Intuitionistic Refinement Calculus

An extended version of [Bou07a] examplified on the game of Nim

Sylvain Boulmé
Laboratoire d'Informatique de Grenoble
`Sylvain.Boulmeimag.fr`

June 20, 2007

Abstract

Refinement calculi are program logics which formalize the "top-down" methodology of software development promoted by Dijkstra and Wirth in the early days of structured programming. I present here the *shallow* embedding of a refinement calculus into Coq constructive type theory. This embedding involves monad transformers and the computational reflexion of weakest-preconditions, using a continuation passing style. It should allow to reason about many ML programs combining non-functional features (state, exceptions, etc) with purely functional ones (higher-order functions, structural recursion, etc). The interest of combining higher-order functions and imperative state modifications is examplified here on the modelization of the game of Nim.

1 Introduction

The refinement calculus of [Mor90] is the kernel of the B method [Abr96] which has been successfully applied in large industrial projects [Beh99]. This paper presents the marriage of this refinement calculus with the Calculus of Inductive Constructions [Coq88, PM93], the constructive type theory of Coq [Coq04, Ber04]. This marriage is interesting for both formalisms. On the Coq side, refinement calculus provides a simple and efficient embedding of computational behaviors which are not natively available in Coq: side-effects and partial functions. Here, partial functions may involve undefined behaviors or non-termination, interpreted in partial or total correctness. On the other side, this marriage shows that all kinds of Coq computation can be fully integrated into refinement calculus: pattern-matching, structural recursion over inductive types, and higher-order functions. Moreover, because Coq is a typed lambda-calculus with types and propositions as *first-class citizens* (see Figure 1), it is more than a higher-order logic: it is also a programming language where propositions and proofs are first-class values. Hence, programming computations of WP (Weakest-Preconditions) in Coq is very easy because substitutions of variable are expressed at an abstract level. This allows to use Coq as the kernel of a refinement prover, with two benefits. First, all libraries and tools developed for Coq can be reused in refinement proofs. Second, refinement rules are formally proved. like most of the meta-theory of refinement calculus. In particular, the reflexion of WP-computations ensures their soundness with respect to purely deductive rules.

Section 2 motivates this work by the formalization of reasoning about higher-order imperative functions in Coq. It illustrates that this reasoning may combine equational reasoning in monads with Hoare logic. This leads me to introduce *Dijkstra Specification Monads* (abbreviated DSM). Section 3 defines DSM as a very simple combination of monads and lattices in Coq, where the order relation is *refinement*. Then, it presents a Tarski fixpoint theorem, which makes DSM adapted to reason about non-terminating programs. Section 4 gives the modular construction of the state DSM. This construction uses a WP calculus embedded in Coq as a CPS (Continuation-Passing Style) semantics of DSM. It is also based on the state monad transformer. Hence, it could be easily adapted for other monad transformers like exception monad transformer. With an example combining partial functions and structural recursion, Section 5 illustrates that refinement formulae may be simplified by computing WP. Then, it shows how interactive refinement proofs mix deduction and WP-computation. Section 6 explains how the state DSM is used to prove higher-order imperative programs, and in particular how Hoare logic is expressed in the state

"Type" is the type of types. In order to avoid logical paradoxes, each of its occurrence is implicitly indexed with a natural, such that in **Type**:**Type**, the left index is strictly lower than the right one.

"Prop" is the type of logical propositions. In Curry-Howard style, propositions are represented as types, and proofs as lambda-terms. Hence, if `A:`**Prop** then `A:`**Type**.

"forall `x:A,B`**"** is the type of functions "**fun** `x:A => b`" when if `x:A` then `b:B`. This type is also logically interpreted as universal quantification or as implication.

"`A -> B`" is a synonym of **forall** `(x:A),B` when `x` does not occur free in `B`.

"`A*B`" is the type of pairs "`(x,y)`" where `x:A` and `y:B`.

"`A /\ B`" represents conjunction and is like `A*B` at **Prop** level.

"exists `x:A,B`**"** represents existential quantification (defined as a dependent-pair).

"`x=y`" means that every property satisfied by `x` is satisfied by `y` (Leibniz's equality).

Figure 1: A very short description of Coq syntax

DSM. At last, Section 7 uses the state DSM to modelize the Nim game, its optimal strategy, and an implementation of this strategy which stores informations in memory that avoid to recompute the whole strategy at each turn of the game.

2 Motivations

This section details the main motivations of this paper, starting from an Ocaml example. It introduces a definition of monads in Coq (Subsection 2.2), recalls briefly their connection with Hoare logic (Subsection 2.3), and introduces the issue of representing non-terminating computations in Coq (Subsection 2.4).

2.1 Reasoning on higher-order imperative functions in Coq

Let me first introduce a higher-order imperative example in Ocaml syntax. Given `nat` the type of natural numbers and `bintree` the type of binary trees:

```
type nat = O | S of nat;;
type bintree = Leaf | Node of bintree*bintree;;
```

Given `n: nat`, and `f: bintree->unit` (`f` is an action parametrized by a binary tree), I consider below a function `enumBT` such that `(enumBT n f)` calls successively `f` over all and only binary trees of height `n`, but only once for a given tree.

The main advantage of this CPS-like implementation is to call `f` as soon as a tree is computed, before computing the next tree. Moreover, whereas the number of binary trees is exponential in function of 2^n (e.g. the size of a balanced binary tree of height n), this function requires only a memory linear in function of 2^n.

```
let rec enumBT n f = match n with
  | O -> f Leaf
  | (S p) -> enumBT p (fun l ->
                          enumBT p (fun r -> f(Node(l,r)))   ;
                          enumlt p (fun r -> f(Node(l,r)) ; f(Node(r,l))))
and enumlt n f = match n with
  | O -> ()
  | (S p) -> enumBT p f ; enumlt p f
```

Function `enumBT` is defined mutually recursively over `n` with `enumlt` which enumerates binary trees with a height strictly lower than `n`. Then, it uses the fact that in a tree of height `(S n)`, either its two

Constants of monads are:

```
K: Type -> Type.
equiv: forall (A:Type), (K A) -> (K A) -> Prop.
val: forall (A:Type), A -> (K A).
bind: forall (A B:Type), (K A) -> (A -> (K B)) -> (K B).
```

To have a lighter syntax, I require (with the commands below) that Coq infers type parameters (like `A`) of `equiv`, `val` and `bind` like a ML compiler:

```
Implicit Arguments equiv [A].
Implicit Arguments val [A].
Implicit Arguments bind [A B].
```

Hence, axioms of monads are expressed as:

```
equiv_refl: forall (A:Type) (x:(K A)), equiv x x.
equiv_sym: forall (A:Type) (x y:(K A)), (equiv x y)->(equiv y x).
equiv_trans: forall (A:Type) (x y z:(K A)), (equiv x y) -> (equiv y z) -> (equiv x z).

bind_compat: forall (A B:Type) (k1 k2:(K A)) (f1 f2: A -> (K B)),
  (equiv k1 k2) ->
     (forall (x:A), equiv (f1 x) (f2 x)) -> (equiv (bind k1 f1) (bind k2 f2)).

bind_val_l: forall (A B:Type) (a:A) (f:A->(K B)), equiv (bind (val a) f) (f a).
bind_val_r: forall (A:Type) (k:(K A)), equiv (bind k (fun a => val a)) k.

bind_assoc: forall (A B C:Type) (k:(K A)) (f: A->(K B)) (g: B -> (K C)),
    equiv (bind (bind k f) g) (bind k (fun a => bind (f a) g)).
```

Figure 2: Coq interface for monads

children have a height equal to `n`, or one of them has a height equals to `n` and the other has a height strictly lower than `n`.

In order to reason about this simple function in Coq, we may use a style inspired by Hoare logic. Unfortunately, a simple Hoare logic does not allow us to reason on such a higher-order imperative function. Indeed, it seems hard even to specify `enumBT` using pre and postconditions: these conditions are predicates over the state. Here the state is unknown (it is the state of `f` closure): it is an implicit parameter of the function. The effect of `f` on this state is also unknown: it is also an implicit parameter of the function. Hence, predicates over the state do not seem well-adapted to specify the problem, whereas it is naturally expressed as a predicate on the (finite) trace of `f` calls. Actually, we need here extensions of Hoare logic as those recently proposed in [Hon05].

Alternatively, I simply propose here to specify this program using an equality on programs. Hence, I have proved in Coq (see [Bou07b]) that for a given `n`, there exists `l` of type `(list bintree)` such that `l` contains only all binary trees of height `n` without duplicates, and such that `(enumBT n f)`$\equiv$`(iter l f)` where "`iter l f`" calls `f` on successive elements in `l`, and relation $\equiv$ is an observational equivalence on expressions. Here, the expression language is formalized as a monad.

2.2 Axioms of monads in Coq

A monad is a categorical structure expressing non-purely functional features like state or exception handling, finite non-determinism and environment interactions [Mog91, Pey93, Wad95]. My Coq axiomatization of monads is given Figure 2:

– `K` is a parametrized type such that `(K A)` represents the type of an expression returning a value of type `A`. Hence, values of the monad are any Coq values.
– `equiv` is an "observational" equivalence on expressions.
– `val` is side-effect free operator to lift a value into an expression.
– `bind` corresponds to a "let-in" construct in ML: it generalizes sequence of imperative languages. The power of this operator is to allow *any* Coq function as second argument.

Using the predefined type `unit` of single value `tt`, we say a monadic expression is an instruction if its type is `(K unit)`. The `skip` instruction is thus `(val tt)`. Sequencing instructions is just applying `bind`:

```
Definition seq (A:Type) (i1:K unit) (i2:K A): K A := bind i1 (fun _ => i2).
Implicit Arguments seq [A].
```

In the following, "`If cond Then i1 Else i2`" is a notation for:

```
  bind cond (fun b:bool => if b then i1 else i2).
```

This lifts the if-then-else construction of Coq into monad expressions. It is easy to prove that the monadic if-then-else is compatible with equivalence of expressions.

More generally, all purely functional constructions are lifted to monad expressions, such that the lifted construction is compatible with equivalence. Hence, monads support naturally pattern-matching, structural recursion, and higher-order functions. For instance, I provide below a function `forN` such that the execution of `(forN n e)` repeats successively `n` times execution of `e`. This Coq definition uses *structural recursion of naturals* for parameter `n` as indicated by the keywords **Fixpoint** and **struct**:

```
Fixpoint forN (n:nat) (e:K unit) {struct n}: (K unit)
 := match n with
    | O => skip
    | (S n) => seq e (forN n e)
    end.
```

Actually, there are two ways of lifting functions at monad expressions, implementing respectively call-by-value or call-by-name evaluation (see [Ben02]). Here, I choose the call-by-value evaluation following ML semantics. Thus, parameter `f` of `enumBT` corresponds to a value of type `bintree->(K unit)` and the Coq type for `enumBT` is `nat -> (bintree -> K unit) -> K unit` (see [Bou07b]).

2.3 Marrying monads with Hoare logic

The previous example is rather simple because `enumBT` do not modify the state directly. On other examples, we may need to specify a modification of the state as in Hoare logic. In particular, let me consider an expression `e` of type `K A` in a state monad: `e` works on a global state of type `St` (hence, the type of `St` is **Type**) and returns a result of type `A`. The state monad provides two specific operators `set: St->(K unit)`, to update the current value of the global state, and `get: (K St)`, to read the current value of the global state. These operators satisfy the following three axioms:

```
get_set: equiv (bind get set) skip.
set_get: forall (st:St), equiv (seq (set st) get) (seq (set st) (val st)).
set_set: forall (st1 st2:St), equiv (seq (set st1) (set st2)) (set st2).
```

A Hoare specification of `e` can be seen as the pair of two predicates: a *precondition* `P: St->Prop` on the initial state, and a *postcondition* `Q: St->St->A->Prop` on the initial state, the final state and the result. In total-correctness semantics, such a specification is interpreted through the following formula:

```
forall (sti:St), (P sti) ->
  exists stf:St, exists r:A,
    (Q sti stf r) /\ (equiv (seq (set sti) e) (seq (set stf) (val r))).
```

Fortunately, Dijkstra has invented a weakest-precondition calculus to simplify this tedious formula. Hence, there is a function `wp` of type:

```
wp: forall (A:Type),(K A) -> (St -> A -> Prop) -> St -> Prop.
Implicit Arguments wp [A].
```

such that the preceding tedious formula is equivalent to

```
forall (sti:St), (P sti) -> (wp e (Q sti) sti)
```

and such that `wp` is a CPS semantics of the state monad where continuations are predicates on the final state and the result (postconditions). In other words, `wp` discharges the user to infer manually the two existential connectors of the tedious formula.

This paper shows that refinement calculus is a marriage of monads and Hoare logic such that WP are hidden in the refinement order. Hence, from the user point of view, reasoning with refinement is very natural, because this order generalizes the equality of monads.

2.4 Non-termination in a logic *a la* Curry-Howard

Even in a logic for total correctness, representing programs that may not terminate is very convenient: it allows us to separate the definition of a program from its proof of termination.

However, representing non-terminating programs is not so natural in Coq. This is because Coq is based on the Curry-Howard paradigm: proofs are represented as lambda-terms. This paradigm is powerful because it provides a simple and elegant logic which allows to marry deductions and computations. But, Coq can not provide a general fixpoint operator, because such an operator would allow to build a proof of false. Thus, Coq authorizes only structural recursion with syntactic restrictions that ensure termination of all computations (however, structural recursion of Coq is powerful enough to express well-founded recursion, see [Coq04]).

Fortunately, the refinement theory presented here provides general fixpoint operators (see Section 3) and thus allows to represent non-terminating programs (see Section 6). And amazingly, in refinement proofs, a flexible marriage between deductions and terminating computations is also available (see Section 5). The trick here is that programs are not represented as elementary lambda-terms of the universe hierarchy: they are not at the level of proofs. Here, programs are represented at the level of propositions (see Section 4). More generally, refinement theory over an expression language allows us to reason very naturally about partial expressions (see examples of Section 5.1).

Formally, my refinement calculus is based on *Dijkstra Specification Monads*: a combination of monads and lattices in Coq which allows to recover fixpoints *a la* Tarski.

3 Lattice theory of Dijkstra Specification Monads

Given a particular monad M, let me explain informally how to define the *Dijkstra Specification Monad* (or DSM in short) of M. It is an extension of M with two non-deterministic operators that transform the "programming language" of M into a "specification language". DSM are special cases of monads. Thus, to distinguish expressions of M and expressions of its DSM, the former are called *programs* and the latter are called *specifications*.

Very roughly, a specification describes *a set of programs that implement it.* This set is closed under observational equivalence. A specification S_1 *refines* a specification S_2, *if and only if every implementation of* S_1 *also implements* S_2. A program is a special case of specification which is only implemented by itself (modulo observational equivalence). More generally, composition operators of M are lifted in its DSM as the closure (under observational equivalence) of their pointwise extension. The DSM of M extends its expression language with two operators called `sync` and `choice` corresponding respectively to the intersection and the union of a family of specifications.

Below, I define DSM *axiomatically.* Two formal models of DSM (DSM associated to the pure and the state monad) using weakest-preconditions are built Section 4. The axiomatization presented here is not complete with respect to these models (some properties that hold for both models can not be derived from axioms). However, this axiomatization has the interest to present an unified view of these models and is sufficiently expressive to develop fixpoint theory. Hence, fixpoint theory presented Subsection 3.2 is generic with respect to DSM models.

Hence, even if all computations of the original monad M terminate, its associated DSM is expressive enough to represent non-terminating expressions. It can thus be used to reason about programs of an extension of M with fixpoint operators. This idea is illustrated in Section 6.3.

3.1 Axioms of DSM

A DSM is a monad, that provides a preorder `refines` (reflexive and transitive) such that its associated equivalence is `equiv`. In other words, `(refines s1 s2) /\ (refines s2 s1)` must be equivalent to `equiv s1 s2`.

```
refines: forall (A:Type), (K A)->(K A)->Prop.
Implicit Arguments refines [A].
```

Furthermore, the operator `bind` must be monotonic (increasing) for this preorder. Hence, all functional features (pattern-matching, structural recursion, ...) are lifted as monotonic constructions. The expression language of monads is extended with two primitive operators:

– `any`: **forall** `(A:`**Type**`), (K A)` such that `any A` is implemented by any concrete value of the type `A`. If `A` is empty, then `any A` has no implementation.
– `sync`: **forall** `(A B:`**Type**`), (A -> (K B)) -> (K B)` (with `A` and `B` as implicit arguments) such that for any `a:A`, every implementation of `sync s` must be an implementation of `s a`. If `A` is empty, we say that `sync s` *aborts*: it is refined by any specification and refines only aborting specifications.

More formally, these operators must satisfy the following five axioms:

```
any_refined: forall (A:Type) (a:A), refines (val a) (any A).
any_refines: forall (A B:Type) (s1:A -> K B) (s2:K B),
  (forall (a:A), refines (s1 a) s2) -> refines (bind (any A) s1) s2.

sync_refines: forall (A B:Type) (s:A -> K B) (a:A), refines (sync s) (s a).
sync_refined: forall (A B:Type) (s1:(K B)) (s2:A -> (K B)),
  (forall (a:A), refines s1 (s2 a)) -> (refines s1 (sync s2)).

bind_sync_indep: forall (A B C:Type) (s1:K C) (s2:K B) (s3:B -> K C),
  (A -> refines s1 (bind s2 s3))
     -> refines s1 (bind (sync (fun _:A => s2)) s3).
```

Axioms `sync_refines` and `sync_refined` express that `sync s` is the greatest lower bound[1] of `s` (where `s` is a family of specifications indexed over `A`). Let me now explain the interest and the meaning of other axioms. First, I need to define `choice` from `bind` and `any`:

```
Definition choice (A B:Type) (s:A -> K B): (K B) := bind (any A) s.
Implicit Arguments choice [A B].
```

Combining axioms of `bind` with respectively `any_refined` and `any_refines`, I derive the following two lemmas:

```
Lemma choice_refined: forall (A B:Type) (s:A -> K B) (a:A), refines (s a) (choice s).
Lemma choice_refines: forall (A B:Type) (s1:A -> K B) (s2:K B),
  (forall (a:A), refines (s1 a) s2) -> (refines (choice s1) s2).
```

These two properties express that `choice s` is the least upper bound of `s`. There is thus a relative symmetry between `choice` and `sync` with respect to `refines`. However, as we will see now, `choice` distributes over `bind`, whereas `sync` generally does not. Indeed, if `any A` is equivalent to `choice (`**fun** `x:A => val x)`, defining `choice` from `bind` and `any` makes distributivity of `choice` over `bind` a consequence of `bind` associativity:

```
Lemma choice_bind_distr: forall (A B C:Type) (s1:A -> K B) (s2:B -> K C),
    equiv (bind (choice s1) s2) (choice (fun x:A => bind (s1 x) s2)).
```

This property is generally[2] not satisfied by `sync`. Indeed, let me define `s1` as **fun** `b:bool => val b` and `s2` as **fun** `x:bool => skip`. Applying our intuitive model, `sync s1` is the intersection of `(val true)` and `(val false)`: it represents the empty set. Thus, the left side of the refinement goal, `bind (sync s1) s2`, is empty. On the right side, **fun** `x => bind (s1 x) s2` is equivalent to **fun** `x:bool => skip`. Hence, its intersection is equivalent to `skip` and thus non-empty.

However, using `sync_refined`, monotonicity of `bind` and then `sync_refines`, I prove the weaker property:

```
Lemma sync_bind_distr_l: forall (A B C:Type) (s1: A -> (K B)) (s2: B -> K C),
    refines (bind (sync s1) s2) (sync (fun x:A => bind (s1 x) s2)).
```

Axiom `bind_sync_indep` expresses thus that when the intersection ranges over a family which is empty or reduced to a single element, then the reverse relation holds.

In particular, the following definition uses `sync` to derive a `require` operator that sets a precondition (I use here the fact that **Prop** is a subtype of **Type**). The proof of `bind_require` below uses axiom `bind_sync_indep`.

```
Definition require (P:Prop):(K unit) := sync (fun _:P => skip).
```

[1]If S_1 refines S_2, I consider that S_1 is *smaller* than S_2, according to their intuitive meaning as sets of implementations. But, in the literature about refinement, the dual view is also often considered.

[2]The distributivity property can not be falsified in the axiomatization, because we can still construct a trivial model of DSM in which this property holds.

Instruction `require P` expresses that `P` is assumed by implementations (any implementation is authorized if `P` does not hold). Using `sync` and `bind` axioms, we show that such preconditions can only be weakened in refinement.

```
Lemma bind_require: forall (A:Type) (p1 p2:Prop) (s1 s2:K A),
   (p2 -> (p1 /\ refines s1 s2)) ->
     refines (seq (require p1) s1) (seq (require p2) s2).
```

```
Lemma require_True: equiv (require True) skip.
```

Symmetrically, I introduce an operator `ensure` to set a postcondition. Instruction `ensure P` expresses that `P` is guaranteed by implementations.

```
Definition ensure (P:Prop): (K unit) := choice (fun _:P => skip).
```

```
Lemma bind_ensure: forall (A:Type) (p1 p2:Prop) (s1 s2:K A),
  (p1 -> (p2 /\ refines s1 s2)) ->
     refines (seq (ensure p1) s1) (seq (ensure p2) s2).
```

```
Lemma ensure_True: equiv (ensure True) skip.
```

3.2 Fixpoint theory of DSM

Using `sync`, I adapt here in Coq the Tarski's proof of existence of a smallest fixpoint for any monotonic function in a complete lattice. The main trick of this proof is the higher-order scheme (i.e. a specification "computing" a specification) in the definition of the smallest fixpoint operator below. Indeed, `sfix F` is defined as the intersection of all prefixpoints of `F` (i.e. `sp` such that `(F sp)` refines `sp`):

```
Definition sfix (A:Type) (F:(K A) -> (K A)) : (K A) :=
  sync (fun sp:(K A) => seq (require (refines (F sp) sp)) sp).
Implicit Arguments sfix [A].
```

I first prove that `sfix` refines every prefixpoint of `F`.

```
Theorem sfix_smallest: forall (A:Type) (F:(K A) -> (K A)) (sp:K A),
    (refines (F sp) sp) -> (refines (sfix F) sp).
```

Indeed, assuming the hypothesis `(refines (F sp) sp)` and applying `bind_require` with `p2` being `True` and then `require_True`, we derive that `seq (require (refines (F sp) sp)) sp` refines `sp`. Then, applying `sync_refines` with variable `a` being `sp`, we derive the expected goal.

Then, I prove that under the assumption that `F` is monotonic, then `sfixF F` is a fixpoint (and it is the smallest by `sfix_smallest`):

```
Theorem sfix_fix: forall (A:Type) (F:(K A) -> (K A)),
 (forall (x y:K A), refines x y -> refines (F x) (F y))
  -> equiv (sfix F) (F (sfix F)).
```

Indeed, let us assume now that `F` is monotonic. Let `x` be a prefixpoint of `F`. We deduce from property `sfix_smallest` that `F (sfix F)` refines `F x`. Then, by transitivity, `F (sfixF)` refines `x` (`x` being a prefixpoint). In other words, for all specification `x`, `F (sfixF)` refines `seq (require (refines (F x) x)) x`. Hence, applying axiom `sync_refined`, we get that `sfixF F` is a prefixpoint of `F`. As `F` is monotonic, `F (sfix F)` is also a prefixpoint of `F`. At last, applying `sfix_smallest`, we get that `sfix F` also refines `F (sfixF F)`. Hence, we have proved the expected goal.

One of the main properties directly derived from the two previous theorems is given below: given two monotonic functions `F1` and `F2`, this property expresses how to relate `sfix F2` and `sfix F1` via a monotonic function `G`. Typically, if `G` is identity, then this lemma expresses monotonicity of smallest fixpoints. Other instanciations of this lemma are given in Section 7.

```
Lemma sfix_refines_G_sfix:
 forall (A1 A2:Type) (F1: K A1 -> K A1) (F2: K A2 -> K A2) (G: K A1 -> K A2),
  (forall (x y:K A1), refines x y -> refines (F1 x) (F1 y))
 -> (forall (x y:K A1), refines x y -> refines (G x) (G y))
 -> (forall (x:K A1), (refines (F2 (G x)) (G (F1 x))))
    -> (refines (sfix F2) (G (sfix F1))).
```

Indeed, let us assume that `F1` is monotonic. Then, by `sfix_fix`, we know that `(F1 (sfix F1))` refines `(sfix F1)`. By monotonicity of `G`, we get that: `(G (F1 (sfix F1)))` refines `(G (sfix F1))`. By hypothesis and transitivity, we deduce that `(G (sfix F1))` is a prefixpoint of `F2`. Hence, by `sfix_smallest`, we conclude that `sfix F2` refines `(G (sfix F1))`.
The biggest fixpoint operator follows a symmetric construction, and has symmetric properties:

```
Definition bfix (A:Type) (F: (K A) -> (K A)) : (K A) :=
  choice (fun sp:(K A) => seq (ensure (refines sp (F sp))) sp).
```

I have also defined a notion of well-founded fixpoints for recursive functions returning a specification. These well-founded fixpoints are unique and preserve such properties as determinism and weak-termination (see [Bou06]).

4 Modular construction of the state DSM

In the previous section, DSM are defined *axiomatically*. On the contrary, this section gives particular *models* of DSM. They are used in next sections to simplify reasoning about refinement formulae.

Let me start with an intuitive model of the pure DSM, the DSM of the pure monad (i.e. the monad of purely functional expressions). In this model, a specification is defined as a pair of a *precondition* (i.e. a proposition assumed by implementations) and a *postcondition* (i.e. a predicate on the result computed by implementations). Formally, the definition below makes `K A` a product type with `Build_K` as constructor, and `pre` and `post` as projections (parameter `A` being implicit):

```
Record K (A:Type): Type := { pre: Prop ;  post: A -> Prop  }.

Definition refines (A:Type) (s1 s2:K A): Prop
 := pre s2 -> ( pre s1 /\ (forall a:A, post s1 a -> post s2 a) ).

Definition val (A:Type) (a:A): K A := Build_K True (fun b => a=b).

Definition bind (A B:Type) (s1:K A) (s2:A -> K B): (K B) :=
  Build_K (pre s1 /\ (forall a:A, post s1 a -> pre (s2 a)))
          (fun b => exists a:A, post s1 a /\ post (s2 a) b).

Definition any (A:Type): K A := Build_K True (fun a:A => True).

Definition sync (A B:Type) (s:A -> K B) : K B :=
  Build_K (exists a:A, pre (s a))
          (fun b => forall a:A, pre (s a) -> post (s a) b).
```

It is straightforward to show that the previous definitions satisfy DSM axioms. Moreover, they are fully compatible with higher-order schemes like the one used in `sfix` definition. In particular, I have carefully avoided to define `K` as an inductive type that encodes the abstract syntax of specifications. Indeed, such an inductive definition would introduce universe constraints forbidding higher-order schemes.

In conclusion, this model is intuitive and simple, but not very interesting in practice: specifications are interpreted as huge formulae, because of `bind` definition. Now, I first present a second model which is logically equivalent to the previous one. It is based on WP and produces simpler formulae. Then, I give a model for the state DSM, the DSM of the state monad.

4.1 Construction of the pure DSM from weakest-preconditions

The WP semantics improves the previous one by translating computational contents of specifications into Coq computations. Due to the presence of non-determinism, there are two notions of weakest-preconditions. They are expressed below using the pre/post semantics of specifications. Hence, let us assume a type `A`, and a specification `s` of type `(K A)`.

Given a postcondition `R` on the result of `s`, the *strong demonic weakest-precondition* of `s` for `R`, noted `(sdemon s R)`, is the necessary and sufficient condition which must hold for `R` to be satisfied however `s` is implemented. Hence, `(sdemon s)` has type `(A->Prop)->Prop` and satisfies:

```
forall R:A->Prop, (sdemon s R) <-> ((pre s) /\ forall a:A, (post s a)->(R a))
```

On the contrary, assuming that `(pre s)` holds, the *angelic weakest-precondition* of `s` for `R` is the necessary and sufficient condition which ensures that there exists an implementation of `s` satisfying `R`.

```
(pre s) -> forall R:A->Prop, (angel s R)<->(exists a:A, (post s a) /\ (R a))
```

In classical logic, these two notions can be defined dually using Excluded-Middle. For instance, we could define `(angel s R)` as `not (sdemon s (fun a => not (R a)))`. In Coq which is an intuitionistic logic, these two notions are not dual. But, a big interest of Coq is that WP are *computed* inside the logic.

Actually, to simplify the definition of `angel` and have better properties, I impose `angel` to satisfy the following property:

```
forall R:A->Prop, (angel s R) <-> (exists a:A, (post s a) /\ (R a))
```

In particular, this implies that `angel` is monotonic (like `sdemon`):

```
forall (R1 R2:A->Prop), (forall a, R1 a -> R2 a) -> (angel s R1)->(angel s R2)
```

Moreover, using the following two properties (which derive from previous definitions), `pre` and `post` can now be defined from `sdemon` and `angel`:

```
(pre s) <-> (sdemon s (fun _ => True)).
forall (a:A), (post s a) <-> (angel s (fun b => a=b)).
```

Hence, instead of defining specifications using a `pre`/`post` pair, I now define them as the following triple:

```
Record K (A:Type) : Type := {
   sdemon: (A -> Prop) -> Prop;
   angel: (A -> Prop) -> Prop;
   WPcorrect: forall R : A -> Prop,
     ( sdemon R <->
          ( sdemon (fun _ => True) /\ forall a:A, (angel (fun b => a=b)) -> R a ))
     /\ (angel R <-> (exists a:A, (angel (fun b => a=b)) /\ R a))
  }.
```

Actually, with this definition, I proved all standard properties of the WP-calculus without introducing abstract syntax for specifications.

In the following, I still continue to use `pre` and `post`, but these constructions are now derived from `sdemon` and `angel` using the previous equivalences. Refinement is directly defined by translating the pre/post semantics:

```
Definition refines (A:Type) (s1 s2:K A): Prop
 := (pre s2) -> (sdemon s1 (fun a => post s2 a)).
```

At last, I present below the definition of pure DSM operators through Coq formulae. Indeed, their `WPcorrect` component is not very human-friendly (see the real Coq code in [Bou06]). Hence, `val`, `bind` and `any` are defined such that

```
forall (A:Type) (a:A) (R:A->Prop),
  (sdemon (val a) R = R a) /\ (angel (val a) R = R a)
forall (A B:Type) (s1:(K A)) (s2:A->(K B)) (R:B->Prop),
   (sdemon (bind s1 s2) R = sdemon s1 (fun a => sdemon (s2 a) R))
/\  (angel (bind s1 s2) R = angel s1 (fun a => angel (s2 a) R))
forall (A:Type) (R:A->Prop),
  (sdemon (any A) R = forall a:A,R a) /\ (angel (any A) R = exists a:A,R a)
```

Now, I must define `sync`. But, WP of `sync` are not very simple. Thus, I derive `sync` from `require`, where `require` is satisfying:

```
forall (P:Prop) (R:unit->Prop),
  (sdemon (require P) R = (P /\ R tt)) /\ (angel (require P) R = R tt)
```

Below, `sync` is defined from other operators, assuming that `choice` and `ensure` are defined according to Section 3.1 (their definitions depend only on `bind`, `val` and `any`). This definition of `sync` is only valid in the pure DSM.

```
Definition sync (A B:Type) (s:A -> K B) :=
  seq (require (exists a:A, pre (s a)))
      (choice (fun b =>  seq (ensure (forall a, pre (s a) -> post (s a) b))
                                  (val b))).
```

These definitions satisfy the DSM axioms. Moreover, on the programming part of the language (here `val` and `bind`), `sdemon` and `angel` perform CPS computations. Examples of WP computed by Coq are given in Section 5.

4.2 Construction of the state DSM from the pure DSM

This section presents how the state DSM is derived from the pure DSM, by applying the state monad transformer. Intuitively, a monad transformer is a function from monads to monads that extends the input monad with specific constructions (see [Lia96, Ben02]). A monad transformer is simply given by a transformation on the type constructor `K`, by an embedding of the expressions of the input monads, and by a generic transformation on composition operators of the input monad. However, the properties of the specific operators of the input monad are not necessarily fully preserved in the output monad.

Hence, in our case, we have to prove that applying the state monad transformer on the pure DSM, we obtain a DSM (it is a state monad by construction). From now, constructions of the pure DSM are prefixed by "`F.`": the non-prefixed names are only used to denote constructions of the state DSM. It is easy to prove that the following definitions satisfy axioms of DSM (and of state monads):

```
Definition K (A:Type) := St -> F.K (St*A).
Definition refines (A:Type) (s1 s2:St -> F.K (St*A))
 := forall st:St, F.refines (s1 st) (s2 st).
Definition val (A:Type) (a:A): K A := fun st => F.val (st,a).
Definition bind (A B:Type) (s:K A) (f:A -> K B): K B
 := fun st => F.bind (s st) (fun p => let (stf,a):=p in (f a stf)).
Definition set (st:St): K unit := fun _ => F.val (st,tt).
Definition get: K St := fun st => F.val (st,st).
Definition any (A:Type): K A
 := fun st => F.bind (F.any A) (fun a => F.val (st,a)).
Definition sync (A B:Type) (s:A->K B) := fun st => F.sync (fun x => s x st).
```

Weakest-preconditions on the state DSM are also derived from the pure DSM:

```
Definition sdemon (A:Type) (s:K A) (R:St->A->Prop) (st:St)
  := F.sdemon (s st) (fun p => let (stf,a):=p in (R stf a)).
Definition angel (A:Type) (s:K A) (R:St->A->Prop) (st:St)
  := F.angel (s st) (fun p => let (stf,a):=p in (R stf a)).
```

This method allows me also to extend the state DSM with exception-handling, using the exception monad transformers. It could also probably be applied with many other monad transformers.

5 WP-computations in interactive refinement proofs

This section presents how to use Coq as an interactive refinement prover. Subsection 5.1 gives examples of weakest-preconditions computed by Coq. Subsection 5.2 explains how refinement proofs in the pure DSM can combine deductions and wp-computations. Then, subsection 5.3 sketches how this is extended when reasoning involves a state.

5.1 Using wp-computations to simplify proofs

In a refinement prover like B, refinement formulae are automatically translated into first-order "proof obligations" through WP-computations. This is expressed in the pure DSM by the following rule:

```
Lemma wp_refines: forall (A:Type) (s1 s2:F.K A),
 (F.pre s2 -> F.sdemon s1 (fun a => F.post s2 a)) -> F.refines s1 s2.
```

Indeed, application of this lemma replaces the current refinement goal by the formula in hypothesis, that Coq can then simplify by computing effectively "`F.pre s2`", "`F.sdemon s1`" and "`F.post s2`".

Let me run this rule on an example. First, I introduce an operator `abort` that sets a false precondition, and thus, behaves like an error-raising operator. Given a type `A`, `abort A` could be defined as `seq (require False) (any A)`. Here, in the pure DSM, I use an (observationally) equivalent definition, but with optimized WP. Hence, `F.abort` is defined such that

```
forall (A:Type) (R:A->Prop),
  (F.sdemon (F.abort A) R = False) /\ (F.angel (F.abort A) R = False).
```

Second, I introduce two function definitions:

– Function `pred` computes the predecessor of a natural n, or aborts if n is zero.

```
Definition pred (n:nat): F.K nat :=
     match n with  | 0 =>   F.abort nat
                   | (S p) => F.val p
     end.
```

– Function `minus` computes $n-m$ when $n \geq m$. If $n < m$, it aborts. It is defined by structural recursion over parameter m as indicated by **struct** keyword.

```
Fixpoint minus (n m:nat) {struct m}: F.K nat :=
     match m with  | 0 => F.val n
                   | (S p) => F.bind (pred n) (fun n' => minus n' p)
     end.
```

At last, let me consider the following three goals (below literals are expanded in Peano numbers by the Coq parser). The first goal does not hold, because the left side aborts, whereas the right side does not. The second goal holds, because the right side aborts. The third goal is an expected property of `minus`.

```
forall (n:nat), F.refines (minus 100 (500+n)) (F.any nat).
forall (n:nat) (sp:F.K nat), F.refines sp (minus 100 (500+n)).
forall (n:nat), F.refines (minus (500+n) 500) (F.val n).
```

After introduction of variables and application of `wp_refines`, the first goal is reduced to formula `(nat -> True) -> False`, the second goal is reduced to `False -> F.sdemon sp (fun _:nat => False)` and the third goal is reduced to `True -> n = n`. Here, we benefit from the fact that `500+n` is reduced by Coq to $\mathtt{S}^{500}$ `n`. Of course, Coq can discharge automatically the two last formulae.

Moreover, combining induction over `m`, transitivity of refinement and lemma `wp_refines`, I have established the correctness of `minus` in Coq interactive prover:

```
Lemma minus_correctness: forall (m n:nat), m <= n ->
  F.refines (minus n m)
            (F.choice (fun k => F.seq (F.ensure (n=m+k)) (F.val k))).
```

This small example illustrates that refinement is very convenient to handle partial functions in Coq and that computations of WP can involve structural recursion and pattern-matching.

5.2 Mixing deductions and WP-computations in the pure DSM

In the presence of higher-order functions, `wp_refines` rule does not suffice. For example, because of higher-order parameters, `sdemon` and `angel` are not necessarily eliminated from the resulting formula. Moreover, reasoning about higher-order functions often requires to find a good instantiation of a higher-order lemma. Hence, the user needs to control WP-computations such that in refinement proofs, deductions and WP-computations may be interlaced. The deduction rules in refinement proofs are: reflexivity, transitivity of refinement, associativity, monotonicity of `bind`, and other lemmas derived from DSM axioms.

In the pure DSM, three simplifying rules involving `bind` operator are given below. The property called `bind_simpl_left` indicates that "`(F.refines (F.bind s1 s2) s3)`" can be deduced from "`(F.sdemon s1 (fun a => F.refines (s2 a) s3))`". If `s1` is simple enough, this last formula is simplified such that `s1` and `sdemon` do not appear any more. Hence, this lemma allows to perform a kind of partial evaluation of `bind` into the left side of the refinement goal. The two others lemmas correspond respectively to a simplification of `bind` in the right part of the refinement goal, or in both part of the refinement goal.

```
Lemma bind_simpl_left: forall (A B:Type) (s1:F.K A) (s2:A -> F.K B) (s3:F.K B),
   F.sdemon s1 (fun a=>F.refines (s2 a) s3) -> F.refines (F.bind s1 s2) s3.

Lemma bind_simpl_right: forall (A B:Type) (s1:F.K A) (s2:A -> F.K B) (s3:F.K B),
   F.angel s1 (fun a=>F.refines s3 (s2 a)) -> F.refines s3 (F.bind s1 s2).

Lemma bind_simpl_both: forall (A B:Type) (s1 s3:F.K A) (s2 s4:A -> F.K B),
  F.sdemon s1 (fun a => F.refines (s2 a) (s4 a))
   -> F.refines s1 s3 -> F.refines (F.bind s1 s2) (F.bind s3 s4).
```

Before to apply these simplification lemmas, it is sometimes needed to have in hypothesis the precondition of the right side of the refinement goal. This may be achieved by using `pre_intro` lemma below, or the weaker `pre_intro_bind` lemma, if its hypothesis is sufficient:

```
Lemma pre_intro: forall (A : Type) (s1 s2:K A),
  (pre s2 -> refines s1 s2) -> refines s1 s2.

Lemma pre_intro_bind: forall (A B:Type) (s1:K A) (s2:A -> K B) (s3:K B),
  (pre s1 -> refines s3 (bind s1 s2)) -> refines s3 (bind s1 s2).
```

5.3 Mixing deductions and WP-computations in the state DSM

I now briefly explain how the previous ideas are extended to reason in the state DSM. In the state DSM, it is convenient to reason with a restricted form of refinement formulae that compares specifications for a given initial state. Below, I define `refInEnv` relation from `refines`:

```
Definition refInEnv (st:St) (A:Type) (s1 s2: K A)
 :=  (refines (seq (set st) s1) (seq (set st) s2)).
Implicit Arguments refInEnv [A].
```

Actually, refinement proofs can use `refInEnv` instead of `refines`, because of the following property:

```
forall (A:Type)(s1 s2:K A), refines s1 s2 <-> forall st, refInEnv st s1 s2.
```

The interest of `refInEnv` is that conditions over the initial state are propagated from the hypotheses without a loss of information, as exemplified below on the state DSM version of `bind_simpl_both`.

```
Lemma bind_simpl_both: forall (A B:Type) (s1 s3:K A) (s2 s4:A -> K B) (st:St),
  sdemon s1 (fun stf a => refInEnv stf (s2 a) (s4 a)) st
    -> refInEnv st s1 s3 -> refInEnv st (bind s1 s2) (bind s3 s4).
```

Actually, interactive refinement proofs look like symbolic debugging of two non-deterministic processes running concurrently. Indeed, applying the simplification rules, the user can "run" step-by-step specifications in the refinement goal. Of course, here, each execution step may generate proof obligations that must be discharged.

More precisely, a proof of `refInEnv st s1 s2` can be considered as a run of two concurrent processes `s1` and `s2` sharing a common state with `st` as initial value. Processes may be executed in parallel if both agree on the new value of the state: this is precisely expressed by `bind_simpl_both` lemma. But, execution of processes may be also interleaved: one of the process modifies the state whereas the other is asleep. In this case, when a process is put to sleep, the current value of the state is saved, such that the first instruction of this process on waking is to restore this value. For example, if both processes were running in parallel, application of `start_left` puts `s2` to sleep; application of `bind_simpl_left` performs a step of execution on the left process while the right process is asleep; and, application of `switch_to_right` wakes up `s2` and puts `s1` to sleep. [3]

```
Lemma start_left: forall (A:Type) (s1 s2:K A) (st:St),
    refInEnv st s1 (seq (set st) s2) -> refInEnv st s1 s2.

Lemma bind_simpl_left: forall (A B:Type) (s1:K A) (s2:A -> K B) (s3:K B) (st1 st3:St),
    sdemon s1 (fun st2 a => refInEnv st2 (s2 a) (seq (set st3) s3)) st1
       -> refInEnv st1 (bind s1 s2) (seq (set st3) s3).

Lemma switch_to_right: forall (A:Type) (s1 s2:K A) (st1 st2:St),
    refInEnv st2 (seq (set st1) s1) s2 -> refInEnv st1 s1 (seq (set st2) s2).
```

Here, the user plays the role of the scheduler. The size of these execution steps can be controlled using associativity of `bind`. And monotonicity allows to choose the good level of abstraction. Hence, the user has a great control on the size and the complexity of the proof obligations generated by wp-computations. Furthermore, interactive refinement proofs are very natural using this interpretation.

[3]The symmetric lemmas also hold, see [Bou06].

6 Proof of programs with the state DSM

This section presents how the theory of Dijkstra Specification Monads can be used to reason about higher-order imperative functions. It is clear from DSM axioms that equational reasoning in monads presented Section 2.1 is also available in DSM (see the description of the formalization given in [Bou07b]). Thus, here, I first explain how reasonings *à la Hoare* on first order imperative programs are encoded into the state DSM (subsection 6.1). Then, I show how the state DSM can express both partial and total correctness of programs which may not terminate (Subsections 6.2, 6.3 and 6.4). I also show some imperfections of DSM: higher-order functions in DSM may be non-monotonic (subsection 6.5) and, programs handling references to functions are not fully supported by the state DSM (subsection 6.6).

6.1 Hoare specifications

Given a predicate `q: A -> Prop`, I define `absPurePost q` as the specification of expressions returning a value satisfying `q`:

```
Definition absPurePost (A : Type) (q : A -> Prop) : K A :=
   choice (fun a => seq (ensure (q a)) (val a)).
Implicit Arguments absPurePost [A].
```

On a state DSM, given a predicate `q: St -> A -> Prop`, I define `absPost q` as the specification of imperative expressions that admits `q` as postcondition:

```
Definition absPost (A : Type) (q : St -> A -> Prop) : K A :=
   choice (fun stf => seq (set stf) (absPurePost (q stf))).
Implicit Arguments absPost [A].
```

Given `e` of type `(K A)` in state DSM, the Hoare specification of `e` (see Section 2.3) can now be expressed by the following refinement formula:

```
refines e (bind get (fun sti => seq (require (P sti)) (absPost (Q sti))))
```

There is however an important difference between specifications in Hoare logic and in DSM with respect to the well-known *frame problem.* In Hoare logic, a "true" post-condition allows the implementation to perform any side-effect: if we want to impose *e* to be "pure" (that means *e* does not change observationally the initial state), then we are obliged to specify in `Q` the condition `stf=sti`. On the contrary, in DSM, specifications indicate explicitly all side-effects that are (observationally) authorized in implementations. Thus, in order to impose `e` to be "pure", we can simply use instead a postcondition `Q'` of type `St -> A -> Prop` relating the initial state to the final result, and replace in the above refinement formula `absPost (Q sti)` by `absPurePost (Q' sti)`.

Actually, in interactive refinement proofs, it is more convenient to work with the following formula which is equivalent to the previous refinement formula:

```
forall sti:St, (P sti) -> refInEnv sti e (absPost (Q sti))
```

This last formula can then be simplified using the following rule:

```
Lemma absPost2wp: forall (A:Type) (s:K A) (q:St->A->Prop) (st:St),
    (sdemon s q st) -> (refInEnv st s (absPost q)).
```

Applying this rule generates proof obligations corresponding approximatively to those of Hoare logic (see the discussion in Section 8.1).

When `(refInEnv st s (absPost q))` holds, the following lemma allows to introduce an assertion on the result and the final state of `s` execution. This lemma is typically needed when `s` is the parameter of a higher-order function, which is required to refines `(absPost q)`.

```
Lemma abspost_bind_ensure:
  forall (A:Type) (s: K A) (q:St->A->Prop) (st:St),
   (refInEnv st s (absPost q))
     -> refInEnv st s (bind s (fun a =>
                           (bind (get _) (fun new_st =>
                             seq (ensure (q new_st a)) (val a))))).
```

6.2 Feasible and non-aborting specifications

A feasible specification is a specification that can be implemented. In the state DSM, we can express this idea using the following pre/post formula. Given `sp: K A`, the specification `sp` is feasible in the initial state `sti` iff

```
(F.pre (sp sti)) -> exists (a:A), exists (stf:St), (F.post (sp sti) (stf,a))
```

Alternatively, we can express this using weakest-preconditions:

```
(sdemon sp (fun _ _ => True) sti) -> (angel sp (fun _ _ => True) sti)
```

But, we can also write this using a very simple refinement formula. Intuitively, this last formula expresses that in the state `sti`, the specification `sp` does not ensure false.

```
refInEnv sti skip (bind sp (fun _ => set sti))
```

These three formulas are logically equivalent: the first gives the intuitive semantics of feasibility, the second is useful to simplify feasibility formula by computation of weakest-preconditions, and the last allows to reuse deduction rules of refinement to reason about feasibility.

Formally, I thus choose to define feasibility using this refinement formula. Then, I prove the equivalence with the wp-based formula.

```
Definition feasible (A:Type) (sp:K A) (st:St)
  := refInEnv st skip (bind sp (fun _ => set st)).

Lemma feasible_wp: forall (A: Type) (sp: K A) st,
  ((sdemon sp (fun _ _ => True) sti) -> (angel sp (fun _ _ => True) sti))
   -> feasible sp st.

Lemma feasible_unfold: forall (A: Type) (sp: K A) st,
  (feasible sp st) ->
      (sdemon sp (fun _ _ => True) sti) -> (angel sp (fun _ _ => True) sti).
```

This second lemma allows to prove that the specification `magic A` defined below is never feasible: it is the bottom of the refinement preorder.

```
Definition magic (A:Type): K A := seq (ensure False) (any A).

Lemma magic_unfeasible: forall (A: Type) (st:St), ~(feasible (magic A) st).
```

A specification `sp: K A` is said to be non-aborting in the initial state `st` if `F.pre (sp st)` holds. Non-abortion can be expressed as two equivalent refinement formulas. First, as the symmetric of feasibility:

```
refInEnv st (bind sp (fun _ => set st)) skip.
```

Or, second, using `absPost`:

```
refInEnv st sp (absPost (fun _ _ => True)).
```

6.3 Non-terminating expressions

Section 3 presents smallest and greatest fixpoint operators. I claim here that non-terminating expressions can be represented by smallest fixpoints in partial correctness semantics, or by greatest fixpoints in total correctness semantics. This is briefly illustrated below on a loop operator. First, I define the "unfolding" of a while-loop computation `w` where `cond` and `body` represents respectively the condition and the body of this computation:

```
Definition unfoldW (cond:K bool) (body w:K unit) : K unit :=
   If cond Then (seq body w) Else skip
```

Then, `while cond body` is defined as the smallest fixpoint of `unfoldW cond body`.

```
Definition while (cond:K bool) (body:K unit) : K unit
 := sfix (unfoldW cond body).
```

Operator `while` corresponds to a while-loop in a partial correctness semantics: termination is not guaranteed. In the state DSM, a `whileWF` operator corresponding to a total correctness semantics can also be defined from `while`. Hence, `whileWF cond body` calls `while cond body` under a higher-order *precondition*, which expresses that, in the state transformation induced by the sequence of `cond` and `body`, the state strictly decreases with respect to a well-founded relation. In other words, `whileWF` requires a property ensuring its termination. When it appears in the left side of a refinement goal, the user has to prove this property. Moreover, `whileWF cond body` is refined by any fixpoint of `unfoldW cond body`, because under its assumption of termination, there is a unique fixpoint (see [Bou06]). Hence, `whileWF` is a particular case of greatest fixpoint. In particular, `while` preserves non-abortion whereas `whileWF` preserves feasibility.

These two ways of representing non-terminating programs are inherited from Hoare logics. In Hoare logics for partial correctness non-terminating programs satisfy a false postcondition, whereas in Hoare logics for total correctness non-terminating programs have a false weakest-precondition.

6.4 Partial and total correctness in the state DSM

The previous subsection claims that the semantics of DSM with respect to termination only rely on the use of "higher-order" operators like `while` or `whileWF`. Hence, even if all computations of the original monad M terminate, its associated DSM is expressive enough to represent non-terminating expressions: this DSM can be used to reason about programs of an extension of M with fixpoint operators. According to the desired semantics, partial or total correctness, these operators have to be interpreted as smallest or particular greatest fixpoints. This idea is now made more precise.

Let me assume a programming language P such that its basic constructions can be interpreted as primitive or derived constructions of the state DSM. For instance, P is a simply typed lambda-calculus extended with `val`, `bind`, `set`, `get`, and `while`. Hence, `while` allows to write non-terminating expressions.

In the construction of the state DSM given Section 4, specifications of the state DSM can be interpreted as a pair of predicate representing a precondition and a postcondition. Thus, we now translate the usual notion of partial correctness for Hoare logic, to define *the partial correctness of refinement proofs in the state DSM with respect to P*: for every P-expression `e` and every specification `s` such that `(refines e s)`, then for every P-expression `e'`, and for every state `sti` and `stf`, if `e` with the initial state `sti` reduces to `e'` with the state `stf`, then `refines (seq (set stf) e') (seq (set sti) s)`. Using the transitivity of refinement, this partial correctness property is reduced to the verification that for every P-expression `e'`, and for every state `sti` and `stf`, if (e,`sti`) reduces to (`e'`,`stf`) then `refines (seq (set stf) e') (seq (set sti) e)`.

In the total correctness semantics, we interpret `while` of P into `whileWF` of the state DSM. And, we now want to verify additionally that for any P-expression `e` and for any state `sti`, if `e` does not abort in the initial state `sti` a (i.e. if `refines (seq (set sti) e) (absPost (`**`fun`**` _ _ => True))`), then computation of `e` in the initial state `sti` is terminating.

If `P` is deterministic, then the usual notion of termination matches exactly feasibility (see Subsection 6.2) : we say that `e` terminates in the initial state `sti` if `e` is feasible in the state `sti`. Thus, we have simply to check that basic constructions of P preserve feasibility. In particular, this property is proved for `whileWF` (see [Bou06]).

However, semantics of real programming languages are never deterministic, even for purely sequential languages. In particular, implementation-defined behaviors (uninitialized variables, evaluation order of some subexpressions) may be expressed as non-deterministic. Thus, I now assume that P has a `any` operator, which is syntactically restricted to non-empty types (hence, there is no magic behavior in P). And, we now want to verify that for any P-expression e and for any state `sti`, if `e` has a true precondition in the initial state `sti`, then *all* computations of `e` in the initial state `sti` are terminating *whatever are the demonic choices of the execution.* This property is called strong-termination (whereas feasibility is weak-termination).

Defining strong-termination requires here to reason about the reduction relation. I have formalized and proved this property in Coq for a particular first-order expression language containing `whileWF`, `any` restricted on non-empty types and `require` (to represent aborting expressions). The reduction relation being expressed as a small-steps semantics, strong-termination is formalized as well-foundation of the reduction relation (see [Bou06]). However, the proof is quite tedious. It may be much simplified by Leroy's technique (see [Ler06]): by using instead a big-step semantics and by formalizing strong-termination as non-divergence, where divergence is coinductively defined. Indeed, in the intuitionistic logic of Coq, non-divergence is weaker than well-foundation.

6.5 Higher-order refinement and non-monotonicity

Refinement formulas are propositions that may appear in `require` or `ensure` instructions: this is *higher-order refinement*. This feature has been used in the fixpoint theory of DSM (see Section 3.2). But this feature may also be useful to express properties of functional parameters: we will see the example of `nimgamewf` function in Subsection 7.3. However, this feature has a drawback: higher-order functions in DSM are not necessarily monotonic. As a counter-example, the function below is not monotonic over `f`. But, when monotonicity holds, it is usually trivially provable.

```
fun (f:nat -> K unit) => require (equiv (f 0) skip)
```

6.6 Representing references to functions in the state DSM

At this point, a natural question arises: can we represent in the state DSM any ML program mixing references and functions? Aliases do not seem a problem at this level: we may encode explicitly the global state of type `St` as a heap, even if reasoning with such an encoding is difficult. Here, a problem comes from the fact that references to functions can only be incompletely represented in the state monad. Moreover, this incompleteness problem is not specific to the state monad, it also happens in the exception monad with exceptions of functions parametrized by an exception.

Hence, given a reference `f` to a function of type `unit->unit`, we can not represent the following ML program in the state monad.

```
  (f := fun () -> (!f)()) ;
  (!f)()
```

Indeed, this program is non-terminating, whereas in the state monad, all computations are terminating. And, this program does not use a fixpoint operator, but only functions and references. Thus, it does not seem to have a natural representation in the state DSM.

However, the state monad can cope with *some* programs handling references to functions. Basically, in these programs, there is a partial order on references such that a reference can only point to functions that work with strictly lower references. For instance, considering a reference `x` to an integer, we can represent the following program in the state monad, because `f` references only functions working on `x`.

```
(f := fun() -> x := 1 + !x) ;
(!f)() ;
(f := fun() -> x := 2 + !x) ;
(!f)()
```

First, we define `K_x` as the type of computations in the state monad where `St` is an integer representing reference `x`. Then, we define `K_x_f` as the type of computations in the state monad where `St` is a pair of an integer (representing `x`) and functions of type `unit -> (K_x unit)` (representing `f`). There is an `embed` function to embed computations of `(K_x A)` into computations of `(K_x_f A)`: `embed` is a particular case of the lift function of the state monad transformer. Using these definitions, the transcription of the code above is straightforward. Such an encoding can be mechanized using a good type system with effects [Fil99, Wad99, Ben02]. This remains to be adapted to the state DSM.

7 A concrete example in the state DSM : the game of Nim

In this section, I illustrate the expressive power of the state DSM on the modelization of the game of Nim. This is a two-player games where players take turns to remove between one or three matches from a stack. The game ends when the stack is empty. The winner is the last who has played. Here, we first want to describe the game in the state DSM, then to study an optimal strategy, and at last to prove the correctness of an implementation of this optimal strategy. The interest of this implementation is to store information in the memory that avoids to recompute the whole strategy at each turn.

This example illustrates the top-down methodology allowed by refinement: in particular, the proof of optimality of the strategy is completely independent of the proof that the implementation refines the strategy, but the combination of this two proof allows to derive that the implementation is an optimal player. This example also combines higher-order reasoning with Hoare specifications. It shows that higher-order refinement is convenient to reason about games. It also performs a kind of data refinement. The interesting part of this data refinement is not the change of representation (here, it only instantiates some polymorphic part of the state), but the fact that this data refinement requires the other player to respect the rules of the game. Hence, this is a case of "data-refinement with rely-guarantee" (see a formalization of this approach for B in [Büc99]).

In the following, the type `K A` of computations in the state DSM is now explicitly parametrized by the global state with the form `K St A`. To improve the readability, I introduce a few definitions:

```
Definition eval_ (St A:Type) (f: St -> A) : K St A :=
  bind (get St) (fun st => val (f st)).
Implicit Arguments eval_ [St A].
Definition require_ (St:Type) (p: St -> Prop): K St unit
 := bind (get St) (fun st => require (p st)).
Implicit Arguments require_ [St].
Definition ensure_ (St:Type) (p: St -> Prop): K St unit
 := bind (get St) (fun st => ensure (p st)).
Implicit Arguments ensure_ [St].
```

At last, I introduce two notations: binary operator `seq` is now denoted by the infix operator "`-;`" and the ternary relation `refInEnv st sp1 sp2` is now written "`[< st >] sp1 >==> sp2`".

7.1 Modelization of two-players games

In a first step, we only describe the execution of two-player games. Here, we name the position of each player with a boolean : "player *true*" or "player *false*". Assuming a type `GameData:` **Type**, the state of a two-player game is of the following type:

```
Record GameSt : Type := mkGameSt {
  player:  bool ;        (* position of the next player *)
  gdata:   GameData      (* data of the game *)
}.
```

The action to change the player position is given by `change_player` below, where `negb` is the boolean negation:

```
Definition dualSt (st:GameSt) : GameSt := (mkGameSt (negb (player st)) (gdata st)).
Definition change_player: K GameSt unit := bind (get _) (fun st => set (dualSt st)).
```

The execution of a two-player game is described by `game not_gameover pt pf` where `not_gameover` returns true while the game is not over, and `pt` and `pf` compute respectively the action player player true and player false.

```
Definition game (not_gameover: GameSt -> bool) (pt pf: K GameSt unit): K GameSt unit :=
    while (eval_ not_gameover)
          ((If eval_ player Then pt Else pf) -;
           change_player).
```

Let us remark here, that players may read and write into field `player` of the state. Moreover, the player true is not necessarily the first player: the first player corresponds to the value of the field `player` at the initial state of the execution. We now study some fundamental properties of `game`.

7.1.1 Refinement of players

Here, we describe a necessary condition to refine `pt` and `pf` in the execution of a game. The lemma below expresses that `game` is itself a monotonic function. It is a trivial consequence of fixpoint monotonicity.

```
Lemma game_monotonic:
  forall (cond1 cond2: GameSt -> bool) (pt1 pt2 pf1 pf2: K GameSt unit),
     (forall st, (cond1 st)=(cond2 st))
  -> (forall st, (player st)=true -> (cond1 st)=true -> [< st >] pt1 >==> pt2)
  -> (forall st, (player st)=false -> (cond1 st)=true -> [< st >] pf1 >==> pf2)
  -> forall st, [< st >] (game cond1 pt1 pf1) >==> (game cond2 pt2 pf2).
```

Actually, this lemma is a bit stronger than a pure monotonicity lemma, since `pt1` (resp. `pf1`) is allowed to refine `pt2` (resp. `pf2`) assuming conditions on the initial state. However, as we will see in the following, this lemma may not be sufficient to refine players. Indeed, we may need to also assume an invariant on the state : a property satisfied during all the execution of the game. We prove thus the stronger theorem (`game_monotonic` is a consequence when `inv` is **fun** `_ => True`):

```
Theorem game_monotonic_with_invariant:
  forall cond1 cond2 pt1 pf1 pt2 pf2 (inv: GameSt -> Prop),
     (forall st, (inv st) -> (cond1 st)=(cond2 st))
  -> (forall st, (player st)=true -> (cond1 st)=true -> (inv st) ->
                 [< st >] pt1 >==> (pt2 -; ensure_ (fun st => inv (dualSt st))))
  -> (forall st, (player st)=false -> (cond1 st)=true -> (inv st) ->
                 [< st >] pf1 >==> (pf2 -; ensure_ (fun st => inv (dualSt st))))
  -> forall st, [< st >] (game cond1 pt1 pf1)
                   >==> (require_ (fun st => inv st) -; game cond2 pt2 pf2).
```

This theorem is directly derived from `sfix_refines_G_sfix`, instantiating the function `G` with the function **fun** `(k: K GameSt unit) => (require_ (`**fun** `st => inv st) -; k)`.

7.1.2 Symmetry of the game

Now, we express the symmetry between player true and player false. This symmetry is expressed using `dualPlayer` defined below. Hence, if `p` is a player then `dualPlayer p` acts like `p` in the dual position.

```
Definition dualPlayer (p: K GameSt unit) :=
  (change_player -; p -; change_player).
```

Trivially, `dualPlayer` is a monotonic and involutive function. Hence, it satisfies also the property:

```
Lemma dualPlayer_left_right: forall p1 p2 st,
    ([< dualSt st >] p1 >==> (dualPlayer p2)) <-> ([< st >] (dualPlayer p1) >==> p2).
```

The symmetry of the game is expressed as:

```
Lemma game_symmetry: forall cond pt pf st,
   [< st >] (game cond pt pf)
       >==> (dualPlayer (game (fun st => cond (dualSt st))
                              (dualPlayer pf)
                              (dualPlayer pt))).
```

This lemma is directly derived from `sfix_refines_G_sfix` instantiating `G` with `dualPlayer`.

In the following, we are interested in a game where `cond` is symmetric (e.g. such that `(cond st)` equals `(cond (dualSt st))`). Thus, the symmetry lemma allows to reduce the study of a strategy to the study of its dual in the dual position. This is particularly interesting when this strategy is its own dual.

7.1.3 A rely-guarantee methodology to refine players

In the following, we describe the rule of the game as a postcondition expressing actions authorized by players when they take their turn. This postcondition of type `GameSt -> GameSt ->` **Prop** is a property relating the initial state of the player (before its turn) to its final state (after its turn). Player are not allowed to modify observationally field `player` of the state. This last idea is expressed by `absPlayer` below: given a rule `gameRule`, `(absPlayer gameRule)` is a non-deterministic player respecting `gameRule` without modifying observationally field `player`. Such a player is called an abstract player.

```
Definition absPlayer (gameRule: GameSt -> GameSt -> Prop): K GameSt unit :=
    bind (get _) (fun st =>
     absPost (fun st' _ => gameRule st st' /\ (player st')=(player st))).
```

The function `absPlayer` is monotonic: implication on postconditions implies refinement of corresponding abstract players.

```
Lemma absPlayer_monotonic: forall (rule1 rule2: GameSt -> GameSt -> Prop) st,
    (forall st st', (rule1 st st') -> (rule2 st st'))
    -> [< st >] (absPlayer rule1) >==> (absPlayer rule2).
```

Moreover, the dual player of an abstract player is of course itself an abstract player. When the rule is symmetric, the corresponding abstract player is its own dual.

```
Lemma absPlayer_autodual: forall gameRule st,
  [< st >] (dualPlayer (absPlayer (fun st st' => gameRule (dualSt st) (dualSt st'))))
     >==> (absPlayer gameRule).
```

Now, we propose to define a methodology to study players of a given game, using an approach with two refinement level. At the first level, we refine the rule of the game by a strategy. This strategy is typically itself an abstract player. By studying the game where one player is the strategy and the other is the rule of the game, we can prove when this strategy allows to win. At the second level, we refine this strategy into an implementation. Of course, here, we would like to assume in this refinement, that the other player respects the rule of the game.

We now give a decomposition theorem to achieve this two-level approach. First, given an implementation `p`, we define now the monitor `invPlayer p inv gameRule` which requires that invariant `inv` is satisfied when `p` takes its turn and ensures that after the execution of `p`, `inv` will be established after the other player has played. Here, the other player is assumed to respect the rule `gameRule`.

```
Definition invPlayer p (inv:GameSt -> Prop) (gameRule:GameSt -> GameSt -> Prop) :=
    require_ (fun st => inv st) -;
    p -;
    ensure_ (fun st => forall st',
                        (gameRule (dualSt st) (dualSt st'))
                         -> (player st')=(player st) -> inv st').
```

We say that `p` satisfies `inv` if `p` refines `(invPlayer p inv gameRule)`. Of course, `invPlayer` is monotonic and the true invariant is always satisfied.

```
Lemma invPlayer_monotonic: forall p1 p2 (inv: GameSt -> Prop) rule st,
    ((inv st) ->  [< st >] p1 >==> p2)
      -> [< st >] (invPlayer p1 inv rule) >==> (invPlayer p2 inv rule).

Lemma invPlayer_trivial_trivok: forall p rule st,
   [< st >] p >==>  (invPlayer p (fun _ => True) rule).
```

Now, we want to instantiate `game_monotonic_with_invariant` such that each player may have its own invariant: indeed, the global invariant of this theorem is not very adapted to modular reasoning about players. Hence, given `invpt` and `invpf` the respective invariants of player true and player false, given `rule` the rule of the game, then the global invariant of the game is:

```
Definition gameInv (invpt invpf: GameSt -> Prop) (rule: GameSt -> GameSt -> Prop) st
  := if (player st)
     then (invpt st)
           /\ forall st', (rule st (dualSt st')) -> (player st')=false -> (invpf st')
     else (invpf st)
           /\ forall st', (rule st (dualSt st')) -> (player st')=true -> (invpt st').
```

This invariant expresses that for the two next turns, the invariant of each player will be satisfied at its turn to play, assuming that the other player respects the rule of the game. We can now express our decomposition theorem. Below, we denote `rule` the rule of the game, `pt1` (resp. `pf1`) is an implementation of player true (resp. false), and `pt2` (resp. `pf2`) is the strategy of player true (resp. false). We assume that `pt1` (resp. `pt2`) satisfies invariant `invpt` (resp. `invpf`). We assume that under the precondition of its invariant, each implementation refines the corresponding strategy. Similarly, we assume that under the precondition of its invariant, each strategy refines the rule of the game. The theorem states that under the precondition that each invariant are initially satisfied, then the game between implementations refines the game between strategies.

```
Theorem game_PlayerRefinement: forall cond pt1 pf1 pt2 pf2 invpt invpf rule,
     (forall st, (player st)=true -> (cond st)=true ->
                    [< st >] pt1 >==> invPlayer pt1 invpt rule)
  -> (forall st, (player st)=false -> (cond st)=true ->
                    [< st >] pf1 >==> invPlayer pf1 invpf rule)
  -> (forall st, (player st)=true -> (cond st)=true -> (invpt st) ->
                    [< st >] pt1 >==> pt2)
  -> (forall st, (player st)=false -> (cond st)=true -> (invpf st) ->
                    [< st >] pf1 >==> pf2)
  -> (forall st, (player st)=true -> (cond st)=true -> (invpt st) ->
                    [< st >] pt2 >==> absPlayer rule)
  -> (forall st, (player st)=false -> (cond st)=true -> (invpf st) ->
                    [< st >] pf2 >==> absPlayer rule)
  ->  forall st, (gameInv invpt invpf rule st)
                  -> [< st >] (game cond pt1 pf1) >==> (game cond pt2 pf2).
```

This theorem is directly derived from `game_monotonic_with_invariant` instantiating the global invariant `inv` with `(gameInv invpt invpf rule)`. Its main interest is that each player can be independently refined. Moreover the hypothesis about each player in `game_monotonic_with_invariant` is now split into three separate properties: first, the implementation satisfies its invariant; second, the implementation refines the strategy; and third, the strategy refines the rule.

7.2 The game of Nim

Now, we describe an abstract notion of Nim game. In this abstract view of the game, we do not precise how players deal with the memory. The interest of this abstraction will be justified in Subsection 7.3. In the following, `Z` is the type of infinite binary integers, defined in Coq library. Whereas the number of matches is always non-negative, I represent it in `Z` because, the number theory of `Z` is well-developed in Coq. Hence, here we assume that there is a function `nb_matches: GameData -> Z` that extracts the number of matches from values of type `GameData`.

To simplify a bit the notations, we write "`nbm st`" to express "`(nb_matches (gdata st))`". The execution of the Nim game is defined by `nimgame` below, where `Zlt_bool` is a predefined function of type `Z -> Z -> bool` such that `(Zlt_bool x y)=true <-> x<y`.

```
Definition nimgame (pt pf: K GameSt unit)
 := game (fun st => Zlt_bool 0 (nbm st)) pt pf.
```

Each player of the Nim game are assumed to follow `nimRule` below. This rule expresses that players must decrease `nb_matches` of a value between 1 and 3, but keep this number non-negative. Implicitely, other parts of `GameSt` can change non-deterministically.

```
Definition nimRule (st st': GameSt): Prop :=
     let nm:=(nbm st) in
     let nm':=(nbm st') in
          0 <= nm'
      /\  nm-3 <= nm' <  nm.
```

By convention, the player `p` wins the game in the state `st`, if the game ends in a state `st'` such that `is_winner p st'` (where `is_winner` is defined below) :

```
Definition is_winner (p: bool) (st': GameSt): Prop
 := (player st')=negb p.
```

First, we prove that when the game is over, then the number of matches is zero. However, we do not prove here that the game will end. We leave this for Subsection 7.3. game will end.

```
Lemma nimgame_end: forall st, (nbm st) >=0 ->
        [< st >] (nimgame (absPlayer nimRule) (absPlayer nimRule))
                 >==> (absPost (fun st _ => nbm st=0)).
```

This lemma is easily proved using the standard rule of Hoare logic for while-loops in partial correctness: the invariant of the loop is here "`nbm st >= 0`".

Let us now introduce the optimal strategy of this game. The theorem expressing that this strategy is optimal will be proved in Subsection 7.3. First, let us remark that all components of this game (e.g. `nimgame`, `nimRule` and `is_winner`) are fully symmetric between player true and player false. Hence, the optimal strategy is the same for these two players. Now, the reasoning to find the optimal strategy is the following: if the initial number of matches is zero, then the first player immediately loses. Hence, if the initial number of matches is between 1 and 3, then the first player can remove all the matches, which makes the second player lose. More generally, if the initial number of matches is not multiple of 4 then the first player will win by removing at each turn the remainder of the division of the number of matches by 4. Indeed, in this case, the second player finds at each turn a number of matches that is multiple of 4, and necessarily leaves to the first player a number of matches that is not multiple of 4. Symmetrically, if the initial number of matches is multiple of 4 then, the first player will lose if the second player applies this optimal strategy.

In summary, the optimal strategy is the following: if the number of matches is multiple of 4, then apply non-deterministically the rule of the game, else remove the remainder of the division by 4. Below infix operator `mod` computes this remainder and `Zeq_bool` is a predefined function of type `Z -> Z -> bool` such that `(Zeq_bool x y)=true <-> x=y`.

```
Definition nimStrategy (st st': GameSt): Prop :=
     let nm:=(nbm st) in
     let r:=(nm mod 4) in
     if (Zeq_bool r 0) then
        nimRule st st'
     else
        (nbm st')=nm - r.
```

The abstract player following this strategy is called `expertPlayer`:

```
Definition expertPlayer: K GameSt unit := absPlayer nimStrategy.
```

First, we prove that `expertPlayer` respects the rule of the game:

```
Lemma expertPlayer_isNimPlayer: forall st,
  0<(nbm st) -> [< st >] expertPlayer >==> (absPlayer nimRule).
```

Then, we prove that the dual player of `expertPlayer` is `expertPlayer`. This lemma will be used to apply `game_symmetry`.

```
Lemma expertPlayer_autodual: forall st,
  [< st >] (dualPlayer expertPlayer) >==> expertPlayer.
```

Now, we prove that if `expertPlayer` is player false, then it wins when the initial state `st` satisfies `(Zeq_bool ((nbm st) mod 4) 0)=(player st)`.

```
Definition false_can_win : K GameSt unit
  := bind (get _) (fun st =>
        require ((nbm st) >= 0) -;
        absPost (fun st' _ => (Zeq_bool ((nbm st) mod 4) 0)=(player st)
                                 -> is_winner false st')).


Theorem expertPlayer_nimgame: forall st,
  [< st >] (nimgame (absPlayer nimRule) expertPlayer) >==> false_can_win.
```

To prove this theorem, I directly apply `sfix_smallest` with `sp` being `false_can_win`. Hence, I avoid here to guess the invariant, which seems a bit tedious to formulate (hence, interactive theorem proving may help users to build invariants). Then, the proof is closed to the informal proof sketched above.

At this point, we can also prove, by invoking `game_symmetry`, that if `expertPlayer` is player true, then it wins on dual initial states: when the initial state `st` satisfies "`(Zeq_bool ((nbm st) mod 4) 0)` equals `(negb (player st))`". This implies optimality of `expertPlayer` since for all initial state `st` (such that `nbm st>=0`), `expertPlayer` wins either in position false or in position true. However, as explained later, I prefer prove optimality on a refinement of the Nim game.

7.3 Refinement using players with memory

We now refine the rule of the Nim game to allow each player to own a private memory zone, where it can store some information. By "private" here, we mean that only the owner of the zone can write into the zone (but, both players can read into each zone). Hence, assuming two types `WDatat` and `WDataf`, we now define `GameData` as

```
Record GameData: Type := mkGameData {
  nb_matches: Z;
  writet: WDatat ; (* write-private data of player true  *)
  writef: WDataf   (* write-private data of player false  *)
}.
```

This type definition introduces also `mkGameData` as a constructor of the record type plus three projections `nb_matches: GameData -> Z` and `writet: GameData -> WDatat` and `writef: GameData -> WDataf`. In the following, we write "`wrt st`" as a notation for "`(writet (gdata st))`", and "`wrf st`" as a notation for "`(writef (gdata st))`".

Thus, if `rule` is a rule, `WRule rule` is a rule that enforces `rule` such that each player can only modify its own zone.

```
Definition WRule (gameRule: GameSt-> GameSt -> Prop) (st st': GameSt): Prop :=
    gameRule st st'
 /\ if (player st) then (wrf st')=(wrf st) else (wrt st')=(wrt st).
```

Now, we define `absWPlayer` as an abstract player which satisfies a `WRule`.

```
Definition absWPlayer (gameRule: GameSt -> GameSt -> Prop) :=
  absPlayer (WRule gameRule).
```

By monotonicity of `absPlayer`, we prove immediately the following property:

```
Lemma absWPlayer_absPlayer:
  forall gameRule st, [< st >] absWPlayer gameRule >==> absPlayer gameRule.
```

Here, we introduce a way to decompose proofs of the form "`p` refines `absWPlayer gameRule`":

```
Lemma absWPlayer_decompose:
  forall p gameRule st,
    ([< st >] p >==> absPlayer gameRule)
    -> ([< st >] p >==>  absPost (fun st' _ => if (player st) then (wrf st')=(wrf st)
                                                           else (wrt st')=(wrt st)))
      -> ([< st >] p >==> absWPlayer gameRule).
```

Now, we refine `expertPlayer` as a player that respects the new rule.

```
Definition expertWPlayer: K GameSt unit := absWPlayer nimStrategy.
```

The two previous lemmas allows respectively to derive:

```
Lemma expertWPlayer_expertPlayer:
  forall st, [< st >] expertWPlayer >==> expertPlayer.
```

```
Lemma expertWPlayer_isNimWPlayer:
  forall st, 0 < (nbm st) -> [< st >] expertWPlayer >==> (absWPlayer nimRule).
```

Trivially, we can directly derive from the properties of `expertPlayer` the sufficient conditions which ensure that `expertWPlayer` wins against `(absWPlayer nimRule)`.

```
Lemma expertWPlayer_as_player_false:
 forall st, (nbm st) >= 0
  -> (Zeq_bool ((nbm st) mod 4) 0) = (player st)
  -> [< st >] (nimgame (absWPlayer nimRule) expertWPlayer)
                 >==> (absPost (fun st' _ => is_winner false st')).
```

```
Lemma expertWPlayer_as_player_true:
  forall st, (nbm st) >= 0
   -> (Zeq_bool ((nbm st) mod 4) 0) = negb (player st)
   ->  [< st >] (nimgame expertWPlayer (absWPlayer nimRule))
                  >==> (absPost (fun st' _ => is_winner true st')).
```

Let me remark here that in this game, `WRule` breaks the symmetry (as defined previously) between the two players: in particular, `(dualPlayer expertWPlayer)` is not equivalent to `expertWPlayer` since they do not write into the same zone of the memory. This justifies here the interest of having defined an abstract view of the game where the symmetry between players can be expressed.

As we do not want to refine again the rule of the game, we are now at the good level to express the termination of the game. First, we define the well-founded relation `Rgame` between two successive state of the game. The code below expresses that `Rgame` is the natural ordering over the number of matches.

```
Definition Rgame (st1 st2: GameSt): Prop := Zwf 0 (nbm st1) (nbm st2).
```

Given `pt` and `pf` two Nim players with respective invariants `invpt` and `invpf`, we define below the computation `nimgamewf pt pf invpt invpf` as the well-founded execution of the Nim game. Here, we need to introduce explicitly invariants to express that `pt` and `pf` satisfies the rule of the game under the precondition of their invariant.

```
Definition nimgamewf (pt pf: K GameSt unit) (invpt invpf: GameSt -> Prop) : K _ unit :=
  require (
       (forall st, (player st)=true -> 0 < (nbm st) ->
                     [< st >] pt >==> invPlayer pt invpt (WRule nimRule))
    /\ (forall st, (player st)=false -> 0 < (nbm st) ->
                     [< st >] pf >==> invPlayer pf invpf (WRule nimRule))
    /\ (forall st, (player st)=true -> 0 < (nbm st) -> (invpt st) ->
                     [< st >] pt >==> absWPlayer nimRule)
    /\ (forall st, (player st)=false -> 0 < (nbm st) -> (invpf st) ->
                     [< st >] pf >==> absWPlayer nimRule)
  ) -;
  (whileWF (* cond *) (eval_ (fun st => Zlt_bool 0 (nbm st)))
           (* body *) ((If eval_ player Then pt Else pf) -; change_player)
           (* inv. *) (gameInv invpt invpf (WRule nimRule))
           (* var. *) Rgame).
```

The key property of `nimgamewf` is the preservation of the feasibility of players.

```
Lemma nimgamewf_feasible: forall pt pf (invpt invpf: GameSt -> Prop),
     (forall st, (player st)=true -> 0 < (nbm st) -> invpt st -> feasible pt st)
  -> (forall st, (player st)=false -> 0 < (nbm st) -> invpf st -> feasible pf st)
  ->  forall st, feasible (nimgamewf pt pf invpt invpf) st.
```

Of course, another important property is the equivalence between `nimgamewf` and `nimgame` (under the preconditions of `nimgamewf`):

```
Lemma nimgame_refines_nimgamewf: forall pt pf invpt invpf st,
      [< st >] (nimgame pt pf) >==> (nimgamewf pt pf invpt invpf).
```

```
Lemma nimgamewf_refines_nimgame: forall pt pf invpt invpf,
     (forall st, (player st)=true -> 0 < (nbm st) ->
                   [< st >] pt >==> invPlayer pt invpt (WRule nimRule))
  -> (forall st, (player st)=false -> 0 < (nbm st) ->
                   [< st >] pf >==> invPlayer pf invpf (WRule nimRule))
  -> (forall st, (player st)=true -> 0 < (nbm st) -> (invpt st) ->
                   [< st >] pt >==> absWPlayer nimRule)
  -> (forall st, (player st)=false -> 0 < (nbm st) -> (invpf st) ->
                   [< st >] pf >==> absWPlayer nimRule)
  -> forall st, (gameInv invpt invpf (WRule nimRule) st)
                   -> [< st >] (nimgamewf pt pf invpt invpf) >==> (nimgame pt pf).
```

We are now in position to prove that `expertWPlayer` is the optimal player in position false (of course, a symmetric theorem is also provable) as stated by `expertWPlayer_optimal_as_player_false`. The statement of this theorem being a bit complex, it may be not obvious that it expresses what we informally mean by "optimal strategy".

Let me detail this statement. Here, we assume `pf` a player false which satisfies an invariant `invpf` (first hypothesis). Of course, we assume that `pf` respects the rule of the game (second hypothesis). We also assume that `pf` is feasible (third hypothesis): I will explain later why this hypothesis is needed. At last, we assume an initial state `st` in which `pf` wins against `(absWPlayer nimRule)`. Under all these hypotheses, the theorem states that `(player st)=(Zeq_bool ((nbm st) mod 4) 0)`. In other words, by `expertWPlayer_as_player_false`, we know that in this initial state `st`, `expertWPlayer` also wins against `(absWPlayer nimRule)`. Hence, the theorem expresses exactly that `expertWPlayer` is a player at least as good as any feasible player against `(absWPlayer nimRule)`. Here, we choose `(absWPlayer nimRule)` as player true, because we want to express that `pf` does not know the strategy of its opponent.

Indeed, players that know the strategies of their opponent can be better than `expertPlayer`. For instance, let us assume that when the number of matches is 2, player false knows that player true removes only one match (the player true plays stupidly in this particular configuration). Then, when the number of matches is 4, player false can be a better player than `expertPlayer`: it removes two matches, then the stupid player true removes only one match, and player false wins. On the contrary, when the number of matches is 4, `expertPlayer` may remove only one match and the stupid player true may win.

At last, let me remark that if `pf` is magic, then all hypotheses are valid except feasibility: magic wins (and loses) in any initial state. Hence, feasibility of `pf` is needed here.

```
Theorem expertWPlayer_optimal_as_player_false: forall pf invpf,
    (forall st, (player st)=false -> 0 < (nbm st) ->
                [< st >] pf >==> invPlayer pf invpf (WRule nimRule))
 -> (forall st, (player st)=false -> 0 < (nbm st) -> (invpf st) ->
                [< st >] pf >==> absWPlayer nimRule)
 -> (forall st, (player st)=false -> 0 < (nbm st) -> (invpf st) -> feasible pf st)
 -> forall st, (nbm st) >= 0
      -> (if (player st)
          then forall st', (WRule nimRule st (dualSt st'))
                             -> (player st')=false -> (invpf st')
          else (invpf st))
       -> ([< st >] (nimgame (absWPlayer nimRule) pf)
               >==> absPost (fun st' _ => is_winner false st'))
        -> (player st)=(Zeq_bool ((nbm st) mod 4) 0).
```

The proof follows the reasoning presented now. Under the hypotheses of the theorem, the boolean value of `(Zeq_bool ((nbm st) mod 4) 0)` is either `(player st)` or `negb (player st)`. In the former case, the result is trivially proved. In the latter case, we prove below that the hypotheses lead to a contradiction. Indeed, we study the execution of `(nimgame expertWPlayer pf)`. By hypothesis, this game refines `absPost (fun st' _ => is_winner false st')`. But, by `expertWPlayer_as_player_true` and `game_PlayerRefinement`, it refines `absPost (fun st' _ => is_winner true st')`. Hence, this game is magic. This is false, because `(nimgame expertWPlayer pf)` is feasible. Indeed, we know that the game `(nimgame expertWPlayer pf)` is refined by `(nimgamewf expertWPlayer pf (fun _ => True) invpf)`, which is itself feasible since `pf` and `expertWPlayer` are.

Of course, a similar theorem could also be proved in the abstract view of the Nim game. But, we can not derive the theorem at the current level from a formulation at the abstract level, since `(absPlayer nimRule)` is a less specialized player than `(absWPlayer nimRule)`.

7.4 Refinement of the optimal strategy

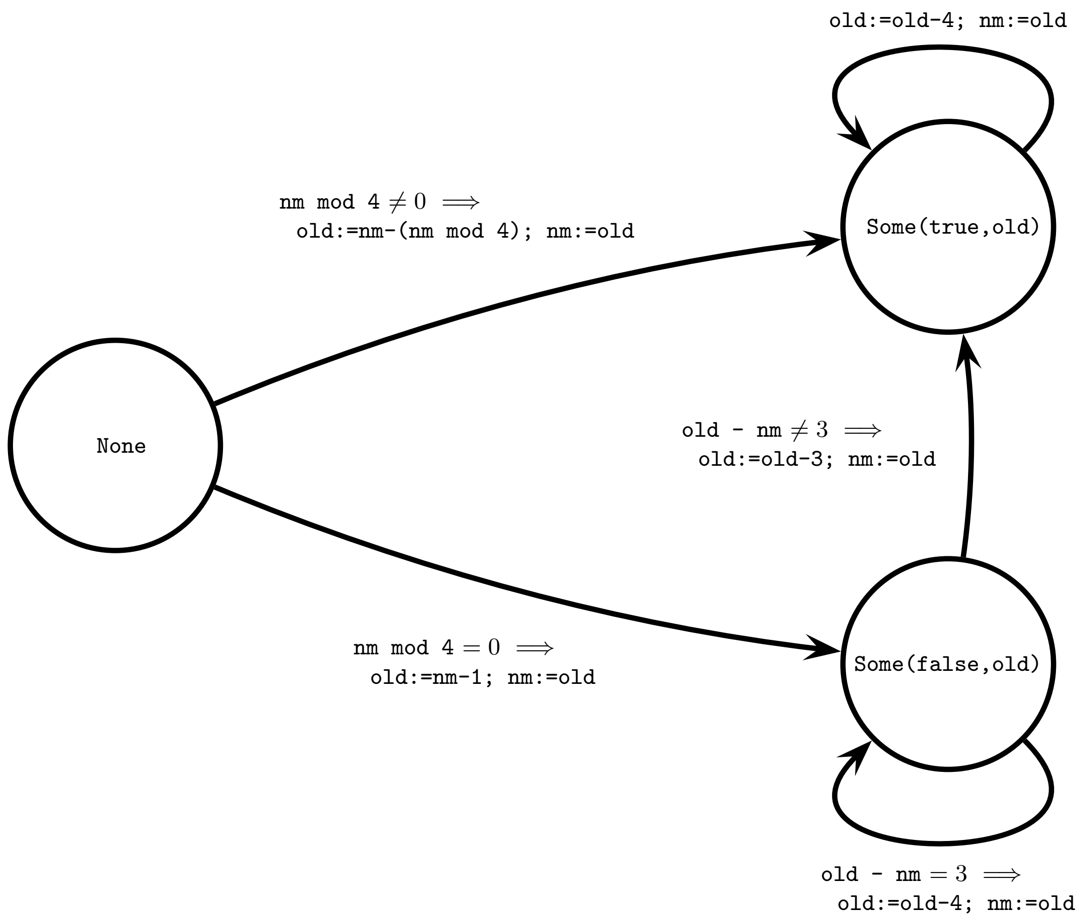
In this section, we refine `expertWPlayer` into a player called `lazyPlayer`. The main interest of this refinement is that `lazyPlayer` computes only one modulo at its first call, then it knows directly the number of matches to remove, from its previous play and the last play of its opponent. More precisely, we define `lazyPlayer` as a false player. In particular, we instantiate `WDataf` with:

```
Definition WDataf : Set := option (bool*Z).
```

Here `option` is a predefined inductive type:

```
Inductive option (A:Type) : Type :=  Some:A -> option A  |  None:option A
```

In our context, the meaning of `WDataf` is the following:
– the value `None` represents the initial value of the state, i.e. when `lazyPlayer` has not yet played.



Description of the figure
– “nm” represents the current number of matches in the stack.
– “old” represents the last number of matches left by lazyPlayer. Hence, after modifying old, lazyPlayer runs “nm:=old”.
Actually, under the assumption that the opponent follows the rule of the game (e.g. after the turn of the opponent, the number of matches nm satisfies “old-3 <= nm < old”), lazyPlayer implements the deterministic strategy “nm:=nm-max(1,nm mod 4)”.

Figure 3: Deterministic automaton implemented by lazyPlayer

– a value Some(b,old) means that lazyPlayer has already played and that, at its last turn, it left old matches in the stack. Moreover, if b then lazyPlayer will necessarily win, otherwise it may lose.
The behavior of lazyPlayer is described by the automaton of figure 3.

The transition function of the automaton given figure 3 is defined in Coq by:

```
Definition runLazyPlayer (nm:Z) (ist:WDataf) : bool*Z :=
  match ist with
  | None => let x := nm mod 4 in if (Zeq_bool x 0) then (false,nm-1) else (true,nm-x)
  | Some (true,old) => (true,old-4)
  | Some (false,old) => if (Zeq_bool (old-nm) 3) then (false,old-4) else (true,old-3)
  end.
```

Then, lazyPlayer is defined by:

```
Definition lazyPlayer: K GameSt unit
 := bind (get _) (fun st =>
       let r := (runLazyPlayer (nbm st) (wrf st)) in
       set (mkGameSt (player st) (mkGameData (snd r) (wrt st) (Some r)))).
```

To prove that `lazyPlayer` refines `expertWPlayer`, we must introduce the invariant below. In particular, we express in this invariant that `old` is the previous number of matches left by `lazyPlayer`, and that in the state `Some(true,old)`, `lazyPlayer` wins (e.g. `old mod 4 = 0`), and conversely that in the state `Some(false,old)`, `lazyPlayer` has previously chosen to remove one matches.

```
Definition invLazyPlayer (st:GameSt): Prop :=
  match (wrf st) with
  | None => True
  | Some (b,old) => old - 3 <= (nbm st) < old
             /\ if b then old mod 4=0 else old mod 4 = 3
  end.
```

We prove that `lazyPlayer` preserves the invariant under the assumption that player true respects the rule of the game.

```
Lemma lazyPlayer_preserves_invLazyPlayer: forall st,
  (player st)=false -> 0<(nbm st) ->
    [< st >] lazyPlayer >==> (invPlayer lazyPlayer invLazyPlayer (WRule nimRule)).
```

We also prove that under the assumption of the invariant, `lazyPlayer` refines `expertWPlayer` (in the false position).

```
Lemma lazyPlayer_refines_expertWPlayer_aux: forall st,
   (player st)=false -> 0 < (nbm st) -> (invLazyPlayer st) ->
    [< st >] lazyPlayer >==> expertWPlayer.
```

At last, we can glue together the previous results, and derive from `game_PlayerRefinement` that a game against `lazyPlayer` refines a game against `expertWPlayer`:

```
Theorem lazyPlayer_refines_expertWPlayer: forall pt invpt,
    (forall st, (player st)=true -> 0<(nbm st) ->
                 [< st >] pt >==> invPlayer pt invpt (WRule nimRule))
 -> (forall st, (player st)=true -> 0<(nbm st) -> (invpt st) ->
                 [< st >] pt >==> (absWPlayer nimRule))
 ->  forall st, (wrf st)=None
    -> (if (player st)
        then (invpt st)
        else forall st', (WRule nimRule st (dualSt st'))
                           -> (player st')=true -> (invpt st'))
    ->  [< st >] (nimgame pt lazyPlayer) >==> (nimgame pt expertWPlayer).
```

7.5 Conclusion of this example: interest of higher-order refinement

In the previous example, I represent players as higher-order parameters. I discuss now the advantage of this representation with respect to other representations in standard refinement calculi.

The idea to use refinement calculus in modelization of two-players game (including the Nim game) can be found in [Bac98]. However, on the contrary of the approach presented here, the calculus of [Bac98] does not support higher-order parameters. The modelization of [Bac98] uses instead the fact that the two non-determistic operators (which would correspond here to `choice` and `sync`) are perfectly symmetric. Hence, the duality between players is expressed by the duality of non-determinism.

This modelization can not be reused here, since our operator `sync` is too weak. Actually, embedding the refinement calculus of [Bac98] in Coq seems to require Excluded-Middle. Moreover, the modelization of [Bac98] is elegant, but we do not see how to adapt it to three-players games without changing the calculus. On the contrary, the approach presented above can be adapted to n-players games, using an enumerated types with n elements instead of booleans to represent positions of players.

Actually, representing players as higher-order parameters seems a very expressive approach. For instance, in event-B (which does not support higher-order parameters), players can be represented as events. Like here, it seems possible to represent a very abstract notion of game, with a generic notion of players. However, it is not clear whether a property like the game symmetry can be simply expressed and applied in proofs.

More generally, higher-order refinement allows to define specific operators (`game` can be considered as an operator like `while`) and to prove some generic laws to reason about these operators. Actually, operator `while` itself is derived from the primitive operators of the state DSM.

8 Experiments with a local notion of state in Coq

I have developed a prototype of library, called ISL, that generalizes the state DSM in order to cope with a local notion of state instead of a global one. I report here briefly two small experiments with this library.

8.1 Proof of a first-order imperative function

I have compared the total-correctness proofs of a first-order imperative program using Why (see [Fil03b]) and using the ISL library. Why is a Hoare logic for a subset of Ocaml (without aliases and without higher-order imperative functions) that generates proofs obligations for several theorem provers, including Coq. The program under experiment was a naive sorting function on arrays. This program involves two while-loops and uses some intermediate functions. I have first proved its total-correctness using Why and then, I have proved it using my ISL library (see [Bou06]). The fact that Coq is the core logic of the two proofs allows a simple comparison between them.

As expected, proof obligations in both developments are quite similar. However, Why performs more automation than the ISL library. Indeed, Why generates directly 15 proofs obligations. On the contrary, with the ISL library, the user has to drive the prover in order to reduce the refinement goal to these proofs obligations. Indeed, for function calls, my wp-computations unfold the definition of functions (functions are used as white-boxes) whereas in Why, wp-computations use their specification (functions are used as black-boxes). Hence, here, before to apply wp-simplifications, the user has to replace functions by their specification using monotonicity rules. In future works, we may imagine a mechanism to perform this task automatically.

8.2 Higher-order reasoning with a local state

This section illustrates that some ML programs can not be naturally represented in the state DSM because the state is global.

I have proved in Coq that, for the `eratosthene` function written in ML below, if `n` is greater than 2, the expression `(eratosthene f n)` calls `f` on successive prime numbers until `n` (see [Bou06]). A notion of local state is needed here, because `eratosthene` is parametrized by the state of its functional parameter `f` and this state is locally extended in the recursive call.

```
let rec eratosthene (f:int->unit) (n:int) =
  if n=2
  then
    (f 2)
  else
    let prime=ref true in
      eratosthene (fun p -> (if n mod p = 0 then prime:=false); f p)
                  (n-1) ;
      if !prime then (f n)
```

8.3 Conclusion of these experiments

The syntax introduced by ISL remains heavy. In order to improve it, I am trying to embed an effect inference mechanism into Coq. This is a key issue for the practicability of refinement calculus in Coq.

9 Conclusion

The concept of *stepwise refinement* (or top-down program development) has been invented by Dijkstra [Dij69], and promoted by Wirth [Wir71]. *Weakest-preconditions* have also been invented by Dijkstra. He uses them to give a semantics to its language of *guarded commands* [Dij75]. This language has inspired the specification language of the *refinement calculus* formalized in [Bac78]. Then, the calculus has been developed by many other authors. In particular, [Abr96] extends the approach by providing a notion of module, called *machine*, and thus allowing to use this approach in real developments of software. And

Type theory	**Refinement calculus**
Suitable as foundations of mathematics	Assumes a pre-existing logic
A logic for general theorem proving	A logic for program proving only
Handles only purely functional programs	May support many programming features
Specifications are abstract algorithms	
Programs are special cases of specifications	
Proofs mix deductions and computations	
Evaluation of specifications is made of lambda calculus and rewriting rules	Evaluation of specifications is made of weakest precondition calculus
Deduction rules form a sequent calculus	Deduction rules form a refinement calculus

Figure 4: Type theory versus refinement calculus

[Bun97] presents a refinement for a higher-order expressions language which shares many aspects with DSM theory.

In parallel, [Gor88] promoted the use of higher-order logics to formalize particular programming logics like Hoare logic. This idea has been used in many works. Among them, [Bod99] formalizes B in Coq using a pre/post semantics. Actually, [vW94] inspired my first attempts to formalize refinement calculus in Coq. I was also interested in combining monads with Hoare logic [Fil03a, Sch03, Nan06] and encoding WP as CPS [Aud99].

With respect to all these works, the main contributions of my formalization seem to be:
– extend refinement calculus with higher-order functions and structural recursion.
– sketch a modular construction of refinement calculus using monad transformers.
– embed its fixpoint theory into constructive type theory
– interpret weakest-preconditions as continuations in an intuitionistic logic.
– propose simplification rules of refinement formulas in interactive proofs.
As a result, Coq now embeds the refinement calculus of [Mor90]. The refinement calculus of [Bac98] has stronger properties, but its perfect symmetry seems to deeply rely on the axiom of Excluded-Middle.

In conclusion, the marriage of type theory and refinement calculus presented here reveals their beautiful complementarity (see Figure 4). Hence, refinement calculus could be for Coq what monads have been to Haskell: a pure way to interact with the impure side of the world. But there is a long road ahead as mentioned in sections 6.6, 8.1 and 8.2. A good challenge for a higher-order refinement calculus is to formalize the informal refinement steps described in [Fil03c].

A long-term goal is to adapt important features of B [Abr96]: abstract machines, invariants and data-refinement. Combining abstract machines and higher-order functions would give an object-oriented flavor to the specification language.

References

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[Aud99] P. Audebaud and E. Zucca. Deriving Proof Rules from Continuation Semantics. *Formal Aspects of Computing*, 11(4):426–447, 1999.

[Bac78] R.-J. Back. *On the correctness of refinement steps in program development*. Ph.D. thesis, University of Helsinki, 1978.

[Bac98] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[Beh99] P. Behm, P. Benoit *et al.*. Météor: A Successful Application of B in a Large Project. In *FM'99*, no. 1708 in LNCS, 369–387. Springer-Verlag, 1999.

[Ben02] N. Benton, J. Hughes *et al.*. Monads and Effects. In *APPSEM 2000*, 42–122. Springer-Verlag, 2002.

[Ber04] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[Bod99] J.-P. Bodeveix, M. Filali *et al.*. A Formalization of the B-Method in Coq and PVS. In *Elec. Proc. of the B-User Group Meeting at FM'99*. 1999.

[Bou06] S. Boulmé. *Higher-Order Refinement In Coq (reports and Coq files)*. Web page: `http://www-lsr.imag.fr/users/Sylvain.Boulme/horefinement/`, 2006.

[Bou07a] S. Boulmé. Intuitionistic Refinement Calculus. In *TLCA 2007*, vol. 4583 of *LNCS*. Springer-Verlag, 2007.

[Bou07b] S. Boulmé. *Higher-Order imperative enumeration of binary trees in Coq*. Online paper: `http://.../Sylvain.Boulme/horefinement/enumBTdesc.pdf`, 2007.

[Bun97] A. Bunkenburg. *Expression Refinement*. Ph.D. thesis, Computing Science Department, University of Glasgow, 1997.

[Büc99] M. Büchi and R. Back. Compositional Symmetric Sharing in B. In J. D. J.M. Wing, J. Woodcoock, ed., *FM'99 - Formal Methods*, vol. 1708 of *LNCS*. Springer-Verlag, 1999.

[Coq88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[Coq04] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.0*. Logical Project, INRIA-Rocquencourt, 2004.

[Dij69] E. W. Dijkstra. Notes on Structured Programming. Tech. Rep. EWD249, Technical U. Eindhoven, The Netherlands, 1969.

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

[Fil99] J.-C. Filliâtre. A theory of monads parameterized by effects. Research Report 1367, LRI, Université Paris Sud, 1999.

[Fil03a] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[Fil03b] J.-C. Filliâtre. *The WHY verification tool*. LRI, 2003.

[Fil03c] J.-C. Filliâtre and F. Pottier. Producing All Ideals of a Forest, Functionally. *Journal of Functional Programming*, 13(5):945–956, 2003.

[Gor88] M. J. C. Gordon. Mechanizing Programming Logics in Higher-Order Logic. In *Current Trends in Hardware Verification and Automatic Theorem Proving*. Springer-Verlag, 1988.

[Hon05] K. Honda, M. Berger *et al.*. An Observationally Complete Program Logic for Imperative Higher-Order Functions. In *IEEE Symp. on LICS'05*. 2005.

[Ler06] X. Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP 2006)*, vol. 3924 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[Lia96] S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In *Proc. of ESOP'96*, vol. 1058, 219–234. Springer-Verlag, 1996.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[Nan06] A. Nanevski, G. Morrisett *et al.*. Polymorphism and separation in Hoare Type Theory. In *Proc. of ICFP'06*. ACM Press, 2006.

[Pey93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. of the 20th symposium on POPL*. ACM Press, 1993.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Proc. of TLCA*, no. 664 in LNCS. 1993.

[Sch03] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HasCasl. In *FASE*, vol. 2621 of *LNCS*. Springer-Verlag, 2003.

[Utt92] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. Ph.D. thesis, University of New South Wales, Kensington, Australia, 1992.

[vW94] J. von Wright. Program refinement by theorem prover. In *6th Refinement Workshop*. Springer-Verlag, 1994.

[Wad95] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, vol. 925 of *LNCS*, 24–52. Springer-Verlag, 1995.

[Wad99] P. Wadler. The marriage of effects and monads. In *Proc. of the ACM SIGPLAN ICFP'98*, vol. 34(1), 63–74. 1999.

[Wir71] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.