The `bits` Function Family

by Sven Nilsen, 2017

Some advanced path semantics require working with power-sets. A power-set is the set of all sets that can be constructed from some elements. Another way to explain it is the set of all sub-sets of a set.

A set either contains an element or not, so the ideal way of representing the membership relation information is by using a bit. The binary encoding of natural numbers using N bits can be interpreted as N elements where each bit tells whether they belong to a sub-set or not.

To iterate through all sets in a power-set, one can count from `0` up to `$2^N$-1` and get the binary encoding of each number. Each bit will then tell whether the element is a member of the sub-set or not.

The `bitsm` function is used to construct the binary representation of a natural number modulus N. The first argument tells how many bits that are used to encode the number. The second argument is a number to be encoded.

> bitsm : nat → nat → [bool]
>
> bitsm := \(n : nat) = \(x : nat) = sift i n { (floor(x / 2^i) % 2) == 1 }

The `sift` loop creates an array/list by collecting items from the loop body.

For example, `bitsm(3)` returns the following values:

> 0: [false, false, false]
> 1: [true, false, false]
> 2: [false, true, false]
> 3: [true, true, false]
> 4: [false, false, true]
> 5: [true, false, true]
> 6: [false, true, true]
> 7: [true, true, true]

The `bits` function is identical to `bitsm` except that the number to be encoded must be less than `$2^N$`.

> bits : nat → nat → [bool]
>
> bits := \(n : nat) = \(x : nat ∧ (< 2^n)) = bitsm(n)(x)

To count downwards from the largest sub-set to the empty sub-set, one can use `bitsmd` and `bitsd`:

> bitsmd := \(n : nat) = \(x : nat) = bitsm(n)(2^n-1-x)
>
> bitsd := \(n : nat) = \(x : nat ∧ (< 2^n)) = bitsmd(n)(x)

Alternative form for `bitsmd` that is defined directly:

$$bitsmd := \backslash(n : nat) = \backslash(x : nat) = sift\ i\ n\ \{\ (floor(x / 2^i)\ \%\ 2) == 0\ \}$$

One could also flip each bit in the binary representation and get the reverse count, because:

$$(\ (floor(x / 2^i)\ \%\ 2) == 1\ ) <=> (\ odd(floor(x / 2^i))\ )$$
$$(\ (floor(x / 2^i)\ \%\ 2) == 0\ ) <=> (\ even(floor(x / 2^i))\ )$$

even[not] <=> odd

Alternative form for `bitsd` that uses `bitsm` instead of `bitsmd`:

$$bitsd := \backslash(n : nat) = \backslash(x : nat \wedge (< 2^n)) = bitsm(n)(2^n\text{-}1\text{-}x)$$

When using a function from the `bits` family, one can write the following:

bits(n)(i)[j]

Sometimes, because the `bits` function has a constraint that can match a loop with range `[0, $2^n$)`, the number of bits `n` can be inferred from the loop, so one can skip the first argument and write:

bits(i)[j]

A shorthand notation for this is:

$$bits_{ij}$$

Or, when this is frequently used, replacing the name “bits” with the greek beta character `β`:

$$\beta_{ij} <=> bits_{ij}$$
$$\beta_i <=> bits_i$$

When counting up or down, one can use a plus or minus superscript:

$$\beta^-{}_{ij} <=> bitsd_{ij}$$

$$\beta^+{}_{ij} <=> bits_{ij}$$

$$\beta^-{}_{ij} <=> \neg\beta^+{}_{ij}$$
$$\beta^+{}_{ij} <=> \neg\beta^-{}_{ij}$$

One should notice that this can be calculated directly:

$$\beta_{ij} <=> (floor(i / 2^j)\ \%\ 2) == 1$$

Using integer division:

$$\beta_{ij} <=> odd(i / 2^j)$$

Sometimes one would like to use `0` and `1` instead of `false` and `true`, for example, when summing or multiplying a bit with another number:

$\sum$ i 2^n, j { β_{ij} }

$\beta_{ij} \cdot k$

This is solved by treating the function `β` as one of several variants, depending on the type of number one wants to construct:

β_A : {n : nat} → nat ∧ (< 2^n) → A

$\beta_{\mathbb{B}}$: {n : nat} → nat ∧ (< 2^n) → bool
$\beta_{\mathbb{C}}$: {n : nat} → nat ∧ (< 2^n) → complex
$\beta_{\mathbb{Z}}$: {n : nat} → nat ∧ (< 2^n) → int
$\beta_{\mathbb{N}}$: {n : nat} → nat ∧ (< 2^n) → nat
$\beta_{\mathbb{Q}}$: {n : nat} → nat ∧ (< 2^n) → rational
$\beta_{\mathbb{R}}$: {n : nat} → nat ∧ (< 2^n) → real

In probability theory there is a common pattern where you flip a probability depending on a bit.

if β_{ij} { p_{kj} } else { 1 – p_{kj} }

This can be written in short form where you return a function of type `real → real`. This function flips the probability depending on the value of the bit:

$\beta_{ij}(p_{kj})$ <=> if β_{ij} { p_{kj} } else { 1 – p_{kj} }

$\beta_{\mathbb{R}\to\mathbb{R}}$: {n : nat} → nat ∧ (< 2^n) → (real → real)