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 Background **2**
2.1 Interaction Trees 2
2.2 Vellvm 3

3 The pattern language **4**
3.1 `Pat`: a DSL for pattern matching on graphs 4
3.2 Matcher functions 6

4 Denotation **9**
4.1 An optimization class 9
4.2 Motivations for Block Fusion 10

5 A voir: Approfondissements **12**
5.1 Loop pattern 12
5.2 Other interpretation levels 13
5.3 Optim efficace 13

Abstract

Abstract

1 Introduction

For my M2 year at ENS de Lyon, I completed a 20 weeks internship in the LIP Computer Science laboratory in ENS Lyon. This internship was supervised by Yannick Zakowski and Gabriel Radanne in the Compilation and Analysis, Software and Hardware Laboratory of the LIP. The goal of this internship was to design and implement a pattern language over control flow graphs to provide a framework for formal proofs on optimizations.

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 block fusion + preuve de correction

Premier exemple: CCstP

2 Background

2.1 Interaction Trees

Interaction Trees (ITrees) are a co-inductive structure designed to represent the dynamic behaviors of a computation. The goal of ITrees is to model recursive and effectful programs, including divergent computations.

Figure 1 show the Coq definition of ITrees. An instance of the type features an *event* type `E: Type → Type`, and a return type `R: Type`. The definition uses three constructors: `Ret` corresponds to halting and returning a value of type `R`; `Tau` corresponds to a *silent step*, i.e. an internal computation, followed by computation `t`; and `Vis` which is a *visible event*, it describes an external computation `e` which returns a value of type `A`, and a continuation `k` which depends on that return value.

```
CoInductive itree (E : Type → Type) (R : Type) : Type :=
  | Ret (r : R)
  | Tau (t : itree E R)
  | Vis {A : Type} (e : E A) (k : A → itree E R).
```

Figure 1: The `itree` datatype

To reason over ITrees, we have multiple notions of *bisimulation*. The most relevant one is *weak bisimulation*, noted `t1 ≈ t2`. We say that `t1 ≈ t2` if they return the same value, and have the same visible events. This relation is an equivalence "up-to-Tau" in the sense that we have `Tau t ≈ t` and `t ≈ Tau t`. This equivalence be refined up to a relationship $R : A \to B \to Prop$: we then have `Ret` $a \approx_R$ `Ret` $b \iff R\ a\ b$.

Effects can easily be added or removed from the semantics of an ITree. The `Vis` constructor represents *uninterpreted events*. By defining an *event handler*, semantics are assigned to these events. Interpreting an ITree then consists of folding that handler over the ITree. This allows the semantics of ITrees to be *modular*.

Furthermore, the semantics of ITrees are also *compositional* with the use of *combinators*. For example, `bind: itree E A → (A → itree E B) → itree E B` allows composing ITrees (with the use of continuations). Other combinators include `iter: (A → itree E (A+B)) → (A → itree E B)` to encode the iterations and hide the co-induction (by hiding each body step in a Tau), and `mrec` for mutual-recursive combinators.

Unlike similar projects, which rely on *operational semantics* and simulation diagrams, ITrees rely on *denotational semantics*. That is, it is based on equations that can be used to prove bisimulation. These equations allow the user to reason equationally, hiding the co-inductive reasoning and the definition of the weak bisimulation. The compositionality of the semantics also allow simpler reasoning than operational semantics, since program counter and similar notions are lifted away.

2.2 Vellvm

TODO: figures (vir syntax and intrep stack), emphs

The goal of the Vellvm project is to formally define the semantic of the LLVM IR and construct verified components for that formalization.

LLVM is a compiler infrastructure designed around a language-independent intermediate representation (IR). It is used to develop frontends for programming languages and backends for instruction set architectures.

The LLVM IR is a RISC-like low level instruction set, but also features high-level informations. This duality allows it to represent any program while still permuting analysis and optimizations. It is based on control flow-graphs, with named labels and registers, and guarantees Single Static Assignment (SSA) form, which is key to many static analyses and optimizations. The LLVM IR is statically typed, and features integer-pointer casts. Optimizations ans analysis on the LLVM IR are done through successive analysis and transformation passes.

Vellvm introduces Verified LLVM IR (VIR), a realistic subset of the LLVM IR. Figure **??** shows a subset of VIR's syntax. VIR's semantics are defined with ITrees: each element of VIR's syntax is represented by a corresponding ITree. Each effect (except control flow) is captured by a `Vis` event, which can be interpreted later. This semantic includes many non-trivial features of LLVM IR, including pointers, LLVM's phi-nodes and undefined behaviors.

Since the semantics of a block or set of blocks can be defined without relying on a "complete" CFG, it is possible to use "open control-flow graphs" (OCFG), which is simply a set of blocks without a defined entry point.

Finally, to interpret the semantics of the different effects of its syntax, Vellvm uses a stack of interpreters. It allows to gradually introduce external elements to the semantic (intrinsics, global and local environments, ...). Figure **??** show that stack of interpretation. The final levels split between a *propositional* model, which interprets the non-determinism of LLVM IR's undefined behaviors, and a *executable* model, which implements one of these behaviors.

3 The pattern language

De façon générale, commencer par des bullet points ou des séries de paragraphe est excellent, mais c'est important de lui donner corps en rédigeant dans un second temps un texte cohésif. Je tente une proposition rapide par exemple pour ce chapeau pour illustrer.

Remarque générale : il faut utiliser beaucoup beaucoup de macros quand on écrit du TeX. Par exemple OCFG va apparaître beaucoup, et on peut hésiter sur la façon de le typeset/écrire : macro

Il peut être pratique d'avoir un nom pour ton langage pour pouvoir y référer.

We now turn our attention to the central piece of our contribution: the design of `Pat`, a DSL of patterns for writing and proving correct program transformations. This DSL is composed of two core components. First, an indexed datatype provides a syntax for the user to specify how they wish to decompose an input OCFG. Second, a *matcher* provides a semantics to the language, specifying the valid decompositions associated to each pattern. Finally, we illustrate on an example the definition and semantic characterization of a pattern extracting the heads of a graph, written in our DSL.

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 `Pat`: a DSL for pattern matching on graphs

At a high level, we look for a language allowing the user to characterize and reason about optimizable subgraphs in an OCFG. To this end, we introduce `Pat`, a general, very expressive DSL for pattern matching on graphs. The specific patterns we are interested in from the perspective of compilation will then be expressed in `Pat`.

```
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 2: The **Pattern** datatype

Attention il faut TOUJOURS, équipper les objets flottants d'un label, ET y faire référence dans le texte. Tu as fait le premier, mais pas le second ici :)

Attention, l'idée d'un tel type indexé n'est pas évident pour beaucoup de lecteurs. Il est bon de prendre ton temps pour l'introduire.

Figure ?? introduces `Pat`'s syntax, defined as an inductive datatype **`Pattern`**. Because the purpose of a pattern is to decompose a graph into a certain structure, the **`Pattern`** datatype reflects this intention by taking as argument a type, which represents the return type of the pattern.[1] This typing information is leveraged in the definition of the matcher, introduced in Section 3.2: a pattern of type **`Pattern`** `S` will be matched against elements of type `S`.

Attention ici il est beaucoup plus digeste et élégant d'éviter une telle succession de paragraphes, et plutôt construire du texte. Je commence pour illustrer ce que j'imagine.

Patterns are built out of seven constructors. The only base case is the `Graph` pattern which trivially match any graph and does not perform any decomposition. On more traditional paper presentation, it corresponds to a single hole □.

The six other constructors recursively decompose the graph, typically enriching the return type of the pattern in doing so. The `When` constructor acts as a filter: given a pattern of return type `S`, it builds a pattern with the same return type, but takes as argument a filtering function `S` → `bool` used to restrict the set of matching graphs to those satisfying the condition. The `Map` constructor simply hardcodes functoriality into the datatype, allowing for post-processing the output of a pattern by a pure function.

TODO: finir d'intégrer les paragraphes ci-après dans le texte ci-dessus.

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.

[1] Such families of types are common in dependently typed languages, and are referred to as Generalized Algebraic Data Types in languages such as OCaml or Haskell.

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. TODO: expand on this note, detailing what filter you would want to write and pointing out why you can't precisely. In particular, explain that you could write `When (Block Graph) (λ '(i,bk,g) ⇒ is_head i g)`, but not `λ p ⇒ When (Block p) (λ s ⇒ ???)`, and why it may matter.

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 ???.Give the exact definition in terms of when/block, and indeed justify.

Je pense qu'il faut insérer ici une discussion assez généreuse sur le choix de ce jeu de constructeurs : remarquer qu'il y a de la redondance, en pointant du doigt que tout pattern fixé peut être encodé avec Focus, Graph et When (à vérifier que c'est vrai) par exemple, mais pas des familles de patterns qui composent; remarquer que Map parait assez naturel, mais que l'on en a pas d'usage en ce moment; expliquer que l'on cherche un point de design alliant facilité d'utilisation et facilité de développer la méta-théorie, et que ce n'est pas encore fixé dans le marbre.

An example of pattern: block fusion. We illustrate the use of `Pat` to implement[2] a bloc fusion optimization. That is: fusing two blocks whose execution always follow each other into a single block. Explain the optimisation in a little bit more details (unless you plan on introducing it in more detail earlier in the paper, in which case put a reference to it).

Note: it's almost never a good idea to force a return carriage via a double slash.

The applicable subgraphs are specified with the pattern `When (Block (Head Graph)) BlockFusion_f`. Name the pattern, like `pfusion` for instance, you will want to refer to it later to give its spec. The pattern starts by the `Block` to match any block `bk` (What does first mean here? I think you just mean to say that's the first thing you match on. I would rather give a name to this block, and not refer to it by the name of the pattern!). Then, `Head` matches a block `head` that has no predecessors (except possibly `bk` as it is not in scope of the pattern anymore), and finally `When _ BlockFusion_f` sets additional constraints on the two blocks required for the optimization to be valid.Give the code for BlockFusion_f and explain these additional constraints in details.

Refer to and explain the figure! "Figure 2 illustrate graphically the shape of the graph decompositions that match `pfusion` ..." Use the names suggested above, and explain the meaning of the different arrows.

3.2 Matcher functions

We now turn to the question of defining the semantics of our patterns, via 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.

[2]Or rather, to *specify* such an optimization. We discuss briefly executability in conclusion.

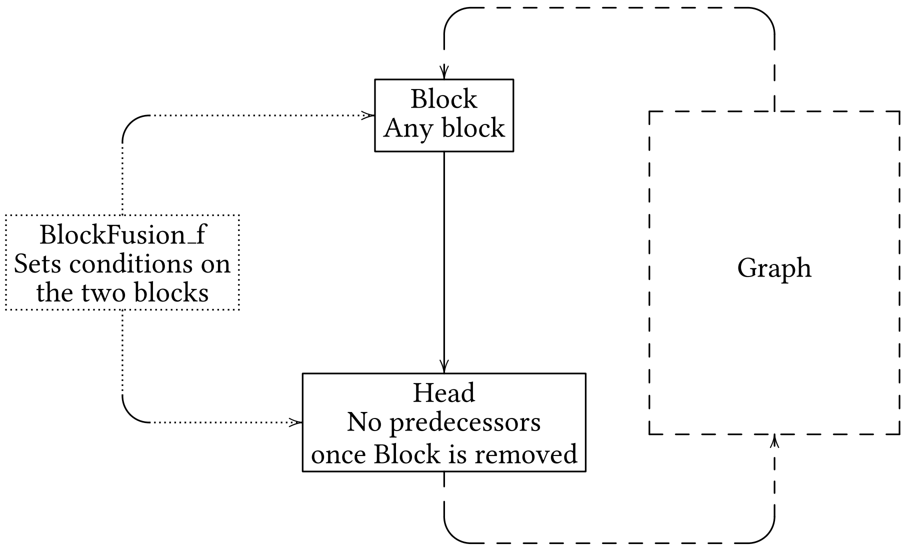


Figure 3: The `BlockFusion` pattern

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 4: 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 5: 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 6: 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 7: 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').
```

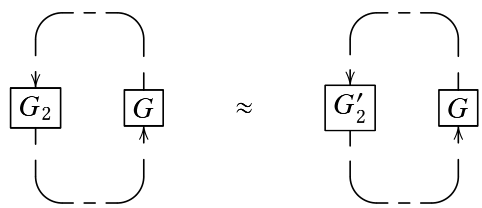
4 Denotation

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.

$$G_2 \approx G'_2 \implies$$



```
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).
```

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.

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 8: The `denote_ocfg_equiv` theorem

4.2 Motivations 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.
```

`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 9: The `fusion` function

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

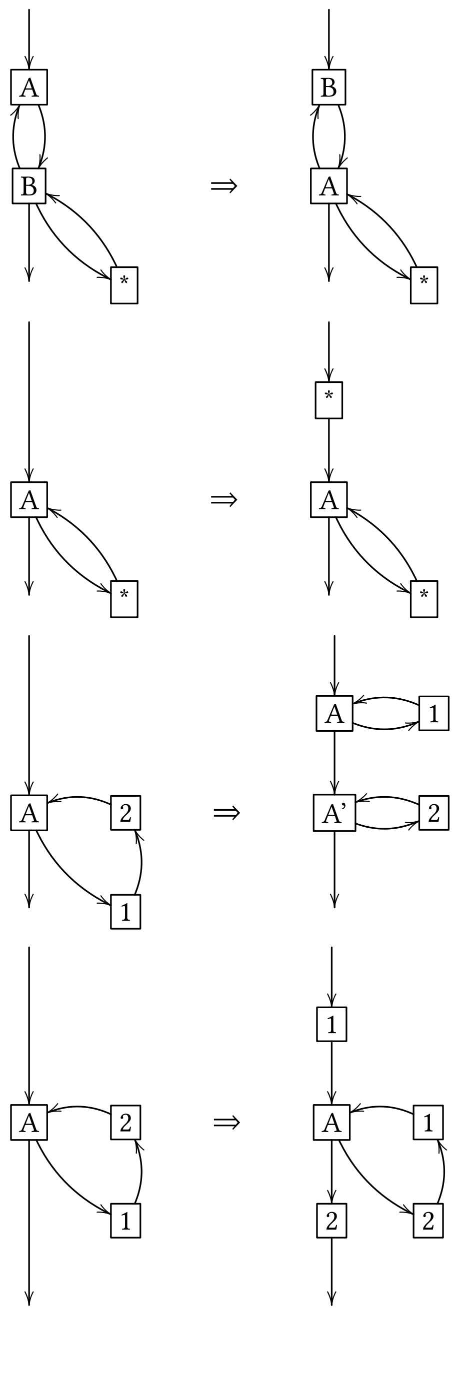
```
Definition σ fusion idA idB := λ(id: bid) ⇒ if decide (id=idA) then idB else id.
```

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