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 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.

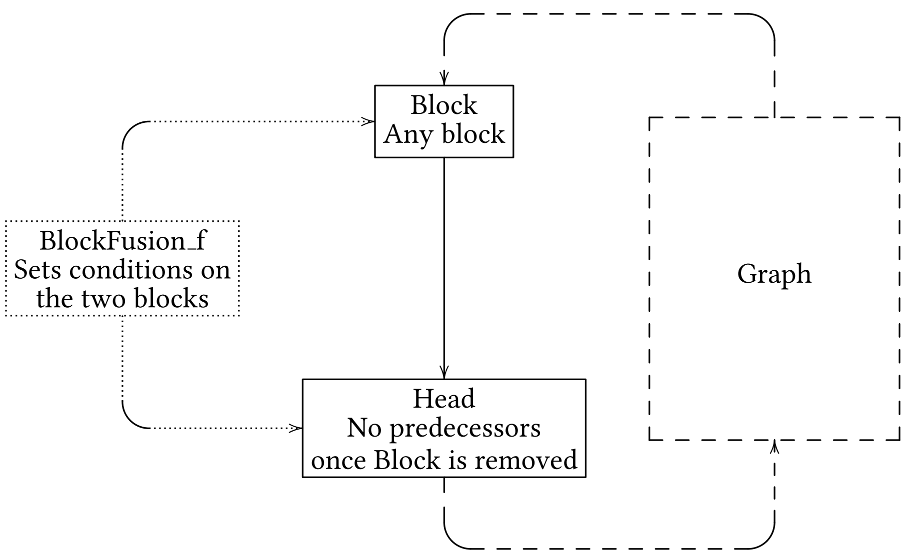


Figure 2: The BlockFusion pattern

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 Denotation

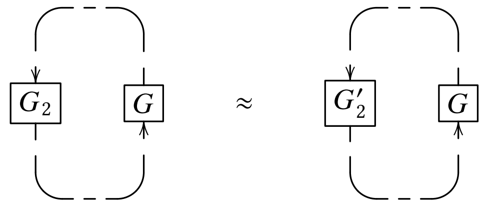
In this section we will informally define an optimization class, show a theorem for proving the correctness of optimizations of that class, and apply this theorem to an implementation of Block Fusion.

4.1 An optimization class

Since the goal of the patterns is to identify subgraphs, we want to focus on optimizations that only modify a section of the graph. (As opposed to ones that may modify everything, like constant propagation.)
Ideally, we want to be able to replace any subgraph with an equivalent subgraph.

However, this ideal theorem is not enough. In the case of Block Fusion for example, since we replace two blocs by one, the change in ids means that either we have to enter by different ids, or we have to exit by different ids.
There needs to be some renaming. We chose to apply the renaming to `to` and to `g1`'s terminators, since that keeps the semantics equivalent.

$G_2 \quad \approx \quad G'_2 \quad \Longrightarrow$




```
Theorem (g1 g2 g2' : ocfg):
  ∀ from to, ⟦g2⟧bs (from,to) ≈ ⟦g2'⟧bs (from, to) →
  ∀ from to, ⟦g2 ∪ g1⟧bs (from,to) ≈ ⟦g2' ∪ g1⟧bs (from, to).
```

We define a function `ocfg_term_rename` which, given a function over ids σ and a graph `g`, returns `g` with σ applied to each id in its blocks' terminators.
This new function gives us the following theorem:

```
Theorem (g1 g2 g2' : ocfg) (σ : bid → bid):
  ∀ from to, ⟦g2⟧bs (from,to) ≈ ⟦g2'⟧bs (from, σ to) →
  ∀ from to, ⟦g2 ∪ g1⟧bs (from,to) ≈ ⟦g2' ∪ ocfg_term_rename σ g1⟧bs (from, σ to).
```

However, this still cannot be applied to Block Fusion. Indeed, if we try to start on the second block, the semantics are obliviously different.
Similar issues can come from having an incorrect origin block. So we have to introduce two sets of ids `nTO` and `nFROM` to set condition on the input and origin ids.

```
Theorem (g1 g2 g2' : ocfg) (σ : bid → bid) (nFROM nTO: gset bid):
  (∀ from to, to ∉ nTO → from ∉ nFROM → ⟦g2⟧bs (from,to) ≈ ⟦g2'⟧bs (from, σ to)) →
  ∀ from to, to ∉ nTO → from ∉ nFROM → ⟦g2 ∪ g1⟧bs (from,to) ≈ ⟦g2' ∪
     ocfg_term_rename σ g1⟧bs (from, σ to).
```

Finally, we need some conditions to make sure that:

- the unions are well-formed,
- `nFROM` and `nTO` are preserved during the (coinductive) proof,
- σ only changes ids from `g2` to `g2'`.

These conditions give us the following final theorem:

```
Theorem denote_ocfg_equiv
  (g1 g2 g2' : ocfg) (σ  : bid → bid) (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' →
    (∀  id, id ∈ inputs g2 → (σ  id) ∈ inputs g2') →
    (∀  id, id ∉ nFROM → (σ  id) = id) →
    (∀  from to, to ∉ nTO → from ∉ nFROM → ⟦g2⟧bs (from,to) ≈ ⟦g2'⟧bs (from, σ to))
        →
    ∀ from to,
    to ∉ nTO → from ∉ nFROM →
    ⟦g2 ∪ g1⟧bs (from,to) ≈ ⟦g2' ∪ ocfg_term_rename σ g1⟧bs (from, σ to).
```

Figure 7: The denote_ocfg_equiv theorem

4.2 motivation for 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.

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, for example to clear blocks created while building SSA form.
- 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.
```

4.3 Block Fusion for real actually I swear

`BlockFusion_f` has two conditions:

- the terminator of the first block is an absolute jump to the second block,
- the second block does not have phi nodes.

The first condition is needed (instead of just checking the successors) because, if there is a conditional jump, evaluating the condition may lead to an error, and so to a difference in semantic after the fusion.

The second condition is needed because of the difference in evaluation between phi-nodes and assignment operations.

With this, we can create a `fusion` function for Block Fusion (`term_rename` applies σ to each id in the terminator).

```
Definition fusion (σ : bid → bid) (idA : bid) (A B: blk): blk := {|
  blk_phis     := A.(blk_phis);
  blk_code     := A.(blk_code) ++ B.(blk_code);
  blk_term     := term_rename σ B.(blk_term);
  blk_comments := fusion_comments A B
|}.
```

Figure 8: The `fusion` function

We also define σ`fusion`, the renaming function for Block Fusion:

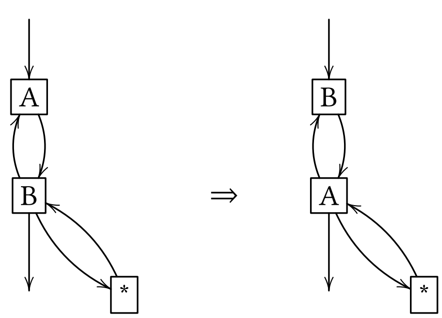
```
Definition σ fusion idA idB := λ(id: bid) ⇒ if decide (id=idA) then idB else id.
```

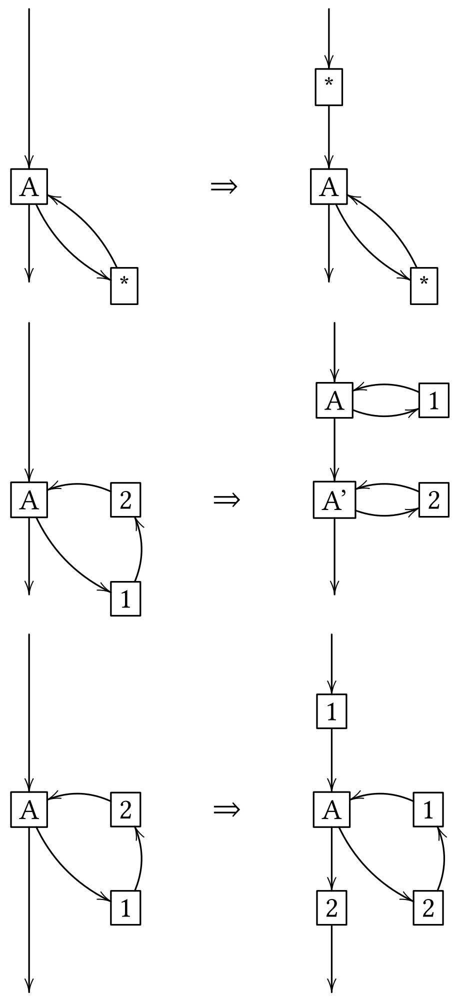
With these, we can prove first that `fusion` is correct, and then that the Block Fusion optimization is correct.

```
Theorem Denotation_BlockFusion_correct {S} G idA A idB B f to P (X:S):
  let σ := σ fusion idA idB in
  let G0 := delete idB (delete idA G) in
  to <> idB →
  f <> idA →
  (idA, A, (idB, B, X)) ∈ (MatchAll (BlockFusion P) G) →
  ⟦ G ⟧bs (f, to) ≈ ⟦<[idB:=fusion σ idA A B]> (ocfg_term_rename σ G0) ⟧bs (f, σ to).
```

5 A voir: Approfondissements

5.1 Loop pattern





5.2 Other interpretation levels

5.3 Optim efficace

Conclusion