Adders and Arrows

Conal Elliott

Draft version of 2021-10-09 10:01 GMT-8

1 Numbers and addition

This note takes a few steps in a journey to construct machine-verified hardware design in a simple, principled manner. As a modest first goal, let's consider addition on n-bit binary representations of natural numbers. While we could rush directly toward a single algorithm, we will instead acquaint ourselves with the conceptual territory, gathering insights that point to a wide variety of efficient algorithms.

Rigorous (and thus dependable) algorithm development begins with precise specification.[1] As our fundamental specification, let us adopt a *unary* representation of natural numbers (built up from zero by repeated applications of successor) and a correspondingly recursive definition of addition:[2,3]

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

infixl 6 _+_
_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)
```

Given that we are seeking efficient implementations, it may seem perverse to choose such an inefficient starting point! Specifications, however, *define* correctness and so are the only aspect of the development that cannot be proved correct. For this reason, it is the aspect most important to keep as simple as possible. We can overcome any inefficiency of specification through with clever alternative representations and implementations, as long as we prove their consistency with an original simple specification. In fact specifications can be "inefficient" to the extreme of being not even computable, as long as they are (mathematically) well-defined. In contrast, a cleverly efficient specification (starting point) should raise doubts that we even understand what it is that we are proving.

Put differently, we want to optimize specifications for their *denotational* clarity and optimize implementations for their *operational* efficiency. It is the role of *proof* to connect denotation and operation with certainty.[4]

[1] Precision is essential so that we do not fool ourselves. "The first principle is that you must not fool yourself, and you are the easiest person to fool." - Richard P. Feynman

[2] This axiomatization of natural numbers and corresponding arithmetic operations were discovered by Giuseppe Peano in the late nineteenth century.

[3] All code in these notes are written in Agda (version 2.6.2), a dependently typed programming language and proof assistant. Where available, definitions are adopted from the Agda standard library (agda-stdlib, version 1.7). All indented (non-inline) code in this document is type-checked (and thus proof-checked) during typesetting.

[4] "Tis much better to do a little with certainty & leave the rest for others that come after than to explain all things by conjecture without making sure of any thing." - Isaac Newton

2 Bounded numbers

Efficient implementations (especially in hardware) typically aim at *statically bounded* subsets of natural numbers. For any $n : \mathbb{N}$, the type $\mathbb{F}\ n$ contains all natural numbers i such that $0 \le i < n$:[5]

```
data 𝔽 : ℕ → Set where
  zero : {n : ℕ} → 𝔽 (suc n)
  suc  : {n : ℕ} → 𝔽 n → 𝔽 (suc n)
```

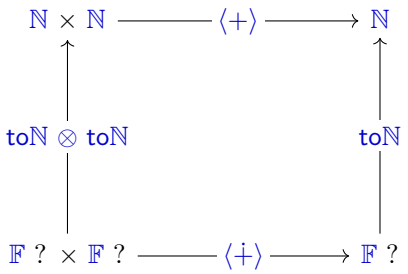
For instance,

- The type $\mathbb{F}$ zero is uninhabited.
- The type $\mathbb{F}$ (suc zero) contains exactly the value zero.
- The type $\mathbb{F}$ (suc (suc zero)) contains exactly the values zero and suc zero.

With types $\mathbb{N}$ and $\mathbb{F}\ m$ in hand we can ask what $\mathbb{F}$ operation $\dot{+}$ is *analogous* to the $\mathbb{N}$ operation $+$. To make sense of this question, we need to establish precisely how we intend $\mathbb{F}\ n$ to "represent" $\mathbb{N}$. There are many possible choices we might make, each leading to a different precise question and correct corresponding answer. A simple and useful choice is suggested by the like constructor names:

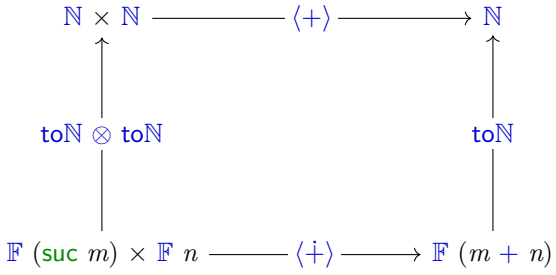
```
toℕ : ∀ {n} → 𝔽 n → ℕ
toℕ zero    = zero
toℕ (suc i) = suc (toℕ i)
```

If we use "$\langle + \rangle$" and "$\langle \dot{+} \rangle$" to refer to the uncurried versions of the known and unknown addition operations on $\mathbb{N}$ and $\mathbb{F}$ respectively, then the analogy we wish to fulfill is embodied in the following diagram:



where $(f \otimes g)\ (x\ ,\ y) = f\ x\ ,\ g\ y$.

What bounds should we use for addition on $\mathbb{F}$? If we add $a : \mathbb{F}\ m$ to $b : \mathbb{F}\ n$, then $0 \le a < m$ and $0 \le b < n$, so $0 \le a \dot{+} b < m + n - 1$, i.e., has type $\mathbb{F}\ (m + n - 1)$. (The "-1" is due to working with whole numbers rather than reals.) This formulation does not quite work, however, since $\mathbb{N}$ has no negatives and so does not have subtraction in the way we might expect. We can instead simply work with suc m in place of m:

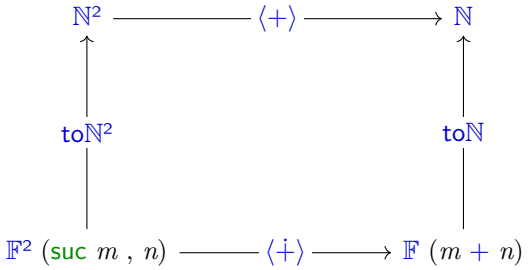


[5] Alternatively, $\mathbb{F}\ n$ is sometimes described as the type of finite sets of size n.

Because we will have many more such diagrams—and their corresponding textual forms—it will be convenient to define and use some shorthands:

```
ℕ² : Set
ℕ² = ℕ × ℕ

𝔽² : ℕ² → Set
𝔽² (m , n) = 𝔽 m × 𝔽 n

toℕ² : ∀ {m,n} → 𝔽² m,n → ℕ²
toℕ² = toℕ ⊗ toℕ
```

With these notational conveniences, the required property of $\mathbb{F}$ addition ($\langle\dot{+}\rangle$) becomes somewhat more compact:



In code,

```
toℕ-∔ : ∀ {(m , n) : ℕ²} → toℕ {m + n} ∘ ⟨∔⟩ ≗ ⟨+⟩ ∘ toℕ² {suc m , n}
```

where "$\mathring{=}$" refers to extensional equality of functions. Solving this equation for $\langle\dot{+}\rangle$ yields the following recursive definition:[6]

```
infixl 6 _∔_
_∔_ : ∀ {m n} → 𝔽 (suc m) → 𝔽 n → 𝔽 (m + n)
_∔_ {m}      zero    j = inject+′ m j
_∔_ {suc _} (suc i) j = suc (i ∔ j)
```

Let's consider the meaning of an $\mathbb{F}$ value to be the corresponding $\mathbb{N}$ value, as given by toℕ. Then toℕ-∔ says that *the meaning of the sum is the sum of the meanings*. The property has a rhyme to it that may sound familiar if you've seen abstract algebra, with its various flavors of *homomorphisms*.

3 Packaging it all up to go

We now have five crucial pieces of information about addition on $\mathbb{F}$:

1. an implementation (bottom arrow, $\langle\dot{+}\rangle$),
2. a specification (top arrow, $\langle+\rangle$),
3. a mapping of implementation input to specification input (left arrow, toℕ²),
4. a mapping of implementation output to specification output (right arrow, toℕ), and
5. a proof of consistency of the pieces above (commutativity condition, toℕ-∔).

[6] Whenever a signature (theorem) in this note is given without definition (proof)—as with toℕ-∔—the definition (proof) is given in the source code.

These five pieces are all aspects of a single, meaningful assembly, so let's wrap them into a convenient package. Parts 3 and 4 are about the inputs and outputs and their semantic relationship and so form the assembly's interface. Parts 1, 2, and 5 become the details behind that interface:

```
+⇉ : ∀ {(m , n) : ℕ²} → toℕ² {suc m , n} ⇉ toℕ {m + n}
+⇉ = arr ⟨+⟩ ⟨+⟩ toℕ-+
```

The symbol "⇉" is intended to suggest a kind of mapping, belonging to a category in which

- *objects* (classifying inputs and outputs for the category) are data mappings (parts 3 and 4 above); and
- *morphisms* (the connections/mappings in the category) are pairs of functions (parts 1 and 2 above)—which can really be morphisms from *any* category—that satisfy a commutative diagram (part 5 above).

This general construction is known as an *arrow category* and offers useful means for composing *horizontally* (i.e., by horizontal abutment with matching vertical arrows)—both sequentially and in parallel—as demonstrated in Section 5. Moreover, the "transpose" (reflection about a diagonal) of a commutative diagram—and thus an arrow morphism—is another one of the same kind. As we will see in Section 7, transposition enables composing *vertically* (i.e., by vertical abutment with matching horizontal arrows)—both sequentially and in parallel. All four forms of composition will play important roles as we progress.

4 Carrying in

So far we have been working with unary number representations (unbounded ℕ and bounded 𝔽 n), affording simple specification and proofs but costing efficiency. Positional number systems represent numbers not as a cascade of increments but as a sequence of digits, interpreted with exponentially progressing weights. This interpretation allows for number representations and arithmetic implementations that are exponentially more efficient than (i.e., logarithmically as expensive as) in unary systems.

A familiar (though not immediately obvious) consequence of the representation change is the need to accommodate a "carry-in" value of either zero or one (a "bit") as a third addend:

```
ℕ³ : Set
ℕ³ = ℕ × ℕ²

𝔽³ : ℕ³ → Set
𝔽³ (p , m , n) = 𝔽 p × 𝔽² (m , n)

add𝔽 : ∀ {(m , n) : ℕ²} → 𝔽³ (2 , m , n) → 𝔽 (m + n)
add𝔽 (cᵢ , a , b) = cᵢ + a + b
```

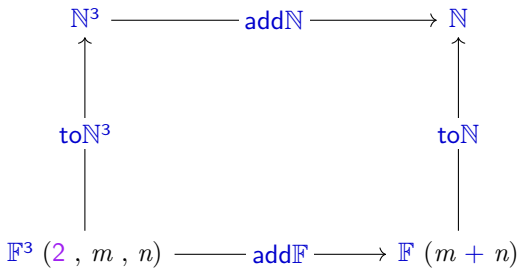
Since + is *left*-associative and c_i : 𝔽 2, the sum c_i + a is well-typed and has type 𝔽 (suc m) (since 2 ≡ suc (suc zero)), and thus c_i + a + b is also well-typed.

Since we are interpreting 𝔽 via ℕ, we can ask what does add𝔽 *mean*, i.e., to what operation on ℕ is add𝔽 precisely analogous? Intuitively, you might guess that ternary addition on 𝔽 means ternary addition on ℕ:

```
addℕ : ℕ³ → ℕ
addℕ (c , a , b) = c + a + b
```

How can we determine whether this guess is correct? More importantly, how can we proceed with justifiable confidence to progressively complex puzzles? No matter how clever we are, as our ambitions spur us on to progressively complex topics and techniques, our intuitions and informal reasoning will eventually fail us (and often without our noticing). In such cases, either we fail to come to confident conclusions, or—worse—we come to confident but incorrect conclusions.

To prepare ourselves for future challenges, let's take care in this next puzzle of ternary addition. Our conjecture, expressed diagrammatically:



where

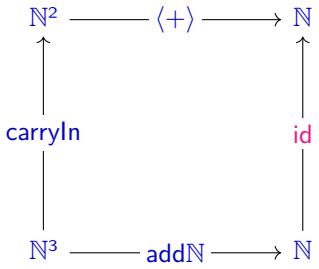
```
toℕ³ : ∀ {p,m,n} → ℙ³ p,m,n → ℕ³
toℕ³ = toℕ ⊗ toℕ²
```

Indeed, we can prove this conjecture and package up all of the pieces as before:

```
toℕ-addℙ : ∀ {m,n} → toℕ ∘ addℙ {m,n} ≗ addℕ ∘ toℕ³
toℕ-addℙ (cᵢ , a , b) rewrite toℕ-∔ (cᵢ ∔ a , b) | toℕ-∔ (cᵢ , a) = refl

addℙ⇉₀ : ∀ {(m , n) : ℕ²} → toℕ³ {2 , m , n} ⇉ toℕ {m + n}
addℙ⇉₀ = arr addℙ addℕ toℕ-addℙ
```

In addition to relating addℕ to the lower-level addℙ, we can also relate it to the original, simpler specification of two-argument addition:[7]

```
carryIn : ℕ³ → ℕ²
carryIn (cᵢ , a , b) = cᵢ + a , b

addℕ-as-⟨+⟩ : addℕ ≗ ⟨+⟩ ∘ carryIn
addℕ-as-⟨+⟩ _ = refl

addℕ⇉ : carryIn ⇉ id
addℕ⇉ = arr addℕ ⟨+⟩ addℕ-as-⟨+⟩
```

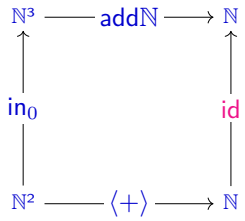


5 Horizontal composition

The morphism addℙ⇉₀ is defined above directly as the constructor arr applied to implementation addℙ, specification addℕ, and commutativity proof toℕ-addℙ. Examining these three parts reveals an intriguing pattern: each is constructed using essentially the same recipe:

[7] There's another relationship between addℕ and ⟨+⟩ that may be more important and more generalizable:

```
in₀ : ℕ² → ℕ³
in₀ (a , b) = (0 , a , b)

⟨+⟩-as-addℕ : ⟨+⟩ ≗ addℕ ∘ in₀
⟨+⟩-as-addℕ _ = refl
```



In this case, however, addℕ is not so much a *specification* for ⟨+⟩ as a *generalization* of ⟨+⟩. This generalization is exactly the sort often needed for proofs by induction, which is indeed the motivation for introducing carry-in.

- Left-associate $(c_i\ ,\ (a\ ,\ b))$ to $((c_i\ ,\ a)\ ,\ b)$.
- Add the first pair, yielding $(c_i + a\ ,\ b)$.
- Add the result, yielding $c_i + a + b$.

In particular,

```
refactor-addℕ : addℕ ≗ ⟨+⟩ ∘ first ⟨+⟩ ∘ assocˡ
refactor-addℕ _ = refl

refactor-addℙ : ∀ {m,n} → addℙ {m,n} ≗ ⟨∔⟩ ∘ first ⟨∔⟩ ∘ assocˡ
refactor-addℙ _ = refl
```

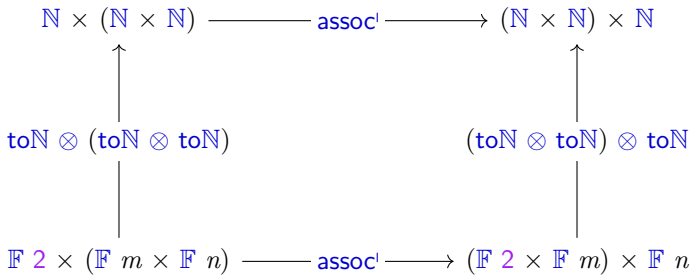
where first $f = f \otimes \mathsf{id}$.

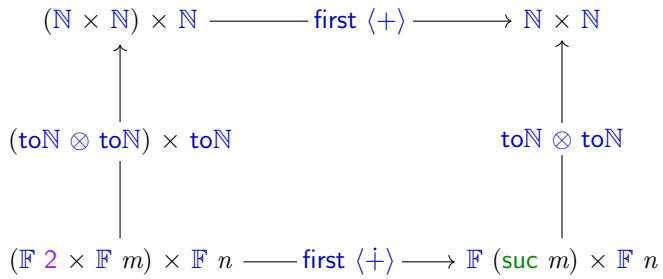
Using categorical operations and $\dot{+}\rightrightarrows$ (defined in Section 3), we can thus apply this recipe *once* (rather than three times), constructing addℙ⇉ entirely at the level of arrow category morphisms:

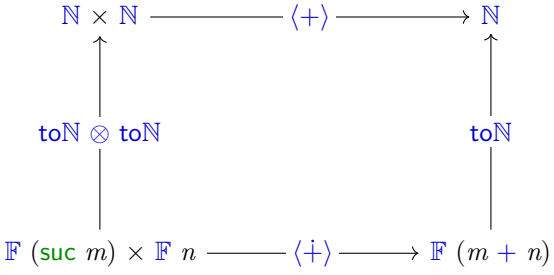
```
addℙ⇉ : ∀ {(m , n) : ℕ²} → toℕ³ {2 , m , n} ⇉ toℕ {m + n}
addℙ⇉ = ∔⇉ ∘ first ∔⇉ ∘ assocˡ
```

We've used the Category and Cartesian instances for comma categories (including their arrow category specialization) to compose our implementation-specification-proof packages, both in sequence and in parallel. (There's only a hint of the parallel here so far in assocˡ and $\otimes$, but eventually there will be much more.) Those two instances encapsulate the knowledge of how to perform these two foundational kinds of compositions and a few other useful operations as well.

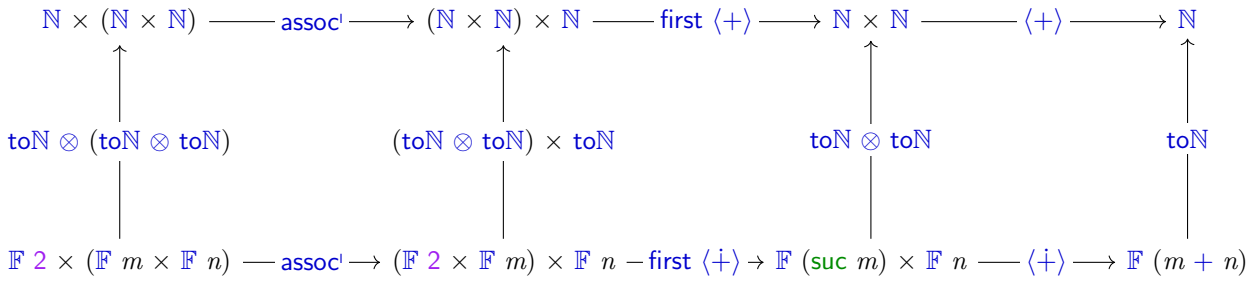
Reading this composition chain from right to left, we have the following three stages:



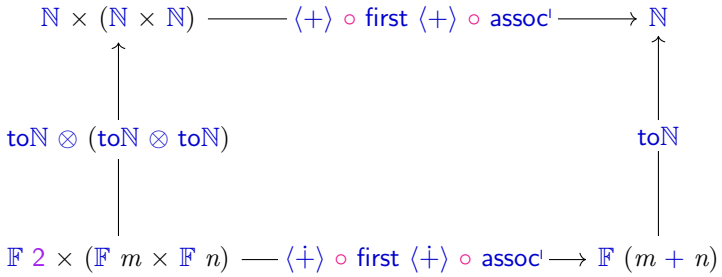




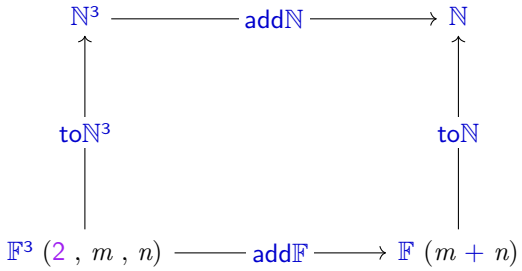
Next, *horizontally* merge these three diagrams into one:



Finally, drop the intermediate vertical arrows and compose the horizontal arrow chains:



Equivalently,

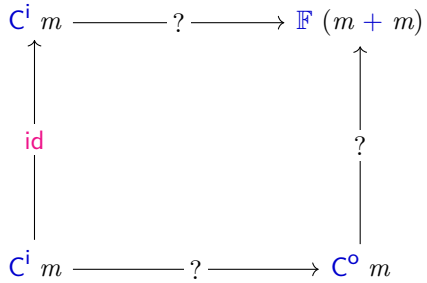


as in Section 4.

In addition to the specifications and implementations (top and bottom arrows), horizontal diagram composition also composes correctness proofs—all dependably and behind the scenes.

6 Carrying out

If we specialize addition to $m \equiv n$, then the result type $\mathbb{F}\ (m + m)$ is isomorphic to $\mathbb{F}\ m \times \mathbb{F}\ 2$, which we can think of as a single "digit" (in base m) and a carry-out bit. We can thus satisfy the following diagram:

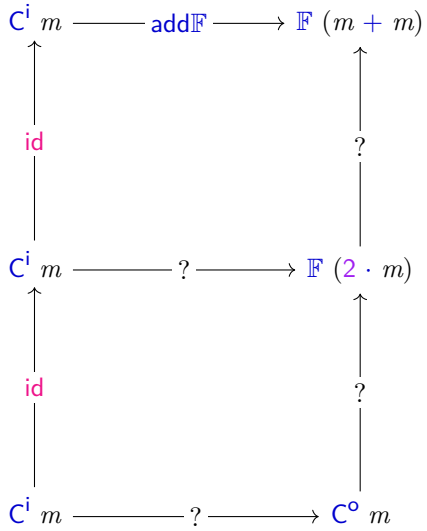


where

```
Cⁱ : ℕ → Set
Cⁱ m = 𝔽³ (2 , m , m)

Cᵒ : ℕ → Set
Cᵒ m = 𝔽² (m , 2)
```

The lower arrow will be addition on $\mathbb{F}$ with carry-in and carry-out.

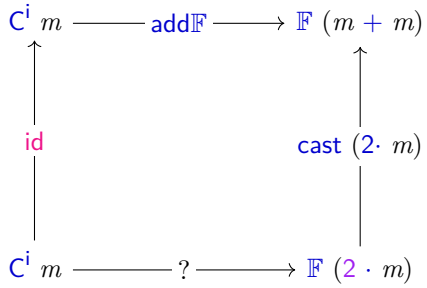
Let's tackle this puzzle in two stages:



The upper property relies on a simple numeric equivalence:

```
2· : ∀ m → 2 · m ≡ m + m
2· m rewrite +-identityʳ m = refl
```

We can thus "cast" $\mathbb{F}$ values:[8]



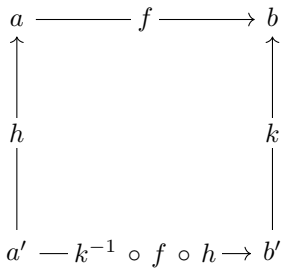
This sort of equation is easy to solve because casting is always invertible. More generally, if the right

[8]This safe cast operation is defined as follows:

```
cast : ∀ {m n} → .(_ : m ≡ n) → 𝔽 m → 𝔽 n
cast {suc m} {suc n} eq zero    = zero
cast {suc m} {suc n} eq (suc k) = suc (cast (cong pred eq) k)
```

Safety is guaranteed by the first argument, which is an equality proof.

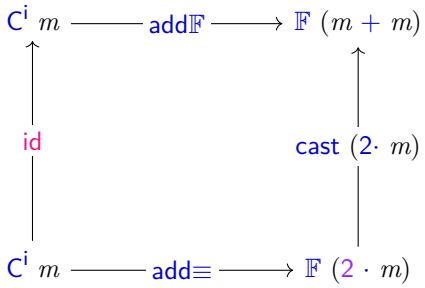
arrow in a commutative diagram is invertible, we can fill in the bottom arrow as follows:



In code:

```
inverse⇉ : ∀ {a b a′ b′ : Set} (f : a → b){h : a′ → a}{k : b′ → b}{k⁻¹ : b → b′}
         → k ∘ k⁻¹ ≗ id → h ⇉ k
inverse⇉ f {h}{k}{k⁻¹} k∘k⁻¹ = arr (k⁻¹ ∘ f ∘ h) f (k∘k⁻¹ ∘ f ∘ h)
```

Filling in the blanks,



where

```
add≡ : ∀ {m} → ℙ³ (2 , m , m) → ℙ (2 · m)
add≡ {m} = cast (sym (2· m)) ∘ addℙ {m , m}
```

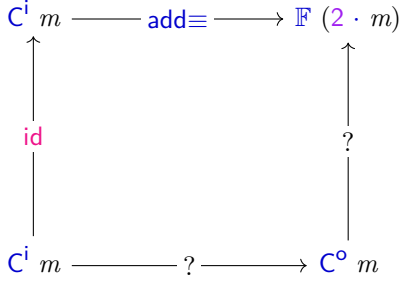
We needn't define add≡ explicitly, however, since it is constructed (and proved correct) by inverse⇉ and can be extracted from the result:

```
add≡⇉ : ∀ {m} → id {a = Cⁱ m} ⇉ cast (2· m)
add≡⇉ {m} = inverse⇉ addℙ (cast-cast-sym (2· m))
```

where

```
cast-cast-sym : ∀ {m n} → (m≡n : m ≡ n) → cast m≡n ∘ cast (sym m≡n) ≗ id
cast-cast-sym refl zero = refl
cast-cast-sym refl (suc i) rewrite cast-cast-sym refl i = refl
```

Let's next work out the lower property, now that we know *its* upper morphism:



Again, we will first identify the codomain mapping (right arrow) and then the implementation (bottom arrow).

If we think of our m-bounded numbers as *digits* in base/radix m, then the result of the unknown right arrow is in base $2 \cdot m$. For any n and m, however, $\mathbb{F}\ (n \cdot m)$ is isomorphic to $\mathbb{F}\ m \times \mathbb{F}\ n$ via two conversion functions defined in Data.Fin.Base:

```
remQuot : ∀ {n} m → 𝔽 (n · m) → 𝔽 n × 𝔽 m
combine : ∀ {n m} → 𝔽 n → 𝔽 m → 𝔽 (n · m)
```

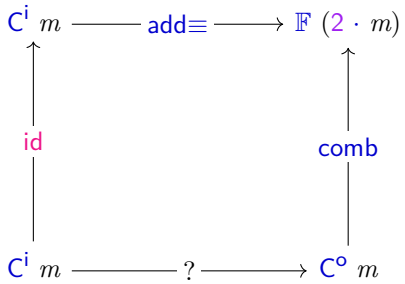
It will be a bit more convenient to have the carry-out bit on the right (as in the quotRem function—also in Data.Fin.Base) and to uncurry combine:

```
qr : ∀ {(m , n) : ℕ²} → 𝔽 (n · m) → 𝔽² (m , n)
qr {m , n} = quotRem m

comb : ∀ {(m , n) : ℕ²} → 𝔽² (m , n) → 𝔽 (n · m)
comb = uncurry combine ∘ swap
```

These two operations form an isomorphism:

```
qr∘comb : ∀ {m,n} → qr {m,n} ∘ comb ≗ id

comb∘qr : ∀ {m,n} → comb ∘ qr {m,n} ≗ id
```

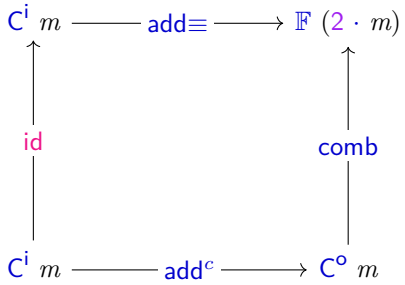
We can use comb for our missing right arrow:



Again, invertibility of the right arrow makes this equation easy to solve:

```
add^c⇉ : ∀ {m} → id {a = Cⁱ m} ⇉ comb {m , 2}
add^c⇉ {m} = inverse⇉ add≡ {k⁻¹ = qr} (comb∘qr {m , 2})
```

Diagrammatically,



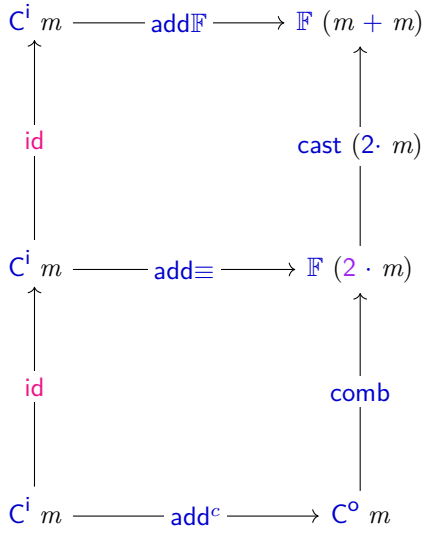
where

```
C : ℕ → Set
C m = Cⁱ m → Cᵒ m

add^c : ∀ {m} → C m
add^c = qr ∘ add≡
```

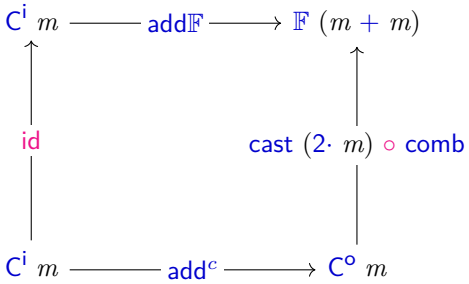
As before, rather than defining the implementation (add^c) explicitly, we can extract it from the commutativity morphism ($\mathsf{add}^c{\rightrightarrows}$).

7 Vertical composition

We have now finished solving our two-stage puzzle:



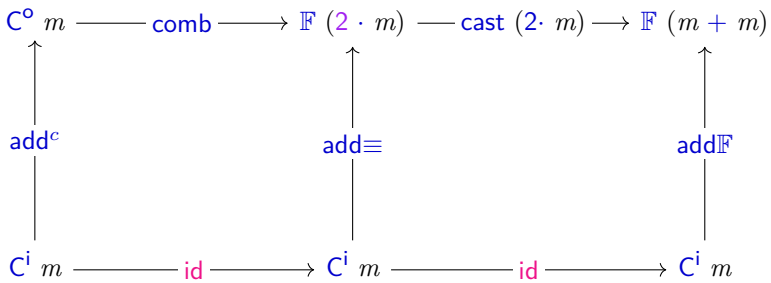
Collapsing to one stage,



Dually to horizontal composition (as used in Section 5), composing diagrams *vertically* requires that the lower arrow ("implementation") of the upper diagram matches the upper arrow ("specification") of the lower diagram. Vertical arrows on the left (domain mappings) compose, as do vertical arrows on the right (codomain mappings). In code,

```
_ : ∀ {m} → id ⇉ cast (2· m) ∘ comb {m , 2}
_ = add≡⇉ ⊙ addᶜ⇉
```

Vertical composition is defined as $G \odot F = (G^{T} \circ F^{T})^{T}$, where $\_^{T}$ is diagram transposition (flipping about a diagonal), and $\_\circ\_$ is the same horizontal sequential composition we used above. We might visualize an intermediate stage in this process as follows:



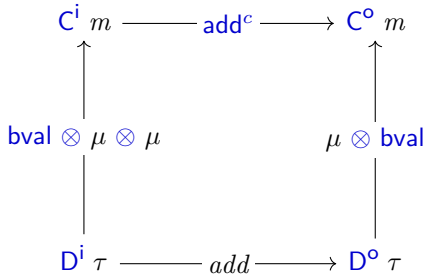
We will continue to use vertical composition to reach downward in stages from the higher level specification to progressively lower level representations and operations, while composing proofs for each stage to a proof for the vertical composition.

8 Alternative number representations

So far we have used unary number representations, leading to expensive addition. Now let's turn our attention to much more compact representations, opening the possibility of efficient addition. Rather than assuming bit vectors, we will want to use a variety of representations, with corresponding addition algorithms. The shape of addition-with-carry changes, replacing $\mathbb{F}\ m$ with an arbitrary representation type τ and replacing $\mathbb{F}\ 2$ (the type of carries) with booleans (here written "$\mathbb{B}$"):

```
Dⁱ : Set → Set
Dⁱ τ = 𝔹 × τ × τ

Dᵒ : Set → Set
Dᵒ τ = τ × 𝔹

D : Set → Set
D τ = Dⁱ τ → Dᵒ τ
```

Each representation type τ will have a corresponding semantic function $\mu : \tau \to \mathbb{F}\ m$ for some $m : \mathbb{N}$. A correct addition implementation *add*: D τ is one that satisfies the following condition:



In code,

```
is-add : ∀ {τ : Set}{m} (μ : τ → 𝔽 m) (add : D τ) → Set
is-add μ add = (μ ⊗ bval) ∘ add ≗ addᶜ ∘ (bval ⊗ μ ⊗ μ)
```

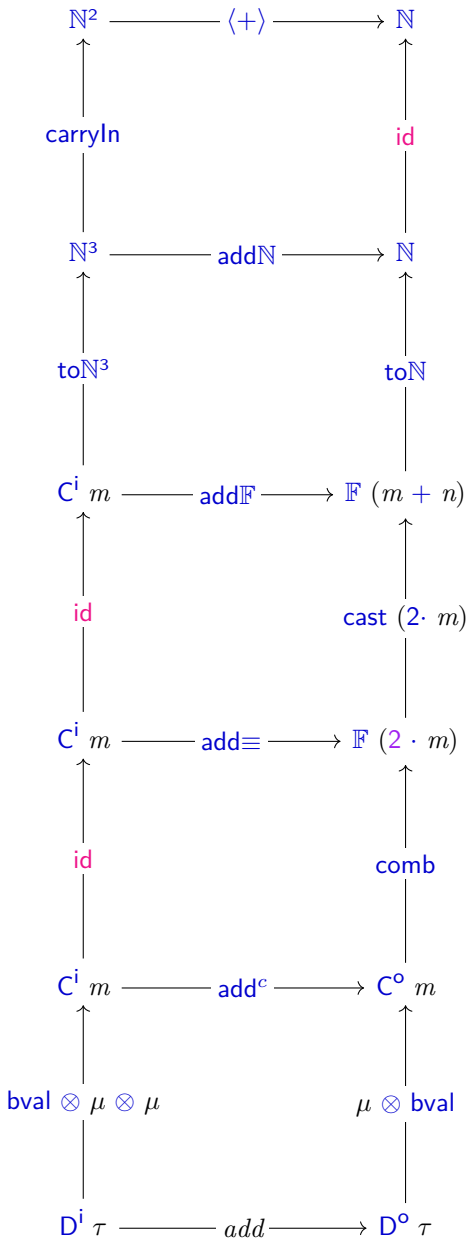
Let's package *add* functions that satisfy this condition for convenient construction and composition::

```
infix 1 _⊣_
record Adder {τ : Set}{m} (μ : τ → 𝔽 m) : Set where
  constructor _⊣_
  field
    add : D τ
    is : is-add μ add
```

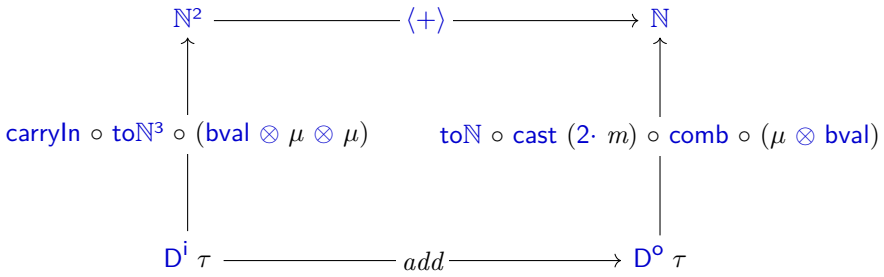
Thus, in a pattern or expression "*add* ⊣ *is*", *is* proves that *add* correctly implements addᶜ. We can easily extract a composable commutativity condition:

```
adder⇉ : ∀ {τ : Set} {m} {μ : τ → 𝔽 m} → Adder μ → bval ⊗ μ ⊗ μ ⇉ μ ⊗ bval
adder⇉ (add ⊣ is) = arr add addᶜ is
```

We have already proved that add^c correctly implements $\mathsf{add}\mathbb{N}$ (and thus $\langle + \rangle$) through a sequence of prior stages, so we can add an additional lower commutative square:



Vertical composition combines these five stages into a single commutative square:

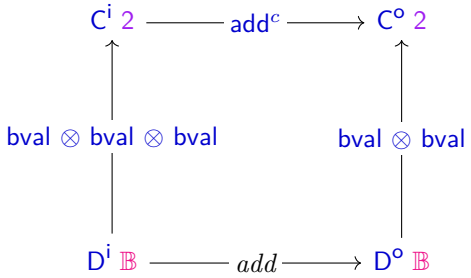


In code:

```
addℕᶜ⇉ : ∀ {τ : Set} {m} {μ : τ → 𝔽 m} (add : D τ) (is : is-add μ add)
       → carryIn ∘ toℕ³ ∘ (bval ⊗ μ ⊗ μ) ⇉ toℕ ∘ cast (2· m) ∘ comb ∘ (μ ⊗ bval)
addℕᶜ⇉ add is = addℕ⇉ ⊙ add𝔽⇉ ⊙ add≡⇉ ⊙ addᶜ⇉ ⊙ adder⇉ (add ⊣ is)
```

8.1 Base two addition

As a first example, let's use booleans to represent $\mathbb{F}\ 2$. In terms of Adder parameters, $\tau = \mathbb{B}$, $m = 2$ and μ = bval. The commutativity condition specializes to the following:



The implementation of this "full" adder (a hardware classic) is built from two "half adders", with its correctness checked exhaustively:[9]

```
addB : D 𝔹
addB (cᵢ , a , b) = let p , d = ½+ (a , b)
                        q , e = ½+ (cᵢ , p)
                    in
                        q , e ∨ d
  where
    ½+ : 𝔹 × 𝔹 → 𝔹 × 𝔹
    ½+ (cᵢ , a) = cᵢ ⊕ a , cᵢ ∧ a

adderB : Adder bval
adderB = addB ⊣ λ { ( false , false , false ) → refl
                  ; ( false , false , true  ) → refl
                  ; ( false , true  , false ) → refl
                  ; ( false , true  , true  ) → refl
                  ; ( true  , false , false ) → refl
                  ; ( true  , false , true  ) → refl
                  ; ( true  , true  , false ) → refl
                  ; ( true  , true  , true  ) → refl
                  }
```

Figure 1 displays this full adder as a circuit graph.

8.2 Combining adders

If we have an adder for representation τ_m and one for representation τ_n with capacities (upper bounds) m and n respectively, we can combine them into a single adder for the representation $\tau_m \times \tau_n$ with capacity $n \cdot m$. For instance, a p-bit adder has capacity 2^p, and a q-bit adder has capacity 2^q, so their $(p+q)$-bit combination has capacity $2^q \cdot 2^p \equiv 2^{p+q}$.

In addition to representation and capacity, adders have their interpretation (μ), implementation, and correctness proof. Let's consider one at a time, starting with interpretation:

[9] Is there a more compositional and insightful proof?

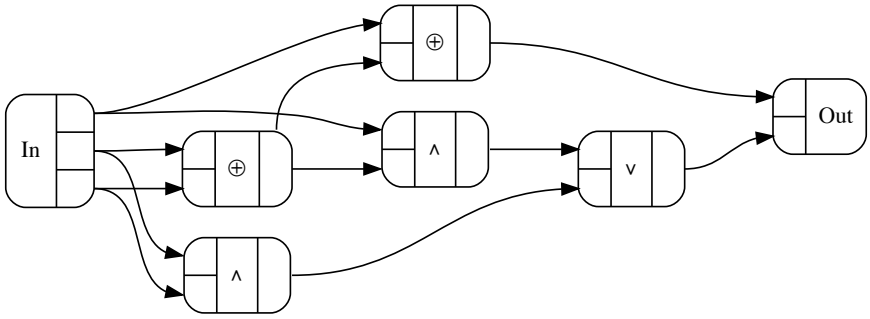


Figure 1: One-bit ("full") adder

```
infixr 5 _•_
_•_ : ∀ {τm τn} {(m , n) : ℕ²} (μm : τm → 𝔽 m) (μn : τn → 𝔽 n)
      → (τm × τn → 𝔽 (n · m))
μm • μn = comb ∘ (μm ⊗ μn)
```

Equivalently, $(\mu_m \bullet \mu_n)\ (x_m\ ,\ x_n) = \mathsf{comb}\ (\mu_m\ x_m\ ,\ \mu_n\ x_n)$. As an example of this interpretation, the numeral "34" in base 10 denotes $4 + 3 \cdot 10$.

The key to combining adder implementations is the combination of carry-in and carry-out designed into our C type. The carry-out of one adder becomes the carry-in of the other:

```
infixr 5 _•̂_
_•̂_ : ∀ {τm τn} → D τm → D τn → D (τm × τn)
(addm •̂ addn) (ci , (xm , xn) , (ym , yn)) =
  let zm , c′ = addm (ci , xm , ym)
      zn , co = addn (c′ , xn , yn)
  in (zm , zn) , co
```

If add_m and add_n are correct adders for interpretations μ_m and μ_n, respectively, then $\mathsf{add}_m \mathbin{\hat{\bullet}} \mathsf{add}_n$ is a correct adder for interpretation $\mu_m \bullet \mu_n$ (proved in the source code):

```
infixr 5 _•̃_
_•̃_ : ∀ {τm τn} {(m , n) : ℕ²} {μm : τm → 𝔽 m}{μn : τn → 𝔽 n} {addm : D τm}{addn : D τn}
  → is-add μm addm → is-add μn addn → is-add (μm • μn) (addm •̂ addn)
```

Now we can package up the various pieces into a combining operation on our Adder type (which encapsulates adder implementations with their correctness proofs):

```
infixr 5 _•̆_
_•̆_ : ∀ {τm τn} {m n} {μm : τm → 𝔽 m}{μn : τn → 𝔽 n}
  → Adder μm → Adder μn → Adder (μm • μn)
(addm ⊣ ism) •̆ (addn ⊣ isn) = (addm •̂ addn) ⊣ (ism •̃ isn)
```

As an example, we can easily construct a verified two-bit adder, defined below and displayed in Figure 2.

```
adderB₂ : Adder (bval • bval)
adderB₂ = adderB •̆ adderB
```

8.3 Base one addition

We can also define a trivial, zero-bit (base one) adder (useful as an identity to $\mathbin{\breve{\bullet}}$, as we will soon see):

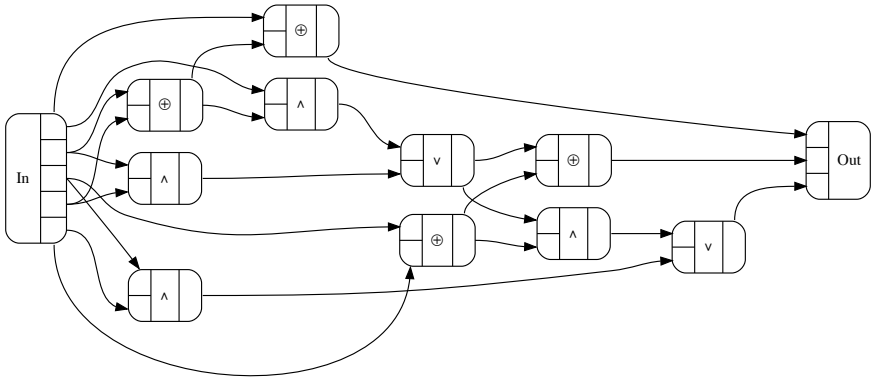


Figure 2: Two-bit adder

```
tval : ⊤ → 𝔽 1
tval tt = zero

adder₁ : Adder tval
adder₁ = add ⊣ is
  where
    add : D ⊤
    add (cᵢ , tt , tt) = tt , cᵢ

    is : is-add tval add
    is (false , tt , tt) = refl
    is (true  , tt , tt) = refl
```

The representation type ⊤ has a single inhabitant tt, here representing zero. The two addend digits and the sum digit can only be tt, so the carry-in bit passes directly through to carry-out. Correctness is verified by checking the two possible carry-in bit values.

8.4 Adding vectors of digits

Fixed-radix number system (e.g., binary and decimal) represent numbers as *vectors* of digits. Start with a type of length-indexed vectors:

```
V : Set → ℕ → Set
V a zero    = ⊤
V a (suc n) = a × V a n
```

Representation capacity grows exponentially with the number of digits. Specifically, if a representation type τ can represent any number below m, then V τ n can represent any number below m^n, thanks to tval and repeated use of •:[10]

```
infixl 10 _↑_
_↑_ : ∀ {τ}{m} (μ : τ → 𝔽 m) (n : ℕ) → (V τ n → 𝔽 (m ^ n))
```

[10]Numeric exponentiation as used here is defined as follows:

```
_^_ : ℕ → ℕ → ℕ
m ^ zero  = 1
m ^ suc n = (m ^ n) · m
```

This definition is equivalent to the version from the Agda standard library but reverses the order of multiplication in the suc clause. It might be simpler to reverse multiplication elsewhere instead.

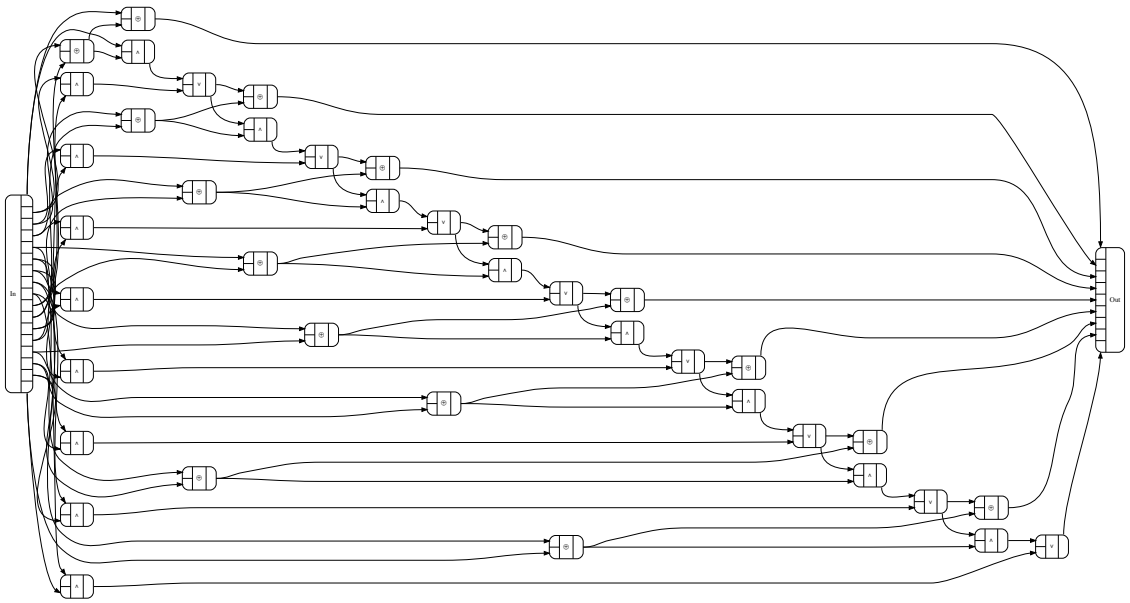


Figure 3: Eight-bit ripple adder: adderB ⇑ 8

```
μ ↑ zero  = tval
μ ↑ suc n = μ • (μ ↑ n)
```

We can correctly add digit vectors via adder_1 and repeated use of $\check{\bullet}$:

```
infixl 10 _⇑_
_⇑_ : ∀ {τ}{m}{μ : τ → 𝔽 m} (a : Adder μ) (n : ℕ) → Adder (μ ↑ n)
a ⇑ zero  = adder₁
a ⇑ suc n = a ∙̌ (a ⇑ n)
```

The commutativity condition verified by adderB ⇑ n:

$$\begin{array}{ccc}
\mathsf{C}^{\mathsf{i}}\ (2 \mathbin{\widehat{}} n) & \xrightarrow{\quad \mathsf{add}^{c} \quad} & \mathsf{C}^{\mathsf{o}}\ (2 \mathbin{\widehat{}} n) \\
\uparrow & & \uparrow \\
\mathsf{bval} \otimes \mu \uparrow n \otimes \mu \uparrow n & & \mu \uparrow n \otimes \mathsf{bval} \\
\downarrow & & \downarrow \\
\mathsf{D}^{\mathsf{i}}\ (\mathsf{V}\ \tau\ n) & \xrightarrow{\quad \mathsf{addB}^{v}\ n \quad} & \mathsf{D}^{\mathsf{o}}\ (\mathsf{V}\ \tau\ n)
\end{array}$$

where

```
addB^v : ∀ n → D (V 𝔹 n)
addB^v n = Adder.add (adderB ⇑ n)
```

As an example, Figure 3 shows a verified eight-bit adder. We can also add vectors of vectors of bits, as illustrated in Figure 4

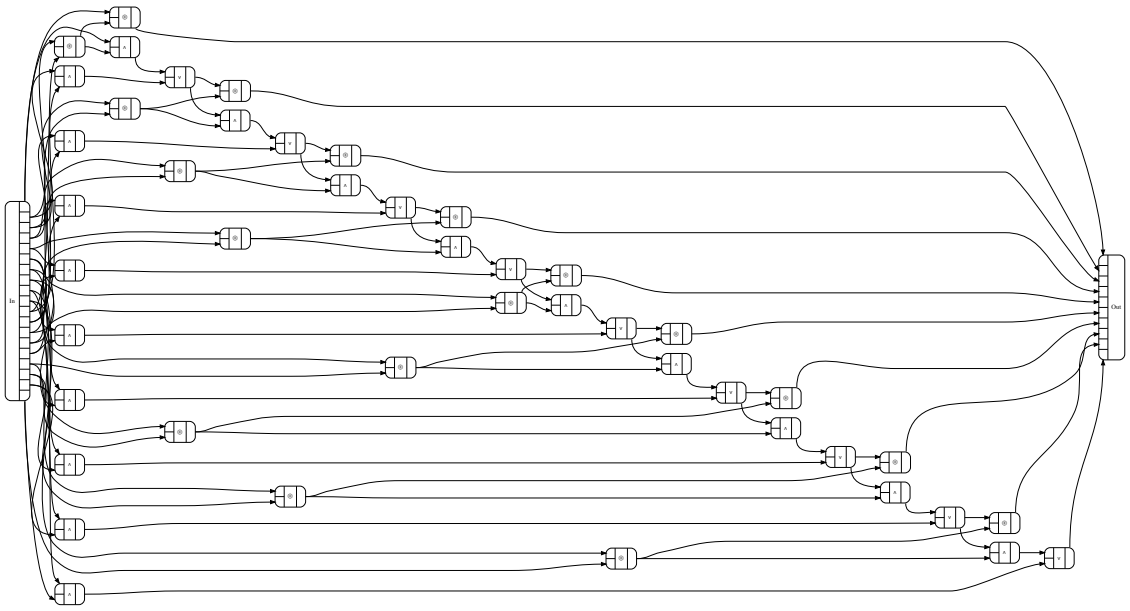


Figure 4: Nine-bit ripple adder: adderB ⇑ 3 ⇑ 3

9 Speculation

In ripple-carry addition, the carry chain causes a delay linear in the number of bits being added. Instead of waiting for a carry-in to be ready, however, we can instead perform two similar computations speculatively, assuming a carry-in of zero for one version and of one for the other. Later, when the carry-in is known, use it to select between the two results.

```
speculate : ∀ {a c : Set} → (𝔹 × a → c) → (𝔹 × a → c)
speculate f (b , a) = if b then f (true , a) else f (false , a)

speculate≗ : ∀ {a c : Set} → (f : 𝔹 × a → c) → speculate f ≗ f
speculate≗ f (false , a) = refl
speculate≗ f (true  , a) = refl
```

Next, wrap up speculate and its correctness as a verified adder:[11]

```
speculate-is-add : ∀ {τ : Set}{m} {μ : τ → 𝔽 m} {add : D τ}
                   → is-add μ add → is-add μ (speculate add)
speculate-is-add {μ = μ}{add} is = ∘≈ʳ {h = μ ⊗ bval} (speculate≗ add) ; is

spec : ∀ {τ}{m}{μ : τ → 𝔽 m} → Adder μ → Adder μ
spec (add ⊣ is) = speculate add ⊣ speculate-is-add is
```

For convenience, combine speculation and vector-of-digits addition:

```
infixl 10 _⇑̇_
_⇑̇_ : ∀ {τ}{m}{μ : τ → 𝔽 m} (a : Adder μ) (n : ℕ) → Adder (μ ↑ n)
a ⇑̇ n = spec a ⇑ n
```

While we can apply this "carry-select" technique to any grouping of consecutive bits, a particularly effective choice is to choose groups whose size is the square root of the total number of bits per summand. For instance, Figure 5 shows a nine-bit carry-select adder circuit with speculation every three bits, semantically

[11] The ∘≈ʳ function extends a proof of $f \approx g$ to a proof of h ∘ $f \approx$ h ∘ g, where ≈ is morphism equivalence. The (unicode) ";" operator is transitivity of morphism equivalence.

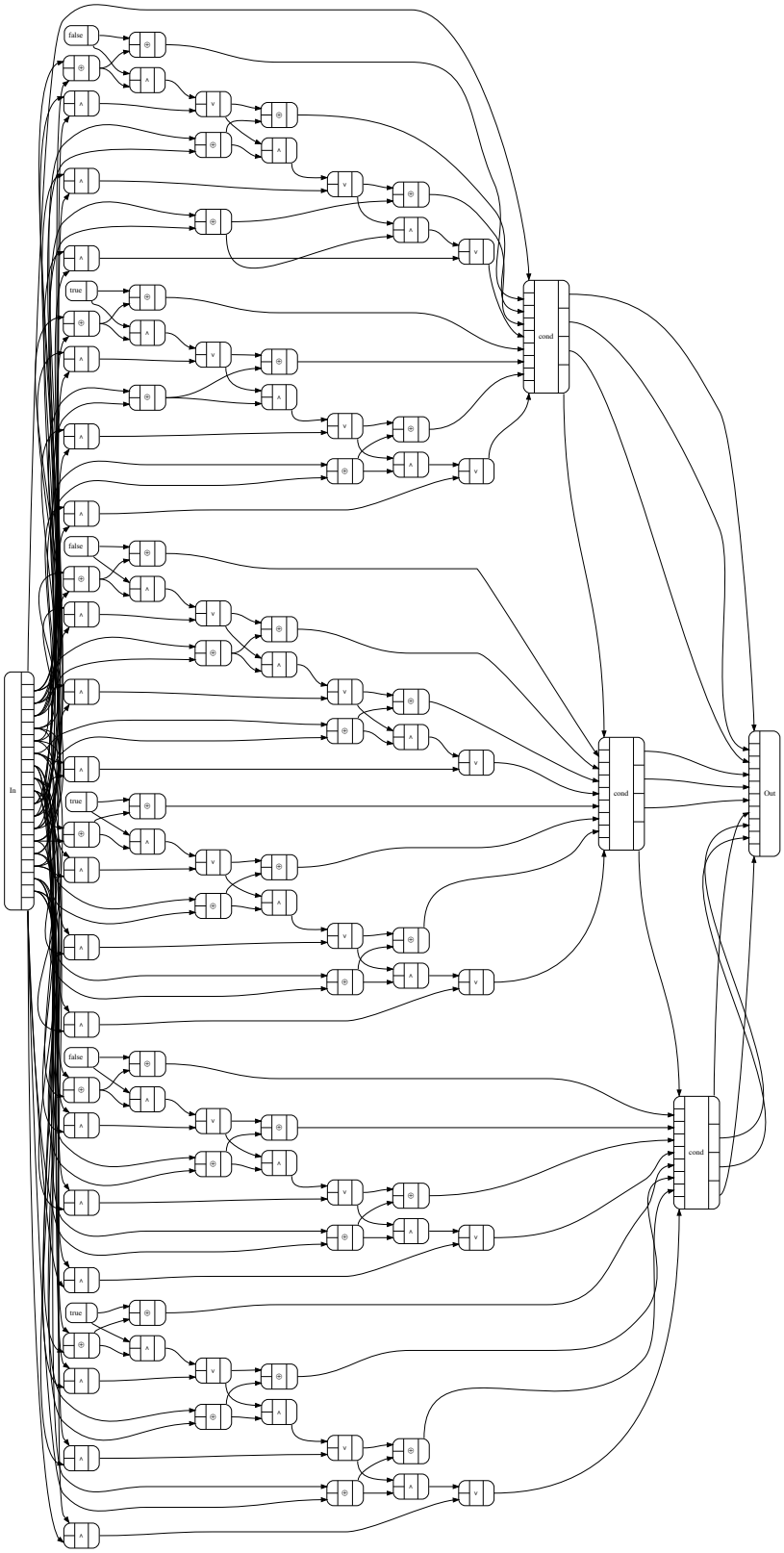


Figure 5: Nine-bit *carry-select* adder: adderB ⇑ 3 $\dot{\Uparrow}$ 3

equivalent to the nine-bit pure ripple adder in Figure 4 with lower latency and higher gate count. More generally,

```
carrySelect : ∀ m n → Adder (bval ↑ m ↑ n)
carrySelect m n = adderB ⇑ m ⇂⇑ n
```

This definition type-checks (quickly), formally verifying the correctness of every instance of the infinite family of intricate carry-select adder circuits.

We can easily go beyond two levels of rippling with one level of speculation. Figure 6 shows an eight-bit carry-select adder circuit with two levels of speculation, generalized as follows:

```
carrySelect₃ : ∀ m n o → Adder (bval ↑ m ↑ n ↑ o)
carrySelect₃ m n o = carrySelect m n ⇂⇑ o
```

Again, all adders in this family are fully verified by type-checking this succinct definition.

10 Sequential computation

The computations above—especially as visualized in Figure 3 and similar graphs—are suggestive of combinational circuits, in which larger logical computations are realized by larger circuit implementations. This implementation style is only one choice of many and offers maximum parallelism. As the pure ripple adder in Figure 3 illustrates, sometimes a purely combinational implementation can be quite wasteful, requiring considerable hardware resources without enabling much useful parallel work due to data dependencies. Carry-select addition decreases latency by increasing gate count and hence area and power.

Instead decreasing latency by adding gates, we can instead design a circuit with only a few gates and reuse them repeatedly during a single logical computation. Ripple adders work well with this second strategy, because they are formed by chaining multiple copies of the full adder shown in Figure 1.

Sequential circuits are state machines that transform input sequences to output sequences of the same length. Each output value is computed from the corresponding input value together with a state, which also updates. Within this pattern, each state machine is characterized by state type and transition function. If the state type is σ and the input and output types are a and b, respectively, then the transition function has type $\sigma \times a \to b \times \sigma$. The semantics of a state machine is then a function from initial state and input sequence of any length to an output sequence of the same length and final state, defined as follows:

```
loop : ∀ {a b σ} → (σ × a → b × σ) → ∀ n → σ × V a n → V b n × σ
loop _ zero (s , tt) = tt , s
loop f (suc n) (s , x , xs) = let y , s′ = f (s , x) in
  first (y ,_) (loop f n (s′ , xs))
```

To implement a multi-bit adder as a small sequential circuit, we must find a way to re-express its meaning in terms of *loop*. Well, there's a technical obstacle, namely that the shape of adder inputs do not quite match the transition function's input. Where a transition function consumes a single vector of values, a vector-of-digits adder consumes a pair of vectors. Fortunately, there is a handy isomorphism between pairs of vectors and vectors of pairs.

Let's give a name to vector-of-bits addition:

```
addBᵛ : ∀ n → D (V 𝔹 n)
addBᵛ n = Adder.add (adderB ⇑ n)
```

We can represent adder input in zipped form:

```
E⇉ : ∀ n → second (unzipᵛ n) ⇉ id
E⇉ n = arr (addBᶻ n) (addBᵛ n) identityʳ
```

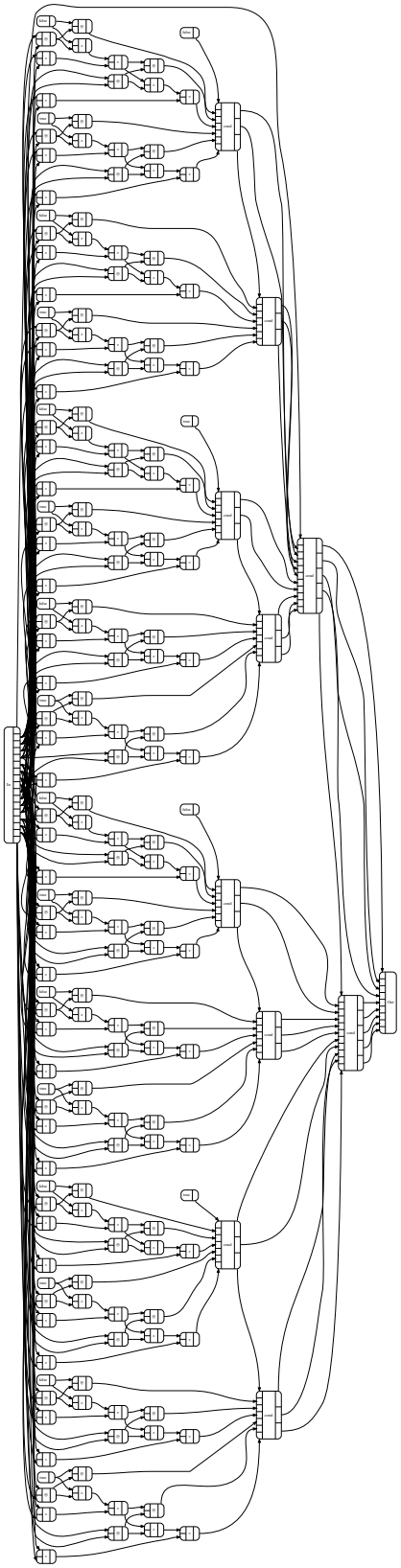
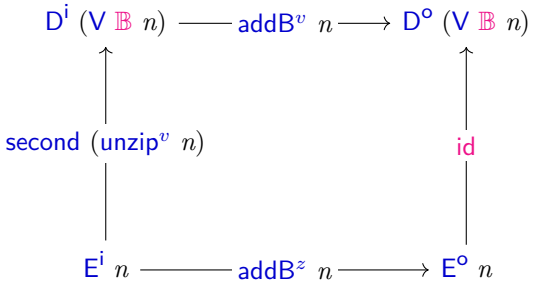


Figure 6: Eight-bit *carry-select* adder: adderB $\Uparrow 2 \dot{\Uparrow} 2 \dot{\Uparrow} 2$

where

```
Eⁱ : ℕ → Set
Eⁱ n = 𝔹 × V (𝔹 × 𝔹) n

Eᵒ : ℕ → Set
Eᵒ n = V 𝔹 n × 𝔹

E : ℕ → Set
E n = Eⁱ n → Eᵒ n


addBᶻ : ∀ n → E n
addBᶻ n = addBᵛ n ∘ second (unzipᵛ n)
```

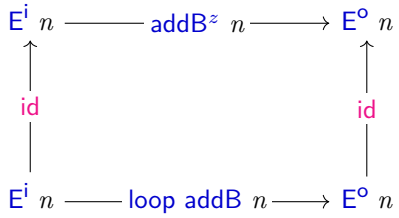
Diagramatically,



We can express addBᶻ via loop:

```
loop-add : ∀ n → loop addB n ≈ addBᶻ n
loop-add zero _ = refl
loop-add (suc n) (s , (x , y) , xys) = let z , s′ = addB (s , x , y) in
  cong (first (z ,_)) (loop-add n (s′ , xys))
```

This equivalence gives rise to yet another level in our tower of commutative diagrams:



As code:

```
loop-add⇉ : ∀ n → id ⇉ id
loop-add⇉ n = arr (loop addB n) (addBᶻ n) (loop-add n)
```