Relational reasoning on monadic semantics

Benjamin Bonneau

▶ To cite this version:

Benjamin Bonneau. Relational reasoning on monadic semantics. 36es Journées Francophones des Langages Applicatifs (JFLA 2025), Jan 2025, Roiffé, France. 10.5281/zenodo.14508391 . hal-04859423

HAL Id: hal-04859423
https://hal.science/hal-04859423v1

Submitted on 30 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Relational reasoning on monadic semantics

Benjamin BONNEAU[1]

[1]Université Grenoble Alpes - CNRS - Grenoble INP - Verimag, France

Effectful programs can be given denotational semantics using expressive enough domains. With applications to formally verified compilation in mind, I mechanise an axiomatization of domains with monadic and iteration operators. Refinements between semantics are derived within this abstraction using heterogeneous relations. By using appropriate rules, it is possible to reason about divergence, non-determinism and undefined behaviors. I give a model based on Labeled Transition System which implements those effects. I demonstrate the use of the library with a compositional proof of a Dead Code Elimination transformation.

1 Introduction

Context Formal program semantics is a prerequisite for reasoning about programs and their transformations, and in particular for formally verified compilation. *Denotational* semantics are based on a domain, informally a collection of mathematical objects expressive enough to capture the features of the language. They associate programs to elements of the domain compositionally, using the operations on the domain, by induction on the syntax. This approach thus separates the derivation of mathematical theories about the domain from the reasoning on programs expressed in some syntax. The benefit is twofold. Firstly, reasoning is not limited by the syntax of the language since the domain can be more expressive. Secondly, multiple languages can share the same domain, which enables some factorizations and simplifies reasoning about transformations between them.

In contrast with several implementations of verified compilers [Ler09, KMNO14] which rely on *operational* semantics, the second iteration of Vellvm [ZBY$^+$21] uses denotational semantics. With their approach, Zakowski et al. define semantics and prove transformations on them in a compositional fashion. Remarkably they use non-deterministic semantics, whereas determinism often plays an important role in operational approaches [Ler09, §2.1].

Zakowski et al. rely crucially on the recent development of *interaction trees* (ITrees) [XZH$^+$20], a coinductive type able to represent divergent semantics which interact with their environment. ITrees and derived structures are interesting domains for the definition of denotational semantics, thanks to their monadic and iteration operators. The main notion used to reason on them is the "equivalence up to Tau", a coinductive relation which equates programs that differ only by finite sequences of internal steps, while ensuring the preservation of diverging behaviors. Xia et al. [XZH$^+$20] proved a set of lemmas on this relation and its interaction with the operators from which the correctness of some program transformations can be proven without explicit coinduction by using some relational reasoning.

However, the class of transformations that can be derived from those lemmas remains unclear. In particular, Zakowski et al. [ZBY$^+$21, §5.3] had to fall back on coinductive reasoning for proving an optimisation. Another limitation of a direct use of ITrees is

that they only model a restricted set of effects. Fortunately, it is possible to derive from them more expressive structures to model for instance the presence of a global state or non-determinism [YZZ22, §4.1]. However formal reasoning is tied to the structure used, which depends on the effects considered. ITrees offer a way of abstracting away some effects by considering the corresponding computations as uninterpreted during chosen parts of the proofs [ZBY⁺21, §4.3,§5.3]. Another idea is to axiomatize the properties satisfied by ITrees and structures derived from them [YZZ22].

Approach I take this idea even further by defining an axiomatic theory for reasoning formally with domains which support the same operators as ITrees. Here, axiomatic means that the theory is not tied to a specific domain and can be applied to any model which implements my interface. I consider refinement relations on those domains and derive a relational theory, using a small set of axioms. It enables some optimisation proofs that are generic in the domain, including loop and inter-procedural transformations as well as function inlining. The development is implemented as a Rocq[1] library and is available online[2]. Hyperlinks ◇ point to the implementations of the concepts discussed.

Contributions I mechanised an axiomatic theory which extends the existing relational theory on ITrees with reasoning on inter-procedural transformations. I also give axioms on non-determinism and undefined behaviors, for models with those effects. From the main axioms, I derive a proof approach for loop transformations for which previous works [XZH⁺20, ZBY⁺21] had to fall back on coinductive proofs specific to their models. I prove that ITrees are still a model of this extended theory and show that Labeled Transition Systems are another one, which natively supports non-determinism and undefined behaviors.

Outline Section 2 describes a simplified version of the axiomatic theory, a subset of the existing relational theory on ITrees. Section 3 presents a method to derive the correctness of some loop transformations, some of the aforementioned proofs are described as examples. The theory is extended with events in section 4 to handle mutually recursive functions. Relations on events are introduced to reason on inter-procedural transformations. A model of the theory, using Labeled Transition Systems, is given in section 5. The definition of some operators and of the refinement relation is discussed. Additional operators and axioms are given to reason on non-determinism and undefined behaviors. Finally, section 6 shows how the theory is used, and demonstrates on a Dead Code Elimination how the axiomatic approach can abstract away effects during the proof of transformations. The section ends with a comparison with the aforementioned abstraction approach of Zakowski et al. [ZBY⁺21].
Except for parts with explicit references, sections 3 and 5 are not prerequisites for the others.

2 Axiomatic interface

In my theory, a domain is a family of types `S : Type → Type` used to represent semantics. The inhabitants of `S A` are semantics returning values of type `A`. The family `S` is a parameter of the development, I expect it to be a monad, equipped with the following operations (◇):

- `ret [A] (x : A) : S A` is a semantics that has no effects and returns `x`.
- `bind [A B] (u : S A) (k : A → S B) : S B` is the semantics that runs `u` until it returns a value `x : A`, then runs `k x`.

[1] `https://coq.inria.fr`
[2] `10.5281/zenodo.14508391` or `https://gricad-gitlab.univ-grenoble-alpes.fr/these-bbonneau/artifacts/tree/JFLA25`

- `iter [A B] (f : A → S (A + B)) (ini : A) : S B` is the semantics that repeatedly runs `f` applied to an iteration state of type `A`. Initially this state is `ini`. If `f` returns a value `inl x`, the iteration continues from the new state `x`. If it returns `inr y`, `iter` stops and returns `y`.

Arguments within `[...]` can be inferred from the others and will be omitted unless a prefix `@` is used. I will use the notation `x ← u; k` for `bind u (λx. k)`. The `k` and `f` in `bind` and `iter` are functions in the host language (Rocq) and can use any pure operations. In particular, one has access to conditionals or more general pattern matchings. Although those descriptions of the semantics of the operators are a valuable source of intuition, they are only informal since the actual definitions depend on the instantiation of `S`. In particular, there is no notion of "execution" in the theory. The meaning of the operators is captured more abstractly with axioms about their interactions with relations on semantics.

I take such refinement relation as parameter:

```
sem_ref [A B] (R : Rel A B) : Rel (S A) (S B)
```

where `Rel` is the family of heterogeneous relations: `Rel A B := A → B → Prop`. The left column of Figure 1 defines notations and operations that will be used on `Rel`. A relation `sem_ref R u v` should be understood as "`v` refines `u`" and will be denoted by $u \gtrsim_R v$. In a compilation setting, `u` is the semantics of the source program and `v` the semantics of the target. A post-condition `R` is used to compare values returned by `u` and `v`, which do not have identical types in general. Since `sem_ref R` is a refinement, it may be asymmetric, even when `R` is symmetric. It is useful to consider a stronger relation $u \approx_R v := u \gtrsim_R v \wedge v \gtrsim_{\dagger R} u$

	`Type`	`EventFam.t`
empty	`False`	`EventFam.emp := λ_. False`
sum	`Inductive A + B :=` `| inl (x : A)` `| inr (y : B)`	`D + E := λA. D A + E A`
function	`A → B`	`EventFam.Fun D E := ∀ [A], D A → E A`
relation	`Rel A B := A → B → Prop`	`EventFam.Rel D E := ∀ [A B : Type] (d : D A) (e : E B) (post : Rel A B), Prop`
identity	$\mathtt{id}\ x\ y := (x = y)$	$\dfrac{}{\mathtt{id}\ e\ e\ =}$
composition	$(R_0;R_1)\ x\ z := \exists y.\, R_0\ x\ y \wedge R_1\ y\ z$	$\dfrac{R_0\ e_0\ e_1\ P_0 \qquad R_1\ e_1\ e_2\ P_1}{(R_0;R_1)\ e_0\ e_2\ (P_0;P_1)}$
order	$R_0 \le R_1 := (\forall x\, y.\, R_0\ x\ y \Rightarrow R_1\ x\ y)$	$R_0 \le R_1 := \left(\begin{array}{l}\forall A\,B\,(d : D\ A)\,(e : E\ B)\,P_0.\\ \quad R_0\ d\ e\ P_0 \Rightarrow\\ \quad \exists P_1.\, R_1\ d\ e\ P_1 \wedge P_1 \le P_0\end{array}\right)$
sum	$\dfrac{R_0\ x\ y}{(R_0+R_1)\ (\mathtt{inl}\ x)\ (\mathtt{inl}\ y)}$ $\dfrac{R_1\ x\ y}{(R_0+R_1)\ (\mathtt{inr}\ x)\ (\mathtt{inr}\ y)}$	$(R_0+R_1)\ A\ B\ d_2\ e_2\ P := ((\lambda d\, e.\, R_0\ d\ e\ P) + (\lambda d\, e.\, R_1\ d\ e\ P))\ d_2\ e_2$
of function	$[f]\ x\ y := (y = f\ x)$	$\dfrac{}{[f]\ e\ (f\ e)\ (=)}$
converse	$\dagger R\ x\ y := R\ y\ x$	$\dfrac{R\ e\ d\ P}{\dagger R\ d\ e\ (\dagger P)}$

Figure 1. Notations and structures on `Type` and `EventFam.t` (used after section 4, ◇)

Relator:

$$\frac{}{\mathtt{u} \gtrsim_{=} \mathtt{u}}\ \textsc{Refl} \qquad \frac{\mathtt{u} \gtrsim_{\mathtt{R_0}} \mathtt{v} \qquad \mathtt{R_0} \leq \mathtt{R_1}}{\mathtt{u} \gtrsim_{\mathtt{R_1}} \mathtt{v}}\ \textsc{Mono} \qquad \frac{\mathtt{u} \gtrsim_{\mathtt{R_0}} \mathtt{v} \qquad \mathtt{v} \gtrsim_{\mathtt{R_1}} \mathtt{w}}{\mathtt{u} \gtrsim_{\mathtt{R_0};\mathtt{R_1}} \mathtt{w}}\ \textsc{Trans}$$

Morphisms:

$$\frac{\mathtt{R\ x\ y}}{\mathtt{ret\ x} \gtrsim_{\mathtt{R}} \mathtt{ret\ y}}\ \textsc{RefRet} \qquad \frac{\mathtt{u} \gtrsim_{\mathtt{r}} \mathtt{v} \qquad \forall \mathtt{x\,y}.\, \mathtt{r\ x\ y} \Rightarrow \mathtt{f\ x} \gtrsim_{\mathtt{R}} \mathtt{g\ y}}{\mathtt{bind\ u\ f} \gtrsim_{\mathtt{R}} \mathtt{bind\ v\ g}}\ \textsc{RefBind}$$

$$\frac{\forall \mathtt{x\,y}.\, \mathtt{r\ x\ y} \Rightarrow \mathtt{f\ x} \gtrsim_{\mathtt{r+R}} \mathtt{g\ y} \qquad \mathtt{r\ xini\ yini}}{\mathtt{iter\ f\ xini} \gtrsim_{\mathtt{R}} \mathtt{iter\ g\ yini}}\ \textsc{RefIter}$$

Monad laws:

$$\frac{}{\mathtt{bind\ (ret\ x)\ k} \approx \mathtt{k\ x}}\ \textsc{RetL} \qquad \frac{}{\mathtt{bind\ u\ ret} \approx \mathtt{u}}\ \textsc{RetR}$$

$$\frac{}{\mathtt{bind\ (bind\ u\ f)\ g} \approx \mathtt{x} \leftarrow \mathtt{u};\ \mathtt{bind\ (f\ x)\ g}}\ \textsc{Assoc}$$

Iteration:

$$\frac{}{\mathtt{iter\ f\ ini} \approx \mathtt{r} \leftarrow \mathtt{f\ ini};\ \begin{cases} \mathtt{iter\ f\ x} & \text{if } \mathtt{r = inl\ x} \\ \mathtt{ret\ y} & \text{if } \mathtt{r = inr\ y} \end{cases}}\ \textsc{IterUnfold}$$

$$\frac{}{\mathtt{iter\ (iter\ f)\ x} \approx \mathtt{iter}\ (\lambda \mathtt{x.\ r} \leftarrow \mathtt{f\ x;\ ret\ (mergeL\ r)})\ \mathtt{x}}\ \textsc{IterCodiagonal}$$

$$\text{where: } \mathtt{mergeL\ (r : A + (A + B)) : A + B} := \begin{cases} \mathtt{inl\ x} & \text{if } \mathtt{r = inl\ x} \\ \mathtt{y} & \text{if } \mathtt{r = inr\ y} \end{cases}$$

Figure 2. Axioms assumed about `ret`, `bind`, `iter` and `sem_ref` (◇)

to factorize the proofs of refinements in both directions. It is still heterogeneous but is *conversive* [LGL17], that is $\mathtt{u} \approx_{\mathtt{R}} \mathtt{v} \Leftrightarrow \mathtt{v} \approx_{\dagger\mathtt{R}} \mathtt{u}$. Using the equality as post-condition yields a homogeneous (that is, between semantics with the same return type) equivalence relation $\mathtt{u} \approx \mathtt{v} := \mathtt{u} \approx_{=} \mathtt{v}$.

Finally, I require that the operators and the refinement relation satisfy some axioms, listed in Figure 2. Those axioms correspond to properties identified by [XZH+20] on ITrees. They fall into four categories:

- The properties Refl, Mono and Trans are a subset of the axioms of *relators* [LGL17, Definition 5]. From them, one derive the same properties for $\approx_{\mathtt{R}}$ and the fact that $\approx$ is an equivalence.
- RefRet, RefBind and RefIter assert that the operators are morphisms for `sem_ref`: applied to related arguments, they produce related results. By composing refinements on subparts of the semantics, they enable modular proof approaches. Again, the same properties (EqRet, EqBind and EqIter) are derived for $\approx_{\mathtt{R}}$. They also imply that $\approx$ is a congruence that can be rewritten under `bind` and `iter`.
- RetL, RetR and Assoc are the monad laws. They only need to hold for $\approx$, which may be weaker than the equality.
- IterUnfold unrolls the first iteration of an `iter`, according to our informal description of the iterator. IterCodiagonal merges two nested `iter` on the same type of state

into a single `iter`. They are the axioms *fixpoint* and *codiagonal* of complete Elgot monads [GMR16, Definition 3.1] with **Set** as the base category.

3 Using the codiagonal property

In this section I will show how to use IterCodiagonal to derive new equivalences.

3.1 Generalized codiagonal property

The first thing to notice is that, although I assumed it only in the case where the two `iter`s use the same iteration type, the codiagonal property may be generalized to nested iterations where the inner one depends on the current state of the outer one.

Lemma 1 (◇)**.** For any types `A` and `C`, family `B` of types indexed by `A`, functions `f (x : A) (y : B x) : S (B x + (A + C))` and `yini (x : A) : B x` and initial state `xini : A`:

$$\begin{gathered}\texttt{iter } (\lambda\texttt{x. iter } (\lambda\texttt{y. f x y}) \texttt{ (yini x)) xini} \\ \approx \\ \texttt{iter } (\lambda\texttt{(x, y). r} \leftarrow \texttt{f x y; ret (hcodiag x r)) (xini, yini xini)}\end{gathered}$$

Where the result iterates over dependent pairs $\texttt{D} := \texttt{\{x : A \& B x\}}$ and:

$$\texttt{hcodiag (x : A) (r : B x + (A + C)) : D + C} := \begin{cases}\texttt{inl (x, y)} & \text{if } \texttt{r = inl y} \\ \texttt{inl (x, yini x)} & \text{if } \texttt{r = inr (inl x)} \\ \texttt{inr z} & \text{if } \texttt{r = inr (inr z)}\end{cases}$$

Proof. This more general rule is derived from the particular case of Figure 2 by using in a first step EqIter to re-index the nested `iter`s on the same type `D` of state. This first derivation has the following shape:

$$\dfrac{\forall \texttt{x : A.}\ \dfrac{\forall \texttt{y : B x.}\ \dfrac{\dots \quad \forall \texttt{r.}\ \dfrac{(\texttt{[I1 x]} + (\texttt{[I0]}+=))\ \texttt{r (f1r x r)}}{\texttt{ret r} \approx_{\texttt{[I1 x]}+(\texttt{[I0]}+=)} \texttt{ret (f1r x r)}}\,\textsc{EqRet}}{\texttt{r} \leftarrow \texttt{f x y; ret r} \approx_{\texttt{[I1 x]}+(\texttt{[I0]}+=)} \texttt{f1 (x, y)}}\,\textsc{EqBind} \quad \dots}{\texttt{iter (f x) (yini x)} \approx_{\texttt{[I0]}+=} \texttt{iter f1 (x, yini x)}}\,\textsc{EqIter} \quad \dots}{\texttt{iter } (\lambda\texttt{x. iter (f x) (yini x)) xini} \approx \texttt{iter (iter f1) (xini, yini xini)}}\,\textsc{EqIter}$$

where:

$$\begin{aligned}&\texttt{f1 (x, y)} := \texttt{r} \leftarrow \texttt{f x y; ret (f1r x r)} \\ &\texttt{f1r (x : A) (r : B x + (A + X)) : D + (D + C)} := \\ &\qquad\begin{cases}\texttt{inl (x, y)} & \text{if } \texttt{r = inl y} \\ \texttt{inr (inl (x, yini x))} & \text{if } \texttt{r = inr (inl x)} \\ \texttt{inr (inr z)} & \text{if } \texttt{r = inr (inr z)}\end{cases} \\ &\texttt{I0 x} := \texttt{(x, yini x)} \qquad \texttt{I1 x y} := \texttt{(x, y)}\end{aligned}$$

This equivalence is rewritten using the transitivity of ≈. The proof is then concluded by using IterCodiagonal followed by some simplifications using the monad laws. ☐

3.2 A method for proving loop transformations

I will now explain how to perform some loop transformations using the codiagonal property. As a first example, I will prove the following equivalence between two infinite iterations:

Example 1 ($\diamond$). For any type `A` and body `f : A → S A`,

$$\texttt{loop1 : S False} := \texttt{iter } (\lambda\texttt{x0. x1} \leftarrow \texttt{f x0; ret (inl x1)) xini}$$
$$\approx$$
$$\texttt{iter } (\lambda\texttt{x0. x1} \leftarrow \texttt{f x0; x2} \leftarrow \texttt{f x1; ret (inl x2)) xini} =: \texttt{loop2}$$

This is the equivalence needed to justify loop unrolling of `while(true) { f }` into `while(true) { f; f }`. It is also a simplified version of a lemma that Zakowski et al. [ZBY+21, §5.3] needed for a block fusion optimisation and for which they used a proof by coinduction on their domain based on ITrees.

Proof of example 1. In order to prove this equivalence, we need to relate two consecutive iterations of `loop1` to a single iteration of `loop2`. This, however, cannot be achieved directly with EqIter which requires the two `iter`s to be synchronized at each iteration. One thus need to synchronize the two semantics on a common notion of "steps". In this example we directly use the iterations of `loop1` as steps, that is, each step will contain exactly one execution of `f` and we only have to change `loop2`. To prove our goal, I will:

1. Rewrite the semantics using nested `iter`s where each innermost iteration is a step.
2. Use the codiagonal property to merge the nested iterations into a single `iter`.
3. Use EqIter to perform a synchronized equivalence proof between the resulting source and target `iter`s.

For the first part, I use IterUnfold twice to turn the first `bind` of `loop2` into a finite `iter`:

$$\begin{aligned}\texttt{iter (f1 x0) (inl ())} &\approx \texttt{x1} \leftarrow \texttt{f x0; iter (f1 x0) (inr x1)}\\ &\approx \texttt{x1} \leftarrow \texttt{f x0; x2} \leftarrow \texttt{f x1; ret (inl x2)}\end{aligned}$$

where:

$$\texttt{f1 (x0 : A) (y : unit + A) : S ((unit + A) + (A + False))} :=$$
$$\begin{cases}\texttt{x1} \leftarrow \texttt{f x0; ret (inl (inr x1))} & \text{if } \texttt{y = inl ()}\\ \texttt{x2} \leftarrow \texttt{f x1; ret (inr (inl x2))} & \text{if } \texttt{y = inr x1}\end{cases}$$

I then apply the generalized codiagonal property, and obtain:

$$\texttt{loop2} \approx \texttt{iter } (\lambda\texttt{(x0, y). r} \leftarrow \texttt{f1 x0 y; ret (hcodiag x0 r)) (xini, inl ())}$$

where the iteration is performed over `D := {x0 : A & y : unit + A}`. Finally, the equivalence between this `iter` and `loop1` is proven using the following function as invariant (that is `r := [inv]` in EqIter):

$$\texttt{inv ((x0, y) : D) : A} := \begin{cases}\texttt{x0} & \text{if } \texttt{y = inl ()}\\ \texttt{x1} & \text{if } \texttt{y = inr x1}\end{cases}$$

□

3.3 CFG representation

This proof approach can be understood graphically by breaking down the semantics into Control Flow Graphs (CFG). In Figure 3, CFG are represented as graphs whose nodes are either straight rectangles containing a fragment of the semantics or rounded rectangles containing a variable bound by a combinator `bind` or `iter`. Edges denote the informal execution paths of the semantics. Variables are assigned values when the execution reach them. Except for the entry edge, the assigned value is computed using the result of the last semantics node. Note that since there is no fixed notion of step for semantics, one could have

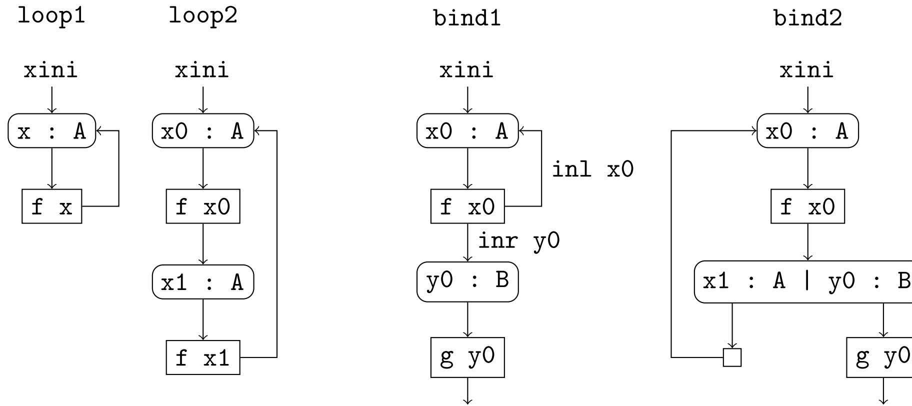

Figure 3. Decomposition of semantics as Control Flow Graphs. `x1 : A | y0 : B` represents a variable of type `A + B`.

for instance chosen to consider a whole iteration of `loop2` as a single node. This would have led to a different CFG, but that could not be related to `loop1`. In the CFG representation, the invariant `inv` used for EqIter is a map from the nodes containing variables of `loop2` to those of `loop1`. In Figure 3, it is obtained by forgetting the indices of the variables. For the premise of EqIter to be satisfied, the map must commute with the semantics of the nodes and edges. Such maps correspond to the CFG morphisms of Gourdin et al. [GBB+23, §2.1].

3.4 Stuttering steps

I apply the approach to derive the "parameter" (⋄) and "composition" (⋄) equations of [XZH+20, §4.1]. They are instances of the *naturality* and *dinaturality* properties of complete Elgots monads [GMR16], I will present here only the first one:

Proposition 1 ("parameter" ⋄)**.** For any types `A`, `B` and `C`, semantics $\texttt{f : A} \to \texttt{S (A + B)}$ and $\texttt{g : B} \to \texttt{S C}$ and initial state `xini : A`:

$$\texttt{bind1} := \texttt{y} \leftarrow \texttt{iter f xini; g y}$$
$$\approx$$
$$\texttt{iter}\left(\lambda\texttt{x. r} \leftarrow \texttt{f x;}\begin{cases}\texttt{ret (inl x)} & \text{if } \texttt{r = inl x}\\ \texttt{z} \leftarrow \texttt{g y; ret (inr z)} & \text{if } \texttt{r = inr y}\end{cases}\right)\texttt{xini} =: \texttt{bind2}$$

A first choice of CFG decompositions is pictured in Figure 3. Notice that the decomposition of `bind2` contains an empty node, for the right-hand side of the bind when `f` returns `inl x`. I call those nodes "stuttering steps" by analogy with transition systems. Since EqIter requires a lock-step equivalence between the two `iter`s, we need those empty nodes to be present in both CFGs. It is not the case for the depicted decompositions since there is no corresponding empty node in `bind1`. Fortunately, it is always possible to add stuttering steps while breaking down a semantics into a CFG by using RetL to introduce a `bind` whose left-hand side is considered a step.

4 Events and mutual recursion

Events The `iter` operator is restricted to loops and tail-call recursion. To handle general recursion, the ITree library uses a technique of McBride [McB15]: the semantics of the bodies of the functions are defined independently, by emitting some observable events

at call sites. A mutual recursion operator, `mrec`, combines the semantics by replacing the events with execution of the corresponding semantics. In order to use the same technique, one need to add a notion of events to the domain. Events can be seen as a generalisation of return values: when emitting an event, a semantics provides not only a description of the event emitted (in the case of recursion, the function called and its arguments) but also a continuation waiting for an answer from the event (in the case of recursion, the value returned by the callee). In the same way we describe the values a semantics can return with a `Type`, we will use a family of events to describe the events it can emit. Such family has sort:

$$\texttt{EventFam.t} := \texttt{Type} \rightarrow \texttt{Type}$$

The index of the family is used to classify the events by their type of answer, that is, given `E : EventFam.t` and a type `A`, `E A` is the type of the events of `E` that answer values of type `A`. Figure 1 describes some structures defined on `EventFam.t`, which are similar to those on `Type`. In particular, there are categories of functions and relations on event families.

I will now modify the signature of domains to include as argument the family of events the semantics can emit:

$$\texttt{S : EventFam.t} \rightarrow \texttt{Type} \rightarrow \texttt{Type}$$

The operators `ret`, `bind` and `iter` are assumed to be defined for any family of events. For instance `bind` is generalized as:

```
bind [E A B] (u : S E A) (k : A → S E B) : S E B
```

The signature of `sem_ref` is extended heterogeneously using the relations on events. That is, we now take as parameter:

```
sem_ref [D E] (Re : EventFam.Rel D E)
              [A B] (R : Rel A B)
    : Rel (S D A) (S E B)
```

I will write this modified version $\gtrsim_{\texttt{Re},\texttt{R}}$. Heterogeneous ($\approx_{\texttt{Re},\texttt{R}}$) and homogeneous ($\approx$) equivalences can still be derived using the identity and converse operations in `EventFam.Rel`. The axioms of Figure 2 are adapted to those additions by using the operations on `EventFam.t`.

Operators I then expect the following operators to be defined ($\diamond$):

- `trigger [E A] (e : E A) : S E A` emits an event `e` and returns its answer.
- `map_ev` $[\texttt{E}_0\ \texttt{E}_1]$ `(f : EventFam.Fun` $\texttt{E}_0\ \texttt{E}_1$`) [A] (u : S` $\texttt{E}_0$ `A) : S` $\texttt{E}_1$ `A` replaces the events of `u` by the events given by `f`.
- `interp [D E] (h : ∀ B. D B → S E B) [A] (u : S D A) : S E A` replaces the events of `u` by the semantics given by `h`.
- `mrec [D E] (f : ∀B. D B → S (D + E) B) [A] (ini : D A) : S E A` is a recursive system indexed by `D`. Figure 4 gives an example of a system with two mutually recursive functions. A call (including the argument values) to a function that returns a value of type `B` is represented by an element `d : D B`. The initial call is `ini`. The implementations of the calls are described by `f` using semantics that can perform recursive calls by emitting events `inl d`.

It is possible to define `map_ev` from `interp` and `trigger` or `interp` from `map_ev` and `mrec`. Figure 5 lists the axioms required on those operators. Most of them are similar to some axioms of Figure 2, replacing `ret` by `trigger`, `bind` by `interp` and `iter` by `mrec`. I also assume that `interp` distributes over `bind` and `iter`.

```
let rec f (n : nat) =
  if n <= 1
  then 0
  else g n
and g (n : nat) =
  let m = f (n - 2) in
  m + 1
```

```
Inductive D : EventFam.t :=
  | Df (n : nat) : D nat
  | Dg (n : nat) : D nat.

Definition body [A] (d : D A)
  : S (D + EventFam.emp) A :=
  match d with
  | Df n ⇒ if n <=? 1
           then ret 0
           else trigger (inl (Dg n))
  | Dg n ⇒ m ← trigger (inl (Df (n - 2)));
           ret (m + 1)
  end.

Definition u (n : nat) : S EventFam.emp nat :=
  mrec body (Df n).
```

Figure 4. Example of a system of recursive functions and definition of its semantics using mrec (◇)

Morphisms:

$$\frac{\texttt{Re d e R}}{\texttt{trigger d} \gtrsim_{\texttt{Re,R}} \texttt{trigger e}}\ \textsc{RefTrigger}$$

$$\frac{\forall \texttt{A}_0\, \texttt{A}_1\, \texttt{d}_0\, \texttt{d}_1\, \texttt{r},\ \texttt{Rd d}_0\ \texttt{d}_1\ \texttt{r} \Rightarrow \texttt{h}_0\ \texttt{A}_0\ \texttt{d}_0 \gtrsim_{\texttt{Re,r}} \texttt{h}_1\ \texttt{A}_1\ \texttt{d}_1 \qquad \texttt{u}_0 \gtrsim_{\texttt{Rd,R}} \texttt{u}_1}{\texttt{interp h}_0\ \texttt{u}_0 \gtrsim_{\texttt{Re,R}} \texttt{interp h}_1\ \texttt{u}_1}\ \textsc{RefInterp}$$

$$\frac{\forall \texttt{A}_0\, \texttt{A}_1\, \texttt{d}_0\, \texttt{d}_1\, \texttt{r},\ \texttt{Rd d}_0\ \texttt{d}_1\ \texttt{r} \Rightarrow \texttt{f}_0\ \texttt{A}_0\ \texttt{d}_0 \gtrsim_{\texttt{Rd+Re,r}} \texttt{f}_1\ \texttt{A}_1\ \texttt{d}_1 \qquad \texttt{Rd ini}_0\ \texttt{ini}_1}{\texttt{mrec f}_0\ \texttt{ini}_0 \gtrsim_{\texttt{Re,R}} \texttt{mrec f}_1\ \texttt{ini}_1}\ \textsc{RefMRec}$$

Interpretation:

$$\frac{}{\texttt{u} \approx_{[\texttt{f}],=} \texttt{map_ev f u}}\ \textsc{MapEvEq}$$

$$\frac{}{\texttt{interp h (trigger e)} \approx \texttt{h e}}\ \textsc{InterpTrigger} \qquad \frac{}{\texttt{interp trigger u} \approx \texttt{u}}\ \textsc{InterpTriggerH}$$

$$\frac{}{\texttt{interp g (interp h) u} \approx \texttt{interp } (\lambda\texttt{_ d. interp g (h d)) u}}\ \textsc{InterpAssoc}$$

Recursion:

$$\frac{}{\texttt{mrec f ini} \approx \texttt{interp} \left(\lambda\texttt{_ c.} \begin{cases} \texttt{mrec f d} & \text{if } \texttt{c = inl d} \\ \texttt{trigger e} & \text{if } \texttt{c = inr e} \end{cases}\right) \texttt{(f ini)}}\ \textsc{MRecUnfold}$$

$$\frac{}{\texttt{mrec (mrec f) ini} \approx \texttt{mrec } (\lambda\texttt{_ d. map_ev mergeL (f d)) ini}}\ \textsc{MRecCodiagonal}$$

Figure 5. Axioms assumed about trigger, map_ev, interp and mrec (◇)

Transformations Although ITrees are also indexed by the family of events they emit, the "equivalence up to Tau" described by [XZH+20] is homogeneous over the events[3]. By considering a more general `sem_ref` with a relation on the events, I am able to use RefMRec with an invariant `Rd` between the source and target functions, arguments and return values. In practice, it is useful for several inter-procedural transformations:

- Renaming the functions by relating the corresponding source and target functions with the equality as relation on the results. Unreachable source functions are eliminated by not relating them to any target function.
- Transmitting pre-conditions on the arguments and post-conditions on the return values, for instance the results of inter-procedural analyses.
- Relating a single source function to multiple target functions. Combined with pre-conditions, it allows the introduction of specialized versions of functions.

However, RefMRec is restricted to transformations that preserve the call structure: each call of the target execution matches a call in the source execution. As such, it cannot be used for inlining transformations. This issue is similar to the one we had with RefIter in section 3. In fact, when considering `iter` as a system of tail recursive functions, example 1 becomes an inlining transformation. Again, the codiagonal property allows us to prove non-synchronized equivalences. Function inlining is obtained by the method described in section 3 using MRecCodiagonal instead of IterCodiagonal (◇).

5 A model: Labeled Transition Systems

The theory I have presented so far is parameterized by a domain `S` which represents semantics with axiomatized operations and relations. Since the interface generalizes properties identified on ITrees, they are a possible model on which the theory can be applied (◇). I will now present another model: non-deterministic Labelled Transition Systems.

5.1 Definition and operations

Domain Labeled Transition Systems (LTS) are a classical approach to formalize objects with potentially diverging execution sequences, where the steps can emit some observable events. I will instantiate `S E A` with the type `LTS E A` of systems which emits events of `E` and returns values of type `A`. Formally, `LTS E A` is a record described in Figure 6, which contains an arbitrary type `istate` of internal states and a predicate `step` defining internal transitions. The states are extended using `stateF` to model the interaction of the LTS with its environment. The names `Ret`, `Tau` and `Vis` come from constructors of ITrees with similar meanings. In fact, `LTS` can be seen as a formalisation of ITrees using transition systems instead of coinductive types and with the addition of non-determinism and undefined behaviors.

Definition 1 (◇)**.** The transitions of a system `u : LTS E A` are defined between elements of `state u` by:

$$\frac{\texttt{step u s s'}}{\texttt{Tau s} \xrightarrow{\tau} \texttt{s'}} \qquad \frac{\texttt{e : E B} \quad \texttt{x : B}}{\texttt{Vis e k} \xrightarrow{\texttt{e,x}} \texttt{Tau (k x)}}$$

An execution is a sequence of transitions starting from `ini u`.

Hence, the transitions originating from a state depend on its kind:

[3] Actually, the ITree library also provides a heterogeneous relation, `rutt`, but with a different type of relation on the events.

```
Inductive stateF (E : EventFam.t) (A : Type) (istate : Type) : Type :=
  | Ret (x : A)
  | Tau (s : istate)
  | Vis [B : Type] (e : E B) (k : B → istate)
  | Err.

Record LTS (E : EventFam.t) (A : Type) : Type := {
  istate : Type;
  state := stateF E A istate;
  step   : istate → state → Prop;
  ini    : state;
}.
```

Figure 6. Definition of the LTS type (⋄)

- `Ret x` and `Err` are terminal states, which signal respectively the end of the execution with a result `x` and an undefined behavior.
- `Tau s` is an internal state, from which the system performs invisible steps (also called τ-steps) described by `step u`. It is a predicate, rather than a function, to model internal non-determinism.
- `Vis e k` signals the emission of an event `e`. If the system obtains an answer `x` from its environment, it resumes its execution in the state specified by the continuation `k`. In this case the choice of `x` models external non-determinism.

Operators I implemented all the operations described in section 2 and 4 for `S := LTS` (⋄). For instance:

- `ret x` has no internal states. Its initial state is `Ret x`.
- The set of internal states of `@bind E A B u k` is the union of the states of `u` and of the states of `k x` for all `x : A`. The `step` predicate of `bind` is defined from the `step` predicate of the current sub-semantics `u` or `k x`. The states `Ret x` of `u` are replaced by the corresponding initial states `ini (k x)` of the continuation.
- `@mrec D E f ini` is implemented using call stacks as states. Each frame contains the index `d : D B` of the corresponding function, its current `istate (f d)` and a continuation which stores the stack remaining after `f d` returns. The `step` predicate is defined according to the transitions of the current function. A recursive call corresponds to a state `Vis (inl d')  k` and is replaced by a τ-step that pushes a new frame for `f d'`. When a function returns, its frame is discarded and the new state is obtained from the continuation.

Interpreters A drawback of LTSs with respect to ITrees is that they are not executable since `step` is a predicate. To mitigate this issue I implemented a library to build executable transition systems without non-determinism (using a partial function for `step`, ⋄). It features all the operators we have seen so far, along with a proof that they implement the corresponding LTSs. Using this library, modular interpreters can be built, proven correct and extracted to OCaml (⋄).

5.2 Refinement relations

Behaviors Following [Ler09, §2.1], I define, given a LTS, its set `behaves` of behaviors. Set-inclusion then induces a homogeneous refinement:

$$u \overset{\text{beh}}{\gtrsim} v := \texttt{behaves}\ v \subseteq \texttt{behaves}\ u$$

This notion is however too strong for some transformations (such as Dead Code Elimination) which can remove undefined behaviors. It is relaxed with a trace relating refinement [ABC+21], using an `improves` relation on the behaviors, which allows the behaviors that end with an undefined behavior to be replaced by any behavior that starts with the same events (⋄):

$$u \overset{\text{beh'}}{\gtrsim} v := \forall b \in \texttt{behaves}\ v.\, \exists b' \in \texttt{behaves}\ u.\, \texttt{improves}\ b'\ b \tag{1}$$

This refinement can be used to specify the correctness of the compilation of a whole program (this is for instance the notion used by CompCert in `transf_c_program_preservation`[4]). However, it cannot be generalized easily to a relation heterogeneous over the events.

Simulation To define a heterogeneous `sem_ref`, I instead use a simulation relation. $\texttt{u} \gtrsim_{\texttt{Re,R}} \texttt{v}$ is defined to hold if an only if there exists an invariant `M : Rel (istate u) (istate v)` which holds initially (or after some steps of `u` and `v`) and which satisfies the following simulation diagram (⋄):

$$\begin{array}{ccc} \texttt{s}_0 & \overset{\texttt{M}}{\text{———}} & \texttt{t}_0 \\ \texttt{u}^{+} \Big\downarrow & & \Big\downarrow \texttt{v}^{+} \\ \texttt{s}_1 & \underset{\overline{\texttt{M}}}{\text{- - - -}} & \texttt{t}_1 \end{array}$$

$$\frac{\texttt{R x y}}{\overline{\texttt{M}}\ \texttt{(Ret x) (Ret y)}} \qquad \frac{\texttt{M s t}}{\overline{\texttt{M}}\ \texttt{(Tau s) (Tau t)}} \qquad \frac{}{\overline{\texttt{M}}\ \texttt{Err t}}$$

$$\frac{\texttt{Re d e post} \qquad \forall \texttt{x y. post x y} \Rightarrow \texttt{M (k x) (l y)}}{\overline{\texttt{M}}\ \texttt{(Vis d k) (Vis e l)}}$$

The invariant is extended as $\overline{\texttt{M}}$ to the various kinds of states. States `Vis` are related using `Re` and with an universal quantification on the answers of the events of both sides. Undefined behaviors `Err` are refined by anything, allowing the target to improve the behaviors of the source. Using a particular case of the "FreeSim" approach [CSL+23], the diagram allows both the source and target to perform some τ-steps before re-establishing the invariant. However, in order to rule out infinite stuttering, they are required to both perform at least a step. Since I consider non-deterministic LTSs, I use a backward simulation: the steps of the target `v` are quantified universally whereas the steps of the source `u` are quantified existentially.

Equipped with this refinement, `LTS` satisfies all axioms of section 2 and 4. Since backward simulation implies the refinement of behaviors, it is possible to use the axiomatic theory to prove a correctness property expressed on the behaviors.

5.3 Undefined behaviors and non-determinism

LTSs support undefined behaviors and non-determinism. Those two effects are exposed in the relational theory by defining additional operators with associated refinement rules (Figure 7).

Undefined Behaviors The operator `ub0 : LTS EventFam.emp False` represents an undefined behavior. It is defined as a system without internal state and with `Err` as initial state. Its type exposes that it does not emit any events nor return. It is lifted using `bind` and `map_ev` to `ub [E A] : LTS E A` which is more convenient to use. The empty return

[4] https://github.com/AbsInt/CompCert/blob/6019bc41/driver/Complements.v#L31

$$\frac{}{\texttt{ub0} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{ret ()}}\ \textsc{UbRef0} \qquad \frac{}{\texttt{x} \leftarrow \texttt{ub; u} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{v}}\ \textsc{UbRef}$$

$$\frac{}{\texttt{ret ()} \gtrsim_{\texttt{Re},\top} \texttt{any0 A}}\ \textsc{AnyTrg0} \qquad \frac{\forall \texttt{x} : \texttt{A}.\, \texttt{u} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{k x}}{\texttt{u} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{x} \leftarrow \texttt{any A; k x}}\ \textsc{AnyTrg}$$

$$\frac{\texttt{x} : \texttt{A}}{\texttt{any0 A} \gtrsim_{\texttt{Re},\lambda \texttt{y _}.\, \texttt{y=x}} \texttt{ret ()}}\ \textsc{AnySrc0} \qquad \frac{\exists \texttt{x} : \texttt{A}.\, \texttt{k x} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{u}}{\texttt{x} \leftarrow \texttt{any A; k x} \gtrsim_{\texttt{Re},\texttt{R}} \texttt{u}}\ \textsc{AnySrc}$$

Figure 7. Refinement rules for undefined behaviors and non-determinism (◇).

type of `ub0` already allows us to freely change its continuation, but with the rule UbRef0 we can even remove `ub0` and derive that a source semantics starting with an undefined behavior is refined by anything (UbRef).

Non-determinism The operator `any0 A : LTS EventFam.emp A` represents a non-deterministic choice of a value of type `A`. It is lifted to `any [E] A : LTS E A`. It is implemented with a single initial internal state, which can step to `Ret x` for any `x : A`. The non-determinism is demonic: a refinement asks us to prove that all behaviors of the target are improvements of some behavior of the source (Equation 1). This is reflected by the post-conditions of AnyTrg0 and AnySrc0 but is more apparent with derived rules:

- When `any` is in the target semantics, AnyTrg asks us to prove the refinement for all possible values `x : A`.
- When `any` is in the source semantics, AnySrc lets us choose a value `x : A` to prove the refinement with.

A form of reasoning about non-determinism is thus possible within the axiomatic theory. Interestingly, it is enabled by the addition of some relational rules, without any change to the signature of `sem_ref` or restrictions on the other axioms. Hence, the reasoning approach I have presented so far remains valid. This contrasts with direct proofs by simulation, where non-determinism prevents the use of forward simulations. However, backward simulations are no longer complete with respect to the inclusion of behaviors in presence of non-determinism [AL91, §1.2]. For instance, the following commutation does not hold for some `u`, although the two LTSs have the same behaviors (◇):

$$\texttt{b} \leftarrow \texttt{any bool; x} \leftarrow \texttt{u; ret (b, x)} \not\gtrsim_{\texttt{id},=} \texttt{x} \leftarrow \texttt{u; b} \leftarrow \texttt{any bool; ret (b, x)}$$

The issue is that some `u` (such as `trigger`s or semantics with diverging behaviors) cannot be simulated in a single step. Hence, a backward simulation would need to choose a boolean for the source `any` before knowing the result of the target one. It could thus not ensure that `b` takes the same value on both sides.

6 Using the theory

I will show how to use the library to define the semantics of a toy language and to prove correct a transformation on it. I will also demonstrate the possibility and benefits of proving correct transformations generic in the domain. After defining the syntax of the language, section 6.1 equips it with a semantics parametrized by its domain. Still considering an arbitrary domain, section 6.2 proves the correctness of a transformation. Section 6.3 applies this proof to an effectful domain. This approach to effect abstraction is compared with the technique of Zakowski et al. [ZBY+21] in section 6.4.

function identifier	f : `fsym`	instruction u, v : `instr` ::=	`skip`
variable	x, y : `var`		$\mid$ y = $o(x^*)$
operator	o : `opsym`		$\mid$ y = $f(x^*)$
function	fc : `function` ::= $f(x^*)${u; `return` y}		$\mid$ u;v
program	p : `prog` ::= fc^+		

Figure 8. Syntax of a toy imperative language (◇)

```
Inductive CallE : EventFam.t :=
  mk_call (f : fsym) (args : list val) : CallE val.
Definition var_state := var→ val.
Definition get_function (p : prog) (f : fsym) : S EventFam.emp function := ...
```

$$\texttt{f[x} \leftarrow \texttt{y]} := \texttt{x}' \mapsto \begin{cases} \texttt{y} & \text{if } \texttt{x}' = \texttt{x} \\ \texttt{f x'} & \text{if } \texttt{x}' \neq \texttt{x} \end{cases}$$

```
⟦u⟧instr (s₀ : var_state) : S CallE var_state
⟦skip⟧instr s₀ := ret s₀
⟦x = o(xs)⟧instr s₀ := v←map_ev ... (⟦o⟧op (List.map s₀ xs)); ret s₀[x←v]
⟦x = f(xs)⟧instr s₀ := v←trigger (mk_call f (List.map s₀ xs)); ret s₀[x←v]
⟦u;v⟧instr s₀ := s₁←⟦u⟧instr s₀; ⟦v⟧instr s₁

⟦f⟧function (args : list val) : S CallE val :=...
⟦p⟧prog (entry : fsym) (args : list val) : S EventFam.emp val :=
  mrec (λ _ (mk_call f vs). fc← map_ev ... (get_function p f);
                            map_ev ... (⟦fc⟧function vs))
       (mk_call entry args)
```

Figure 9. Semantics parameterized by the type `val` of values and the domain `S` (◇)

6.1 Syntax and generic semantics

The syntax of the language is defined in Figure 8. A program consists of a list of functions, each of which takes a list of arguments and returns a single value. Functions are implemented using an inductive type of instructions which operate over a state of local variables. Variables can be assigned imperatively the result of an operation or of a function call.

In the first two sections, I will not commit to a specific set of operations nor to a domain for the semantics. Figure 9 defines the semantics of the language for some parameters `val` and `S`. This definition requires that `S` implements the operations I described in sections 2 and 4 and that a semantics is provided for the operators:

$$[\![\texttt{o}]\!]_{\texttt{op}}\ \texttt{(vs : list val) : S EventFam.emp val}$$

This signature gives operators access to all the effects of the domain, except the function calls. Calls are represented using the family `CallE`. An events of this family contains the identifier `f` of the callee and the values `args` of its arguments. The answer is a single value. $[\![\texttt{u}]\!]_{\texttt{instr}}$ $\texttt{s}_0$ describes the semantics of the execution of an instruction `u` from a local state $\texttt{s}_0$ as an element of the domain, which can emit `CallE` events and which returns the local state after `u`. It is defined in a denotational way, by recursion on the syntax. The semantics of a function is derived from the semantics of its body by initialising the local state from the values of the arguments and recovering the returned value from the final local state. Finally, we tie the recursive knot using `mrec` to define the semantics of a program. Given an `entry` function and values `args` for its arguments, $[\![\texttt{p}]\!]_{\texttt{prog}}$ `entry args` is an element of the domain representing the semantics of the whole program, which does not emit any events.

```
g(x) { return x }                      g(x) { return x }
f(x, y) {                              f(x) {
  z = g(x);                              z = g(x);
  w = z + y;                             skip;
  return x              dce_prog         return x
}                         ⟶            }
entry(x) {                             entry(x) {
  y = f(x, x);                           y = f(x);
  return y                               return y
}                                      }
```

Figure 10. Example of a Dead Code Elimination transformation (◇)

6.2 Dead Code Elimination

I will now consider a transformation which removes some local variables and operations which do not affect the observable behavior of the program, a kind of Dead Code Elimination (DCE) or ghost code erasure. Because operators have side-effects, some of them need to be kept even if their results are not used. A semantic criterion is used to exclude them from the DCE:

Definition 2 (`op_removable`)**.** An operator `o` is *removable* if for any arguments `vs`:

$$[\![\texttt{o}]\!]_{\texttt{op}}\ \texttt{vs} \gtrsim_{\top} \texttt{ret ()}$$

that is if, ignoring the value returned, its semantics can be replaced by an effect-free computation.

The transformation is inter-procedural and can remove some function arguments but not function calls. An example is given Figure 10. The general transformation is described syntactically in Figure 11 using predicates which relate corresponding parts of the source and target programs. Those predicates take parameters specifying which variables are removed. The main predicate, `dce_instr lva` $\texttt{lvv}_0$ `src trg` $\texttt{lvv}_1$, describes how a source instruction `src` can be erased into a target instruction `trg`. The parameter `lva : live_args` specifies which function arguments are removed and $\texttt{lvv}_0$ `: live_vars` (resp. $\texttt{lvv}_1$) which local variables are removed at the program point before (resp. after) the instruction. Rule RemoveOp allows the removal of operations which satisfy two conditions: their result must not affect the remainder of the computation and the operator must be removable. The first one is enforced by setting the assigned variable as `Dead` after the instruction. The transformation of a whole program is specified by a predicate `dce_prog entry src trg`. It requires each source function to be erased into the corresponding target function for some global choice `lva` of removal of the function arguments which does not affect the `entry` function.

I prove that a transformation matching this static specification yields a target program which refines the source one:

Theorem 1 (`dce_prog_spec` ◇)**.** If two programs `src` an `trg` are related by `dce_prog` for an `entry` function, then the semantics of `trg` refines the semantics of `src` for any arguments `argsv`:

$$[\![\texttt{src}]\!]_{\texttt{prog}}\,\texttt{entry argsv} \gtrsim_{=} [\![\texttt{trg}]\!]_{\texttt{prog}}\,\texttt{entry argsv}$$

Idea of the proof. The proof is done compositionally, following the structure of the syntactic description by giving a specification of each predicate using the semantics of the objects involved. I use the following relations between calls and states:

Definition 3. Given a specification `lva : live_args` of the removal of function arguments, `dce_Call lva : EventFam.Rel.t CallE CallE` is the relation on events generated by:

$$\frac{\texttt{dce_list (lva f) args args'}}{\texttt{dce_Call lva (mk_call f args) (mk_call f args') eq}}$$

That is, source calls are replaced by target calls of the same function `f` but with arguments filtered using `lva f`. The post-condition is the equality, meaning that corresponding source and target calls return the same value.

Definition 4 (`dce_var_state`)**.** Two variable states $\texttt{s}_0$ and $\texttt{s}_1$ are related for a specification `lvv : live_vars` if they agree on the values of the live variables.

Those relations are used to state the correctness of the erasure of the instructions:

Lemma 2 (⋄)**.** If `trg` is a valid erasure of an instruction `src` according to the syntactic criterion $\texttt{dce_instr lva lvv}_0\ \texttt{src trg lvv}_1$, then from any source and target states $\texttt{s}_0$ and $\texttt{s}'_0$ related for $\texttt{lvv}_0$, the semantics of `trg` refines the semantics of `src`:

$$[\![\texttt{src}]\!]_{\texttt{instr}}\ \texttt{s}_0 \gtrsim_{\texttt{dce_Call lva},\ \texttt{dce_var_state lvv}_1} [\![\texttt{trg}]\!]_{\texttt{instr}}\ \texttt{s}'_0$$

This lemma is proven by induction on `dce_instr`. Compositionality is achieved thanks to RefBind for DceSeq and to RefMRec for the derivation of the refinement between the whole programs from the pairwise refinements between the functions (using `dce_Call` as invariant). □

```
Inductive liveness := Live | Dead.
Definition live_args := fsym→ list liveness.
Definition live_vars := var → liveness.
```

$$\frac{}{\texttt{dce_list [] u u}} \qquad \frac{\texttt{dce_list lv src trg}}{\texttt{dce_list (Live :: lv) (x :: src) (x :: trg)}}$$

$$\frac{\texttt{dce_list lv src trg}}{\texttt{dce_list (Dead :: lv) (x :: src) trg}}$$

$$\frac{\texttt{List.Forall (}\lambda\texttt{x. lvv x = Live) ys}}{\texttt{dce_instr lva lvv (x = o(ys)) (x = o(ys)) lvv[x} \leftarrow \texttt{Live]}}\ \textsc{KeepOp}$$

$$\frac{\texttt{op_removable o}}{\texttt{dce_instr lva lvv (x = o(ys)) skip lvv[x} \leftarrow \texttt{Dead]}}\ \textsc{RemoveOp}$$

$$\frac{\texttt{dce_list (lva f) ys ys'} \quad \texttt{List.Forall (}\lambda\texttt{x. lvv x = Live) ys'}}{\texttt{dce_instr lva lvv (x = f(ys)) (x = f(ys')) lvv[x} \leftarrow \texttt{Live]}}\ \textsc{DceCall}$$

$$\frac{\texttt{dce_instr lvv}_0\ \texttt{u u' lvv}_1 \quad \texttt{dce_instr lvv}_1\ \texttt{v v' lvv}_2}{\texttt{dce_instr lva lvv}_0\ \texttt{(u;v) (u';v') lvv}_2}\ \textsc{DceSeq}$$

```
Definition dce_prog (entry : fsym) (src trg : prog) : Prop := ...
```

Figure 11. Selected parts of the syntactic specification of a DCE (⋄). A proposition `List.Forall P u` holds if and only if the predicate `P` is satisfied by all elements of the list `u`.

```
⟦Const c⟧op [] := retS c
⟦Add⟧op [m; n] := retS (m + n)
⟦Div⟧op [m; n] := if n =? 0 then ubS else retS (m / n)
⟦Any⟧op [] := b←anyS bool; retS (if b then 1 else 0)
⟦Input⟧op [] := λm. r←triggerLTS (inr EInput); retLTS (r, m)
⟦Output⟧op [n] := λm. _←triggerLTS (inr (EOutput n)); retLTS (0, m)
⟦Read⟧op [p] := λm. retLTS (m p, m)
⟦Write⟧op [p; v] := λm. retLTS (0, m[p←v])
```

Figure 12. Semantics of operators in a domain S with external events and a global memory state (◇). The semantics are either defined directly in S or by unfolding its definition and using the operators in LTS. The indexes specify for each operator the corresponding domain. The semantics of ub and any are described in subsection 5.3.

6.3 Instantiation of the domain

I have thus proven that the DCE we described statically implies the refinement of the semantics defined for an arbitrary domain S. I can now instantiate S to some particular domain expressive enough for the operators I want to consider. I will use a domain based on LTS (section 5) but with some external IO events and a global memory state (◇):

```
Inductive EventFam.State (M : Type) (E : EventFam.t) : EventFam.t :=
  mk [A] (e : E A) (s : M) : EventFam.State M E (A * M).

Definition S (E : EventFam.t) (A : Type) : Type :=
  memory→ LTS ((EventFam.State memory E) + IO) (A * memory).
```

This domain is formally seen as the result of applying successively two monad transformers to LTS:

- A first transformer adds external events hidden from the signature of the semantics by defining `S0 E A := LTS (E + IO) A`.
- Then the state monad transformer adds a memory state as input, in the events and as output: `S E A = memory → S0 (EventFam.State memory E) (A * memory)`.

The operators of sections 2 and 4, plus any and ub, are implemented for S from their definitions for LTS. The refinement is also defined from sem_ref on LTS by quantifying universally over input memories and requiring the equality between output memories and the external events:

$$\mathtt{u} \overset{\mathtt{S}}{\gtrsim}_{\mathtt{Re,R}} \mathtt{v} \;:=\; \forall \mathtt{m},\, \mathtt{u}\;\mathtt{m} \overset{\mathtt{LTS}}{\gtrsim}_{\mathtt{RState\ memory\ Re\ +\ id,\ R\times=}} \mathtt{v}\;\mathtt{m}$$

where RState is defined by:

$$\frac{\mathtt{Re\ d\ e\ R}}{\mathtt{RState\ M\ Re\ (mk\ d\ m)\ (mk\ e\ m)\ (R\times =)}}$$

Figure 12 describes the implementation of some operators. Effectful operators such as Div (partial), Any (non-determinism), Input (external events) or Write (global memory state) can be defined since op_sem is specified as an element of S. All operators except Input, Output and Write are removable. I prove in particular by unfolding the definition of the

refinement on `S` that `Read`, which accesses the memory state, is removable. I thus obtain the correctness of a transformation which removes memory reads using `dce_prog_spec`, although it never mention a memory state. The key ingredients that enabled this proof are that:

- The domain `S` has a notion of refinement `sem_ref` with the signature expected by my theory (which does not mention the memory).
- All properties needed about `S` to justify the correctness of the transformation can be expressed using `sem_ref` as properties of the combinators (that is, the axioms of my theory) or as local properties (here `op_removable`).

This abstraction over the domain enables some factorizations and makes the proofs robust to a change of the domain. But as demonstrated, it also simplifies the proof: if it was done directly with the refinement on `LTS`, we would have needed to take the memory into account and specifies proof invariants on it. On the other hand, doing so could have allowed some reasoning on the memory and maybe the removal of some `Write` operations.

6.4 Comparison with abstraction using events

Zakowski et al. [ZBY+21, §4.3,§5.3] offer another approach to abstract away some effects. Instead of directly defining the semantics in a domain `S` expressive enough, programs are denoted into a simpler domain `S0` by replacing effectful operations with the emission of events of some family $\mathcal{E}$. Those denotations can then be interpreted into `S` by using a monad morphism which replaces the events by their implementations:

$$\texttt{interp}_{\mathcal{E}}\ \texttt{[A]: S0}\ \mathcal{E}\ \texttt{A} \rightarrow \texttt{S EventFam.emp A}$$

this interpretation has the following property:

$$\texttt{u} \overset{\texttt{S0}}{\gtrsim}_{\texttt{id,R}} \texttt{v} \Rightarrow \texttt{interp}_{\mathcal{E}}\ \texttt{u} \overset{\texttt{S}}{\gtrsim}_{\ldots,\texttt{R}} \texttt{interp}_{\mathcal{E}}\ \texttt{v}$$

which can be used to derive a refinement in `S` from a refinement in the simpler domain `S0`. Since a refinement requires the events to be preserved, this approach cannot be used if some effectful operations are affected. In our case, `Read` could not be removed. More generally, one cannot use refinements (such as `op_removable`) proven in `S`. On the other hand, this approach only requires a refinement on a fixed domain `S0`, whereas I ask for a refinement on an arbitrary domain satisfying some relational properties. Reasoning on `S0` can enable the use of techniques specific to this domain, for instance coinduction.

7 Related work

On relational theories Xia et al. [XZH+20] prove a relational theory on ITrees and show that the correctness of some transformations can be derived without further coinductive reasoning. Nearly all axioms I use are properties proven in this theory. Yoon et al. [YZZ22] identify an interface satisfied by ITrees and other monadic domains, and which includes some axioms for relational reasoning. However, some of their axioms rule out asymmetric relations and non-deterministic domains. Moreover, they do not include `mrec` in their interface. The axioms of Figure 2 on `sem_ref`, `ret` and `bind` are equivalents to a subset of the properties needed for `sem_ref` to be a *relator* for the monad `S` [LGL17] modulo the equivalence relation $\approx$.

On equational theories of iteration Iterative monads [AMV04] extends monads with iteration by requiring the existence and unicity of solutions to *guarded* equations. On ITrees and LTS the τ-steps could be used as guards, however in both cases `sem_ref` is instantiated with a *weak* refinement which does not distinguish semantics that differ only by finite sequences of τ-step. Hence $\approx$ is too coarse to be a congruence for a notion of guardedness based on τ-steps. Following the interface defined for the ITrees, I only assume a canonical iteration operator `iter`. The axioms we use about this operator are a subset of those of complete Elgot monads [GMR16]. The other iteration operator, `mrec`, can also be understood with this framework by considering event handlers instead of continuations [XZH⁺20].

On relations on events A refinement heterogeneous over the events, `rutt`, is defined in the ITree library[5]. It is parameterized by a relation on the events and a relation on their answers. There is an injection from the type of pairs of such relations into my `EventFam.Rel`, but the converse is not true. Michelland et al. [MZG24] define a notion of `EventLattice`[6], similar to the relations on events used by `rutt` but specialized for abstract interpretation. Koenig et al. [KS21] specify and compose simulations on open components using *simulation conventions*, a concept similar to `EventFam.Rel`.

On non-determinism There have been several proposals to extend ITrees with non-determinism. Zakowski et al. [ZBY⁺21] consider sets of ITrees as domain. Although the `bind` operator can be defined, one of the directions of its associativity does not hold. Other formalisms [CSL⁺23, CHH⁺23] avoid this problem by defining variants of ITrees which natively support non-determinism thanks to dedicated constructors. Works that prove transformations corrects with explicit simulations often avoid non-determinism in order to use forward simulations ([Ler09, §2.1]). Yet even when the final target language is deterministic, non-determinism is useful to quantify over some choices resolved by later passes. This is harder with deterministic semantics and often involves ad-hoc simulations. For instance in CakeML [KTMK⁺19, §7.1], a "permute oracle" models the effect of a garbage collector before its behavior is fully determined.

References

[ABC⁺21] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter et Jérémy Thibault : An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Transactions on Programming Languages and Systems*, 43(4):1–48, décembre 2021.

[AL91] Martín Abadi et Leslie Lamport : The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. Publisher: Elsevier.

[AMV04] Jiří Adámek, Stefan Milius et Jiří Velebil : From Iterative Algebras to Iterative Theories. *Electronic Notes in Theoretical Computer Science*, 106:3–24, décembre 2004.

[CHH⁺23] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski et Steve Zdancewic : Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages*, 7(POPL):1770–1800, janvier 2023.

[5] `https://github.com/DeepSpec/InteractionTrees/blob/e7fed212/theories/Eq/Rutt.v`
[6] `https://gitlab.inria.fr/sebmiche/itree-ai/blob/icfp24-artifact/theories/Core/Events.v`

[CSL+23] Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher et Derek Dreyer : Stuttering for Free. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):1677–1704, octobre 2023.

[GBB+23] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux et Alexandre Bérard : Formally Verifying Optimizations with Block Simulations. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):59–88, octobre 2023.

[GMR16] Sergey Goncharov, Stefan Milius et Christoph Rauch : Complete Elgot Monads and Coalgebraic Resumptions. *Electronic Notes in Theoretical Computer Science*, 325:147–168, octobre 2016.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish et Scott Owens : CakeML: a verified implementation of ML. *In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego California USA, janvier 2014. ACM.

[KS21] Jérémie Koenig et Zhong Shao : CompCertO: compiling certified open C components. *In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1095–1109, Virtual Canada, juin 2021. ACM.

[KTMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens et Michael Norrish : The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019.

[Ler09] Xavier Leroy : A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363–446, décembre 2009.

[LGL17] Ugo Dal Lago, Francesco Gavazzo et Paul Blain Levy : Effectful applicative bisimilarity: Monads, relators, and Howe's method. *In 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, Reykjavik, Iceland, juin 2017. IEEE.

[McB15] Conor McBride : Turing-Completeness Totally Free. *In Mathematics of Program Construction*, volume 9129, pages 257–275. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science.

[MZG24] Sébastien Michelland, Yannick Zakowski et Laure Gonnord : Abstract Interpreters: A Monadic Approach to Modular Verification. *Proceedings of the ACM on Programming Languages*, 8(ICFP):602–629, août 2024.

[XZH+20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce et Steve Zdancewic : Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, janvier 2020.

[YZZ22] Irene Yoon, Yannick Zakowski et Steve Zdancewic : Formal reasoning about layered monadic interpreters. *Proceedings of the ACM on Programming Languages*, 6(ICFP):254–282, août 2022.

[ZBY+21] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva et Steve Zdancewic : Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.*, 5(ICFP), août 2021. Place: New York, NY, USA Publisher: Association for Computing Machinery.