

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

1	Introduction	2
2	Key concepts	2
2.1	LLVM and Vellvm	2
2.2	ITrees	2
3	The pattern language	2
3.1	Defining the language	3
3.2	Matcher functions	4
4	Cas d'étude: Block Fusion	6
4.1	motivation for Block Fusion	6
4.2	preuve de correction Théorème: denote_ocfg_equiv	7
4.3	defis	7
4.4	hypothèses	7
4.5	lemmes	7
5	implémentation (+ raison pour s'arrêter à naïve (rapide))	7
6	A voir: Approfondissements	7
6.1	Loop pattern	7
6.2	Other interpretation levels	8
6.3	Optim efficace	8

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, mécanisme 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 is 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 example

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

For example, we want to capture a subgraph for the **BlockFusion** fusion optimization. That is: fusing two blocks whose execution always follow each other into a single block.

We can recognize the applicable subgraphs with the pattern `When (Block (Head Graph)) BlockFusion_f`. **Block** matches any first block, then **Head** matches a block that has no predecessors (except possibly **Block**), and finally `When _ BlockFusion_f` sets additional conditions on the two blocks for the optimization.

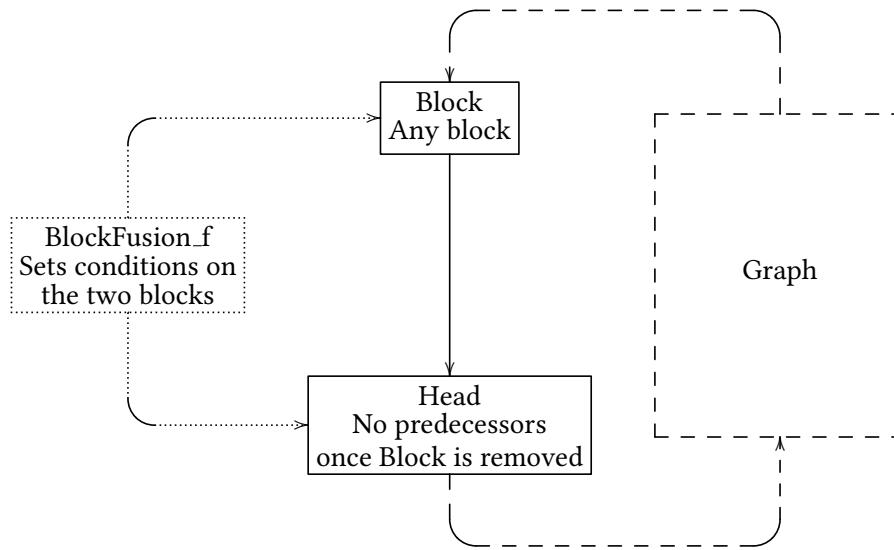


Figure 2: The **BlockFusion** pattern

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.

```

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

```

Figure 3: The MatchAll function

```

Definition flat_map_r {A B C} (f : B  $\rightarrow$  list C) :=
  fix flat_map_r (l : list (A*B)) : list (A*C) :=
    match l with
    | []  $\Rightarrow$  []
    | (a, b)::q  $\Rightarrow$  (map ( $\lambda c \Rightarrow (a, c)$ ) (f b)) ++ flat_map_r q
  end.

```

Figure 4: The flat_map_r function

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

Proving the correctness for Graph, When and Map is immediate thanks to builtin lemmas on filter and map.

The proof mechanism for Block, Head and Branch are similar. We will now detail it for Head.

MatchAll relies on the heads function to match the Head constructor.

The goal of that function is to find all the "heads", i.e. blocks without predecessors, in an OCFG. To do that, it folds a heads_aux function over the map. That function calls the predecessors function on each block, and appends the result to the return list if the block doesn't have predecessors.

```

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.

```

Figure 5: The heads function

With these function, we can define the semantics corresponding to each function. We have to define them first for the auxiliary function for the semantics proof.

```

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

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

```

Figure 6: The semantic definition for Head/heads

Finally, we can prove the semantics for the auxiliary function, the `heads` function and `MatchAll Head`.

```

Definition heads_aux_P G0 (s:list (bid*blk*ocfg)) G :=
   $\forall$  id b G', (id, b, G')  $\in$  s  $\leftrightarrow$  heads_aux_sem G0 G G' id b.

Lemma heads_aux_correct:
   $\forall$  G G0,
  heads_aux_P G0 (map_fold (heads_aux G0) [] G) G.

Lemma heads_correct:
   $\forall$  G G' id b,
  (id, b, G')  $\in$  (heads G)  $\leftrightarrow$  heads_sem G G' id b.

Theorem Pattern_Head_correct {S}:
   $\forall$  (G: ocfg) (P: Pattern S) id b X,
  (id, b, X)  $\in$  (MatchAll (Head P) G)  $\leftrightarrow$ 
   $\exists$  G', heads_sem G G' id b  $\wedge$  X  $\in$  (MatchAll P G').

```

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.

In the previous section, we already gave a pattern for `BlockFusion`, we will use a slight variation, which allows further composing:

```

Definition BlockFusion S (P: Pattern S) := When (Block (Head P)) BlockFusion_f.

```

4.2 preuve de correction Théorème: `denote_ocfg_equiv`

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

intro

4.3 defis

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

4.4 hypothèses

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

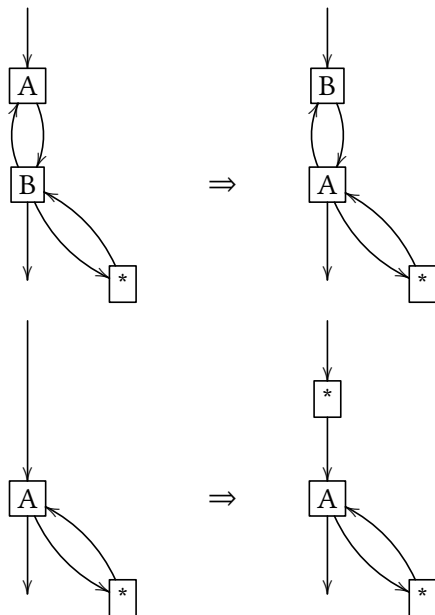
4.5 lemmes

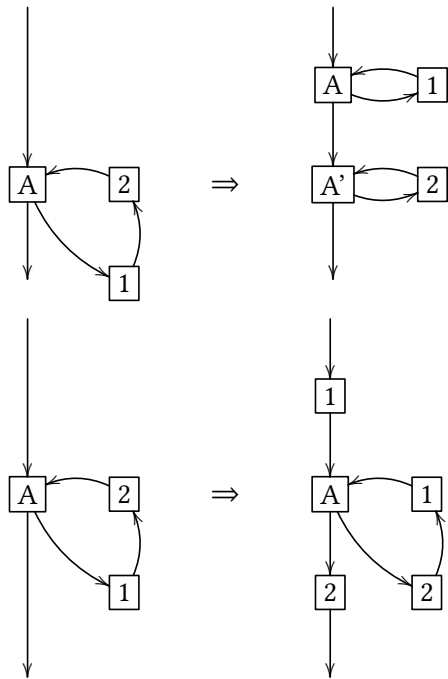
- `bk_phi_rename_eutt`

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