Formal Specification of VM and I/O devices and description validation, version 2.0

Ivar Rummelhoff
Norsk Regnesentral/Norwegian Computing Center, P.O. Box 114 Blindern, NO-0314 Oslo, Norway

2019-11-01, revised: 2020-04-17, 2020-12-14, 2023-05-25

1 Introduction

The core of this document is a formal specification of the iVM abstract machine, version 1.1 (including *CHECK*). Is has been formalized and mechanically checked using the *Coq Proof Assistant*. The full source code is publicly available at:

https://github.com/immortalvm/ivm-formal-spec

The mathematical machine description in section 2 has been extracted from the Coq formalization. This section will be included on the Piql film together with the less formal "How to Build a Machine" and precise requirements for the necessary I/O devices. Here we shall define the *interface* between these devices and the machine; but details concerning the devices themselves is beyond the scope of the current document.

We have specified the machine using *monads*, a mathematical concept much used in computer science. As a result, the specification resembles an implementation of the machine in a functional programming language. This makes it easy to confirm that O2.2 describes the same machine, provided the reader has a certain "mathematical maturity". Nevertheless, we have included precise definitions of all but the most basic conscepts in order to reduce the risk of misunderstandings. We also explain the Coq syntax when it is not obvious from the context and point out when we diverge from "standard Coq" in order to simplify the presentation (such as writing $\mathbb{1}$ instead of *unit* for the type with one element).

The iVM abstract machine is so simple that it is hard to write a C compiler targeting it directly. For this reason, we have defined an intermediate "assembly language" and an assembler which translates it into the machine language. Since this is a non-trivial transformation, we have also written a number of automatic tests for it. Instead of checking the assembler output directly, these tests run the resulting machine code on a prototype implementation of the abstract machine written in `F#`. This is a functional programming language, so the prototype implementation is virtually identical to the specification below. As a consequence, the tests also confirm the "sanity" of the machine specification.

Section 2 has been extracted from the Coq formalization using a modified version of the tool `coqdoc`, which makes it possible to mix formalized mathematics with explanations in natural language. Details in the Coq code that are not necessary to understand the specification have been left out. The complete source code can be found at the URL mentioned above.

2 Formal specification of the iVM abstract machine version 2

This section contains a mathematical definition of the abstract machine used to interpret the contents of this film. It has been formalized in a system for formal mathematics called Coq, which is based on higher-order type theory. The text in this section has been extracted from the Coq description. It involves some formal logic and type theory (where for simplicity we assume the principles of propositional and functional extensionality), but we have not included the actual proofs here.

2.1 Basic types

We will be using three simple inductive types from the Coq library (a.k.a. *unit*, *bool* and *nat*):

`Inductive` $\mathbb{1}$:= $\mid \bullet : \mathbb{1}.$	`Inductive` $\mathbb{B}$:= $\mid true : \mathbb{B}$ $\mid false : \mathbb{B}.$	`Inductive` $\mathbb{N}$:= $\mid O : \mathbb{N}$ $\mid S : \mathbb{N} \to \mathbb{N}.$

We use decimal numbers as shortcuts. For instance, 3 is an abbreviation for $S\ (S\ (S\ O))$. $\mathbb{N}$ is considered a subtype $\mathbb{Z}$, the type of (positive and negative) integers. Furthermore, `if` x `then` y `else` z means `match` x `with` $true \Rightarrow y \mid false \Rightarrow z$.

We also need some "generic" types:

`Inductive` *option* A := $\mid Some : A \to option\ A$ $\mid None : option\ A.$	`Inductive` *list* A := $\mid nil : list\ A$ $\mid cons : A \to list\ A \to list\ A.$

Here A is an arbitrary type. We use $[\,]$ and $x :: y$ as shortcuts for *nil* and *cons* $x\ y$, respectively.

We will also be using some "dependent" types, such as lists with length n, known as "vectors":

`Definition` $vector\ A\ n := \{\ u : list\ A \mid length\ u = n\ \}.$

For technical reasons, *vector* is not actually defined like this. However, we do have implicit inclusions $vector\ A\ n \hookrightarrow list\ A$ for every n. Such inclusions are known as "coercions".

2.1.1 Binary numbers

If we interpret *true* and *false* as 1 and 0, then a list or vector of Booleans can be considered a binary number in the interval $[0, 2^n)$ where $n = length\ u$.

`Equations` $fromBits\ (\_ : list\ \mathbb{B}) : \mathbb{N} :=$
  $fromBits\ [] := 0;$
  $fromBits\ (x :: u) := 2 \times fromBits\ u + x.$

This definition is formulated using the Equations extension to Coq. Observe that the least significant bit comes first. Taking *fromBits* as a coercion, we will often be using elements of $list\ \mathbb{B}$ and $vector\ \mathbb{B}\ n$ as if they were natural numbers. Conversely, we can extract the n least significant bits of any integer:

`Equations` $toBits\ (n : \mathbb{N})\ (\_ : \mathbb{Z}) : vector\ \mathbb{B}\ n :=$
  $toBits\ O\ \_ := []\ ;$
  $toBits\ (S\ n)\ z := (z\ mod\ 2 =?\ 1) :: toBits\ n\ (z\ /\ 2).$

Here $z\ mod\ 2 =?\ 1$ is *true* if the equality holds, otherwise *false*. Moreover, $/$ and *mod* are defined so that $z\ mod\ 2$ is either 0 or 1 and $z = 2 \times (z\ /\ 2) + z\ mod\ 2$ for every z. In particular, all the bits in $toBits\ n$ (-1) are *true*. Thus, $toBits\ n$ is essentially the ring homomorphism $\mathbb{Z} \to \mathbb{Z}/2^n\mathbb{Z}$.

2.1.2 Bytes and words

In this text we shall mainly be concerned with bit vectors consisting of 8, 16, 32 and 64 bits:

```
Definition 𝔹^8 := vector 𝔹 8.
Definition 𝔹^16 := vector 𝔹 16.
Definition 𝔹^32 := vector 𝔹 32.
Definition 𝔹^64 := vector 𝔹 64.
```

The elements of $\mathbb{B}^8$ are called "bytes". If we concatenate a list of bytes, we get a bit vector which represents a natural number. More precisely:

```
Equations fromLittleEndian (_: list 𝔹^8): ℕ :=
  fromLittleEndian [] := 0;
  fromLittleEndian (x :: r) := 256 × (fromLittleEndian r) + x.

Equations toLittleEndian n (_: ℤ) : vector 𝔹^8 n :=
  toLittleEndian 0 _ := [];
  toLittleEndian (S n) z := (toBits 8 z) :: (toLittleEndian n (z / 256)).
```

2.1.3 Monads

In this text a "monad" is a structure consisting of one generic type m: `Type` → `Type` and two operations that satisfy three axioms. We express this as a "type class":

```
Class Monad (m: Type → Type): Type :=
{
  ret: ∀ {A}, A → m A;
  bind: ∀ {A}, m A → ∀ {B}, (A → m B) → m B;

  monad_right: ∀ A (ma: m A), bind ma ret = ma;
  monad_left: ∀ A (a: A) B (f: A → m B), bind (ret a) f = f a;
  monad_assoc: ∀ A (ma: m A) B f C (g: B → m C),
      bind ma (λ a ⇒ bind (f a) g) = bind (bind ma f) g;
}.
```

Writing the type parameters as $\{A\}$ and $\{B\}$ means that when we apply *ret* and *bind*, the type arguments will be left implicit. Moreover, we use the following notation:

$ma \gg= f$	means	$bind\ ma\ f$
$a ::= ma;\ mb$	means	$bind\ ma\ (\lambda\ a \Rightarrow mb)$
$ma;;\ mb$	means	$bind\ ma\ (\lambda\ \_ \Rightarrow mb)$

Here _ represents a variable that is never used.

The simplest monad is the "identity monad", were $m\ A = A$ for every A:

```
#[refine, export] Instance IdMonad: Monad id :=
{
  ret A x := x;
  bind A ma B f := f ma;
}.
Defined.
```

We shall often say that a function `Type` → `Type` is a monad if the rest of the monad structure is clear from the context.

2.1.4 Monad transformers

A "morphism" between monads is a structure preserving family of functions $m_0\ A \to m_1\ A$:

```
Class Morphism m₀ `{Monad m₀} m₁ `{Monad m₁} (F: ∀ {A}, m₀ A → m₁ A) :=
{
  morph_ret: ∀ A (x: A), F (ret x) = ret x;
```

```
  morph_bind: ∀ A (ma: m₀ A) B (f: A → m₀ B),
      F (ma ≫= f) = (F ma) ≫= (λ x ⇒ F (f x));
}.
```

Here ‘$\{Monad\ m_0\}$ means that that m_0: Type → Type must have a (usually implicit) monad structure, similarly for m_1. A “monad transformer” constructs a new monad from an existing monad m such that there is a morphism from m to the new monad:

```
Class Transformer (t: ∀ (m: Type → Type) ‘{Monad m}, Type → Type): Type :=
{
  transformer_monad: ∀ m ‘{Monad m}, Monad (t m);
  lift: ∀ {m} ‘{Monad m} A, m A → (t m) A;
  lift_morphism: ∀ {m} ‘{Monad m}, Morphism _ _ lift;
}.
```

Here _ represents a term that can be deduced from the context.

We get a simple transformer by composing m with *option*:

```
Definition Opt m ‘{Monad m} A := m (option A).

#[refine, export] Instance OptionTransformer: Transformer Opt :=
{
  transformer_monad m _ :=
    {|
      ret _ x := ret (Some x);
      bind _ ma _ f := a ::= (ma: _);
                       match a with None ⇒ ret None | Some x ⇒ f x end;
    |};
  lift _ _ _ ma := x ::= ma; ret (Some x);
}.
Defined.
```

Here we define *ret* and *bind* in terms of the corresponding operations of *m*. The axioms are easy to verify. Monads of the form *Opt m* can represent computations that may fail to produce a value:

```
Definition error' {m} ‘{Monad m} {A}: Opt m A := ret None.
```

Observe that *error'* ≫= *f* = *error'* for every *f*.

2.1.5 State monad transformer

As discovered by Eugenio Moggi in 1989, monads can be used to represent computations with side-effects such as input and output. In particular, we can define a transformer which extends any existing monad with the ability to access and modify a value referred to as the “current state”:

```
Definition ST S m ‘{Monad m} A := S → m (A × S).

#[refine, export] Instance StateTransformer S: Transformer (ST S) :=
{
  transformer_monad m _ :=
    {|
      ret _ x := λ s ⇒ ret (x, s);
      bind _ ma _ f := λ s ⇒ p ::= ma s; f (fst p) (snd p);
    |};
  lift _ _ _ ma := λ s ⇒ x ::= ma; ret (x, s);
}.
Defined.
```

In other words, *ST S* is a monad transformer for every type *S*. We shall be especially interested in monads of the form *Opt* (*ST S m*) where the elements represent “computations”:

```
Section opt_st_section.
```

```
Context {m} `{Monad m} {S: Type}.
Let C := Opt (ST S m).
Definition tryGet' {A} (f: S → option A) : C A :=
  λ s ⇒ ret (f s, s).
Definition get' {A} (f: S → A) : C A :=
  λ s ⇒ ret (Some (f s), s).
Definition update' (f: S → S): C 𝟙 :=
  λ s ⇒ ret (Some •, f s).
Definition tryUpdate' (f: S → option S): C 𝟙 :=
  λ s ⇒
  match f s with
  | Some s' ⇒ ret (Some •, s')
  | None ⇒ ret (None, s)
  end.
```

We have simplified these definitions by placing them inside a "section" with a list of shared `Context` parameters and a local abbreviation C. We shall follow the convention that computations (and functions returning computations) have names ending with an apostrophe. For this reason we also define:

```
  Definition return' {A} (x: A) : C A := ret x.

End opt_st_section.
```

2.2 Generic abstract machine

We have split specification of the machine in three parts. In section 2.3 we specify the I/O operations in terms of a separate monad, *IO*, whereas in the current section we describe the parts of the machine that are independent of these operations. Finally, we put the pieces together in section 2.4.

The current state of the generic virtual machine has three components: a program counter (PC), a stack pointer (SP), and the memory contents. The memory is a collection of memory cells. Each cell has a unique address of type $\mathbb{B}^{64}$ and stores one byte of data. The addresses of the available cells should form a consecutive subset of the natural numbers, see *initialCoreState* below.

```
Record CoreState :=
  mkCoreState {
      PC: 𝔹⁶⁴;
      SP: 𝔹⁶⁴;
      memory: 𝔹⁶⁴ → option 𝔹⁸;
    }.
```

A "record" is an inductive type with a single constructor (*mkCoreState*), where we get projections for free. For example, $PC\ s : \mathbb{B}^{64}$ for every $s : CoreState$.

```
Definition initialCoreState
             (program: list 𝔹⁸)
             (argument: list 𝔹⁸)
             (start: 𝔹⁶⁴)
             (memorySize: ℕ)
             (_: start + memorySize ≤ 2^64)
             (_: length program + 8 + length argument ≤ memorySize) : CoreState → ℙ :=
  let stop := (start + memorySize)%ℕ in
  let mem := program ++ toLittleEndian 8 (length argument) ++ argument in
  λ s ⇒ PC s = start
        ∧ SP s = stop :> ℕ
        ∧ ∀ (a: 𝔹⁶⁴), start ≤ a < stop →
            ∃ x, memory s a = Some (nth (a - start) mem x).
```

In other words, a valid initial state has PC pointing to the first address of the available memory and SP pointing to the first address after the available memory. Moreover, the memory should initially contain a "program", an "argument" and between them 8 bytes containing the length of the argument. The rest of the available memory is arbitrary, but well-defined (*Some x*).

```
Section generic_machine_section.
```

We assume an I/O monad of form *Opt m*.

```
Context m `{Monad m}.

Let IO := Opt m.

Definition Comp := Opt (ST CoreState m).

Definition fromIO {A} : IO A → Comp A := lift (option A).

Instance fromIO_morphism: Morphism IO Comp fromIO.
```

Observe that *fromIO error' = error'*.

2.2.1 Memory access

It is an error to access an unavailable memory address:

```
Definition loadByte' (a: ℤ) : Comp 𝔹⁸ :=
  mem ::= get' memory;
  match mem (toBits 64 a) with
  | None ⇒ error'
  | Some value ⇒ return' value
  end.

Equations loadBytes' (n: ℕ) (a: ℤ) : Comp (vector 𝔹⁸ n) :=
  loadBytes' 0 _ := return' [];
  loadBytes' (S n) start :=
    x ::= loadByte' start;
    u ::= loadBytes' n (start + 1);
    return' (x :: u).

Definition load' (n: ℕ) (a: ℤ) : Comp ℕ :=
  bytes ::= loadBytes' n a;
  return' (fromLittleEndian bytes).
```

That is, *load'* n a s_0 = (*Some* x, s_1) if $s_0 = s_1$ and the n bytes at addresses a, ..., $a+n$-1 represent the natural number $x < 2^n$.

```
Definition storeByte' (a: ℤ) (value: 𝔹⁸) : Comp 𝟙 :=
  mem ::= get' memory;
  let u: 𝔹⁶⁴ := toBits 64 a in
  match mem u with
  | None ⇒ error'
  | _ ⇒ let newMem (v: 𝔹⁶⁴) := if v =? u then Some value else mem v in
         update' (λ s ⇒ s⟨|memory := newMem|⟩)
  end.
```

Here $s\langle|memory := newMem|\rangle$ is like to s, except that the field *memory* has been changed to *newMem*.

```
Equations storeBytes' (_: ℤ) (_: list 𝔹⁸) : Comp 𝟙 :=
  storeBytes' _ [] := return' •;
  storeBytes' start (x :: u) :=
    storeByte' start x;;
    storeBytes' (start + 1) u.

Definition store' (n: ℕ) (start: ℤ) (value: ℤ) : Comp 𝟙 :=
  storeBytes' start (toLittleEndian n value).
```

2.2.2 Program counter

Definition $setPC'$ $(a\colon \mathbb{Z})\colon$ $Comp\ \mathbb{1}$:=
  $update'$ $(\lambda\ s \Rightarrow s\langle|PC := toBits\ 64\ a|\rangle)$.

Definition $next'$ $(n\colon \mathbb{N})$: $Comp\ \mathbb{N}$:=
  a ::= get' PC;
  $setPC'$ $(a + n)$;;
  $load'$ n a.

In other words, $next'$ n loads the next n bytes and advances PC accordingly.

2.2.3 Stack

A stack is a LIFO (last in, first out) queue. The elements on our stack are 64 bits long, and SP points to the next element that can be popped. The stack grows "downwards" in the sense that SP is decreased when new elements are pushed.

Definition $setSP'$ $(a\colon \mathbb{Z})\colon$ $Comp\ \mathbb{1}$:=
  $update'$ $(\lambda\ s \Rightarrow s\langle|SP := toBits\ 64\ a|\rangle)$.

Definition pop': $Comp\ \mathbb{B}^{64}$:=
  a ::= get' SP;
  v ::= $load'$ 8 a;
  $setSP'$ $(a + 8)$;;
  $return'$ $(toBits\ 64\ v)$.

Equations $popVector'$ $(n\colon \mathbb{N})$: $Comp\ (vector\ \mathbb{B}^{64}\ n)$:=
  $popVector'$ 0 := $return'$ [];
  $popVector'$ $(S\ n)$:= u ::= $popVector'$ n; x ::= pop'; $return'$ $(x :: u)$.

Definition $push'$ $(value\colon \mathbb{Z})\colon$ $Comp\ \mathbb{1}$:=
  $a0$ ::= get' SP;
  let $a1$:= $a0$ - 8 in
  $setSP'$ $a1$;;
  $store'$ 8 $a1$ $value$.

Equations $pushList'$ $(\_\colon list\ \mathbb{Z})$: $Comp\ \mathbb{1}$:=
  $pushList'$ [] := $return'$ $\bullet$;
  $pushList'$ $(x :: u)$:= $push'$ x;; $pushList'$ u.

Since $popVector'$ returns elements in the order they were pushed, it is essentially a left inverse to $pushList'$.

2.2.4 Generic I/O interface

In the generic machine an I/O operation is simply an element io: $vector\ \mathbb{B}^{64}\ n \to IO\ (list\ \mathbb{B}^{64})$ for some n. When a corresponding operation is encountered, the machine will pop n elements from the stack, execute io on these elements, and push the result onto the stack.

Record $IO\_operation$:=
  $mkIO\_operation$ {
    $ioArgs$: $\mathbb{N}$;
    $operation$: $vector\ \mathbb{B}^{64}\ ioArgs \to IO\ (list\ \mathbb{Z})$;
  }.

Definition $from\_IO\_operation$ $(op\colon IO\_operation)$: $Comp\ \mathbb{1}$:=
  $arguments$ ::= $popVector'$ $(ioArgs\ op)$;
  $results$ ::= $fromIO$ $(operation\ op\ arguments)$;
  $pushList'$ $results$.

Context $(IO\_operations\colon list\ IO\_operation)$.

Definition $ioStep'$ $(n\colon \mathbb{N})$: $Comp\ \mathbb{1}$:=
  nth $(255 - n)$ $(map\ from\_IO\_operation\ IO\_operations)$ $error'$.

In other words, we assume a list of I/O operations that will be mapped to "opcodes" 255, 254, ...

2.2.5 Single execution step

The other opcodes of the machine are as follows:

0	*EXIT*	8	*PUSH0*	16	*LOAD1*	32	*ADD*	40	*AND*
1	*NOP*	9	*PUSH1*	17	*LOAD2*	33	*MULT*	41	*OR*
2	*JUMP*	10	*PUSH2*	18	*LOAD4*	34	*DIV*	42	*NOT*
3	*JZ_FWD*	11	*PUSH4*	19	*LOAD8*	35	*REM*	43	*XOR*
4	*JZ_BACK*	12	*PUSH8*	20	*STORE1*	36	*LT*	44	*POW2*
5	*SET_SP*			21	*STORE2*				
6	*GET_PC*			22	*STORE4*			48	*CHECK*
7	*GET_SP*			23	*STORE8*				

```
Definition exec' opcode : Comp 𝟙 :=
  match opcode with
  | NOP ⇒ return' •

  | JUMP ⇒ pop' ≫= setPC'
  | JZ_FWD ⇒
      offset ::= next' 1;
      x ::= pop';
      if x =? 0
      then pc ::= get' PC;
           setPC' (pc + offset)
      else return' •
  | JZ_BACK ⇒
      offset ::= next' 1;
      x ::= pop';
      if x =? 0
      then pc ::= get' PC;
           setPC' (pc - (1 + offset))
      else return' •
  | SET_SP ⇒ pop' ≫= setSP'
  | GET_PC ⇒ get' PC ≫= push'
  | GET_SP ⇒ get' SP ≫= push'

  | PUSH0 ⇒ push' 0
  | PUSH1 ⇒ next' 1 ≫= push'
  | PUSH2 ⇒ next' 2 ≫= push'
  | PUSH4 ⇒ next' 4 ≫= push'
  | PUSH8 ⇒ next' 8 ≫= push'

  | LOAD1 ⇒ pop' ≫= load' 1 ≫= push'
  | LOAD2 ⇒ pop' ≫= load' 2 ≫= push'
  | LOAD4 ⇒ pop' ≫= load' 4 ≫= push'
  | LOAD8 ⇒ pop' ≫= load' 8 ≫= push'
  | STORE1 ⇒ a ::= pop'; x ::= pop'; store' 1 a x
  | STORE2 ⇒ a ::= pop'; x ::= pop'; store' 2 a x
  | STORE4 ⇒ a ::= pop'; x ::= pop'; store' 4 a x
  | STORE8 ⇒ a ::= pop'; x ::= pop'; store' 8 a x

  | ADD ⇒ x ::= pop'; y ::= pop'; push' (x + y)
  | MULT ⇒ x ::= pop'; y ::= pop'; push' (x × y)
  | DIV ⇒ x ::= pop'; y ::= pop'; push' (if x =? 0 then 0 else y / x)
  | REM ⇒ x ::= pop'; y ::= pop'; push' (if x =? 0 then 0 else y mod x)
  | LT ⇒ x ::= pop'; y ::= pop'; push' (if y <? x then -1 else 0)
  | AND ⇒
```

```
      u ::= pop';
      v ::= pop';
      push' (map₂ (λ x y ⇒ x && y) u v)
  | OR ⇒
      u ::= pop';
      v ::= pop';
      push' (map₂ (λ x y ⇒ if x then true else y) u v)
  | XOR ⇒
      u ::= pop';
      v ::= pop';
      push' (map₂ (λ x y ⇒ if x then (if y then false else true) else y) u v)
  | NOT ⇒ u ::= pop'; push' (map (λ x ⇒ if x then false else true) u)
  | POW2 ⇒ n ::= pop'; push' (2 ^ n)

  | CHECK ⇒
      n ::= pop';
      if n >? 1
      then error'
      else return' •

  | n ⇒ ioStep' n
  end.
```

Here *map* and map_2 denote the "bitwise" transformations:

$$map : (\mathbb{B} \to \mathbb{B}) \to \mathbb{B}^{64} \to \mathbb{B}^{64} \qquad\qquad map_2 : (\mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}^{64} \to \mathbb{B}^{64} \to \mathbb{B}^{64}$$

Now we can define what it means for our abstract machine to perform one execution step:

```
Definition oneStep' : Comp 𝔹 :=
  opcode ::= next' 1;
  match opcode with
  | EXIT ⇒ return' true
  | _ ⇒ exec' opcode;; return' false
  end.
```

We can also run the machine for n steps or until it stops:

```
Equations nSteps' (_: ℕ) : Comp 𝔹 :=
  nSteps' 0 := return' false;
  nSteps' (S n) :=
    done ::= oneStep';
    if done
    then return' true
    else nSteps' n.

End generic_machine_section.
```

2.3 Actual input and output

In this section we define the I/O operations in terms of a corresponding monad. For simplicity, we will again use a monad of the form *Opt* (*ST S m*); but whereas a concrete implementation of the iVM machine would typically reflect *CoreState* in its program state, the I/O state might be distributed across the system as a whole. For instance, *newFrame'* (defined below) might write the current frame to disk.

2.3.1 Bitmap images

The I/O state has several components. First, an image is a two-dimensional matrix of square pixels, counting from left to right and from top to bottom.

```
Record Image (C: Type) :=
  mkImage {
```

$iWidth$: $\mathbb{B}^{16}$;
$iHeight$: $\mathbb{B}^{16}$;
$iPixel$: $\mathbb{N} \to \mathbb{N} \to option\ C$;
$iBounded$: $\forall\ x\ y,\ iPixel\ x\ y \neq None \leftrightarrow x < iWidth \land y < iHeight$;
}.

Here C represents the color of a single pixel. The arguments to *iWidth* and *iHeight* (of type *Image C*) are implicit in the type of *iBounded*. The simplest image consists of 0×0 pixels:

```
Definition noImage {C} : Image C :=
  {|
    iWidth := toBits 16 0;
    iHeight := toBits 16 0;
    iPixel := λ _ _ ⇒ None;
    iBounded := _;
  |}.
```

Gray represents the gray scale of the input images, and *Color* the colors of the output images:

```
Definition Gray := 𝔹^8.
Definition Color := 𝔹^64 × 𝔹^64 × 𝔹^64.
```

For output images under construction we use the type *Image (option Color)*. When all the pixels have been set, we can extract an image of type *Image Color*. Thus, we have a function

$$tryExtractImage : Image\ (option\ Color) \to option\ (Image\ Color).$$

The exact definition of this function in Coq is beyond the scope of text. It basically involves checking that every pixel $\neq$ *None*.

2.3.2 Sound and text

Sound represents a clip of stereo sound. Both channels have the same sample rate, and each pair of 16-bit words (*left*, *right*) contains one sample for each channel. For convenience, *sSamples* lists the samples in reverse order.

```
Record Sound :=
  mkSound {
      sRate: 𝔹^32;
      sSamples: list (𝔹^16 × 𝔹^16);
      sDefined: 0 = sRate → sSamples = [];
    }.
```

```
Definition emptySound (rate: 𝔹^32) : Sound :=
  {|
    sRate := rate;
    sSamples := [];
  |}.

Definition noSound := emptySound 0.
```

Textual output from the machine uses the encoding UTF-32. Again, we use reverse ordering.

```
Definition OutputText := list 𝔹^32.

Definition OutputBytes := list 𝔹^8.

Definition CurrentOutput := (Image (option Color)) × Sound × OutputText × OutputBytes.

Definition FlushedOutput := (Image Color) × Sound × OutputText × OutputBytes.

Definition tryFlush (co: CurrentOutput) : option FlushedOutput :=
  match co with
  | (i, s, t, b) ⇒
```

```
    match tryExtractImage i with
    | Some ei ⇒ Some (ei, s, t, b)
    | None ⇒ None
    end
  end.
```

2.3.3 The I/O monad

The complete I/O state of our machine can now be defined as:

```
Record IoState :=
  mkIoState {
      currentInput: Image Gray;
      charsRead: ℕ;
      currentOutput: CurrentOutput;
      flushedOutput: list FlushedOutput;
    }.
```

```
Definition initialIoState :=
  {|
    currentInput := noImage;
    charsRead := 0;
    currentOutput := (noImage, noSound, [], []);
    flushedOutput := [];
  |}.
```

As the machine executes, more elements will be prepended to flushedOutput. In other words, we use a reverse ordering here as well. The I/O monad is based on the identity monad:

Definition IO_0 := *ST IoState id*.

Definition *IO* := *Opt* IO_0.

2.3.4 Input operations

```
Definition readFrame' (allInputFrames: list (Image Gray)) (i: ℕ) : IO (𝔹^16 × 𝔹^16) :=
  update' ( λ s ⇒ s⟨|currentInput := nth i allInputFrames noImage|⟩ );;
  frame ::= get' currentInput;
  return' (iWidth frame, iHeight frame).
```

Here *nth i allInputFrames noImage* is the *i*th element of the list, or *noImage* if the list is too short.

```
Definition readPixel' (x y: ℕ) : IO 𝔹^8 :=
  frame ::= get' currentInput;
  match iPixel frame x y with
  | None ⇒ error'
  | Some c ⇒ return' c
  end.
```

```
Definition readChar' (allInputChars: ℕ → 𝔹^32) : IO 𝔹^32 :=
  i ::= get' charsRead;
  update' ( λ s ⇒ s⟨|charsRead := (1 + i)%ℕ|⟩ );;
  return' (allInputChars i).
```

In other words, we represent textual input from the user as an infinite stream of 32-bit characters known in advance. However, the intention is to represent interactivity.

2.3.5 Output operations

```
Definition empty (width height: 𝔹^16) : Image (option Color) :=
```

```
{|
  iWidth := width;
  iHeight := height;
  iPixel x y := if (x <? width) && (y <? height) then Some None else None;
  iBounded := _;
|}.
```

Here p && q = `if` p `then` q `else` *false*.

```
Definition newFrame' (width height rate: ℕ) : IO 𝟙 :=
  tryUpdate' (λ s ⇒
    match tryFlush (currentOutput s) with
    | Some fl ⇒
      let newCo := (empty (toBits 16 width) (toBits 16 height), emptySound rate, [], []) in
      Some (s⟨|flushedOutput := fl :: flushedOutput s|⟩ ⟨|currentOutput := newCo|⟩)
    | None ⇒ None
    end).

Definition replaceOutput o s : IoState := s⟨|currentOutput := o|⟩.


Definition trySetPixel (image: Image (option Color))
                       (x y: ℕ) (c: Color): option (Image (option Color)) :=
  if (x <? iWidth image) && (y <? iHeight image)
  then let p xx yy := if (xx =? x) && (yy =? y)
                      then Some (Some c)
                      else iPixel image xx yy in
       Some (image ⟨| iPixel := p |⟩)
  else None).

Definition setPixel' (x y r g b : ℕ) : IO 𝟙 :=
  o ::= get' currentOutput;
  match o with (image, sound, text, bytes) ⇒
    match trySetPixel image x y (toBits 64 r, toBits 64 g, toBits 64 b) with
    | Some newImage ⇒ update' (replaceOutput (newImage, sound, text, bytes))
    | None ⇒ error'
    end
  end.


Definition addSample' (l r : ℕ) : Comp 𝟙 :=
  o ::= get' currentOutput;
  match o with (image, sound, text, bytes) ⇒
    if sRate sound =? 0 then error'
    else let ns := sound ⟨| sSamples := (toBits 16 l, toBits 16 r) :: sSamples sound |⟩ in
         update' (replaceOutput (image, ns, text, bytes))
  end.

Definition putChar' (c: ℕ) : IO 𝟙 :=
  o ::= get' currentOutput;
  match o with (image, sound, text, bytes) ⇒
    update' (replaceOutput (image, sound, (toBits 32 c) :: text, bytes))
  end.

Definition putByte' (b: ℕ) : IO 𝟙 :=
  o ::= get' currentOutput;
  match o with (image, sound, text, bytes) ⇒
    update' (replaceOutput (image, sound, text, (toBits 8 b) :: bytes))
  end.
```

2.3.6 List of I/O operations

The generic machine defined in section 2.2 expects I/O operations of a certain form.

```
Equations nFun (n: ℕ) (A B: Type): Type :=
  nFun O _ B := B;
  nFun (S n) A B := A → (nFun n A B).
```

In other words, $nFun\ n\ A\ B = \underbrace{A \to A \to \ldots \to}_{n} B$.

```
Equations nApp {n A B} (f: nFun n A B) (v: vector A n): B :=
  nApp y [] := y;
  nApp f (x :: v) := nApp (f x) v.
```

```
Definition io_operation n f := {| operation := nApp (f: nFun n 𝔹⁶⁴ (IO (list ℤ))) |}.
Definition IO_operations (allInputFrames: list (Image Gray)) (allInputChars: ℕ → 𝔹³²) :=
  [
    io_operation 1 (λ i ⇒ wh ::= readFrame' allInputFrames i; return' [fst wh : ℤ; snd wh : ℤ]);
    io_operation 2 (λ x y ⇒ p ::= readPixel' x y; return' [p: ℤ]);
    io_operation 3 (λ w h r ⇒ newFrame' w h r;; return' []);
    io_operation 5 (λ x y r g b ⇒ setPixel' x y r g b;; return' []);
    io_operation 2 (λ l r ⇒ addSample' l r;; return' []);
    io_operation 1 (λ c ⇒ putChar' c;; return' []);
    io_operation 1 (λ c ⇒ putByte' c;; return' []);
    io_operation 0 (c ::= readChar' allInputChars; return' [c: ℤ])
  ].
```

2.4 Integration

Putting everthing together, we can say what it means to run the machine from start to finish.

```
Definition State := CoreState × IoState.
```

```
Definition finalState
               program argument (start: 𝔹⁶⁴) (memorySize: ℕ) inputFrames inputChars
               (Ha: start + memorySize ≤ 2^64)
               (Hb: length program + 8 + length argument ≤ memorySize)
               (cs: CoreState)
               (Hc: initialCoreState program argument start memorySize Ha Hb cs) : State → ℙ :=
  let ioOps := IO_operations inputFrames inputChars in
  λ s ⇒ ∃ n, nSteps' ioOps n cs initialIoState = ((Some true, fst s), snd s).
```

This is a bit imprecise: If the machine terminates normally (encountering *EXIT*), it should also try to flush the current output before terminating, cf. *tryFlush* above.

finalState is a partial function in the following sense:

```
Lemma finalState_unique: ∀ p a st ms i c Ha Hb cs Hc s₁ s₂,
    finalState p a st ms i c Ha Hb cs Hc s₁ → finalState p a st ms i c Ha Hb cs Hc s₂ → s₁ = s₂.
```

In a framework with general recursion we could have defined this partial function directly.