈ (f b).
```

With this, we can have a correctness and completeness proof for applying MatchAll to each constructor.

Graph The following theorem shows that `MatchAll` correctly identifies the `Graph` pattern.

```
Theorem Pattern_Graph_correct: ∀ G G', G' ∈ (MatchAll Graph G) ↔ G' = G.
```

The proof is immediate from unfolding the definition of `MatchAll Graph`.

When The following theorem shows that `MatchAll` correctly identifies the `When` pattern.

```
Theorem Pattern_When_correct {S}:
  ∀ (P: Pattern S) f X G,
  X ∈ (MatchAll (When P f) G) ↔ f X = true ∧ X ∈ (MatchAll P G).
```

The proof is immediate from the lemma `elem_of_list_filter` given by the `stdpp` library:

```
elem_of_list_filter :
  ∀ {A : Type} (P : A → Prop) {H : ∀ x : A, Decision (P x)} (l : list A) (x : A),
  x ∈ filter P l ↔ P x ∧ x ∈ l
```

Map The following theorem shows that `MatchAll` correctly identifies the `Map` pattern.

```
Theorem Pattern_Map_correct {S T}:
  ∀ (P: Pattern S) (f: S → T) X G,
  X ∈ (MatchAll (Map P f) G) ↔ ∃ y, X = f y ∧ y ∈ (MatchAll P G).
```

The proof is immediate from the lemma `elem_of_list_fmap` given by the `stdpp` library:

```
elem_of_list_fmap :
  ∀ {A B : Type} (f : A → B) (l : list A) (x : B),
  x ∈ map f l ↔ ∃ y : A, x = f y ∧ y ∈ l
```

Focus `MatchAll` relies on the `focus` function to match the `Focus` constructor.

```
Fixpoint focus_rec l (g1 g2: ocfg) :=
  match l with
  | [] ⇒ [(g1, g2)]
  | (id,b)::q ⇒ focus_rec q g1 g2 ++ focus_rec q (delete id g1) (<[id:=b]> g2)
end.

Definition focus (G: ocfg) := focus_rec (map_to_list G) G ∅.
```

The semantic of `Focus` is characterized by the following definition:

```
Record focus_sem (G G1 G2: ocfg): Prop := {
  SUB1: G1 ⊆ G;
  SUB2: G2 ⊆ G;
  PART: G1 ## G2;
  CUP: G1 ∪ G2 = G
}.
```

The correctness of `MatchAll` for `Focus` is proven by the following theorem:

```
Theorem Pattern_Focus_correct {S}:
  ∀ (G G1: ocfg) (P:Pattern S) X,
    (G1, X) ∈ (MatchAll (Focus P) G) ↔
    ∃ G2, focus_sem G G1 G2 ∧ X ∈ (MatchAll P G2).
```

The proof of `Pattern_Focus_correct` relies on the following lemma, which has not been proven:

```
Lemma focus_correct: ∀ G, ∀ G1 G2, (G1, G2) ∈ (focus G) ↔ focus_sem G G1 G2.
```

Block `MatchAll` relies on the `blocks` function to match the `Block` constructor:

```
Definition blocks_aux (G: ocfg) : (bid*blk) → (bid*blk*ocfg) :=
  λ '(id, b) ⇒ (id, b, delete id G).

Definition blocks (G: ocfg): list (bid*blk*ocfg) :=
  map (blocks_aux G) (map_to_list G).
```

The semantic of `Block` is characterized by the following definition:

```
Record blocks_aux_sem (G0 G G': ocfg) id b : Prop :=
  {
    EQ: G' = delete id G0;
    IN: G !! id = Some b
  }.

Definition blocks_sem (G G': ocfg) id b := blocks_aux_sem G G G' id b.
```

The correctness of `MatchAll` for `Block` is proven by the following theorem:

```
Theorem Pattern_Block_correct {S}:
  ∀ (G: ocfg) (P: Pattern S) id (b:blk) X,
  (id, b, X) ∈ MatchAll (Block P) G ↔
  ∃ G', blocks_sem G G' id b ∧ X ∈ MatchAll P G'.
```

The proof of `Pattern_Block_correct` relies on the following lemmas:

```
Lemma blocks_aux_correct: ∀ G G0 G' id b,
  (id, b, G') ∈ map (blocks_aux G0) (map_to_list G) ↔ blocks_aux_sem G0 G G' id b.


Lemma blocks_correct: ∀ G G' id b,
  (id, b, G') ∈ blocks G ↔ blocks_sem G G' id b.
```

Head `MatchAll` relies on the `heads` function to match the `Head` constructor:

```
Definition is_empty (S: gset bid) := decide (S = ∅).

Definition heads_aux (G: ocfg) id b acc : list (bid*blk*ocfg) :=
  if is_empty (predecessors id G)
  then (id, b, delete id G)::acc
  else acc.

Definition heads (G: ocfg): list (bid*blk*ocfg) := map_fold (heads_aux G) [] G.
```

The semantic of `Head` is characterized by the following definition:

```
Record heads_aux_sem (G0 G G': ocfg) id b := {
  EQ: G' = delete id G0;
  IN: G !!id = Some b;
  PRED: predecessors id G0 =  ∅
}.

Definition heads_sem (G G':ocfg) (id:bid) b := heads_aux_sem G G G' id b.
```

The correctness of `MatchAll` for `Head` is proven by the following theorem:

```
Theorem Pattern_Head_correct {S}:
  ∀ (G: ocfg) (P: Pattern S) id b X,
  (id, b, X) ∈ (MatchAll (Head P) G) ↔
  ∃ G', heads_sem G G' id b ∧ X ∈ (MatchAll P G').
```

The proof of `Pattern_Head_correct` relies on the following lemmas:

```
Definition heads_aux_P G0 (s:list (bid*blk*ocfg)) G :=
  ∀ id b G', (id, b, G') ∈ s ↔ heads_aux_sem G0 G G' id b.

Lemma heads_aux_correct:
  ∀ G G0,
  heads_aux_P G0 (map_fold (heads_aux G0) [] G) G.

Lemma heads_correct:
  ∀ G G' id b,
  (id, b, G') ∈ (heads G) ↔ heads_sem G G' id b.
```

Branch `MatchAll` relies on the `branches` function to match the `Branch` constructor:

```
Definition branches_aux (G: ocfg) id b acc : list (bid*blk*ocfg) :=
  match b.(blk_term) with
    | TERM_Br _ l r ⇒ (id, b, (delete id G))::acc
    | _ ⇒ acc
  end.

Definition branches (G: ocfg): list (bid*blk*ocfg) :=
  map_fold (branches_aux G) [] G.
```

The semantic of `Branch` is characterized by the following definition:

```
Record branch_aux_sem (G0 G G': ocfg) id b := {
  EQ: G' = delete id G0;
  BR: ∃ e l r, b.(blk_term) = TERM_Br e l r;
  IN: G !! id = Some b
}.

Definition branch_sem G G' id b := branch_aux_sem G G G' id b.
```

The correctness of `MatchAll` for `Head` is proven by the following theorem:

```
Theorem Pattern_Branch_correct {S}:
  ∀ G P B id (s:S),
  (id,B,s) ∈ (MatchAll (Branch P) G) ↔
  ∃ G', branch_sem G G' id B ∧ s ∈ (MatchAll P G').
```

The proof of `Pattern_Branch_correct` relies on the following lemmas:

```
Definition branches_aux_P G0 (s:list (bid*blk*ocfg)) G :=
  ∀ id b G', (id, b, G') ∈ s ↔ branch_aux_sem G0 G G' id b.

Lemma branches_aux_correct:
  ∀ G G0,
  branches_aux_P G0 (map_fold (branches_aux G0) [] G) G.

Lemma branches_correct:
  ∀ G G' id b,
  (id,b,G') ∈ (branches G) ↔ branch_sem G G' id b.
```

4 Cas d'étude: Block Fusion

In this section, we will define the Block Fusion optimization, describe a corresponding OCFG pattern, and outline the proof of correctness of the optimization using the pattern.

4.1 motivation for Block Fusion

The Block Fusion optimization consists of picking two blocks A and B, such that A is the only predecessor of B and B is the only successor of A, and replacing them with a single block containing the code of A and B.

This optimization is relevant for three main reasons:

- It is a commonly used optimization, usually to clear blocks created by others.
- It is an optimization that modifies the graph.
- It is simple to prove on paper that the optimization is correct.

4.2 the `BlockFusion` pattern

The `BlockFusion` pattern is defined in the code as `When (Block (Head _)) BlockFusion_f`. `BlockFusion_f` is a boolean function which will be talked about in more detail in the following subsection.
`Block Head _` identifies two blocks A and B, such that B doesn't have predecessors in the graph with A removed. `BlockFusion_f` gives additional conditions on A and B such that Block Fusion will be correct.

For this pattern we can define a corresponding semantic, which we'll then use to prove the correctness of the optimization.

4.3 preuve de correction Théorème: `denote_ocfg_equiv`

défis d'interprétation (φ & term), renomage
exemple plus précis d'une preuve par coinduction

intro

4.4 defis

présentation des défis que pose le formalisme + le niveau d'interprétation
schéma idée de base → problèmes noms et phi → hypothèses etc.

4.5 hypothèses

- hyp principale: `denote_ocfg_equiv_cond`
- hyps sur nTO et nFROM ! Schémas
- `dom_renaming`

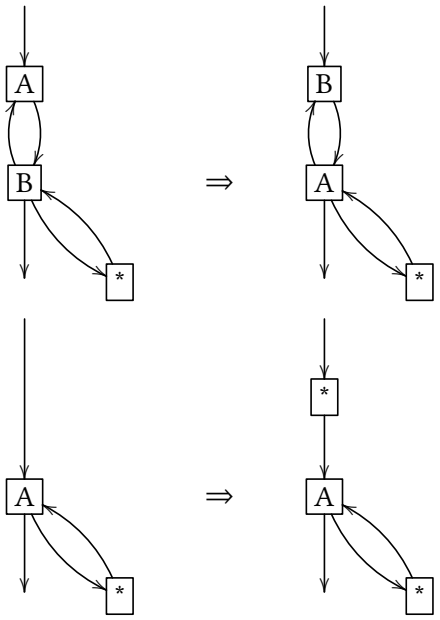
4.6 lemmes

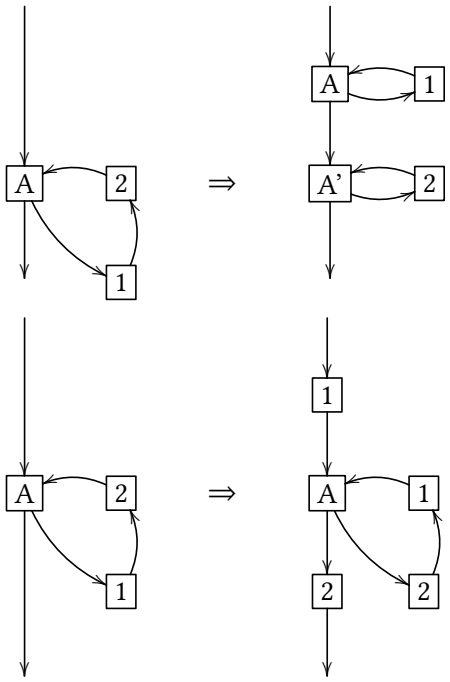
- `bk_phi_rename_eutt`

5 implémentation (+ raison pour s'arrêter à naïve (rapide))

6 A voir: Approfondissements

6.1 Loop pattern





6.2 Other interpretation levels

6.3 Optim efficace

Conclusion