Reaching for the Star: Tale of a Monad in Coq

Pierre Nigron ✉
Sorbonne Université, CNRS, Inria, LIP6, Paris, France

Pierre-Évariste Dagand ✉
Sorbonne Université, CNRS, Inria, LIP6, Paris, France

Abstract

Monadic programming is an essential component in the toolbox of functional programmers. For the pure and total programmers, who sometimes navigate the waters of certified programming in type theory, it is the only means to concisely implement the imperative traits of certain algorithms. Monads open up a portal to the imperative world, all that from the comfort of the functional world. The trend towards certified programming within type theory begs the question of *reasoning* about such programs. Effectful programs being encoded as pure programs in the host type theory, we can readily manipulate these objects through their encoding. In this article, we pursue the idea, popularized by Maillard [21], that every monad deserves a dedicated program logic and that, consequently, a proof over a monadic program ought to take place within a Floyd-Hoare logic built for the occasion. We illustrate this vision through a case study on the `SimplExpr` module of CompCert [18], using a separation logic tailored to reason about the freshness of a monadic gensym.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases monads, hoare logic, separation logic, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.14

Supplementary Material *Software*: `https://github.com/Artalik/CompCert/tree/ITP`

Funding *Pierre Nigron*: "Ministère des armées – Agence de l'Innovation de Défense" grant
Pierre-Évariste Dagand: Emergence(s) – Ville de Paris grant

1 Introduction

This article dwells on the challenges of verifying imperative algorithms implemented in a proof assistant. As certified programming becomes more commonplace, proof assistants are indeed being used as the ultimate integrated development environment [5, 10]. The question of specifying and proving the correctness of such programs is part of a long tradition, starting from various generalizations of monads [11, 33, 4] accounting for dependent types and YNot [24], an axiomatic extension of type theory featuring imperative traits, as well as the family of Dijsktra monads [3, 21, 22, 32] in F* and their intuitionistic counterparts in Agda [35], including the recent activity around algebraic presentations of effects and their embedding in Coq and Agda [6, 7, 37, 20, 19]. This article reports on an experiment in revisiting a proof of Leroy [18] with the help of Hoare [14] and Reynolds [29], under the direction set by Plotkin and Power [28].

Before reaching for the top on the shoulders of these giants, let us warm up with a classical monadic verification problem due to Hutton and Fulger [15] involving labelled binary trees

```
Inductive Tree (X: Type) :=
| Leaf: X → Tree X
| Node: Tree X → Tree X → Tree X.
```

The challenge consists in implementing a function `label: Tree X → Tree nat` that labels every leaf with a fresh symbol, here a natural number. In order to implement this

12th International Conference on Interactive Theorem Proving (ITP 2021).
Editors: Liron Cohen and Cezary Kaliszyk; Article No. 14; pp. 14:1–14:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

relabeling procedure in Coq, we are naturally led to define the following variant of the state monad [26]:

```
Definition Fresh X := nat → X * nat.


Definition ret (x: X): Fresh X :=
  fun n ⇒ (x, n).
Definition bind (m: Fresh X)(f: X → Fresh Y): Fresh Y :=
  fun n ⇒ let (x, n') := m n in f x n'.
Definition gensym (tt: unit): Fresh nat :=
  fun n ⇒ (n, 1+n).


Notation "'do' x '← ' e1 ';' e2" := (bind e1 (fun x ⇒ e2)).
```

Tree relabeling is then the straightforward imperative program one would have written in any ML-like language:

```
Fixpoint label {X} (t: Tree X): Fresh (Tree nat) :=
  match t with
  | Leaf _ ⇒
    do n ← gensym tt;
    ret (Leaf n)
  | Node l r ⇒
    do l ← label l;
    do r ← label r;
    ret (Node l r)
  end.
```

The function `label` is correct if the structure of the tree is preserved and each leaf stores a unique number. Setting aside the question of preserving the tree structure, Hutton and Fulger [15] offered the following formal specification for the latter property:

```
Lemma label_spec : ∀ t n ft n',
    label t n = (ft, n') → n < n' ∧ flatten ft = interval n (n'-1).
```

where `flatten` accumulates each leaf value during a left-to-right traversal and `interval a b` computes the list of integers in the interval $[a, b]$. Note that this specification is extremely prescriptive as it requires that `label` consecutively numbers the leaves of the tree from the initial state n of the fresh name generator to its final state n' in a left-to-right fashion.

It is easy to deduce the absence of duplicates, captured by the `NoDup` predicate in Coq standard library:

```
Definition relabel (t: Tree X): Tree nat := fst (label t 0).

Lemma relabel_spec : ∀ t ft, relabel t = ft → NoDup (flatten ft).
```

which makes for a reasonable public API to expose, unlike the property established by `label_spec`. The correctness of relabeling rests on our ability to prove `label_spec`. To do so, it is obviously possible to treat `label` as a pure function (since it is one, after all) and therefore directly manipulate the functional encoding of our variant of the state monad. For example, to reason about a sequence of operations, we would use the inversion lemma

```
Remark bind_inversion: ∀ m f y n1 n3,
    (do x ← m; f x) n1 = (y, n3) →
    ∃ v n2, m n1 = (v, n2) ∧ f v n2 = (y, n3).
```

that reifies, through an existential, the intermediate state that occurs between the first and second operation, thus allowing us to reason piece-wise about the overall program.

Here, the proof proceeds by induction over the tree `t`. For instance, in the `Node` case, we are given the hypothesis

```
(do l ← label t1;
 do r ← label t2;
 ret (Node l r)) n = (t', n')
```

which we invert twice using `bind_inversion` so as to reveal the intermediate states `n2`, `n3` and intermediate results `t1'`, `t2'`:

- `label t1 n = (t1', n2)`
- `label t2 n2 = (t2', n3)`
- `Node t1' t2' = t'`
- `n3 = n'`

We can then proceed by induction over the first two hypothesis in order to deduce `flatten t1 = interval n (n2-1)` (with `n < n2`) on the one hand and `flatten t2 = interval n2 (n3-1)` (with `n2 < n3`) on the other hand. Properties of intervals allow us to deduce that `flatten (Node t1 t2) = interval n n'`, which establishes the desired invariant. The resulting proof is thus a back-and-forth between reasoning steps related to the monadic structure of the program (for example, `bind_inversion` above) and reasoning steps related to the invariants preserved by the program (for example, concatenating intervals above).

In order to decouple the monadic structure (whose role is to sequentialize effects) from specific interpretations of this structure (which defines its admissible semantics), one can follow the mantra of the algebraic presentations of effects [28]: start with syntax (by means of signatures) and obtain monads. In Coq, we can easily give the term algebra corresponding to the `Fresh` monad using the folklore free monad construction [19]:

```
Inductive FreeFresh X :=
| ret : X → FreeFresh X
| gensymOp : unit → (nat → FreeFresh X) → FreeFresh X.

Fixpoint bind (m: FreeFresh X)(f: X → FreeFresh Y): FreeFresh Y :=
  match m with
  | ret v ⇒ f v
  | gensymOp _ k ⇒ gensymOp tt (fun n ⇒ bind (k n) f)
  end.

Definition gensym (tt: unit): FreeFresh nat := gensymOp tt (@ret nat).
```

In effect, we are defining a *syntax* for an embedded imperative language (sequenced through the `bind` construct) featuring all Coq values (through the `ret` constructor) as well as a `gensym` operator. To give a semantics to this language, an avid Coq programmer would claim that an interpreter is as good a denotational semantics as anything else:

```
Fixpoint eval (m: FreeFresh X): nat → X * nat :=
  match m with
  | ret v ⇒ fun n ⇒ (v, n)
```

```
  | gensymOp _ k ⇒ fun n ⇒ eval (k n) (1 + n)
  end.
```

Alternatively, a zealous disciple of Dijsktra (who may well be his grand nephew [35]) would perhaps give a semantics based on predicate transformers, using for example a weakest-precondition calculus:

```
Fixpoint wp (m: FreeFresh X)(Q: X → nat → Prop): nat → Prop :=
  match m with
  | ret v ⇒ fun n ⇒ Q v n
  | gensymOp _ k ⇒ fun n ⇒ wp (k n) Q (1+n)
  end.
```

To get them to come to an agreement, we would prove the adequacy of both semantics:

```
Lemma adequacy: ∀ m Q n n' v,
    wp m Q n → eval m n = (v, n') → Q v n'.
```

Whilst we have argued against reasoning directly about the semantics of monadic programs (which amounts to `eval m` here), the adequacy lemma gives us an opportunity to switch to a more predicative reasoning style. In particular, Hoare triples [14], dear to the heart of imperative programmers, can be obtained through a simple notational trick

```
Notation "{{ P }} m {{ Q }}" := (∀ n, P n → wp m Q n)
```

from which we can readily prove the usual rules of Hoare logic [27]

```
Lemma rule_value: ∀ Q v,
    (*-----------------------*)
    {{ Q v }} ret v {{ Q }}.

Lemma rule_composition: ∀ m f P Q R,
    {{ P }} m {{ Q }} →
    (∀ v, {{ Q v }} f v  {{ R }}) →
    (*-------------------------------*)
    {{ P }} do x ← m; f x {{ R }}.

Lemma rule_gensym: ∀ k,
    (*---------------------------------------------------------*)
    {{ fun n ⇒ n = k }} gensym tt {{fun v n' ⇒ v = k ∧ n' = 1+k}}.

Lemma rule_consequence: ∀ P P' Q Q' m,
    {{ P' }} m {{ Q' }} →
    (∀ n, P n → P' n) →
    (∀ x n, Q' x n → Q x n) →
    (*-----------------------*)
    {{ P }} m {{ Q }}.
```

or, put otherwise, we obtain a shallow embedding of Hoare logic within the logic of Coq.

While, syntactically, the code of `label` is unchanged, it is now a mere abstract syntax tree. Accordingly, the correctness lemma is naturally expressed as a Hoare triple:

```
Lemma label_spec: ∀ t k,
    {{ fun n ⇒ n = k }}
```

```
  label t
{{ fun ft n' ⇒ k < n' ∧ flatten ft = interval k (n'-1) }}.
```

This specification remains unsatisfactory: we have still over-specified the behavior of a counter whereas, *in fine*, we are only ever interested in the property `NoDup (flatten t)`. To prove it, we only need the assurance that every call to `gensym tt` produce a number distinct from any previous call (which is indeed verified by an implementation that produces consecutive numbers but this is an implementation detail).

In the remaining of this article, we argue that separation logic [29] is the perfect vehicle for this kind of specification. Our plan is to unleash the power of the wonderful ecosystem created by the MoSel [17] (and, by extension, Iris [16]) –initially introduced to model and reason about fine-grained models of concurrent systems and languages– to bear on the verification of our monadic programs. Our contributions are the following:

- We instantiate the MoSel framework (Section 2) with a custom logic to reason (exclusively) about freshness over monadic programs. The result is a tailor-made program logic embedded within Coq supporting modular reasoning about freshness, MoSel offering a wonderful environment to harness this flexibility;
- We resume our formalization of `relabel` in this framework (Section 3) and highlight the key point of the methodology;
- We offer a larger case study (Section 4) by porting the `SimplExpr` module of CompCert [18] to our framework. This module extensively relies on a monad offering a fresh name generator together with non catchable exceptions. Crucially, we show that separation logic can be used locally while the resulting theorems can be integrated in a larger (pre-existing) development standing solely in `Prop`.

Our Coq development is available online[1]. The symbol [🐓] in the electronic version of the paper will lead the reader to the corresponding source code.

2 Supporting Modular Specifications [🐓]

Separation logic [29] prominently features a *frame* rule that enables modular reasoning about properties supporting a notion of *disjointedness*. This is particularly relevant for freshness: we naturally expect to be able to reason separately about two programs producing fresh identifiers, without interference. We now formalize this intuition by instantiating the MoSel [17] framework with a minimalist separation logic to reason about generated symbols.

The type of assertions `hprop` corresponds to predicates over finite sets[2] of identifiers:

```
Definition hprop := gset ident → Prop.
```

Through this definition, `hprop` inherits the logical apparatus of `Prop` (through pointwise lifting): existential quantification, universal quantification, conjunction, *etc.* This also includes any Coq propositions `P`, called *pure* propositions and written ⌜ P ⌝

```
Definition hpure (P : Prop) : hprop := fun _ ⇒ P.
```

The defining feature of a separation logic is the presence of a *separating conjunction*

[1] https://github.com/Artalik/CompCert/tree/ITP
[2] Implemented by the `gset` type in the `Coq-std++` library [23]

```
Definition hstar (P1 P2 : hprop) : hprop := fun idents ⇒
  ∃ ids1 ids2, P1 ids1 ∧ P2 ids2 ∧ ids1 ## ids2 ∧ idents = ids1 ∪ ids2.
```

that splits a given set of identifiers `idents` in a two sets `ids1` and `ids2` that are distinct (`ids1 ## ids2`), form a partition of `idents` (`idents ≡ ids1 ∪ ids2`), each satisfying its respective predicate. Unlike standard conjunction (where both propositions must hold for the *whole* set of identifiers), the separating conjunction translates the independence of both predicates by extracting two independent subsets of identifiers. Dually, the *separating implication*, written `P1 -∗ P2`, amounts to the predicate

```
fun ids1 ⇒ ∀ ids2, ids1 ## ids2 ∧ P1 ids2 → P2 (ids1 ∪ ids2).
```

and consists, intuitively, in offering `P2` provided that one can extend the existing set of identifiers so as to satisfy `P1`.

The assertion `emp = fun idents ⇒ idents = ∅` states that no identifier has been generated. We can also assert the freshness of an identifier `ident` (written `& ident`) by stating that it is the sole identifier in the supporting set

```
Definition hsingle ident : hprop := fun idents ⇒ idents = {[ ident ]}.
```

and, more generally, the operator `&& h` states that the set of identifiers amounts precisely to the identifiers in `h`. The interplay between the separating connectives and this characterization of freshness allows us to prove the absence of duplicates, such as the following instrumental lemma[3]:

```
Lemma singleton_neq : ∀ l l’, ⊢ & l -∗ & l’ -∗ ⌜l ≠ l’⌝.
```

From such an algebra of logical connectives, we instantiate the MoSel [17] framework. As a result, we obtain a full-featured interactive environment for reasoning about and manipulating statements in the corresponding separation logic. MoSel introduces the type `iProp` of (suitably-encoded) separation logic assertions, which subsumes `hprop` and its connectives. The relationship between the separation logic and `Prop` is preserved through a (somewhat more noisy) characterization

```
Lemma equivalence (P: iProp) idents: P () idents ↔ (⊢ && idents -∗ P).
```

3 Monadic Proof in Separation Logic [🐓]

Equipped with a separation logic, we can redefine our weakest precondition calculus to take advantage of the added structure

```
Fixpoint wp (m: FreeFresh X)(Q: X → iProp): iProp :=
  match m with
  | ret v ⇒ Q v
  | gensymOp _ k ⇒ ∀ (v: ident), & v -∗ wp (k v) Q
  end.
```

from which we naturally derive Hoare triples and their associated logic [9] as a shallow embedding

[3] The infix operator ⊢ embeds assertions expressed in the internal separation logic into the ambiant logic of Coq `Prop`ositions.

```
Notation "{{ P }} m {{ v ; Q }}" := (P  -∗ wp m (fun v ⇒ Q))
Lemma rule_gensym : ⊢ {{ emp }} gensym tt {{ ident; & ident }}.
Lemma rule_consequence: ∀ P P' Q Q' m,

    (⊢{{ P' }} m {{ v; Q' v }}) →
    (P ⊢ P') →
    (∀ v, Q' v ⊢ Q v) →
  (∗-----------------------∗)
   ⊢{{ P }} m {{ v; Q v }}.
Lemma frame: ∀ P Q P' m,

    (⊢{{ P }} m {{ v; Q v }}) →
    (∗----------------------------∗)
   ⊢{{ P ∗ P' }} m {{ v; Q v ∗ P' }}.
```

while the statement of the earlier lemmas `rule_value` and `rule_composition` remains essentially unchanged (but their signification did change!).

We are now able to specify `label` by actively exploiting the separating conjunction[4]:

```
Lemma label_spec_aux : ∀ t,
     ⊢ {{ emp }}
         label t
       {{ ft; ([∗ list] x ∈ (flatten ft), & x) ∗ ⌜sameShape t ft⌝ }}.
```

Through this move to separation logic, we have discharged the handling of freshness down to the logic, which conveniently provides us with the frame rule (`rule_frame`) to abstract over disjoint sets of identifiers. The proof of `label_spec_aux` is thus significantly simpler and consists only in *local* invariants. This is in stark contrast with our earlier proof in Section 1, where we had to maintain a global invariant across the whole execution of the program.

Thanks to MoSel, the proof script now sums up to the following instructions, which are almost intelligible. The MoSel framework provides the underlined tactics, which we extended with custom tactics (underlined with dashes) specifically manipulating the Hoare triples:

```
induction t.
 - iBind.
   + eapply rule_gensym.
   + iRet . simpl; auto.
 - simpl label.  iBind.
   + eapply IHt1.
   + iBind.   Frame.
     * eapply IHt2.
     * iRet .
       iIntros "[[HA %] [HB %]]".
       iSplitL; auto. simpl.
       iApply big_sepL_app.
       iFrame.
```

```
subgoal 1 :
{{ emp }} gensym ()  {{ v; ?Q0 v}}
subgoal 2 :
{{ ?Q0 v }} ret Leaf v
{{ v'; ([∗ list] x0 ∈ flatten v', & x0) ∗ ⌜sameShape (Leaf x) v'⌝ }}
```

```
subgoal 1 :
{{ ([∗ list] x ∈ flatten v, & x) ∗ ⌜sameShape t1 v⌝ }} label t2
{{ v0; ?Q1 v0}}
subgoal 2 :
{{ ?Q1 v0 }} ret Node v v0
{{ v'; ([∗ list] x ∈ flatten v', & x) ∗ ⌜sameShape (Node t1 t2) v'⌝ }}
```

```
subgoal 1 :
{{ emp }} label t2 {{ v0; ?Q2 v0}}
subgoal 2 :
{{ ?Q2 v0 ∗ ([∗ list] x ∈ flatten v, & x) ∗ ⌜sameShape t1 v⌝ }}
ret Node v v0
{{ v'; ([∗ list] x ∈ flatten v', & x) ∗ ⌜sameShape (Node t1 t2) v'⌝ }}
```

[4] The notation `([* list] x in l, P x)` asserts that every element `x` of the list `l` satisfies the predicate `P`. In the present case, we state that all the elements in the flattened tree are fresh.

In the leaf case, the proof essentially boils down to applying `rule_gensym`. The power of the approach strikes in the node case, where we gain access to the recursive cases through the composition rule, at which point the proof is over: the frame rule allows us to automatically combine the results of both sub-calls.

However, at this stage, we only have a proof in `iProp` while our users are expecting a pure Coq proposition, living in `Prop`. We can first narrow the gap between the two worlds by showing that the non-pure post-condition of `label_spec_aux` amounts to a pure one

```
∀ idents, ⊢ ([∗ list] i ∈ idents, & (i: ident)) -∗ ⌜NoDup idents⌝.
```

and, consequently, we obtain a specification with a pure post-condition

```
Lemma label_spec: ∀ t,
    ⊢ {{ emp }} label t {{ ft; ⌜NoDup (flatten ft) ∧ sameShape t ft⌝ }}.
```

The gap is finally bridged through an adequacy lemma, relating the execution of monadic programs with the generator set to 0

```
Definition run (m: FreeFresh X): X := fst (eval m 0).
```

with pure post-conditions obtained in the separation logic

```
Lemma adequacy : ∀ {X} {m: FreeFresh X} {Q},
    (⊢ {{ emp }} m {{ v; ⌜Q v⌝ }}) →
    Q (run m).
```

As a corollary, we obtain a publicly-usable relabeling function together with a specification expressed at a suitable level of detail:

```
Definition relabel (t: Tree X): Tree nat := run (label t).
```

```
Lemma relabel_spec : ∀ t ft,
    relabel t = ft → NoDup (flatten ft) ∧ sameShape t ft.
```

4 Case study: `SimplExpr`

To evaluate our approach, we tackle a pre-existing certified program, namely the `SimplExpr` module of the CompCert certified compiler. This module implements a simplification phase over C expressions, pulling side-effects out of expressions and fixing an evaluation order. In the following, we offer a side-by-side comparison of the original specification with ours, exploiting separation logic (Section 2) to reason about freshness. We first materialize the underlying monad in Section 4.1 together with its dynamic and predicate transformer semantics. We then delve into the benefits of having a rich logic of assertions (Section 4.2) to carry the proofs. We finally demonstrate how these properties can then be translated to and interact with pure Coq propositions (Section 4.3), so as to be usable in the correctness proof of the whole compiler.

4.1 The monad

As for our introductory example, we crucially rely on a syntactic description of the monad `mon` used by the `SimplExpr` module. This monad, which has received some attention in the literature [34], exposes two operations: an `error e` operator, to report a run-time error `e`;

a `gensym ty` operator, to generate a fresh symbol associated with a type `ty`, and a `trail` operator, to get the association list of identifiers to types constructed thus far.

Following the usual free monad construction, we reify this interface through a datatype:

```
Inductive mon (X : Type) : Type :=
| ret : X → mon X
| errorOp : Errors.errmsg → mon X
| gensymOp : type → (ident → mon X) → mon X
| trailOp : unit → (list (ident * type) → mon X) → mon X.

Definition error {X} (e : Errors.errmsg) : mon X := errorOp e.
Definition gensym (t : type) : mon ident := gensymOp t ret.
Definition trail (_ : unit): mon (list (ident * type)) := trailOp tt ret.
```

The definition of the monadic bind follows naturally. As before, we will use the user-friendly notation `do _ ← _ ; _` in code.

Note that an `error` does not require a continuation: at run-time, it corresponds to an uncatchable exception. It is used by the compiler to abort when some input program falls outside the semantic domain of C (delineated by the mechanized semantics given by CompCert).

The dynamic semantics of `mon` is slightly richer than the one of `FreeFresh` (Section 1). First, we must handle the addition of an uncatchable error during execution. We piggy-back on CompCert's implementation of the error monad

```
Inductive res (A: Type) : Type :=
| OK: A → res A
| Error: errmsg → res A.
```

and, essentially, inline the usual error monad transformer over the state monad necessary to maintain the internal state of the `gensym` operator. However, unlike earlier, `gensym` now associates fresh identifiers with their provided type. This is reflected in the semantics, which maintains an association list of ident and types together with the next fresh ident:

```
Record generator : Type := mkgenerator { gen_next : ident;
                                         gen_trail: list (ident * type) }.
```

The dynamic semantics amounts to the usual interpretation of errors in `res` and stateful operations in `generator → M (generator * X)`:

```
Fixpoint eval {X} (m : mon X) : generator → res (generator * X) :=
  match m with
  | ret v ⇒ fun s ⇒ OK (s, v)
  | errorOp e ⇒ fun s ⇒ Error e
  | gensymOp ty f ⇒
    fun s ⇒
      let h := gen_trail s in
      let n := gen_next s in
      eval (f n) (mkgenerator (n+1) ((n,ty) :: h))
  | trailOp _ f ⇒
    fun s ⇒
      let h := gen_trail s in
```

```
      eval (f h) s
  end.
```

The compiler pass is ran with an `initial_generator` that is provided from the OCaml driver, remaining opaque to Coq until after extraction:

```
Definition run {X} (m: mon X): res X :=
  match eval m (initial_generator tt) with
  | OK (_, v) ⇒ OK v
  | Error e ⇒ Error e
  end.
```

The predicate transformer semantics is given by a straightforward weakest-precondition calculus:

```
Fixpoint wp {X} (e1 : mon X) (Q : X → iProp) : iProp :=
  match e1 with
  | ret v ⇒ Q v
  | errorOp e ⇒ True
  | gensymOp _ f ⇒ ∀ l, & l -∗ wp (f l) Q
  | trailOp _ f ⇒ ∀ l, wp (f l) Q
  end.
```

where the semantics of `gensym` follows exactly our earlier definition. The semantics of `error` does not require any precondition (but, as we shall see in the adequacy lemma, this also means that our post-conditions are only true *if* the compiler did not raise an error). The specification of `trail` is purposefully non-committal: CompCert does not make any assumption about the output of `trail` (in a rather elegant twist, the fact that the identifiers produced by `trail` are all distinct is a decidable property that is checked at run-time in a later compilation pass: `trail` is indeed free to return any list of identifiers but CompCert will simply refuse to compile a piece of code triggering an invalid output.)

As in Section 1, we derive Floyd-Hoare triples `{{ P }} m {{ v; Q }}` from our weakest-precondition calculus, together with the usual structural rules. The monad-specific operators are specified as follows:

```
Lemma rule_gensym ty : ⊢{{ emp }} gensym ty {{ ident; & ident }}.

Lemma rule_error Q e: ⊢{{ True }} error e {{ v; Q v }}.

Lemma rule_trail : ⊢{{ emp }} trail tt {{ _; emp  }}.
```

In particular, the operator `error` amounts to a "get out of proof free card", allowing us to discharge any post-condition by refusing to do any work. We relate the dynamic and predicate transformer semantics through an adequacy lemma

```
Lemma adequacy: ∀ m Q v,
   (⊢ {{ emp }} m {{ v; ⌜ Q v ⌝}}) →
   run m = OK v →  Q v.
```

that only proves the post-condition when the evaluation succeeds in producing a value.

4.2 Proofs and Programs [🐓]

The expression simplification pass is part of the CompCert front-end. It consists of 3 files: `cfrontend/SimplExpr.v` (which contains the monadic programs), `cfrontend/`

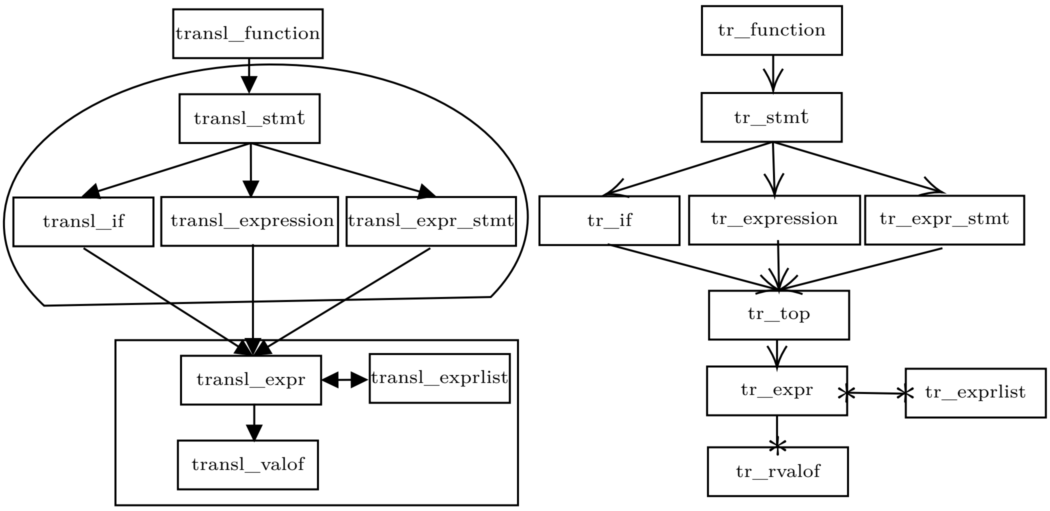

Figure 1 `SimplExpr` call graph (left) and the corresponding specifications (right)

`SimplExprspec.v` (which contains a Prolog-like specification of the monadic programs through inductive relations, as well as the proof relating the monadic programs to their specification) and `cfronted/SimplExprproof.v` (which contains the proof of correctness of the compilation pass, exploiting the relational specifications). Syntactically, `cfrontend/SimplExpr.v` is left unchanged when we swap in our monad: we were careful to implement the same interface as the previous one. However the semantics is very different: whereas the previous monad was building an actual computation, ours is just building an abstract syntax tree. We therefore need to add suitable call to `run` to turn this syntax into an actual computation.

We give an overview of the `SimplExpr` module through its call graph (Figure 1). The raison d'etre of this module is to define `transl_function: Csyntax.function → res function` that performs the simplification over functions. This is the (only) entry-point into the error monad `res`. It hosts the call `run`. `transl_function` recursively depends on a host of helpers operating in the `error` and `trail` fragment of the monad, grouped in the circular frame (Figure 1). Crucially, none of the functions invoke a fresh symbol generator themselves. A third group of functions, all dispatched from `transl_expr` and collected in the rectangular frame (Figure 1), consists of those functions that actually generate fresh symbols and must therefore belong to the full-fledged monad `mon`.

In the following, we present several programs extracted or modified from CompCert, together with their specifications. In those, aspects related to the freshness of names is a means toward an overall correctness result. Consequently, programs and specifications involve a backbone of operations and properties dealing with freshness, fleshed out with further transformations and properties implementing the desired compilation pass. In order to see the forest (of freshness) for the trees, we adapt a typographical legerdemain: we typeset in a tiny font size the parts of the program and proof that do not involve freshness. As part of our work, we were led to replace definitions from the original CompCert with new ones: when recalling the original, we display it on a gray background to set it apart.

Let us begin our exploration of the `SimplExpr` module through `transl_expr`, which involves both fresh name generation and errors

```
Fixpoint transl_expr (dst: destination) (a: Csyntax.expr) : mon (list statement * expr)
```

Its argument dst may wrap, in the For_set case, an identifier within a value of type set_destination

```
Inductive set_destination : Type :=
  | SDbase (tycast ty: type) (tmp: ident)
  | SDcons (tycast ty: type) (tmp: ident) (sd: set_destination).

Inductive destination : Type :=
  | For_val
  | For_effects
  | For_set (sd: set_destination).
```

The type destination specifies how to pass along the result of a given expression, *i.e.* whether the contribution of an expression lies in its returned value, or solely in its side effects, or in a temporary variable in which its denotation has been saved.

For correctness of this optimization pass, it is crucial that this identifier is fresh with respect to any identifier that transl_expr may produce. The function transl_expr itself is defined by pattern-matching over the source AST, we focus here on the assignment case:

```
| Csyntax.Eassign l1 r2 ty ⇒
  do (sl1, a1) ← transl_expr For_val l1;
  do (sl2, a2) ← transl_expr For_val r2;
  let ty1 := Csyntax.typeof l1 in
  let ty2 := Csyntax.typeof r2 in
  match dst with
  | For_val | For_set _ ⇒
    do t ← gensym ty1;
    ret (finish dst
          (sl1 ++ sl2 ++ Sset t (Ecast a2 ty1) ::
             make_assign a1 (Etempvar t ty1) :: nil)
          (Etempvar t ty1))
  | For_effects ⇒
    ret (sl1 ++ sl2 ++ make_assign a1 a2 :: nil,
      dummy_expr)
  end
```

It performs two recursive calls with destinations that do not involve fresh identifiers (For_val). However, when its own destination is a value (For_val) or an assignment (For_set), it also performs a call to gensym. The specification needs to reflect the fact that the identifiers generated by the recursive calls are distinct between each other and distinct from the identifier potentially generated in the assignment case. In CompCert, this is achieved by explicitly threading the lists (in this case, tmp, tmp1 and tmp2) of identifiers generated and asserting their disjointness:

```
Inductive tr_expr: temp_env → destination → Csyntax.expr → list statement → expr →
                   list ident → Prop :=

  | tr_assign_val: ∀ le dst e1 e2 ty sl1 a1 tmp1 sl2 a2 tmp2 t tmp ty1 ty2,
      tr_expr le For_val e1 sl1 a1 tmp1 →
      tr_expr le For_val e2 sl2 a2 tmp2 →
```

```
        incl tmp1 tmp → incl tmp2 tmp →
        list_disjoint tmp1 tmp2 →
        In t tmp → ~In t tmp1 → ~In t tmp2 →
        ty1 = Csyntax.typeof e1 →
        ty2 = Csyntax.typeof e2 →
        tr_expr le dst (Csyntax.Eassign e1 e2 ty)
            (sl1 ++ sl2 ++
             Sset t (Ecast a2 ty1) ::
             make_assign a1 (Etempvar t ty1) ::
             final dst (Etempvar t ty1))
            (Etempvar t ty1) tmp
```

In order to express the precondition on `transl_expr`, stating that any potential identifier in `dst` is fresh, CompCert introduces the following predicate

```
Definition sd_temp (sd: set_destination) :=
  match sd with SDbase _ _ tmp ⇒ tmp | SDcons _ _ tmp _ ⇒ tmp end.

Definition dest_below (dst: destination) (g: generator) : Prop :=
  match dst with
  | For_set sd ⇒ Plt (sd_temp sd) g.(gen_next)
  | _ ⇒ True
  end.
```

that, in a very operational manner, asserts that the identifiers stored in `dst` occurred earlier in the execution of the fresh name generator and are therefore distinct from any future identifier (since they are produced as consecutive numbers).

Having access to a notion of freshness in our language of assertions, we can prevent these operational details from leaking out and simply assert that such an identifier must be fresh:

```
Definition dest_below (dst: destination) : iProp :=
  match dst with
  | For_set sd ⇒ & (sd_temp sd)
  | _ ⇒ emp
  end.
```

The implementation of `transl_expr` is then abstracted away thanks to the relational specification given by `tr_expr` as follows

```
Lemma transl_meets_spec:
   (∀ r dst g sl a g' I,
    transl_expr dst r g = Res (sl, a) g' I →
    dest_below dst g →
    ∃ tmps, (∀ le, tr_expr le dst r sl a (add_dest dst tmps)) ∧
            contained tmps g g')
```

where `g` and `g'` represent the state of the fresh name generator at the beginning and, respectively, the end of the transformation. These are necessary to assert that any ident in `dst` is indeed fresh (through `dest_below`) and that the temporaries produced by `transl_expr` will not conflict with any earlier or later use of the generator (through `contained tmps g g'`, which guarantees that all the identifiers in `tmps` were produced between `g` and `g'`).

In our setting, the freshness of the identifiers produced in the subcalls and of the locally generated identifier is captured with separating conjunctions:

```
Fixpoint tr_expr (le : temp_env) (dst : destination) (e : Csyntax.expr)
        (sl : list statement ) (a : expr)  : iProp :=

  | Csyntax.Eassign e1 e2 ty ⇒
    match dst with
    | For_val | For_set _ ⇒
      ∃ sl2 a2 sl3 a3 t,
       tr_expr le For_val e1 sl2 a2 ∗
       tr_expr le For_val e2 sl3 a3 ∗
       & t ∗
       dest_below dst ∗
       ⌜ sl = sl2 ++ sl3 ++ Sset t (Ecast a3 (Csyntax.typeof e1)) ::
            make_assign a2 (Etempvar t (Csyntax.typeof e1)) ::
            final dst (Etempvar t (Csyntax.typeof e1)) ∧
  a = Etempvar t (Csyntax.typeof e1)⌝
```

Similarly, the relationship between `transl_expr` and `tr_expr` is now straightforward, the constraint that `dst` is fresh with respect to the identifiers produced by `transl_expr` being naturally expressed through a separating implication

```
Lemma transl_meets_spec :
  (∀ r dst,
     ⊢ {{ emp }} transl_expr dst r
       {{ res; dest_below dst -∗ ∀ le, tr_expr le dst r res.1 res.2 }})
```

Through this process, we have entirely removed the painstaking need to track the operational state of the name generators and maintain global invariants about the relative freshness of program fragments. Doing so, we have elevated our specification and successfully decoupled it for the operational aspects of generating fresh identifiers. As an added bonus, we can now rely on MoSel to prove that our implementation meets its specification. In practice, we observe that the length of the proof scripts is divided by two when moving to MoSel but we shall resist from the temptation of drawing any conclusion from such an unreliable metric.

4.3 Leaving `iProp` [🐓]

Reasoning about freshness occurs only in the group of functions below `transl_expr` in the call graph. For the functions (and their respective specifications) above `transl_expr`, the set of fresh identifiers ranged over by the specification is always existentially quantified. Since, by construction, `iProp` is isomorphic to `gset ident → Prop` (Section 2), we have integrated this discipline in a wrapper-specification

```
Inductive tr_top: destination → Csyntax.expr → list statement → expr →  Prop :=

| tr_top_base: ∀ dst r sl a tmp,
    tr_expr le dst r sl a () tmp →
    tr_top dst r sl a.
```

As a consequence, functions above `transl_expr` do not need to propagate freshness invariants. As a result, `Prop` is a sufficient vehicle to write their specifications. However, to show that these functions satisfy their specifications, we took on ourselves to port the proofs to MoSel as well. For example, the function `transl_stmt`, which translates statements, is specified as follow in our setting

```
Lemma transl_stmt_meets_spec : ∀ s,
  ⊢ {{ emp }} transl_stmt s {{ res; ⌜ tr_stmt s res ⌝}}
```

which is merely a iota away from the original

```
Lemma transl_stmt_meets_spec:
  ∀ s g ts g' I, transl_stmt s g = Res ts g' I → tr_stmt s ts
```

While a purely cosmetic change, this has allowed us to streamline the proofs, which were designed around inversion lemmas over the monadic structure (themselves wrapped in tactics). Note that this effort was not strictly necessary: we could have kept the pre-existing definitions and their proofs.

To restore the overall compiler correctness proof [🐓], we must re-establish a simulation lemma relating source and target programs. This work is carried solely over the specifications of the various functions (right-hand side of Figure 1). Above `tr_top` (included), the specifications lives in `Prop` so the proofs remain unchanged. For `tr_expr`, where the specification lives in `iProp`, we resort to reasoning in separation logic: we have therefore updated the original predicates so as to fully exploit the separating connectives to handle freshness. We carry this part of the simulation proof in MoSel. To bridge the gap between `iProp` and `Prop`, which occurs when we go through `tr_top`, we resort to lemmas such as `singleton_neq` (Section 2) that translates freshness assertions into propositional facts.

5 Related Work

Early on, dependent type theory was used to develop various models of Hoare logic [25, 30], including several ones based on separation logic [24, 8, 16]. However, these formalisms were introduced to reason about *models* of imperative or concurrent programs: type theory was not yet recognized as a vehicle for writing effectful programs. CompCert was instrumental in showing that non-trivial effectful programs could be written within a proof assistant. This inspired the work of Swiertra [34], aiming at rationalizing and generalizing the indexed state monad construction introduced by Leroy specifically in `SimplExpr`.

The Dijkstra Monad [13, 31, 3, 2, 32] research program, spearheaded by Swamy and collaborators, has demonstrated that effectful programming has its place in the context of certified programming in F⋆. On their journey, the designer of F⋆ have shown the benefits of a modular approach to effects (polymonads), each equipped with a suitable program logic (Dijkstra monad) which –in some instances– could be automatically derived from the underlying monad (using an interpretation in the continuation monad). However, this line of work actively exploits the refinement-based approach to typing of F⋆ (relying extensively on an SMT solver to decide the conversion of indices). As-is, this would be ill-fitted for a proof assistant based on dependent type theory, where conversion is not as rich and relying on functional values at the type level would make for a painful experience. Our approach is rooted in the pragmatics of indexed programming in dependent type theory and of Coq in particular. In that respect, MoSel offers the ultimate development environment for reasoning –in a natural manner– about effectful programs in Coq.

Before us, this approach has been pursued in the context of the FreeSpec project [19] in Coq. While its scope was limited to modeling and reasoning about (hardware) interfaces, FreeSpec has shown the benefits of a syntactic treatment of monads (through the free monad construction) and how to construct domain-specific logics for those through pre/post pairs. The key contribution of FreeSpec is a generic treatment of effects, which we could easily borrow to factor out our monadic constructions.

In Agda, Swierstra and Baanen [35] have shown how the FreeSpec approach (based on free monads) and the Dijsktra Monads (deriving program logics from monads) could be fruitfully combined. This results in a library of predicate transformers, operating over the syntactic model of the monad. We followed this approach to the letter, specializing our definition to the monads at hand for pedagogical purposes. Being in Coq, we also benefit from the impredicativeness of `Prop` and, by extension, `iProp`, which saves us from tiptoeing around universe stratification when defining the predicate transformer semantics.

While many of the work above is focused on emulating some form of Hoare logics in type theory, there is also a parallel and rich line of work betting on the power of equational reasoning for effectful programs. Gibbons and Hinze [12] were instrumental in illustrating –on paper– how to use algebraic presentations of monads to prove the correctness of programs implemented in those. In particular, they revisited the relabel program from Hutton and Fulger and gave a purely equational proof of correctness. Affeldt *et al.* [1] realized this vision in the Coq theorem prover, extensively relying on SSReflect [36] to enable a compositional treatment of monads and to effectively reason about monadic programs by rewriting.

Interaction Trees [37] are a middle ground between the purely equational treatment of Affeldt *et al.* and the syntactic treatment of FreeSpec. Much like FreeSpec, interaction trees are constructed from a signature of possible operations. However, the authors dispense with the free monad construction altogether and directly manipulate the free completely iterative monad, *i.e.* infinite unfoldings of the signature's control-flow graph. Program equivalence is thus proved by establishing a bisimilarity between two unfoldings: in practice, this is achieved through equational reasoning; substituting equivalent program fragments for each others. The treatment of diverging computations is worthwile and would deserve further attention in our setting.

6 Conclusion

This paper reports on an experiment: use one of the most advanced piece of technology for reasoning about imperative features –separation logic, embodied by the MoSel framework– to reason about certified monadic programs in Coq. To exercise this approach, we ported the `SimplExpr` module of CompCert to use a separation logic for reasoning about fresh names. Our version of `SimplExpr` is feature-complete and integrated in the rest of compiler pipeline. The definition of the monad and its separation logic introduce an additional 750 lines of code [🐓] (ignoring the 30 000 lines of code of Iris/MoSel). Conversely, the specifications and their proofs go from 1100 lines of code originally down to 650 lines of code [🐓]. The correctness proof stands at around a thousand lines of code [🐓].

We should be careful when interpreting these numbers, as code size is but a poor metric to judge the quality of a development. It is however clear that, while certainly encouraging, this experiment points towards developing an integrated library of monads and their operational semantics (à la FreeSpec [19] and interaction trees [37]) as well as their predicate transformer semantics (à la Dijkstra monad [3]). This effort should also be aimed at providing a library of ready-made separation logics for reasoning about common effects, which would allow us to amortize some of those 750 additional lines of code.

As far as proof engineering goes, it would be interesting to study how our proofs fare compared to the original ones when the underlying code evolves. We believe that the abstract reasoning style enabled by separation logic provides more opportunities for automation, which should smooth out the proof update process. Further experiment is required to confirm or refute this hypothesis.

References

1 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 226–254. Springer, 2019. doi:10.1007/978-3-030-33636-3_9.

2 Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL):65:1–65:30, 2018. doi:10.1145/3158153.

3 Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017. doi:10.1145/3093333.3009878.

4 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.

5 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.

6 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144, 2013. doi:10.1145/2500365.2500581.

7 Edwin Brady. Resource-dependent algebraic effects. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, pages 18–33, 2014. doi:10.1007/978-3-319-14675-1_2.

8 Arthur Charguéraud. Program verification through characteristic formulae. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 321–332. ACM, 2010. doi:10.1145/1863543.1863590.

9 Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP):116:1–116:34, 2020. doi:10.1145/3408998.

10 Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. The end of history? using a proof assistant to replace language design with library design. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 3:1–3:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SNAPL.2017.3.

11 Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4):709–745, 2003. doi:10.1017/S095679680200446X.

12 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 2–14, 2011. doi:10.1145/2034773.2034777.

13 Michael Hicks, Gavin M. Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. In Paul Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 79–99, 2014. doi:10.4204/EPTCS.153.7.

14 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

15 Graham Hutton and Diana Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *Proceedings of the Symposium on Trends in Functional Programming*, Nijmegen, The Netherlands, May 2008.

16 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

17 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018. doi:10.1145/3236772.

18 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

19 Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 32–46. ACM, 2020. doi:10.1145/3372885.3373812.

20 Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 338–354, 2018. doi:10.1007/978-3-319-95582-7_20.

21 Kenji Maillard. *Principles of Program Verification for Arbitrary Monadic Effects. (Principes de la Vérification de Programmes à Effets Monadiques Arbitraires)*. PhD thesis, École Normale Supérieure, Paris, France, 2019.

22 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29, 2019. doi:10.1145/3341708.

23 Iris development team. coq-std++. URL: https://plv.mpi-sws.org/coqdoc/stdpp.

24 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008. doi:10.1145/1411204.1411237.

25 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi:10.1017/S0956796808006953.

26 Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.

27 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018.

28 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. doi:10.1007/3-540-45315-6_1.

29 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

30 Christoph Sprenger and David A. Basin. A monad-based modeling and verification toolbox with application to security protocols. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern,* *Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2007. doi:10.1007/978-3-540-74591-4_23.

31 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

32 Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP):121:1–121:30, 2020. doi:10.1145/3409003.

33 Wouter Swierstra. A hoare logic for the state monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009. doi:10.1007/978-3-642-03359-9_30.

34 Wouter Swierstra. The hoare state monad (proof pearl), 2009.

35 Wouter Swierstra and Tim Baanen. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP):103:1–103:26, 2019. doi:10.1145/3341707.

36 Iain Whiteside, David Aspinall, and Gudmund Grov. An essence of ssreflect. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2012. doi:10.1007/978-3-642-31374-5_13.

37 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.