Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Embedding the refinement calculus in Coq

João Alpuim [a], Wouter Swierstra [b]

[a] University of Hong Kong, Hong Kong
[b] Universiteit Utrecht, Netherlands

ARTICLE INFO

Article history:
Received 17 July 2016
Received in revised form 27 March 2017
Accepted 4 April 2017
Available online xxxx

Keywords:
Refinement calculus
Coq
Predicate transformers
Free monad
Dependent types

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 [10,12,16]. The *refinement calculus* [5,17,3] 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 [17, 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 [4,13], 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 indexed containers [1,13], 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).

E-mail addresses: alpuim@cs.hku.hk (J. Alpuim), w.s.swierstra@uu.nl (W. Swierstra).

http://dx.doi.org/10.1016/j.scico.2017.04.003
0167-6423/

- 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 [17], extends Dijkstra's Guarded Command language with a new language construct for specifications. The specification [pre, 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 **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 [17] 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, $S \sqsubseteq M_0 \sqsubseteq M_1 \sqsubseteq \ldots \sqsubseteq C$. Typically, the intermediate programs, such as M_0 and M_1, mix executable code fragments and specifications.

To find such derivations, Morgan [17] 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:

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

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

Please cite this article in press as: J. Alpuim, W. Swierstra, Embedding the refinement calculus in Coq, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.04.003