Free Semiarrows

Alexandre Garcia de Oliveira

Fatec-Rubens Lara · Obsidian Systems

MSFP 2026

The inspectability problem

- Arithmetic circuits over a prime field $\mathbb{F}_p$ are the building blocks of the constraint systems used in proof systems.
- A value of type *p a b* is a circuit component with input wires *a* and output wires *b*; we wire components in *sequence* and in *parallel*.
- In the arrow world we would happily write the squaring circuit as

```
square_arr = arr (\x -> x * x)
```

- Perfectly valid — but the multiplication is now hidden inside a Haskell closure.
- A compiler that wants to count nonlinear constraints (R1CS), or optimise the circuit, cannot look inside `arr`.

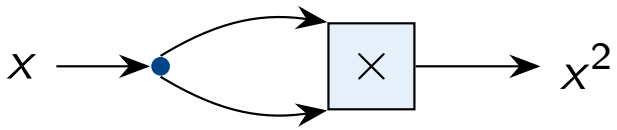
The question

Can we keep sequential + parallel composition, but remove the escape hatch, so that *every* gate and *every* wire is visible in the term?

The same circuit, with nothing hidden

In the free-semiarrow world the squaring circuit is written

```
square :: FreeSemi Gate F F
square = SDimap WCopy WId (SLift Mul)
```



- The term is a GADT value: every gate is named, every wiring operation is explicit, and there is no `arr` to hide anything.
- A cost-counting interpreter can recurse over the term and count *exactly one* nonlinear gate. A symbolic R1CS compiler can do the same.
- Removing `arr` also removes `arr id` — so we lose the identity for sequential composition. That is exactly the step from *arrows* to *semiarrows*.

Background: profunctors and two tensors

- A *profunctor* $p : \mathcal{C}^{op} \times \mathcal{C} \to \text{Set}$ (contravariant in inputs, covariant in outputs, via `dimap`). The category Prof carries two tensor products:

Two tensors, two jobs

Day convolution ($\star$), the **parallel** tensor: components side by side,

$$(P \star Q)(S, T) \cong \int^{ABCD} P(A, B) \times Q(C, D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T),$$

with unit $J(A, B) = \mathcal{C}(A, I) \times \mathcal{C}(I, B)$.

Profunctor composition ($\bullet$), the **sequential** tensor: output chained into input through a shared middle type,

$$(P \bullet Q)(A, B) = \int^{Z} P(A, Z) \times Q(Z, B),$$

with unit the hom-profunctor $\mathcal{C}(A, B)$.

- In Haskell ($\mathcal{C} = \text{Set}$, $\otimes = \times$) these read
  `(⋆) :: p a b -> p c d -> p (a,c) (b,d)` and
  `(⋙) :: p a b -> p b c -> p a c`.
- A monoid under $\star$ = *monoidal profunctor*; a monoid under $\bullet$ (+ strength) = *arrow*, whose unit is `arr id` (Rivas & Jaskelioff).

Semiarrows: monoid on one side, semigroup on the other

The slogan

A semiarrow is one profunctor p that is simultaneously a **monoid** under Day convolution ($\star$) — parallel composition *with* unit — and a **semigroup** under profunctor composition ($\bullet$) — sequential composition *without* identity — with two laws linking the tensors: interchange and unit interaction.

- The setting (thesis, Def. 4.3): a *semiarrow category* is a symmetric monoidal category $(\mathcal{C}, \otimes, I)$ together with a *semigroup category* $(\mathcal{C}, \oplus)$ — same category, second tensor, no unit — plus two natural transformations

$$\zeta : I \to I \oplus I \qquad \iota : (P \oplus Q) \otimes (R \oplus S) \to (P \otimes R) \oplus (Q \otimes S).$$

- A *semiarrow* (Def. 4.4) is $(P, e, m_\otimes, m_\oplus)$: a monoid $(P, e, m_\otimes)$ and a semigroup $(P, m_\oplus)$ on the same object, satisfying

$$m_\oplus \circ (e \oplus e) \circ \zeta = e \qquad \text{and} \qquad m_\otimes \circ (m_\oplus \otimes m_\oplus) = m_\oplus \circ (m_\otimes \oplus m_\otimes) \circ \iota.$$

- Instantiate $\mathcal{C} = \mathrm{Prof}$, $\otimes = \star$, $\oplus = \bullet$: these two laws are exactly **U1** and **I1**.

The eight laws, concretely

- A *semigroup* is a monoid that has dropped its unit. Dropping the unit of sequential composition is exactly dropping `arr id`.

Definition (semiarrow, Haskell form)

A monoidal profunctor p with $(\ggg) : p\ a\ b \to p\ b\ c \to p\ a\ c$, satisfying:

P1–P2	`dimap` respects composition on each side
M1–M3	$\star$ is a monoid: parallel units + associativity
S1	$(p \ggg q) \ggg r = p \ggg (q \ggg r)$ *sequential associativity*
I1	$(f \ggg g) \star (h \ggg k) = (f \star h) \ggg (g \star k)$ *interchange*
U1	$\mathit{mpempty} \ggg \mathit{mpempty} = \mathit{mpempty}$ *unit interaction*

- **I1** in circuit terms: independent sub-circuits can be sequenced and parallelised in either order.
- **U1** relates the sequential multiplication to the parallel unit: a pipeline of empty circuits is still empty.

Why drop the sequential identity?

- For some structures a sequential identity simply does not exist.

```
data Moore a b = Moore { output :: b, step :: a -> Moore a b }

instance Semiarrow Moore where
  m1 ⋙ m2 = Moore (output m2)
                  (\a -> step m1 a ⋙ step m2 (output m1))
```

- Sequential composition introduces a *one-step delay*: to advance, `m2` consumes `m1`'s *stored* (not current) output.
- An identity would have to output its current input — but a Moore machine's output is always a stored value from the previous step.

And for circuits?

An identity *wire* does exist — but it is already available *structurally*, as the `WId` wiring morphism. Dropping the categorical identity for (⋙) costs circuits no expressiveness; what we gain is uniformity and inspectability.

Semiarrows in Haskell, and the Wiring GADT

```
class Profunctor p => MonoPro p where
  mpempty :: p () ()
  (⋆)     :: p a b -> p c d -> p (a,c) (b,d)

class MonoPro p => Semiarrow p where
  (⋙) :: p a b -> p b c -> p a c
```

- In the free construction, `dimap` does *not* take arbitrary functions — only canonical structural maps:

```
data Wiring a b where
  WId     :: Wiring a a           WFst  :: Wiring (a,b) a
  WSwap   :: Wiring (a,b) (b,a)   WSnd  :: Wiring (a,b) b
  WAssocL :: Wiring (a,(b,c)) ((a,b),c)
  WAssocR :: Wiring ((a,b),c) (a,(b,c))
  WCopy   :: Wiring a (a,a)       WDel  :: Wiring a ()
```

- Exactly the generators of cartesian structure: identity, swap, associativity, projections, diagonal (fan-out), terminal map.
- Every `Wiring` term is fully inspectable: a compiler can pattern-match on it and treat each case symbolically.

A relational semantics for constraint checking

- In a constraint system the output is *not computed* but supplied externally as a witness; the circuit only *verifies* it.

```
data CoState r a b = CoState { witness :: b, accepts :: a -> r }
type Rel = CoState Bool

instance Semiarrow (CoState Bool) where
  CoState b p ⋙ CoState c q =
    CoState c (\a -> p a && q b)
```

- Sequential composition feeds the first stage's witness into the second stage's predicate.
- Only a *semigroup* on `Bool` (conjunction) is needed — no sequential identity anywhere.
- A primitive `mulRel z :: Rel (F,F) F` stores a witness z and checks $z \equiv x * y$: constraint checking, not evaluation.

The free semiarrow

The free semiarrow over a base profunctor p is a GADT with one constructor per semiarrow operation:

```
data FreeSemi p a b where
  SLift  :: p a b -> FreeSemi p a b
  SSeq   :: FreeSemi p a b -> FreeSemi p b c -> FreeSemi p a c
  SPar   :: FreeSemi p a b -> FreeSemi p c d
                           -> FreeSemi p (a,c) (b,d)
  SEmpty :: FreeSemi p () ()
  SDimap :: Wiring a b -> Wiring c d -> FreeSemi p b c
                                     -> FreeSemi p a d
```

- `SLift` embeds the base profunctor: the unit of the adjunction.
- `SSeq` is $(\ggg)$, `SPar` is $\star$, `SEmpty` is `mpempty`, `SDimap` is re-indexing by `Wiring` morphisms.
- The instances are trivial by construction: every operation just builds the corresponding node in the syntax tree.

Interpretation, and why it is unique

Given a semiarrow q and a natural transformation $\varphi : p \Rightarrow q$:

```
interpretFree phi (SLift p)        = phi p
interpretFree phi (SSeq s1 s2)     = ... ⋙ ...
interpretFree phi (SPar s1 s2)     = ... ⋆ ...
interpretFree _   SEmpty           = mpempty
interpretFree phi (SDimap f g s)   =
  dimap (runWiring f) (runWiring g) (interpretFree phi s)
```

Theorem (semiarrow homomorphism)

For any natural transformation $\varphi : p \Rightarrow q$ into a semiarrow q, $\mathit{interpretFree}_{\varphi}$ is the unique semiarrow homomorphism $\mathit{FreeSemi}\ p \Rightarrow q$ extending φ.

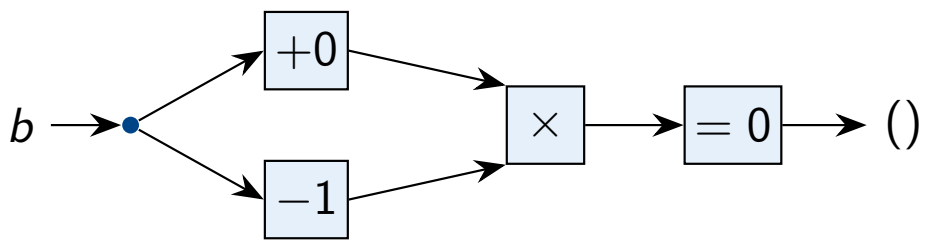
- Existence: each clause is the corresponding case of the recursion, and q's eight laws carry over.
- Uniqueness: the homomorphism equations determine h by structural induction on the GADT.

Application: an arithmetic-circuit DSL

Circuits over $\mathbb{F}_{97}$ (small, for executable examples):

```
data Gate a b where
  Add         :: Gate (F,F) F      Const    :: F -> Gate () F
  Mul         :: Gate (F,F) F      MulConst :: F -> Gate F F
  Neg         :: Gate F F          AddConst :: F -> Gate F F
  AssertZero  :: Gate F ()
```

- Type parameters record the wires: `Mul :: Gate (F,F) F` takes two field elements, produces one.
- `AssertZero` returns `()`: constraint gates fit naturally into the type structure.
- Nonlinear gates (`Mul`, `AssertZero`) correspond to R1CS constraints; linear gates are free.

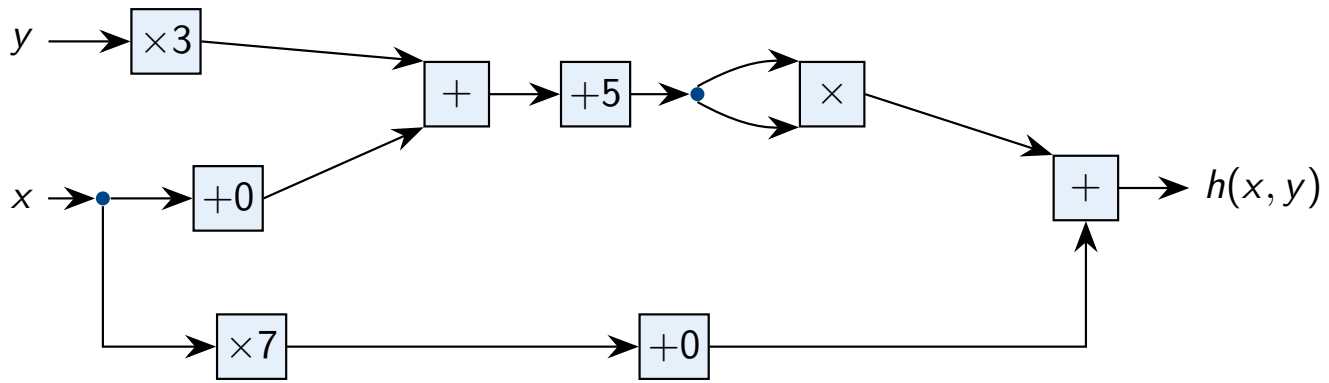


booleanity: check $b \cdot (b - 1) = 0$, i.e. $b \in \{0, 1\}$. Two R1CS constraints: one $\times$, one $= 0$.

Every fan-out is explicit

```
booleanity :: FreeSemi Gate F ()
booleanity =
  SDimap WCopy WId (SPar (SLift (AddConst 0))
                         (SLift (AddConst (mkF (-1)))))
    ⋙ SLift Mul
    ⋙ SLift AssertZero
```

- `WCopy` turns b into (b, b) — fan-out is a visible wiring step, never an implicit variable reuse.
- Inside `SPar`: the left branch passes b through unchanged, the right branch computes $b - 1$.
- Then multiply, then assert zero. Reading the term *is* reading the circuit.



tovCompress: $h(x, y) = (x + 3y + 5)^2 + 7x$ — one nonlinear gate.

The R1CS cost model is just . . . a semiarrow

- R1CS: each constraint is a rank-1 (bilinear) form $\langle \vec{a}, \vec{w} \rangle \cdot \langle \vec{b}, \vec{w} \rangle = \langle \vec{c}, \vec{w} \rangle$ — so the cost is the number of genuine multiplications.

```
newtype Cost a b = Cost { getCost :: Int }

instance Semiarrow Cost where
  Cost m ⋙ Cost n = Cost (m + n)

costGate Mul        = Cost 1
costGate AssertZero = Cost 1
costGate _          = Cost 0

cost = getCost . interpretFree costGate
```

- `Cost` is a semiarrow, so `interpretFree costGate` lifts the per-gate assignment to *any* circuit term — for free.

```
cost square       -- 1: one Mul gate
cost booleanity   -- 2: Mul + AssertZero
cost toyCompress  -- 1: one Mul (inside square)
```

The laws become a normalization procedure

- The semiarrow laws can be *oriented* as rewrites on the syntax:

```
normSeq (SSeq f g) h  = normalize (SSeq f (SSeq g h))
normSeq (SPar f g) (SPar h k)
                      = normalize (SPar (SSeq f h) (SSeq g k))
normSeq f g           = SSeq f g
```

- First equation: sequential associativity (**S1**), oriented right — one representative per pipeline.
- Second equation: the interchange law (**I1**) as a rewrite — it exposes independence directly in the syntax.
- The rewrite is justified *because* it is an instance of a semiarrow law, not an ad hoc compiler optimization.

The moral

Arrows are too powerful for inspectable DSLs: `arr` hides arbitrary computation. Semiarrows sit at the right level: exactly the structural operations circuits need — parallel, sequential, wiring — with no escape hatch for opaque closures.

Related work

- **Monoidal profunctors & semiarrows.** Monoidal profunctors: Oliveira, Jaskelioff & de Melo (MSFP 2022). The term *semiarrow* and the eight laws: the author's PhD thesis (2023). New here: the free construction, *interpretFree* + uniqueness, the `Wiring` GADT, and the circuit application.
- **Notions of computation as monoids.** Rivas & Jaskelioff: monads, applicatives, arrows as monoids. Semiarrows fit the same frame: a *semigroup* under $\bullet$ + a monoid under $\star$.
- **Arrows.** Hughes; Freyd categories (Jacobs et al.). It is `arr` that supplies the extra structure arrows need — semiarrows drop it.
- **GArrows.** Megacz also removes `arr` for inspectability. Here the host structure is a semiarrow, and *interpretFree* comes with a uniqueness guarantee.
- **Free applicatives.** The free semiarrow is the profunctorial generalisation adding a sequential layer.

More in the paper, and future work

- More details in the paper:
  - full proof of the homomorphism theorem (existence + uniqueness);
  - the Moore-machine and relational (`CoState Bool`) semiarrows worked out;
  - the complete normalization procedure and its justification.
- Future work:
  - pin down `Wiring` as a universal property (the cartesian generators: copy Δ, discard ε, symmetry);
  - recover wiring automatically à la Bernardy & Spiwack (evaluating linear functions into symmetric monoidal categories);
  - *traced* semiarrows for circuits with feedback;
  - a "semiarrow calculus": variable-binding syntax that translates using only `Wiring` generators — the arrow calculus without `arr`;
  - quotient `FreeSemi` by the laws in Cubical Agda and check the homomorphism theorem by machine.

Thank you!