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

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 Team 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.

Compilers are extremely complex objects, gcc's source code is made of about 15 million lines of code [4]. Because of that complexity, it can be hard to assert that compilers follow their specification. For instance, code compiled for aeronautics is usually done without optimizations to reduce the likelihood of unintended behaviors.

One possible solution to this dilemma is *verified compilation*. That is, compilers that are proven to preserve the semantics of the source code. Proving this implies defining formal definitions for the semantics of the source language, the target language and proving the preservation of behavior between them. Verified compilers usually rely on proof assistants like Coq [1], which allow writing executable code, i.e. the compiler, and formal proofs on that code.

Verified compilers have been a research subject for some time now, notably since the CompCert project [3], which established a verified compiler for the ISO C 99 language. It is commercialized by AbsInt [2], for industries like nuclear and aeronautics. Our work takes place in the context of the Vellvm project [8], which aims at formalizing the semantics of LLVM IR. LLVM IR is an intermediate representation used as a front-end and back-end for many languages and architectures. It is based on a Control-Flow Graph (CFG) model, with named blocs and abstract registers.

Vellvm has had some applications, notably to prove some IR to IR optimizations [8], as well as to establish a verified front-end from Helix to LLVM IR [9]. One key challenge that was identified from establishing that front-end was fresh identifier generation. This is a reoccurring issue with formal proofs on named systems [5]. This observation calls for better abstraction for reasoning on graphs. Previous experiments aimed at designing combinators to ease the construction of graphs capturing high-level control flow abstractions, a solution suitable to write front-ends.

My work focused on a orthogonal notion. When writing optimization passes, there is often a need to identify a subgraph with specific structural properties, apply a transformation on it, and link the transformed subgraph back with the context.

To address this problematic, I designed a *Domain Specific Language* (DSL) of *patterns* over graphs, `Pat`. It is equipped with a *matcher* which, given a graph, computes the set of *decompositions* that match a given pattern. This matcher is proven to follow a specification capturing at a higher level the structural properties expressed by `Pat`.

Following this, I wrote an implementation of the Block Fusion optimization based on `Pat`, and proved it preserves the semantics of the graph. To establish this proof, I have on the one hand proved a generic substitution theorem, and on the other hand leveraged the specification of

[1] https://coq.inria.fr/

[2] https://www.absint.com/

```
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

a pattern to show the structural properties of a captured subgraph matches the requirements for the optimization.

In the remaining of this report, Section 2 introduces the necessary prerequisites on Interaction Trees and Vellvm. Section 3 describes the `Pat` DSL and its semantics. And finally, Section 4 covers the Block Fusion case study.

2 Background

2.1 LLVM and Vellvm

The goal of the Vellvm project is to *formally define* the semantic of the LLVM IR and build verified components based on that formalization. LLVM [2] 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 an instruction set similar to low-level assembly languages like RISC, but features high-level informations. This duality allows it to represent any program while still permitting 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 and analysis on the LLVM IR are done through successive *analysis and transformation passes*.

2.2 Interaction Trees

The core of Vellvm's formalization relies on Interaction Trees [7] (ITrees).

ITrees are a data structure designed to represent the dynamic behaviors of a computation. Itrees are a coinductive structure, meaning they represent potentially infinite trees. This allows ITrees to model recursive and effectful programs, including divergent computations.

Figure 1 show the Coq definition of ITrees. It has two parameters: 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` corresponds to 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.

Xia et al. [7] provide concrete examples of ITrees. For example, we can define an effect type `IO`, corresponding to simple inputs and outputs over integers. With `IO`, we can establish an ITree `echo` which loops forever, echoing each input received to the output:

```
Inductive IO : Type → Type :=
  | Input : IO nat
  | Output : nat → IO unit.

CoFixpoint echo : itree IO void :=
  Vis Input (λ x ⇒ Vis (Output x) (λ _ ⇒ echo)).
```

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, composing ITrees can be handled by the combinator `bind`.

To reason over ITrees, we have multiple notions of *bisimilarity*. The most relevant one for our work is *weak bisimilarity*, noted `t1 ≈ t2`. We say that `t1 ≈ t2` if they return the same value, and have the same visible events. Notably, this relation is an equivalence "up-to-Tau", in the sense that we have `Tau t ≈ t` and `t ≈ Tau t`.

Unlike similar projects, which rely on *operational semantics* and simulation diagrams, ITrees rely on *denotational semantics*. It is based on equations that can be used to prove bisimulation. These equations allow the user to reason without using coinductive reasoning or 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.

2.3 Vellvm's semantics

Vellvm introduces Verified LLVM IR (VIR), a *realistic subset* of the LLVM IR.

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 ϕ-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.

To interpret the semantics of the different effects of its syntax, Vellvm uses a stack of interpreters. It gradually introduces external elements to the semantic (intrinsics, global and local environments, ...). Figure 2 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 an *executable* model, which implements one of these behaviors.

Since this internship mainly focused on structural properties, we will only interact with level 0. That is, with control flow as the only interpreted events.

3 The pattern language

In this section we will focus on our main contribution: the design of `Pat`, a DSL of patterns for writing and proving correct program transformations. This DSL is composed of two core components. The first is a datatype, **`Pattern`**, which provides a syntax for the user to specify how they wish to decompose an input OCFG. The second is a *matcher* function, which provides semantics to apply the language to an OCFG, specifying the valid decompositions associated to

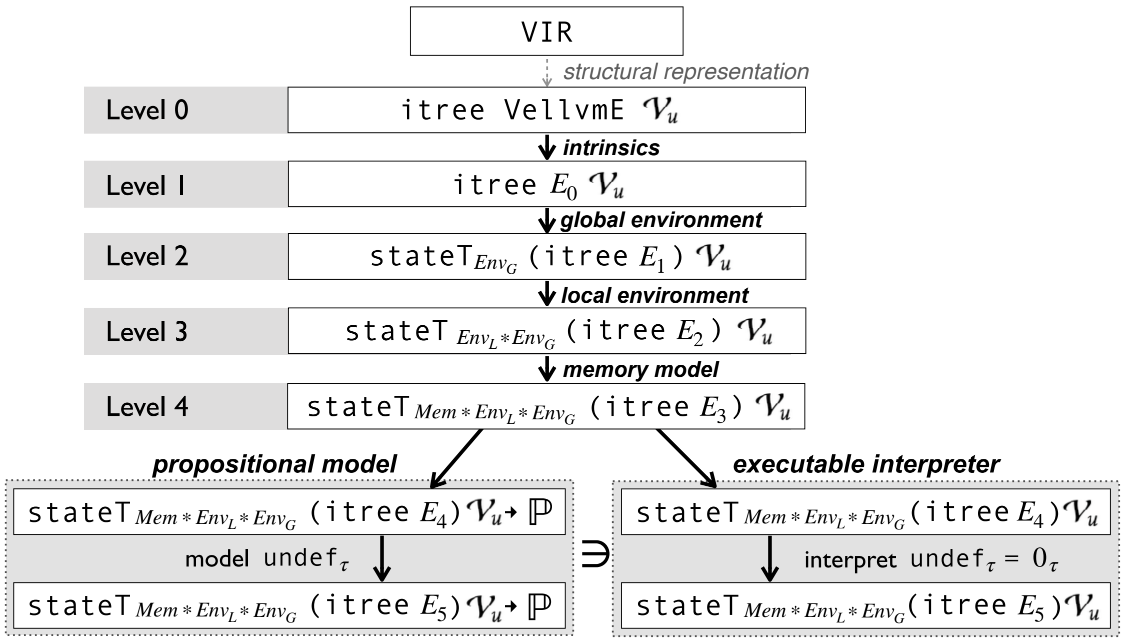

VIR
structural representation
Level 0
itree VellvmE $\mathcal{V}_u$
intrinsics
Level 1
itree E_0 $\mathcal{V}_u$
global environment
Level 2
stateT$_{Env_G}$ (itree E_1) $\mathcal{V}_u$
local environment
Level 3
stateT $_{Env_L * Env_G}$ (itree E_2) $\mathcal{V}_u$
memory model
Level 4
stateT$_{Mem * Env_L * Env_G}$ (itree E_3) $\mathcal{V}_u$
propositional model
executable interpreter
stateT$_{Mem * Env_L * Env_G}$ (itree E_4)$\mathcal{V}_u \to \mathbb{P}$
model undef$_\tau$
stateT$_{Mem * Env_L * Env_G}$ (itree E_5)$\mathcal{V}_u \to \mathbb{P}$
$\ni$
stateT$_{Mem * Env_L * Env_G}$(itree E_4)$\mathcal{V}_u$
interpret undef$_\tau$ = 0_τ
stateT$_{Mem * Env_L * Env_G}$(itree E_5)$\mathcal{V}_u$

Figure 2: Vellvm's levels of interpretation

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.

3.1 Pat: a DSL for pattern matching on graphs

At a high level, we want 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 able to be expressed in Pat.

Figure 3 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

```
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 (block_id * blk * S)
| Head: ∀ {S}, Pattern S → Pattern (block_id * blk * S)
| Branch: ∀ {S}, Pattern S → Pattern (block_id * blk * S)
```

Figure 3: The **Pattern** datatype

this intention by taking as argument a type, which represents the return type of the pattern.[3] This typing information is leveraged in the definition of the matcher, introduced in Section 3.2: matching a pattern of type **`Pattern`** `S` will always return elements of type `S`.

The **`Pattern`** type is build of seven constructors. The only base case is the `Graph` constructor 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. That is: given a return type `S` and a function `S` → `T`, it builds a pattern of return type `T`. The `Focus` constructor, given a pattern `P`, builds a pattern that tries to match `P` on any subset of the graph given as argument. The `Block` constructor extracts any block from the graph and matches the pattern given as argument against the rest of the graph. The `Head` constructor extracts a block without predecessors from the graph and matches the pattern given as argument against the rest of the graph. The `Branch` constructor extracts a block whose terminator is a branch from the graph and matches the pattern given as argument against the rest of the graph.

An example of pattern: block fusion. We illustrate the use of `Pat` to implement[4] a bloc fusion optimization. Block Fusion typically consists of finding two blocks (A and B) such that A is the only predecessors of B, B is the only successor of A, and B is not the graph's entry point. Since an execution of A is always followed by B, and an execution of B is always preceded by A, we can replace them by a single block that follows the execution of both A and B.

The applicable subgraphs are specified with the pattern:

```
pfusion := When (Block (Head Graph)) BlockFusion\_f
```

The pattern starts with `Block` to match any block `A` Then, `Head` matches a block `B` that has no predecessors (except possibly `A` as it is not in scope of the pattern anymore), and finally `When _ BlockFusion_f` asserts that `B` is the only successor of `A`.

Figure 4 illustrates graphically the shape of the graph decompositions that match `pfusion`. The full lines (`A`, `B` and the arrow between them) represent the parts fixed by `pfusion`. The dashed lines refer to parts that are allowed, but not necessarily them: other blocks in the graphs with arrows coming towards `A` or from `B` (but not in the middle of them). The dotted lines show the `When` constructor.

Discussion around the constructors The set of constructors for `Pat` is not minimal. Indeed, `λ p ⇒ Block p` could be written as `λ p ⇒ When (Focus p) (λ '(l, _) ⇒ size l = 1)`, and `λ p ⇒ Branch p` could be written as `λ p ⇒ When (Block p) (λ '(id, b, _) ⇒ term_is_branch b)`. Finding a minimal set was not a

[3]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.

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

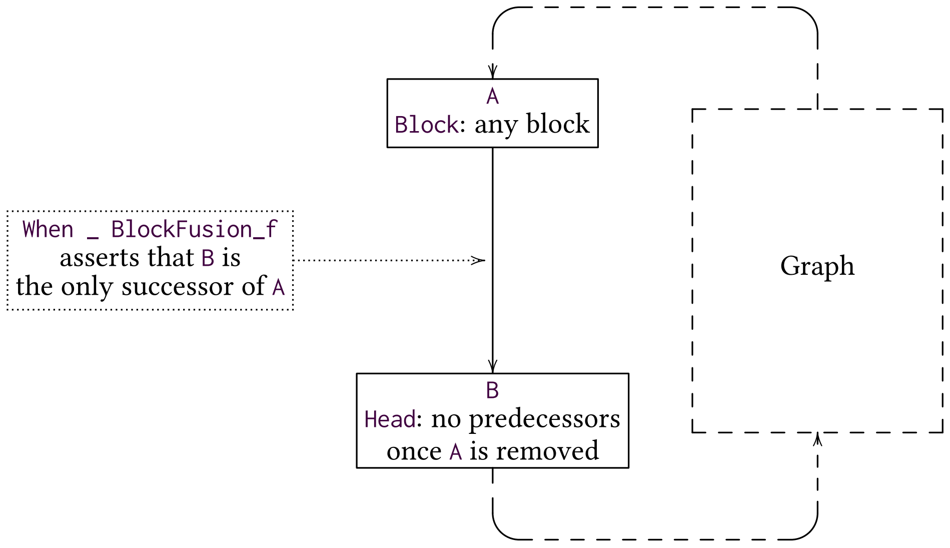


Figure 4: The `pfusion` pattern

goal of the DSL as the main concern was ease of use, both for the user, for designing a matcher, and for establishing the meta-theory.

`Map` and the functoriality it brings feel natural to add and are easy to implement and prove. But we have no immediate use for it at this time. An argument against `Map` could be made, as every other constructor allows working backwards to rebuild the graph by recombining the extracted part (as long as it matches with something).

`Map` also makes `Head` a minimal constructor. Indeed, while `Head Graph` can be written as `When (Block Graph) (λ '(id, b, g) ⇒ is_head id g)`, there is no function f such that `Head Map Graph (λ _ ⇒ ())` can be written as `When (Block Map (λ _ ⇒ ())) f`, since we loose the information on the rest of the graph.

While a user will want to work using closed instances of pattern, that is objects of type `Pattern S`, our proofs work on open instances of pattern, that is objects of type `Pattern S → Pattern (f S)`, to allow compositionality. We will call these objects *opened patterns*. Note that constructors other than `Graph` are opened patterns.

3.2 Matcher functions

We now turn to the question of defining the semantics of our patterns, via a *matcher function*. A matcher function takes a pattern `p` and an OCFG `G` as arguments, and returns a set of decompositions of `G` that match `p`. We have mainly focused on *specifying* patterns, by implementing a matcher that returns *all* valid decompositions. Although rather meant as a specification than as a realistic implementation, the `MatchAll` function, depicted in Figure 5, is nonetheless executable. This not only allows testing, but also drastically reduces the distance left to bridge to have an efficient implementation that is provably sound.

`MatchAll` returns a list of each decomposition that matches the pattern. It recursively computes the decomposition according to the sub-pattern. We can identify three way the constructors are

```
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 5: The `MatchAll` function

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

Figure 6: The `flat_map_r` function

computed: `MatchAll Graph` returns the singleton containing the graph. `MatchAll When`/`Map` first computes the sub-pattern and then applies the filter/map function over the resulting list. For the extracting constructors, `Focus`, `Block`, `Head` and `Branch`, the goal is to first extract an element from the graph, and then apply the sub-pattern.

To extract an element, `MatchAll` relies on decomposition functions, whose definitions are given in Figure 7. These functions each rely on an auxiliary function which is folded over the graph to find each matching element. For example, the `Focus` constructor relies on the `focus` function. It folds an auxiliary function `focus_aux` over the graph to get each subgraph, and then maps an additional function to get the complementary subgraphs. The folding recursively computes each subgraph: it starts with the singleton containing the empty map, and, on each iteration, `focus_aux` returns a list containing each of the map given as argument, and each of them with a new block inserted.

Extracting an element returns a list of type `list S*ocfg`. We then have to apply the sub-pattern over the OCFG on the right hand side. To do that, for each element of the return list, the function `flat_map_r` (Definition provided on Figure 6) matches the sub-patterns to the right-hand side, which gives another `list T*ocfg`, maps the original left-hand side back on it, so we have a `list S*T*ocfg`, and appends the resulting lists together.

3.3 Specification

We have claimed that `MatchAll` *returns all valid decompositions*. To capture this statement, we have provided specifications for each constructor of `Pat`, and proved they are exactly what is captured by `MatchAll`. For this purpose, we will give a generic specification for *opened patterns*, i.e. patterns parameterized by a sub-pattern.

A specification for an opened pattern `PatF` of type **`Pattern`** `S →` **`Pattern`** `(f S)` is a func-

```
Definition focus_aux id b acc : list ocfg := acc ++ (map (insert id b) acc).
Definition focus (G: ocfg) := map (λ G' ⇒ (G', G \ G')) (map_fold focus_aux [∅] G).

Definition blocks_aux (G: ocfg) : (block_id*blk) → (block_id*blk*ocfg) :=
  λ '(id, b) ⇒ (id, b, delete id G).
Definition blocks (G: ocfg): list (block_id*blk*ocfg) :=
  map (blocks_aux G) (map_to_list G).

Definition heads_aux (G: ocfg) id b acc : list (block_id*blk*ocfg) :=
  if is_empty (predecessors id G)
  then (id, b, delete id G)::acc
  else acc.
Definition heads (G: ocfg): list (block_id*blk*ocfg) := map_fold (heads_aux G) [] G.

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

Figure 7: `MatchAll`'s subfunctions

tion `R_PatF` of type `ocfg → Pattern S → f S → Prop`. We say that `MatchAll` is correct for that specification if we have:

```
∀ (G: ocfg) (pat: Pattern S) (X:f S), X ∈ MatchAll (PatF pat) G ↔ R_PatF G pat X
```

Similarly, a specification for a closed pattern `pat: Pattern S` is `R_pat: ocfg → S → Prop`. `MatchAll` is correct for it if: `∀ (G:ocfg) (X:S), X ∈ MatchAll pat G ↔ R_pat G X`.

Let us start with the correction for `Graph`: `∀ G G', G' ∈ (MatchAll Graph G) ↔ G' = G`. Its proof is immediate by definition of `MatchAll Graph`.

The correction for `λ P f → When P f` is given by the following theorem:

```
∀ (P: Pattern S) f X G, X ∈ (MatchAll (When P f) G) ↔ f X = true ∨ X ∈ (
    MatchAll P G).
```

We can also give a theorem for the correctness of `Map`. Proving their correctness is then immediate using builtin lemmas on `filter` and `map`.

The same proof mechanism is used for `Focus`, `Block`, `Head` and `Branch`. We will now detail it for `Head`. Figure 8 gives a semantic definition for `Head`. It is composed of two parts: `heads_aux_sem` which gives semantics for the auxiliary function, which is used to prove its correctness, and `heads_sem` which gives the actual semantics for `Head` and is used to enunciate the `Pattern_Head_correct` theorem. Finally, as show in Figure 9 we can prove the semantics for the auxiliary function, the `heads` function and `MatchAll Head`. We first prove `heads_aux_correct` by induction on `G`, then `heads_correct` and `Pattern_Head_correct` follow immediately.

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

(* The semantics for heads and Head *)
Definition heads_sem (G G':ocfg) (id:block_id) b := heads_aux_sem G G G' id b.
```

Figure 8: The specification of `Head`

```
(* heads_aux follows its semantics *)
Lemma heads_aux_correct:
  ∀ G G' G0 id b,
  (id, b, G') ∈ (map_fold (heads_aux G0) [] G) ↔ heads_aux_sem G0 G G' id b.

(* heads follows its semantics *)
Lemma heads_correct:
  ∀ G G' id b,
  (id, b, G') ∈ (heads G) ↔ heads_sem G G' id b.

(* MatchAll Head follows the given semantics *)
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').
```

Figure 9: The correctness statement of `MatchAll Head`

4 Formal verification of an optimization pass: the case of block fusion

In Section 3, we have introduced a pattern for detecting in a graph pairs of blocks that may be fused. In this section, we turn our attention to using this pattern for defining the Block Fusion optimization itself, and establish its proof of correctness. Rather than doing an ad-hoc proof, we first establish a more general theorem for justifying a broad class of program transformations, and apply it to the case of block fusion.

Recall from Section 2 that we write $t \approx_R u$ to express t and u are weakly bisimilar itrees, whose leafs are related by the relation R, and $\approx$ when $R = eq$. On the Vellvm side, we write $[\![G]\!]$ for the denotation of a (possibly open) LLVM IR control flow graph into a function from pairs of block ids to itrees. Note that we consider here the first level of the denotation: effects remains uninterpreted.

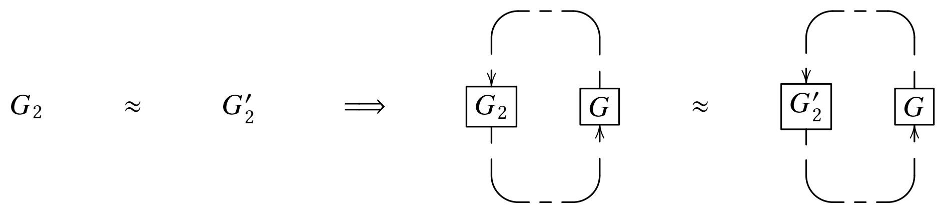
More specifically, the denotation of a graph G is noted $[\![G]\!](from, to)$ where $from$ and to are block ids. $from$ corresponds to the block from which the execution is done, it is used for the denotation of ϕ-nodes. to corresponds to the block we want to execute. If to is in G, $[\![G]\!](from, to)$ corresponds to executing G starting from to, entering from $from$. Then we either have an infinite (but possibly eventful) computation, or the computation terminates and returns a value, or a jump to a block outside of G. If to is not in G, $[\![G]\!](from, to)$ corresponds to returning the jump $(from, to)$.

In the remaining of this section, we overload the equivalence notations between graphs: $G_1 \approx_R G_2$ should be understood as $[\![G_1]\!] \approx_R [\![G_2]\!]$.

4.1 On subgraph substitution under context

The purpose of a pattern is to identify a subgraph of the adequate shape, and optimize it: we hence focus here on optimizations that only modify a subcomponent of the graph—as opposed to graph-wide transformations that may modify everything, such as constant propagation. We hence need to consider two problems: capturing the notion of equivalence of graph we will consider between the patterned subgraph and its replacement, and justifying that this equivalence respects the context.

Weak bisimilarity of interaction trees are proven to respect all relevant contexts in Vellvm. One could therefore simply consider the substitution of equivalent subgraphs G_2 and G'_2 in a larger context G, where graphs are equivalent if their denotations are bisimilar. Represented graphically, that means having:



However, this simple theorem is unusable for our purpose. Let us consider the case of block fusion: we replace two blocs b_1 and b_2 by a block b. If we chose the block id for b the same as for b_2, then we must reflect on the context the need to enter the subgraph from a different entry point. In contrast, if we chose the block id for b the same as for b_1, then we must reflect on the context that we exit the subgraph from a different entry point—as block provenance determines the semantics of ϕ-nodes.

Hence, we must take some renaming into account, whether at the subgraph's input interface, or its output interface. Both are viable, but we favor the former. First because performing and reasoning about the renaming of terminators (to account for renamed inputs) is marginally simpler than of ϕ-nodes (to account for renamed outputs). Second because the resulting obligation for the two equivalent subgraphs becomes simpler to spell out: we need to start the denotations from renamed block ids, but can therefore establish an exact weak bisimulation.

We start by defining the lifting of a renaming function σ on block ids to one on graphs `ocfg_term_rename` σ by applying it to each of its blocks's terminators. 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. We use this renaming function to establish a first contextual lemma: if two graphs g_2 and g'_2 are equivalent when executed from σ related entry points, then the same holds true when extending the graphs with an arbitrary context g_1, provided it is renamed on the other side. That is:

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

This second statement remains insufficient to show the soundness of an optimization such as block fusion. Indeed, if we try start executing from the fused block b_2, the semantics are obviously different, one executing the code from b_1 and not the other.

More generally, we need to constrain the valid entry blocks of the graph. To do so, we introduce a set of block ids `nTO` to restrict the equations to valid input block ids by listing the invalid inputs. With this in hand, we can state the main contextual theorem we establish, displayed on Figure 10. The first series of hypotheses are well-formedness conditions expressing that the context (g_1) and the subgraphs (g_2 and g_2') are disjoint, capturing that the invalid entries belongs to the subgraphs, and establishing necessary conditions on them for the coinductive arguments. The second group of hypothesis simply characterizes the fact that our renaming is a finite renaming from the domain of g_2 to the domain of g_2', despite its very lose typing as a total function. Finally, the third hypothesis is the semantically interesting one: the two subgraphs are equivalent at valid inputs. Given these, we conclude to the equivalence at valid inputs of the overall graphs.

The formal proof is quite technical to mechanize, but relatively intuitive. We sketch here its high level structure, for the interested reader. We proceed by coinduction, and are in one of two cases:

- if we start the computation in g_1, we compute the same block on each side. We hence match them thanks to an up-to bind principle, synchronize both computations after the next jump, and conclude by coinduction;

- if we start the computation in g_2, we prove and use a meta-theorem for Vellvm establishing that the denotation of a graph is equivalent to the denotation of any of its subgraphs, followed by the denotation of the whole graph. We can hence see each side of the computation as starting by computing the denotation of respectively g_2 and g_2': by up-to bind principle, we can match their prefixes using the hypothesis. One last hiccups need to be accounted for: in all generality, g_2 and g_2' could be pure computations, and therefore contain no coinductive guard: we cannot conclude by coinduction immediately. Instead, with additional meta-theory from the itrees, we enrich the intermediate postcondition to know that we are exiting out of g_2/g_2', and therefore are back to the case (1) where we can play the game a second time and conclude.

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. Block Fusion is a relevant optimization for us to choose for multiple reasons. Firstly, it is a commonly used optimization, notably to clean blocks created while building SSA form. Secondly, it is an optimization that modifies the graph. And finally, it is simple on paper to prove that the optimization is correct.

In Section 3, we gave `pfusion`, a closed pattern for Block Fusion. In our work, we used an opened variation, which allows further composing. Figure 11 gives the definition of that pattern, `BlockFusion`, as well as the definition of `BlockFusion_f`. `BlockFusion_f` asserts that B is the only

```
Theorem denote_ocfg_equiv (g1 g2 g2' : ocfg) (σ  : block_id → block_id) (nTO: gset
    block_id) :
    (* Well formedness properties, where ## is disjointness of sets or map domains *)
    inputs g2' \ inputs g2 ⊆ nTO →
    nTO ⊆ inputs g2 ∪ inputs g2' →
    nTO ## outputs g1 →
    g1 ## g2 →
    ocfg_term_rename σ g1 ## g2' →

    (* σ is a finite map from the inputs of g2 to the inputs of g2' *)
    (∀  id, id ∈ inputs g2 → (σ  id) ∈ inputs g2') →
    (∀  id, id ∉ inputs g2 → (σ  id) = id) →

    (* The substituted subgraph is equivalent to the original one *)
    (∀  from to, to ∉ nTO →
      ⟦g2⟧ (from,to) ≈ ⟦g2'⟧ (from, σ to)) →

    (* Then, the graphs are equivalent from any valid initial start *)
    ∀ from to, to ∉ nTO →
      ⟦g2 ∪ g1⟧ (from,to) ≈ ⟦g2' ∪ ocfg_term_rename σ g1⟧ (from, σ to).
```

Figure 10: The `denote_ocfg_equiv` theorem

```
Definition BlockFusion_f {S}: (block_id * blk * (block_id * blk * S) → bool) :=
  λ '(_, A, (idB, B, _)) ⇒ is_seq A idB && no_phi B.

Definition BlockFusion {S} (P: Pattern S) := When (Block (Head P)) BlockFusion_f
```

Figure 11: The `BlockFusion` pattern

successor of A. More specifically, it asserts that the terminator `A` is an absolute jump to `B`. It also asserts that `B` has an empty ϕ-node.

If the first condition only looked at successors, it would accept conditional jumps where both destinations are `B`. However, it would also accept if the condition of the jump could evaluate to an error. Because of that, we need to only accept absolute jumps.

In the case of multiple assignments in a single ϕ-node, the semantic of the ϕ-node will first evaluate each expression, and then do each expression. So, if we replaced a ϕ-node with a sequence of assignments, the resulting semantics could be different, in the case where one of the expressions evaluates to an error.

Now that we have finished detailing the `BlockFusion` pattern, we can define the corresponding optimization. Figure 12 gives the definition of `fusion`, the function that applies the transformation, andσ`fusion`, the corresponding renaming function.

σ`fusion idA idB` is the identity on every block id except `idA`, where it returns `idB` instead. `fusion` takes a renaming function σ and two blocks `A` and `B` as arguments. It returns the block composed of `A`'s ϕ-nodes, the code of both blocks in sequence, and the terminator of `B` on which σ is applied.

To prove the correctness of Block Fusion, we want to leverage the theorem `denote_ocfg_equiv`

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

Definition fusion (σ : block_id → block_id) (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)
|}.
```

Figure 12: The fusion function

```
Lemma fusion_correct {S} G idA A idB B P (X:S):
  let σ := σ fusion idA idB in
  (idA, A, (idB, B, X)) ∈ (MatchAll (BlockFusion P) G) →
  ∀ f to : block_id, to ∉ {[idB]} →
    ⟦ {[idA := A; idB := B]} ⟧(f, to) ≈ ⟦{[idB := fusion σ idA A B]} ⟧(f, σ to).

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 →
  (idA, A, (idB, B, X)) ∈ (MatchAll (BlockFusion P) G) →
  ⟦ G ⟧ (f, to) ≈ ⟦<[idB:=fusion σ idA A B]> (ocfg_term_rename σ G0) ⟧(f, σ to).
```

Figure 13: the correctness theorems of Block Fusion

established in the previous subsection. To do that, we first need to establish the semantic lemma on the transformed subgraph. Informally, we have to show that `A ∪ B ≈ fusion A B`. Figure 13 shows the Coq statements for both theorems: `fusion_correct` and `Denotation_BlockFusion_correct`.

The proof of `fusion_correct` is done by going through the semantics of the two subgraphs and matching the corresponding parts with each other.

From there, proving `Denotation_BlockFusion_correct` is straight-froward as we only need to apply `denote_ocfg_equiv` with `nTO := {[idB]}` and`σ := σfusion idA idB`. `fusion_correct` gives us the semantic lemma, and the well formedness properties are trivial to prove.

With this, we have an executable implementation for Block Fusion, and a correctness proof for it, which highlights the required edits to the context for the optimization to be correct.

5 Future Works

The work done during this internship introduces some possible future works. Notably a more efficient matcher implementation, and being able to specify other structural properties.

5.1 Efficient implementation

The current implementation of the matcher is very inefficient.

One way to have a more efficient matcher is to have a matcher that only returns a single component that matches, `MatchOne`. It would have guarantees that, one the one hand, what it returns does match the pattern, and on the other hand, the matcher only return nothing if nothing could have matched.

Since `MatchAll` also has a role of specification, guarantees on `MatchOne` can be written in relation for `MatchAll`. For example, the latter guarantee can be written as:

```
∀ P G, MatchOne P G = None ↔ MatchAll P G = [].
```

Non-trivial work would have to be done, for example, to go beyond a naive depth-first search along the components.

Another optimization that could be done to the matcher is to have matches more complex than constructor by constructor. For example, being able to "pass down" conditions from `When` constructors into the functions matching the extraction constructors. This would notably allow eliminating the `Branch` pattern without losing efficiency.

5.2 Other structural properties

Another extension of our work would be begin able to identify other structural properties. One that is very relevant for many optimizations is identifying loops.

LLVM's has a specification for loops [5]. Loops are defined as strongly connected components (1) such that all edges from outside the component point to the same node (2). They also need to be maximal subsets with these properties (3).

However, this definition introduces challenges. Mainly, finding strongly connected components is a challenging by itself. For example, the formal proof of Tarjan's algorithm [6] by Chen et al. [1] in Coq is over 700 lines. And adapting it to Vellvm's OCFGs would take significant additional work.

6 Conclusion

During this internship, I designed `Pat`, a DSL of patterns over graphs to express structural properties. I implemented a matcher to compute the set of decompositions that match a given pattern and proved that it follows the specification of `Pat`. I then wrote an implementation of the Block Fusion optimization based on `Pat`, and proved it preserves the semantics of the graph. And, to establish this proof, I proved a generic substitution theorem and established a pattern whose specification captures a subgraph with a structure matching the requirements for the optimization.

The Coq implementation established during the internship is available at https://github.com/lfrenot/vellvm/tree/gmap-minimal.

[5] https://llvm.org/docs/LoopTerminology.html

References

[1] Ran Chen et al. “Formal Proofs of Tarjan’s Strongly Connected Components Algorithm in Why3, Coq and Isabelle”. In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 13:1–13:19. ISBN: 978-3-95977-122-1. DOI: 10.4230/LIPIcs.ITP.2019.13. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2019.13.

[2] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[3] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[4] Víctor Rodríguez. *Cutting Edge Toolchain (Latest Features in GCC/GLIBC)*. 2019. URL: https://static.sched.com/hosted_files/ossna19/db/ELC-2019-compressed.pdf.

[5] Bastien Rousseau. “A DSL of Combinators for Vellvm”. 2021. URL: https://bastienrousseau.github.io/resources/report_cflang.pdf.

[6] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

[7] Li-yao Xia et al. “Interaction trees: representing recursive and impure programs in Coq”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 51:1–51:32. DOI: 10.1145/3371119. URL: https://doi.org/10.1145/3371119.

[8] Yannick Zakowski et al. “Modular, compositional, and executable formal semantics for LLVM IR”. In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473572. URL: https://doi.org/10.1145/3473572.

[9] Vadim Zaliva et al. “HELIX: From Math to Verified Code”. AAI28262508. PhD thesis. USA, 2020. ISBN: 9798569901869.