A Mechanized Theory of Program Refinement

Boubacar Sall, Frédéric Peschanski, Emmanuel Chailloux

A Mechanized Theory of Program Refinement

Boubacar Demba Sall, Frédéric Peschanski, and Emmanuel Chailloux

Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6
F-75005 Paris, France
{boubacar.sall, frederic.peschanski, emmanuel.chailloux}@lip6.fr

Abstract. We present a mechanized theory of program refinement that allows for the stepwise development of imperative programs in the Coq proof assistant. We formalize a design language with support for gradual refinement and a calculus which enforces *correctness-by-construction*. A notion of *program design* captures the hierarchy of refinement steps resulting from a development. The underlying theory follows the *predicative programming* paradigm where programs and specifications are both easily expressed as predicates, which fit naturally in the dependent type theory of the proof assistant.

Keywords: Stepwise refinement · Program verification · Predicative programming · Correctness by construction · Type theory · Proof assistant · Coq

1 Introduction

Program development by *stepwise refinement* [8,23,25] is inherently an interactive activity alternating programming steps and proof steps feeding back on each other. It therefore makes sense to perform refinement steps within an interactive proof assistant whose very purpose is to help in specification, proof composition and mechanical proof verification. This however requires that a theory of program refinement be embedded in the formalism of the proof assistant (p.a.). Most theories of refinement fall in two groups. The theories (e.g. [9,19]) in the first group are based upon the calculus of relations, and represent programs as well as specifications uniformly as set-theoretic relations on program states. In the second group, the theories (e.g. [20,3]) are underpinned by Hoare logic or the wp-calculus[1]. The view there is that programs relate sets of program states represented in logic as predicates, while specifications are pairs of predicates. In this paper, we investigate the *predicative programming* [10,22] approach to refinement which can be viewed as an expression of the relational point of view in predicative terms rather than in set theory. This investigation lead to the development of a mechanized theory of stepwise refinement towards imperative programs (similar to those studied in [20]) in the realm of the Coq p.a. [24]. We have the following contributions.

[1] calculus of weakest preconditions

Firstly, we formalize in type theory a language of stepwise program design, and a calculus which enforces correctness by construction. The language is an extension of a simple imperative language with assignment, sequence, if statements and iteration. The extension is to allow the expression of an hierarchy of refinement steps, each step associating a specification to an implementation. The calculus stems from a synthesis of ideas from predicative and relational theories of refinement. On the one hand, the relational point of view unifies the usual assertions (e.g. precondition, post-condition, invariant) under a single and more general notion of specification, hence simplifying the formulation of the theory. On the other hand, the predicative point of view makes it easier to write specifications and handle proof obligations (p.o.s).

Secondly, we uncover necessary and sufficient conditions for a while statement to refine a given specification. In particular, the loop body must be given an adequate relational specification. The advantage is that theses specifications are more flexible than invariants.

Finally, we have mechanized the aforementioned calculus as well as the underlying theory in the Coq p.a. so that the declaration of a refinement step automatically triggers the generation of p.o.s (in the language of the p.a.) to ensure the correctness of the refinement. All the artifacts presented in the paper (definitions and theorem statements), as well as all the proofs, have been formalized in about 4000 lines of Coq script and made available in a companion repository[2]. This framework thus permits certified imperative program design by gradual refinement, and demonstrates that, in terms of mechanized semantics in type theory, the relational point of view is a viable alternative to Hoare-logic style semantics. Moreover, the Coq p.a. provides a full blown functional language to write specifications with the benefit of type checking, type inference and parametric polymorphism for free.

The outline of the paper is as follows. In Section 2 we give an overview of program development by stepwise refinement in our proposed framework. In Section 3 we introduce the language of program designs, we describe the rules of the calculus, and we formulate the corresponding correctness theorem. In Section 4 we formalize the semantics of our language, we define the refinement relation based on this semantics, and we justify the design rules introduced in Section 3. In Section 5 we turn our attention to making the framework more practical by applying some automatic simplifications to the p.o.s. A discussion of related work and the conclusion follow.

2 An overview of stepwise program design

To give an overview of our framework as experienced by the user, we use a classical example: the (integer) square root computation. We begin by declaring the following abstract definition of the square root computation:

$$\textbf{Program Definition } \mathsf{Sqrt} := \langle r'^2 \leq x < (r'+1)^2 \rangle_x.$$

[2] https://github.com/bsall/AMToPR-ICFEM-2019

With the Definition keyword we have named our computation Sqrt, and declared its high level specification. The role of the Program keyword will be clarified shortly, it has to do with the handling of p.o.s. Most specifications we will write are akin to *before-after predicates* which describe the relation between the initial state and the final state of a program. We use the usual convention that variables are primed to mean their value after execution, and unprimed to mean their value before execution. The specification[3] of the square root computation reads as follows: from the initial value of variable x we aim to compute its square root and make the result available as the final value of r (denoted r'). Additionally, the value of x is required to remain unchanged. Our objective is to elaborate, incrementally, a program to fulfill this specification. Once we have decided on a more precise implementation, we open braces to write this implementation. In our example, this first refinement step leads to the following situation:

```
(1) Program Definition Sqrt := $\langle\ r'^2 \leq x < (r'+1)^2\ \rangle_x$ {
        $r := 0$;
        $h := x + 1$;
        $\langle\ r^2 \leq r'^2 \leq x < h'^2 \leq h^2\ \wedge\ r' + 1 = h'\ \rangle_x$
    }.
```

Our initial specification has developed into a *specified block* [11] whose header is the initial specification. The body of the block introduces a new variable h so that $[r..h]$ delimits the search space. The last statement specifies the search strategy we intend to implement, i.e. narrow the search space around x (first conjunct), up to the point where the initial specification is fulfilled (second conjunct). The notations being used to design the square root computation (e.g. $\langle ... \rangle$ $\{...\}$, ;) are syntactic sugar for invoking specific constructions rules defined in Section 3.2. These rules ensure the correctness of our design with respect to the semantics defined in Section 4. In particular the rules stipulate that to construct the specified block above, one must provide a proof that the body of the block refines (in the sense of Section 4.2) the header of the block. Thanks to the Program keyword we can mimic the construction of a specified block and let the p.a. save the p.o. corresponding to the missing proof for later. This p.o. is to be discharged separately so that our design is not cluttered with the details of the proofs. At this point, even though the p.a. has performed some type checking automatically, the Sqrt definition is not yet complete: for example it cannot be referred to in other definitions. To complete the definition, we must provide the missing proof by writing a proof script which is generally of the following form:

```
Next Obligation. $t_1$. ... $t_i$. nia. ... $t_j$. ... $t_k$. Qed.
```

The Next Obligation command is to fetch and display the next p.o. among those left to be discharged. The following sequence of t_i commands invoke built-in proof tools called *tactics*, for example the nia tactic used to deal with non linear arithmetic is very helpful in our case. Finally, the Qed command instructs the p.a. to check our proof for validity. By repeating the process we have just described

[3] The notation $\langle S \rangle_{x_1,\ldots,x_n}$ is a shorthand for $\langle S \wedge x'_1 = x_1 \wedge \cdots \wedge x'_n = x_n \rangle$.

to make specifications more and more precise, we ultimately obtain the design of Listing 1.1 below after three more refinement steps (labeled (2), (3) and (4)).

```
(1) Program Definition Sqrt := $\langle\, r'^2 \le x < (r'+1)^2 \,\rangle_x$ {
        $r := 0$;
        $h := x + 1$;
(2)     $\langle\, r^2 \le r'^2 \le x < h'^2 \le h^2 \;\wedge\; r' + 1 = h' \,\rangle_x$ {
          while $r + 1 \neq h$ do
(3)         $\langle\, r^2 \le r'^2 \le x < h'^2 \le h^2 \;\wedge\; (r < r' \;\vee\; h > h') \,\rangle_x$ {
(4)           $\langle\, r < m' < h \,\rangle_{x,r,h}$ {
                $m := r + (h - r)/2$
              };
              if $m^2 \le x$ then $r := m$ else $h := m$ end
            }
          done
        }}.
```

Listing 1.1. The square root program design with refinement steps (1) to (4)

The second refinement step has consisted in deciding to implement the search strategy as a loop. The body of the loop required two more refinement steps (numbered (3) and (4)). An important remark is that all the successive steps are visible in the final design: one can imagine collapsing specified blocks showing only the associated specifications, or instead expanding blocks to dig into the implementation details. Once all p.o.s are discharged we have built an object carrying programming instructions, the design decisions (refinement steps) that lead to those instructions, and the proofs of correctness of all refinements. We call this object a *program design*. Keeping all refinement steps around is important because the corresponding design decisions tend to be lost as time passes, rendering program evolution ever more harder. Thanks to the proofs carried by the program design, our framework is able to assemble a global certificate of design correctness. To assemble such a certificate we write the following script:

```
Definition sqrt_proof := CbC.soundness Sqrt.
```

In other words, the sqrt_proof term results from applying the soundness theorem of our design rules (the Cbc.soundness term) to the Sqrt design. This term is a proof that can be independently checked for validity. The soundness theorem of the design rules is formulated in Section 3. We refer the reader to the companion repository for a detailed example[4] of how the refinement process is carried out for the square root computation.

3 The calculus of program designs

In this section we formally present the calculus of program designs. We begin with the language of program designs. Then, we formulate the construction rules of the calculus whose purpose is to enforce *correctness-by-construction*.

[4] https://github.com/bsall/AMToPR-ICFEM-2019/tree/master/src/examples/

Statement S ::=

effect f	(with the state transformer $f : T \to T$)
$\mid \langle R \rangle$	(specification statement with $R : T \to T \to \mathsf{Prop}$)
$\mid S_1$ **;** S_2	(sequence)
$\mid$ **if** C **then** S_1 **else** S_2 **endif**	(if statement with condition $C : T \to \mathsf{Prop}^5$)
$\mid S_1 \{ S_2 \}$	(specified block with S_1 block free)
$\mid$ **while** C **do** S **done**	(iteration with condition $C : T \to \mathsf{Prop}$)

Fig. 1. The design language w.r.t. the type T of program states

3.1 The language of program designs

To design programs in the way described previously, we need a language with support for gradual refinement. Moreover, adhering to the *correctness by construction* [18] paradigm, our objective is to impose further restrictions so that only correct program designs can be constructed. The syntax of the core language we study is given in Figure 1. It is a very classical imperative language, close in spirit to the language studied in [21], but embedded in the Coq p.a. Since *Coq* is underpinned by a dependent type theory, type checking, type inference and parametric polymorphism are for free. In the p.a., the syntax is encoded as an inductive type implicitly parameterized by the type T of program states.

The statements of the language. Sequential composition as well as the statements related to the if and while keywords are self-explanatory. The effect statement reflects the notion of state transformation as a syntactic constructor. This provides a nice generalization of various kinds of effects. For example the **skip** instruction is defined as **effect** $(\lambda\ s \Rightarrow s)$. Assignment statements are also derivable. For example, the assignment v **:=** $v + w$ for v and w variables of type Nat translates to **effect** $(\lambda\ (v, w) \ \Rightarrow\ (v + w, w))$ where T is $\mathsf{Nat} \times \mathsf{Nat}$.

The $\langle R \rangle$ construct is a *specification statement* [20]. It can be thought of as standing for a "program fragment yet to be implemented", or alternatively as a procedure call to a program specified by R. The notation $\langle ... \rangle$ is from [17]. The encoding in Coq of before-after predicates is straightforward. For example $\lambda\ (i\ i' : \mathsf{nat}) \Rightarrow i' > i$ specifies a program that increases variable i. More generally, $\langle R \rangle$ designates a program P such that : (1) when started in state s the set of possible outputs of P is $\{\ s' \mid R\ s\ s'\ \}$, and (2) P terminates on input s exactly when $(\exists\ s' \cdot R\ s\ s')$ is true (i.e. the set of possible outputs is not empty). Non deterministic behavior is reflected by a number of possible outputs greater than one. Following [22], we equate abnormal termination in an error state with non termination, therefore **abort** $\equiv \langle\ \lambda\ (s\ s' : T) \Rightarrow \mathsf{False}\ \rangle$.

The specified block $S_1 \{ S_2 \}$ represents a pair of statements resulting from the refinement of S_1 by S_2 as explained in Section 2. This is the feature enabling gradual refinement. S_1 is called the abstraction of the block and S_2 is called the concretization. We assume that the abstraction S_1 does not itself contain specified blocks, we say that it is *block free.* Note that programming constructs

[5] Prop is the built-in type of logical propositions in the Coq p.a.

$$
\left[
\begin{array}{ll}
\varphi(\textbf{effect}\ f) & \stackrel{\text{def}}{=} (\textbf{effect}\ f)\ \{\ \ (\textbf{effect}\ f)\ \ \} \\
\varphi(\langle R \rangle) & \stackrel{\text{def}}{=} \langle R \rangle\ \{\ \ \langle R \rangle\ \ \} \\
\varphi(S_1 ; S_2) & \stackrel{\text{def}}{=} \varphi_a(S_1)\ ;\ \varphi_a(S_2)\ \{\ \ \varphi_c(S_1)\ ;\ \varphi_c(S_2)\ \ \} \\
\varphi(\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}) & \stackrel{\text{def}}{=} \textbf{if}\ C\ \textbf{then}\ \varphi_a(S_1)\ \textbf{else}\ \varphi_a(S_2)\ \textbf{endif}\ \{ \\
 & \qquad \textbf{if}\ C\ \textbf{then}\ \varphi_c(S_1)\ \textbf{else}\ \varphi_c(S_2)\ \textbf{endif} \\
 & \quad \} \\
\varphi(S_1\ \{\ S_2\ \}) & \stackrel{\text{def}}{=} S_1\ \{\ \ \varphi_c(S_2)\ \ \} \\
\varphi(\textbf{while}\ C\ \textbf{do}\ S\ \textbf{done}) & \stackrel{\text{def}}{=} \textbf{if}\ C\ \textbf{then}\ \langle\ \lambda\ s\ s' \Rightarrow [\![\varphi_a(S)]\!]\ s\ s' \wedge \neg C\ s'\ \rangle\ \textbf{endif}\ \{ \\
 & \qquad \textbf{while}\ C\ \textbf{do}\ \varphi_c(S)\ \textbf{done} \\
 & \quad \}
\end{array}
\right.
$$

Fig. 2. The design projection function φ

are welcome in the abstraction of specified blocks. For example, the abstraction of a block can be of the if-then-else form. During program design, when we write $S_1\ \{\ S_2\ \}$, we record a design decision which communicates an abstract intent (S_1), and some (hopefully more concrete and correct) means of realizing it (S_2).

The φ projection function. We now define the projection function φ, which will be used in the formulation of the rules and properties of the calculus. The definition is given in Figure 2 above. From a statement S, φ computes an abstraction $\varphi_a(S)$, and a (generally distinct) concretization $\varphi_c(S)$, such that both statements are block free. We write $\varphi(S) = \varphi_a(S)\ \{\ \varphi_c(S)\ \}$ to match the syntax of the language. For example, if we consider the program design of Listing 1.1, the corresponding abstraction is the outermost specification $\langle\ r'^2 \leq x < (r'+1)^2\ \rangle_x$, and the concretization is the statement resulting from the intermediate specifications (lines in blue labeled (1) up to (4)) being ignored.

φ is defined so that the effect and $\langle R \rangle$ statements are respectively their own abstraction. Concerning sequential composition and the if-the-else construct, φ simply extracts the (abstraction,concretization) pairs from the inner statements and combines them in parallel.

In the case of the specified block the corresponding abstraction is just S_1 by definition, and the concretization is the one of the body of the block.

The abstraction of a loop is constructed from the abstraction of its body, which therefore must be specified with care. We come back to this in Section 4.3. The $[\![\cdot]\!]$ operator refers to the interpretation of statements as binary relations on program states. This interpretation is detailed in Section 4.1. To give an example, $[\![i := i+1]\!]$ denotes the predicate $i' = i+1$ (encoded as $(\lambda\ i\ i'\ \Rightarrow\ i' = i+1)$ in the p.a.).

Actually we use $\varphi_a(S)$ to reason about S while abstracting away from implementation details, and we compute $\varphi_c(S)$ to get a block free statement which can be translated into a programming language provided S is precise enough (i.e. no specification statements among the leaves of the syntax tree). The calculus we

$$\frac{}{\textsf{Design}\ (\textbf{effect}\ f)} \qquad \frac{}{\textsf{Design}\ \langle R\rangle} \qquad \frac{\textsf{Design}\ S_1\ \wedge\ \textsf{Design}\ S_2}{\textsf{Design}\ S_1;S_2}$$

$$\frac{\textsf{Design}\ S_1\ \wedge\ \textsf{Design}\ S_2}{\textsf{Design}\ (\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif})} \qquad \frac{\textsf{Design}\ S_2\ \wedge\ \varphi_a(S_2) \sqsubseteq S_1}{\textsf{Design}\ (S_1\ \{\ S_2\ \})}$$

$$\frac{\begin{array}{l}\textsf{Design}\ S\ \ \wedge\ \ K;K\ \sqsubseteq\ K, \quad \text{with } K \overset{\text{def}}{=} (\textbf{if}\ C\ \textbf{then}\ \varphi_a(S)\ \textbf{endif})\\ \wedge\ \textsf{well\_founded}\ (\lambda\ s\ s' \Rightarrow C\ s' \wedge ([\![\varphi_a(S)]\!]\ s'\ s) \wedge C\ s)\end{array}}{\textsf{Design}\ (\textbf{while}\ C\ \textbf{do}\ S\ \textbf{done})}$$

Fig. 3. The calculus of program designs

define in Section 3.2 allows to characterize those statements S such that $\varphi_c(S)$ is indeed a refinement of $\varphi_a(S)$.

3.2 Enforcing correctness-by-construction

The usefulness of a program design rests upon the correctness of what this design communicates. In order to enforce design correctness, we restrict our language by imposing strict construction rules. We say that a program design S is correct if and only if the predicate Design S can be derived from the rules of Figure 3. The two basic instructions effect and $\langle R\rangle$ are, unsurprisingly, the axioms of the proof system. All the statements involved in the other compound instructions must *already* be correct designs. The rule for specified blocks involves the refinement relation $\sqsubseteq$. This relation will be formally defined in Section 4.2, but for now we describe it as follows: we say that $S_2 \sqsubseteq S_1$ if and only if every specification satisfied by S_1 is also satisfied by S_2. In fact, when S_1 and S_2 are block free, our formalization allows to derive formally this description of refinement in terms of Hoare triples, i.e. we have established the following equivalence:

$$S_2 \sqsubseteq S_1 \quad \leftrightarrow \quad \forall\ P\ Q\ \cdot\ \{P\}\ S_1\ \{Q\}\ \rightarrow\ \{P\}\ S_2\ \{Q\}$$

The requirements for a block $S_1\ \{\ S_2\ \}$ to be a correct design are as follows. First, it must be the case that S_2 is a correct design, and moreover that $\varphi_a(S_2)$ refines S_1. In the rule for specified blocks and while loops, we can abstract away from implementation details and only consider $\varphi_a(S)$ because we assume S to be a correct program design. The rule for while loops is more intricate. The requirement that $K;K$ refines K is to ensure that K is a correct over-approximation of the loop's behavior. The well-foundedness requirement ensures as it is usually the case that the loop terminates on all inputs of interest. Ultimately, the central theorem of our proposed framework is the following one.

Theorem 1 (Correctness of program designs).

(soundness) $\forall\ S \cdot \textit{Design}\ S \rightarrow \varphi_c(S) \sqsubseteq \varphi_a(S)$
(completeness) $\forall\ S_1\ S_2 \cdot\ \ S_2 \sqsubseteq S_1 \rightarrow \exists\ S \cdot \textit{Design}\ S\ \wedge\ \varphi(S) = S_1\ \{\ S_2\ \}$

$\llbracket\cdot\rrbracket$: Statement $\to$ Spec	
$\llbracket$**effect** $f\rrbracket$	$\overset{\text{def}}{=} \lambda\ s\ s' \Rightarrow s' = (f\ s)$
$\llbracket\langle R\rangle\rrbracket$	$\overset{\text{def}}{=} R$
$\llbracket S_1$;$S_2\rrbracket$	$\overset{\text{def}}{=} \llbracket S_1\rrbracket\Box\llbracket S_2\rrbracket$
$\llbracket$**if** C **then** S_1 **else** S_2 **endif**$\rrbracket$	$\overset{\text{def}}{=} \llbracket S_1\rrbracket \lhd C \rhd \llbracket S_2\rrbracket$
$\llbracket S_1\ \{\ S_2\ \}\rrbracket$	$\overset{\text{def}}{=} \llbracket S_2\rrbracket$
$\llbracket$**while** C **do** S **done**$\rrbracket$	$\overset{\text{def}}{=}$ lfp[6] $(\lambda\ \mathcal{X} \Rightarrow (\llbracket S\rrbracket\Box\mathcal{X}) \lhd C \rhd \llbracket$**skip**$\rrbracket)$

Table 1. The predicative semantics of the design language

This theorem explains that whenever we are able to derive Design S for some program design, the associated concretization refines the associated abstraction. Clearly it is possible to have $\varphi_c(S) \sqsubseteq \varphi_a(S)$ while Design S is not derivable. For example consider the following program design

$$S \overset{\text{def}}{=} \textbf{skip}\ \{\ \langle\ \lambda\ s\ s'\ \Rightarrow\ \text{False}\ \rangle\ \{\ \textbf{skip}\ \}\ \}.$$

We have **skip** $\sqsubseteq$ **skip**, and yet Design S cannot be derived. So as one would expect the design rules are not complete in the absolute sense. However they are complete in the weaker sense that for any concretization S_2 and abstraction S_1 such that $S_2 \sqsubseteq S_1$, it is possible to come up with a correct design S whose associated concretization and abstraction are respectively S_2 and S_1. When we complete a design, the soundness part of Theorem 1 allows to construct a tangible lambda term certifying that our design is indeed correct: this lambda term is the certificate of correctness we alluded to in Section 2. The completeness part of Theorem 1 reassures us that the design rules we restrict ourselves to use do not themselves restrict the kind of programs that can be obtained by applying these rules.

4 Predicative semantics and refinement relation

In this section we discuss the key properties justifying the design rules of our calculus. First we present the predicative interpretation of statements and define the refinement relation in terms of this interpretation. Then, we examine the particular case of loops.

4.1 Predicative semantics

Except for loops and specified blocks, our interpretation of statements as predicates corresponds to the predicative semantics of [22]. We denote by Spec the

[6] least fixpoint

type of specifications, i.e. the type of binary relations on the type T of program states.

$$\mathsf{Spec} \overset{\text{def}}{=} T \to T \to \mathsf{Prop}$$

The predicative interpretation of statements associates to each statement a specification of type Spec. This interpretation is inductively defined on the syntax of statements as indicated in Table 1 above.

State transformation, specification statements and specified blocks. The specification associated to (effect f) explains that the after state is the image by f of the before state. The interpretation of $\langle R \rangle$ is just R since R is already a specification. For specified blocks, we choose the interpretation of the supposedly better implementation among the statements composing the block.

Alternative. The if-then-else statement denotes the specification described in Equation (1) below. Here, A and B are of type Spec and C has type $T \to \mathsf{Prop}$.

$$A \lhd C \rhd B \overset{\text{def}}{=} \lambda\ (s\ s' : T) \Rightarrow C\ s \wedge (A\ s\ s')\ \vee\ \neg(C\ s) \wedge (B\ s\ s') \tag{1}$$

The notation used is borrowed from [15], and expresses a selection between two specifications depending on C, i.e. either C is true ($\lhd$) of the input state and A is selected, or C is false ($\rhd$) of the input state and B is selected. Note that conditions C of the if and while statements send states to Prop rather than to bool. The choice of Prop for the type Spec makes writing specifications easier because unlike bool, Prop is closed under universal and existential quantification. Once Prop is chosen for specifications, it makes sense for the sake of uniformity, to also choose Prop for conditions. For example, choosing bool for conditions would require reflecting bool in Prop to write Equation (1) above. As a consequence of choosing Prop, the decidability of the conditions is not enforced by typing. However, the predicative semantics of the if-then-else and while statements implicitly address this decidability issue since whenever such statements are defined on a state s, it follows that $(C\ s\ \vee\ \neg C\ s)$ is true.

Sequence. For sequential composition, we need a notion of composition for specifications. We define below the *angelic* and *demonic* composition of specifications.

Definition 1 (Sequential composition of specifications). *Let S_1 and S_2 be of type Spec. The angelic and demonic composition operators are respectively defined as follows:*
$S_1 \boxdot S_2 \overset{\text{def}}{=} \lambda\ s\ s' \Rightarrow\ \exists\ s_x \cdot S_1\ s\ s_x\ \wedge\ S_2\ s_x\ s'$
$S_1 \,\square\, S_2 \overset{\text{def}}{=} \lambda\ s\ s' \Rightarrow\ (S_1 \boxdot S_2)\ s\ s'\ \wedge\ \forall\ s_x \cdot S_1\ s\ s_x\ \to \exists\ s' \cdot S_2\ s_x\ s'$

Angelic composition is just relational composition. Demonic composition is relational composition further restricted to account for the fact that the interpretation of $\langle S_1 \rangle;\langle S_2 \rangle$ is defined only on those states s such that S_2 is defined for all possible outputs of S_1 on s. As an illustration of the difference between the two operators, consider the following example:

$$\{(1,2),(1,3)\} \boxdot \{(2,4)\} = \{(1,4)\},\text{ but } \{(1,2),(1,3)\} \Box \{(2,4)\} = \{\}$$

We see that angelic composition does not properly capture the possibility of failure because the input 1 should not be present in the domain of the composition since it may cause an error if the output is 3. Therefore, as in [9], we use demonic composition [5,4] to formalize the sequential composition of specifications. Consequently, the specification associated to the sequential composition of statements is the demonic composition of their interpretations. The $\Box$ notation for demonic composition is from [9]. For angelic composition, we use a similar notation rather than (;) to prevent confusion with sequential composition of statements.

Iteration. The while statement is interpreted as a least fixpoint (lfp for short) of a function from specifications to specifications. An encoding of the lfp in type theory is given below. It says that a pair $(s, s') \in \mathsf{lfp}(F)$ iff (s, s') is in all specifications X such that $(F\ X\ \subseteq X)$.

$$\mathsf{lfp}\ (F : \mathsf{Spec} \to \mathsf{Spec}) \overset{\text{def}}{=} \lambda\ s\ s' \Rightarrow \forall\ X \cdot (\forall\ s\ s' \cdot F\ X\ s\ s' \to X\ s\ s') \to X\ s\ s'$$

Demonic composition is right-monotonic w.r.t. inclusion of specifications. As a consequence, the function $(\lambda\ \mathcal{X} \Rightarrow (\llbracket S \rrbracket \Box \mathcal{X}) \lhd C \rhd \llbracket \textbf{\textsf{skip}} \rrbracket)$ is monotonic. Hence, by the Knaster-Tarski fixpoint theorem, its least fixpoint is defined since the binary predicates ordered by implication (i.e. relations ordered by inclusion) form a complete lattice. We have formalized enough fixpoint theory to prove that lfp indeed represents the least fixpoint. An interesting by-product of this least fixpoint definition of the while statement is the existence of the following well-founded relation associated to each while statement:

$$\prec_P^C \quad \overset{\text{def}}{=} \quad \lambda\ s\ s' \ \Rightarrow\ C\ s' \land \llbracket P \rrbracket\ s'\ s \land (\exists\ s'' \cdot\ \llbracket \textbf{\textsf{while}}\ C\ \textbf{\textsf{do}}\ P\ \textbf{\textsf{done}} \rrbracket\ s'\ s'')$$

This relation has been used to proceed by well-founded induction in proving many intermediate results and some of the subsequent theorems.

4.2 The refinement relation

The refinement relation occupies unsurprisingly a central place in our development. Intuitively, it corresponds to a kind of translation of the classical relational interpretation (as found in e.g. [23]) in predicative terms.

Definition 2 (Predicative refinement). *We say that S_2 refines S_1 if and only if whenever S_1 terminates on some state s, S_2 terminates on s and all observable behaviors of S_2 on s are observable behaviors of S_1 on s:*

$$\begin{aligned} S_2 \sqsubseteq S_1 \quad \overset{\text{def}}{=} \quad \forall\ s \cdot (\exists\ s' \cdot \llbracket S_1 \rrbracket\ s\ s')\ \to \quad & (\forall\ s' \cdot \llbracket S_2 \rrbracket\ s\ s'\ \to\ \llbracket S_1 \rrbracket\ s\ s') \\ \land\ & (\exists\ s' \cdot \llbracket S_2 \rrbracket\ s\ s') \end{aligned}$$

This definition reflects the fact that reducing non-determinism or enlarging the domain of a statement moves it down the refinement ordering. The correctness of the design rules relies on important properties of the refinement relation, some of which are stated in the following Lemma.

Lemma 1 (Properties of refinement). *Let P and Q designate statements, and let C designate a condition. The following properties hold:*
1. $P \sqsubseteq P$ *2.* $P \sqsubseteq Q \ \rightarrow\ Q \sqsubseteq R \ \rightarrow\ P \ \sqsubseteq\ R$

3. $P_1 \sqsubseteq Q_1 \rightarrow P_2 \sqsubseteq Q_2 \rightarrow$ **if** C **then** P_1 **else** P_2 **endif**
$\sqsubseteq$ **if** C **then** Q_1 **else** Q_2 **endif**

4. $P_1 \sqsubseteq Q_1 \ \rightarrow\ P_2 \sqsubseteq Q_2 \ \rightarrow\ P_1;P_2 \ \sqsubseteq\ Q_1;Q_2$
5. $P \sqsubseteq Q \ \rightarrow$ **while** C **do** P **done** $\sqsubseteq$ **while** C **do** Q **done**

Essentially, this lemma states that the refinement relation is a preorder, and that the control structures of our language are monotonic w.r.t. refinement. Property (2) justifies the design rule for specified blocks, whereas properties (3) and (4) justify the design rules for the if statement and sequential composition. Note that these properties of refinement need never be explicitly used to discharge proof obligations. Indeed, the properties are seamlessly applied during the composition of program designs. For example, the nesting of specified blocks implicitly invokes the transitivity of refinement. Actually, the composition of program designs feels like programming, but also consists in constructing (behind the scenes) the main frame of the proof of correctness.

4.3 The special case of loops

The predicative semantics of loops corresponds to the least fixpoint of a rather complex function. Even though it is possible to prove refinements by applying the least fixpoint axioms directly, it proves to be a rather impractical method. Hence, we need another way to handle the while construct. In particular we would like to be as uniform as possible and rely on the notion of specification only. Note that Lemma 2 below provides an alternative characterization of loops as if statements under some conditions.

Lemma 2 (Loop Summarization).
well_founded $(\lambda\ s\ s' \Rightarrow C\ s' \wedge [\![P]\!]\ s'\ s \wedge C\ s)$
$\rightarrow$ (**if** C **then** P **endif; if** C **then** P **endif**) $\sqsubseteq$ **if** C **then** P **endif**
$\rightarrow$ **while** C **do** P **done** $\equiv$ **if** C **then** $\langle\ \lambda\ s\ s' \Rightarrow [\![P]\!]\ s\ s' \wedge \neg(C\ s')\ \rangle$ **endif**

This summarization lemma is inspired by the relational approach described in [9], and suggests the following design method for loops. First, one should start with a loop body satisfying the conditions of Lemma 2. This first step consists in finding an abstract specification of what the loop body is intended to achieve. Then, one applies the summarization lemma and checks that the loop summary refines the desired specification. Finally, the loop body may be further refined with a more precise implementation. In this way, the lfp definition never appears explicitly in refinement statements to prove. This method rests on the following theorem, which builds on the previous Lemma to give necessary and sufficient conditions for a while statement to refine a given specification.

Theorem 2 (Loop refinement rule).

$$\begin{array}{ll} & \textsf{while}\ C\ \textsf{do}\ P\ \textsf{done}\ \sqsubseteq\ R \\ \leftrightarrow & \exists\, K\ L\ \cdot\ P\ \sqsubseteq\ L\ \wedge\ \textit{well\_founded}\ (\lambda\ s\ s' \Rightarrow C\ s' \wedge [\![L]\!]\ s'\ s \wedge C\ s) \\ & \qquad\qquad \wedge\ K = \textsf{if}\ C\ \textsf{then}\ L\ \textsf{endif}\ \wedge\ K;K \sqsubseteq K \\ & \qquad\qquad \wedge\ \textsf{if}\ C\ \textsf{then}\ \langle\ \lambda\ s\ s' \Rightarrow [\![L]\!]\ s\ s'\ \wedge\ \neg(C\ s')\ \rangle\ \textsf{endif}\ \sqsubseteq\ R \end{array}$$

Combined with properties (2) and (5) of Lemma 1, this theorem justifies the design rule for the while statement. Note that the second hypothesis of Lemma 2 implies that the relation $(\lambda\ s\ s' \Rightarrow C\ s \wedge [\![P]\!]\ s\ s')$ is transitive. This means that in general the loop body must have a non deterministic specification for loop summarization to apply. Hence, the ability to specify non deterministic behavior is key even when the end goal is a deterministic program.

Note 1. The most common way of dealing with loops is through the use of invariants and variants. In the case of the square root algorithm presented in section 2, one can prove correctness in Hoare logic using $(r^2 \leq x < h^2)$ as invariant and $(h - r)$ as variant. If we consider that the intention of the programmer is to have the loop body maintain the invariant and decrease the variant, then the corresponding specification is the following:

$$\langle\ (r^2 \leq x < h^2\ \wedge\ r'^2 \leq x' < h'^2)\ \wedge\ (h - r > h' - r')\ \rangle_x$$

This specification states that the invariant is true at the beginning of execution as well as at the end of execution, and also that the variant is lower at the end than it was at the beginning. From the invariant and the variant, one can reason to deduce that the objective is to shrink the search interval and make progress by either increasing r or decreasing h. However, the intention to implement this objective is not so well conveyed by the specification of the loop body when it is written under the invariant-variant mindset. Because specifications are more flexible, the programmer has the opportunity to convey his intentions in a more intelligible way as we did in our example of section 2. In that alternative specification of the loop body, one better sees the search interval closing up towards x.

5 Refinement in a calculus of weakest prespecifications

In this section we turn our attention to the simplification of p.o.s. Consider the typical situation where a specified block $S_1\ \{\ S_2\ \}$ is introduced. As required by the design rules, the p.a. should prompt us to prove the statement $\varphi_a(S_2) \sqsubseteq S_1$. However, the current definition of the refinement relation $\sqsubseteq$ is too primitive as a means to compute such p.o.s. This is mostly due to demonic composition, i.e. the $\square$ operator. For example, the statement $P_1;\ldots;P_n \sqsubseteq Q_1;\ldots;Q_n$ yields, after unfolding the $\sqsubseteq$ and $\square$ operators, a formula containing a profusion of existential quantifiers, and whose size is exponential in n. To avoid such a situation and simplify p.o.s, we recast the definition of the refinement relation in a calculus akin to the classical wp-calculus.

We begin by observing that in Definition 2 the right hand side (r.h.s.) of the implication is $(\kappa(\llbracket S_2 \rrbracket, \llbracket S_1 \rrbracket)\ s\ s)$, where κ is a relational operator called the *conjugate kernel* in [7] and the *the weakest prespecification* in [16,14]. We now translate this operator in a predicative form.

Definition 3 (The weakest prespecification). *Let R_1 and R_2 be two specifications. The weakest prespecification of R_2 w.r.t. R_1 is defined as follows:*
$\kappa(R_2, R_1) \overset{\text{def}}{=} \lambda\ s\ s' \Rightarrow (\forall\ s'' \cdot R_2\ s'\ s'' \ \to\ R_1\ s\ s'') \ \wedge\ (\exists\ s'' \cdot R_2\ s'\ s'')$

Consider $K = \kappa(R_2, R_1) \Box R_2$. Then the output state s' of $\kappa(R_2, R_1)$ on some input state s is such that R_2 terminates on s', and whatever R_2 does to s', the overall resulting behavior of K is a behavior of R_1. Specializing κ to the statements of our language and simplifying the resulting expressions, leads to the following specification transformer.

Definition 4 (The wpr transformer). *We define, by induction on the syntax, the function* wpr *of type* Statement → Spec → Spec *as follows:*

$$\left[\begin{array}{ll}
\mathit{wpr}(\textbf{effect}\ f,\ R) & \overset{\text{def}}{=} \lambda\ s\ s' \Rightarrow R\ s\ (f\ s') \\
\mathit{wpr}(S_1;S_2,\ R) & \overset{\text{def}}{=} \mathit{wpr}(S_1,\ \mathit{wpr}(S_2,\ R)) \\
\mathit{wpr}(\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif},\ R) & \overset{\text{def}}{=} (\mathit{wpr}(S_1,\ R)^{-1} \lhd C \rhd \mathit{wpr}(S_2,\ R)^{-1})^{-1} \\
\mathit{wpr}(\textbf{while}\ C\ \textbf{do}\ S\ \textbf{done},\ R) & \overset{\text{def}}{=} \mathit{lfp}\left(\lambda\ \mathcal{X} \Rightarrow (\mathit{wpr}(S,\ \mathcal{X})^{-1} \lhd C \rhd R^{-1})^{-1}\right) \\
\mathit{wpr}(S_1\ \{\ S_2\ \},\ R) & \overset{\text{def}}{=} \mathit{wpr}(S_2,\ R) \\
\mathit{wpr}(\langle R_2 \rangle,\ R_1) & \overset{\text{def}}{=} \kappa(R_2, R_1)
\end{array}\right.$$

where $R^{-1} \ \overset{\text{def}}{=}\ \lambda\ s\ s' \Rightarrow R\ s'\ s$

In fact $\mathsf{wpr}(S, R)$ is equivalent to $\kappa(\llbracket S \rrbracket, R)$. Also, wpr is monotonic in its second argument, therefore the least fixpoint in the definition of wpr for the while statement is defined. The wpr transformer is encoded as a Coq fixpoint definition by pattern matching on the first argument. The following theorem shows that refinement can be defined in terms of wpr. The definition is a translation of the relational definition in terms of κ from [7].

Theorem 3. $\forall\ S_1\ S_2 \cdot S_2 \sqsubseteq S_1 \quad \leftrightarrow \quad \forall s \cdot (\exists s' \cdot \llbracket S_1 \rrbracket\ s\ s') \to \mathit{wpr}(S_2, \llbracket S_1 \rrbracket)\ s\ s$

By using this wpr-based definition of refinement, we get simpler p.o.s for the same reasons that wp computes simpler p.o.s. In some way we have dealt with the $\Box$ operator on the l.h.s. of $\sqsubseteq$ (i.e. S_2 in Theorem 3). Indeed, $\mathsf{wpr}(P_1;P_2;...;P_n, R)$ simplifies to a formula whose size is linear in n after unfolding definitions, and $\mathsf{wpr}(\textbf{effect}\ f, R)$ simplifies to a formula with no additional quantifiers. Remains the r.h.s. of $\sqsubseteq$ to consider (i.e. $\llbracket S_1 \rrbracket$ in Theorem 3). We observe that:

$$\begin{array}{ll}
\llbracket(\textbf{effect}\ f);R\rrbracket & \equiv \lambda\ s\ s' \Rightarrow R\ (f\ s)\ s' \\
\llbracket(\textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif});R\rrbracket & \equiv \llbracket S_1;R\rrbracket \lhd C \rhd \llbracket S_2;R\rrbracket \\
\llbracket(S_1;S_2);R\rrbracket & \equiv \llbracket S_1;(S_2;R)\rrbracket
\end{array}$$

By recursively applying the equations above, $[\![S_1]\!]$ may in some cases simplify to a formula with no additional quantifiers. In our development these simplifications are done automatically each time a p.o. is computed. Of course the size of p.o.s may still become unmanageable. But, by refining in small steps, one also keeps the size of p.o.s in check. It is the case that the wpr transformer is not of much help when loops are involved, but thanks to Theorem 3 we can fall back on Theorem 2 to avoid the difficulties related to using directly the least fixpoint definition.

Note 2. One might wonder whether it would be possible to achieve the simplification of p.o.s by using the wp-calculus. Indeed this is possible, based on the following connection between wpr and wp from [16]: $\forall\, s \cdot \mathsf{wpr}(S, R)\ s\ s \leftrightarrow \mathsf{wp}(S, (R\ s))\ s$. However wpr is a more natural concept in some cases. For example consider the situation where we must refine $\langle R\rangle$. If we know that Q is a step towards a solution, we can compute a candidate $P = \mathsf{wpr}(Q, R)$ so that $K = \langle P\rangle;Q$ is a potential solution. Then, if indeed $K \sqsubseteq R$ we hopefully have a smaller problem to think about (namely finding an implementation for $\langle P\rangle$). If unfortunately K is not a solution then the problem cannot be solved in this way. This kind of reasoning is not so natural to achieve with the concept of wp since the latter computes a condition (as the name suggests), whereas the decomposition of R into $\langle P\rangle;Q$ requires P to be a specification.

6 Related work

In this section we put our research work in the context of mainly three areas: the notion of program design, the use of predicative and relational semantics and the more specific comparison with related Coq developments.

On program designs. A similar notion of program *development* is proposed in [21] to capture the refinement history. A development is defined as a "multiway branching tree" of refinement steps. In this language each specification statement is given an identifier, and refinement steps reference the specifications they refine by these identifiers. This is very much in the line of *literate programming* [17] with the important difference that formal specifications replace informal ones. Comparatively, our program designs are based on the notion of specified blocks introduced in [11]. Since specified blocks can be nested they naturally represent a tree of design decisions without the need for explicit identifiers: the abstract syntax tree provides enough structure to capture the relevant information. This also means that unlike in [21] we have no need to structure a specific database of program and specification fragments.

On the predicative and relational approaches. The semantics of our language of program designs is close to the predicative interpretation of programming constructs of [10,11,12], with important differences. Firstly, we follow [22] by using demonic composition to represent sequential composition, and by equating non termination, termination in an error state, and relational undefinedness instead of using a time variable or a fictitious program state to distinguish between terminating and non-terminating behaviors as in [10,15,23]. Representing non termination by undefinedness means however that we cannot for example, specify a program choosing nondeterministically to either terminate or loop forever on a given input. The second difference has to do with our refinement proof rule for loops which is more in the line of [9] where the focus is on assigning abstract specifications to loop bodies rather than on attaching outer specifications to loops as in [11]. A loop proof rule similar to ours follows from the results presented in [9], but our rule has weaker requirements and is provably complete.

On refinement in the Coq proof assistant. There are other works of mechanization of refinement theories in the Coq p.a., in particular [6] and [1], both based on the refinement calculus [20]. The goal of the development presented in [1] is to derive imperative programs by applying validated refinement rules in proof mode. As a consequence the final program design entangles the intermediate refinement steps together with their proof of correctness. The mechanization of the refinement calculus presented in [6] supports a quite expressive language (with pattern matching and structural recursion), however the language does not include features to structure the refinement steps. Our work also differs with existing approaches in the way we treat loops. In [1] one must specify loop invariants while our formalization allows to specify loop bodies as input-output relations, which is more general. In [6] one has to work with the fixpoint characterization of loops while we use a more convenient rule. Moreover, we use weakest prespecifications (wpr) instead of weakest preconditions (wp) to compute p.o.s.

7 Conclusion and future works

We have presented a formalized theory of stepwise refinement. The formalization is the result of our study of both relational and predicative points of view on stepwise refinement, which lead us to a calculus benefiting from some cross-fertilization between the two points of view. We have mechanized this formalization thus allowing for correct-by-construction imperative program design in the Coq p.a., even though the scalability of our framework is yet to be improved by extending our language with procedures and a notion of module.

One way to take this work further would be to extend this formalization to *data refinement* [13], which allows for correct transformation of data representations. One immediate application is then the possibility to refine programs to go from mathematical unbounded data structures (e.g. Peano integers) to bounded data structures (e.g. machine integers), thus permitting a more faithful translation into an actual programming language such as C, Ada or Caml.

Working directly with predicates instead of an embedding of the syntax would greatly simplify matters as this would eliminate a level of indirection. Therefore another way to pursue this work is to allow expressing imperative programs directly with predicates hence fully realizing *predicative programming*. As this would require manipulating Coq terms directly, we believe it can be done using a tool such as *Template Coq* [2], or by developing a dedicated plugin.

References

1. Alpuim, J., Swierstra, W.: Embedding the refinement calculus in coq. Science of Computer Programming **164**, 37–48 (2018)
2. Anand, A., Boulier, S., Tabareau, N., Sozeau, M.: Typed template coq–certified meta-programming in coq. In: CoqPL 2018-The Fourth International Workshop on Coq for Programming Languages. pp. 1–2 (2018)
3. Back, R.J.: A calculus of refinements for program derivations. Acta Informatica **25**(6), 593–624 (1988)
4. Backhouse, R., Van Der Woude, J.: Demonic operators and monotype factors. Mathematical Structures in Computer Science **3**(4), 417–433 (1993)
5. Berghammer, R., Zierer, H.: Relational algebraic semantics of deterministic and nondeterministic programs. T.C.S. **43**, 123–147 (1986)
6. Boulmé, S.: Intuitionistic refinement calculus. In: International Conference on Typed Lambda Calculi and Applications. pp. 54–69. Springer (2007)
7. Desharnais, J., Jaoua, A., Mili, F., Boudriga, N., Mili, A.: A relational division operator: the conjugate kernel. T.C.S. **114**(2), 247–272 (1993)
8. Dijkstra, E.: Notes on structured programming. In: Structured Programming. Academic Press (1972)
9. Frappier, M., Mili, A., Desharnais, J.: A relational calculus for program construction by parts. Science of Computer Programming **26**(1-3), 237–254 (1996)
10. Hehner, E.C.: Predicative programming part I. Communications of the ACM **27**(2), 134–143 (1984)
11. Hehner, E.C.: Specified blocks. In: Verified Software: Theories, Tools, Experiments, pp. 384–391. Springer (2008)
12. Hehner, E.C.: A practical theory of programming. Springer Science & Business Media (2012)
13. Hoare, C.A.R.: Proof of correctness of data representations. In: Programming Methodology, pp. 269–281. Springer (1978)
14. Hoare, C.A.R., He, J.: The weakest prespecification. Information Processing Letters **24**(2), 127–132 (1987)
15. Hoare, C.A.R., Jifeng, H.: Unifying theories of programming, vol. 14. Prentice Hall Englewood Cliffs (1998)
16. Josephs, M.B.: An introduction to the theory of specification and refinement. In: IBM research Report RC 12993. IBM Thomas J. Watson Research Division (1987)
17. Knuth, D.E.: Literate programming. The Computer Journal **27**(2), 97–111 (1984)
18. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer Science & Business Media (2012)
19. Mili, A.: A relational approach to the design of deterministic programs. Acta Informatica **20**(4), 315–328 (1983)
20. Morgan, C.: The refinement calculus. In: Program Design Calculi, pp. 3–52. Springer (1993)
21. Morgan, C.: The refinement calculus, and literate development. In: Formal Program Development, pp. 161–182. Springer (1993)
22. Sekerinski, E.: A calculus for predicative programming. In: International Conference on Mathematics of Program Construction. pp. 302–322. Springer (1992)
23. Spivey, J.M., Abrial, J.: The Z notation. Prentice Hall Hemel Hempstead (1992)
24. Team, T.C.D.: The coq proof assistant, version 8.8.0 (Apr 2018)
25. Wirth, N.: Program development by stepwise refinement. Communications of the ACM **14**(4), 221–227 (1971)