

# Specification and (nearly complete) Proofs for Registers with Bounded Types

Dan Pittman dan@auxon.io

September 20, 2019

## Abstract

A literate Agda file containing a specification-as-types as well as proofs of the bounds properties of the registers library.

## Introduction

Register access and manipulation exists at the lowest levels of a software stack. due to its foundational nature, its correctness should be considered to be of the utmost importance. When software interacts with registers on the hardware, it does so through the manipulation of data referred to by *a priori* specified pointers. It is common practice to treat the data at these pointers as unsigned integers whose width is determined by the register's width.

We then naturally arrive at a bounded range of positive integers which can be represented given the number of bits available. Denotationally, we can think of these unsigned integers as the natural numbers less than or equal to some upper bound.

## Ranges

When we think about ranges logically, we'd normally consider them as the set:

$$\text{Range}(l, u) := \{x \mid l \leq x \leq u\}$$

However, when working within the confines of Rust's type system, it is easier to consider a range like so:

$$\text{Range}(l, u) := \{x \mid x \geq l \wedge x \leq u\}$$

The following type represents the statement above. Its constructor requires a proof that  $x \geq l$  as well as a proof that  $x \leq u$ . From those proofs, we build a

new type, `InRange`, parameterized by its lower bound, the value, and its upper bound.

```
data InRange (l u : ℕ) : Set where
  in-range : (x : ℕ) → (x ≥ l) → (x ≤ u) → InRange l u

get-rv : ∀ {l u} → InRange l u → ℕ
get-rv (in-range x _ _ ) = x
```

## Registers

As stated in the introduction, we intend to enforce some properties about interacting with registers. Typically, a register is made of up fields, which occupy some range of bits present in that register. When reading, writing, or modifying a register, the preferred API is one which delineates the fields such that the programmer can consider the fields' *logical* values, rather than the value relative to their position in the register. In the following section, we will cover the logical representation of these register fields.

### Fields

For simplicity's sake, we will illustrate with an 8-bit register with three fields, `On`, `Dead`, and `Color`.

0	1	2	3	4	5	6	7
On	Dead	Color			Unused		

When we interact with these fields, we'd like to do so with their logical values—0 or 1 for `On` and `Dead`, and the range 0...7 for `Color`—as opposed to their values relative to their position in the register. Because of this expectation, we are exposed to the possibility of invoking a field's API with a value which exceeds its upper bound. Therefore, we'd like for the API we expose to disallow such a possibility altogether, and therefore preventing undefined behavior from rearing its ugly head.

To compute a field's range, we take its width in the register in which it resides and use it to compute the maximum logical value. This is a simple base-two shifting operation:

$$(1 \ll \text{width}) - 1$$

We'll need to define our own `«`. Once we do, we can define a type which uses `InRange` to represent a field in a register.

```
_«_ : ℕ → ℕ → ℕ
0 « _ = 0
```

```

{-# CATCHALL #-}
x « (suc y) = (x * 2) « y
{-# CATCHALL #-}
x « 0 = x

width-max : ℕ → ℕ
width-max w = ((1 « w) - 1)

data Field : (w o : ℕ) → Set where
  mk-field : {w o : ℕ} → InRange 0 (width-max w) → Field w o

field-mask : ∀ {w o} → Field w o → ℕ
field-mask {w} {o} _ = (width-max w) « o

```

Now, because `Field` requires an `InRange` proof, we can think it merely a refinement to the more general case we've proven above. Let's prove an example of a field with our register above.

We begin with a proof that the third color, whose `Color`-relevant value is 2, does indeed fit into the `Color` field.

```

color-can-be-two : 2 ≤ (width-max 3)
color-can-be-two = s≤s (s≤s z≤n)

```

Then, we can use that proof when constructing a `Color` field whose value is 2.

```

color-is-two : Field 3 2
color-is-two = mk-field (in-range 2 n≥z color-can-be-two)

```

## A Register is the sum of its parts

We begin with a binary representation of  $\mathbb{N}$ , where the least significant bit is first. Through this definition, we can do bitwise operations on the registers and their fields.

```

data Bin : Set where
  end : Bin
  zero : Bin → Bin
  one : Bin → Bin

inc-bin : Bin → Bin
inc-bin end = one end
inc-bin (one b) = zero (inc-bin b)
inc-bin (zero b) = one b

nat→bin : ℕ → Bin

```

```

nat→bin 0 = end
nat→bin (suc n) = inc-bin (nat→bin n)

bin→nat : Bin → ℕ
bin→nat end = 0
bin→nat (zero b) = 2 * bin→nat b
bin→nat (one b) = 1 + 2 * (bin→nat b)

```

Next, we prove that our  $\mathbb{N} \rightarrow \text{Binary} \rightarrow \mathbb{N}$  conversion abides the initially given  $\mathbb{N}$  for all  $\mathbb{N}$ .

```

lemma-suc-inc : ∀ b → bin→nat (inc-bin b) ≡ suc (bin→nat b)
lemma-suc-inc end = refl
lemma-suc-inc (zero b) = refl
lemma-suc-inc (one b) =
  begin
    bin→nat (inc-bin b) + (bin→nat (inc-bin b) + 0)
  ≡⟨ cong (λ x → ( x + (bin→nat (inc-bin b) + 0))) (lemma-suc-inc b) ⟩
    suc (bin→nat b) + (bin→nat (inc-bin b) + 0)
  ≡⟨ cong (λ x → ( suc (bin→nat b) + (x + 0))) (lemma-suc-inc b) ⟩
    suc (bin→nat b) + (suc (bin→nat b) + 0)
  ≡⟨ cong suc (+suc (bin→nat b) ((bin→nat b) + 0)) ⟩
    suc (suc (bin→nat b + (bin→nat b + 0)))
  ■ where open ≡-Reasoning

nat→bin→nat : ∀ n → bin→nat (nat→bin n) ≡ n
nat→bin→nat zero = refl
nat→bin→nat (suc n) =
  begin
    bin→nat (nat→bin (suc n))
  ≡⟨ ⟩
    bin→nat (inc-bin (nat→bin n))
  ≡⟨ lemma-suc-inc (nat→bin n) ⟩
    suc (bin→nat (nat→bin n))
  ≡⟨ cong suc (nat→bin→nat n) ⟩
    suc n
  ■ where open ≡-Reasoning

_&b_ : Bin → Bin → Bin
_&b end = end
(one x) &b (one y) = one (x &b y)
(zero x) &b (one y) = zero (x &b y)
(one x) &b (zero y) = zero (x &b y)
(zero x) &b (zero y) = zero (x &b y)
{-# CATCHALL #-}
end &b _ = end

```

```

_&_ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ 
 $x \& y = \text{bin} \rightarrow \text{nat} ((\text{nat} \rightarrow \text{bin } x) \& \text{b } (\text{nat} \rightarrow \text{bin } y))$ 

_»_ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ 
0 » _ = 0
(suc x) » (suc y) =  $\lfloor (\text{suc } x) / 2 \rfloor$  » y
{-# CATCHALL #-}
(suc x) » 0 = (suc x)

```

We know that a `Field` carries with it a proof that its value resides within its bounds. When we consider a register as merely the composite of its fields, we, through construction, have a safe API to the register itself, because our only interaction with the register is through its fields. This can be demonstrated by asserting that the summation of the widths of the fields said to be contained in a register is  $\leq$  the total width of the register.

```

data Register : (w :  $\mathbb{N}$ ) → Set where
  end :  $\forall \{w\} \rightarrow \text{Register } w$ 
  with-field :  $\forall \{w \text{ fw } fo : \mathbb{N}\} \rightarrow$ 
    Field fw fo →
    fw + fo ≤ w →
    Register w →
    Register (w - fw)

```

Here, the construction of a register is an inductive family where each field added to it deducts from the available width for the register. With this as our type for a register, we know with certitude:

1. Only fields which fit in a register can be said to reside within that register, that's the  $fw + fo \leq w$  part.
2. The constraints put on a field by this definition allow us to prove that interaction with fields never contravene their bounds. We demonstrate that with a proof of reading any arbitrary field in any register below in `read-prf`.

The operation `(RegVal & FieldMask) >> FieldOffset` tells us how to read a field from a register. Before we can get to a proof regarding this operation, we first must get some lemmas out of the way.

First, we prove that `&-ing` any value produces a value less than or equal to that value.

```

&-≤ :  $\forall n m \rightarrow (n \& m) \leq m$ 
&-≤ n zero rewrite &-zero n = z≤n
&-≤ n (suc m) = begin
  n & (suc m) ≡⟨ sym (cong (λ x → (x & (suc m))) (nat → bin → nat n)) ⟩
  (bin → nat (nat → bin n)) & (suc m)

```

```

≡⟨ sym (cong (λ x → ((bin→nat (nat→bin n)) & x)) (nat→bin→nat (suc m))) ⟩
(bin→nat (nat→bin n)) & (bin→nat (nat→bin (suc m)))
≤⟨ &b-≤ (nat→bin n) (nat→bin (suc m)) ⟩
bin→nat (nat→bin (suc m)) ≡⟨ nat→bin→nat (suc m) ⟩
(suc m)
■ where open ≤-Reasoning

```

Next, we prove that a right or left shift by zero preserves the value on the left-hand side.

```

«-identity : ∀ n → n « 0 ≡ n
«-identity 0 = refl
«-identity (suc n) = refl

»-identity : ∀ n → n » 0 ≡ n
»-identity 0 = refl
»-identity (suc n) = refl

```

Now, a proof that shifting a value left then right by the same value yields the initial value.

```

«-cancel : ∀ n m → (n « m) » m ≡ n
«-cancel 0 m rewrite shl-zero m | shr-zero m = refl
«-cancel (suc n) m rewrite suc-«-cancel n m = begin
  suc ((n « m) » m) ≡⟨ cong suc («-cancel n m) ⟩
  suc n
■ where open ≡-Reasoning

```

Now, finally, we discharge each of those lemmas in a proof that register reads abide their bounds.

```

read-prf : ∀ fw rv fo → (rv & ((width-max fw) « fo)) » fo ≤ (width-max fw)
read-prf zero rv fo rewrite width-max-zero | shl-zero fo | &-zero rv | shr-zero fo = z≤n
read-prf (suc fw) rv fo = begin
  (rv & ((width-max (suc fw)) « fo)) » fo
  ≤⟨ »-≤ (&-≤ rv ((width-max (suc fw)) « fo)) ⟩
  ((width-max (suc fw)) « fo) » fo ≡⟨ «-cancel (width-max (suc fw)) fo ⟩
  width-max (suc fw)
  ■ where open ≤-Reasoning

```