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 $\longrightarrow$ 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 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.

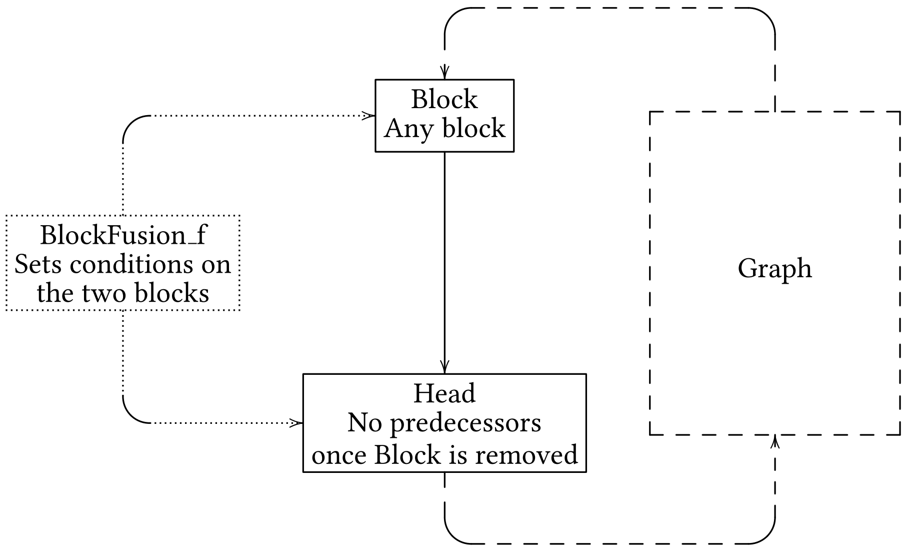


Figure 2: The `BlockFusion` pattern

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 ⇒ [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

```
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 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 =  ∅
}.

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 :=
  ∀ 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.

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').
```

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

Before implementing and proving an implementation for Block Fusion, we first established a larger theorem for optimizations that modify the control flow graph.
The goal of that theorem is to allow replacing a part of the OCFG with an equivalent subgraph. And it is as follows:

```
Theorem denote_ocfg_equiv
  (g1 g2 g2' : ocfg) (σ  : bid_renaming) (nFROM nTO: gset bid) :
    inputs g2 ∩ inputs g2' ## nFROM → nFROM ⊆ inputs g2 ∪ inputs g2' →
    inputs g2' \ inputs g2 ⊆ nTO → nTO ⊆ inputs g2 ∪ inputs g2' → nTO ## outputs g1
        →
    g1 ## g2 → ocfg_term_rename σ g1 ## g2' →
    dom_renaming σ nFROM g2 g2' →
    denote_ocfg_equiv_cond g2 g2' nFROM nTO σ →
    ∀ from to' to,
    to' = σ to →
    to ∉ nTO → from ∉ nFROM →
    ⟦g2 ∪ g1⟧bs (from,to) ≈ ⟦g2' ∪ ocfg_term_rename σ g1⟧bs (from, to').
```

Figure 7: The denote_ocfg_equiv theorem

We will now detail this theorem.

4.2.1 Arguments

The theorem takes 6 arguments:

- g1: The part of the graph that doesn't change[1].
- g2 and g2': The part of the graph that gets replaced, and its replacement.
- σ: a bid → bid function that maps the inputs of g2 over the inputs of g2'.
- nFROM and nTO: Sets of block ids from/to which the execution would not be equivalent.

4.2.2 Conclusion

The conclusion of the theorem is at follows:

```
    ∀ from to' to,
    to' = σ to →
    to ∉ nTO → from ∉ nFROM →
    ⟦g2 ∪ g1⟧bs (from,to) ≈ ⟦g2' ∪ ocfg_term_rename σ g1⟧bs (from, to').
```

When replacing g2 with g2', a discrepancy in block ids may appear. For example, when doing Block Fusion, we go from entering with one id and exiting with another, to entering and exiting with the same.

[1]As can be seen in the figure, it does change slightly, we will discuss this later

Because of that, we need to do some renaming in the rest of the graph to keep the same semantics after the optimization.
There are two possibilities, either we change the phi-nodes (and block fusion gets the id of the input block), or we change the terminators (and block fusion gets the id of the output block). Since the semantics of terminators is simpler, we decided to do the latter.

`ocfg_term_rename` then appliesσ to each block id in the terminator of each block of `g1`.

4.2.3 denote_ocfg_equiv_cond

`denote_ocfg_equiv_cond` is the equivalence condition on `g2` and `g2'`. It is straightforward.

```
Definition denote_ocfg_equiv_cond
  (g g': ocfg) (nFROM nTO :gset bid) (σ : bid → bid) :=
   ∀ origin header,
     header ∉ nTO →
     origin ∉ nFROM →
     ⟦g⟧bs (origin, header) ≈ ⟦g'⟧bs (origin, σ header).
```

4.2.4 dom_renaming

`dom_renaming` is the condition that fixes σ's domain.
It has two conditions:

- If an id is in `g2`, σ id is in `g2'`.
- If an id not in `nFROM`, it is a fixpoint.

```
Record dom_renaming (σ  : bid_renaming) (nFROM : gset bid) (g g': ocfg) : Prop :=
  {
    in_dom : ∀ id, id ∈ inputs g → (σ  id) ∈ inputs g';
    out_dom : ∀ id, id ∉ nFROM → (σ  id) = id
  }.
```

4.2.5 Conditions on nTO

`inputs g2' \ inputs g2 ⊆ nTO` If we add a block from `g2` to `g2'`, the execution would be different if we enter the graph though that block.

`nTO ⊆ inputs g2 ∪ inputs g2'` If we enter somewhere that is unchanged, the execution doesn't change.

`nTO ## outputs g1` When we exit `g1` into `g2` or `g2'`, the execution needs to still be the same.

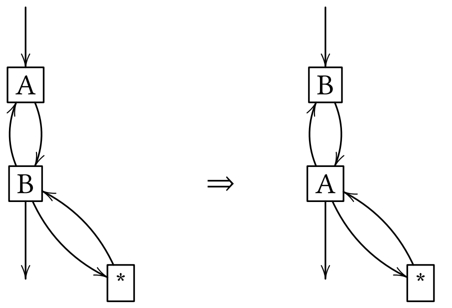
4.2.6 Conditions on nFROM

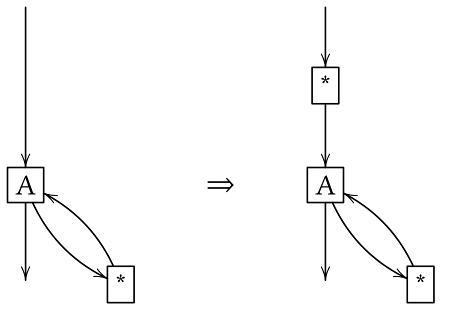
`inputs g2 ∩ inputs g2' ## nFROM` If we come from somewhere that is in both g2 and g2', the execution should be valid.

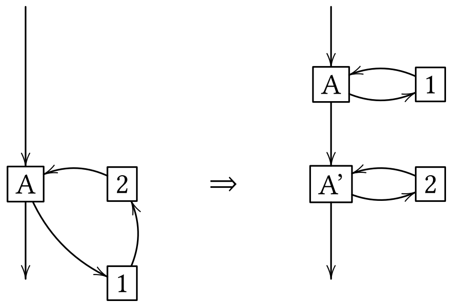
`nFROM` ⊆ `inputs g2` ∪ `inputs g2'` If we come from somewhere that was not changed, the execution should still be valid.

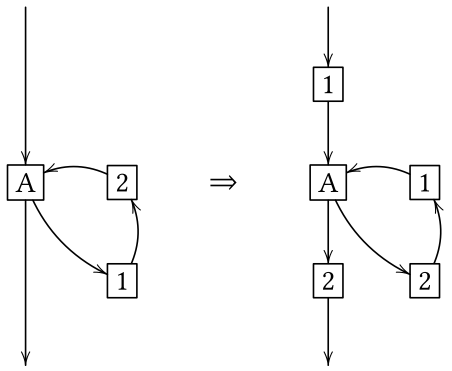
5 A voir: Approfondissements

5.1 Loop pattern









5.2 Other interpretation levels

5.3 Optim efficace

Conclusion