Embedding the Refinement Calculus in Coq

João Alpuim

University of Hong Kong

Wouter Swierstra

Universiteit Utrecht

Abstract

The *refinement calculus* and *type theory* are both frameworks that support the specification and verification of programs. This paper presents an embedding of the refinement calculus in the interactive theorem prover Coq, clarifying the relation between the two. As a result, refinement calculations can be performed in Coq, enabling the interactive calculation of formally verified programs from their specification.

1. Introduction

The idea of *deriving* a program from its specification can be traced back to Dijkstra (1976), Floyd (1967) and Hoare (1969). The *refinement calculus* (Back, 1978; Morgan, 1990; Back and Wright, 1998) defines a formal methodology that can be used to construct a derivation of a program from its specification step by step. Crucially, the refinement calculus presents single language for describing both programs and specifications.

Deriving complex programs using the refinement calculus is no easy task. The proofs and obligations can quickly become too complex to manage by hand. Once you have completed a derivation, the derived program must still be transcribed to a programming language in order to execute it – a process which can be rather error-prone (Morgan, 1990, Chapter 19).

To address both these issues, we show how the refinement calculus can be embedded in Coq, an interactive proof assistant based on dependent types. Although others have proposed similar formalizations of the refinement calculus (Back and von Wright, 1990; Hancock and Hyvernat, 2006), this paper presents the following novel contributions:

- After giving a brief overview of the refinement calculus (Section 2), we begin by developing a library of predicate transformers in Coq, based on

Email addresses: `alpuim@cs.hku.hk` (João Alpuim), `w.s.swierstra@uu.nl` (Wouter Swierstra)

indexed containers (Altenkirch and Morris, 2009; Hancock and Hyvernat, 2006), making extensive use of dependent types (Section 3). We define a *refinement relation*, corresponding to a morphism between indexed containers, enabling us to prove several simple refinement laws in Coq.

- Next we show how to embed effects, such as mutable state and general recursion, in Coq using a free monad. We assign semantics to programs in this monad using the predicate transformers and refinement relation we described previously (Section 4).
- These definitions give us the basic building blocks for formalizing derivations in the refinement calculus. They do, however, require that the derived program is known *a priori*. We address this and other usability issues (Section 5).
- Finally, we validate our results by performing a small case study [1]. In particular, we show how we can use this library to calculate interactively the key ingredients of a data structure for *persistent arrays* (Section 6).

2. Refinement calculus

The refinement calculus, as presented by Morgan (1990), extends Dijkstra's Guarded Command language with a new language construct for specifications. The specification $[\mathsf{pre}, \mathsf{post}]$ is satisfied by a program that, when supplied an initial state satisfying the precondition pre, can be executed to produce a final state satisfying the postcondition post. Crucially, this language construct may be mixed freely with (executable) code constructs.

Besides these specifications, the refinement calculus defines a *refinement relation* between programs, denoted by $p_1 \sqsubseteq p_2$. This relation holds when $\textbf{forall } P, \mathsf{wp}(p_1, P) \Rightarrow \mathsf{wp}(p_2, P)$, where wp denotes the usual weakest precondition semantics of a program and its desired postcondition. Intuitively, you may want to read $p_1 \sqsubseteq p_2$ as stating that p_2 is a 'more precise specification' than p_1.

A program is said to be *executable* when it is free of specifications and only consists of executable statements. Morgan (1990) refers to such executable programs as *code*. To calculate an executable program C from its specification S, you must find a series of refinement steps, $S \sqsubseteq M_0 \sqsubseteq M_1 \sqsubseteq ... \sqsubseteq C$. Typically, the intermediate programs, such as M_0 and M_1, mix executable code fragments and specifications.

To find such derivations, Morgan (1990) presents a catalog of lemmas that can be used to refine a specification to an executable program. Some of these lemmas define when it is possible to refine a specification to code constructs. These lemmas effectively describe the semantics of such constructs. For example, the following law may be associated with the **skip** command:

[1] All the code and examples presented in this paper can be found online at `https://github.com/jalpuim/dtp-refinement`.

$$
\begin{array}{ll}
& [x = X \wedge y = Y, x = Y \wedge y = X] \\
\sqsubseteq & \quad \{ \text{ by the following assignment law } \} \\
& [x = X \wedge y = Y, t = Y \wedge y = X]; x ::= t \\
\sqsubseteq & \quad \{ \text{ by the following assignment law } \} \\
& [x = X \wedge y = Y, t = Y \wedge x = X]; y ::= x; x ::= t \\
\sqsubseteq & \quad \{ \text{ by the following assignment law } \} \\
& [x = X \wedge y = Y, y = Y \wedge x = X]; t ::= y; y ::= x; x ::= t \\
\sqsubseteq & \quad \{ \text{ by the law for skip } \} \\
& \textbf{skip}; t ::= y; y ::= x; x ::= t
\end{array}
$$

Figure 1: Derivation of the swap program

Lemma 1 (skip). *If* $\mathsf{pre} \Rightarrow \mathsf{post}$*, then* $[\mathsf{pre}, \mathsf{post}] \sqsubseteq \textbf{skip}$.

Besides such primitive laws, there are many recurring patterns that pop up during refinement calculations. For example, combining the rules for sequential composition and assignment, the *following assignment* lemma holds:

Lemma 2 (following assignment). *For any term* E*,*

$$[\mathsf{pre}, \mathsf{post}] \sqsubseteq [\mathsf{pre}, \mathsf{post}\ [w \backslash E]]; w ::= E$$

We illustrate how these rules may be used to calculate the definition of a program from its specification. Suppose we would like to swap the values of two variables, x and y. We may begin by formulating the specification of our problem as:

$$[x = X \wedge y = Y, x = Y \wedge y = X]$$

Using the two lemmas we saw above, we can refine this specification to an executable program. The corresponding calculation is given in Figure 1. Note that we have chosen to give a simple derivation that contains some redundancy, such as the final **skip** statement, but uses a modest number of auxiliary lemmas and definitions.

For such small programs, these derivations are manageable by hand. For larger or more complex derivations, it can be useful to employ a computer to verify the correctness of the derivation and even assist in its construction. In the coming sections we will develop a Coq library for precisely that.

3. Predicate transformers

In this section, we will assume there is some type S, representing the state that our programs manipulate. In Section 4 we will show how this can be instantiated with a (model of a) heap. For now, however, the definitions of specifications, refinement, and predicate transformers will be made independently of the choice of state.

We begin by defining a few basic constructions in Coq:

```
Definition Pred (A : Type) : Type := A → Type.
```

This defines the type Pred A of predicates over some type A. Using this definition we can define a subset relation between predicates as follows:

```
Definition subset (A : Type) (P₁ P₂ : Pred A) := forall x, P₁ x → P₂ x.
```

A predicate P_1 is a subset of the predicate P_2, if any state satisfying P_1 also satisfies P_2. In the remainder of this paper, we will write $P_1 \subseteq P_2$ when the property subset P_1 P_2 holds.

Next we can define the PT data type, consisting of a precondition and postcondition:

```
Record PT (A : Type) : Type :=
  MkPT {pre : Pred S;
        post : forall s : S, pre s → A → Pred S}.
```

The postcondition is a relation between the input state, a proof that this input state satisfies the precondition, the value of type A returned by the computation, and the output state. To avoid the need for 'ghost variables', we allow this relation to refer to both the input and output states (Nanevski et al., 2008b; Swierstra, 2009a,b). As this data type will be used to represent specifications, we will use the notation $[P, Q]$ rather than the more verbose MkPT P Q.

As its name suggests, the PT type has an obvious interpretation as a *predicate transformer*, i.e., a function mapping predicates to predicates:

```
Definition semantics {A : Type} (pt : PT A) : Pred (A * S) → Pred S :=
  fun P s ⇒ {p : pre pt s & forall s' v, post pt s p v s' → P (v, s')}.
```

The semantics function computes the weakest precondition necessary to guarantee that the desired postcondition P holds after executing a program satisfying the given specification pt. Intuitively, the precondition of the specification must hold and the postcondition must imply P. We will sometimes write $[\![pt]\!]$ rather than semantics pt for the sake of brevity. In what follows, we will sometimes leave out implicit argument, such as the argument $\{A : \text{Type}\}$ in the semantics function, from the typeset code presented here.

Next, we characterize the refinement relation between two values of type PT as follows:

```
Inductive Refines (pt₁ pt₂ : PT A) : Type :=
  Refinement : forall (d : pre pt₁ ⊆ pre pt₂),
    (forall s p v, post pt₂ s (d s p) v ⊆ post pt₁ s p v) →
    Refines pt₁ pt₂.
```

We consider pt_2 to be a refinement of pt_1 when the precondition of pt_1 implies the precondition of pt_2 and the postcondition of pt_2 implies the postcondition of pt_1. As our postconditions are relations, we need to do some work to describe the latter condition. In particular, we need to transform the assumption that the initial state holds for the precondition of pt_1 to produce a proof that the precondition of pt_2 also holds for the same initial state. To do so, we use the first condition, d, that the precondition of pt_1 implies the precondition of pt_2. We will use the notation, $pt_1 \sqsubseteq pt_2$, for the proposition $\mathsf{Refines}\ pt_1\ pt_2$.

To validate the correctness of this definition, we will show that it satisfies the characterization of refinement in terms of weakest precondition semantics given in Section 2. To do so, we have proven the following soundness result:

$$\begin{aligned}&\textbf{Theorem}\ \mathsf{soundness}\ : \textbf{forall}\ (pt_1\ pt_2 : \mathsf{PT}\ A),\\ &\quad pt_1 \sqsubseteq pt_2\ \leftrightarrow\ \textbf{forall}\ P, [\![pt_1]\!]\ P \subseteq [\![pt_2]\!]\ P.\end{aligned}$$

In other words, the Refines relation adheres to the characterization of the refinement relation in terms of predicate transformer semantics. The proof is almost trivial after unfolding the various definitions involved.

Even if we have not yet fixed the state space S, we can already prove that the structural laws of the refinement calculus, such as strengthening of postconditions, hold:

$$\begin{aligned}&\textbf{Lemma}\ \mathsf{strengthen}\ \ (P : \mathsf{Pred}\ S)\ (Q_1\ Q_2 : \textbf{forall}\ s, P\ s \to A \to \mathsf{Pred}\ S)\\ &\quad (H : \textbf{forall}\ (s : S)\ (v : A)\ (p : P\ s), Q_1\ s\ p\ v \subseteq Q_2\ s\ p\ v) :\\ &\quad \mathsf{Refines}\ ([P, Q_2])\ ([P, Q_1]).\end{aligned}$$

To prove this lemma, we need to show that $P \subseteq P$ and that the postcondition Q_1 implies Q_2. The first proof is trivial; the second follows immediately from our hypothesis. Similarly, we can show that the refinement relation is both transitive and reflexive.

These definitions by themselves are not very useful. Before we can perform any *program* derivation, we first need to fix our *programming language.*

4. Embedding in Coq

In our previous work (Swierstra and Alpuim, 2016), we showed how to define a *deep embedding* of small, imperative programming language. By defining predicate transformers associated with the various syntactic constructs, we could then calculate a program from its specification. Such a deep embedding of the language allows us to inspect the abstract syntax of our object language, enabling us to generate programs calculated from their specifications. This approach does have its drawbacks. In particular, the imperative programming language was too restrictive to cover many, more interesting examples. Furthermore, the mutable references were only allowed to store integers. In this extended paper, we address these issues.

To begin with, we fix our choice of state S to be a finite map from an abstract type Ptr to some type of values v. All our developments are parametrised over the particular choice of type to store on the heap. Although we use the finite map modules from Coq's standard library to model the heap, several alternative representations exist (Nanevski et al., 2008a; Swierstra and Altenkirch, 2007; Swierstra, 2009b).

Instead of a deep embedding of the imperative language, we will provide a deep embedding only of the *effects* of our object language, namely mutable state and unbounded recursion by defining a suitable *free monad*. We will then piggyback on Coq's programming language, Gallina, to define programs using these effects. We can then customize Coq's extraction to OCaml to map these syntactic constructs to their native OCaml counterparts.

We model the effects as an inductive data type in Coq:

```
Inductive WhileL (a : Type) : Type :=
  | New    : v → (Ptr → WhileL a) → WhileL a
  | Read   : Ptr → (v → WhileL a) → WhileL a
  | Write  : Ptr → v → WhileL a → WhileL a
  | While  : (S → Prop) → (S → bool) → (WhileL unit) →
               WhileL a → WhileL a
  | Spec   : PT a → WhileL a
  | Return : a → WhileL a.
```

This data type has constructors for the creation of (New), access of (Read), and assignment to (Write) mutable references. Each of these constructors takes a continuation as argument, representing the remaining computation to perform. Loops may be introduced using the While constructor that takes four arguments: the loop invariant of type $S \to$ Prop; the condition of type $S \to$ bool; the loop body of type WhileL unit; and the remaining computation. The constructor Spec contains the specification of an unfinished program fragment. The refinement laws we will define shortly determine how such specifications may be refined to executable code. Finally, the Return constructor simply returns a 'pure' Coq value.

Semantics

Before discussing the refinement calculation further, we need to fix the semantics of our language. We shall do so by associating a predicate transformer, i.e., a value of type PT, with every constructor of the WhileL data type.

Each rule in in Figure 2 associates pre- and postconditions, i.e., a value of type PT, with the constructors of the WhileL data type. We use the somewhat suggestive notation, $\{\,P\,\}\ c\ \{\,Q\,\}$ to associate with the statement c the conditions $[P, Q]$. We also use the notation $p \overset{s}{\mapsto} v$ to denote that s maps the pointer p to value v, and $p \overset{s}{\mapsto} \_$ to denote that p has some associated value in s. Finally, the notation $P\ (s\ [p \mapsto v])$ denotes that the condition P should hold after updating the state s, mapping the pointer p to value v.

These rules are not added as axioms to Coq; nor are they the constructors of an inductive data type. Rather, we can assign semantics to our WhileL data type directly, as a recursive function:

$$\frac{}{\{\mathsf{True}\}\ \mathsf{Return}\ y\ \{s = s' \land x = y\}}\ \textsc{Return}$$

$$\frac{p \overset{s}{\mapsto} v \qquad \{P\}\ k\ v\ \{Q\}}{\{P\}\ \mathsf{Read}\ p\ k\ \{Q\}}\ \textsc{Read}$$

$$\frac{p \overset{s}{\mapsto} \_ \qquad \{P\}\ k\ \{Q\}}{\{P\ (s\ [p\ \mapsto\ v])\}\ \mathsf{Write}\ p\ v\ k\ \{Q\ (s\ [p\ \mapsto\ v])\ x\ s'\}}\ \textsc{Write}$$

$$\frac{p \notin dom\ (s) \qquad \{P\}\ k\ p\ \{Q\}}{\{P\ (s\ [p\ \mapsto\ v])\}\ \mathsf{New}\ v\ k\ \{Q\ (s\ [p\ \mapsto\ v])\ x\ s'\}}\ \textsc{New}$$

$$\frac{\{P_1\}\ b\ \{Q_1\} \qquad \textbf{forall}\ s, \neg\ c(s) \land I\ s \to P_2\ s \qquad \{P_2\}\ k\ \{Q_2\}}{\left\{\begin{array}{c} I\ s \land (\forall\ t, c(t) \land I\ t \to P_1\ t) \land \\ \forall\ t\ t', c(t) \land I\ t \land Q_1\ t\ t' \to I\ t' \end{array}\right\}\ \mathsf{While}\ c\ \textbf{do}\ b\ \textbf{od}\ k\ \left\{Q_2\right\}}\ \textsc{While}$$

Figure 2: Semantics of WHILE

Fixpoint semantics $(c : \mathsf{WhileL}\ a) : \mathsf{PT}\ a$

In addition to the rules from Figure 2, this function simply maps specifications, represented by the Spec constructor, to their associated predicate transformer.

Let us examine the rules in Figure 2 a bit more closely. Each precondition may refer to an initial state s; each postcondition is formulated as a relation between an initial state s, satisfying the precondition, the result returned x, and the final state s'. For example, the postcondition of the Return rule states that the initial state s is equal to the final state s' and the result of the computation is indeed the argument of the Return constructor.

The other rules are defined by induction on the program. For example, the rule for Read states that, provided the current state maps the pointer p to the value v, and the remainder of the program has an associated precondition P and postcondition Q, the composite program has the same pre- and postcondition. When the state changes, such as in the rules for Write and New, this style of definition can be a bit confusing. The Write rule, for instance, computes the precondition and postcondition associated with the remaining program k. The composite program Write $p\ v\ k$ then requires that the precondition P associated with k holds *after* updating the state. Similarly, the postcondition Q should relate the state after being updated with the result x and final state s'. The New rule follows exactly the same pattern.

Finally, the WHILE rule is the most complex. Besides the precondition, P_1, and postcondition, Q_1, associated with the body of the loop b, the WHILE rule requires the programmer to specify the loop invariant, I, and continuation k.

The precondition of the WHILE rule consists of three conjuncts:

- the invariant I must hold initially;
- the boolean guard c and the invariant must together imply the precondition of the loop body;
- the loop body must preserve the invariant.

Furthermore, when the loops is finished and c no longer holds, this must imply the precondition of the continuation k. Finally, the entire expression has the postcondition Q_2 associated with the continuation k. Note that this formulation captures *partial correctness*; there is no variant ensuring that the loop must terminate eventually.

Using these semantics, we now define a refinement relation between statements programs in the WhileL language:

```
Definition WhileRefines (c1 c2 : WhileL a)
  := Refines (semantics c1) (semantics c2).
```

Once again, we will use the notation $c_1 \sqsubseteq c_2$ when WhileRefines c_1 c_2 holds.

Composing programs

It is rather straightforward to show that WhileL supports a monadic bind operation:

```
Fixpoint bind {a b : Type} (w : WhileL a) (k : a → WhileL b) : WhileL b
  := match w with
  | New v c ⇒ New v (fun p ⇒ bind (c p) k)
  | Read p c ⇒ Read p (fun v ⇒ bind (c v) k)
  | Write p v c ⇒ Write p v (bind c k)
  | While Inv cond body c ⇒ While Inv cond body (bind c k)
  | Spec pt ⇒ Spec (bindPT pt (fun x ⇒ semantics (k x)))
  | Return x ⇒ k x
  end.
```

The function is inductively defined on the first program w. The first four cases simply propagate the bind operation to their respective continuations. The only interesting case is that for specifications. The Spec case creates a new PT, by using its associated predicate transformer composition operator $bindPT$, corresponding to taking the relational composition of the specification and the semantics associated with the continuation. Finally, the Return case applies k to the value x.

Using this definition of bind, we can assemble larger, composite programs that use different effectful operations.

Example: square root

With these definitions in place, we can now formalize a refinement derivation such as the one in Figure 1. However, we do so with a program which calculates the square root of a number using binary search, taken from our previous work. This program is not particularly interesting algorithmically, but illustrates how to use all of the refinement rules we have seen so far.

We begin by defining the specification:

```
**Definition** sqrtSpec $(P : \mathsf{Ptr}) : \mathsf{PT}\ \mathsf{nat} :=$
  $[\ (\textbf{exists}\ t, \mathsf{find}\ s\ P = \mathsf{Some}\ t), (\ v^2 \leqslant t < (v+1)^2)]$
```

The precondition requires that P is a valid pointer in our heap, pointing to some value t. The postcondition guarantees that after execution, the result v will be the square root of the original value t.

Next we can sketch a candidate sqrt program:

```
$t \leftarrow \mathsf{read}\ P$
$Q \leftarrow \mathsf{new}\ Q\ (t+1)$
$R \leftarrow \mathsf{new}\ R\ 0$
while (read $R + 1 \not\equiv$ read $Q$)
  $q \leftarrow \mathsf{read}\ Q$
  $r \leftarrow \mathsf{read}\ R$
  let $mid := (q + r)\ /\ 2$
  if $(t < mid^2)$
  then write $Q\ mid$
  else write $R\ mid$
$r \leftarrow \mathsf{read}\ R$
return $r$
```

The program starts by creating two auxiliary variables R and Q, which will serve as the search bounds. The lower bound R is initially set to 0, while the upper bound Q is set to $(t + 1)$. The **while** loop then halves (using integer division) the interval between these bounds on every iteration until the bounds are consecutive numbers. The loop also needs an invariant, which should imply the post-condition. We define this in Coq as:

```
**Definition** sqrtInv $(Q\ R : \mathsf{nat})\ (t : \mathsf{nat})\ (s : \textbf{heap}\ \mathsf{nat}) :=$
  **exists** $q, r.\mathsf{find}\ s\ Q = \mathsf{Some}\ q \wedge \mathsf{find}\ s\ R = \mathsf{Some}\ r \wedge R \neq Q \wedge$
           $r^2 \leqslant n < q^2.$
```

This invariant can be read as follows: for any value q and r referenced by pointers Q and R, respectively, the original value t must be in between the square of q and r. Upon completion, the program returns the value referenced by R.

Proving this program satisfies its specification, amounts to proving the following sqrtCorrect lemma.

```
Lemma sqrtCorrect (P : Ptr) :
  sqrtSpec P ⊑ sqrt P.
```

To do so, we can unfold the definitions of sqrt, sqrtSpec, and the semantics we defined above, yielding a complex verification condition.

This form of post-hoc verification is very different from the interactive program calculation that we would like to perform. We need to write the final program *before* we start our proof. After unfolding definitions and β-reduction, the proof goal that we are left with is often large and unwieldy.

In the next section we will develop machinery to enable the interactive discovery of programs, rather than the mere transcription of an existing proof.

5. Interactive refinement

Although we can now take any pen-and-paper proof of refinement and verify this in Coq, we are not yet playing to the strengths of the *interactive* theorem prover that we have at hand. In this section, we will show how to develop lemmas and definitions on top of those we have seen so far that facilitate the interactive calculation of a program from its specification.

We start by defining a function that determines when a statement is executable, i.e., when there are no occurrences of the Spec constructor:

```
Fixpoint isExecutable (c : WhileL a) : Prop
```

Rather than fixing the exact program upfront, we can now reformulate the correctness lemma of swap as follows:

```
Definition deriveSqrt (P : Ptr) :
{ c : WhileL nat
  & (sqrtSpec P) ⊑ c
  & isExecutable c }.
```

The notation $\{x : A \,\&\, P\ x \,\&\, Q\ x\}$ in Coq is used to denote a dependent triple consisting of a witness $x : A$, a proof that x satisfies the property P and a proof that x satisfies the property Q.

To prove this lemma we need to provide an executable c : WhileL nat and a proof that sqrtSpec $\sqsubseteq$ c. This is a superficial change – we could now complete the proof by providing our sqrt program as the witness c and reuse our previous correctness lemma. Instead of doing this, however, we wish to explore how to reformulate typical refinement calculus laws to enable the interactive construction of a suitable program.

We begin by observing that the semantics of our WhileL language proceeds structurally over the various language constructs. As a result, we can define a lemma for every language construct, describing precisely when a refinement step introducing that construct is valid. This corresponds to unfolding the definition of our semantics, for every construct of the WhileL language. For example, the rule for the Write statement is formulated below.

```
Lemma writeRefines (w w' : WhileL a) (ptr : Ptr) (y : v)
  (d : pre (semantics w) ⊆ pre (semantics (Write ptr y w')))
  (h : forall (s : S) (p : pre (semantics w) s) (x : a),
    post (semantics w') (update s ptr y) (snd (d s p)) x
     ⊆ post (semantics w) s p x)
  : w ⊑ Write ptr y w'.
```

Despite the apparent complexity, this lemma is trivial to prove: the two hypotheses are exactly what is needed to show that w is refined by Write $ptr\ v\ w'$. The first proof obligation states that the precondition of w should imply the precondition of Write $ptr\ v\ w'$; the second states that when the postcondition of w' holds on the *updated state*, i.e. after assigning v to the pointer ptr, the postcondition of w must also hold.

During the interactive refinement of a specification, however, we typically are not interested in refining two arbitrary WhileL programs, but rather calculating an executable WhileL program from its specification. Therefore, we can specialize the writeRefines lemma above to the case when the first program is a specification. Unfolding the definition of semantics and massaging the required hypotheses slightly yields the following lemma:

```
Lemma writeSpec (ptr : Ptr) (y : v) (spec : PT a) (w : WhileL a)
  (H : forall s, pre spec s → {x : v & find s ptr = Some x})
  (Step : Spec
    ([fun s ⇒ {t : S & prod (pre spec t) (s = (update t ptr y))},
      fun s pres x s' ⇒
        (post spec (projT1 pres) (fst (projT2 pres)) x s')]) ⊑ w) :
  Spec spec ⊑ Write b ptr y w.
```

Essentially, this lemma states that to refine a specification with a Write statement, we need to prove that the pointer is already allocated on the heap (the argument H). Furthermore, we need to show after performing this update, we can refine the specification on the updated heap to some program w. Here the usage of relations to represent our postconditions introduce a bit of clutter, having to take deconstruct various parts of the precondition using the projections projT1, fst, and projT2 to formulate the desired postcondition.

We can define similar lemmas describing the verification conditions associated with introducing other constructors of the WhileL language, such as reading from memory, allocating a fresh pointer, introducing an if clause, or introducing a loop.

Note that in some cases, the continuation argument requires a more complex, higher-order hypothesis. For example, the readSpec lemma has the following general shape:

```
Lemma readSpec (ptr : Ptr) (spec : PT a) (w : b → WhileL a)
(H : ...)
(Step : forall v, Spec [...]) ⊑ w v) :
Spec spec ⊑ Read b ptr w.
```

Here the *Step* argument quantifies over the result of the read, v; the specification makes precise that v is stored on the heap at address ptr. This complexity arises as we *are* interested in the result produced by Read, whereas Write does not produce and interesting result for the remainder of the computation. This is a natural generalization of the case for Write to handle those cases where some command produces a result used by the remaining computation.

We can now introduce custom tactics for each of these lemmas. These tactics apply the lemma and call some rudimentary proof automation:

Ltac WRITE $ptr\ v$:= eapply (writeSpec $ptr\ v$); simpl_goal.

Here the simpl_goal tactic unfolds various definitions, triggers β-reduction, and generally cleans up the proof context and goal. The tactic for While follows a similar structure:

Ltac WHILE $I\ c$:= eapply (whileSpec $I\ c$); simpl_goal.

The constructors that have a non-trivial continuation may introduce new variables, hence we adapt our tactics accordingly.

Ltac READ $ptr\ v$:= eapply (readSpec ptr); [| intros v]; simpl_goal.

We can also provide custom machinery for constructs defined in Coq itself, such as conditionals:

Lemma ifSpec ($cond$: bool) ($spec$: PT $v\ a$) ($wt\ we$: WhileL $v\ a$) :
  (**if** $cond$ **then** Spec $spec \sqsubseteq wt$ **else** Spec $spec \sqsubseteq we$) $\rightarrow$
  (Spec $spec \sqsubseteq$ (**if** $cond$ **then** wt **else** we)).

The associated tactic applies this lemma, remembers the condition, and splits the goal into two sub-goals accounting for the condition's value:

Ltac IFF c := eapply (ifSpec c); remember c as b; destruct b; simpl_goal.

Finally, we can conclude any refinement calculation provided we have a suitable value and can show that the precondition implies the postcondition for any state s:

Lemma returnStep (v : a) (w : WhileL a)
  (H : **forall** (s : S) (P : pre (semantics w) s), post (semantics w) s pre $v\ s$) :
  $w \sqsubseteq$ Return v.

We also define a corresponding tactic, RETURN v.

We now have the ingredients to tackle our problem: interactively proving deriveSqrt. That is, we must produce some code c, together with a proof that it refines our specification. The proof proceeds by introducing a new existential variable for c, which will be constructed while we perform the refinement calculation. We proceed by finding a refinement derivation by applying tactics

```
econstructor.split.
 - READ $P$ $t$.
   NEW $(s + 1)$ $Q$.
   NEW $0$ $R$.
   WHILE (sqrtInv $Q$ $R$ $s$) ($Cond$ $Q$ $R$).
    + ... (* A proof of pre sqrtSpec $\to$ sqrtInv *)
    + READ $Q$ $vInQ$.
      READ $R$ $vInR$.
      IF $(s <? ((vInQ + vInR) / 2)^2)$
       * WRITE $Q$ $((vInQ + vInR) / 2)$
         RETURN $tt$.
          ... (* heap manipulation *)
       * WRITE $P$ $((vInQ + vInR) / 2)$
         RETURN $tt$.
          ... (* heap manipulation *)
    + READ $R$ $vInR$.
      RETURN $tt$.
       ... (* heap manipulation *)
 - ... (* proving $c$ to be executable *)
```

Figure 3: Proof sketch of deriveSqrt

one by one, that each modify the remaining specification. We have sketched the proof in Figure 3. Although it leaves out several proof obligations, it is hopefully clear that the proof structure closely follows the code. Once we have shown our derivation correct, we still need to prove that the resulting program is executable—but this is trivial for any complete program.

The missing proof steps are not too long, but they detract from the overall structure of the proof that we wish to illustrate here. We refer the interested reader to our Coq development. Upon completing this proof, we can project out the $c : \mathsf{WhileL\ unit}$ value to obtain the verified program.

In this style, it is possible to start from a specification and incrementally write a program that satisfies it. Every step of the way, you can inspect how the specification evolves after applying a specific command. For larger developments, however, it may sometimes be desirable to intersperse verification and program development. In the calculation of sqrt above, the only real verification is done after each RETURN statement. One way to remedy this is by explicitly reformulating the specification:

```
Lemma changeSpec (pt₁ pt₂ : PT a) (w : WhileL a)
  (d : pre pt₂ ⊆ pre pt₁)
  (h : forall (s : S) (x : pre pt₂ s) v, post pt₁ s (d s x) v ⊆ post pt₂ s x v)
  (H : Spec pt₁ ⊑ w) :
  Spec pt₂ ⊑ w.
```

The changeSpec lemma follows immediately from transitivity of our refinement relation. Essentially, it allows us to replace the current specification with any other specification that refines it. We can also introduce a corresponding tactic:

```
Ltac ASSERT P := unshelve eapply (changeSpec P).
```

This tactic applies the changeSpec lemma, thereby replacing the current specification with P. The unshelve tactic requires the user to complete the proof obligations associated with this application of changeSpec, before continuing the rest of the program calculation. Using these tactics, we are free to mix verification and calculational steps.

6. Case study: persistent arrays

As a final case study, we describe the derivation of a library for *persistent arrays*. A persistent array is a data-structure that stores a regular array and maintain the state of previous versions of the same array. Persistent arrays can be used as efficient building blocks of other data-structures, as in the union-find data-structure. In our development, we will follow the formalization by Conchon and Filliâtre (2007) done previously in Coq.

The formalization by Conchon and Filliâtre uses two different modules. The first provides an implementation and interface for persistent arrays; whereas the second implements the union-find algorithm using this module. Any side effects are encapsulated in the persistent array module, providing a pure interface for the union-find structure. We will sketch how to calculate the persistent array module on which the union-find data structure relies using our library.

The module for persistent arrays has the following signature in OCaml:

```
module type PersistentArray = sig
  type a t
  val init : int → (int → a) → a t
  val get : a t → int → a
  val set : a t → int → a → a t
end
```

This module provides quite standard operations for persistent arrays. The function init takes an integer corresponding to the size of the array and functional representation of the array and returns a new persistent array t holding values of type a. The function get should takes such a value of type $a\ t$, an index of the array and returns the value stored there. Finally the function set accepts an array, an index and a value, and returns a new container with the old value on the given index replaced by the value passed as argument.

The derivation of each of these function will consist of three parts: a specification and an interactive proof demonstrating that we can find an implementation that meets the specification. For the sake of presentation, however, we will begin discussing the functions to familiarize readers with the algorithms we intend to derive. The code for these functions is given in Figure 4.

The inductive data-type PAData represents the data to be stored in the heap. The Arr constructor holds an array, again represented as a function from indexes to values – for simplicity, we restrict arrays to only store natural numbers and we let this function be total. Just as in the work by Conchon and Filliâtre, the regular array lookup is represented as function application; whereas updating an array at some position with some value, can be done using the update operation.

The Diff constructor holds a position, a value, and a pointer to another array. It represents a single change in the *referenced* array at the given position with the given value. The Diff constructor is what makes the arrays *persistent*.

The implementation of init simply creates a new array and returns the pointer associated to it. The get function recursively looks up a value in the chain of Diffs until it finds an Arr constructor. The set function behaves differently depending on the value referenced by t. In the Arr case, the array is modified, and t is set as a new indirection step and the (modified) array is returned. On the other hand, if the value in t is already a Diff, then a new indirection step is created an returned.

Note that Conchon and Filliâtre show a slightly different and more efficient way to implement get and set. However, this requires a new *reroot* function, which would introduce more complexity to our formalization.

We can now formalize specifications of the persistent array functions. When doing so, we notice that the important property to be verified is that each function should ensure that all pointers it uses should refer to a well-formed

```
Inductive PAData : Type :=
  Arr (nat → nat)
   | Diff nat nat Ptr
init n f = new (Arr f) (fun ptr ⇒ Return ptr)
get t i =
  read t ≫= fun tval ⇒
  match tval with
     | Arr a ⇒ lookup a i
     | Diff (j, v, t′) ⇒
       if i ≡ j then v else get t′ i
set t i v =
  read t ≫= fun tval ⇒
  match tval with
     | Arr a ⇒
       let old = lookup a i in
       new (Arr (update a i v)) ≫= fun res ⇒
       write t (Diff i old res) ≫
       return res
     | Diff _ ⇒ new (Diff i v t)
```

Figure 4: Functions manipulating persistent arrays

structure of a persistent array. In order to express that property, we define the following predicate, taken from Conchon and Filliâtre (2007):

```
Inductive pa_model (s : heap) : Ptr → (nat → nat) → Type :=
  | pa_model_array :
    forall p f, find s p = Some (Arr f) → pa_model s p f
  | pa_mode_diff :
    forall p i v p′, find s p = Some (Diff i v p′) →
      forall f, pa_model s p′ f →
      pa_model s p (update f i v).
```

This predicate allows us to relate a heap, a pointer and a function that represents the referenced array. One may informally explain this predicate as: pa_model s p f holds when a pointer p present in heap s, points to a valid persistant array represented by f. The constructor pa_model_array states that any pointer that references an array is a valid pointer and the function it represents is the f to which it is dereferenced. The constructor pa_mode_diff says that any pointer p referencing a Diff i v p' is also a valid pointer, as long as one can prove this property inductively for p'. The function representing the array will be an update of the resulting f in position i of v.

After having defined this predicate, we are now ready to formulate our specifications and derive their implementation.

The derivation of init

The init function is not only straightforward in its definition but also in its specification:

$$
\begin{array}{l}
\textbf{Definition}\ \mathsf{initSpec}\ (n : \mathsf{nat})\ (f : \mathsf{nat} \to \mathsf{nat}) : \mathsf{PT}\ \mathsf{Ptr} := \\
\quad [\,\textbf{fun}\ s \Rightarrow \mathsf{True} \\
\quad ,\textbf{fun}\ s\ \mathit{pres}\ v\ s' \Rightarrow \\
\qquad (\textbf{forall}\ p'\ x : \mathsf{find}\ s\ p' = \mathsf{Some}\ x \to p' \neq v) \wedge \mathsf{pa\_model}\ s'\ v\ f\,].
\end{array}
$$

The precondition is trivial: init may be called regardless of the current state of the heap. The postcondition is slightly more interesting. It states that the returned pointer v should be fresh and that it points to a valid array, represented by f. Deriving a suitable implementation for such the init function is straightforward.

The derivation of set

The specification for the set function is slightly more elaborate:

$$
\begin{array}{l}
\textbf{Definition}\ \mathsf{setSpec}\ (\mathit{ptr} : \mathsf{Ptr})\ (i : \mathsf{nat})\ (v : \mathsf{nat}) : \mathsf{PT}\ \mathsf{Ptr} := \\
\quad [\,\textbf{fun}\ s \Rightarrow \textbf{exists}\ f, \mathsf{pa\_model}\ s\ \mathit{ptr}\ f \\
\quad ,\textbf{fun}\ s\ \mathit{pres}\ \mathit{newPtr}\ s' \Rightarrow \\
\qquad \mathsf{pa\_model}\ s'\ \mathit{newPtr}\ (\mathsf{update}\ f\ i\ v) \wedge \mathsf{pa\_model}\ s'\ \mathit{ptr}\ f\,].
\end{array}
$$

The precondition requires a function f to exist, such that *ptr* points to a valid persistent array modeled by f. The postcondition ensures that, both the returned pointer and the original pointer will point to valid persistent arrays, and that the former is a suitably modified version the latter.

The derivation of the implementation of setSpec requires more manual proofs and requires several auxiliary properties relating the heap and pa_model. However, the application of these properties was quite systematic. Notably, this shows that there may be further room for customizable automation. When dealing with *custom* predicates such as pa_model, our rudimentary automation to simplify terms containing heaps does not have much effect. We believe that further effort, such as constructing suitable hint databases or tactic customization opportunities, the proof-to-program ratio can be much improved.

The derivation of get

The specification for the get function is straightforward:

$$
\begin{array}{l}
\textbf{Definition}\ \mathsf{getSpec}\ (\mathit{ptr} : \mathsf{Ptr})\ (i : \mathsf{nat}) : \mathsf{PT}\ \mathsf{nat} := \\
\quad [\,\textbf{fun}\ s \Rightarrow \textbf{exists}\ f : \mathsf{pa\_model}\ s\ \mathit{ptr}\ f \\
\quad ,\textbf{fun}\ s\ \mathit{pres}\ v\ s' \Rightarrow v = f\ i\,].
\end{array}
$$

Somewhat surprisingly, this turned out to be the hardest of the three functions to derive. Despite the simplicity of the specification, the implementation is quite complex. Notably, it follows pointers in the heap and is not (obviously) structurally recursive. We chose to derive a variant of the original definition, introducing a While loop. The proof obligations can grow quite unwieldy – once again highlighting the need for further automation to keep their complexity in check.

We have sketched how to derive a library for persistent arrays. We expect that this development can be further combined with the development of other libraries that use persistent arrays as its key components, such as the existing Coq verification of the union-find data structure.

7. Discussion

The choice of our PT types and definition of refinement relation are not novel. Similar definitions of *indexed containers* (Altenkirch and Morris, 2009) and *interaction structures* (Hancock and Setzer, 2000a,b) can already be found in the literature. Indeed, part of this work was triggered by Peter Hancock's remark that these structures are closely related to *predicate transformers* and the refinement relation between them, as we have made explicit in this paper.

We are certainly not the first to explore the possibility of embedding a refinement calculus in a proof assistant. One of the first attempts to do so, to the best of our knowledge, was by Back and Von Wright (Back and von Wright, 1989). They describe a formalization of several notions, such as weakest precondition semantics and the refinement relation, in the interactive theorem prover HOL. This was later extended to the *Refinement Calculator* (Butler et al., 1997), that built a new GUI on top of HOL using Tcl/Tk. More recently, Dongol et al. have extended these ideas even further in HOL, adding a separation logic and its associated algebraic structure (Dongol et al., 2015). There are far fewer such implementations in Coq, Boulmé (2007) being one of the few exceptions. In contrast to the approach taken here, Boulmé explores the possibility of a monadic, shallow embedding, by defining the *Dijkstra Specification Monad.* Where Boulmé's work explores the lattice theoretic structure and fixpoint theory of refinement relation in Coq, it lacks custom refinement such as those presented here.

There is a great deal of work marrying effects and dependent types. Swierstra's thesis explores one potential avenue: defining a functional semantics for effects (Swierstra, 2009b; Swierstra and Altenkirch, 2007). For some effects, such as non-termination, defining such a functional semantics in a total language is highly non-trivial. Therefore, systems such as Ynot take a different approach (Nanevski et al., 2008b). Ynot extends Coq with several axioms, corresponding to the different operations various effects support, such as reading from and writing to mutable state. The type of these axioms captures all the information that a programmer may use to reason about such effects. These types are similar to those presented here in Figure 2. Contrary to the approach taken here, however, Ynot lets users write their programs without considering their specification. Users only need to write proofs after specifying the pre- and postconditions for a certain function. The refinement calculus, on the other hand, starts from a specification, which is gradually refined to an executable program.

In the future, we hope to investigate how these various approaches to verification may be combined. One obvious next step would be to re-use the separation logic and associated proof automation defined by later installments of Ynot (Chlipala et al., 2009) as the model of the heap in our refinement calculus. Furthermore, we have (for now) chosen to ignore the variants associated with loops. As a result, the programs calculated may diverge. Embellishing our definitions with loop variants is straightforward, but will make our definitions even more cumbersome to use.

Type theory and the refinement calculus are both frameworks that combine specification and calculation. By embedding the refinement calculus in type theory, we study their relation further. The interactive structure of many proof assistants seems to fit well with the idea of *calculating* a program from its specification step-by-step. How well this approach scales, however, remains to be seen. For now, the embedding presented in this paper identifies an alternative point in the spectrum of available proof techniques for the construction of verified programs.

Acknowledgments. The second author would like to thank Peter Hancock for his patience in explaining the relation between interaction structures and the refinement calculus. The second author's visit to Scotland was funded by the *London Mathematical Society's* Scheme 7 grant.

Thorsten Altenkirch and Peter Morris. Indexed containers. In *Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on*, 2009.

R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2, 1989.

Ralph-Johan Back and J Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.

Ralph-Johan R Back and Joakim von Wright. Refinement concepts formalised in Higher Order Logic. *Formal Aspects of Computing*, 2(1):247–272, 1990.

R.J.R. Back. *On the Correctness of Refinement in Program Development*. PhD thesis, University of Helsinki, 1978.

Sylvain Boulmé. Intuitionistic refinement calculus. In *Typed Lambda Calculi and Applications*, pages 54–69. Springer, 2007.

M.J. Butler, J. Grundy, T. Långbacka, R. Ruksenas, and J. von Wright. The refinement calculator. In *Formal Methods Pacific*, 1997.

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *International Conference on Functional Programming*, ICFP '09, 2009.

Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on ML*, pages 37–46. ACM, 2007.

Edsger W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

Brijesh Dongol, Victor B.F. Gomes, and Georg Struth. A program construction and verification tool for separation logic. In *Mathematics of Program Construction*, volume 9129 of *LNCS*, 2015.

Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1):189–239, 2006.

Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *Computer Science Logic*, pages 317–331, 2000a.

Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming*, 2000b.

Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Carroll Morgan. *Programming from specifications.* Prentice-Hall, Inc., 1990.

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming*, 2008a.

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming*, ICFP '08, 2008b.

Wouter Swierstra. A Hoare logic for the state monad. In *Theorem Proving in Higher Order Logics*, pages 440–451. Springer, 2009a.

Wouter Swierstra. *A functional specification of effects.* PhD thesis, University of Nottingham, 2009b.

Wouter Swierstra and Joao Alpuim. From proposition to program: embedding the refinement calculus in Coq. In *International Symposium on Functional and Logic Programming*, pages 29–44. Springer, 2016.

Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: a functional semantics for the awkward squad. In *Haskell Workshop*, pages 25–36, 2007.