Proof Engineering with Predicate Transformer Semantics

Christa Jenkins[1,3][0000−0002−5434−5018], Mark Moir[2], and Harold Carr[3]

[1] The University of Iowa, Iowa City, Iowa, USA christa-jenkins@uiowa.edu
[2] Oracle Labs, New Zealand
[3] Oracle Labs, USA

Abstract. We present a lightweight, open source Agda framework for manually verifying effectful programs using predicate transformer semantics. We represent the abstract syntax trees (AST) of effectful programs with a generalized algebraic datatype (GADT) *AST*, whose generality enables even complex operations to be primitive AST nodes. Users can then assign bespoke predicate transformers to such operations to aid the proof effort, for example by automatically decomposing proof obligations for branching code. Our framework codifies and generalizes a proof engineering methodology used by the authors to reason about a prototype implementation of LibraBFT, a Byzantine fault tolerant consensus protocol in which code executed by participants may have effects such as updating state and sending messages. Successful use of our framework in this context demonstrates its practical applicability.

1 Introduction

Interactive theorem provers (ITPs) based on dependent type theory provide a flexible way to formally verify program properties, as they unify the language of specification and computation into an expressive pure functional programming language [9]. By virtue of referential transparency, properties of pure functions (those without side effects) can be proven with equational reasoning. However, when the computation of interest is inherently effectful, other techniques may be required. For example, in the case of distributed systems, participants perform state updates, emit messages, and invoke subroutines that may throw exceptions; the network's ability to tolerate faults rests upon participants' behaviors.

One approach to reasoning about effects, described by Swierstra and Baanen [13], is to model the abstract syntax tree (AST) of the effectful program with a datatype and assign to this type both an operational semantics and *predicate transformer* semantics (PTS). PTS provides a structured way for reasoning about effectful code [13, 11], reducing the goal of showing a given postcondition P holds of a program m to proving the weakest-precondition $wp\ m\ P$ of P w.r.t. that program. As $wp\ m$ maps *arbitrary* postconditions to preconditions, we may view it as giving a meaning (semantics) to m in terms of functions from postconditions to preconditions (predicate transformers).

```
prog : (St → Maybe Wr) → RWS Unit
prog g = pass inner where
  inner : RWS (Unit × (List Wr → List Wr))
  inner = do m ← gets g
             maybe (λ w → do tell [ w ]
                             return (unit , λ _ → []))
                   (return (unit , λ x → x ++ x)) m
```

Fig. 1: An example effectful program

Contributions In this paper, we describe a generic Agda framework for manually verifying effectful programs using PTS. This framework is designed to reduce the mental overhead for proof engineers (hereafter, "users") by tailoring the phrasing of intermediate proof obligations. In particular, our framework allows directly assigning predicate transformers not only to expected effectful operations, but also to monadic bind and pure operations for branching. Careful phrasing of proof obligations also facilitates a limited form of proof synthesis when the goal type can be decomposed with a unique type-correct constructor.

Our framework was developed as a part of our efforts to verify safety properties of an implementation of LibraBFT in Agda. We have previously reported on some aspects of that work [4]. Details of the work presented in this paper are available in the same open source repository [3]. LibraBFT (a.k.a. Diem-BFT) [2] is a real-world Byzantine-fault tolerant consensus protocol. We believe the techniques we describe generalize to other domains and ITPs, including those with greater levels of automation [1].

2 Proof Engineering with Predicate Transformers

As motivation, we consider the small effectful program *prog* in Figure 1. The type of *prog g* is *RWS Unit*[4], which says that it produces no interesting values (*Unit* is the unitary type) and that it uses the effects of the *reader, writer, state* monad [5]: it may read and write to state of type *St*, emit messages of type *Wr*, and read from an environment of type *Ev*. To avoid confusion, we say that an effectful program of type *RWS A* returns or *produces* a value of type *A*, as opposed to *emitting* messages (always of type *List Wr*). We assume the reader is familiar with Haskell-style **do**-notation and the basics of dependent type theory; we explain Agda-specific syntax and conventions as they are introduced.

Given a function *g* : *St* → *Maybe Wr* that may compute a message from a state, *prog g* calls *pass* on *inner*; *pass* has the effect of applying the function of type *List Wr* → *List Wr* returned by *inner* to the messages emitted by *inner*, the result of which is then emitted by *prog g*. In *inner*, we use *gets* to

[4] Many names in the repository—including *RWS*—have *AST* suffixes for disambiguation, which are usually omitted in this paper for brevity.

apply g to the current state, binding the result as $m : \mathit{Maybe}\ \mathit{Wr}$. Using the *maybe* operation to scrutinize m, if a message item $w : \mathit{Wr}$ is produced by g (i.e., $m \equiv \mathit{just}\ w$), then we emit it and return a constant function that returns the empty list, in effect erasing what was just emitted. In case $m \equiv \mathit{nothing}$, $\mathit{prog}\ g$ returns a function that concatenates a list of messages with itself.

The operational semantics for *RWS* programs is given by *runRWS*, whose type is shown below (the definition of *runRWS* is in module *Dijkstra.AST.RWS*; module *X.Y.Z* lives in `src/X/Y/Z.agda` in [3]).

```
Input      = Ev × St
Output A   = A × St × List Wr
runRWS : ∀ {A} → RWS A → Input → Output A
```

To run a program of type *RWS A*, *runRWS* requires as input an environment value and prestate, and the result of executing such a program is a triple consisting of a value of type *A*, a poststate, and a list of emitted messages.

Direct Approach to Proof Engineering Suppose we are tasked with verifying that *prog* satisfies postcondition *ProgPost* below, which expresses the property that the prestate and poststate are equal and that there are no outputs.

```
ProgPost : Input → Output Unit → Set
ProgPost (e₁ , s₁) (u₁ , s₂ , o) = s₁ ≡ s₂ × 0 ≡ length o
```

(*Set* is a classifier for types in Agda; the type of *ProgPost* says that it is a relation between inputs to, and outputs of, programs of type *RWS Unit*.)

Attempting the proof directly, we begin with:

```
progPost : ∀ g i → ProgPost i (runRWS (prog g) i)
progPost g (e , s) = ?
```

where the ? marks a hole in the proof. When Agda unfolds the proof obligation at the hole, the result is unwieldy! To give an idea, here is (a cleaned-up version of) the obligation for just the second conjunct of the postcondition *ProgPost*:

```
0 ≡ length (snd (fst (runRWS
      (maybe (return (unit , λ x → x ++ x))
        (λ w → tell [ w ] ≫ return (unit , λ _ → [])) (g s)) (e , s)))
      (snd (snd (runRWS
        (maybe (return (unit , λ x → x ++ x))
          (λ w → tell [ w ] ≫ return (unit , λ _ → [])) (g s)) (e , s)))))
```

(The proofs in this section are in the *Dijkstra.AST.Examples.PaperIntro* module with more detail, including the entire goal for the hole above in gory detail that a user attempting a direct proof encounters.)

This obligation has parts of the program text from *inner* repeated *twice,* once for the function it returns and once for the messages it emits. The issue is that the execution of *prog* is stuck on the branching operation *maybe* whose

scrutinee $g\ s$ is not in weak head normal form (that is, it is not of the form *nothing* or *just* x). Because our proof obligation refers directly to the evaluation of *prog* twice, the continuation in *inner* is also referenced twice.

One way to proceed in this case is to use the **with** keyword to abstract over the expression blocking progress:

```
progPost g (e , s) with g s
... | nothing = ?
... | just w  = ?
```

with the result that the new subgoals this generates ($s \equiv s \times 0 \equiv 0$ in both cases) are much easier to understand than the original goal.

Though *prog* is a small example, and the property we wish to verify is relatively simple, the preceding discussion illustrates difficulties that are greatly magnified as the complexity of the task increases. First, proof obligations for "stuck" code can explode in size, especially when the code branches, making it difficult to read the proof state and identify *why* execution is stuck. Second, once identified, using **with** to inspect the cases of the scrutinee of branching code means recapitulating the effectful operations performed up to that point. In the example considered above, there were no updates to the state s before the *gets* operation was used, but had there been for example a preceding *puts* p (for some $p : St \to St$), the user would instead have to write **with** $g\ (p\ s)$.

Simplifying Proof Obligations with PTS Our Agda framework is designed to address these issues. Following the general approach of Swierstra and Baanen [13], our framework uses a datatype for the ASTs of effectful computations (*AST*, Figure 2); *RWS* is an instance of this datatype. To prove that *ProgPost* holds of *prog*, we first prove a *precondition* obtained by the function

```
predTrans : ∀ {A} → RWS A → (Output A → Set) → Input → Set
```

which assigns to each *RWS* program (by induction over its AST) a function that maps postconditions to preconditions. Put another way, it gives each *RWS* program a *semantics* as a *predicate transformer*, as it transforms predicates over *Output* types to predicates over the *Input* type.

We begin our proof that precondition for *ProgPost* holds with:

```
progPostWP : ∀ g i → predTrans (prog g) (ProgPost i) i
progPostWP g (e , s) = ?
```

where the proof obligation at the hole is now much more understandable:

```
(r : Maybe Wr) → r ≡ g s →
      ((j : Wr) → r ≡ just j → (r₁ : Unit) → r₁ ≡ unit →
        (o′ : List Wr) → o′ ≡ [] → (s ≡ s) × (0 ≡ length o′))
  × (r ≡ nothing → (o′ : List Wr) → o′ ≡ [] →
        (s ≡ s) × (0 ≡ length o′))
```

The user's next few steps are entirely type directed: introduce the premises of the implication to the context, then (because this leaves us with a product) decompose the proof obligation into two subobligations (the detailed commentary in *Dijkstra.AST.Examples.PaperIntro* shows how to develop the proof largely automatcally using Emacs's Agda mode). This yields:

```
proj₁ (progPostWP g (e , s) m mId) = ?
proj₂ (progPostWP g (e , s) m mId) = ?
```

At this point, we note two things. First, the framework has enabled us to alias $g\ s$ as m (we choose the name m, rather than r as shown in the proof obligation, to better match the local name in *prog*), and the remaining proof obligations *only* mention m. Term $mId : m \equiv g\ s$ enables us to undo this aliasing as needed, as dependent pattern matching on *mId* replaces m with $g\ s$ in the proof state. Recall that, in the direct proof above, $g\ s$ was repeated in the proof obligation. If we had a more complex expression as the scrutinee, it too would be repeated in the direct proof—but in a proof using our framework, *only the alias* is repeated! Second, and relatedly, in the direct approach, the user must explicitly invoke **with** on the scrutinee $g\ s$ in order to generate two subobligations (one each for *nothing* and *just*). With our framework, these two cases are *already* present in the proof obligation, in the form of a product of obligations in which, while proving each component, the user assumes that the alias m is either *nothing* or *just* w for some $w : Wr$.

The rest of the proof is similarly type-directed by the framework and is fairly straightforward; see *Dijkstra.AST.Examples.PaperIntro* for the details. Finally, having proved *progPostWP*, we can obtain a proof of the desired postcondition by using *sufficient*, which is a proof that the preconditions computed by *predTrans* are *sufficient* for proving the given postcondition for the given program.

```
progPost : ∀ g i → ProgPost i (runRWS (prog g) i)
progPost g i = sufficient (prog g) (ProgPost i) i (progPostWP g i)
```

3 Framework for PTS

In this section, we describe our generic framework for modeling effectful computations and reasoning about them with predicate transformer semantics. We define a generalized algebraic datatype (GADT) for the ASTs of effectful programs, parameterized by a collection of operations supplied by the user. To set up the framework for a particular set of effects, the user provides an operational and predicate transformer semantics (PTS) for the operations, then proves that these two semantics agree; this enables verifying a postcondition by proving the sufficient precondition generated by the PTS. It is at the point of specifying the predicate transformer semantics that one may tailor the computed proof obligations to suit one's needs, for example, introducing aliasing or reducing the goal to some set of subgoals; proof obligations can be rephrased in any convenient way, provided the two semantics can be shown to agree.

```
record ASTOps : Set where
  field
    Cmd    : (A : Set) → Set
    SubArg : {A : Set} (c : Cmd A) → Set
    SubRet : {A : Set} {c : Cmd A} (r : SubArg c) → Set
open ASTOps
data AST (OP : ASTOps) : Set → Set where
  ASTreturn : ∀ {A} → A                                          → AST OP A
  ASTbind   : ∀ {A B} → AST OP A → (A → AST OP B)                → AST OP B
  ASTop     : ∀ {A} → (c : Cmd OP A)
              → (f : (r : SubArg OP c) → AST OP (SubRet OP r))
                                                                 → AST OP A
```

Fig. 2: Datatype for effectful code

3.1 *AST*

Figure 2 shows the definition of *AST*, the GADT for effectful program ASTs (in Agda, *Set* is a classifier for types, and the sort *Set* → *Set* is for type constructors; to improve readability, our Agda code listings omit universe levels [14]). These definitions are in the *Dijkstra.AST.Core* module. We make the monadic *unit* operation (*ASTreturn*) a constructor and, deviating from the recipe of Hancock and Setzer [6], we also make the *bind* operation (*ASTbind*) a constructor. In Section 3.2, we will see that, by making the bind operator a constructor, we avoid the need for an additional lemma to assign a predicate transformer to composite computations (such as needed by Swierstra and Baanen [13], §4). The last constructor, *ASTop*, enables us to describe effectful operations.

AST is parameterized by a value of type *ASTOps*, which comprises three fields that describe the syntax of effectful operations. First, *Cmd* is the family of types for commands, indexed by a type *A* for the result value of the command. Next, *SubArg* is the family of argument types for the subcomputations (if any) of a given command. Finally, *SubRet* is the family of types for values returned by subcomputations (these need not be the same as the given type *A* for the result type of the whole command). Curly braces around bound type variables indicate that we wish Agda to infer instantiations of that type argument.

The *ASTop* constructor of *AST* takes as arguments a command *c* and a (*r* : *SubArg OP c*)-indexed family *f* of subcomputations of the command, each producing values of type *SubRet OP r*, where the type of values produced by the whole operation is *A*. As *AST* has an explicit sequencing constructor (*ASTbind*), we understand *f* not as a continuation for the next computation, but a family of computations subordinate to the command itself (see Example 1). These changes give us the flexibility to encode complex operations as primitive nodes of the AST, enabling us to assign bespoke predicate transformers to them when we reason about the behaviors of the programs in which they occur.

```
record ASTTypes : Set where
  field
    Input : Set
    Output : (A : Set) → Set
  Exec : Set → Set
  Exec A = Input → Output A
```

```
record ASTOpSem (OP : ASTOps)
                (Ty : ASTTypes) : Set where
  open ASTTypes Ty
  field
    runAST : ∀ {A} → AST OP A
                   → Exec A
```

Fig. 3: Operational semantics for an *AST*

```
data RWSCmd (A : Set) : Set where
  RWSgets : (g : St → A) → RWSCmd A
  RWSpass : RWSCmd A
  ...
RWSSubArg : {A : Set} (c : RWSCmd A) → Set
RWSSubArg (RWSgets g) = Void
RWSSubArg RWSpass     = Unit
...
RWSSubRet : {A : Set} {c : RWSCmd A} (r : RWSSubArg c) → Set
RWSSubRet { _ } {RWSgets g} ()
RWSSubRet {A} {RWSpass}   _ = A × (List Wr → List Wr)
...
RWSOps : ASTOps
Cmd RWSOps    = RWSCmd
SubArg RWSOps = RWSSubArg
SubRet RWSOps = RWSSubRet
```

Fig. 4: Commands and operational types for *RWS*

Operational Semantics Figure 3 shows what is required for *running AST* programs. First, the user must supply the *Input* type and *Output* type family (type argument *A* is the type of values returned by the program). The type of *executions* of a program producing values of type *A* is a function *Exec A* that maps an *Input* to an *Output A*. The user then provides an instance of *ASTOpSem* for the desired operations and types. The only field of *ASTOpSem* is *runAST*, a function that transforms the AST of an effectful computation producing values of type *A* to a function of type *Exec A*.

Example 1. Figure 4 sketches the instantiation of *ASTOps* for modeling *RWS* programs. Due to space constraints, we only show two operations and omit the (straightforward) definition of the operational semantics; for the full definition, see module *Dijkstra.AST.RWS*. *RWS* operations *gets* : ∀ {*A*} → (*St* → *A*) → *A* and *pass* : ∀ {*A*} → *RWS* (*A* × (*List Wr* → *List Wr*)) → *RWS A* are modeled with constructors *RWSgets* and *RWSpass*. *RWSgets* has no *RWS* subcomputations, so the arity (given by *RWSubArg* (*RWSgets g*)) is

```
record ASTTypes : Set where
  ...
  Pre          = Input → Set
  Post A       = Output A → Set
  PredTrans A  = Post A → Pre
record ASTPredTrans (OP : ASTOps) (Ty : ASTTypes) : Set where
  open ... -- opens to avoid explicit references are omitted hereafter
  field
    returnPT : ∀ {A} → A                  → PredTrans A
    bindPT   : ∀ {A B} → (A → PredTrans B) → Input
               → Post B                   → Post A
    opPT     : ∀ {A} → (c : Cmd OP A)
               → ((r : SubArg OP c) → PredTrans (SubRet OP r))
                                          → PredTrans A
  predTrans : ∀ {A} → AST OP A      → PredTrans A
  predTrans (ASTreturn x) P i  = returnPT x P i
  predTrans (ASTbind m f) P i  = predTrans m (bindPT (predTrans ∘ f) i P) i
  predTrans (ASTop c f) P i    = opPT c (predTrans ∘ f) P i
```

Fig. 5: Predicate transformer semantics for an *AST*

Void (the empty type). *RWSpass* has a single subcomputation (so the arity is *Unit*); the type it returns is $A \times (List\ Wr \to List\ Wr)$, where A is the type of the entire *RWSpass* computation. The *ASTTypes* instance for *RWS* (not shown) sets the fields to the definitions of *Input* and *Output* given in Section 2.

3.2 Predicate Transformer Semantics

We can now define what it means to give a PTS to an *AST*. It is at *this step* where the proof engineer instantiating the framework decides what proof obligations are generated for effectful operations and sequencing; we will see this more concretely in Example 2. The step that follows, described in Section 3.3, requires showing that PTS *agree* with the operational semantics.

In Figure 5, we continue the listing of the record *ASTTypes*, which includes some definitions for convenience: *Pre* and *Post* are the definitions of preconditions and postconditions, and *PredTrans* defines a predicate transformer as a function that produces preconditions from postconditions. A user wishing to assign a PTS to a particular choice of AST operations *OP* and input and output types *Ty* provides three items, expressed as the three fields of *ASTPredTrans*.

Each field of *ASTPredTrans* corresponds to a clause of the definition of *predTrans*, the function that assigns a predicate transformer to *AST OP* programs. Field *returnPT* is a family of predicate transformers for *ASTreturn*. Field *bindPT* is for composite operations of the form $ASTbind\ m\ f : AST\ OP\ B$. Its purpose is to take a postcondition $P : Post\ B$ for the entire computation and rephrase it as a postcondition (of type $Post\ A$) for $m : AST\ OP\ A$. The

```
RWSbindPost : (outs : List Wr) {A : Set} → Post A → Post A
RWSbindPost outs P (x , st , outs′) = P (x , st , outs ++ outs′)

RWSpassPost : ∀ {A} → Post A → Post (A × (List Wr → List Wr))
RWSpassPost P ((x , f) , s , o) = ∀ o′ → o′ ≡ f o → P (x , s , o′)

RWSPT : ASTPredTrans RWSOps RWSTypes
returnPT RWSPT x P (e , s) = P (x , st, [])
bindPT   RWSPT f (e , s₀) P (x , s₁ , o) =
  ∀ r → r ≡ x → f r (RWSbindPost o P) (e , s₁)
opPT     RWSPT (RWSgets g)   f P (e , s) = P (g s , s , [])
opPT     RWSPT {A} RWSpass f P (e , s) = f unit (RWSpassPost P) (e , s)
```

Fig. 6: Predicate transformer semantics for *RWS*

definition of *bindPT* supplied by the user should use the assumed family of predicate transformers (the argument of type $A \to PredTrans\ B$) to express a sufficient precondition of f to prove P, and then make that precondition the *postcondition* for m (see Example 2). Field *opPT* gives a predicate transformer for every command $c : Cmd\ OP\ A$, given a family of predicate transformers for the subcomputations (if any) given as arguments to that command.

Given these pieces, the function *predTrans* that assigns a predicate transformer to every $m : AST\ OP\ A$ is defined by induction over m. We again call attention to the fact that, because our *AST* type has the constructor *ASTbind*, users can tailor a convenient predicate transformer to assign to composite operations directly, rather than requiring an explicit compositionality lemma.

Example 2. Figure 6 shows the instantiation of *ASTPredTrans* (omitting operations other than *RWSgets* and *RWSpass*). For *returnPT*, the precondition we return is that the postcondition P holds for the returned value, the current state, and an empty list of messages (there are no state changes or messages emitted). The *post*condition we return for *bindPT* is trickier: P is the postcondition we wish to hold for $ASTbind\ m_1\ m_2$, and what we return is the postcondition that should hold of m_1 to establish this. Because x (the result of executing m_1) may be instantiated to an unwieldy expression, we alias x as r and give this to the predicate transformer f that is assigned to m_2. We also have that m_1 emitted o as output, so we use *RWSbindPost* to express that the postcondition should hold for the result of appending these to the emitted messages of m_2.

We treat *RWSgets* similarly to *returnPT*: we require the postcondition to hold of $g\ s$, where g is the user-supplied getter function. For *RWSpass*, we apply the predicate transformer f assigned to the subcomputation (think *inner* from Section 2) to $RWSpassPost\ P$, which says that postcondition P holds when we modify the output of the subcomputation with the returned function $h : List\ Wr \to List\ Wr$.

3.3 Agreement of Semantics

In Sections 3.1 and 3.2, we described how to assign operational and predicate transformer semantics to a set of effectful operations. We now describe how to show that two such semantics *agree*. The obligations to show one direction of this agreement are formalized by *ASTSufficientPT*, shown in Figure 7.

```
record ASTSufficientPT {OP : ASTOps} {Ty : ASTTypes}
  (OpSem : ASTOpSem OP Ty) (PT : ASTPredTrans OP Ty) : Set where
  Sufficient : (A : Set) (m : AST OP A) → Set
  Sufficient A m = ∀ P i → (wp : predTrans m P i) → P (runAST m i)
  field
    returnSuf : ∀ {A} x → Sufficient A (ASTreturn x)
    bindSuf   : ∀ {A B} (m : AST OP A) (f : A → AST OP B)
                  → Sufficient A m → (∀ x → Sufficient B (f x))
                  → Sufficient B (ASTbind m f)
    opSuf     : ∀ {A} → (c : Cmd OP A) (f : SubArg OP c → AST OP (SubRet OP c))
                        → (∀ r → Sufficient (SubRet OP c) (f r))
                        → Sufficient A (ASTop c f)
  sufficient : ∀ {A} → (m : AST OP A) → Sufficient A m
  sufficient = ...
```

Fig. 7: Sufficiency lemmas for operational semantics and PTS

Sufficient says that, for a given effectful program $m : AST\ OP\ A$, the predicate transformer $predTrans\ m$ returns, for every postcondition P, a precondition *sufficient* for proving that, for any input i, P is true of the result obtained from running m with input i using the operational semantics; henceforth we abbreviate this and say that *the predicate transformer for m is sufficient*. To prove sufficiency, our framework imposes three obligations on the user, corresponding to the three constructors of *AST*; they are as follows.

1. Instantiating the field *returnSuf* requires that the predicate transformer corresponding to $ASTreturn\ x$ (for any $x : A$) is sufficient.
2. For field *bindSuf*, the user assumes that the predicate transformers assigned to m and (all instances of) f are sufficient, and must prove that the predicate transformer obtained from $ASTbind\ m\ f$ is sufficient.
3. Finally, for field *opSuf*, assuming that, for an arbitrary command c and subcomputation f, the predicate transformer obtained from the result of running f with any possible response value is sufficient, the user must show that the predicate transformer for $ASTop\ c\ f$ is sufficient.

With these three obligations met, the proof of sufficiency (*sufficient*) proceeds by a straightforward induction.

```
data BranchCmd (A : Set) : Set where
  BCif      : Bool                    → BranchCmd A
  BCeither  : ∀ {B C} → Either B C    → BranchCmd A
  BCmaybe   : ∀ {B}   → Maybe B       → BranchCmd A

BranchSubArg : ∀ {A} → BranchCmd A → Set
BranchSubArg (BCif x)          = Bool
BranchSubArg (BCeither {B} {C} x) = Either B C
BranchSubArg (BCmaybe {B} x)   = Maybe B

BranchSubRet : ∀ {A} {c : BranchCmd A} → BranchSubArg c → Set
BranchSubRet {A} _ = A

module ASTExtension (O : ASTOps) where
  BranchOps : ASTOps
  Cmd BranchOps A                  = Either (Cmd O A) (BranchCmd A)
  SubArg BranchOps (left x)        = SubArg O x
  SubArg BranchOps (right y)       = BranchSubArg y
  SubRet BranchOps {_} {left x} r  = SubRet O r
  SubRet BranchOps {_} {right y} r = BranchSubRet r

  unextend : {A} → AST BranchOps A → AST O A
  unextend = ...
```

Fig. 8: Extending *ASTOps* with branching commands

The other direction of agreement is captured by *Necessary* (not shown), which says that, for an effectful program $m : AST\ OP\ A$, for every postcondition P and input i, if the output achieved by running m on i satisfies P, then i satisfies the precondition returned by $predTrans\ m\ P$. Similar to *Sufficient*, the framework imposes an obligation on the user for each constructor of *AST*, and uses these to prove *Necessary*; details can be found in module *Dijstra.AST.Core*.

4 Generic Branching Operations

Branching operations may be used *in* effectful code, but do not *themselves* have effects. Therefore, our framework can *generically* extend any set of commands and their predicate transformer semantics to include a few common branching operations (see module *Dijkstra.AST.Core*), re-expressing their proof obligations to avoid the issues outlined in Section 2.

4.1 Branching Commands

Figure 8 shows how we model these branching commands, similar to how we model effectful commands (Section 3.1). Datatype *BranchCmd* enumerates the branching commands (see module *Dijkstra.AST.Branching*); so far, we support commands for Booleans (*BCif*), coproducts (*BCeither*), and the *Maybe* type

```
module OpSemExtension
  {O : ASTOps} {T : ASTTypes} (OpSem : ASTOpSem O T) where
  BranchOpSem : ASTOpSem BranchOps T
  runAST BranchOpSem m i = runAST OpSem (unextend m) i
module PredTransExtension
  {O : ASTOps} {T : ASTTypes} (PT : ASTPredTrans O T) where
  BranchPT : ASTPredTrans BranchOps T
  returnPT  BranchPT          = returnPT PT
  bindPT    BranchPT          = bindPT   PT
  opPT      BranchPT (left x) = opPT     PT x
  opPT      BranchPT (right (BCif c)) f P i =
    (c ≡ true  → f true  P i) × (c ≡ false → f false P i)
  opPT      BranchPT (right (BCeither e)) f P i =
              (∀ l → e ≡ left  l → f (left  l) P i)
            × (∀ r → e ≡ right r → f (right r) P i)
  opPT      BranchPT (right (BCmaybe mb)) f P i =
              (∀ j → mb ≡ just j    → f (just j) P i)
            × (      mb ≡ nothing → f nothing  P i)
```

Fig. 9: Operational and predicate transformer semantics for branching commands

(*BCmaybe*). Each constructor of *BranchCmd* takes the scrutinee (the subject of case analysis) for the operation it models. For *BranchSubArg*, the arity of the family of subcomputations for a branching command is given by the type of the scrutinee (e.g., there are two subcomputations for *if*, so its arity is given by the two-element type *Bool*). Finally, for *BranchSubRet*, the type of result values for the subcomputations is *A*, the type of result values for the entire computation.

BranchOps extends a given set of operations *O* : *ASTOps* with the branching commands just described. We take the set of command codes to the disjoint union of the command codes of *O* and *BranchOps*, and extend the *SubArg* and *SubRet* fields accordingly. We also define the function *unextend* to traverse a program AST and remove all branching commands; this is used in the developments discussed next for extending existing operational and predicate transformer semantics to the branching operations.

Semantics In Figure 9, we assign operational and predicate transformer semantics to *BranchOps*. The operational semantics for the extended set of commands reduces to the operational semantics *OpSem* : *ASTOpSem O T* for the base set of commands via *unextend*, as shown by the definition of *BranchOpSem*. For the predicate transformer semantics, the precondition returned for the given postcondition *P* is expressed as a product of properties, where each component of the product corresponds to a particular branch taken. For example, for the case of *BCeither e* (where *e* : *Either B C*), the first component of the product is for the case in which *e* is of the form *left l* for some *l* : *B*. Recall from Section 2 that

```
record ASTTypes : Set where
  ...
  _⊆ₒ_ : ∀ {A} → (P₁ P₂ : Post A) → Set
  P₁ ⊆ₒ P₂ = ∀ o → P₁ o → P₂ o
  _⊑_ : ∀ {A} → (pt₁ pt₂ : PredTrans A) → Set
  pt₁ ⊑ pt₂ = ∀ P i → pt₁ P i → pt₂ P i
  MonoPT : ∀ {A} → PredTrans A → Set
  MonoPT pt = ∀ P₁ P₂ → P₁ ⊆ₒ P₂ → ∀ i → pt P₁ i → pt P₂ i
record ASTPredTransMono {OP} {Ty} (PT : ASTPredTrans OP Ty) : Set where
  field
    returnPTMono : ∀ {A} (x : A) → MonoPT (returnPT x)
    bindPTMono   : ∀ {A B} (f₁ f₂ : A → PredTrans B)
      → (∀ x → MonoPT (f₁ x)) → (∀ x → MonoPT (f₂ x))
      → (∀ x → f₁ x ⊑ f₂ x)
      → ∀ P₁ P₂ i → P₁ ⊆ₒ P₂ → bindPT f₁ i P₁ ⊆ₒ bindPT f₂ i P₂
    opPTMono     : ∀ {A} (c : Cmd OP A)
                     (f₁ f₂ : (r : SubArg OP c) → PredTrans (SubRet OP r))
      → (∀ r → MonoPT (f₁ r)) → (∀ r → MonoPT (f₂ r))
      → (∀ r → f₁ r ⊑ f₂ r)
      → ∀ P₁ P₂ i → P₁ ⊆ₒ P₂ → opPT c f₁ P₁ i → opPT c f₂ P₂ i
  predTransMono : ∀ {A} (m : AST OP A) → MonoPT (predTrans m)
  predTransMono = ...
```

Fig. 10: Monotonicity of PTS

expressing the proof obligation in this way means that: the proof state is often much more comprehensible; the proof does not need to recapitulate (for example, using **with**) the case distinction performed by the code being verified; and the immediate next step in the proof effort is entirely type directed (copattern matching generates the two subobligations).

4.2 Semantic Agreement for Branching Operations

So far, we have augmented an arbitrary set of effectful operations with branching commands and extended the operational and predicate transformer semantics accordingly. In this section, we describe our result showing that, if the original operational and predicate transformer semantics agree—and furthermore the PTS satisfies a certain monotonicity property, then the two extended semantics for branching commands agree. This ensures that the extension can be used to verify effectful code with branching. Furthermore, the monotonicity property is valuable in its own right when it is easier to prove that the precondition for a postcondition that is stronger than the one required in a given context holds.

Monotonicity properties Figure 10 gives a partial listing of *ASTPredTransMono*, the record that describes the monotonicity lemma required by our framework, as

```
module SufficientExtension
  {O} {T} {OS : ASTOpSem O T} {PT : ASTPredTrans O T}
  (M : ASTPredTransMono PT) (S : ASTSufficientPT OS PT) where
  BranchPTMono : ASTPredTransMono BranchPT
  BranchPTMono = ...
  unextendPT : ∀ {A} (m : AST BranchOps A)
    → predTrans BranchPT m ⊑ predTrans PT (unextend m)
  unextendPT = ...
  extendPT : ∀ {A} (m : AST BranchOps A)
    → predTrans PT (unextend m) ⊑ predTrans BranchPT m
  extendPT = ...
  BranchSuf : ASTSufficientPT BranchOpSem BranchPT
  BranchSuf = ...
```

Fig. 11: Sufficiency of PTS for branching operations

well as some further definitions within the scope of the *ASTTypes* record, which we now describe. The type $P_1 \subseteq_o P_2$ expresses entailment of postcondition P_2 by P_1 on all outputs (that is, that P_1 is at least as strong a postcondition as P_2). For two predicate transformers f_1 and f_2, $f_1 \sqsubseteq f_2$ is read as saying that f_2 is a *refinement* of f_1 because, for the same postcondition P, we have that f_2 produces a precondition no stronger than that of f_1 (as long as both preconditions are sufficient for proving P holds of some program, a weaker precondition is preferable because it is more general). Finally, monotonicity of a predicate transformer is expressed by *MonoPT*, where *MonoPT f* says that f sends stronger postconditions to stronger preconditions.

Next, we consider the types of the fields *returnPTMono*, *bindPTMono*, and *opPTMono* of record *ASTPredTransMono*, which the user must provide to enable proving that the predicate transformer for any effectful computation m : *AST Op A* is monotonic. The first, *returnPTMono*, requires that the predicate transformers in the family assigned to *ASTreturn* are monotonic. The monotonicity property for composite computations (*bindPTMono*) says that we can map both refinement of (families of) predicate transformers ($\forall\ x \to f_1\ x \sqsubseteq f_2\ x$) and entailment of postconditions ($P_1 \subseteq_o P_2$) over *bindPT*. Similarly, for operations, *opPTMono* says that we can map predicate transformer refinement and postcondition entailment over the function *opPT* that assigns a predicate transformer to every command c : *Cmd Op A*.

Agreement for the extended PTS Figure 11 shows a sketch of the proof that the extended PTS produces, from any program m : *AST BranchOps A* and postcondition P : *Post A*, a precondition sufficient for proving that P holds of the result of running m with the operational semantics (*BranchSuf* in the figure). We prove a similar result for *Necessary* (not shown; see module *Dijkstra.AST.Branching*). The lemmas *unextendPT* and *extendPT* are the workhorses of these proofs:

taken together, they state that, for every such effectful program m, the predicate transformer obtained from *BranchPT* (Figure 9) is equivalent to the one obtained from invoking the original predicate transformer *PT* on the result of using *unextend* to traverse the AST of m and remove branching commands.

5 Conclusion and Related Work

We have presented an Agda framework for modeling effectful programs and reasoning about them with predicate transformer semantics. Our framework gives users greater flexibility in expressing intermediate proof obligations by using a novel (to our knowledge) GADT definition of program ASTs that enables assigning bespoke predicate transformers to complex operations. Demonstrating the framework's generality, we show that the common branching operations *if*, *maybe*, and *either*, can be added *á la carte* (in the sense of Swierstra [12]) to any existing set of effects, with the PTS extended to cover the new operations provided that the original PTS satisfies a monotonicity condition.

Our framework codifies and generalizes techniques used in our work [4] formally verifying properties of the LibraBFT consensus protocol; we have used it to prove some properties in that context, and this has confirmed that the framework is usable for and can significantly ease real-world verification tasks.

Our work is most closely related to that of Swierstra and Baanen [13] on assigning predicate transformer semantics to effectful programs. Like us, they represent these programs with a datatype—in their case, a free monad [6]—and assign to it operational and predicate transformer semantics, proving agreement of these semantics in order to carry out verification tasks. Our approach differs from theirs by making the datatype of program ASTs a GADT that has the monadic bind operation as a constructor, enabling us to avoid the use of a lemma for decomposing proof obligations found in [13] §4 for composite computations, and an expanded notion of effectful command that enables complex operations to be assigned bespoke predicate transformer semantics directly. These differences give greater control to users of the framework in managing the intermediate proof obligations generated during verification tasks.

PTS has also been used profitably with *Dijkstra monads* [11, 7] which leverage the monadic nature of predicate transformers (specifically, they arise from a *Set*-valued continuation monad transformer [8]) to combine effectful code with a formal specification. The dependently typed programming language F* supports Dijkstra monads [10] and integrates these with SMT solvers for automatic reasoning about effectful code. The main advantage of our GADT-based approach (and of the free monad approach of [13]) is *freedom of interpretation:* the *AST* datatype describes only the syntax of programs, meaning multiple operational and predicate transformer semantics may be assigned to the same program.

Acknowledgements: We are grateful to Victor Miraldo and Lisandra Silva for valuable discussions and feedback on earlier versions of this paper.

References

1. Ahman, D., Hritcu, C., Maillard, K., Martínez, G., Plotkin, G.D., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 515–529. ACM (2017). https://doi.org/10.1145/3009837.3009878
2. Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the Libra blockchain (6 2019), developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2019-06-28.pdf
3. BFT consensus in Agda (April 2022), github.com/oracle/bft-consensus-agda/releases/tag/sefm22-submission
4. Carr, H., Jenkins, C., Moir, M., Miraldo, V.C., Silva, L.: Towards formal verification of HotStuff-based byzantine fault tolerant consensus in Agda. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods. pp. 616–635. Springer International Publishing, Cham (2022)
5. Gill, A.: The RWS monad. https://hackage.haskell.org/package/transformers-0.6.0.4/docs/Control-Monad-Trans-RWS-Lazy.html (2022)
6. Hancock, P.G., Setzer, A.: Interactive programs in dependent type theory. In: Clote, P., Schwichtenberg, H. (eds.) Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1862, pp. 317–331. Springer (2000). https://doi.org/10.1007/3-540-44622-2_21
7. Jacobs, B.: Dijkstra and hoare monads in monadic computation. Theor. Comput. Sci. **604**(C), 3045 (nov 2015). https://doi.org/10.1016/j.tcs.2015.03.020, https://doi.org/10.1016/j.tcs.2015.03.020
8. Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., Tanter, É.: Dijkstra monads for all. Proc. ACM Program. Lang. **3**(ICFP), 104:1–104:29 (2019). https://doi.org/10.1145/3341708
9. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics). Elsevier Science Inc. (2006)
10. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in f*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 256270. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837655, https://doi.org/10.1145/2837614.2837655
11. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 387–398. ACM (2013). https://doi.org/10.1145/2491956.2491978
12. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758, https://doi.org/10.1017/S0956796808006758
13. Swierstra, W., Baanen, T.: A predicate transformer semantics for effects (functional pearl). Proc. ACM Program. Lang. **3**(ICFP), 103:1–103:26 (2019). https://doi.org/10.1145/3341707

14. The Agda Team: The Agda user manual: Language Reference: Universe Levels (2022), https://agda.readthedocs.io/en/latest/language/universe-levels.html