On the Mechanisation of Rely-Guarantee in Coq

Nelma Moreira David Pereira Simão Melo de Sousa

Technical Report Series: DCC-2013-01
Version 1.0 January 2013

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
`http://www.dcc.fc.up.pt/Pubs/`

On the Mechanisation of Rely-Guarantee in Coq

Nelma Moreira   David Pereira [*]   Simão Melo de Sousa
`{nam,dpereira}@ncc.up.pt,desousa@di.ubi.pt`

Abstract

In this report we describe the formalisation of a simple imperative language with concurrent/parallel and atomic execution statements within the Coq theorem prover. Our formalisation includes the specification of a simple imperative programming language with statements for parallel and atomic execution of code, an underlying small-step structural semantics and a proof system that is sound with respect to such semantics. With this formalisation we give a first step towards the certified verification of shared-variable concurrent/parallel programs under the context of Cliff Jones' Rely-Guarantee reasoning and in the logic of Coq.

1 Introduction

Rely-Guarantee (RG) is one of the well established formal approaches to the verification of shared-variable parallel programs. In particular, our study and mechanisation follows very closely the work described by Coleman and Jones in [CJ07]. We note that RG was already addressed in terms of its encoding within a proof-assistant through the work of Nieto [Nie02], who mechanised a parametric version of RG, and whose proofs follow the ones previously introduced by Xu *et. al.* [XdRH97].

Therefore, in this section we describe what we believe to be the first effort to provide a complete formalisation of RG for the Coq proof assistant. We encode the programming language, its small-step operational semantics, a Hoare-like inference system, and a mechanised proof of its soundness with respect to operational semantics.

It is also important to stress that RG is the base for more recent formal systems, namely those that extend RG with Reynolds' *separation logic* [Rey02]. Recent formal systems resulting from this synergy between RG and separation logic are RGSep by Vafeiadis *et. al.* [VP07], *local rely-guarantee reasoning* by Feng *et. al.* [Fen09], and also *deny-guarantee* by Dodds *et. al* [DFPV09]. Although none of the cited works are addressed in this thesis, it is our conviction that the contribution we give with our mechanisation can be a guide for the mechanisation of the cited formal systems in the future, within the Coq proof assistant.

2 The Coq Proof Assistant

In this section we provide the reader with a brief overview of the Coq proof assistant. In particular, we will look into the definition of (dependent) (co-)inductive types, to the

[*]This work was partially funded by Funda cão para a Ciência e Tecnologia (FCT) and program POSI, and by RESCUE project PTDC/EIA/65862/2006.

implementation of terminating recursive functions, and to the proof construction process in Coq's environment. A detailed description of these topics and other Coq related subjects can be found in the textbooks of Bertot and Casterán [BC04], of Pierce *et.al.* [PCG+12], and of Chlipala [Chl11].

2.1 The Calculus of Inductive Constructions

The Coq proof assistant is an implementation of Paulin-Mohring's *Calculus of Inductive Constructions* (CIC) [BC04], an extension of Coquand and Huet's *Calculus of Constructions* (CoC) [CH88] with (dependent) inductive definitions. In rigor, since version 8.0, Coq is an implementation of a weaker version of CIC, named the *predicative Calculus of Inductive Constructions* (*p*CIC), whose rules are described in detail in the official Coq manual [The].

Coq is supported by a rich typed λ-calculus that features polymorphism, dependent types, and very expressive (co-)inductive types, which is built on the *Curry-Howard Isomorphism* (CHI) programs-as-proofs principle [How]. In CHI, a typing relation $t : A$ can either be seen as a term t of type A, or as t being a proof of the proposition A. A classical example of the CHI is the correspondence between the implication elimination rule (or *modus ponens*)

$$\frac{A \rightarrow B \qquad A}{B},$$

and the function application rule of λ-calculus

$$\frac{f : A \rightarrow B \qquad x : A}{f\,x : B},$$

from where it is immediate to see that the second rule is the same as the first, if we erase the typing information. Moreover, interpreting the typing relation $x : A$ as the logical statement *"x proves A"*, and interpreting $f : A \rightarrow B$ as *"the function f transforms a proof of A into a proof of B"*, then we conclude that the function application of the term x to f yields the conclusion *"$f\,x$ proves B"*. Under this perspective of looking at logical formulae and types, CIC becomes both a functional programming language with a very expressive type system and, simultaneously, a higher-order logic where users can define specifications about the developed programs, and build proofs that show that such programs are correct with respect to the specifications defined.

In the CIC there exists no distinction between terms and types. Therefore, all types also have their own type, called a *sort*. The set of sorts supported by CIC is the set

$$\mathcal{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}.$$

The sorts Prop and Set ensure a strict separation between *logical types* and *informative types*: the former is the type of propositions and proofs, whereas the latter accommodates data types and functions defined over those data types. An immediate effect of the non-existing distinction between types and terms in CIC is that computation occur both in programs and in proofs.

In CIC, terms are equipped with a built-in notion of *reduction*. A reduction is an elementary transformation defined over terms, and computation is simply a series reductions over a term. The set of all reductions form a confluent and strong normalising system, *i.e.*, all terms have a unique *normal form*. The expression

$$E, \Gamma \vdash t =_{\beta\delta\zeta\iota} t'$$

means that the terms t and t' are convertible under the set of reduction rules of the CIC, in a context Γ and in an environment E. In this case, we say that t and t' are $\beta\delta\zeta\iota$-convertible, or simply convertible. The reduction rules considered have the following roles, respectively: the reduction β, pronounced *beta reduction*, transforms a β-redex $(\lambda x : A.e_1)e_2$ into a term $e_1\{x/e_2\}$; the reduction δ, pronounced *delta reduction*, replaces an identifier with its definition; the reduction ζ, pronounced *zeta-reduction*, transforms a local definition of the form $\mathsf{let}\, x := e_1 \,\mathsf{in}\, e_2$ into the term $e_2\{x/e_1\}$; finally, the reduction ι, pronounced *iota reduction*, is responsible for computation with recursive functions.

A fundamental feature of Coq's underlying type system is the support for *dependent product types* $\Pi x : A.B$, which extends functional types $A \to B$ in the sense that the type of $\Pi x : A.B$ is the type of functions that map each instance of x of type A to a type of B where x may occur in it. If x does not occur in B then the dependent product corresponds to the function type $A \to B$.

2.2 Inductive Definitions and Programming in Coq

Inductive definitions are another key ingredient of Coq. An inductive type is introduced by a collection of *constructors*, each with its own arity. A value of an inductive type is a composition of such constructors. If T is the type under consideration, then its constructors are functions whose final type is T, or an application of T to arguments. Moreover, the constructors must satisfy *strictly positivity constraints* [PM93] for the shake of preserving the termination of the type checking algorithm. One of the simplest examples is the classical definition of Peano numbers:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

The definition of `nat` is not written in pure CIC, but rather in the specification language Gallina. In fact, this definition yields four different definitions: the definition of the type `nat` in the sort `Set`, two *constructors* `O` and `S`, and an automatically generated induction principle `nat_ind` defined as follows.

```
∀ P:nat → Prop, P 0 → (∀  n:nat, P n → P (S n)) → ∀ n:nat, P n.
```

The induction principle expresses the standard way of proving properties about Peano numbers, and it enforces the fact that these numbers are built as a finite application of the two constructors `O` and `S`. By means of *pattern matching*, we can implement recursive functions by deconstructing the given term and producing new terms for each constructor. An example is the following function that adds two natural numbers:

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (p + m)
  end
where "n + m" := (plus n m).
```

The `where` clause, in this case, allows users to bind notations to definitions, thus making the code easier to read. The definition of `plus` is possible since it corresponds to an exhaustive pattern-matching, *i.e.*, all the constructors of `nat` are considered, and because recursive calls are performed on terms that are *structurally* smaller than the given recursive argument. This is a strong requirement of CIC that forces all functions to be terminating. We will see ahead that non-structurally recursive functions can still be programmed within Coq via a translation into equivalent structurally decreasing ones.

More complex inductive types can be defined, namely inductive definitions which depend on values. A classic example is the family of vectors of length $n \in \mathbb{N}$, whose elements have a type `A`:

```
Inductive vect (A : Type) : nat → Type :=
| vnil : vect A 0
| vcons : ∀ n : nat, A → vect A n → vect A (S n)
```

As an example, the code below shows how to create the terms representing the vectors $[a,b]$ and $[c]$ of length 2 and 1, whose elements are the constructors of another inductively defined type `A`.

```
Inductive A:Type := a | b | c.

Definition v1 : vect A 2 := vcons A 1 a (vcons A 0 b (vnil A)).
Definition v2 : vect A 1 := vcons A 0 c (vnil A).
```

A natural definition over values of type `vect` is the concatenation of vectors, defined as follows:

```
Fixpoint app(n:nat)(l₁:vect A n)(n':nat)(l₂:vect A n'){struct l₁} :
 vect (n+n') :=
 match l₁ in (vect _ m') return (vect A (m' + n')) with
  | vnil ⇒ l₂
  | vcons n₀ v l₁' ⇒ vcons A (n₀ + n') v (app n₀ l₁' n' l₂)
 end.
```

Note that there is a difference between the pattern-matching construction `match` used in the addition of two natural numbers, and the one used in the concatenation of vectors: in the latter, the returning type depends on the sizes of the vectors given as arguments. Therefore, the extended `match` construction in `app` has to bind the dependent argument m' to ensure that the final return type is a vector of size $n + n'$. The computation of `app` with arguments `v1` and `v2` yields the expected result, that is, the vector $[a,b,c]$ of size 3 (since the value `2+1` is convertible to the value `3`):

```
Coq <   Eval vm_compute in app A 2 v1 1 v2.
    = vcons 2 a (vcons 1 b (vcons 0 c (vnil A)))
    : vect (2 + 1)
```

The `vm_compute` command performs the reductions within a virtual machine [GL02] ensuring a more efficient computation within Coq's environment.

2.3 Proof Construction

As we have seen, the type system behind Coq is an extended λ-calculus that does not provide built-in logical constructions, besides universal quantification and the `Prop` sort. Logical constructions are encoded using inductive definitions and the available primitive quantification. For instance, the conjunction of two propositions $A \wedge B$ is encoded through the inductive type `and`, defined as follows:

```
Inductive and (A B : Prop) : Prop :=
| conj : A → B → and A B
where "A ∧ B" := (and A B).
```

The induction principle associated to `and`, and automatically generated by Coq, is the following:

```
and_ind : ∀ A B P : Prop, (A → B → P) → A ∧ B → P
```

Disjunction is encoded in a similar way, with two constructors, each corresponding to a each of the branches of the disjunction. Negation is defined as a function mapping a proposition A into the constant `False`, which in turn is defined as the inductive type with no inhabitants. The constant `True` is encoded as an inductive type with a single constructor `I`. Finally, the existential quantifier $\exists\, x : \texttt{T}, \texttt{P}(x)$ is defined through the following inductive definition:

```
Inductive ex (A:Type) (P : A → Prop) : Prop :=
| ex_intro : ∀ x:A, P x → ex P
```

The inductive definition `ex` enforces that we have to provide a *witness* that the predicate `P` verifies the property expected on the term x, in the spirit of constructive logic, where connectives are seen as functions taking proofs and producing new proofs.

The primitive way of the Coq proof construction process is to explicitly build CIC terms. However, proofs can be built more conveniently and interactively in a backward fashion through a language of commands called *tactics*. Although tactics are commonly used when a user is in the *proof mode* of Coq, activated by the `Theorem` command (and similar commands), they can also be used to interactively construct programs. However, that must be done carefully, since tactics may produce undesirably large terms. Let us take a look at an example of constructing a simple proof of the commutativity of the conjunction of two propositions A and B. First, we need to tell Coq that we are going to enter in the proof mode, by using the `Theorem` command.

```
Coq < Theorem and_comm :
Coq <    forall A B:Prop,
Coq <       A /\ B -> B /\ A.
1 subgoal

  ============================
   forall A B : Prop, A /\ B -> B /\ A
```

The first part of the proof is to move the universally quantified propositions `A` and `B` to the context, together with the hypothesis `A ∧ B`:

```
Coq <   intros A B H.
1 subgoal

  A : Prop
  B : Prop
  H : A /\ B
  ============================
   B /\ A
```

Next, we deconstruct the hypothesis `H` and obtain isolated terms `A` and `B` holding in the current proof context. This is achieved by the `destruct` tactic:

```
Coq <   destruct H.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   B /\ A
```

Now that we know that both `A` and `B` hold, we have to deconstruct the goal in order to isolate each of the components of the conjunction. This is done by using the tactic `constructor` that applies, in this case, the unique constructor `and`, yielding two new sub-goals, one for proving `A`, and another to prove `B`.

```
Coq <   constructor.
2 subgoals

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   B
subgoal 2 is:
 A
```

To finish the proof it is enough to apply the tactic `assumption`, that looks into the context and notices that both `A` and `B` are known to be true.

```
Coq <   assumption.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   A

Coq <   assumption.
Proof completed.

Coq < Qed.

Coq < and_comm is defined
```

The command `Qed` marks the end of the proof. This command has a very important role: it checks that the term that was progressively constructed using the tactics is in fact an inhabitant of the type of the theorem that we have allegedly just proved. This allows one to develop new tactics without formal restrictions and prevents possible bugs existing in the tactics from generating wrong proof terms, since they are checked once more at the end of the proof. When using `Qed` the proof term becomes *opaque* and cannot be unfolded and subjected to reductions. In order to have the contrary behaviour, the user must use the command `Defined` instead of `Qed` to terminate the proof.

2.4 Well-founded Recursion

As pointed out earlier, all the functions defined in Coq must be terminating. The usual approach is to implement functions through the Fixpoint command and using one of the arguments as the structurally recursive argument. However, this is not possible to be employed in the implementation of all terminating functions. The usual way to tackle this problem is via an encoding of the original formulation of the function into an equivalent structurally decreasing function. There are several techniques available to address the development of non-structurally decreasing functions in Coq, which are clearly documented in [BC04]. Here we will consider the particular technique for translating a general recursive functions into a equivalent *well-founded recursive function*.

A given binary relation $\mathcal{R}$ over a set S is said to be *well-founded* if for all element $x \in S$, there exists no strictly infinite descendent sequence $(x, x_0, x_1, x_2, \dots)$ of elements of S such that $(x_{i+1}, x_i) \in \mathcal{R}$. Well-founded relations are available in Coq through the definition of the inductive predicate Acc and the predicate well_founded :

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (∀  y : A, R y x → Acc A R y) → Acc A R x

Definition well_founded (A:Type)(R:A → A → Prop) := ∀ a:A, Acc A R a.
```

First, let us concentrate in the inductive predicate Acc. The inductive definition of Acc contemplates a single constructor, Acc_intro, whose arguments ensures the inexistence of infinite R-related sequences, that is, all the elements y that are related to x must lead to a finite descending sequence, since y satisfies Acc, which in turn is necessarily finite. The definition of well_founded universally quantifies over all the elements of type A that are related by R.

The definition of Acc is inductively defined, and so it can be used as the structurally recursive argument in the definition of functions. Current versions of Coq provides two high level commands that ease the burden of manually constructing a recursive function over Acc predicates: the command Program [Soz07a, Soz07b] and the command Function [BC02]. The command Function allows the user to explicitly specify what is the recursive measure for the function being implemented. In order to give an insight on how we can use Function to program non-structurally recursive functions, we will present different implementations of the function that adds two natural numbers. A first way of implementing such a function is as follows:

```
Function sum(x:nat)(y:nat){measure id x}:nat :=
  match x with
  | 0 ⇒ y
  | m ⇒ S (sum (m-1) y)
  end.
Proof.
  abstract(auto with arith).
Defined.
```

The annotation measure id x indicates the Function command that the measure to be considered is the function id over the recursive argument x. A proof obligation is generated by the operation of Function, and discharged by the given tactic. This obligation requires a proof that x used in the recursive branch of sum is smaller than the original x under the less-than order the natural numbers. The abstract tactic takes as argument another tactic that can solve the current goal, and saves this goal as a separate lemma. The usage of abstract can be very useful, namely when the λ-term proving the goal is of considerable size, which can have severe implications during computation or type-checking.

Another way of implementing sum is by instructing Function to consider explicitly the recursive argument as being a term that proves that the relation < is well founded.

```
Function sum1(x:nat)(y:nat){wf lt x}:nat :=
  match x with
  | 0 ⇒ y
  | m ⇒ S (sum1 (m-1) y)
  end.
Proof.
 abstract(auto with arith).
 exact lt_wf.
Defined.
```

The implementation of sum is identical to the implementation of sum1, except for the annotation wf. In this case, Function yields two proof obligations: the first one is similar to the one of sum, and the second asks for a proof that the relation less-than on natural numbers is a well founded relation. Both obligations are discharged automatically due to auto tactics, with the help of know lemmas and theorems from the database arith. The third and final way of building functions using Function is by using the struct annotation. In this case, the definition will be carried out as structurally recursive function, like if it was defined using Fixpoint.

```
Function sum2(x:nat)(y:nat){struct x}:nat :=
    match x with
      | 0 ⇒ y
      | S m ⇒ S (sum2 m y)
    end.
  Proof.
    abstract(auto with arith).
    exact lt_wf.
  Defined.
```

Besides allowing more general definitions of recursive functions than the usage of Fixpoint allows, the command Function also automatically generates a fixpoint equation and an induction principle to reason about the recursive behaviour of the implemented function.

Performing reductions that involve proofs of well-founded induction with a given relation is usually an issue in Coq. Such computations may take too much time to compute due to the complexity of the proof term involved. One way to get around is to use a technique proposed by Barras, whose idea is to add sufficient Acc_intro constructors, in a lazy way, on top a Acc term, so that this term is never reached during computation. The beauty of this technique is that the resulting term is logically equivalent to the original proof of the well founded relation. The general structure of the the function is the following:

```
Variable A : Type.
Variable R : relation A.
Hypothesis R_wf : well_founded R.

Fixpoint guard (n : nat)(wf : well_founded R) : well_founded R :=
  match n with
  | 0 ⇒ wf
  | S m ⇒ fun x ⇒ Acc_intro x (fun y _ ⇒ guard m (guard m wf) y)
  end.
```

In each recursive call, when matching a term Acc x H constructed by the guard function, the reduction mechanisms will find only Acc_intro terms, instead of a complex proof term. This improves computation considerably and yields better performance for the implemented function. For exemplifying how the function `guard` can be used when using the vernacular `Function`, we present a re-implementation of `sum1` where we discharge the second proof obligation by providing the type-checker with the result of `guard`:

```
Function sum1(x:nat)(y:nat){wf lt x}:nat :=
  match x with
  | 0 ⇒ y
  | m ⇒ S (sum1 (m-1) y)
  end.
Proof.
 abstract(auto with arith).
 exact(guard 100 _ lt_wf).
Defined.
```

2.5 Other Features of Coq

There are many other features of Coq that are very useful when conducting formalisations of mathematical theories, or certified program development. Here we enumerate only the features that are more relevant to the work presented in this thesis:

- an extraction mechanism, first introduced by Paulin-Morhing [PM89], by Paulin-Morhing and Werner [PMW93], and also by Letouzey [Let04]. This mechanism allows users to extract functional programs in OCaml, in Haskell or in Scheme directly from Coq developments. Based on the distinction between informative and logical types, extraction erases the logical contents and translates the informational ones into the functional languages mentioned above;

- it supports *type classes*, which extends the concept of type class as seen in standard functional programming languages in the sense that it allows proofs and dependent arguments in the type class definition. Type classes were developed by Sozeau and Oury [SN08] without extending the underlying Coq type system and relying on *dependent records*;

- a module system developed by Chrzaszcz [Chr03] which allows users to conduct structured developments in a similar fashion to the one available in OCaml;

- a *coercion* mechanism that automatically provides a notion of sub-typing;

- a new general rewriting mechanism implemented by Sozeau [Soz09] that allows users to perform rewriting steps on terms, where the underlying equality relation is not the one primitively available in Coq.

3 Hoare Logic

In this section we review *Floyd-Hoare logic*, usually called solely Hoare logic, which resulted from the works of Floyd [Flo67] and Hoare [Hoa69]. Using Hoare logic we are able to prove a program correct by applying a finite set of inference rules to an initial program specification of the form

$$\{P\}\,C\,\{Q\}, \qquad (1)$$

such that P and Q are logical assertions and C is a program. The intuition behind such a specification, widely known as *Hoare triple* or as *partial correctness assertion* (PCA), is that if the program C starts executing in a state where the assertion P is true, then if C terminates, it will obligatorily terminate in a state where the assertion Q holds. The assertions P and Q are usually called *preconditions* and *postconditions*, respectively.

3.1 A Simple Imperative Programming Language and its Semantics

The set of inference rules of Hoare logic is tightly connected to the inductive syntax of the target programming language, in the sense that each program construction is captured by an inference rule. Here we consider a typical imperative language with assignments, two-branched conditional instructions, and *while* loops. We will denote this language by IMP. The syntax of IMP programs is inductively defined by

$$\begin{aligned} C, C_1, C_2 \ ::= \ & \mathsf{skip} \\ & \mid x \ := \ e \\ & \mid C_1 \ ; \ C_2 \\ & \mid \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \\ & \mid \mathsf{while}\ b\ \mathsf{do}\ C_1\ \mathsf{end} \end{aligned}$$

where x is a variable of the language, and e is an arithmetic expression. For the simplicity of the presentation, here we omit the language of expressions and assume that variables of IMP can have only one of two types: integers and Booleans. Programs of IMP are interpreted in a standard *small-step structural operational semantics* [Plo81], where there exists the notion of state (a set of variables and corresponding assign values) and programs are executed by means of a *evaluation function* that take *configurations* $\langle C, s\rangle$ into new configurations The expression

$$\langle C, s\rangle \Longrightarrow^\star \langle \mathsf{skip}, s'\rangle \tag{2}$$

intuitively states that operationally evaluating the program p in the state s leads to the termination of the program, in the state s', in a finite number of individual evaluation steps, guided by the syntactical structure of p. The individual evaluation rules for IMP programs are the following:

$$\frac{}{\langle\, x := e, s \,\rangle \implies \langle\, \mathsf{skip}, s[e/x] \,\rangle}\ (\textsc{Assgn})$$

$$\frac{\langle\, C_1, s \,\rangle \implies \langle\, C_1', s' \,\rangle}{\langle\, C_1; C_2, s \,\rangle \implies \langle\, C_1'; C_2, s' \,\rangle}\ (\textsc{SeqStep})$$

$$\frac{}{\langle\, \mathsf{skip}; C_2, s \,\rangle \implies \langle\, C_2, s \,\rangle}\ (\textsc{SeqSkip})$$

$$\frac{[\![b]\!]_{\mathbb{B}}(s) \ = \ \mathbf{true}}{\langle\, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s \,\rangle \implies \langle\, C_1, s \,\rangle}\ (\textsc{IfTrue})$$

$$\frac{[\![b]\!]_{\mathbb{B}}(s) \ = \ \mathbf{false}}{\langle\, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s \,\rangle \implies \langle\, C_2, s \,\rangle}\ (\textsc{IfFalse})$$

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ C\ \mathsf{end}, s\,\rangle \implies \langle\, \mathsf{if}\ b\ \mathsf{then}\ (C\ ;\ \mathsf{while}\ b\ \mathsf{do}\ C\ \mathsf{end})\ \mathsf{else\ skip\ fi}, s\,\rangle}\ (\textsc{While})$$

The function $[\![b]\!]_{\mathbb{B}}$ is a function that denotationaly evaluates Boolean expressions in states, and returns the corresponding Boolean value for the Boolean expression given as argument. This kind of semantics, and alternative ones, can be seen in standard textbooks about programming language semantics such as [NN92, Win93, Hen90].

3.2 Hoare Logic's Proof System

Hoare logic is a proof system made up of a set of inference rules that correspond to fundamental laws about programs. Each inference rules consists of zero or more premisses and a unique conclusion.

A *deduction* assumes the form of a tree whose nodes are labeled by specifications, and whose sub-trees are deductions of the premisses of the inference rules applied to the nodes.The leafs of deduction trees are nodes to which no more inference rules can be applied, and the root of the tree is the specification of the correctness of the program under consideration. These trees are usually named as *proof trees*, or *derivation trees* and represent a correctness proof of a program.

The Hoare logic inference rules for proving the partial correctness of IMP programs are the following:

Skip: the following rule simply states that the preconditions and postconditions of a skip program must be the same, since this command does not change the state of computation.

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}}\ (\text{HL-}\textsc{Skip})$$

Assignment: if we want to show that the assertion P holds after the assignment of the expression e to the variable x, we must show that $P[e/x]$ (the substitution of the free occurrences of x by e, in P) holds before the assignment. We will apply this rule backwards. We know P and we wish to find a precondition that makes P true after the assignment $x := e$.

$$\frac{}{\{P[e/x]\}\, x := e\, \{P\}}\ (\text{HL-}\textsc{Assgn})$$

Composition: if C_1 brings a state satisfying P into a state satisfying Q', and that if C_2 brings a state satisfying Q' into a state satisfying Q, then if P is true, the execution of C_1 followed by C_2, brings the program into a state satisfying the postcondition Q.

$$\frac{\{P\}\, C_1\, \{Q'\} \qquad \{Q'\}\, C_2\, \{Q\}}{\{P\}\, C_1; C_2\, \{Q\}}\ (\text{HL-}\textsc{Seq})$$

Conditional: if b is true in the starting state, then C_1 is executed and Q becomes true; alternatively, if b is false, then C_2 is executed. The preconditions are enforced depending on whether b is true or false. This additional information is often crucial for completing the respective sub-proofs.

$$\frac{\{b \wedge P\}\, C_1\, \{Q\} \qquad \{\neg b \wedge P\}\, C_2\, \{Q\}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\, C_1\, \mathsf{else}\, C_2\ \mathsf{fi}\ \ \{Q\}}\ (\text{HL-}\textsc{If})$$

While: given an invariant P which must be true in each iteration of the while loop, then when b is false, that is, when the loop condition fails, the invariant must be true, no matter how many times the loop has been repeated before.

$$\frac{\{b \wedge P\}\, C\, \{P\}}{\{P\}\ \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}\ \ \{\neg b \wedge P\}}\ (\text{HL-}\textsc{While})$$

Weakening: if we have proved that $\{P'\}\, C\, \{Q'\}$, and if we also know that P that implies P', and that Q' implies Q, then we can strengthen the precondition and weaken the postcondition.

$$\frac{P \to P' \qquad \{P'\}\, C\, \{Q'\} \qquad Q' \to Q}{\{P\}\, C\, \{Q\}}\ (\text{HL-}\textsc{Weak})$$

We denote this proof system by HL. We say that a Hoare triple $\{P\}\, C\, \{Q\}$ is *derivable* in HL, and write $\vdash_{\mathsf{HL}}\ \{P\}\, C\, \{Q\}$ if we can build a proof tree for the triple $\{P\}\, C\, \{Q\}$ using the previous rules. We may also have a derivation in the presence of a set of assumption $\mathcal{A}$ and we write $\mathcal{A} \vdash_{\mathsf{HL}}\ \{P\}\, C\, \{Q\}$. Side conditions are introduced by the usage of the (HL-Weak) rule in the derivation process. This rule allows to relate external first-order assertions with the local specifications.

Proof trees can be constructed by a special purpose algorithm called *verification condition generator* (VCGen) [Dij75], which uses specific heuristics to infer the side conditions from one particular derivation. The input for a VCGen algorithm is a Hoare triple, an the output is a set of first-order proof obligations. For this to happen, the inference rules of the proof system must be changed so that the following conditions always hold:

1. assertions occurring in the premisses must be sub-formulas of the conclusion, so that discovering intermediate assertions is required;

2. the set of inference rules must be unambiguous in order for the derivation tree construction process can be syntax-oriented.

Instead of HL, we can consider an alternative Hoare proof system that is syntax directed and that enjoys the sub-formula property. We consider a version of IMP with annotated commands, defined by following grammar:

$$\begin{aligned} C, C_1, C_2 \ ::= &\ \mathsf{skip} \\ &\mid x\ :=\ e \\ &\mid C_1\ ;\ \{P\}\ C_2 \\ &\mid \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \\ &\mid \mathsf{while}\ b\ \mathsf{do}\ \{I\}\ \ C\ \mathsf{end}. \end{aligned}$$

The set of rules of the considered proof system, which we denote by HLa, is the following:

$$\frac{P \rightarrow Q}{\{P\}\ \mathsf{skip}\ \{Q\}}\ (\text{HL-AnnSkip})$$

$$\frac{P \rightarrow Q[e/x]}{\{P[e/x]\}\ x := e\ \{Q\}}\ (\text{HL-AnnAssgn})$$

$$\frac{\{P\}\ C_1\ \{Q'\} \qquad \{Q'\}\ C_2\ \{Q\}}{\{P\}\ C_1;\ \{Q'\}\ C_2\ \{Q\}}\ (\text{HL-AnnSeq})$$

$$\frac{\{b \wedge P\}\ C_1\ \{Q\} \qquad \{\neg b \wedge P\}\ C_2\ \{Q\}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \ \{Q\}}\ (\text{HL-AnnIf})$$

$$\frac{b \rightarrow I \qquad \{I \wedge b\}\ C\ \{I\} \qquad I \wedge \neg b \rightarrow Q}{\{P\}\ \mathsf{while}\ b\ \mathsf{do}\ \{I\}\ C\ \mathsf{end}\ \{Q\}}\ (\text{HL-AnnWhile})$$

The system HLa can be proved to infer the same proof trees as the system HL. Such proof is available in the work of Frade and Pinto [FP11], as well as the treatment of extensions to the underlying programming language and the formal treatment of such extensions at the level of the corresponding proof systems.

In recent years, the particular subject of program verification by means of Hoare logic and related concepts has evolved considerably, mainly in terms of tool support, such as the Why system [BFMP11, BFM$^+$12] and the Boogie system [dBBGdR06, DR05].

4 Rely-Guarantee Reasoning

The first formal system that addressed the specification and verification of parallel programs was the one developed by Owiki and Gries [OG76]. In their approach, a sequential proof had to be carried out for each parallel process, which also had to incorporate information that established that each sequential proof does not interfere with the other sequential proofs. This makes the whole proof system non-compositional, as it depends on the information of the actual implementation details of the sequential processes. The inference rule for this approach is summarised as follows:

$$\frac{\{P_1\}C_1\{Q_1\} \qquad \{P_2\}C_2\{Q_2\} \qquad \begin{array}{c} C_1 \text{ does not interfere with } C_2 \\ C_2 \text{ does not interfere with } C_1 \end{array}}{\{P_1 \wedge P_2\}\mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}\{Q_1 \wedge Q_2\}}$$

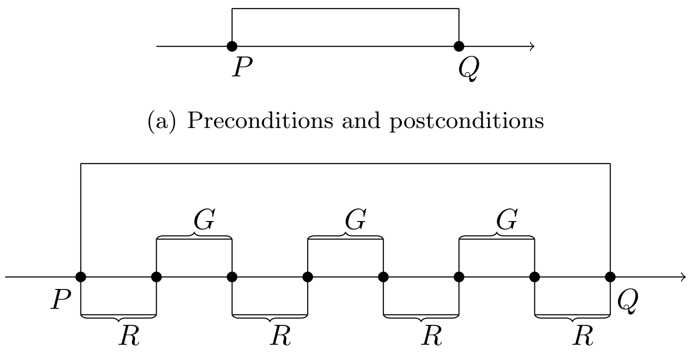
Based on the previous system, Jones introduced RG in his PhD thesis [Jon81], which resulted in a formal approach to shared-variable parallelism that brings the details of interference into specification, in an explicit way. In RG, besides preconditions and postconditions, specifications are enriched with *rely conditions* and *guarantee conditions*: a rely condition models steps of execution of the environment; a guarantee condition describes steps of execution of the program. Therefore, in the context of parallel program specification and design, the rely conditions describes the level of interference that the program is able to tolerate from the environment, while the guarantee conditions describes the level of interference that the program imposes on the environment. From the specification point-of-view, the rely condition can be seen as a way of requiring the developer to make the necessary assumptions about the environment in which the program is going to execute. From the user point-of-view, it is the responsibility of the user itself to ensure that the environment complies with the previous assumptions.

As a final remark, note that when the program's computation is described by rely and guarantee conditions, which are to be decomposed during the proof construction process, the result of such decomposition can only have at least the same level of interference as their parent conditions, that is, they cannot produce more interference. Still, and from the logical point of view, these decompositions may be weakened or strengthen, but they still must comply with the conditions from which they have originated. The point here is that weakening or strengthening decomposed rely and guarantee conditions may allow to establish a larger number of environments where the complete parallel program may be deployed.

Preconditions and Postconditions *v.s.* Rely and Guarantee Conditions

The difference between rely and guarantee conditions and preconditions and postconditions can be stated in the following way: preconditions and postconditions view the complete execution of the underlying program as a whole, whereas rely and guarantee conditions analyse each possible step of the execution, either resulting from the interference of the environment, or by a step of computation of the program. This is captured graphically in Figure 1, borrowed from Coleman and Jones [CJ07].

Figure 1 Rely and guarantee *vs.* preconditions and postconditions.



(a) Preconditions and postconditions

(b) Rely and guarantee conditions.

5 The IMPp Programming Language

IMPp is a simple parallel imperative programming language that extends IMP, which was already introduced. The IMPp language extends IMP by introducing an instruction for the atomic execution of programs, and also one instruction for parallel execution of programs. Moreover, it also considers lists of natural numbers as part of the datatypes primitively supported.

As in IMP, the language IMPp considers a language of arithmetic expressions and a language of Boolean expressions. We denote these languages of expressions by AExp and

BExp, respectively, and we inductively defined them by the following grammars:

$$\mathsf{AExp} \ni e, e_1, e_2 \ ::= \ x \mid n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2,$$

$$\mathsf{BExp} \ni b, b_1, b_2 \ ::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b_1 \wedge b_2 \mid e_1 = e_2 \mid e_1 < e_2,$$

where x is a variable identifier and l is a list of natural numbers. The operators hd and tl correspond to the head and the tail of a given list l, respectively. The Boolean operator is_cons has the goal of checking if the input list l is empty or not. The language of IMPp programs is inductively defined by

$$\begin{array}{rcl} \mathsf{IMPp} \ni C, C_1, C_2 & ::= & \mathsf{skip} \\ & | & x ::= e \\ & | & \mathsf{atomic}(C') \\ & | & C_1; C_2 \\ & | & \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \\ & | & \mathsf{while}\ b\ \mathsf{do}\ C\ \mathsf{done} \\ & | & \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, \end{array}$$

where x is a variable, $e \in$ AExp, and $b \in$ BExpr. The program C' that is the argument of the atomic instruction must be a sequential program built using the grammar of IMP programs. Given the following definition of variable identifiers (borrowed from Pierce's *et. al* [PCG+12]),

```
Inductive id : Type := Id : nat → id.
```

the syntax of IMPp is defined in Coq as follows:

```
Inductive aexp : Type :=
| ANum : nat → aexp
| AId : id → aexp
| APlus : aexp → aexp → aexp
| AMinus : aexp → aexp → aexp
| AMult : aexp → aexp → aexp.

Inductive bexp : Type :=
| BTrue : bexp
| BFalse : bexp
| BEq : aexp → aexp → bexp
| BLt : aexp → aexp → bexp
| BNot : bexp → bexp
| BAnd : bexp → bexp → bexp.

Inductive stmt : Type :=
| Stmt_Skip : stmt
| Stmt_Ass : id → aexp → stmt
| Stmt_Seq : stmt → stmt → stmt
| Stmt_If : bexp → stmt → stmt → stmt
| Stmt_While : bexp → stmt → stmt
| Stmt_Atom : stmt → stmt
| Stmt_Par : stmt → stmt → stmt.

Notation "'skip'" := Stmt_Skip.
Notation "x ':=' e" := (Stmt_Ass x e).
Notation "C1 ; C2" := (Stmt_Seq C1 C2).
Notation "'while' b 'do' C 'end'" := (Stmt_While b C).
Notation "'if' b 'then' C1 'else' C2 'fi'" := (Stmt_If b C1 C2).
Notation "'par' C1 'with' C2 'end'" := (Stmt_Par C1 C2).
Notation "'atomic(' C ')'" := (Stmt_Atom C).
```

6 Operational Semantics of IMPp

IMPp programs are evaluated by means of a small-step operational semantics, in the style of Plotkin's *structural operational semantics* [Plo81]. The semantic must be small-step in order to capture a fine-grained interleaving between the computation of the program under consideration, and the interference caused by other parallel processes running in the environment. Formally, the semantics of IMPp is a relation

$$\overset{c}{\Longrightarrow} : \langle \mathsf{IMPp}, \Sigma \rangle \to \langle \mathsf{IMPp}, \Sigma \rangle \tag{3}$$

between pairs $\langle C, s\rangle$, called configurations, such that C is a IMPp program, and s is a state (set of mappings of variables to values). The set of all states is denoted by Σ. The type of values that are supported by the semantics, and the notion of state (and the particular case of the empty state) are defined in Coq as follows:

```
Definition val := nat.

Definition st := id → val.

Definition empty_st : st := fun _ ⇒ 0.
```

Moreover, we defined the update of a variable in a state in the following way:

```
Definition upd (s : st) (x:id) (e : val) : st :=
  fun x':id ⇒ if beq_id x x' then x else s x'.
```

Before giving the structure of the relation $\overset{c}{\Longrightarrow}$ we describe the interpretation of arithmetic and Boolean expressions.We can define a recursive function that evaluates arithmetic expressions into their final result in type val. Such a function is defined as follows:

```
Function aeval (s : st) (e : aexp) {struct e} : val :=
  match e with
  | ANum n ⇒ n
  | AId x ⇒ s x
  | APlus e1 e2 ⇒ VNat ((asnat (aeval s e1)) + (asnat (aeval s e2)))
  | AMinus e1 e2 ⇒ VNat ((asnat (aeval s e1)) - (asnat (aeval s e2)))
  | AMult e1 e2 ⇒ VNat ((asnat (aeval s e1)) * (asnat (aeval s e2)))
  end.
Notation "⟦e⟧𝔼(s)" := (aeval s e).
```

The same approach is taken for Boolean expressions.

```
Function beval (s : st) (b : bexp){struct b} : bool :=
 match b with
 | BTrue ⇒ true
 | BFalse ⇒ false
 | BEq e1 e2 ⇒
   if beq_nat (asnat (aeval s e1)) (asnat (aeval s e2)) then true else false
 | BLt e1 e2 ⇒
   if blt_nat (asnat (aeval s e1)) (asnat (aeval s e2)) then true else false
 | BNot b1 ⇒ negb (beval s b1)
 | BAnd b1 b2 ⇒ andb (beval s b1) (beval s b2)
 end.
Notation "⟦b⟧𝔹(s)" := (beval s b).
```

Given the interpretation functions for arithmetic and Boolean expressions, we can now describe the relation that captures the computation of an IMPp program, C starting in some state s. The relation $\overset{c}{\Longrightarrow}$ is inductively as follows:

$$\frac{}{\langle x ::= e, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{skip}, s[e/x]\rangle}\ (\textsc{Assgn})$$

$$\frac{\langle C, s\rangle \overset{\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle}{\langle \mathsf{atomic}(C), s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle}\ (\textsc{Atomic})$$

$$\frac{}{\langle \mathsf{skip}; C, s\rangle \overset{c}{\Longrightarrow} \langle C, s\rangle}\ (\textsc{Seq-1})$$

$$\frac{\langle C_1, s\rangle \overset{c}{\Longrightarrow} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \overset{c}{\Longrightarrow} \langle C_1'; C_2, s'\rangle}\ (\textsc{Seq-2})$$

$$\frac{[\![b]\!]_{\mathbb{B}}(s) = \mathsf{true}}{\langle \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \overset{c}{\Longrightarrow} \langle C_1, s\rangle}\ (\textsc{If-true})$$

$$\frac{[\![b]\!]_{\mathbb{B}}(s) = \mathsf{false}}{\langle \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \overset{c}{\Longrightarrow} \langle C_2, s\rangle}\ (\textsc{If-false})$$

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ C\ \mathsf{done}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{if}\ b\ \mathsf{then}\ C; (\mathsf{while}\ b\ \mathsf{do}\ C\ \mathsf{done})\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, s\rangle}\ (\textsc{While})$$

$$\frac{\langle C_1, s\rangle \overset{c}{\Longrightarrow} \langle C_1', s'\rangle}{\langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{par}\ C_1'\ \mathsf{with}\ C_2\ \mathsf{end}, s'\rangle}\ (\textsc{Par-1})$$

$$\frac{\langle C_2, s\rangle \overset{c}{\Longrightarrow} \langle C_2', s'\rangle}{\langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2'\ \mathsf{end}, s'\rangle}\ (\textsc{Par-2})$$

$$\frac{}{\langle \mathsf{par}\ \mathsf{skip}\ \mathsf{with}\ \mathsf{skip}\ \mathsf{end}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{skip}, s\rangle}\ (\textsc{Par-end})$$

The above set of rules is encoded in Coq through the following inductive predicate, considering that the definition `(stmt*st)` stands for the type of configurations:

```
Inductive cstep : (stmt * st) → (stmt * st) → Prop :=
|CS_Ass: ∀ s x e,
  cstep ((x ::= e),s) (skip,s [aeval s e/x])
|CS_Atom : ∀ C s s',
    star _ (step) (C,s) (skip,s') → cstep (atomic(C),s) (skip,s')
|CS_SeqStep : ∀ s C1 C1' s' C2,
  cstep (C1,s) (C1',s') → cstep ((C1;C2),s) ((C1';C2),s')
|CS_SeqFinish : ∀ s C2,
  cstep ((skip;C2),s) (C2,s)
|CS_IfFalse : ∀ s C1 C2 b,
  ¬b2assrt b s → cstep (if b then C1 else C2 fi,s) (C2,s)
|CS_IfTrue : ∀ s C1 C2 b,
  b2assrt b s → cstep (if b then C1 else C2 fi,s) (C1,s)
|CS_While : ∀ s b C ,
  cstep (while b do C end,s) (if b then (C;while b do C end) else skip fi,s)
|CS_Par1 : ∀ s C1 C1' C2 s' ,
  cstep (C1,s) (C1',s') → cstep (par C1 with C2 end,s) (par C1' with C2 end,s')
```

```
|CS_Par2 : ∀ s C₁ C₂ C₂' s',
  cstep (C₂,s) (C₂',s') → cstep (par C₁ with C₂ end,s) (par C₁ with C₂' end,s')
|CS_Par_end : ∀ s,
  cstep (par skip with skip end,s) (skip,s).

Infix "⟹c" := cstep.
```

The constructors that form the `cstep` correspond to the reduction rules presented before, in the exact same order of occurrence. In the example that follows, we show a reduction performed by applying the relation $\overset{c}{\Longrightarrow}$.

Example 1. *Let x_1 and x_2 be two variables. Let s be the state such that the values of x_1 and x_2 are 0. Let C be the* IMPp *program defined as follows:*

$$\mathsf{par}\ \{\ x_1\ := \ 1\ \}\ \mathsf{with}\ \{\ x_2\ := \ 2\ \}\ \mathsf{end}$$

*Two reductions may occur from $\langle\, C, s\,\rangle$: either it reduces by the rule (*PAR-1*) and updates s by mapping the value 1 to the variable x_1, that is,*

$$\langle\, \mathsf{par}\{x_1\ := \ 1\}\mathsf{with}\{x_2\ := \ 2\}\mathsf{end}, s\,\rangle \overset{c}{\Longrightarrow} \langle\, \mathsf{par}\ \{\ \mathsf{skip}\ \}\ \mathsf{with}\ \{\ x_2\ := \ 2\ \}\ \mathsf{end}, s[1/x_1]\,\rangle,$$

*or it reduces by rule (*PAR-2*) and updates s by mapping 2 to the variable x_2, that is,*

$$\langle\, \mathsf{par}\{x_1\ := \ 1\}\mathsf{with}\{x_2\ := \ 2\}\mathsf{end}, s\,\rangle \overset{c}{\Longrightarrow} \langle\, \mathsf{par}\ \{\ x_1\ := \ 1\ \}\ \mathsf{with}\ \{\ \mathsf{skip}\ \}\ \mathsf{end}, s[2/x_2]\,\rangle.$$

7 Reductions Under Interference

The semantics of the relation $\overset{c}{\Longrightarrow}$ is not enough to capture possible interference between parallel programs. In order to capture it adequately, we need an extended notion of transition between configurations $\langle\, C, s\,\rangle$ that takes into account a possible preemption of C by an external program. If this is the case, then the resulting configuration must keep C unchanged, but the state s may be subject to an update, caused exactly by the interference of that external program's operation. Formally, we consider a new relation between configurations as follows

$$\langle C, s\rangle \overset{R}{\Longrightarrow} \langle C', s'\rangle \overset{\text{def}}{=} (\langle C, s\rangle \overset{c}{\Longrightarrow} \langle C', s'\rangle) \vee (C = C' \wedge (s, s') \in R), \qquad (4)$$

such that R is a relation on states that determines if a state can change into another state by the interference of the environment. The relation (4) is encoded in Coq as follows:

```
Definition interf R :=
     fun cf cf':stmt*st ⇒ (fst cf) = (fst cf') ∧ R (snd cf) (snd cf').

Definition prog_red R :=
    fun cf cf':stmt*st ⇒ cf ⟹c cf' ∨ interf R cf cf'.
```

Example 2. *Let s be a state such that the variable x_1 has the value 1 and the variable x_2 has the value 1 also. Let R be the rely condition defined by $(x, x+1) \in R$, that is, that tolerates that the environment can increase the value of a given variable by 1, and let C be the* IMPp *program defined as follows:*

$$x_1 := 1; x_2 := 2$$

The following are the two possible reductions of the configuration $\langle\, C, s\,\rangle$, considering interference:

$$\langle x_1 := 1; x_2 := 2, s\rangle \Longrightarrow \langle \mathsf{skip}; x_2 := 2, s[1/x_1]\rangle$$

or

$$\langle x_1 := 1; x_2 := 2, s\rangle \Longrightarrow \langle x_1 := 1; x_2 := 2, s[2/x_2]\rangle.$$

Now that we already have a definition for one step of computation of a program under interference, we extend it to a finite number of reductions, thus capturing the behaviour of computations. This is tantamount to the reflexive and transitive closure of $\overset{R}{\Longrightarrow}$, that is,

$$\langle C, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle. \tag{5}$$

Obviously, we may also consider a predetermined number of computations, in order to analyse just a fixed number of steps of reduction under interference, instead of considering all the possible computations. In Coq we have the following definitions for fixed number of reductions, and also for any finite number of reductions:

```
Inductive starn (A:Type)(R:relation A) : nat → A → A → Prop :=
|starn_refl : ∀ x:A,
   starn A R 0 x x
|starn_tran : ∀ x y:A,
   R x y → ∀ (n:nat)(z:A), starn A R n y z → starn A R (S n) x z.

Definition prog_red_n R n :=
  fun cf cf′ ⇒ starn (stmt*st) (prog_red R) n cf cf′.
Notation "cf =R^n=> cf′" := (prog_red_n R n cf cf′).

Inductive star (A:Type)(R:relation A) : A → A → Prop :=
| star_refl : ∀ x:A, star A R x x
| star_trans : ∀ x y:A, R x y → ∀ z:A, star A R y z → star A R x z.

Definition prog_red_star R :=
  fun cf cf′ ⇒ star (stmt*st) (prog_red R) cf cf′.
Notation "cf =R*=> cf′" := (prog_red_star R cf cf′).
```

Example 3. *Let us consider the same initial state and program as in the previous example. We are able to prove that, after four reductions, the computation of C leads to a state where the variable x_1 contains the value 2. This property is obtained by first performing three reductions leading the configuration $\langle\, C, s\,\rangle \overset{c^3}{\Longrightarrow} \langle\, \mathsf{skip}, s[1/x_1][2/x_2]\,\rangle$. With one more reduction, and because the relation R is defined as $(x, x+1) \in R$, we can prove $\langle\, \mathsf{skip}, s[1/x_1][2/x_2]\,\rangle \overset{R}{\Longrightarrow} \langle\, \mathsf{skip}, s[1/x_1][2/x_2][2/x_1]\,\rangle$.*

8 A Proof System for Rely-Guarantee

In this section we introduce an inference system for proving the partial correctness of IMPp programs along the lines of RG. This system, which we name HL-RG, extends sequential Hoare logic with a notion of interference that is explicit at the specification level. Let R be a relation establishing the rely condition, and let G be a relation establishing the guarantee condition. A triple in HL-RG has the form of

$$\{R, P\}\; C\; \{Q, G\},$$

and we shall write

$$\vdash\; \{R, P\}\; C\; \{Q, G\}$$

if we can prove from the set of inference rules of HL-RG that the program C is partially correct with respect to its rely and guarantee conditions, and also with respect to its preconditions and postconditions.

Before presenting the inference system HL-HG, we must introduce the notion of *stability*. In RG, we say that an assertion P is *stable* with respect to the interference of the environment, captured by a relation R, if P is not changed due to the actions caused by R, that is

$$\mathsf{stable}\ R\ P \overset{\text{def}}{=} \forall s, s' \in \Sigma,\ P(s) \to (s, s') \in R \to P(s'). \tag{6}$$

The particular effect of stability conditions on preconditions and postconditions can be described as follows: if P is a precondition of the program C that is satisfied, and if the environment R acts upon P, then P remains satisfied, allowing in this way to "move" the satisfiability of P into the state where the actual execution of C starts; the same applies to a postcondition Q, that is, if Q is stable with respect to R, then Q can me moved into the state where the program C has finished its execution. We now present the definition of stability in Coq. But for that, we introduce first the definitions of assertions.

```
Definition assrt := st → Prop.

Definition b2assrt (b:bexp) : assrt :=
   fun s:st ⇒ beval s b = true.

Definition assg_subs (x:id) (e:aexp) (Q:assrt) : assrt :=
   fun s:st ⇒ Q (s [(aeval s e)/x]).

Definition assrtT (b: bexp) : assrt :=
   fun s:st ⇒ b2assrt b s.

Definition assrtF (b: bexp) : assrt :=
   fun s:st ⇒ ¬b2assrt b s.

Definition assrtAnd (P Q:assrt) : assrt :=
   fun s:st ⇒ P s ∧ Q s.

Definition assrtOr (P Q: assrt) : assrt :=
   fun s:st ⇒ P s ∨ Q s.

Definition assrtImp (P Q: assrt) : Prop :=
   ∀ s:st, P s → Q s.
```

With the previous definitions, we can define the notion of stability in Coq, as follows:

```
Definition stable (R:relation st)(P:assrt) :=
   ∀ x y:st, P x ∧ R x y → P y.

Lemma stable_starn :
   ∀ n:nat, ∀ R P, stable R P → stable (starn _ R n) P.

Lemma stable_star :
   ∀ R P, stable R P → stable (star _ R) P.

Lemma stable_and :
   ∀ R₁ R₂ P, stable R₁ P → stable R₂ P → stable (rstAnd R₁ R₂) P.

Lemma stable_impl :
   ∀ R₁ R₂ P, stable R₂ P → rstImp R₁ R₂ → stable R₁ P.
```

We now give a simple example that shows an assertion being stable with respect to a possible rely condition.

Example 4. *Let R be a relation defined by $R \stackrel{\text{def}}{=} \{(x, x+k) \mid k > 0\}$, and let $P(s) \stackrel{\text{def}}{=} s > 0$. It is easy to see that the assertion P is stable with respect to R since if we know that $P(x)$ holds, then x must be a positive number, and due to the action of R, we obtain $P(x+k)$ which is also true.*

Inference Rules

We will now describe each of the inference rules of the HL-RG inference system. This system extends sequential Hoare logic by adding two new rules, one for each of the commands that extends IMPp with respect to IMP. Moreover, the rules for the sequential part, as well as the rules for atomic execution and parallel execution of commands are enriched with the stability conditions required for the rules to be sound. In Coleman and Jone's presentation, such stability rules are implicit, but when conducting the development in a proof system like Coq, the stability conditions must be made explicit. We now introduce the inference rules of the HL-RG proof system.

Skip. In the case of the skip command, there exists no program reductions. Thus, only the environment R can change the underlying state, and so, for the precondition P and the postcondition Q, the hypotheses must establish their stability with respect to R. The inference rule for skip is defined as follows:

$$\frac{\mathsf{stable}\ R\ P \qquad \mathsf{stable}\ R\ Q \qquad P \to Q}{\{R, P\}\ \mathsf{skip}\ \{Q, G\}}\ (\text{HG-Skip})$$

Assignment. In the case of assignment, the environment R may cause interference in the precondition P or the postcondition Q, but it does not affect the execution of the assignment. Moreover, it must be known in advance that the change in the state due the assignment must satisfy the guarantee condition G. The inference rule for the assignment is defined as follows:

$$\frac{\begin{array}{l}\mathsf{stable}\ R\ P \\ \mathsf{stable}\ R\ Q \qquad (\forall s \in \Sigma, (s, s[e/x]) \in G) \qquad P \to Q[e/x]\end{array}}{\{R, P\}\ x ::= e\ \{Q, G\}}\ (\text{HG-Assgn})$$

Sequence. In the case of the sequential composition of programs C_1 and C_2, we need to prove that C_1 behaves correctly with respect to its specification and, if that is the case, we have to prove that C_2 respects the same condition, considering that the postcondition of C_1 becomes the precondition of C_2. The inference rule for the composition of programs is defined as follows:

$$\frac{\{R, P\}\ C_1\ \{Q', G\} \qquad \{R, Q'\}\ C_2\ \{Q, G\}}{\{R, P\}\ C_1; C_2\ \{Q, G\}}\ (\text{HG-Seq})$$

Conditional choice. In the case of the conditional statement, as long as the specifications for the statements of the branches are given, we can prove the correct specification of the whole conditional. Still, the assertion stating the result of evaluating the Boolean guard must be immune to the interference of the environment. With the stability ensured, there is no risk that the interference of the environment breaks the expected flow of the execution of the program. The inference rule for the conditional choice command is defined as follows:

$$\frac{\begin{array}{ccc}\mathsf{stable}\ R\ [\![b]\!] & & \{R, P \wedge b\}\ C_1\ \{Q, G\} \\ \mathsf{stable}\ R\ [\![\neg b]\!] & \mathsf{stable}\ R\ P & \{R, P \wedge \neg b\}\ C_2\ \{Q, G\}\end{array}}{\{R, P\}\ \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \{Q, G\}}\ (\text{HG-IF})$$

Loops. In the case of while loops, the classic inference rule of Hoare logic is extended with stability conditions, in a similar way as in the conditional rule. The environment may interfere with the Boolean guard, and so stability must be ensured in order to preserve the correct evaluation of loops. Moreover, stability must also apply to the invariant. The inference rule for the while loop is defined as follows:

$$\frac{\mathsf{stable}\ R\ [\![b]\!]_{\mathbb{B}} \qquad \mathsf{stable}\ R\ [\![\neg b]\!]_{\mathbb{B}} \qquad \{R, b \wedge P\}\ C\ \{Q, G\}}{\{R, P\}\ \mathsf{while}\ b\ \mathsf{do}\ C\ \mathsf{done}\ \{\neg b \wedge P, G\}}\ (\text{HG-WHILE})$$

Atomic execution. Atomic statement execution ensures that a given program executes with no interference of the environment whatsoever. Hence, the rely condition in this case is the identity relation, here denoted by ID. Moreover, the command C that is going to be executed atomically must be a valid sequential program, and the precondition and postcondition can still suffer interference from the environment, hence they must be proved stable with respect to the global rely condition R. The inference rule for the atomic execution of programs is defined as follows:

$$\frac{\begin{array}{cc}\mathsf{stable}\ R\ P & \\ \mathsf{stable}\ R\ Q & \{ID, P\}\ C\ \{Q\}\end{array}}{\{R, P\}\ \mathsf{atomic}(C)\ \{Q, G\}}\ (\text{HG-ATOMIC})$$

Consequence. The consequence rule is just a simple extension of the consequence rule in sequential Hoare logic, where the rely and guarantee conditions R and G can be strengthened or weakened. The inference rule for the consequence is defined as follows:

$$\frac{\begin{array}{ccc}P \to P' & R \to R' & \\ Q' \to Q & G' \to G & \{R', P'\}\ C\ \{Q', G'\}\end{array}}{\{R, P\}\ C\ \{Q, G\}}\ (\text{HG-CONSEQ})$$

Parallel composition. In the case of parallel composition of two programs C_1 and C_2 we assume that the specifications of the individual programs ensure that they not interfere with each. Hence, the hypotheses must contain evidences that the guarantee condition of one of the component programs becomes part of the environment of the other component program, and vice versa. The adequate stability conditions for both the component programs are also required. The inference rule for the parallel composition of programs is defined as follows:

$$\frac{\begin{array}{ccc}(G_l \cup G_r) \to G & & \\ (R_l \cup G_l) \to R_r & \mathsf{stable}\ (R_r \cup G_r)\ Q_1 & \\ (R_r \cup G_r) \to R_l & \mathsf{stable}\ (R_l \cup G_l)\ Q_2 & \\ (R_l \wedge R_r) \to R & \mathsf{stable}\ (R_r \cup G_r)\ P & \{R_l, P\}\ C_1\ \{Q_1, G_l\} \\ (Q_l \wedge Q_r) \to Q & \mathsf{stable}\ (R_l \cup G_l)\ P & \{R_r, P\}\ C_2\ \{Q_2, G_r\}\end{array}}{\{R, P\}\ \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}\ \{Q, G\}}\ (\text{HG-PAR})$$

In COQ, we define the inference system HL-RG by the following inductive predicate:

```
Inductive triple_rg (R G:StR) : assrt → stmt → assrt → Prop :=
| RSkip: ∀ (P Q:assrt),
   Reflexive G → stable R P → stable R Q → P[→]Q →
   triple_rg R G P skip Q

| RAsgn : ∀ v a P Q,
   stable R P → stable R Q → (∀  s, G s (upd s v (aeval s a))) →
   (∀  s, P s → Q (upd s v (aeval s a))) →
   triple_rg R G P (v ::= a) Q

| RAtom : ∀ P Q c b,
   (∀  x y, star _ G x y → G x y) →
   stable R P → stable R Q → triple G P c Q →
   triple_rg R G P (atomic c end) Q

| RIf: ∀ P c1 c2 Q b,
   Reflexive G →
   stable R (assrtT b) → stable R (assrtF b) →
   stable R P →
   triple_rg R G (assrtAnd (assrtT b) P) c1 Q →
   triple_rg R G (assrtAnd (assrtT b) P) c2 Q →
   triple_rg R G P (ifb b then c1 else c2 fi) Q

| RSequence: ∀ c1 c2 P K Q,
   Reflexive G →
   triple_rg R G P c1 K → triple_rg R G K c2 Q →
   triple_rg R G P (c1;c2) Q

| RConseq: ∀ R' G' P P' Q Q' c,
   assrtImp P P' → assrtImp Q' Q →
   rstImp R R' → rstImp G' G →
   triple_rg R' G' P' c Q' →
   triple_rg R G P c Q

| RLoop : ∀ P b c,
   Reflexive G → stable R P →
   stable R (assrtT b) → stable R (assrtF b) →
   triple_rg R G (assrtAnd (assrtT b) P) c P →
   triple_rg R G P (while b do c end) (assrtAnd (assrtT b) P)

| RConcur : ∀ Rl Rr Gl Gr P Q1 Q2 Q cr cl,
  Reflexive Gl → Reflexive Gr →
  rstImp R (rstAnd Rl Rr) → rstImp (rstOr Gl Gr) G →
  rstImp (rstOr Rl Gl) Rr → rstImp (rstOr Rr Gr) Rl →
  assrtImp (assrtAnd Q1 Q2) Q →
  stable (rstOr Rr Gr) Q1 → stable (rstOr Rl Gl) Q2 →
  stable (rstOr Rr Gr) P → stable (rstOr Rl Gl) P →
  triple_rg Rl Gl P cl Q1 →
  triple_rg Rr Gr P cr Q2 →
  triple_rg R G P (par cl with cr end) Q.
```

In the specification of the `RAtom` constructor, we use as premise the term `triple`. This represents a valid deduction tree using the sequential Hoare proof system, which we proved correct with respect to the sequential fragment of IMPp, but that we do not present it in this dissertation. The proof of the soudness of sequential Hoare logic captured by `triple` follows along the lines of the works of Leroy [Ler10] and Pierce *at al.* [PCG$^+$12], and is based also in a small-step reduction semantics.

9 Soundness of HL-RG

We will now proceed with the proof of soundness of HL-RG in the Coq proof assistant. Following along the lines of our reference work, Coleman and Jones [CJ07], we prove the soundness of the system with respect to its functional correctness, *i.e.*, that preconditions and postconditions are ensured, and we also prove that it satisfies the constraints imposed by the guarantee condition. Together, these lead to the correctness of the whole proof system.

Here we do not describe in detail the actual Coq scripts that were required to build the proofs. Instead, we provide proof sketches that indicate the way those scripts were constructed.

Respecting Guarantees

In RG, the role of the guarantee condition is to bound the amount of interference that a program may impose in the environment. In particular, the guarantee condition of a program is part of the rely condition of all the other programs running in parallel with it. Therefore, if the configuration $\langle C, s \rangle$ reduces to a configuration $\langle C', s' \rangle$ after some finite number of steps, and if $\langle C', s' \rangle \overset{c}{\Longrightarrow} \langle C'', s'' \rangle$ holds, then we must show that $(s', s'') \in G$, where G is the established guarantee condition. Proving all such reductions that occur along the execution of C ensures that the complete computation of C satisfies the constraints imposed by G. The satisfaction of this property was introduced by Coleman and Jones in [CJ07], and is formally defined by

$$\begin{gathered}\mathsf{within}(R, G, C, s)\\ \overset{\text{def}}{=}\\ \forall C' s',\ (\langle C, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle) \to \forall C'' s'',\ (\langle C', s'\rangle \overset{c}{\Longrightarrow} \langle C'', s''\rangle) \to (s', s'') \in G.\end{gathered} \tag{7}$$

An important consequence of the previous definition is that given two programs C and C', we can prove that the states resulting from their parallel computation are members of the set of states that result from the reflexive and transitive closure of the rely and guarantee conditions of each other. Formally, we have

$$\forall C_1\, C_2\, C_1'\, s\, s',\ (\langle\, \mathsf{par}\, C_1\, \mathsf{with}\, C_2\, \mathsf{end}, s \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{par}\, C_1'\, \mathsf{with}\, C_2\, \mathsf{end}, s' \,\rangle) \to (s, s') \in (R \cup G_l)^\star \tag{8}$$

$$\forall C_1\, C_2\, C_2'\, s\, s',\ (\langle\, \mathsf{par}\, C_1\, \mathsf{with}\, C_2\, \mathsf{end}, s \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{par}\, C_1'\, \mathsf{with}\, C_2'\, \mathsf{end}, s' \,\rangle) \to (s, s') \in (R \cup G_r)^\star, \tag{9}$$

where G_l and G_r are, respectively, the guarantee conditions of the programs C_1 and C_2. These properties will be fundamental for proving the soundness of the parallel computation inference rule for the RG proof system HL-RG. Other properties of within are the following: if a reduction of the program exists, or one step of interference occurs, then the within still holds, that is,

$$\forall s\ s',\ (\langle C, s\rangle \overset{c}{\Longrightarrow} \langle C', s'\rangle) \to \mathsf{within}(R, G, C, s) \to \mathsf{within}(R, G, C', s')$$

and

$$\forall s\, s',\ (s, s') \in R \to \forall C,\ \mathsf{within}(R, G, C, s) \to \mathsf{within}(R, G, C, s').$$

The previous properties are naturally extended to a finite set of reductions under interference starting from a configuration $\langle C, s\rangle$. Formally,

$$\forall s\ s',\ (\langle C, s\rangle \xRightarrow{R^\star} \langle C', s'\rangle) \rightarrow \mathsf{within}(R, G, C, s) \rightarrow \mathsf{within}(R, G, C', s'). \tag{10}$$

Another property of interest and that we will need for the soundness proof of the parallel statement is that if $\mathsf{within}(R, G, C, s)$ holds and if the $\langle C, s\rangle$ reduces to $\langle C', s'\rangle$ then this reduction can be interpreted as a finite series of intermediate steps, where each step is captured either by R – meaning that the environment has interveen – or by G – meaning that a program reduction occurred. Formally, this notion is expressed as follows:

$$\forall s\ s',\ (\langle C, s\rangle \xRightarrow{R^\star} \langle C', s'\rangle) \rightarrow \mathsf{within}(R, G, C, s) \rightarrow (s, s') \in (R \cup G)^\star. \tag{11}$$

Soundness Proof

The soundness of HL-RG requires the notion of *Hoare validity*. In classic Hoare logic we state this condition as follows: if a program C starts its computation in a state where the precondition P holds then, if C terminates, it terminates in a state where the postcondition Q holds. In the case of parallel programs, this notion must be extended to comply with the rely and guarantee conditions. Thus, the validity of a specification $\{R, P\}\ C\ \{Q, G\}$, which we write $\models\ \{R, P\}\ C\ \{Q, G\}$, has the following reading: if a program C starts its computation in a state where the precondition P holds and if the interference of the environment is captured by the rely condition R then, if C terminates, it terminates in a state where the postcondition Q holds, and also all the intermediate program reduction steps satisfy the guarantee condition G. Formally, the definition of Hoare validity for HL-RG is defined as

$$\begin{gathered}\models\ \{R, P\}\ C\ \{Q, G\}\\ \overset{\text{def}}{=}\\ \forall C\ s, P(s) \rightarrow \forall s', (\langle C, s\rangle \xRightarrow{R^\star} \langle \mathsf{skip}, s'\rangle) \rightarrow Q(s') \wedge \mathsf{within}(R, G, C, s).\end{gathered} \tag{12}$$

The soundness of the proof system goes by induction on the size of the proof tree, and by case analysis on the last rule applied. Since the proof system is encoded as the inductive type `tripleRG`, a proof obligation is generated for each of its constructors. For each constructor C_i in the definition of type `tripleRG` a proof obligation of the form

$$\vdash\ \{R, P\}\ C_i\ \{Q, G\}\ \rightarrow \models\ \{R, P\}\ C_i\ \{Q, G\},$$

is generated. This means that we have to prove

$$\vdash\ \{R, P\}\ C_i\ \{Q, G\} \rightarrow\ \forall s,\ P(s) \rightarrow \forall s',\ (\langle C_i, s\rangle \xRightarrow{R^\star} \langle \mathsf{skip}, s'\rangle) \rightarrow Q(s') \tag{13}$$

and also

$$\vdash\ \{R, P\}\ C_i\ \{Q, G\} \rightarrow (\forall s,\ P(s) \rightarrow \forall s',\ (\langle C_i, s\rangle \xRightarrow{R^\star} \langle \mathsf{skip}, s'\rangle) \rightarrow \mathsf{within}(R, G, C_i, s)). \tag{14}$$

We call to (13) the *Hoare part* of the proof, and we call to (14) the *Guarantee part* of the proof, respectively.

Skip

The statement skip produces no reduction. Therefore, the only transition available to reason with is the environment, which satisfies the rely relation.

Hoare part. From $\langle \mathsf{skip}, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ we know that $(s, s') \in R^\star$ and, from the stability of Q with respect to R, we obtain $Q(s)$ from $Q(s')$. The rest of the proof trivially follows from the hypothesis $P \to Q$.

Guarantee part. From the definition of within and from $P(s)$ for some state s, we know that $\langle \mathsf{skip}, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ and $\langle \mathsf{skip}, s\rangle \overset{c}{\Longrightarrow} \langle C, s'\rangle$ for some state s'. This is, however, an absurd since no reduction $\langle \mathsf{skip}, s\rangle \overset{c}{\Longrightarrow} \langle C, s'\rangle$ exists in the definition of $\overset{c}{\Longrightarrow}$.

Assignment

The assignment is an indivisible operation which updates the current state with a variable x containing a value given by an expression e. The precondition P and the postcondition Q can be streched by the interference of the environment R due to the stability conditions. This only happens right before, or right after the execution of the assignment.

Hoare part. By induction on the length of the reduction $\langle x ::= e, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ we obtain $\langle \mathsf{skip}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{skip}, s[e/x]\rangle$ and $\langle \mathsf{skip}, s[e/x]\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$. Since skip implies the impossibility of reductions, we are able to infer that $(s[e/x], s') \in R^\star$. By the stability of the postcondition, we obtain $Q(s[e/x])$ from $Q(s')$.

In what concerns the case where the environment causes interference, we prove by induction on the length of $\langle x ::= e, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ that exists $s'' \in \Sigma$ such that $(s, s'') \in R$ and $\langle x ::= e, s''\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$. By the stability of the precondition $P(s)$, we conclude $P(s'')$. From the induction hypothesis, we conclude that $Q(s')$.

Guarantee part. For proving the guarantee satisfaction, *i.e.*, to prove $\mathsf{within}(R,G,x ::= e,s)$, we first obtain that if $\langle x ::= e, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle$ then both $C' = \mathsf{skip}$ and $s' = s[e/x]$ must hold. Hence, we conclude $(s, s[e/x]) \in G$ by the hypotheses.

Sequence

For the conditional statement, the proof follows closely the proof that is constructed to prove the soundness of the inference rule in the case of sequential Hoare logic, for both the cases of the Hoare part and the guarantee part.

Hoare part. The proof goes by showing that since we have the reduction

$$\langle\, C_1; C_2, s \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s' \,\rangle$$

for $s, s' \in \Sigma$, then there exists an intermediate state $s'' \in \Sigma$ such that $\langle\, C_1, s \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'' \,\rangle$ and $\langle\, C_2, s'' \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s' \,\rangle$ hold. Using $\langle\, C_1, s \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'' \,\rangle$ and the induction hypotheses, we show that the postcondition of C_1 is the precondition of C_2, and by $\langle\, C_2, s'' \,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s' \,\rangle$ we obtain the postcondition of C_2, which finishes the proof.

Guarantee part. For proving $\mathsf{within}(R, G, C_1; C_2, s)$ we need the following intermediate lemma:

$$\begin{array}{c}\mathsf{within}(R, G, C_1, s)\\ \to \\ (\forall s' \in \Sigma,\ \langle C_1, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle \to \mathsf{within}(R, G, C_2, s')) \to \mathsf{within}(R, G, C_1; C_2, s).\end{array} \tag{15}$$

By applying (15) to $\mathsf{within}(R,G,C_1;C_2,s)$, we are left to prove first that $\mathsf{within}(R, G, C_1, s)$, that is immediate from the hypothesis $\models \{R, P\}\ C_1\ \{Q', G\}$. From the same inductive hypothesis, we obtain $Q'(s')$, where $s' \in \Sigma$ is the state where C_1 finishes its execution. For the second part of the proof, which corresponds to prove that

$$\forall s' \in \Sigma,\ \langle C_1, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle \to \mathsf{within}(R, G, C_2, s'),$$

we obtain $Q'(s')$, and from $Q'(s')$ and $\models \{R, Q'\}\ C_2\ \{Q, G\}$ we obtain $\mathsf{within}(R, G, C_2, s')$, which closes the proof.

Conditional

For the conditional statement, the proof follows closely the proof that is constructed to prove the soundness of the inference rule in the case of sequential Hoare logic, for both the cases of the Hoare part and the guarantee part.
Hoare part. The proof goes by induction on the structure of the reduction

$$\langle\, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$$

and by case analysis in the value of the guard b. For the cases where no interference occurs, the proof follows immediately from the hypothesis. When interference occurs, we use the stability of the guard b with respect to the rely condition R, which keeps the evaluation of b unchanged. Once this is proved, the rest of the proof follows also from the hypotheses.
Guarantee part. In order to prove

$$\mathsf{within}(R, G, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s),$$

we require the following auxiliary lemmas:

$$\mathsf{within}(R, G, C_1, s) \to [\![b]\!]_{\mathbb{B}}(s) \to \forall C_2,\ \mathsf{within}(R, G, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s) \tag{16}$$

and

$$\mathsf{within}(R, G, C_2, s) \to \overline{[\![b]\!]}_{\mathbb{B}}(s) \to \forall C_1,\ \mathsf{within}(R, G, \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s). \tag{17}$$

Since $\mathsf{within}(R, G, C_1, s)$ and $\mathsf{within}(R, G, C_2, s)$ are already in the hypotheses, we just perform a case analysis on the value of the guard b, and directly apply (16) and (17) for each of the cases, which finishes the proof.

While Loop

The proof of the soundness of the rule for the while is obtained using adequate elimination scheme because the induction on the length of the derivation is not enough to ensure that the loop amounts to a fixpoint. The same principle needs to be used for both the Hoare part and the guarantee part of the proof.

Hoare part. To prove the soundness for the Hoare part, we first prove the validity of the following (generic) elimination scheme:

$$\begin{array}{l}\forall x : A, \forall P : A \to A \to \texttt{Prop}, (x,x) \in P \to \\ \ (\forall n\, x\, y\, z,\ (x,y) \in R \to (y,z) \in R^n \to \\ \ \ (\forall y_1\, k,\ k \leq n \to (y_1, z) \in R^k \to (y1, z) \in P) \to (x,z) \in P) \to \\ \forall x\, y,\ (x,y) \in R \to (x,y) \in P.\end{array}$$

This inductive argument states that for a predicate P to hold along a reduction defined by the closure of the relation R, then it must hold for the case (x,x) and, if after n reductions it satisfies (y,z), then for all reductions carried in less that n steps P must hold. The idea is to instantiate this elimination scheme to the case of the while loop. In order to correctly apply this predicate, we first need to transform the validity condition

```
∀ s, P(s) →
  ∀ s′, star _ (prog_red R) (while b do c end, s) (skip,s′) →
     ([⊥]b)[∧]P)(s′).
```

into its equivalent form

```
∀ s, P s →
  (∀  s’, star _ (prog_red R) (while b do c end, s) (skip,s’) →
    ∀ p p’,
      star _ (prog_red R) p p’ →
      fst p = (while b do c end) →
      fst p’ = skip →
      P (snd p) → ([⊥]b)[∧]P) (snd p’)).
```

Once we apply the elimination principle to our goal, we are left with two subgoals: the first goal considers the case where the number of reductions is zero, so we are asked to prove that `while b do c end = skip`. This goal is trivially proved by the injectivity of the constructors.

The second goal states that the current state of the program results from a positive number of reductions. Hence, we perform case analysis on the last reduction, which originates two new subgoals: one for the case when the reduction is the result of the program execution; and another when the reduction is due to the interference of the environment. In the former case, we know that the loop has reduced to a conditional instruction, which leaves us with two cases:

- if the Boolean guard is false, we are left with a reduction $\langle\, \mathsf{skip}, s\,\rangle \overset{R^n}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$, which implies that $(s, s') \in R^n$. We use the latter fact and the stability of the guard with respect to the rely condition to move the postcondition $P \wedge [\![\bar{b}]\!]_{\mathbb{B}}(s')$ to $P \wedge [\![\bar{b}]\!]_{\mathbb{B}}(s)$, which is available from the hypotheses;

- if the Boolean guard evaluates to a true, we know that the loop reduces to

  $$\langle\, C; \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s\,\rangle \overset{R^n}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle,$$

  which we decompose into

  $$\langle\, C, s\,\rangle \overset{R^m}{\Longrightarrow} \langle\, \mathsf{skip}, s''\,\rangle,$$

  and $\langle\, \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s''\,\rangle \overset{R^{(n-m)}}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$, for some $m \in \mathbb{N}$ such that $m < n$. The rest of the proof follows from simple logical reasoning with the hypotheses.

For the case where the last reduction has been performed by the interference of the environment, we first move the precondition to the state resulting from the action of the rely condition and end the proof by the same kind of reasoning used above.
Guarantee part. The proof of the satisfaction of the guarantee relation goes in a similar way as the Hoare part. Since, by the hypotheses, we know that all the program reductions of C are constrained by the guarantee condition G, then a finite composition of C is also constrained by G, which allows us to conclude that the loop satisfies the guarantee condition.

Unfortunately, we were not able to complete this proof within the Coq formalisation. Here we give the partial proof that we have obtained and show the point where we are stuck. The goal is to prove that

$$\mathsf{within}(R, G, C, s) \ \rightarrow\ \mathsf{within}(R, G, \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s),$$

regarding that we already know that $\{R, P\}\, C\, \{Q, G\}$ holds. From this last hypothesis and because we know that $\langle\, \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$, we also know that the postcondition $(P \wedge [\![\bar{b}]\!])(s')$ holds. Next, we perform case analysis on the value of the Boolean guard b, that is:

- if $[\![b]\!]_{\mathbb{B}}(s) = \texttt{false}$ then we know that $\mathsf{within}(R, G, \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s)$ can be replaced by

  $$\mathsf{within}(R, G, \mathsf{if}\, b\, \mathsf{then}\, (\mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end})\, \mathsf{else}\, \mathsf{skip}\, \mathsf{fi}, s)$$

  and from the latter statement we are able to conclude $\mathsf{within}(R, G, \mathsf{skip}, s)$ holds and that is true, as we have showed before.

- if $[\![b]\!]_{\mathbb{B}}(s) = \texttt{true}$, then we know that $\mathsf{within}(R, G, \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s)$ can be replaced by

  $$\mathsf{within}(R, G, \mathsf{if}\, b\, \mathsf{then}\, (\mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end})\, \mathsf{else}\, \mathsf{skip}\, \mathsf{fi}, s)$$

  , which in turn is equivalent to $\mathsf{within}(R, G, C; \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s)$. By the properties of the within predicate on sequential execution, we reduce the following goal to a proof that $\mathsf{within}(R, G, C, s)$ (which is immediate from the hypotheses) and to a proof that

  $$\forall n \in \mathbb{N}, \forall s'' \in \Sigma, (\langle\, C, s\,\rangle \overset{R^n}{\Longrightarrow} \langle\, \mathsf{skip}, s''\,\rangle) \rightarrow \mathsf{within}(R, G, \mathsf{while}\, b\, \mathsf{do}\, C\, \mathsf{end}, s'').$$

  At this point we got stuck in the proof, since we were not able to find an appropriate logical condition that allow us to prove it.

Consequence

Proving both the Hoare part and the guarantee part for the inference rule is pretty straightforward. The proof goes by induction on the length of $\langle\, C, s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$, and by the properties of the implication on the preconditions and postconditions, and also by the properties of the implication on the rely and the guarantee conditions.

Atomic Execution

For proving the soundness of the inference rule for the atomic execution of programs, we must show that the environment causes interference either right before, or right after the execution of program given as argument for the atomic statement. Moreover, we have to show that if $\langle\, C, s\,\rangle \overset{\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$ then $Q(s')$ and $(s, s') \in G^\star$ hold.

Hoare part. First we obtain the hypotheses gives the possible interference of the environment, and the atomic execution of C. Considering that we have

$$\langle\,\mathsf{atomic}(C), s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\,\mathsf{skip}, s'\,\rangle,$$

those hypothesis are the following:

$$(s,t) \in R^\star, \tag{18}$$

$$(t',s') \in R^\star, \tag{19}$$

$$\langle C, t\rangle \overset{ID^\star}{\Longrightarrow} \langle \mathsf{skip}, t'\rangle, \tag{20}$$

considering that $t, t' \in \Sigma$. Next, we prove that $P(t)$ can be inferred from $P(s)$ by the stability condition on (18). Moreover, we use the soundness of sequential Hoare logic to prove that from hypothesis $\{P\}\,C\,\{Q\}$ and from $P(t)$ we have $\langle\,C, t\,\rangle \overset{\star}{\Longrightarrow} \langle\,\mathsf{skip}, t'\,\rangle$. From the previous reduction we conclude that $Q(t')$ holds, and by (19) we also conclude that $Q(s')$ also holds, which finishes the proof.
Guarantee part. Considering the hypotheses (18), (19) and (20), we deduce $(t,t') \in G^\star$ using the same reasoning that we employed to obtain $Q(t')$. By the transitivity of the guarantee condition we know that $(t,t') \in G$, wich allows us to conclude that the atomic execution of C respects its guarantee condition.

Parallel Execution

To prove of the soundness of the inference rule (HG-Par) we are required to build the proofs that show that the program C_1 satisfies its commitments when executing under the interference of C_2, and the other way around.
Hoare part. The proof of the Hoare validity for a statement concerning the parallel computation of two programs C_1 and C_2 is carried in two steps. First we prove that starting with C_1 leads to an intermediate configuration where C_1 finishes and also that in that configuration, the original program C_2 has reduced to C_2'. We then prove that C_2' terminates and, by the stability condition, we stretch the postcondition of $C_1$to the same state where C_2 finished. The second phase consists in an equivalent reasoning, but starting with the execution of C_2 and using a similar approach.

The first part requires the reasoning that follows. We start the proof by obtaining a new set of hypotheses that allows us to conclude $Q_1(s')$. From the hypothesis

$$\langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s\rangle \overset{(R_1 \wedge R_2)^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle, \tag{21}$$

and from the hypotheses $\{R_1, P\}\ C_1\ \{Q_1, G_1\}$, $\{R_2, P\}\ C_2\ \{Q_2, G_2\}$, and $P(s)$ we obtain

$$\mathsf{within}(R_1, G_1, C_1, s), \tag{22}$$

$$\mathsf{within}(R_2, G_2, C_2, s), \tag{23}$$

$$(\langle C_1, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \to Q_1(s'), \tag{24}$$

$$(\langle C_2, s\rangle \overset{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \to Q_2(s'). \tag{25}$$

From (21), (22) and (23) we can conclude that there exists state s'' such that the program C_1 finishes executing in s'', and such that the program C_2' stars its execution in s'', where

C_2' is the result of the execution of C_2 under the interference of the environment, which also contains the interference caused by C_1. Hence, the following properties hold:

$$\langle C_1, s\rangle \overset{R_1^\star}{\Longrightarrow} \langle \mathsf{skip}, s''\rangle, \tag{26}$$

$$\langle C_2, s\rangle \overset{R_2^\star}{\Longrightarrow} \langle C_2', s''\rangle, \tag{27}$$

$$\langle \mathsf{par\ skip\ with}\ C_2'\ \mathsf{end}, s''\rangle \overset{(R_1\wedge R_2)\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle. \tag{28}$$

Since both (23) and (27) hold, then by (10) we conclude $\mathsf{within}(R_2, G_2, C_2', s'')$. Moreover, from $\mathsf{within}(R_2, G_2, C_2', s'')$ and (28) we also conclude

$$\mathsf{within}(R_2, G_2, \mathsf{par\ skip\ with}\ C_2'\ \mathsf{end}, s''). \tag{29}$$

To conclude this part of the proof, we need to show that $Q_1(s') \wedge Q_2(s')$. For that, we split $Q_1(s') \wedge Q_2(s')$ and show that $Q_1(s')$ holds by the stability conditions. To prove $Q_2(s')$, we reason as before, but considering that the command C_2 ends its execution first that C_1.
Guarantee part. The proof goes by applying the following property of within with respect to parallel computation:

$$\begin{gathered}\mathsf{within}(R \vee G_2, G_1, C_1, s) \to \mathsf{within}(R \vee G_1, G_2, C_2, s) \to \\ \mathsf{within}(R, G, \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s).\end{gathered} \tag{30}$$

Using (30) we are left to prove $\mathsf{within}(R \vee G_2, G_1, C_1, s)$ and $\mathsf{within}(R \vee G_1, G_2, C_2, s)$. Here we present only the proof of the former, since the proof of the later is obtained by similar reasoning.

From the hypotheses we know that $\mathsf{within}(R \vee G_2, G_1, C_1, s)$ is equivalent to

$$\mathsf{within}((R_1 \wedge R_2) \vee G_2, G_1, C_1, s).$$

From the hypotheses, we know that $\mathsf{within}(R_1, G_1, C_1, s)$ and that $(R_1 \wedge R_2) \vee G_2 \to R_1$. By the properties of the implication and the predicate within we conclude the proof.

10 Examples and Discussion

We will now analyse the effectiveness of our development of HL-RG in the verification of simple parallel programs. We start by the following example.

Example 5. *This example is a classic one in the realm of the verification of concurrency. The idea is to do a parallel assignment to a variable x initialised beforehand with some value greater of equal to* 0*. The corresponding* IMPp *program code is the following:*

$$C \overset{\text{def}}{=} \mathsf{par}\ x ::= x + 1\ \mathsf{with}\ x ::= x + 2\ \mathsf{end}.$$

The rely condition states that the value of the variable x after a reduction is greater or equal than before the reduction occurs. The guarantee is defined in the exact same way, that is,

$$R \overset{\text{def}}{=} G \overset{\text{def}}{=} \lambda s \lambda s'.[\![x]\!]_{\mathbb{N}}(s) \leq [\![x]\!]_{\mathbb{N}}(s').$$

Finally, the precondition states that initially the value of x is greater or equal to 0*, and the postcondition states that the final value of x is greater or equal to 2, that is,.*

$$P \overset{\text{def}}{=} \lambda x.[\![x]\!]_{\mathbb{N}}(s) \geq 0,$$

$$Q \overset{\text{def}}{=} \lambda x.[\![x]\!]_{\mathbb{N}}(s) \geq 2.$$

The full specification corresponds to $\{R, P\}\ C\ \{Q, G\}$ *and the first thing we do to prove this specification valid is to apply the inference rule* HG-PAR*. The application of* HG-PAR *requires us to provide the* Coq *proof assistant with the missing value, namely, the rely conditions* R_1 *and* R_2 *such that* $R_1 \wedge R_2 \rightarrow R$*; the two guarantee conditions* G_1 *and* G_2 *such that* $G_1 \vee G_2 \rightarrow G$ *and the postconditions* C_1 *and* C_2 *such that their conjuntion implies* C*. We instantiate these new variables with*

$$\begin{array}{lcllcl} R_1 & \overset{\text{def}}{=} & R & \qquad R_2 & \overset{\text{def}}{=} & R \\ G_1 & \overset{\text{def}}{=} & G & \qquad G_2 & \overset{\text{def}}{=} & G \\ C_1 & \overset{\text{def}}{=} & x \geq 1 & \qquad C_2 & \overset{\text{def}}{=} & x \geq 2. \end{array}$$

The goal associated with the stability conditions are proved in a straightforward way. Next, we prove

$$\{R_1, P\}\ x := x + 1\{Q_1, G_1\}$$

and

$$\{R_2, P\}\ x := x + 2\{Q_2, G_2\}$$

correct by applying the inference rule (HG-Assgn)*, and by simple arithmetic reasoning.*

One of the main difficulties of using RG is the definition of the rely and guarantee conditions. This dues to the fact that the rely and the guarantee conditions have describe the state of computation of the programs as a whole, which is not always easy. In order to cope with such difficulties, rules for adding "ghost" variables into the programs under consideration were introduced [XdRH97]. In the case of our reference work and in our development, this rule is not taken into consideration. Unfortunatelly, the absence of such rule is a strong constraint to the set of parallel programs that we can address with our development. Moreover, our proof system is more restricted than the our reference work. This means that, in the future, we have to rethink the design choices that we have made and find ways to improve it.

11 Related Work

We have described the formalisation, within the Coq proof assistant, of a proof system for RG, following the concepts introduced by Coleman and Jones in [CJ07]. Related work includes the work of these authors, and also the work of Prensa Nieto [Nie02, PN03] in the Isabelle proof assistant.

Our formalisation essentially confirms most of the work introduced by Coleman and Jones [CJ07], but extended with atomic execution of programs. Thus, our development shows that the ideas forwarded by these authors seem to be correct, dispite an incorrect rule they use to decompose sequential composition of programs, and also assuming that we were not able to finish the guarantee part for the inference rule for loops not because it is unsound, but just beacuse we did not yet found the correct way of doing it. Therefore, we consider that our development effort can serve as a guide for future formalisations within Coq that address other approaches to RG, or to some of its extensions.

In what concerns to the comparison of our work with the one of Nieto[Nie02, PN03], the main diferences are the following: the author formalised a notion of parameterised parallelism, that is, parallel programs are defined as a list of sequential programs. This restriction unable the specification of nested parallelism. Nevertheless, the system mechanised by Nieto contains an extra rule for introducing "ghost" variables into the original program's code and eases the recording og interference. This allowed for that work to be succesfully used to prove the correctness of larger set of examples than us.

12 Conclusions

In this report we have described the mechanisation of an Hoare-like inference system for proving the partial correctness of simple, shared-variable parallel programs. The work presented follows very closely the work of Jones and Coleman [CJ07], but ended up in a more restrictive proof system. This is manly the consequence of the set of hypothesis that are required to show that the parallel execution rule is sound with respect to the small-step semantics that we have decided to use. Still, the main goal of the work we have presented is achieved: we have decided follow this line of work in order to get a better knowledge of the difficulties that arise when formalising a proof system shared-variable parallel within the context of RG principle. In particular we have understood how the definition of the rely and guarantee conditions can be a hard job, even for very simple programs.

These programs are written in the IMPp language that extends IMP with instructions for atomic and parallel execution of programs. We mechanise a small-step operational semantics that captures a fine-grained notion of computation under interference of the environment. We have also proved the soundness of the inference system HG, which is an extension of the inference system proposed by Coleman and Jones in [CJ07] with a command for the atomic execution of programs.

Although RG has become a mature theory and is a well-known method for verification of shared-variable parallel programs, it is usually difficult to define in it rely and guarantee conditions that specify the behaviours of parallel programs over the whole execution state. Nevertheless, we believe that our formalisation that can serve as a starting step to develop more modern and suited models [VP07, Fen09, DFPV09] that handle parallelism and concurrency in a more adequate and flexible way. It is included in our list of future research topics to extend our formalisation in that way.

Another important outcome of this work is our increase in the knowledge of RG that will allow us to have a stronger base to address our next goal, which is to investigate *concurrent Kleene algebra* (CKA), an algebraic framework for reasoning about concurrent programs. In particular, we are interested in the way it handles RG reasoning.

References

[BC02] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In Victor Carreño, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 31–46. Springer, 2002.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BFM$^+$12] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform, version 0.72.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.72 edition, May 2012. `https://gforge.inria.fr/docman/view.php/2990/7919/manual-0.72.pdf`.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[Chl11] Adam Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[Chr03] Jacek Chrzaszcz. Implementing modules in the Coq system. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 270–286. Springer, 2003.

[CJ07] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

[dBBGdR06] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *LNCS*. Springer, 2006.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[DR05] Robert Deline and Rustan. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (MSR), March 2005.

[Fen09] Xinyu Feng. Local rely-guarantee reasoning. *SIGPLAN Not.*, 44(1):315–327, January 2009.

[Flo67] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[FP11] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252 – 277, 2011.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. *SIGPLAN Not.*, 37(9):235–246, September 2002.

[Hen90] Matthew Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics.* Wiley, 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[How] W.A. Howard. *The formulae-as-types notion of construction*, pages 479–490.

[Jon81] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, June 1981.

[Ler10] Xavier Leroy. Mechanized semantics. In *Logics and languages for reliability and security*, volume 25 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 195–224. IOS Press, 2010.

[Let04] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, July 2004.

[Nie02] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL.* PhD thesis, Technische Universität München, 2002. Available at `http://tumb1.biblio.tu-muenchen.de/publ/diss/allgemein.html`.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[PCG$^+$12] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations.* Electronic textbook, 2012. `http://www.cis.upenn.edu/~bcpierce/sf`.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PM89] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, 664:328–345, 1993.

[PMW93] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[PN03] Leonor Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362, 2003.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[SN08] Matthieu Sozeau and Oury Nicolas. First-Class Type Classes. *LNCS*, August 2008.

[Soz07a] Matthieu Sozeau. Program-ing finger trees in Coq. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 13–24, New York, NY, USA, 2007. ACM.

[Soz07b] Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 237–252. Springer Berlin Heidelberg, 2007.

[Soz09] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009.

[The] The Coq development team. Coq reference manual. `http://coq.inria.fr/distrib/V8.3/refman/`.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *IN 18TH CONCUR*, pages 256–271. Springer, 2007.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[XdRH97] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.