Deferring the Details and Deriving Programs

Liam O'Connor
UNSW Australia
Sydney, NSW, Australia
liamoc@cse.unsw.edu.au

Abstract

A commonly-used technique in dependently-typed programming is to encode invariants about a data structure into its type, thus ensuring that the data structure is correct by construction. Unfortunately, this often necessitates the embedding of explicit proof terms within the data structure, which are not part of the structure conceptually, but merely supplied to ensure that the data invariants are maintained. As the complexity of the specifications in the types increases, these additional terms tend to clutter definitions, reducing readability. We introduce a technique where these proof terms can be supplied later, by constructing the data structure within a *proof delay* applicative functor. We apply this technique to Trip, our new language for Hoare-logic verification of imperative programs embedded in Agda, where our applicative functor is used as the basis for a verification condition generator, turning the typed holes of Agda into a method for stepwise derivation of a program from its specification in the form of a Hoare triple.

CCS Concepts • **Theory of computation → Logic and verification**; **Hoare logic**; **Type structures**; **Program reasoning**; *Type theory*; • **Software and its engineering → Software verification**; *Imperative languages*;

Keywords hoare logic, agda, verification conditions, applicative functor, types, imperative programming

ACM Reference Format:
Liam O'Connor. 2019. Deferring the Details and Deriving Programs. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3331554.3342605

TyDe '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6815-5/19/08…$15.00
https://doi.org/10.1145/3331554.3342605

1 Introduction

In traditional proof assistants such as those in the LCF tradition, the types used to model data structures are usually very simple inductive types. Data invariants are described as separate predicates, and operations are shown to preserve these invariants in separately-proven lemmas.

When we set about programming with dependent types, however, we have nearly unlimited specification power in the types themselves. Rather than specify a simple inductive type and prove that data invariants are maintained by every operation, we can bake the data invariants directly into the type, thus ensuring that the data invariants are maintained by construction. For example, this ordered list type from Lindley and McBride [27] is parameterised by its lower and upper bounds, requiring $n + 1$ proofs of element ordering for a list of length n:

```
data OList (m n : ℕ) : Set where
  Nil : (m ≤ n) → OList m n
  Cons : (x : ℕ) → (m ≤ x) → OList x n → OList m n
```

Another elegant example is the binary search tree data type of McBride [31], which carries ordering evidence in its leaves:

```
data BST (m n : ℕ) : Set where
  Leaf : (m ≤ n) → BST m n
  Branch : (x : ℕ) → BST m x → BST x n → BST m n
```

While these definitions seem quite appealing, attempting to construct even a simple two-element BST value leads to some rather unsightly results:

```
tree : BST 2 3
tree = Branch 3
         (Branch 2
           (Leaf (s≤s (s≤s z≤n)))
           (Leaf (s≤s (s≤s z≤n))))
         (Leaf (s≤s (s≤s (s≤s z≤n))))
```

The proofs embedded in the tree are not interesting and trivial to automate, but nonetheless clutter the definition and make the overall structure harder to discern. This problem only grows worse as our definitions become more complex. In Section 3, we introduce an imperative language embedded within Agda called Trip, where terms are typed by their correctness specifications. The proofs we must embed inside these terms grow to the hundreds of lines of proof for simple programs of approximately ten lines of code. If these

proofs were to be nested directly within the code, the overall program would become nearly unreadable.

We resolve this readability problem by introducing a *proof delay* applicative functor. This allows us to first sketch the big picture, and only fill in details afterwards. The applicative functor collects any outstanding proof obligations as we sketch, and requires us to provide proofs of these goals before ultimately producing the actual data structure.

When applied to Trip programs, this framework becomes a *verification condition generator*, allowing programs to be written without any proofs, all the while gathering the implications that must be discharged in order to show correctness.

In Section 2, we introduce the proof delay applicative functor in Agda, and provide some small examples of its use. In Section 3, we introduce the core of the Trip language, give it semantics in terms of state-relations, and show soundness of the specifications carried on the types of Trip terms. In Section 4, we apply the proof delay applicative to Trip and create a readable surface syntax for the language using Agda's new macro system, and demonstrate our verification framework on a number of examples. We discuss the design of both Trip and our proof delay applicative in Section 5, as well as examine related and future work.

2 The Proof Delay Applicative

A computation of a type X which may delay some obligations until later is written as Delay X. The Delay type is defined as follows:

```
record Delay (X : Set ℓ) : Set (Level.suc ℓ) where
  constructor Prf
  field
    goals : List Set
    prove : HList goals → X
```

The field goals is a list of types, containing each of the propositions that must be proven in order to produce our result of type X. The prove field is the actual computation of the X value that requires proof of each of the propositions in goals.

The type HList is a heterogenous list [21], indexed by a list of types corresponding to the type for each element:

```
data HList : List Set → Set where
  [] : HList []
  _::_ : ∀{S}{SS} → S → HList SS → HList (S :: SS)
```

2.1 Construction

There are two operations to construct a basic Delay computation. The first is pure, which given a value, produces a Delay computation that returns that value without deferring any proofs until later:

```
pure : X → Delay X
pure x = Prf [] (const x)
```

The second operation is later, which, as the name suggests, constructs a Delay X by requiring a value of type X to be provided later:

```
later : ∀{X} → Delay X
later {X} = Prf (X :: []) λ { (x :: []) → x }
```

2.2 Composition

We compose Delay computations using the application operator ⊛ which applies a Delay computation of a function to a Delay computation of its argument by requiring the delayed obligations of *both* computations to be satisfied before computing the function application:

```
_⊛_ : Delay (A → B) → Delay A → Delay B
Prf goals₁ prove₁ ⊛ Prf goals₂ prove₂
  = Prf (goals₁ ++ goals₂)
        λ hl → prove₁ (takeH hl) (prove₂ (dropH hl))
```

Because we concatenate (using ++) the goals of the function computation with the goals of the argument computation, when we actually discharge these obligations the incoming HList will be similarly concatenated. We use the functions takeH and dropH to break apart this HList in order to discharge the obligations of each sub-computation. While analogous to the conventional take and drop functions on homogenous lists, the HList versions of these functions have only one explicit parameter, inferring the size of the two sublists from the provided type indices:

```
takeH : ∀{SS TS} → HList (SS ++ TS) → HList SS
takeH {[]} ys = []
takeH {S :: SS} (x :: xs) = x :: takeH {SS} xs

dropH : ∀{SS TS} → HList (SS ++ TS) → HList TS
dropH {[]} ys = ys
dropH {S :: SS} (x :: xs) = dropH {SS} xs
```

2.3 Syntactic niceties

The two operations ⊛ and pure make the type Delay into an *applicative functor* [32]. This allows us to use the *idiom brackets* notation of McBride and Paterson [32], as implemented in Agda version 2.6.0, to express Delay computations. These brackets denote nested Delay applications, for example pure f ⊛ a ⊛ b ⊛ c, as pure function applications within banana brackets, i.e. (| f a b c |).

We also define a synonym for the prove field selector using Agda's mix-fix syntax, providing a clear syntactic delineation between overall structure and detailed proof:

```
structure:_proofs:_done = Delay.prove
```

2.4 A non-monadic applicative functor

Unlike many of the applicative functors in common use in functional programming, our Delay functor is *not* a monad. This becomes apparent when we examine the type of a hypothetical monadic bind operation for Delay:

```
_>>=_ : Delay A → (A → Delay B) → Delay B
```

Like ⊛, the >>= operator gives us a way to compose two Delay computations, but unlike ⊛, the second computation is *dependent* on the results of the first. This means that if our Delay type were a monad, we could not determine all the deferred obligations of a computation at once, as some of these obligations would only be deferred *after* proofs for other goals had been provided.

As our HList type is an n-ary generalisation of a simple *non-dependent* product type, we have no way to express such a dependency, and thus Delay is not a monad. We can remedy this by defining our Delay type more generally, using a single type for its deferred obligations, rather than a list of types:

```
record MDelay (X : Set ℓ) : Set (Level.suc ℓ) where
  constructor MPrf
  field
    goals : Set
    prove : goals → X
```

The application operation now combines the goals of the two subcomputations with a simple product type:

```
_⊛_ : MDelay (A → B) → MDelay A → MDelay B
d₁ ⊛ d₂ = MPrf (goals d₁ × goals d₂)
            λ { (p₁ , p₂ ) → prove d₁ p₁ (prove d₂ p₂) }
```

Furthermore, the monadic join operator can now be defined using a Σ-type to express the telescopic dependency between goals:

```
join : MDelay (MDelay A) → MDelay A
join d = MPrf (Σ (goals d) (goals ∘ prove d))
           λ { (g , g′) → prove (prove d g) g′ }
```

This definition is slightly simpler than that of Delay, but it results in the type of deferred obligations forming a tree structure that resembles the structure of the outline we have already written. We have to describe our structure twice: once in the outline, and once in the proofs. Furthermore, we are repeating ourselves for little to no apparent benefit, as our Delay type is sufficient for even the most sophisticated examples in this paper.

The greater static knowledge afforded by applicative functors has lead to them being preferred over monads in many domains for efficiency reasons [7, 28, 29, 35]. In our case, however, we prefer applicative functors to maintain a clean separation of concerns between structure and proof.

2.5 Small examples

Recalling the ordered list data type introduced in Section 1, we now wrap each constructor in the Delay applicative, deferring any ordering proofs until later:

```
nil : Delay (OList m n)
nil = (| Nil later |)

_cons_ : (x : ℕ) → Delay (OList x n) → Delay (OList m n)
x cons xs = (| (Cons x) later xs |)
```

These constructors allow ordered lists to be constructed just as any other list, albeit within the proof delay applicative. To extract the final OList value, we must provide a HList of ordering proofs:

```
example : OList 1 5
example = structure: 1 cons 2 cons 3 cons 4 cons 5 cons nil
          proofs:
            s≤s z≤n
          :: s≤s z≤n
          :: s≤s (s≤s z≤n)
          :: s≤s (s≤s (s≤s z≤n))
          :: s≤s (s≤s (s≤s (s≤s z≤n)))
          :: s≤s (s≤s (s≤s (s≤s (s≤s z≤n))))
          :: []
          done
```

Our binary search tree example is similar, wrapping the constructors in the Delay applicative and deferring the ordering proofs on Leaf nodes until later:

```
leaf : Delay (BST m n)
leaf = (| Leaf later |)

branch : (x : ℕ) → Delay (BST m x) → Delay (BST x n)
           → Delay (BST m n)
branch x l r = (| (Branch x) l r |)
```

With this example we can begin to glimpse the usefulness of the Delay applicative for describing proof-carrying data structures. The tree structure of the data is visually apparent, and the uninteresting proofs are relegated to a separate section of code, where they can be conveniently omitted from this paper in the interest of brevity:

```
example₂ : BST 2 10
example₂ = structure: branch 3
                        (branch 2 leaf leaf)
                        (branch 5
                          (branch 4 leaf leaf)
                          (branch 10 leaf leaf))
           proofs: ⟨omitted for brevity⟩
           done
```

programs	$\mathcal{P}, \mathcal{Q}$	::=	$\mathcal{P}; \mathcal{Q}$	(seq. composition)
		|	$\mathcal{P} + \mathcal{Q}$	(nondet. choice)
		|	$\mathcal{P}^\star$	(Kleene star)
		|	g	(guard)
		|	$\mathcal{U}$	(state update)
updates	$\mathcal{U}$	$\in$	$\Sigma \to \Sigma$	
assertions	φ, ψ, α, g	$\in$	$\Sigma \to \mathbb{B}$	
states	Σ			
booleans	$\mathbb{B}$			

Figure 1. The language of Regular Imperative Programs

Ordering proofs are easily decidable and therefore not difficult to automate, either using Agda's proof search features [26] or by exploiting computation within the Agda type checking process [35]. Initial experiments show that these techniques are highly compatible with our approach, discharging all deferred ordering obligations entirely automatically.

3 The Core of Trip

A very similar distinction between structure and proof can be found in software verification, particularly when verifying imperative programs using program logics such as those of Floyd [14] or Hoare [18]. Typically, we would write out our program in full, interspersed with local *assertions* about the state. By using the axioms of the program logic, we then derive a set of *verification conditions*, i.e. logical formulae that together imply that our assertions hold for all executions of our program. The verification is completed by discharging each of the verification conditions by proof.

3.1 Regular Imperative Programs

Before verifying imperative programs we must first define an imperative language in which to write them. We will base our definitions on the language of regular imperative programs, so named for its resemblance to regular expressions, presented in Figure 1. Expressing imperative programs as a Kleene algebra in this way is not new, but the exact origins of this language are unclear, emerging from folklore in the Netherlands and the US during the mid-1970s [8, 13, 17, 39]. The semantics of this language are given in terms of binary *relations* on states. For a given program $\mathcal{P}$, we say $(\sigma_1, \sigma_2) \in [\![\mathcal{P}]\!]$ iff the state σ_2 could result from executing $\mathcal{P}$ in state σ_1. The semantics are given as relations, rather than functions, because we allow our programs to be nondeterministic, with the choice operator being written as $\mathcal{P} + \mathcal{Q}$. This non-determinism can be constrained by using guard statements (g), which only execute when the guard g is satisfied. For example, traditional **if** statements can be recovered using this translation:

$$\textbf{if } g \textbf{ then } \mathcal{P} \textbf{ else } \mathcal{Q} \textbf{ fi} \quad \simeq \quad (g; \mathcal{P}) + (\neg g; \mathcal{Q})$$

Similarly, standard loop constructs such as **while** can be encoded using the Kleene star $\mathcal{P}^\star$, which runs the program $\mathcal{P}$ a non-deterministic number of times:

$$\textbf{while } g \textbf{ do } \mathcal{P} \textbf{ od} \quad \simeq \quad (g; \mathcal{P})^\star; \neg g$$

The language is parametric in the definition of states (Σ). For the verification of small programs or specific algorithms, states are usually defined to be the mapping of all variable names to their values, but for large-scale software verification projects, this definition could be expanded to include models of hardware or heap memory [44].

3.2 Typed Regular Imperative Programs

The core of our language Trip is an Agda encoding of the language in Figure 1, where program terms are typed by their Hoare logic specifications.

Our development is similarly parameterised by the type used to represent states. We define an Assertion to be a state-dependent proposition:

```
Assertion = {⦇ σ : State ⦈} → Set
```

We use Agda's *instance arguments* [9] for the state parameter so that assertions can be written as simple logical formulae such as `n > 0` rather than less readable lambda abstractions like `(λ σ → n σ > 0)`.

The type of Trip programs consists of a pair of Assertions, denoting its the pre- and post-conditions respectively. If a term P has type $[\ \varphi\ ,\ \psi\ ]$, that means that all executions of P from a state where φ holds will have a final state that satisfies ψ. This means that a typing judgement $P : [\ \varphi\ ,\ \psi\ ]$ is equivalent to a Hoare triple $\{\varphi\}\ P\ \{\psi\}$. We prove this theorem with respect to the relational semantics of Trip in Section 3.4.

```
data [_,_] : Assertion → Assertion → Set₁ where
```

The types of our terms are based on the Hoare logic rules for regular imperative programs, given in Figure 2. For sequential composition, we have an intermediate assertion α which is the post-condition of the first program and the pre-condition of the second:

```
SEQ : [ φ , α ] → [ α , ψ ] → [ φ , ψ ]
```

Non-deterministic choice requires the two operands to have the same specification:

```
CHO : [ φ , ψ ] → [ φ , ψ ] → [ φ , ψ ]
```

Because the Kleene star can run the given program any number of times, including zero, it must maintain the same assertion (the *loop invariant*) throughout:

```
STAR : [ φ , φ ] → [ φ , φ ]
```

Guard statements only successfully execute when the given guard holds. Therefore, any assertion that follows from the guard in the pre-condition is a valid post-condition.

```
GUARD : (g : Assertion) → [ (g → φ) , φ ]
```

$$\textsc{Seq}\ \frac{\{\varphi\}\ \mathcal{P}\ \{\alpha\} \qquad \{\alpha\}\ \mathcal{Q}\ \{\psi\}}{\{\varphi\}\ \mathcal{P};\mathcal{Q}\ \{\psi\}} \qquad \textsc{Choice}\ \frac{\{\varphi\}\ \mathcal{P}\ \{\psi\} \qquad \{\varphi\}\ \mathcal{Q}\ \{\psi\}}{\{\varphi\}\ \mathcal{P}+\mathcal{Q}\ \{\psi\}} \qquad \textsc{Star}\ \frac{\{\varphi\}\ \mathcal{P}\ \{\varphi\}}{\{\varphi\}\ \mathcal{P}^\star\ \{\varphi\}}$$

$$\textsc{Guard}\ \frac{}{\{g \to \varphi\}\ g\ \{\varphi\}} \qquad \textsc{Update}\ \frac{}{\{\varphi \circ \mathcal{U}\}\ \mathcal{U}\ \{\varphi\}} \qquad \textsc{Consequence}\ \frac{\varphi \to \varphi' \qquad \{\varphi'\}\ \mathcal{P}\ \{\psi'\} \qquad \psi' \to \psi}{\{\varphi\}\ \mathcal{P}\ \{\psi\}}$$

Figure 2. The Hoare logic rules for regular imperative programs.

In Figure 1, state updates are modelled as merely functions from state to state, with an update axiom in Figure 2 that expresses the pre-condition in terms of the post-condition. In Trip, however, we use a more general type that allows the update function to make use of knowledge from the statement's pre-condition. Here, state updates are a function from states that satisfy the pre-condition φ to states that satisfy the post-condition ψ:

```
UPD : (Σ: φ → Σ: ψ) → [ φ , ψ ]
```

In a delightful confluence of notation, $\Sigma\!:\ \varphi$ is a dependent product of a state σ and a proof that φ holds for σ:

```
Σ:_ : Assertion → Set
Σ: φ = Σ State (λ σ → φ ⟪ σ ⟫)
```

Our state update is general enough to encode other traditional Hoare logic staples. For example, the rule of consequence in Figure 2, which allows us to move from one assertion to another by proving a logical implication, can be viewed as a state update that leaves the state unchanged:

```
CONS : Π: (φ → ψ) → [ φ , ψ ]
CONS φ→ψ = UPD λ { ( σ , φ ) → σ , (φ→ψ ⟪ σ ⟫ φ) }
```

Here $\Pi\!:\ \varphi$ is the dual of our earlier $\Sigma\!:\ \varphi$. The notation $\Pi\!:\ \varphi$ states that φ holds under *any* state:

```
Π:_ : Assertion → Set
Π: φ = (∀ ⟪ σ : State ⟫ → φ ⟪ σ ⟫)
```

The venerable no-op **skip** statement is just CONS with a tautology:

```
SKIP : [ φ , φ ]
SKIP = CONS (λ x → x)
```

3.3 Semantics

Like the language on which it is based, we shall give the semantics of Trip in terms of binary relations on states:

```
StateRel = State → State → Set
```

We shall need some operations on state relations, specifically forward composition ($\fatsemi$) and union ($\cup$):

```
_⨟_ : StateRel → StateRel → StateRel
(R ⨟ S) σ₁ σ₃ = ∃[ σ₂ ] (R σ₁ σ₂ × S σ₂ σ₃)
```

```
_∪_ : StateRel → StateRel → StateRel
(R ∪ S) σ₁ σ₂ = R σ₁ σ₂ ⊎ S σ₁ σ₂
```

In addition, we also define the reflexive transitive closure $R\star$ of a relation R as an inductive datatype:

```
data _★ (R : StateRel) : StateRel where
  refl : (R ★) σ σ
  step : R σ₁ σ₂ → (R ★) σ₂ σ₃ → (R ★) σ₁ σ₃
```

These operators allow us to give semantics to sequential composition, non-deterministic choice and Kleene star respectively:

```
⟦_⟧ : [ φ , ψ ] → StateRel
⟦ SEQ P Q  ⟧ = ⟦ P ⟧ ⨟ ⟦ Q ⟧
⟦ CHO P Q  ⟧ = ⟦ P ⟧ ∪ ⟦ Q ⟧
⟦ STAR P   ⟧ = ⟦ P ⟧ ★
⟦ GUARD g  ⟧ = ⟦ g ⟧g
⟦ UPD u    ⟧ = ⟦ u ⟧u
```

The semantics of guard statements $[\![\ g\ ]\!]\mathrm{g}$ are the subset of the identity relation where the state satisfies g:

```
⟦_⟧g : Assertion → StateRel
⟦ g ⟧g σ₁ σ₂ = g ⟪ σ₁ ⟫ × σ₁ ≡ σ₂
```

State update statements are given semantics simply by interpreting the state update function as a relation:

```
⟦_⟧u : (Σ: φ → Σ: ψ) → StateRel
⟦ u ⟧u σ₁ σ₂ = ∀ prf-φ → let σ₂′ , _ = u (σ₁ , prf-φ)
                         in  σ₂ ≡ σ₂′
```

3.4 Soundness

Our soundness result connects the specification of a program to its semantics. It states that for any execution of a program $P : [\ \varphi\ ,\ \psi\ ]$ from initial state σ_1 to final state σ_2, if the initial state σ_1 satisfies the pre-condition φ then the final state σ_2 will satisfy the post-condition ψ.

```
sound : (P : [ φ , ψ ]) → ⟦ P ⟧ σ₁ σ₂ → φ ⟪σ₁⟫ → ψ ⟪σ₂⟫
```

Recall that the typing rule for a sequential composition $P\ ;\ Q$ requires an intermediate assertion α as both the post-condition for P and the pre-condition for Q. Therefore, soundness for the sequential composition is established simply by

the composition of inductive hypotheses for P and Q by transitivity of implication (i.e. function composition):

```
sound (SEQ P Q) (_ , p , q) = sound Q q ∘ sound P p
```

For a non-deterministic choice $P + Q$, soundness follows straightforwardly from the inductive hypotheses, depending on which of P or Q was the origin of the execution in question:

```
sound (CHO P Q) (inj₁ p) = sound P p
sound (CHO P Q) (inj₂ q) = sound Q q
```

Soundness for Kleene star is established by a structural induction on the reflexive transitive closure, effectively unrolling the inductive step of $\mathcal{P}^\star$ to the equivalent $(\mathcal{P}; \mathcal{P}^\star)$ and the base case to **skip**:

```
sound (STAR P) refl       = id
sound (STAR P) (step p ps) = sound (STAR P) ps ∘ sound P p
```

A guard g has a pre-condition of $g \longrightarrow \varphi$ and a post-condition of φ. Seeing as the semantics of guards require all executions to satisfy g, we can establish the post-condition by modus ponens (i.e. function application):

```
sound (GUARD g) (p , refl) = _$ p
```

For state updates, the state update function itself carries the proof of its soundness. Therefore, the overall soundness proof must merely observe the determinism of the update function when interpreted as a relation:

```
sound (UPD x) (p) = snd-upd x p
  where
    snd-upd : (u : Σ: φ ⟶ Σ: ψ)
            → ⟦ u ⟧u σ₁ σ₂ → φ ⦃ σ₁ ⦄ → ψ ⦃ σ₂ ⦄
    snd-upd u sem prf-φ with u (_ , prf-φ) | sem prf-φ
    ... | _ , prf-ψ | refl = prf-ψ
```

3.5 Deterministic constructs

Now that we have defined our core language and established the soundness of its Hoare logic types, any derivable language construct from that sound core is necessarily also sound. For example, the deterministic conditional **if** statement we derived earlier can be given a type resembling the typical Hoare logic rule for **if**:

```
IFTHENELSE : (g : Assertion)
           → [ φ × g , ψ ]
           → [ φ × ¬ g , ψ ]
           → [ φ , ψ ]
```

The implementation is comprised solely of already-defined constructs. The translation is essentially the same as the one in Section 3.1, with the addition of the rule of consequence, used to make the assertions fit together:

```
IFTHENELSE g P Q
  = CHO (SEQ (SEQ (CONS _,_) (GUARD g)) P)
        (SEQ (SEQ (CONS _,_) (GUARD (¬ g))) Q)
```

The rule for **while** loops is also similar to Section 3.1, save for the addition of the rule of consequence:

```
WHILE : (g : Assertion) ⟶ [ g × φ , φ ] ⟶ [ φ , ¬ g × φ ]
WHILE g P = SEQ (STAR (SEQ (SEQ (CONS (flip _,_))
                                 (GUARD g))
                           P))
                (SEQ (CONS (flip _,_))
                     (GUARD (¬ g)))
```

4 The Surface of Trip

As can be glimpsed in our core language translations for **if** and **while**, nesting proof terms directly inside the program whenever the rule of consequence is used produces significantly less readable results than a typical pen-and-paper Hoare logic derivation. This is for two main reasons:

- The proof terms do not indicate what the intermediate assertion being established is, but merely how to prove it.
- The proof terms clutter the program with a potentially large number of terms that are not computationally relevant.

In pen-and-paper Hoare logic derivations, the use of the rule of consequence is usually left implicit. Instead, the program is annotated with an assertion, and the proof of the implication required to establish that assertion is presented afterwards.

For the surface language of Trip, we can achieve a similar effect with our Delay applicative. In our surface language, use of the consequence rule is indicated by an assert statement that uses the Delay applicative to defer proving the implication until later:

```
assert : (φ : Assertion) ⟶ Delay [ φ , ψ ]
assert φ = ⦇ CONS later ⦈
```

The user provides the *pre-condition* φ explicitly but the post-condition is left implicit. This is because of the structure of many of the Hoare logic rules given in Section 3.2 are such that the pre-condition is inferrable from the post-condition, in the spirit of the weakest pre-condition calculi of Dijkstra [10]. Thus it is more likely that Agda will be able to infer the post-condition ψ from the subsequent parts of the program, and require specification of the pre-condition φ.

We also define wrappers for sequential composition, **if**, **while**, and state update in our Delay applicative:

```
P ; Q = ⦇ SEQ P Q ⦈

if g then p else q fi = ⦇ (IFTHENELSE g) p q ⦈

while g begin P end = ⦇ (WHILE g) P ⦈

upd u = pure (UPD u)
```

4.1 Initial attempt at swap

With the basic constructs of our surface language defined, we can now attempt to write and verify some basic programs. One of the simplest is the program that swaps two variables using a temporary storage variable.

For this program we shall define our state parameter to be the following record type:

```
record SwapState : Set where
  field
    i : ℕ
    j : ℕ
    temp : ℕ
```

The specification of a swap procedure is interesting because it requires the use of logical constants or *freeze variables* to refer to the values that the variables i and j had at the beginning of the program. In Trip, ordinary Agda variables, acting as metavariables for Trip, can fulfil this purpose:

```
swp : ∀{I J : ℕ} → [ i ≡ I × j ≡ J , j ≡ I × i ≡ J ]
```

Unfortunately, the syntax for state updates in our implementation still leaves much to be desired. Also, the proof term p can be seen in the structure section, despite the fact that we do not want to nest proofs within the program structure:

```
swp = structure:
        upd (λ { (σ , p) → record σ { temp = i ⦃ σ ⦄ } , p }) ;
        upd (λ { (σ , p) → record σ { i = j      ⦃ σ ⦄ } , p }) ;
        upd (λ { (σ , p) → record σ { j = temp ⦃ σ ⦄ } , p })
        proofs: []
        done
```

4.2 Record update macros

Ideally, we would like to write simple assignment statements such as temp := i rather than the syntactically noisy record update syntax built in to Agda. Because Agda's record system does not support bidirectional first class accessors, however, we cannot define such a statement as an ordinary Agda definition. Instead, we must turn to meta-programming.

Agda's meta-programming facilities directly expose the implementation of parts of the Agda type checker and elaborator behind an interface in a monad called TC. This feature is similar to elaborator reflection in Idris [3] or tactic metaprogramming in Lean [11]. We define a meta-program that, given a field name, generates a setter function for the state record which updates that field:[1]

```
fieldSetter : {X : Set} → Name → TC (X → State → State)
```

We also define a version of the Hoare logic update axiom from Figure 2 as a special case of our general rule for state updates. This assignment principle is parameterised by our setter function and a (possibly state-dependent) value with which to update our variable:

```
assn : {φ : Assertion}
       {X : Set}
       (e : {⦃ σ : State ⦄} → X)
       (set : X → State → State)
     → Delay [ (λ ⦃ σ ⦄ → φ ⦃ set e σ ⦄)
             , (λ ⦃ σ ⦄ → φ ⦃ σ ⦄) ]
assn e set = upd λ { (σ , p) → (set (e ⦃ σ ⦄) σ , p) }
```

Lastly, we define an Agda macro to generate the appropriate code for the syntax $a := b$. An Agda macro is a TC procedure that generates a term to unify with a hole, which is always the last argument given to the macro. Some of the macro's arguments can also be *quoted*, i.e. represented as an explicit syntax tree. In our macro, the only quoted term is the field name, used to generate the field setter which is in turn used as the parameter to our assignment principle:

```
macro
  assnM : {φ : Assertion}{X : Set}
          (e : {⦃ σ : State ⦄} → X)
        → Name
        → Term → TC ⊤
  assnM {φ} e fld hole = do
    setter ← fieldSetter fld
    trm ← quoteTC (assn {φ} e setter)
    unify hole trm
  syntax assnM b a = a := b
```

4.3 Swap, redux

With our new assignment macro, our swap procedure is much more palatable:

```
swp′ : ∀{I J : ℕ} → [ i ≡ I × j ≡ J , j ≡ I × i ≡ J ]
swp′ = structure:
         temp := i ; i := j ; j := temp
       proofs: []
       done
```

Because our assn principle once again allows (weakest) preconditions to be mechanically derived from the post-condition, no in-program assertions or proofs are necessary here.

4.4 Guarded assignments

Suppose we now wished to write and verify program that would compute the sum of all the elements in a list. Our state would include the list itself, the current total, and a loop counter i:

```
record SumState : Set where
  field
    i : ℕ
```

[1] We omit implementation here in the interest of brevity, as it is over 50 lines long.

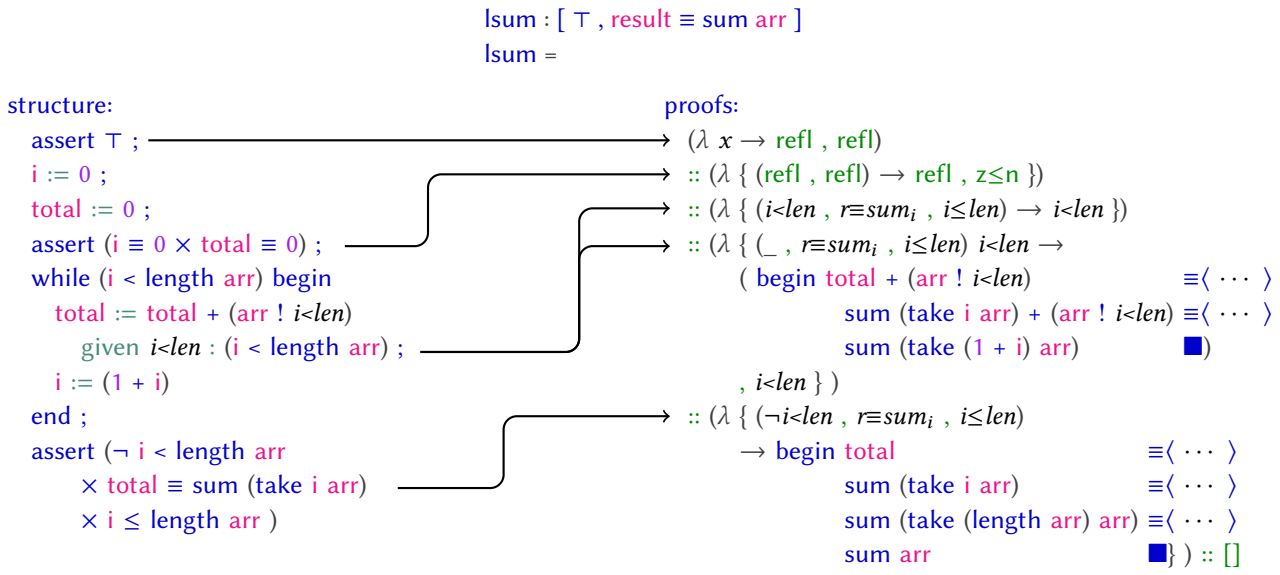

Figure 3. Verified list sum. Our verification condition generator in action.

```
arr : List ℕ
total : ℕ
```

Our imagined program would resemble the following, assuming some appropriate list indexing operation ‼:

```
i := 0 ;
total := 0 ;
while (i < length arr) begin
  total := (total + arr ‼ i) ;
  i := (1 + i)
end
```

Unfortunately, such a list indexing operation would need a type like:

```
_‼_ : List A → ℕ → A
```

This type is not inhabited in Agda, where function types refer to *total* functions, as there is no valid behaviour if the given list index is out of range. Even though we know from our assertions that every indexing is in range, the type of the function does not reflect this knowledge. Therefore, we shall use a more precise type instead, which requires evidence that the index is valid for the list:

```
_!_ : ∀ {n} → (ls : List A) → n < length ls → A
_!_ {n = suc _} (x :: ls) (s≤s n) = ls ! n
_!_ {n = zero } (x :: ls) (s≤s n) = x
```

This corrected indexing operation presents us with a problem, however, because it means that the expression used to update the total in our example now needs access to a *proof* that the given index is less than the length of the list. We know that the index is in bounds from our assertions, but there is no means in our current syntax to get a proof of that fact into a variable assignment.

Therefore, we introduce a more general form of assignment syntax, called a *guarded assignment*. With it, we could write our total update as:

```
total := total + (arr ! i<len) given i<len : (i < length arr)
```

This additional assertion (i < length arr) , called the guard, is proven to follow from the statement's pre-condition as a deferred obligation, separately from the program.

The assignment principle for this syntax is significantly more general than our principle for simple assignments (assn). Because the update expression now depends upon the pre-condition, the pre-condition can no longer be merely the post-condition composed with the update. Instead, we require a proof that the pre-condition φ implies the guard α, and a proof that the post-condition ψ holds after the state has been updated:

```
grd : {φ ψ α : Assertion}{X : Set}
    (e : {| σ : State |} → α {| σ |} → X)
    (set : X → State → State)
    → Delay (Π: (φ → α))
    → Delay (Π: λ {|σ|} → φ → (g : α) → ψ {|set (e g) σ|})
    → Delay [ φ , ψ ]
grd e set p₁ p₂ = ⦇ update p₁ p₂ ⦈
  where update = λ φ→α φ→α→ψ → UPD λ where
          (σ , φ) → let
              α = φ→α {| σ |} φ
              ψ = φ→α→ψ {| σ |} φ α
            in set (e {| σ |} α) σ , ψ
```

```
csum : ∀{arr₀}
     → [ arr ≡ arr₀ × length arr > 0
       , length arr ≡ length arr₀
       × (∀ i (i < length arr)
         → arr [ i ]= sum (take (suc i) arr₀) )
       ]
csum {arr₀} = let
  Inv : ℕ → ℕ → Assertion
  Inv i i′ ⦃ σ ⦄ = (length arr ≡ length arr₀)
                 × (i ≤ length arr)
                 × (∀ j (j<i : j < i)
                   → arr [ j ]= sum (take (suc j) arr₀))
                 × drop i arr ≡ drop i arr₀
                 × total ≡ sum (take i′ arr₀)
in structure:
  assert (arr ≡ arr₀ × length arr > 0) ;
  i := 0 ;
  total := 0 ;
  assert (i ≡ 0 × total ≡ 0 × arr ≡ arr₀ × i < length arr) ;
  while (i < length arr) begin
    total := total + (arr ! i<len)
      given i<len : (i < length arr) ;
    assert ( (i < length arr) × Inv i (suc i)) ;
    arr := (arr [ i<len ]:= total)
      given i<len : (i < length arr) ;
    assert (Inv (suc i) (suc i)) ;
    i := (1 + i)
  end ;
  assert (¬ (i < length arr) × Inv i i )
  proofs: ⟨ 100 lines of proof ⟩ done
```

Figure 4. Cumulative sum

Our macro for the syntax is broadly similar to the macro for assn, except that it uses the proof delay applicative to defer proving both of the implications until later:

```
guardedM : {φ ψ : Assertion}{X : Set}(α : Assertion)
           (e : ⦃ σ : State ⦄ → α ⦃ σ ⦄ → X)
         → Name
         → Term → TC ⊤
guardedM {φ} {ψ} α e fld hole = do
  setter ← fieldSetter fld
  trm ← quoteTC (guarded {φ} {ψ} e setter later later)
  unify hole trm
syntax guardedM τ (λ g → v) fld = fld := v given g : τ
```

This syntax is general enough to encompass any assignment we could need, but it does generate two deferred obligations. Therefore, the more specialised simple assignment syntax is still preferable wherever possible.

4.5 Verified list sum

Having now completed our definitions for the surface syntax of Trip, we can now return to our list sum problem. Figure 3 shows a verified list sum implementation in Trip, with the proofs slightly expurgated for presentation purposes. The arrows in the figure show how each proof obligation is generated, either from an assert statement or a guarded assignment.

We use an explicit assert at the end of the loop to fix the loop invariant, which expresses that the total will be the sum of the first i elements of the list, as well as the book-keeping invariant that i is at most the length of the list.

4.6 A more involved example

Figure 4 shows a more involved example of a verified Trip program. This program uses the same SumState type as the previous list sum example, but this time the list is mutated to contain the running total up to that point in the original list. For example, the list 3 :: 1 :: 7 :: 9 :: [] would become 3 :: 4 :: 11 :: 20 :: [] after executing csum.

This example necessitates a few more definitions for dealing with lists. Firstly, we need a way to update a list at a particular index. Like our previous list getter, this will require a proof that the desired index is in range:

```
_[_]:=_ : (xs : List A){n : ℕ} → n < length xs → A → List A
_[_]:=_ (x :: xs) {zero} n<len a = a :: xs
_[_]:=_ (x :: xs) {suc n} (s≤s n<len) a = x :: (xs [ n<len ]:= a)
```

We must also define a predicate that holds iff the given element is in the given list at a given index, for use in our specifications and assertions. This is just a dependent pair of a proof that the index is in range and a proof that the element is found at that index:

```
_[_]=_ : List A → ℕ → A → Set
xs [ n ]= v = Σ (n < length xs) λ n<len → (xs ! n<len) ≡ v
```

The left hand column of Figure 4 provides the specification and the loop invariant for our program. The specification makes use of a logical freeze variable to refer to the original list before any changes are made. The post-condition states that the length of the list is unchanged, and that the ith element of the list will be the sum of the first $i+1$ elements of the original list. As can be seen by the use of Agda quantifiers inside the specification, the full power of Agda as a logic is available when specifying Trip programs.

The loop invariant for csum is somewhat involved, consisting of five conjuncts, stating:

1. that the length of the list is unchanged,
2. the book-keeping invariant that the loop counter i is at most the length of the list,
3. that the list up to i consists of our desired running totals,
4. that the list from i onward is still identical to the original list, and
5. that the variable total is the sum of all elements so far.

The local definition of this loop invariant (*Inv*) is parameterised by the value of the variable i. This allows us to assert the invariant modulo substitutions for i throughout the loop (for example, after the assignment to total).

When the loop finishes and i is equal to the length of the list, the first and fourth conjuncts easily imply our post-condition. Nonetheless, establishing that the invariant is maintained and that the post-condition is met required approximately one hundred lines of Agda proof. As part of future work, we hope to integrate proof automation into our framework, hopefully reducing the amount of tedious proof obligations (see Section 5.3).

5 Discussion

5.1 Typed holes as a refinement calculus

One of the main motivations for encoding the specifications of Trip programs in their types is the ability to apply "hole-driven" development to derive the program from its specification. At any point while writing the program, we can leave a hole in a location in the program and Agda will present us with a specification, derived from the surrounding statements and assertions. Figure 5 provides an illustration of this process for our swap example.

Morgan [33] defines a minimal imperative language with *specification statements*, analogous to holes, which can be incrementally refined into an implementation through specialised axioms in a *refinement calculus*. The experience of deriving Trip programs from their specifications using typed holes strongly resembles a tool-assisted version of Morgan's refinement calculus.

One difference between Morgan's refinement calculus and our specification-typed holes is that Morgan's specification statements include a *frame*, denoting which variables may be modified by that part of the program. This is to enable compositional reasoning and to avoid having to explicitly freeze all unchanged state. Our framework is generic on the state, however, and does not explicitly model variables, which makes it difficult to impose frame restrictions on specific parts of the state. In future, we hope to add procedures and function calls into Trip, which would necessitate a more fine-grained handling of state. With that in place, we could also annotate our specifications with frames.

5.2 Related Work

5.2.1 Separating proofs from outlines

While our proof delay applicative functor is novel, there are some other approaches also aimed at the separation of proofs and outlines in other proof assistants:

The Coq `Program` tactic The Coq proof assistant [19] includes a feature called `Program` [42] where the algorithmic structure of a program may be written as normal, but the type of the program can form a rich specification. Proving that the program meets the specification is deferred until later, where the user will write a Coq proof script to discharge these deferred obligations.

Predicate subtyping in PVS The proof assistant PVS [37] has a similar feature to Coq's `Program` feature, where types are annotated with refinements, and the type checker generates proof obligations for the user to discharge [40].

Both of these approaches are designed as a built-in feature of the proof assistant, requiring extension to the tactic language or the type checking algorithm to enable interactive proof to supplement automatic typechecking for programs with predicate subtypes. By contrast, our applicative functor does not rely on language extensions or meta-programming, but rather encodes the delineation between structure and proof directly in Agda itself.

Ornaments The ornaments idea, introduced originally by McBride [30] and subsequently developed by many others [4, 5, 23, 45], is aimed at a similar fusion of internal and external approaches to proof in dependently-typed languages. Using ornaments, one can take a basic type and ornament it with additional type indices or proof obligations, such as turning a binary tree into a binary search tree. This gives rise to an isomorphism between the ornamented type and the product of the plain type and a descriptive structure that carries all the additional proof obligations. For our binary search tree example, it would establish an isomorphism between a binary search tree and the product of a plain binary tree with a proof that it satisfies the binary search tree invariants. Such an isomorphism is therefore performing much the same task as our Delay functor, allowing proofs to be deferred until after the basic structure is defined.

One of the main advantages of the ornaments approach is that by starting with a basic type and enriching it with further details, operations can be defined generically across many types that share the same structure. With our Delay functor approach, however, the user starts from a richly-typed data structure, and defers obligations as-needed. This is more flexible, in that the user can freely choose to defer some obligations and not others, but also precludes the kind of generic programming possible with ornaments.

From a practical standpoint, ornaments do not integrate with Agda as cleanly: they typically require encoding of

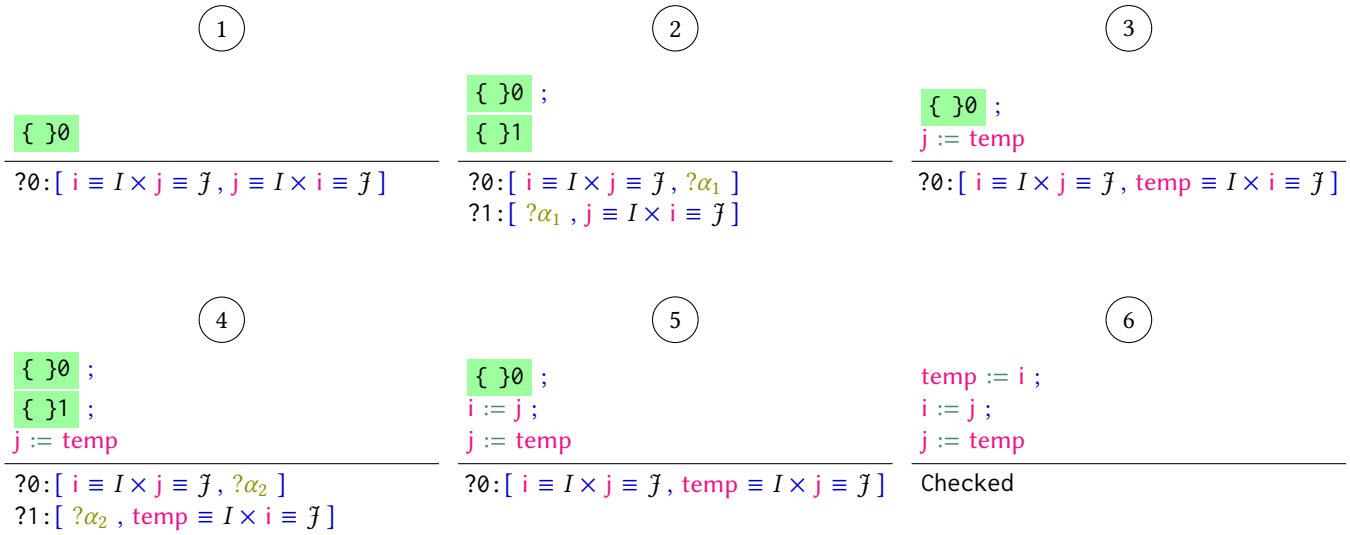

Figure 5. Typed holes as a refinement calculus

datatypes in a closed universe, and such encodings can be syntactically cumbersome. Our approach, on the other hand, is a standard applicative functor, for which Agda has built-in syntactic conveniences.

5.2.2 Imperative software verification

There have been countless other approaches to verifying imperative software, far too many to list here. We shall narrow our focus to those that are based on program logics and verification conditions.

Hoare logic languages and VCGs The widely-used verification framework Why3 [12] contains a stand-alone language for both programming and specification. It is not embedded in any particular proof assistant, but is capable of generating verification conditions in a variety of proof assistants, or deferring to automated theorem proving tools such as SMT solvers. By contrast, Trip's partially shallow embedding within Agda means that it is easily extensible just by writing Agda definitions, and specifications and proofs are written using standard Agda code.

The languages Whiley [38] and Dafny [25], as well as the VCC [6] tool for C programs, all rely on automated theorem provers exclusively. When these tools succeed, this is very convenient as no manual proving must be done. When they fail, however, it can be extremely difficult to coax the verification through the theorem prover, even when the user is capable of proving the verification conditions by hand.

VCGs in Proof Assistants The simpl language defined by Schirmer [41] strongly resembles Trip. Like Trip, it is embedded in a proof assistant (specifically Isabelle/HOL [34]). It is also generic on the type used for state, and shallowly embeds expressions and state updates. Unlike Trip, however, it has a deterministic semantics, and includes features like exceptions and procedures. It includes a verification condition generator, but it is defined as an Isabelle tactic, and is intended for use in *post-hoc* verification of software. Unlike Trip, it does not facilitate the derivation of a program's code from its specification.

Simpl has been used to give a semantics for C [46] using a typed memory model [44]. This C semantics was used for the verification of the seL4 microkernel [22] as well forming the basis of the automatic abstraction tool AutoCorres [15, 16]. This suggests that the expressivity of Simpl is sufficient for verification of software at scale, which bodes well for Trip.

The Bedrock framework [2] is another framework that is embedded inside a proof assistant, this time Coq. Bedrock targets low-level systems, and supports significant amounts of proof automation. Like Simpl, it is intended for *post-hoc* verification — using a verification condition generator tactic on a completed program.

Hoare-state indexed monads Swierstra [43] presents a variant of the state monad where monadic computations are typed by their specifications. It uses the aforementioned `Program` feature in Coq as a miniature verification condition generator. Unlike Trip, the Hoare-state monad is entirely shallowly-embedded inside Coq, and does not support nondeterminism.

5.3 Future Work

5.3.1 Other uses for the proof delay applicative

The proof delay applicative is very general, and there are likely many use cases we have not yet considered. In the area of software verification, verifying concurrent programs [36] often produces large numbers of proof obligations for non-interference, which would be better off delayed.

Other uses of the `Program` feature in Coq might also be good uses for the proof delay applicative. For example, an encoding of refinement types in Agda which delays refinement obligations until later is a promising avenue for further research. The wealth of previous work on ornaments [4, 5, 23, 30, 45] may also provide some interesting examples.

5.3.2 Proof automation

Often we wish to delay proofs until later because they are not interesting proofs. Certainly, when verifying Trip programs, many of the obligations generated are trivial. By integrating existing Agda proof automation techniques [1, 20, 24] with Trip, or with the proof delay applicative more generally, it may be possible to automatically discharge many of the deferred obligations, only requiring the user to prove those obligations that could not be proved automatically.

5.3.3 Language importer

As previously mentioned, the similar language Simpl has been used to give a semantics to C programs. Designing a similar language importer around Trip would need to generate Trip terms programmatically. Seeing as Agda's meta-programming interface does not (yet) support file I/O, this importer would likely have to be written outside Agda, generating and pretty-printing Agda text.

5.3.4 State management and procedures

For large-scale verifications to be feasible in Trip, we must extend the language to support procedures and function calls. Seeing as procedures also have local state, this would necessitate a more fine-grained handling of state than merely treating it as a monolithic abstract parameter, particularly if we wish to support recursion. As previously mentioned, the refinement calculus of Morgan [33] provides a possible method to handle procedures, local constants and local variables by integrating *frames* into the specification.

6 Conclusion

We have presented Trip, a proof-carrying imperative embedded language inside Agda which is typed by its specifications, to allow Hoare logic style derivation and verification of imperative programs. We presented a soundness proof of these specification-types against a relational semantics for the language. To avoid messy proofs and clutter in the Trip derivations, we introduced the proof delay applicative, which allows us to cleanly separate structure from proof not only in Trip derivations, but in general. The entire development outlined in this paper, including all proofs and auxiliary definitions, can be found at this address:

`http://www.github.com/liamoc/dddp`

Acknowledgments

I would like to thank Kai Engelhardt, who indirectly inspired this paper. Thanks also to Willem-Paul de Roever and Rob van Glabbeek who assisted me with research archaeology, and to anonymous reviewers who, among other suggestions, pointed out that ornaments are highly related.

References

[1] Guillaume Allais. 2015. Presburger Arithmetic Solver for Agda. https://github.com/gallais/agda-presburger.

[2] Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, 391–402. https://doi.org/10.1145/2500365.2500592

[3] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, 284–297. https://doi.org/10.1145/2951913.2951932

[4] Pierre-Evariste Dagand. 2017. The essence of ornaments. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796816000356

[5] Pierre-Evariste Dagand and Conor McBride. 2012. Transporting Functions Across Ornaments. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 103–114. https://doi.org/10.1145/2364527.2364544

[6] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. 2009. VCC: Contract-based modular verification of concurrent C. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. 429–430. https://doi.org/10.1109/ICSE-COMPANION.2009.5071046

[7] Justin Dawson, Mark Grebe, and Andy Gill. 2017. Composable Network Stacks and Remote Monads. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, 86–97. https://doi.org/10.1145/3122955.3122968

[8] W.P de Roever. 1979. Recursive program schemes: semantics and proof theory. Mathematical Centre tracts, no. 70. *Journal of Symbolic Logic* 44, 4 (1979), 658–659. https://doi.org/10.2307/2273307

[9] Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, 143–155. https://doi.org/10.1145/2034773.2034796

[10] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[11] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 34 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110278

[12] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.

[13] Michael J. Fischer and Richard E. Ladner. 1977. Propositional Modal Logic of Programs. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC '77)*. ACM, 286–294. https://doi.org/10.1145/800105.803418

[14] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of a Symposium on Applied Mathematics (Mathematical Aspects of Computer Science)*, J. T. Schwartz (Ed.), Vol. 19. American Mathematical Society, Providence, 19–31.

[15] David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *International Conference on Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Ed.). Springer, Princeton, New Jersey, USA, 99–115. https://doi.org/10.1007/978-3-642-32347-8_8

[16] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 429–439. https://doi.org/10.1145/2594291.2594296

[17] David Harel and Vaughan R. Pratt. 1978. Nondeterminism in Logics of Programs. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/512760.512782

[18] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[19] INRIA. 2009. The Coq Proof Assistant Reference Manual.

[20] Wojciech Jedynak. 2015. Ring Solver for Agda. https://github.com/wjzz/Agda-reflection-for-semiring-solver.

[21] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, 96–107. https://doi.org/10.1145/1017472.1017488

[22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[23] Hsiang-Shang Ko and Jeremy Gibbons. 2016. Programming with ornaments. *Journal of Functional Programming* 27 (2016). https://doi.org/10.1017/S0956796816000307

[24] Pepijn Kokke and Wouter Swierstra. 2015. Auto in Agda. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 276–301.

[25] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, 348–370.

[26] Fredrik Lindblad and Marcin Benke. 2006. A Tool for Automated Theorem Proving in Agda. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs (TYPES'04)*. Springer-Verlag, 154–169. https://doi.org/10.1007/11617990_10

[27] Sam Lindley and Conor McBride. 2013. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, 81–92. https://doi.org/10.1145/2503778.2503786

[28] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, 325–337. https://doi.org/10.1145/2628136.2628144

[29] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's Do-notation into Applicative Operations. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, 92–104. https://doi.org/10.1145/2976002.2976007

[30] Conor McBride. 2010. Ornamental Algebras, Algebraic Ornaments. (2010). http://personal.cis.strath.ac.uk/~conor/pub/OAAO/Ornament.pdf Manuscript available online.

[31] Conor McBride. 2014. How to Keep Your Neighbours in Order. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, 297–309. https://doi.org/10.1145/2628136.2628163

[32] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

[33] Carroll Morgan. 1994. *Programming from Specifications (2nd Ed.)*. Prentice Hall International Ltd.

[34] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. https://doi.org/10.1007/3-540-45949-9

[35] Liam O'Connor. 2016. Applications of Applicative Proof Search. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, 43–55. https://doi.org/10.1145/2976022.2976030

[36] Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs. *Acta Informatica* 6, 4 (01 Dec 1976), 319–340. https://doi.org/10.1007/BF00268134

[37] Sam Owre, John Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE '11)*, Deepak Kapur (Ed.). Springer Berlin Heidelberg, 748–752.

[38] David J. Pearce and Lindsay Groves. 2013. Whiley: A Platform for Research in Software Verification. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, 238–248.

[39] Vaughan R. Pratt. 1976. Semantical consideration on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science (SFCS '76)*. 109–121. https://doi.org/10.1109/SFCS.1976.27

[40] John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (Sep. 1998), 709–720. https://doi.org/10.1109/32.713327

[41] Norbert Schirmer. 2005. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Franz Baader and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–414.

[42] Matthieu Sozeau. 2007. Subset Coercions in Coq. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs (TYPES'06)*. Springer-Verlag, 237–252.

[43] Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. Springer-Verlag, 440–451. https://doi.org/10.1007/978-3-642-03359-9_30

[44] Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, Bytes, and Separation Logic. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, 97–108. https://doi.org/10.1145/1190216.1190234

[45] Thomas Williams and Didier Rémy. 2017. A Principled Approach to Ornamentation in ML. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 21 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158109

[46] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. Mind the Gap: A Verification Framework for Low-Level C. In *International Conference on Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Ed.). Springer, Munich, Germany, 500–515. https://doi.org/10.1007/978-3-642-03359-9_34