# Internship Report - A DSL of Combinators for Vellvm

June Rousseau (ENS Rennes)
supervised by Yannick Zakowski (LIP Laboratory)

December, 2021

**Abstract**

Vellvm is a formalization in the Coq proof assistant of the LLVM IR. As such, it is used to prove correct compilation passes, front-ends in particular. While reasoning semantically about generated code is a complex task per se, a significant overhead is entailed simply to ensure that the produced code is well-formed. In particular, LLVM IR requires that the labels of the blocks of code constituting the CFG be unique. To simplify the code generation in Vellvm and optimizations on the control-flow graph, we present a design-specific language, `CFLang`, which ensures by construction the well-formedness of the generated graph. To guarantee the usability of these combinators in the context of verified compilation performed by equational reasoning — as is done in Vellvm —, we provide for each combinator their characteristic semantic equation. Finally, we illustrate the usability of our DSL by writing a compiler from IMP to VIR and validate the well-formedness of the generated code as well as its correctness.

## Contents

**1 Introduction** **2**

**2 Background** **3**
2.1 Interaction Trees . . . . . 3
2.2 Vellvm . . . . . 5

**3 Combinators and `CFLang`** **6**
3.1 Motivation . . . . . 6
3.2 Approach . . . . . 7
3.3 Combinators . . . . . 7
3.4 CFLang . . . . . 10

**4 Case study - Compiler IMP to VIR** **12**

**5 Related Work** **14**

**6 Conclusions and future work** **15**
6.1 Conclusions . . . . . 15
6.2 Limits and Future directions . . . . . 15

# 1 Introduction

**Context** In 2003, the compiler framework LLVM [1] is published. Nowadays, the LLVM framework is very successful and widely used, which can be explained by its modularity. The LLVM framework has been designed to make analysis and transformation for arbitrary source-languages. Indeed, the core of the compilation chain is its intermediate representation, *LLVM IR*, on which many static analysis and optimizations passes are performed. This genericity led to the creation of many frontend targeting LLVM IR, and the code can be turned into hardware-specific assembly language by the various backends. These characteristics make LLVM an interesting compilation framework. As a consequence, LLVM is widely used in the industry — a bug in the compilation chain is even more serious. This makes the LLVM framework a good candidate to formalize it.

In 2009, CompCert [2] came out as a breakthrough in the domain of certified compilation, proving that it is possible to formally certify real-world languages — C99 in this case. CompCert needed decades of research to emerge. It is very specific, and a such project to write a verified compiler for another real-world language would also need a hugh effort. Most of the projects aspiring to define the formal semantic of such languages relied on the *operational semantic*. The operational semantics are intuitive and expressive, and they well support the inductive principles of the theorem prover ; but they have significant drawbacks. Firstly, they are not *compositional* — a compositional semantic can be defined only with the syntax of the language, without adding syntactic constructs such as a program counter. Secondly, they are not *modular* — adding new effects in the semantic require a huge effort, particularly with the correctness theorem of the compiler. Finally, they are not *executable* — in a proof assistant such as Coq, theses semantics are defined relationally and a reference interpreter is developed simultaneously. A proof of correctness of the interpreter is needed. For a language with many effects and evolutive as LLVM IR, the operational semantics is not easily maintainable for an evolutive language such as LLVM IR.

In 2019, *Xia et al.* introduced a co-inductive data structures, the Interactions Trees [3], and built a library around it providing a set of tools easing the design of *denotation semantic* for arbitrary effectful (and potentially divergent) languages. The resulting semantics typically enjoys modularity, compositionality and executability. Moreover, the structure ships with relations capturing notions of behavioral equivalence and behavioral refinements, allowing one to reason equationally about these properties, and in particular hiding from the user the need for explicit co-inductive arguments to establish such termination-sensitive results.

In 2021, the project Vellvm [4] aims to define a formal semantics for LLVM IR in Coq. It introduced a new semantic for VIR (Verified IR), a realistic subset of LLVM IR (including undefined behaviors), using the interactions trees. The adequacy of the semantics is validated through differential testing against `clang`, while its usefulness has been stressed through the proof of elementary optimizations, and most importantly the proof of correctness of a front-end for HELIX [5] targeting VIR.

**Problem** The ITrees allow to define modular, compositional and executable semantics. These three characteristics are essentials in the development and upkeep of a semantic for a large-scale language such as VIR. The combination of their support for equational reasoning with the compositionality of the resulting semantics allows for a satisfyingly algebraic proof-method to establish the correctness of program transformations. But aside from these elegant considerations looms some much nastier, menial issues. LLVM IR is a named language based of control-flow graphs: ensuring that the manipulated graphs and sub-graphs are well-formed is crucial to conduct any semantic reasoning about the language. Naturally, these well-formedness properties, such as the uniqueness of the labels in a graph, are often intrinsically global, and thus intrinsically anti-compositional.

For the programmer who writes a frontend targeting VIR, ensuring a clean generation of fresh names to ensure that the generated code is well-formed is a fairly minor annoyance. For the proof engineer who intends to prove correct this front-end however, it is a huge thorn in their foot. For instance, in HELIX [5], a simple freshness monad is used to generate fresh names during the compilation. In the absence of any further structure, more than 2k LoC of specification and proofs have been necessary to prove that the generated graphs are indeed always well-formed — essentially reasoning about the non-intersection of intervals of seeds used to generate names, hence ruling out conflicts. While such reasoning is at some level unavoidable, it is here conducted very specifically for this compiler, and would have to be essentially restarted from scratch when considering another program transformation.

**Goal & Proposal** Our goal is to discharge the frontend programmer of the generation of the labels, and therefore its proof of correctness. We want to abstract the label generation in an intermediate representation between the frontend and VIR, such as the user does not manipulate labels.

To tackle this goal, we have designed a set of combinators to compositionally create new control-flow graph. They link arbitrary well-formed control-flow graphs (pieces of VIR syntax) together. Moreover, we prove sufficient conditions ensuring that the combinators produce well-formed graphs. On top of these unsafe combinators, we build a design-specific language (DSL) living in a freshness monad and taking care of generating appropriate names. The DSL ensures the sufficient conditions to use the combinators. Thus, we can be sure that the code generated by the DSL is well-formed by construction.

**Contributions** Our contribution are the following:

- We design a set of high-level combinators of arbitrary open control-flow graph (OCFG);
- We develop a meta-theory of freshness monad which generates fresh label names;
- We design a domain-specific language, named `CFLang`, aiming at filling the gap between the label generation with the freshness monad and the unsafe combinators of OCFG;
- We provide an equational theory for `CFLang`, allowing the user to use an algebraic proof-method
- We perform a case study over a compiler from IMP to VIR, using `CFLang` as intermediate representation to generate the labels (ongoing work).

The remaining of this report is organized as follows. First, we introduce the necessary background on Interaction Trees and Vellvm in Section 2. In Section 3, we present our set of combinators on arbitrary control-flow graph and the DSL `CFLang`. In Section 4 we describe the compiler from IMP to VIR. Finally, the related work and the conclusions are covered in Section 5 and Section 6.

# 2 Background

## 2.1 Interaction Trees

**Definition** Interactions trees [3] (ITrees) are a co-inductive data-structure. A value of this structure, a tree, represents the dynamic behaviors of a computation; the datatype is expressive enough to model recursive and effectful programs, including divergent computations. Formally, the type `itree E R` is defined in Fig. 1, where `E : Type → Type` is a set of *events* and `R: Type` is the return type of the ITree. As shown, the ITrees are built using three constructors: (1) `Ret`, represents the trivial computation, halting and returning a pure value of type `R`, (2) `Tau` is a silent step, representing an internal computation followed by the rest of the computation t, and (3) `Vis e k` is a *visible event*. `e` is an *external interaction*, returning a value of type `A` (the answer), and `k` is a continuation, ie. `k : A → itree E R`, such that `k` can behave differently for different values of the response to the event. Interestingly, we may notice that the type `itree E` is a variant of the free monad [6], parameterized by a `E`. The `bind` operation enables the sequence of computations. Moreover, it is iterative, meaning that they can encode loops and recursion with the operators `iter` and `mrec`.

**Example** Let us consider the external event type of IO, representing simple interactions of read and write natural numbers. The corresponding datatype of interaction — the `E` fed to the type of computations — is defined in Fig. 2a. The ITree defined in Fig. 2b represents a divergent computation that infinitely reads a natural number, and writes it in the environment.

```
CoInductive itree (E : Type → Type) (R : Type) : Type :=
| RetF (r : R)
| TauF (t : itree E R)
| VisF {A : Type} (e : E A) (k : A → itree E R).
```

Figure 1: Coq definition of the datatype ITree

```
Definition IO : Type → Type :=
| Read : IO nat
| Write : nat → IO unit.
```

(a) Definition of the Event IO

```
CoFixpoint echo : itree IO void :=
Vis Read (fun x ⇒ Vis (Write x)
                      (fun _ ⇒ echo)).
```

(b) ITree representing the computation *echo*

Figure 2: Simple example of ITree – Echo with IO

**Equivalence** In order to compare ITrees, the library provides two relations, encoding two notions of bisimulations over the computations. The first one is the *strong bisimulation*, written $t_1 \cong t_2$: it relates two ITrees with the exact same shape. The second one is the *weak bisimulation*, written $t_1 \approx t_2$: it relates two ITrees returning the exact same value, emitting the exact same visible events, but potentially disagreeing on the amount of fuel, i.e. the number of tau-nodes, required to run these computations. This relation is also called "equivalent up-to-tau". Thus, and contrary to the strong bisimulation, the weak bisimulation typically respects equations such as $\texttt{Tau}\ t \approx t$.

Additionally to behavioral equivalence, both strong and weak bisimulations can be used to express (heterogeneous) refinements of programs: they are parameterized by a relation $R : A \to B \to Prop$, where $A$ and $B$ are the return types of the computations being compared, such that $t_1 : \texttt{itree}\ E\ A$ and $t_2 : \texttt{itree}\ E\ B$ are related by $t_1 \approx_R t_2$ with $\texttt{Ret}\ a \approx_R \texttt{Ret}\ b \Leftrightarrow R\ a\ b$.

**Modular and compositional semantic** The Interaction Trees are modular, in the sense that we can easily add or remove effects to the semantic. Indeed, the effects of a computation encoded by an ITree is visible thanks to the constructor `Vis`. The constructor `Vis` can be understand like uninterpreted events. To define semantics to events, we define an *event handler*. The interpretation of an ITree consists on folding the event handler all over the ITree. It should commute with monadic structure. In other words, an event handler is good if we can reason on the ITree before interpretation.

Additionally, the Interaction Trees can be combined compositionally thanks to functions called "combinators". An important combinator is `iter: (A → itree E (A+B)) → (A → itree E B)` encoding the iterations and hiding the co-induction to the user. Given a starting state $a : A$, the combinator *iter* either iterates again a body $body :\ A \to \texttt{itree}\ E\ A + B$ with a silent step (producing a `Tau`), or stops the computation. Moreover, the ITrees also provides other combinators such as *mrec* to encode mutual-recursive combinators.

**Operational VS Denotational Semantics** On the one hand, the usual proof methods for the operational semantics are based on *simulation diagram*. These semantics are usually defined with a *small-step semantic* with a predicate `step : config → config → Prop` where `step c1 c2` is the transition from a configuration c1 to configuration c2. We may notice the transition is not typically expressed as a function that computes c2 from c1. The method of *simulation diagram* is represented in Fig. 3. The black part is the hypotheses and the blue part is the conclusions. Let $R$ be the equivalence relation between a state for the source language $Si$ and a state for the target language $Si'$. For each step from $S1$ to $S2$ in the semantics of the source program that emits the observable effect labeled by $t$, should correspond one or many steps from $S1'$ equivalent with $S1$, and emitting the same observable labeled by $t$. $S1$ represent the state in the execution of a program from the source language and $S1'$ its equivalent state from the target language.

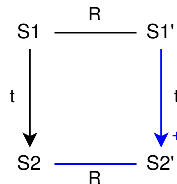



Figure 3: Simulation Diagram

.

On the other hand, the denotational semantic of the ITrees is based on an algebraic proof-method

$$
\begin{aligned}
exp &::= \ \dots \ \ instr ::= \dots \\
id,\ bid &::= string \\
term &::= \texttt{branch}\ (exp,\ bid,\ bid)\ \mid\ \texttt{return}\ (exp)\ \mid\ \dots \\
phi &::= \Phi\ (list\ (bid, exp)) \\
block &::= \{entry : bid;\ phis : list\ (id, phi);\ code : list\ (id, instr);\ term : term\} \\
cfg &::= \{name : id;\ args : list\ bid;\ entry : id;\ body : list\ block\} \\
mcfg &::= \texttt{mrec}\ (cfg\ ,\ \dots, cfg)
\end{aligned}
$$

Figure 4: An excerpt of VIR's syntax

with equational rewriting. The theory provides a bunch of equations that can be used to prove the relation of bisimulation. In particular, the equations hide the coinductive reasoning from the user. They can therefore treat the definition of the weak bisimulation as a black box: they only have to reason purely equationally. In practice, the simulation diagram required to conduct the algebraic reasoning is simpler than the one necessary in the operational approach due to the compositionality of the semantics: notions such as program counters and syntactic contexts are lifted away.

However, it is possible to use the semantic equations by rewriting them (eg. in a context $C$, to replace $C[t_1]$ with $C[t_2]$ if we have $t1 \approx t2$) only if the context is *proper*, meaning that it respects the equivalence. Both strong and weak bisimulation have been proven to be congruent for all the combinators on the ITrees (bind, iter, ...), and the library provides the necessary instances to perform the rewrites. In brief, if the ITrees are defined with the combinators only, we can safely rewrite the equations.

## 2.2 Vellvm

Vellvm [4] is a project aiming at defining formally the semantic of the LLVM IR and building verified components upon it.

**LLVM IR** The LLVM IR is a low level code representation close to an abstract RISC-like instruction set, with high-level information. It has been designed to be low-level enough to represent any arbitrary program, with enough high-level information to permit extensive analysis and optimizations. The LLVM IR is language based on control-flow graphs, using named labels and registers. The virtual registers are in Static Single Assignment (SSA) form [7]: graphs guarantee that any register always has a unique definition site, and that this definition site dominates all its use sites. This invariant has proved to be invaluable for efficient and scalable static analyses and optimizations. The language is statically typed, with *primitive types* (boolean, integer, floating point and pointer) and structured types — although the type system is quite weak, essentially used to infer statically the size of the manipulated data.

Finally, the low-level memory model provides casts between pointers and integers.

VIR (Verified LLVM IR) is a realistic subset of the LLVM IR. Fig. 4 shows an excerpt of the syntax of VIR[1].

**Semantic** The semantic of VIR is ITree-based, meaning that each piece of syntax is represented by an ITree: this itree is a potentially infinite unfolding of the dynamics of the program, where all effects other than the control flow are still represented as syntactic (i.e. uninterpreted) events in the tree. This semantics captures non-trivial features of the language — pointers, LLVM's phi-nodes, poison values or undefined behavior.

These representations are defined recursively on the syntax of the programs, each syntactic sub-component admitting its own representation. In particular, proper compositionality at the graph-level is ensured by defining the semantics of a complete CFG as a fixed-point of the function associating to each block identifier its representation. Because each component admits a well defined meaning in isolation, we can prove compositonal semantic equations — up-to weak bisimulation typically — characterizing their behavior in terms of the semantics of their syntactic sub-components. Vellvm provides a full set of such equations, allowing for reasoning through rewriting about the semantics of VIR programs.

[1]For this work, we are only concerned with the control-flow: we therefore focus mainly on the *block* and *cfg* parts of the datatype.

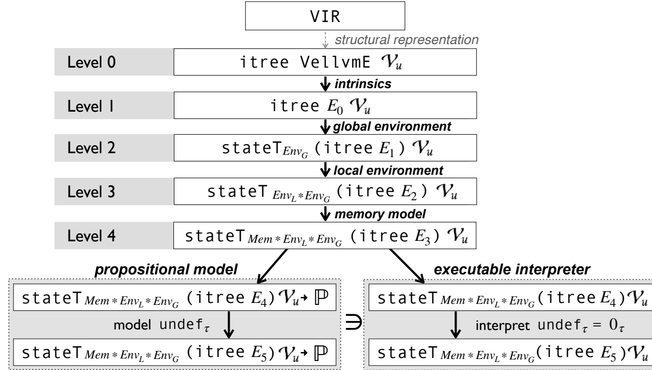



Figure 5: Levels of interpretations

.

**Effects and interpretation** Finally, Vellvm introduces a semantic for the effects with an interpreter. In fact, there is a stack of interpret introducing gradually the semantic of the effects. Each level of interpretation makes the state monad more complex. This *tower of interpretation* allows to reason about the semantic with the simplest monad necessary: typically, one can pretend the semantics to be deterministic if no undefined values are involved.

Fig. 5 presents the stack of interpretation of the effects. We may notice that the two last levels separates the propositional model and the executable interpreter. In fact, the model accounts for the non-determinism of VIR (such as the undefined behaviors) by interpreting them propositionally. They are suitable for specification, but not for extraction. The executable interpreter on the other hand is proved to implement *one of the allowed behavior* as specified by the model.

# 3 Combinators and `CFLang`

## 3.1 Motivation

Weak bisimulation is proved to be a congruence for the itree-combinators, the fixpoint operator *iter* in particular. These combinators ship with a set of semantic equations characterizing their meaning. These itree-level combinators are then used to represent the syntax of a language, for example IMP or VIR. Hence, we can increase our level of reasoning abstraction by lifting these equations through new semantic equations at the syntax-level. For instance, let $c_1$ and $c_2$ be two pieces of code (eg. IMP code) and let $c_1;;c_2$ be the sequence of $c_1$ and $c_2$. If we write $[\![c]\!]_{IMP}$ the denotation of $c$, which is an ITree, then the equation of the sequence goes as follows: $[\![c_1;;c_2]\!]_{IMP} \approx [\![c_1]\!]_{IMP};[\![c_2]\!]_{IMP}$, where $t_1;t_2$ is a notation for `bind t1 (fun _ ⇒ t2)`. Simply put, this equation proves that the meaning of the sequence of two programs is indeed the sequence of their respective meanings. Thanks to this compositionality, a program transformation between two languages whose semantics is defined in this style can be proved directly by induction on the syntax of the source language. Each syntactic sub-component is proved correct by a combination of symbolic rewriting and Hoare-style relational reasoning. Simulation diagrams are therefore never made explicit, we only need to reason about the local invariant relating the memory state at each syntactic program points.

However, VIR is a CFG-based language: it therefore does not explicitly contains any construction for high-level control-flow mechanisms such as sequencing or iterating. Compiling a source language down to VIR, such as is done for HELIX for instance, therefore currently requires to manually encode these constructions. Furthermore, because VIR is named, there are many opportunities to break fundamental well-formedness conditions: typically, two blocks could be named identically, resulting on one of them being shadowed by the other.

Unfortunately, this kind of well-formedness properties is global, and is therefore intrinsically anti-compositional. An ad-hoc solution is to use a non-controlled freshness monad to generates the label names in the compiler: intuitively, the compiler is passed by argument a seed used to generate fresh

names. The operations on the monad are written such that each generation of a fresh label increments the seed, ensuring that it will never produce a duplicate label. This solution was notably adopted in HELIX. However, although the freshness monad is a convenient tool, nothing constraints the compiler to only generate names through the operations provided by the monad: it could craft a name at any-point, or inadequately pass the state monad by argument during a recursive call. In practice, proving that the graphs generated by the HELIX compiler are well-formed turn out to represent a significant amount of tedious work. Furthermore, this reasoning about interval arithmetic and naming ends up cluttering the semantic arguments of correctness of the compiler.

Although more well-formedness conditions could be of value, through this internship, we have specifically focused our attention onto the uniqueness of labels condition. Formally, the uniqueness of the

```
Definition wf_cfg_bid (c : cfg) : Prop :=
list_norepet (map entry c).
```

Figure 6: Coq Property of the Uniqueness of Labels

labels, defined in Fig. 6, gathers all the block labels of the cfg in a list, and ensures that there is no duplicate.

In face of this observation, our goal is to help the verified-compilation enthusiast in two ways: providing higher-level semantic abstractions by reflecting high-level control-flow constructs in VIR on one hand; ensuring the well-formedness of generating graphs by construction on the other.

## 3.2 Approach

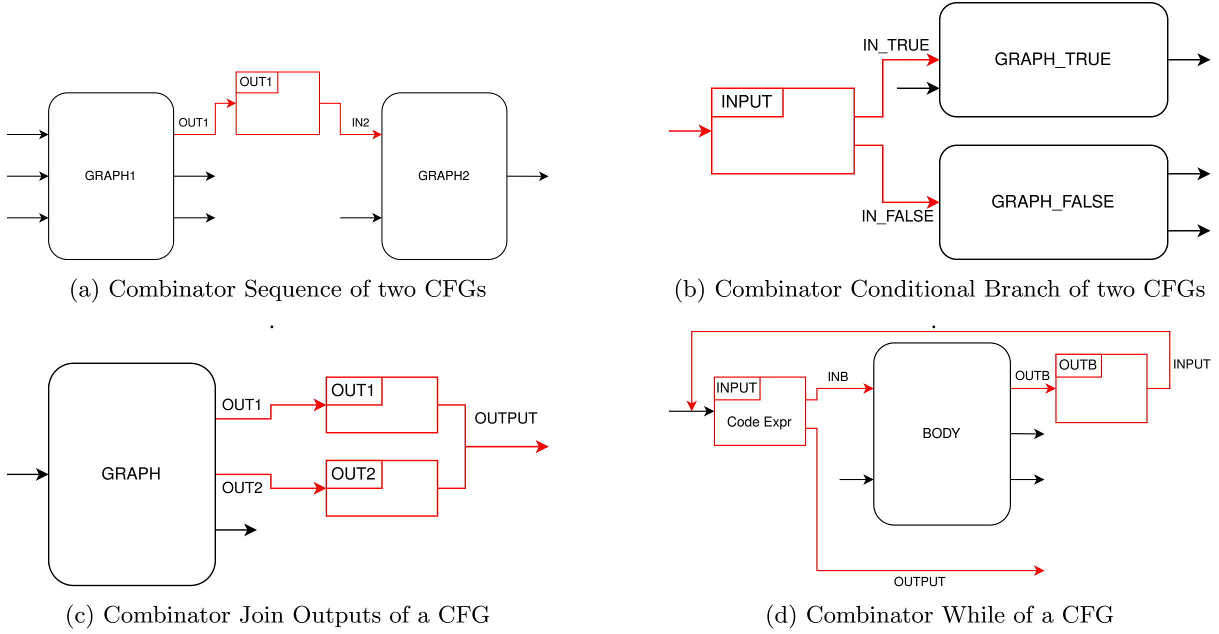
In order to lighten the proof burden of a verified frontend by discharging the proof of well-formedness of the generated graphs, we propose to add abstraction layers between the potential frontend and VIR. In a first step, we define a set of functions, called *combinators*, used to build control-flow graphs compositionally. The combination preserves the well-formedness — at least regarding the unicity of the labels. We characterize them semantically, but some preconditions are needed to use the equations. Indeed, unlike an intrinsically typed representation, the combinators do not guarantees the well-formedness statically by the type system — they do not guarantee the well-formedness at all. We instead established and proved the necessary conditions embedding the well-formedness, but we do not want let the user discharge manually these conditions. In a second step, we provide a DSL, `CFLang`, interfacing the combinators and a freshness monad for the generation of the label names. This second layer of abstraction generates well-formed fresh labels for the basic blocks that guarantee the preconditions needed to use the equational theory of the combinators. The compilation of a `CFLang` program generates a well-formed VIR code and a set a equations of denotation characterizes semantically a `CFLang` program.

To summarize, we provide a DSL of graph generation combinators that prevent the user to explicitly craft the names of the labels used. We guarantee that as long as they only use our DSL to write their program transformation, the resulting graphs will be well-formed by construction. Finally, we ship each combinator with a proved semantic equation, allowing for an high-level, abstract reasoning about the correctness of the resulting compiler.

## 3.3 Combinators

**Overview** As explained above, we design a set of combinators encoding usual high-level control-flow operations, such as the sequence or the conditional branching, into operations on control flow graphs. When designing such combinators in a dependently-typed language such as Gallina, there is tension between the ease of defining the combinators themselves, and the ease of using these combinators: on one extreme, the intrinsically typed approach ensures all properties by well-typedness, while an untyped approach leads to necessary side conditions when using the combinators. In our work, we resolve this tension in two steps: the combinators defined in this section are weakly typed, but we build in a second step a sound interface to constrain their use on top of it, as described in the following section.

In particular, we observe that the combinators we present though this section manipulate directly the syntax of VIR, and therefore *named* CFGs — as opposed to manipulating an anonymous representation. The names of the new blocks introduced by a given combinators, or of the inputs and outputs linked,




(a) Combinator Sequence of two CFGs

(b) Combinator Conditional Branch of two CFGs

(c) Combinator Join Outputs of a CFG

(d) Combinator While of a CFG

Figure 7: Combinators

are therefore provided explicitly by argument. The combinators therefore cannot ensure properties such as the uniqueness of label names, or the validity of their semantics, by construction: they are indeed not statically typed. We however formulate through a set of lemmas the sufficient conditions under which the combinators behave nicely: typically, one needs to ensure that the fresh label passed by argument to create a new block is indeed fresh relative to the labels contained in the graphs passed by argument.

Finally, we stress another design choice: the combinators do not perform any renaming. When a unification between an output label of a graph, and an input label of another one is need, we instead create a fresh block with no code acting as a direct wire between them. This greatly simplifies the construction and theory, and while the resulting graphs naturally are indebted with an execution cost, we anticipate this to be easily resolved by a simple *block fusion* optimization such as the one already existing — and proved correct — in Vellvm.

We present the following combinators:

- *basic block*: given a piece of straight code, the label of the block and the label of the next block, creates a basic block with an unconditional branch to the next block
- *conditional*: given two graphs, the name of their respective input blocks, and a condition, either jumps in the first graph if the condition is true, or jumps in the second one if the condition is false;
- *join*: given two outputs of a graph, makes them jump onto the same output;
- *sequence*: given two graphs $g_1$ and $g_2$, with $out_1 \in outputs(g_1)$ and $in_2 \in inputs(g_2)$, links them such that if the flow leave $g_1$ by $out_1$, it jumps into the block $in_2$;
- *while loop*: iterates a body described as a graph, given its input and output labels, while a condition described as a straight code is true.

We represent on Fig. 7 four of these combinators. In each case, we depict in red the additional pieces of syntax introduced by the combinator, while the black ones are sub-components taken as arguments.

For instance, Fig. 7a is a representation of the CFG generated by the sequence combinator: `cfg_seq GRAPH1 GRAPH2 OUT1 IN2`. It links sequentially GRAPH1 to GRAPH2: intuitively, doing so should simply consist in unifying the output of GRAPH1 with the input of GRAPH2, however this would require to perform some renaming. Instead, the combinator takes as argument the labels representing the output and input meant to be linked, and introduces an empty block labelled by the former and consisting of a direct jump to the latter. Additional outputs of GRAPH1 (resp. inputs of GRAPH2) are treated in the

resulting graph as outputs (resp. inputs). From this representation, we may notice some well-formedness requirements:

- OUT1 needs to be free in G1 and G2 (there is no block already named OUT1);
- G1 and G2 must both be well-formed;
- The labels of G1 and the labels of G2 must be disjoint.

We will describe how we ensure these requirements are systematically met in Section 3.4.

As its name suggest, the *while combinator* iterates over a body while a boolean expression is verified. To do so, the combinator creates a block computing the expression and doing the conditional branch. If the condition is true, the flow jumps in the BODY by the block INB. Finally, the combinator creates an empty block that links the output of the BODY with the INPUT block, which introduces the iteration.

The remaining combinators are pretty straightforward.

The operations we have described are purely syntactic manipulations, consuming and producing graphs. In order to prove that they deserve their respective evocative names, we now turn to the proof of the semantic equations we have established.

**Semantics** In order to characterize semantically the combinators, we provide characteristic semantic equations for each combinator. As highlighted in the previous paragraph, our combinators are however too weakly typed, they can be ran on nonsensical arguments. We therefore identify for each of them the sufficient conditions for their correctness, and prove the semantic characterizations assuming these constraints. Once again, we will ensure these constraints are satisfied by construction in a second time.

Although this technical detail is implicit in these equations, we emphasize that due to the modular approach to building semantics using ITrees, we are able to establish these equations independently from the implementation of any of the effects of the language. They are established as equivalences between uninterpreted trees, and the Vellvm's meta-theory ensures that these equations lift to the top-level notion of refinement of programs.

```
Lemma denote_cfg_seq : ∀g1 g2 out1 in2 from to,

wf_seq g1 g2 out1 in2 →
In to (inputs g1) →

(denote_cfg (cfg_seq g1 g2 out1 in2) from to)
            ≈
(d  ← denote_cfg g1 from to ;;
match d with
| inr dv ⇒ ret (inr dv)
| inl (src, target) ⇒
   if target =? out1
   then denote_cfg g2 out1 in2
   else denote_cfg g2 src target
end).
```

Figure 8: Denotation of the sequence

Fig. 8 describes the characteristic equation for the sequence combinator. This lemma should be read as follows: assuming that the three well-formedness assumptions described in the previous paragraph hold (predicate `wf_seq`), assuming that the block we start the evaluation from belongs to the first graph, then the denotation of the graph built using the sequence combinator is precisely the sequence of the denotation of the first graph followed by the one of the second graph. This latter sequence is slightly more verbose than desired in the equation. At the end of the execution of the first graph, we need to consider three options depending on the returned value. If the function has returned, we are done. If the label jumped to is the one expected, we need rename it before feeding it to the second graph. Otherwise, we simply pass it along to the second graph.

Fig. 9 describes the characteristic equation for the while combinator. The hypotheses `has_post` means that if the flow enters in the `body` by the block `inB`, then the flow must exit `body` by jumping into `outB`. In the equation, the `evaluate_conditional` denotes the code of the condition, raising an exception (VIR event) if the expression is not a boolean. Moreover, the `itree` *c o*mbinator *iter* do the iteration.

```
Lemma denote_cfg_while_loop : ∀expr_code cond body input inB output outB from,

wf_while expr_code cond body input inB output outB →
(∀ bfrom, has_post (denote_cfg body bfrom inB) (fun vob ⇒ ∃bfrom, vob = inl (bfrom, outB)))
     →

(denote_cfg (cfg_while_loop expr_code cond body input inB output outB) from input)
        ≈
(iter
  (fun (_ : unit) ⇒
      b ←evaluate_conditional expr_code cond;;
      if b : bool
      then
        vob ←denote_cfg body input inB;;
        match vob with
        | inr v ⇒ Ret (inr (inr v))
        | inl _ ⇒ Ret (inl tt)
        end
      else Ret (inr (inl (input,output)))) tt).
```

Figure 9: Denotation of the while

The proof of this semantic equation for the sequence essentially relies on the equations provided by Vellvm and the Interaction Trees. We unfold the definition of the combinator on the left-hand side, and proceed by symbolic execution of the VIR syntax. After enough rewriting, we match up a common prefix on both side of the equation: the semantics of the first graph. The theory of *eutt* ensures that we can match these prefixes and proceed with their continuations. At this stage, more symbolic execution on the left hand side allow us to compute along the empty block introduced to jump to IN2, leading to two computations that once again match exactly, allowing us to conclude.

Most other combinators follow a similar pattern: once the appropriate sufficient conditions are identified, the proof follows smoothly by equational reasoning.

The semantic characterization for the *while combinator* Fig. 9 however required another proof method. Indeed, for the other combinators such as the sequence, the equations basically hide the empty blocks, whereas the equation for the while is more abstract. Moreover, reasoning on an equation involving the ITree combinator *iter* is much more complex than equation involving only *bind*. The equations of Vellvm were not sufficient and we tackled the proof differently. We had to reason co-inductively on the interactions trees, and to introduce intermediate lemmas relating interaction trees with the strong bisimulation, breaking the abstraction *eutt*. It raises an interesting question for a future work: how can we avoid to break the abstraction *eutt* and to spare totally the co-inductive proof for future theorems.

We are now able to combine arbitrary control-flow graphs, preserving the well-formedness properties of the VIR code, and manipulate the semantic of the combinators with the equations. But, the user still have to ensure annoying preconditions with the name generation. The next step is to provide an interface between the combinators and the name generation.

### 3.4 CFLang

So far, the combinators provide a way to build compositionally arbitrary control-flow graphs, but they are too lenient: they say nothing as to how the fresh names of the labels should be picked. As a consequence, their use is quite impractical: if used to write a compiler such as the one for HELIX, they would structure the proof of well-formedness of the resulting graphs, but would still require a lot of manual work to discharge at each call the corresponding side conditions. In order to alleviate this problem, we build a new DSL of control flow graphs, named `CFLang`. Programs in `CFLang` are essentially refined into VIR graphs built using exclusively combinators and explicit straight line code. This refinement however adds an additional technical bit to the approach: a freshness monad is used to introduce adequately the fresh names provided to the combinators. As a result, when writing a VIR program using `CFLang`, the programmer cannot craft explicitly a label, allowing us to ensure that the resulting graphs are always well-formed. Thus, the user can compile the source language to `CFLang`, without generating nor proving anything about the name of the labels.

```
Record scfg : Type :=
{ graph : cfg ;
  ins : list block_id ;
  outs : list block_id }.
```

Figure 10: Definition SCFG

**SCFG** Given a control-flow graph, we want to hide some inputs and some outputs of the cfg. Indeed, we want to *internalize* the labels we statically know won't ever be linked again to another graph — such as the one involved in a loop — and we want to expose only the labels that may be linked. Thus, we introduce an intermediate data-structure exposing a subset of the labels of a graph.

A *Safe Control-Flow Graph*, dubbed SCFG, is a control-flow graph with a subset of its inputs and a subset of its outputs visible to the user. Formally, the datatype `SCFG` is defined as a record with three fields, as described in Fig. 10. A SCFG $dg$ should respect some well-formedness properties to be valid. We say that $dg$ is well-formed if:

- (ins $dg$) $\subseteq$ (inputs (graph $dg$)) - ins exposes a subset of the inputs of the graph
- (outs $dg$) $\subseteq$ (outputs (graph $dg$)) - outs exposes a subset of the outputs of the graph
- `wf_cfg_bid (graph dg)` - the graph is well-formed regarding the uniqueness of the labels
- (ins $dg$) $\cap$ (outs $dg$) $= \emptyset$ - the visible inputs and the visible outputs are disjoints

The SCFG has two purposes. First, to guarantee statically that the well-formedness of our graph cannot be compromised by a future linking. For instance, linking two times the same outputs to different inputs would create two times an empty block with the same label and thus breaks the unicity property: we prevent this situation by internalizing such labels. Secondly, the SCFG fills the gap between an anonymous representation (such as `CFLang`, introduced below) and a named representation of the CFG (such as the one from VIR, as manipulated by the combinators). It always exposes a valid subset of the concrete names of the graph.

$$
\begin{aligned}
\mathit{cgraph} ::=\ & \texttt{CBlock}\ \mathit{code} \\
\mid\ & \texttt{CSeq}\ \mathit{cgraph}_1\ \mathit{cgraph}_2 \\
\mid\ & \texttt{CIfThenElse}\ \mathit{cond}\ \mathit{cgraph}_T\ \mathit{cgraph}_F \\
\mid\ & \texttt{CWhile}\ \mathit{code}\ \mathit{cond}\ \mathit{cgraph}_B
\end{aligned}
$$

Figure 11: The syntax of `CFLang`, where *code* is a list of VIR instructions and *cond* is a VIR expression
.

**Syntax `CFLang`** `CFLang` (Control-Flow Language) is a DSL manipulating control-flow graphs without labels. It describes the control flow of graphs with one input and one output. Fig. 11 shows the grammar of `CFLang`. Introducing a concrete syntax for the DSL ensures statically that a compiler is written strictly using the underlying combinators, and does not introduce any unsafe construction akin to break the invariants.

As such, a program in `CFLang` is refined into a SCFG by the function
`evaluate : CFLang → (FreshState → FreshState * SCFG)`. The labels are generated thanks to a freshness monad. The state of the freshness monad maintains a counter, used to create an fresh label. The compiler from `CFLang` to SCFG has been proved to produce a well-formed SCFG, meaning that the VIR code encapsulated in the SCFG is well-formed.

As a new intermediate layer, we have to characterize the semantic of the abstraction. It consists in providing an equational theory for `CFLang`. The equations are close to the equations of the combinators, but the labels are concrete and belong to the exposed labels of the SCFG. To illustrate this similarity, we provide the semantic equation of `CSeq` in Fig. 12. We may notice that the equation is identical to the equation Fig. 8, but the labels `out1` and `in2` have been replaced by the head of ins and outs of the produced SCFG. The proof-method is a mix of the equational theory of the combinators and the meta-theory if the freshness monad to ensure the preconditions.

```
Lemma denote_cseq : ∀(s1 s2 s3 : FreshState) (c1 c2 : CFLang) g1 g2 (ins1 outs1 ins2 outs2
    to target from : block_id),

((evaluate c1) s1) = (s2, {| graph := g1; ins := ins1 ; outs := outs1 |}) →
((evaluate c2) s2) = (s3, {| graph := g2; ins := ins2 ; outs := outs2 |}) →
In to (inputs g1) →

(denote_cflang (CSeq c1 c2) s1 from to)
          ≈
(d  ← denote_cfg g1 from to ;;
match d with
| inr dv ⇒ ret (inr dv)
| inl (src, target) ⇒
   if eqb_bid target (hd default_bid outs1)
   then denote_cfg g2 (hd default_bid outs1) (hd default_bid ins2)
   else denote_cfg g2 src target
end).
```

Figure 12: Denotation of CSeq

**Meta-theory of the freshness monad** The freshness monad maintains a state with a counter for the label generation. At each generation of a fresh label, the counter is incremented and the state is ensuring the freshness of the labels. We denote $\sigma_i$ a state and $CB(\sigma_i)$ the counter of the state. For *simplicity*, we conflate here the natural numbers generated by the counter with the labels themselves; in practice, there are a few technical difficulties involved in bridging the gap, but they essentially boil down to the naming schema used `name : nat → block id` being injective. The generation of a fresh label from a state $\sigma_0$ creates a label $CB(\sigma_0)$ and returns a new state $\sigma_1$ such that $CB(\sigma_0) < CB(\sigma_1)$.

The compilation of `CFLang`, $compile\ c\ \sigma_0 = (\sigma_1, dg)$, generates label in the interval $CB(\sigma_0)$ to $CB(\sigma_1)$. Formally, we have $\forall\ label \in (labels\ dg),\ label \in [CB(\sigma_0), CB(\sigma_1)]$, which is the main theorem of the meta-theory. As a corollary, we can prove that if two pieces of code $c_1$ and $c_2$ are compiled successively, $compile\ c_1\ \sigma_0 = (\sigma_1, dg_1)$ and $compile\ c_2\ \sigma_2 = (\sigma_3, dg_2)$ with $CB(\sigma_1) \leq CB(\sigma_2)$, all the labels of $dg_2$ are strictly greater than the labels of $dg_1$. In particular, it ensures the disjointness between the labels of $dg_1$ and the labels of $dg_2$. Similar reasoning let us define the set of invariant necessary to ensure the preconditions of the equations of denotation of the unsafe combinators

**Recipe** The DSL can evolve, depending the needs and its expressivity. Thus, we propose a recipe to add new combinator to the DSL:

1. Define the unsafe combinator given the graphs and the labels we want to link — it may contain additional information such as a condition;
2. Define its equation of denotation and prove it: it may require some preconditions encapsulated in a well-formedness property;
3. Add a new term in syntax of `CFLang`, define its compilation in SCFG and prove that the invariant still correct;
4. Define (and prove) the lemma stating that using the DSL, the WF hypotheses needed for the denotation lemma are ensured.

For instance, we propose as further work to apply this recipe to the implementation of a *for loop*, as used in HELIX notably.

# 4 Case study - Compiler IMP to VIR

In order to demonstrate the efficiency and the usability of our approach, we perform a case study by writing a compiler from IMP to VIR, using `CFLang` for the label generation.

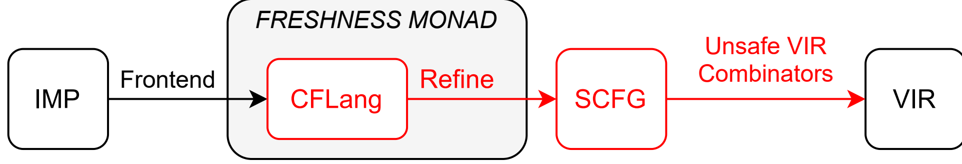



Figure 13: Compilation chain of the compiler from IMP to VIR

**Compiler** The compiler from IMP to `CFLang` is pretty straightforward. It compiles the IMP expressions into VIR instructions. The register names are generated thanks to a freshness monad, but is not managed by out abstractions yet — it could be interesting to integrate the register name generation into an abstraction as a further work. The IMP control-flow instructions are compiled into `CFLang` instructions. Since the combinators and `CFLang` were inspired by IMP, for each IMP control-flow instruction, it exists its `CFLang` version. Finally, to complete the compile chain, it suffices to compile the generated `CFLang` program into a SCFG, extract the CFG and add the remaining information (such as the entry point) to create a VIR program. Fig. 13 summarize the compilation chain. The new abstract layers in the chain are the red parts of the figure. We recall that thanks to the `CFLang`, the *unsafe* VIR combinators are safely used.

**Correctness** Since the abstraction `CFLang` to VIR produces a well-formed code, the whole compilation chain from IMP also produces a well-formed code.

The whole semantic of IMP has been defined in [3]. We recall it in Fig. 14.

```
Fixpoint denote_expr (e : expr) : itree eff value :=
match e with
| Var v     ⇒ trigger (GetVar v)
| Lit n     ⇒ ret n
| Plus a b ⇒ l  ← denote_expr a ;; r  ← denote_expr b ;; ret (l + r)
| _ ⇒ ...
end.

Fixpoint denote_imp (s : imp) : itree E unit :=
match s with
| Assign x e ⇒ v  ← denote_expr e ;; trigger (SetVar x v)
| Seq a b ⇒ denote_imp a ;; denote_imp b
| If i t e ⇒
    v  ← denote_expr i ;;
    if is_true v
    then denote_imp t
    else denote_imp e
| While t b ⇒
    iter (fun _ ⇒
           v  ← denote_expr t ;;
           if is_true v
           then denote_imp b ;; ret (inl tt)
           else ret (inr tt))
| Skip ⇒ ret tt
end.
```

Figure 14: Denotational semantic of IMP

There is a typeclass constraint `ImpState -< E`, indicating that `E` permits `ImpState` actions. The actions of `ImpState` are `GetVar (x : var) : ImpState value` and `SetVar (x : var) (v : value) : ImpSatet unit`.

We recall that to prove the correctness of a compiler using the ITrees, we have to prove that the ITree representing the semantic of the source code is bisimilar (regarding the *eutt* abstraction) with the ITree representating the semantic of the compiled code. But, the memory model in both ITrees are not necessarily the same and the bisimulation does not hold over uninterpreted ITrees. Here, this is because the compiler introduces intermediate registers. In a first place, we have to explain how the reads/writes in the memory of IMP are related with the effects on the memory of VIR. In other words, we have

to define an invariant relating the environment of IMP with the environment of VIR. Informally, if a variable $k$ in the IMP environment is associated to a value $v$, it should exist a register associated to $k$, containing an address pointing to the same value $v$. Formally, the relation is defined in Fig. 15.

```
Definition Rmem (vmap : StringMap.t int)(env : Imp.env) (venv : local_env) (vmem :
    memory_stack) : Prop :=
∀ k v, alist_find k env = Some v  ←>
( ∃ reg, StringMap.find k vmap = Some reg ∧
∃ addr, alist_find (Anon reg) venv = Some (UVALUE_Addr addr) ∧
∃ v32, read vmem addr (DTYPE_I (Npos 32\%positive)) = inr (UVALUE_I32 v32) ∧
Int32.intval v32 = Z.of_nat v ).
```

Figure 15: Relation environment IMP and memory VIR

In order to compare the itrees of IMP and VIR, we have to interpret the effects. But it suffices to interpret only the four first levels on VIR to compare the itrees. As shown in Fig. 5, the four first levels of interpretations concerns the local environment, global environment and memory events whereas the last levels concerns the undefined behaviors. IMP does not generates undefined behaviors, but it generates memory event. Thus, the lowest level of interpretation is the fourth, allowing us to reason on the simplest monad manipulating the memory.

The last step of the verified compiler is to prove its correctness theorem. During the internship, I had no time to prove the complete theorem of correctness, which is indeed an ongoing work. As a preliminary result, writing a compiler targeting VIR is much easier with `CFLang` as intermediate representation. Even if I had no time to do the whole semantic construct, the equational theory of CFLang seems to work and ease the reasoning. Unfortunately, it remains some preconditions not managed by the theory of the freshness monad, such as the `has_post` precondition for the while combinator in Fig. 9, stating that the denotation of the body should jump in the correct output block.

# 5 Related Work

**Reason over a freshness monad** In the paper Tale of Monad in Coq [8], *P.Nigron and P.-E. Dagand* deal with the verification of imperative programs in a proof assistant. The main issue is to represent the effects of the imperative programs in a pure language — i.e. without effects — such as Gallina (in Coq). The previous work in this domain led to a generalization of monads and algebraic effects. In particular, this work focuses on the use of the freshness monad to generate labels, a common monadic problem. They propose a dedicated separation logic to reason about the monadic effects (here, the freshness), and they propose an implementation of this logic by instantiating the `Iris/MoSel` framework [9]. The separation logic allows to reason about local invariant, whereas the global invariant of disjointness holds thanks to the separation logic.

The logic could be use to ease the reasoning on the freshness monad for the engineer, but it does not spare the label generation. The main hurdle to compare our work is that Vellvm does not support the `Iris` framework yet. However, our work goes further and adds two layers of abstractions: one for the labels generation and one for the compositional combination of cfgs. It completely avoids the reasoning on the labels and allows to reason purely equationally, which is far easier than doing the reasoning with the separation logic in Iris.

**Nameless Label** As explained previously, we propose a set of combinators not statically typed and living in a named world. Theses approaches have advantages and drawbacks: it is easier to reason over named labels and to define the semantic equations of the combinators than a statically typed structure, but the preconditions needed to use them have to be ensured by a second layer of abstraction.

During its internship, Nicolas Chappe has explored another approach of the combinators, using anonymous labels. He proposed a set of low-levels combinators, such as merge (new graph by concatenation of two graphs side-by-side, but without linking between them) or reorder (permutation of the inputs/outputs), statically typed using vectors with anonymous labels. The intermediate representation is a function taking a list of labels and naming the blocks of the graphs. It ensures that, if the input

list of labels has no duplicate, the uniqueness of the block id of the graph holds. This approach is syntactically elegant. It is based on three categories of labels: the inputs, the outputs and the internals, which is similar with the approach of the ASM labels in [3]. However, the semantic characterization of such combinators is hardly non-trivial, and their proof needs a relabeling meta-theory and non-trivial arithmetic.

The work of *A. Rouvoet et al.* [10] addresses similar issues, in a different context: how to deal with the uniqueness and the well-definition of label name in the context of intrinsically typed compilation targeting labeled bytecode. They propose a solution based on a *nameless* and *co-contextual* representation of the typed labels. A clever use of *proof-relevant separation algebra* [11] (based on separation logic) abstracts the co-context and their composition, and provides an high-level language to reason about the nameless labels. The framework, implemented in Agda, is a bit complicated to compare with our in Coq.

# 6 Conclusions and future work

## 6.1 Conclusions

**Summary** Verify the correctness of a compiler is often closely bound with well-formedness of the generated code, in particular when the compiler introduces new names. The usual way to deal with name generation is to define an ad-hoc freshness monad, specific to the compiler, but it is annoying to prove its correctness and not interesting semantically. The Intermediate Representation of LLVM led to the creation of various frontend. For a project aspiring to formalize the intermediate representation of LLVM, we do not want to prove the theory of an ad-hoc freshness monad for each new verified compiler targeting VIR.

In consequence, we propose to abstract the label generation and to embed the freshness monad into a DSL, `CFLang`. The language provides a set of high-level combinators over VIR control-flow graphs. Moreover, the library provides a set of denotation lemmas, allowing to construct the semantic of the compiler equationally. Finally, an ongoing work tends to demonstrate the usability of the equation of denotation to prove the correctness of a compiler from IMP to VIR, using `CFLang` for the label generation.

## 6.2 Limits and Future directions

**Expressitivy** The combinators and the DSL were inspired by the IMP language. Thus, we wanted that the combinators and `CFLang` to be at least expressive enough to compile IMP. A natural question we ask is the expressivity of `CFLang`. Specially, is it expressive enough to compile real world language ? In order to push the limits of our approach on a real-world compiler, it could be interesting to use it for the label generation of HELIX. It probably needs some remaining work to easily use the combinators with HELIX, such as implement a new combinator for the *for loops.*

In addition, the abstractions were initially design to deal with the *code generation.* An interesting follow-up the question of the *code transformation* - i.e. optimizations on `CFLang`. Here too, is the language expressive enough to do optimizations on the CFG ? Thus, a future direction is to use our abstractions to do optimizations over control-flow graphs (e.g. “loop unfolding”).

The main expressivity limit of the `CFLang`, and the underlying combinators, is that it does not handle the phi-nodes, an important concept in SSA. A further step could be to construct a `CFLang` program from a VIR cfg, but the abstractions needs to manage the phi-nodes, which is not the case currently.

**Renaming** Create a new empty block to do the linking in order to avoid the relabeling is a questionable design choice. It adds useless blocks and is based on a later elimination of the dead block optimization ; an additional step which could be spared.

**Low level combinators** Finally, we could try an approach with low-levels combinators inspired by the combinators of Nicolas, following the recipe to design such combinators (merge, reorder, ...) and compare the expressivity with the high-level combinators.

# References


[1] C. Lattner and V. 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*, 2004.

[2] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, 2009.

[3] L.-y. Xia, Y. Zakowski, P. He, *et al.*, "Interaction trees: Representing recursive and impure programs in coq," no. POPL, 2019.

[4] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic, "Modular, compositional, and executable formal semantics for llvm ir," *Proc. ACM Program. Lang.*, no. ICFP, 2021.

[5] V. Zaliva and F. Franchetti, "Helix: A case study of a formal verification of high performance program generation," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, 2018.

[6] W. Swierstra, "Data types à la carte," *J. Funct. Program.*, 2008.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, 1991.

[8] P. Nigron and P.-E. Dagand, "Reaching for the Star: Tale of a Monad in Coq," in *12th International Conference on Interactive Theorem Proving (ITP 2021)*, 2021.

[9] R. Krebbers, J.-H. Jourdan, R. Jung, *et al.*, "Mosel: A general, extensible modal framework for interactive proofs in separation logic," *Proc. ACM Program. Lang.*, no. ICFP, 2018.

[10] A. Rouvoet, R. Krebbers, and E. Visser, "Intrinsically typed compilation with nameless labels," *Proc. ACM Program. Lang.*, no. POPL, 2021.

[11] A. Rouvoet, C. Bach Poulsen, R. Krebbers, and E. Visser, "Intrinsically-typed definitional interpreters for linear, session-typed languages," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020.