Free Semiarrows

Alexandre Garcia de Oliveira
Fatec-Rubens Lara Santos, Brazil
Obsidian Systems
alexgrcol (at) hotmail.com

This work studies *semiarrows*: profunctors that carry both a monoid structure under the Day convolution (parallel composition with unit) and a semigroup structure under profunctor composition (sequential composition without identity), linked by an interchange law and a unit interaction law. The term and definition were introduced in the author's doctoral thesis, generalising monoidal profunctors and arrows.

We construct the *free semiarrow* over an arbitrary base profunctor p, presented as a GADT *FreeSemi p* with five constructors corresponding to the semiarrow operations. We give the interpretation function *interpretFree*, which, given a natural transformation from p to any target semiarrow q, produces the unique semiarrow homomorphism extending the natural transformation.

As a running example we develop an arithmetic-circuit DSL over a prime field. Primitive gates form a GADT; *FreeSemi RelGate* is the resulting circuit term algebra for constraint checking. Because no *arr* combinator is available, every fan-out and structural routing must appear explicitly, making every circuit term inspectable. We present an interpretation of the R1CS cost model via *interpretFree*.

1 Introduction

Functors, applicative functors [9], monads [11, 17], profunctors, and arrows [5, 6] are now part of the vocabulary of the programmer who writes mathematically structured programs. In [15], Rivas and Jaskelioff show that monads and applicative functors are both instances of the same idea—a monoid in a monoidal category of endofunctors—differing only in the choice of tensor product: functor composition for monads and the Day convolution for applicative functors. The same perspective applies to arrows: an arrow is a monoid in a category of strong profunctors under profunctor composition.

In prior work [12, 13], the author studied *monoidal profunctors*, which are monoids in a category of profunctors under the Day convolution alone. Monoidal profunctors give parallel composition with a unit but no sequential composition at all. Arrows sit at the other extreme: they give sequential composition with an identity and, via strength, can simulate parallel composition. The term *semiarrow* [13] was introduced to name the structure that lies between: a profunctor that has *both* a full monoidal-profunctor structure (parallel composition with unit) *and* a semigroup structure under profunctor composition (sequential composition without identity), linked by an interchange law and a unit interaction law.

This paper constructs the *free semiarrow* over an arbitrary base profunctor. The construction is motivated by a concrete application: an arithmetic-circuit DSL for constraint systems used in proof systems.

The inspectability problem. Consider a simple *squaring circuit*: a circuit that takes one field element x over a prime field $\mathbb{F}_p$ and outputs its square x^2. Arithmetic circuits of this kind are the basic building blocks of the constraint systems used in proof systems, where a computation is expressed as a fixed network of addition and multiplication gates over $\mathbb{F}_p$. Throughout, a value of type *p a b* should be read

Submitted to:
MSFP 2026

as a circuit component (or process) with input wires of type a and output wires of type b; the type parameters track the wires, and the structural operations wire such components together in *sequence* (feeding outputs to inputs) and in *parallel* (placing components side by side). In the arrow world one might write

$$square\_arr = arr\ (\lambda x \to x * x)$$

This is perfectly valid, but the implementation of multiplication is hidden inside a Haskell closure. A compiler that wishes to count nonlinear constraints, emit *rank-1 constraint system* (R1CS) constraints (Section 5.2), or apply circuit optimisations cannot inspect the body of *arr*.

In the free-semiarrow world, the same 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* (and with it *arr id*) forces every structural rewiring to be expressed through the *Wiring* GADT rather than through opaque Haskell functions. For circuit compilation this is not a restriction: arithmetic circuits have a fixed set of primitive gates and a fixed wiring structure, so there is no computation to hide.

Contributions.

1. We define semiarrows (Section 2.4) with the complete set of eight laws, and present the corresponding Haskell interface (Section 3).
2. We present the free semiarrow *FreeSemi p* as a GADT with five constructors (Section 4), give its semiarrow instances, and define the interpretation function *interpretFree*.
3. We prove that *interpretFree* φ is the unique semiarrow homomorphism from *FreeSemi p* to any target semiarrow q extending φ (Theorem 4.1).
4. We develop an arithmetic-circuit DSL (Section 5) and interpret its R1CS cost model via *interpretFree*.

Paper organisation. Section 2 reviews profunctors, the Day convolution, the two-tensor structure, and gives the categorical definition of semiarrows (Section 2.4). Section 3 presents the corresponding Haskell interface. Section 4 constructs the free semiarrow and proves the homomorphism theorem. Section 5 develops the arithmetic-circuit application. Section 5.3 discusses why semiarrows, unlike arrows, are suitable for inspectable DSLs and explains the normalization procedure induced by the semiarrow laws. Section 6 discusses related work and Section 7 concludes.

2 Background

2.1 Profunctors

A *profunctor* $p : \mathscr{C}^{op} \times \mathscr{C} \to \mathbf{Set}$ is a bifunctor contravariant in its first argument and covariant in its second. In Haskell, profunctors are encoded as a type class:

```
class Profunctor p where
  type Hom p :: * → * → *
  type Hom p = (→)
  dimap :: Hom p a b → Hom p c d → p b c → p a d
```

The *Hom p* type family names the morphisms used for re-indexing; when $Hom\ p = (\to)$, we recover the standard setting in which *dimap f g* pre-composes with f and post-composes with g.

The function arrow $(\to)$ is the prototypical profunctor: $dimap\ f\ g\ h = g \circ h \circ f$.

For the circuit DSL, all wiring is expressed using *Wiring* (a syntactic subset of ordinary Haskell functions) and the target semiarrows accept Haskell functions for re-indexing.

The morphisms used for re-indexing need not be Haskell functions: the thesis [13] replaces them by the arrows of any chosen category, which is what lets one thread effects through the wiring. We do not need that generality here. In this paper re-indexing always uses ordinary Haskell functions, and inside the free construction it uses an even smaller set—the *Wiring* terms of Section 3.1—which keeps every routing step inspectable.

2.2 Day Convolution

Day convolution [4] is the standard way of taking two functors and combining them using the tensor of the base category. For functors $F, G : \mathscr{C} \to \mathbf{Set}$ on a monoidal category $(\mathscr{C}, \otimes, I)$ it is

$$(F * G)(C) \;=\; \int^{A\,B} \mathscr{C}(A \otimes B, C) \times F\,A \times G\,B.$$

The coend $\int$ here can be read informally as "collect the pieces $F\,A$ and $G\,B$ over all the ways of splitting C as $A \otimes B$". Profunctors are themselves functors (out of $\mathscr{C}^{op} \times \mathscr{C}$), so the same recipe applies. Specialising it to that functor category, with the product tensor on $\mathscr{C}^{op} \times \mathscr{C}$, gives—up to natural isomorphism—the coend

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

[4, 15]. The unit for this tensor is the profunctor

$$J(A,B) \;=\; \mathscr{C}(A,I) \times \mathscr{C}(I,B)$$

where I is the monoidal unit of $\mathscr{C}$. An element of $J(A,B)$ is a pair of morphisms $(A \to I,\ I \to B)$; a monoid structure $e : J \to P$ corresponds, by the coend computation, to an element of $P(I,I)$.

In Haskell, where $\mathscr{C} = \mathbf{Set}$ and $\otimes = \times$, this simplifies a lot. A *monoidal profunctor* [12, 15] is then just a profunctor with two extra operations: an empty one, and a way to run two profunctors in parallel that pairs up their inputs and their outputs,

$$P(A,B) \times P(C,D) \;\to\; P(A \times C, B \times D).$$

In Haskell these two operations become:

```
class Profunctor p ⇒ MonoPro p where
  mpempty :: p () ()
  (⋆) :: p a b → p c d → p (a,c) (b,d)
```

The expected laws are the standard monoid laws up to the canonical isomorphisms $(a \times (), a)$, $((), a) \cong a$, and $(a \times b) \times c \cong a \times (b \times c)$:

1. *Left unit:* $mpempty \star p = p$ (up to $((), a) \cong a$)
2. *Right unit:* $p \star mpempty = p$ (up to $(a, ()) \cong a$)
3. *Associativity:* $(p \star q) \star r = p \star (q \star r)$ (up to the associator)

2.3 Profunctor Composition

The second tensor on profunctors is *profunctor composition*

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

which threads the output of P into the input of Q through a shared intermediate type Z [15]. It is associative, and its unit is the hom-profunctor $\mathscr{C}(A,B)$, which behaves as the identity on every object. So the two tensors do different jobs: Day convolution is a *parallel* tensor (it places components side by side), while profunctor composition is a *sequential* tensor (it puts them one after another). A monoid under this sequential tensor—with a little extra structure called strength—is exactly an *arrow* [5, 1, 6], and the unit of that monoid is the sequential identity, which in Haskell is *arr id*.

2.4 Two Tensors and Semiarrows

Two distinct tensor products exist on the category of profunctors when the category itself is monoidal [15, 13]. The *Day convolution* ($\star$), discussed in Section 2.2, is the parallel tensor: it pairs inputs and outputs side by side. *Profunctor composition* ($\bullet$), discussed in Section 2.3, is the sequential tensor: it chains the output of one profunctor into the input of another through a shared intermediate type. These two tensors give rise to two distinct algebraic structures.

Recall that a *monoid* in a monoidal category is an object with an associative multiplication and a unit [8]. Taking the tensor to be Day convolution gives monoidal profunctors [12]; taking it to be profunctor composition (plus strength) gives arrows [5, 1, 6].

A *semigroup* [13] is a monoid that has dropped its unit: it keeps only the associative multiplication $M \otimes M \to M$. That small change is exactly the step from arrows to semiarrows: we keep sequential composition but drop its identity (*arr id*).

A *semiarrow* is then the structure that is a monoid under one tensor and a semigroup under the other: a monoid under Day convolution ($\star$), giving the full monoidal-profunctor structure of parallel composition with unit, and a semigroup under profunctor composition ($\bullet$), giving sequential composition without identity. The two structures are linked by an interchange law and a unit interaction law, ensuring that parallel and sequential composition interact coherently.

Every arrow is a semiarrow, but not conversely: an arrow requires strength and an identity for sequential composition (*arr id*), while a semiarrow requires neither.

Definition 2.1 (Semiarrow [13])**.** *A* semiarrow *is a monoidal profunctor p equipped with a binary operation*

$$(\ggg) : p\,a\,b \to p\,b\,c \to p\,a\,c$$

satisfying the following eight laws.

P1 Profunctor left: $dimap\ (f \circ g)\ id\ p = dimap\ g\ id\ (dimap\ f\ id\ p)$

P2 Profunctor right: $dimap\ id\ (f \circ g)\ p = dimap\ id\ f\ (dimap\ id\ g\ p)$

M1 Parallel left unit: $mpempty \star p = p$ *(up to* $((), a) \cong a$*)*

M2 Parallel right unit: $p \star mpempty = p$ *(up to* $(a,()) \cong a$*)*

M3 Parallel associativity: $(p \star q) \star r = p \star (q \star r)$ *(up to associator)*

S1 Sequential associativity: $(p \ggg q) \ggg r = p \ggg (q \ggg r)$

I1 Interchange: $(f \star g) \ggg (h \star k) = (f \ggg h) \star (g \ggg k)$

U1 Unit interaction: $mpempty \ggg mpempty = mpempty$

Law **I1** states that two independent pairs of processes, composed sequentially and then placed in parallel, equal the two placed in parallel and then composed sequentially. In circuit terms: independent sub-circuits can be sequenced and parallelised in either order. Law **U1** relates the sequential multiplication to the parallel unit: composing two empty parallel circuits sequentially gives an empty parallel circuit.

There is no identity for $(\ggg)$. An identity would require *arr id*, which is exactly what semiarrows omit.

Remark 2.2 (Why drop the sequential identity?)**.** *For some structures, such as the Moore machines of Example 3.4, a sequential identity simply does not exist. For others, such as arithmetic circuits, an identity* wire *does exist—but it is already available structurally as the WId constructor of the Wiring GADT (Section 3.1), applied through dimap. Dropping the* categorical *identity for* $(\ggg)$ *therefore costs circuits no expressiveness: the identity wire is recovered as a wiring morphism rather than as an opaque arr id. What is gained is uniformity (the same free construction covers Moore machines and circuits alike) and inspectability (no arr means no hidden computation), at the price of a weaker algebraic structure—a semigroup rather than a monoid under sequential composition.*

3 Semiarrows in Haskell

Definition 2.1 gave the categorical definition of semiarrows. In Haskell, the operations become methods of a type class that extends *MonoPro*, and the eight laws become proof obligations on any instance.

Definition 3.1. *A Haskell* semiarrow instance *for p is an instance of:*

```
class MonoPro p ⇒ Semiarrow p where
  (⋙) :: p a b → p b c → p a c
```

satisfying the eight laws of Definition 2.1, including ***U1****:* $mpempty \ggg mpempty = mpempty$*.*

3.1 The Wiring GADT

In the free-semiarrow construction, *dimap* takes values in a restricted class of morphisms rather than arbitrary Haskell functions. This restriction is the key to structural inspectability.

Definition 3.2. *The* wiring *type Wiring a b is the GADT of canonical structural maps between product types:*

```
data Wiring a b where
  WId     :: Wiring a a
  WSwap   :: Wiring (a,b) (b,a)
  WAssocL :: Wiring (a,(b,c)) ((a,b),c)
```

```
WAssocR :: Wiring ((a,b),c) (a,(b,c))
WFst    :: Wiring (a,b) a
WSnd    :: Wiring (a,b) b
WCopy   :: Wiring a (a,a)
WDel    :: Wiring a ()
```

The constructors correspond to the generators of cartesian structure: identity, swap, associativity, projections, diagonal, and terminal map. Semantically, $runWiring :: Wiring\ a\ b \to a \to b$ interprets each constructor as the expected Haskell function.

Every *Wiring* term is fully inspectable: a compiler can pattern-match on it and treat each case symbolically (see Section 5).

Remark 3.3. *In an arrow, dimap may use any function $a \to b$ for re-indexing. By restricting to Wiring, the free semiarrow ensures that no arbitrary computation is hidden in a re-indexing step. Every fan-out and structural routing must be expressed explicitly.*

3.2 Examples of Semiarrows

Example 3.4. *A* Moore machine *is a reactive process that stores output state and advances on each input tick:*

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

The profunctor instance re-indexes the output and the inputs of every future step; the monoidal instance runs two machines in lockstep, pairing their outputs and splitting the paired input:

```
instance Profunctor Moore where
  dimap f g (Moore b k) = Moore (g b) (dimap f g . k . f)
instance MonoPro Moore where
  mpempty = Moore () (\_ -> mpempty)
  Moore b k ⋆ Moore d l =
    Moore (b,d) (λ(a,c) -> k a ⋆ l c)
```

Sequential composition introduces a one-step delay: the current output of m1 ⋙ m2 is output m2, and to advance with input a the machine advances m2 using m1's stored *(not current) output:*

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

There is no identity for (⋙): an identity would have to produce its current input as output, but a Moore machine's output is always a stored value from the previous step. This is precisely the reason semiarrows lack a sequential identity.

Example 3.5. *For the arithmetic-circuit application of Section 5, we want a relational semantics that models* constraint checking *rather than evaluation. The naive choice $a \to [b]$ (a relation that enumerates*

or computes possible outputs) does not fit: in a constraint system the output is not computed but supplied externally as a witness*, and the circuit's job is only to* verify *that the witness satisfies the gate equations. The representation that models this verifier reading is:*

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

A value of type Rel a b stores a chosen output b and a predicate a → Bool on inputs. Sequential composition feeds the first stage's witness into the second stage's predicate:

```
instance Semiarrow (CoState Bool) where
  CoState b p ⋙ CoState c q =
    CoState c (λa → p a ∧ q b)
```

This requires only a semigroup operation on Bool (conjunction); no sequential identity is needed. The monoidal instance uses:

```
instance MonoPro (CoState Bool) where
  mpempty = CoState () (\_ → True)
  CoState b p ⋆ CoState d q =
    CoState (b,d) (λ(a,c) → p a ∧ q c)
```

A primitive relation mulRel z :: Rel (F,F) F stores a witness z and checks $z \equiv x * y$ for the actual inputs (x,y). The circuit does not compute z; it checks whether the supplied z satisfies the gate equation.

Example 3.6. *The function arrow (→) is a semiarrow with $f \ggg g = g \circ f$. It is in fact an arrow, so every arrow is a semiarrow. The function arrow has an identity for (⋙), namely id; this is what distinguishes arrows from semiarrows.*

4 Free Semiarrows

4.1 The FreeSemi GADT

The free semiarrow over a base profunctor *p* is defined as the following GADT:

```
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
```

Each constructor corresponds to exactly one semiarrow operation. *SLift p* embeds a value of the base profunctor *p* into the free semiarrow and is the unit of the adjunction. *SSeq s1 s2* is sequential composition corresponding to (⋙). *SPar s1 s2* is parallel composition corresponding to (⋆). *SEmpty* is

the unit for parallel composition corresponding to *mpempty*. *SDimap f g s* is profunctorial re-indexing by *Wiring* morphisms, corresponding to *dimap*.

The GADT is a raw syntax tree, not quotiented by the semiarrow laws. *FreeSemi p* is the free semiarrow in the sense that *interpretFree* (below) gives the unique semiarrow homomorphism out of it.

The instances for *FreeSemi p* are trivial by construction:

```
instance Profunctor (FreeSemi p) where
  type Hom (FreeSemi p) = Wiring
  dimap = SDimap
instance MonoPro (FreeSemi p) where
  mpempty = SEmpty
  (⋆) = SPar
instance Semiarrow (FreeSemi p) where
  (⋙) = SSeq
```

Every operation constructs a corresponding node in the syntax tree.

4.2 Interpretation

Given a semiarrow q and a natural transformation $\varphi : p \Rightarrow q$, we define a function $interpretFree_{\varphi} : FreeSemi\ p \Rightarrow q$ by recursion on the syntax tree:

```
interpretFree :: (Semiarrow q, Hom q ~ (→))
  ⇒ (∀x y. p x y → q x y)
  → FreeSemi p a b → q a b
interpretFree φ (SLift p)      = φ p
interpretFree φ (SSeq s1 s2) = interpretFree φ s1 ⋙ interpretFree φ s2
interpretFree φ (SPar s1 s2) = interpretFree φ s1 ⋆ interpretFree φ s2
interpretFree _ SEmpty         = mpempty
interpretFree φ (SDimap f g s) = dimap (runWiring f) (runWiring g)
  (interpretFree φ s)
```

The *SDimap* case calls *runWiring* to convert each *Wiring* morphism into an ordinary Haskell function before calling *dimap* in the target semiarrow.

Theorem 4.1 (Semiarrow homomorphism)**.** *Let p be a profunctor and let q be a semiarrow. For any natural transformation $\varphi : p \Rightarrow q$, the function $interpretFree_{\varphi}$ is the unique semiarrow homomorphism from FreeSemi p to q extending φ:*

1. $interpretFree_{\varphi}(SDimap\, f\, g\, s) = dimap\,(runWiring\, f)\,(runWiring\, g)(interpretFree_{\varphi}(s))$
2. $interpretFree_{\varphi}(SSeq\, s_1\, s_2) = interpretFree_{\varphi}(s_1) \ggg interpretFree_{\varphi}(s_2)$
3. $interpretFree_{\varphi}(SPar\, s_1\, s_2) = interpretFree_{\varphi}(s_1) \star interpretFree_{\varphi}(s_2)$
4. $interpretFree_{\varphi}(SEmpty) = mpempty$

Proof. Existence. *Each clause holds by the definition of interpretFree, directly from the corresponding case of the recursion. Since each operation maps to the corresponding operation in q, and q satisfies the eight laws of Definition 2.1, the image under interpretFree satisfies all eight laws.*

Uniqueness. *Let $h : FreeSemi\ p \to q$ be any semiarrow homomorphism extending φ. A semiarrow homomorphism must satisfy: $h(SLift\ p) = \varphi(p)$, $h(SSeq\ s_1\ s_2) = h(s_1) \ggg h(s_2)$, $h(SPar\ s_1\ s_2) = h(s_1) \star h(s_2)$, $h(SEmpty) = mpempty$, and $h(SDimap\ f\ g\ s) = dimap\ (runWiring\ f)\ (runWiring\ g)\ (h(s))$. These equations uniquely determine h by structural induction on the GADT. Hence $h = interpretFree_{\varphi}$.*

5 Arithmetic Circuits

5.1 The Gate GADT

We model arithmetic circuits over the prime field $\mathbb{F}_{97}$ (a small field used for executable examples; it is not cryptographically realistic). The base profunctor is:

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

Each constructor is a primitive operation of the circuit. The type parameters record the number of input and output wires at the type level: for example, *Mul* :: *Gate* (F,F) F takes two field elements and produces one.

AssertZero returns $()$ rather than a field element: constraint gates—gates that enforce an algebraic identity—fit naturally into the type structure. Non-linear gates (*Mul*, *AssertZero*) correspond to R1CS [16] constraints; linear gates (*Add*, *Neg*, *MulConst*, *AddConst*) are free.

Square. The circuit $x \mapsto x^2$ requires exactly one multiplication gate. Fan-out—duplicating the input wire—is expressed via *WCopy*:

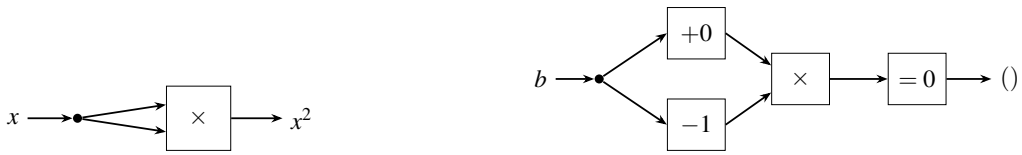
```
square :: FreeSemi Gate F F
square = SDimap WCopy WId (SLift Mul)
```

SDimap WCopy WId pre-processes the input: *WCopy* turns x into the pair (x,x), which is passed to *SLift Mul* (Figure 1a). No other gate is needed; the circuit has exactly one R1CS constraint.

Booleanity check. To verify $b \in \{0,1\} \subset \mathbb{F}_{97}$ it suffices to check $b \cdot (b-1) = 0$. The circuit fans out b, computes b and $b-1$ in parallel, multiplies, then asserts the result is zero:

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

Step by step (Figure 1b): *WCopy* turns b into (b,b). Inside *SPar*, the left branch *SLift* (*AddConst* 0) passes b through unchanged (adding zero is the identity wire); the right branch *SLift* (*AddConst* (*mkF* (-1))) computes $b-1$. The pair $(b,b-1)$ is multiplied by *SLift Mul*, and the product is asserted zero by *SLift AssertZero*. The circuit has two R1CS constraints: one *Mul* and one *AssertZero*.



(a) *square* (b) *booleanity*

Figure 1: Two small circuits as *FreeSemi Gate* terms. A filled dot is a fan-out (*WCopy*) and each box is one gate. The only gates that cost an R1CS constraint are the multiplications ($\times$) and the zero-assertion ($=0$); the $+0$ and -1 boxes are linear and free.

Toy compression round. To illustrate parallel composition and wire projection, consider $h(x,y) = (x+3y+5)^2+7x$:

```
toyCompress :: FreeSemi Gate (F, F) F
toyCompress =
  SDimap WCopy WId (SPar computeLin computeSevenX)
   ⋙ SPar square (SLift (AddConst 0))
   ⋙ SLift RAdd
  where
    computeLin =
      SPar (SLift (AddConst 0)) (SLift (MulConst (mkF 3)))
       ⋙ SLift Add
       ⋙ SLift (AddConst (mkF 5))
    computeSevenX = SDimap WFst WId (SLift (MulConst (mkF 7)))
```

The data flow is shown gate by gate in Figure 2. *WCopy* duplicates the input. The branch *computeLin* builds $x+3y+5$ one gate at a time: *AddConst* 0 keeps x, *MulConst* 3 forms $3y$, *Add* sums the two, and *AddConst* 5 adds the constant. The branch *computeSevenX* projects x with *WFst* and scales it by *MulConst* 7 to get $7x$. Then *square* fans $x+3y+5$ out and multiplies it by itself, the $7x$ wire passes through *AddConst* 0 unchanged, and a final *Add* sums the two. The only non-linear gate is the *Mul* inside *square*, so the whole circuit costs one R1CS constraint.

5.2 Cost Model

A *rank-1 constraint system* (R1CS) is the constraint representation used by many proof systems for verifiable computation: a computation is encoded as a list of constraints, each of the *rank-1* (bilinear) form $\langle \vec{a},\vec{w}\rangle \cdot \langle \vec{b},\vec{w}\rangle = \langle \vec{c},\vec{w}\rangle$, where $\vec{w}$ is the vector of wire values and each side is an affine combination of wires—hence the name: each constraint multiplies exactly two affine forms ($\ell_1 \cdot \ell_2 = \ell_3$). Affine (linear) gates fold into these combinations for free, so the size—and the dominant cost—of an R1CS is the number of genuine multiplications. Accordingly we charge one unit per non-linear gate (*Mul*, *AssertZero*) and zero for all linear gates and wiring.

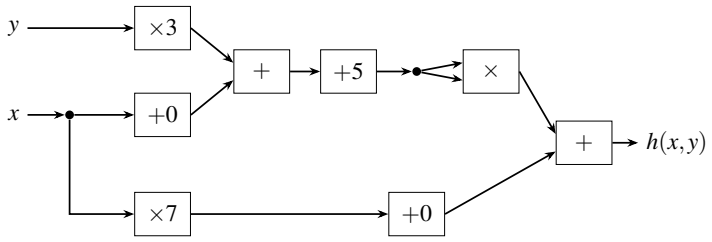


Figure 2: *toyCompress*, computing $h(x,y) = (x+3y+5)^2 + 7x$, drawn gate by gate. A filled dot is a fan-out (*WCopy*); the upper row is *computeLin* feeding *square*, the lower row is *computeSevenX*. The only non-linear gate is the multiplication ($\times$) inside *square*, so the circuit costs one R1CS constraint.

The cost model is a semiarrow *Cost* that carries an integer. Sequential and parallel composition add costs; *mpempty* and *SDimap* contribute zero:

```
newtype Cost a b = Cost {getCost :: Int}
instance Semiarrow Cost where
  Cost m >>> Cost n = Cost (m + n)
instance MonoPro Cost where
  mpempty = Cost 0
  Cost m * Cost n = Cost (m + n)
```

The per-gate assignment maps non-linear gates to 1 and everything else to 0:

```
costGate :: Gate a b -> Cost a b
costGate Mul = Cost 1
costGate AssertZero = Cost 1
costGate _   = Cost 0
```

Because *Cost* is a semiarrow, *interpretFree costGate* lifts *costGate* to any *FreeSemi Gate* term, summing costs through *SSeq* and *SPar*:

```
cost :: FreeSemi Gate a b -> Int
cost = getCost . interpretFree costGate
```

Running *cost* on the three circuits (using the *Gate* variants from `Circuit.hs`):

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

interpretFree costGate folds *costGate* over every *SLift* node and adds costs through *SSeq* and *SPar*. The result for *toyCompress* is 1: all linear gates contribute 0, and the only non-linear gate is the *Mul* inside *square*.

5.3 Discussion

Section 1 motivated semiarrows through the inspectability problem: an arrow term such as *arr* ($\lambda x \to x * x$) hides its multiplication inside a Haskell closure, so no interpreter operating on the value can count

gates or emit constraints. We can now state the consequence precisely for the free semiarrow. Because *FreeSemi* has no *arr* combinator, every computation must be expressed using its five constructors, and every primitive operation must appear as an *SLift* node wrapping a *Gate* value. Any interpreter (cost counter, evaluator, symbolic compiler) can inspect every node by pattern-matching; the type of each gate is known at every node ($Mul :: Gate\ (F,F)\ F$); and wiring and routing are explicit (fan-out is *WCopy*, projection is *WFst* or *WSnd*). The interchange law **I1** guarantees that parallel-then-sequential equals sequential-then-parallel for independent branches. In circuit compilation this is essential: the constraint emitter for *toyCompress* can treat *computeLin* and *computeSevenX* independently, because **I1** ensures the order of composition does not change the result. Arrows do not need **I1** in this form because *arr* can express any function; semiarrows, lacking *arr*, expose the independence structure directly in the syntax. The unit interaction law **U1** ensures $mpempty \ggg mpempty = mpempty$: a pipeline of empty circuits is still empty. Arrow's *arr id* serves as the sequential identity and plays an analogous coherence role; semiarrows replace *arr id* with the weaker **U1**, which is sufficient for the circuit application while avoiding opaque function embedding.

These laws can be oriented as a normalization procedure on the syntax.

$$
\begin{aligned}
&normalize :: FreeSemi\ p\ a\ b \to FreeSemi\ p\ a\ b\\
&normalize\ (SLift\ p) = SLift\ p\\
&normalize\ SEmpty = SEmpty\\
&normalize\ (SDimap\ f\ g\ s) = SDimap\ f\ g\ (normalize\ s)\\
&normalize\ (SPar\ f\ g) = SPar\ (normalize\ f)\ (normalize\ g)\\
&normalize\ (SSeq\ f\ g) = normSeq\ (normalize\ f)\ (normalize\ g)\\
&normSeq :: FreeSemi\ p\ a\ b \to FreeSemi\ p\ b\ c \to FreeSemi\ p\ a\ c\\
&normSeq\ (SSeq\ f\ g)\ h =\\
&\quad normalize\ (SSeq\ f\ (SSeq\ g\ h))\\
&normSeq\ (SPar\ f\ g)\ (SPar\ h\ k) =\\
&\quad normalize\ (SPar\ (SSeq\ f\ h)\ (SSeq\ g\ k))\\
&normSeq\ f\ g =\\
&\quad SSeq\ f\ g
\end{aligned}
$$

The first equation of *normSeq* is sequential associativity oriented to the right. It does not add new behavior; it merely chooses one representative for a pipeline whose parenthesization is irrelevant by the semiarrow law. The second equation is the interchange law **I1** oriented as a rewrite: a sequential composition of two parallel terms is rewritten as a parallel composition of two sequential terms, when the types line up. This exposes the fact that the two branches are independent in the sense already guaranteed by **I1**. The final equation leaves all other sequential compositions unchanged.

For circuits, this matters because the algebraic laws justify rearranging the syntax before interpretation. For example, if a circuit has two independent branches such as *computeLin* and *computeSevenX*, the interchange law permits reasoning about them branchwise. A cost interpreter, a relational interpreter, or a constraint emitter can then follow the same semiarrow structure: the meaning is preserved because the rewrite is exactly an instance of a semiarrow law, not because of an ad hoc compiler optimization.

Summing up, arrows are too powerful for inspectable DSLs: *arr* hides arbitrary computation. Semiarrows occupy the right level: they provide exactly the structural operations (parallel, sequential, wiring) that arithmetic circuits need, with no escape hatch for opaque closures.

6 Related Work

Monoidal profunctors and semiarrows. The monoidal profunctor structure was studied systematically by the author in [12], where the gap in the two-tensor table (Figure 1 in that paper) was first identified. The term *semiarrow* and the definition with the eight laws (Definition 2.1) appear in the author's doctoral thesis [13]; the present paper reuses only that definition. Everything else is new and was not in the thesis: the free construction *FreeSemi p* (Section 4), the interpreter *interpretFree* and the proof that it is the *unique* semiarrow homomorphism out of *FreeSemi p* (Theorem 4.1), the *Wiring* GADT as an inspectable replacement for *arr*, and the entire arithmetic-circuit application with its R1CS cost model (Section 5). In short, the thesis introduced semiarrows as an algebraic structure; this paper gives their free term algebra and a concrete use for it.

Notions of computation as monoids. Rivas and Jaskelioff [15] give a unified account of monads, applicative functors, and arrows as monoids in monoidal categories of (endo)functors or profunctors. Semiarrows fit the same framework: a semigroup under profunctor composition combined with a full monoid under Day convolution.

Arrows. Hughes [5] introduced arrows and the *Arrow* type class. The Haskell *Arrow* class requires *arr* and *first*; *arr* lifts arbitrary Haskell functions, and *arr id* serves as the identity for sequential composition. Categorically, arrows are captured by Freyd categories [6], and it is *arr* that supplies the extra structure they need. Semiarrows drop *arr*, and with it both unrestricted function lifting and the sequential identity, so that extra structure is no longer required.

Free applicative functors. Applicative functors [9] are the unary-functor analogue of monoidal profunctors, and free applicatives are their syntactic counterpart. The free semiarrow is the profunctorial generalisation that adds the sequential layer.

GArrows. Megacz's GArrows [10] also remove *arr* from arrows for inspectability, replacing it with a category of structural morphisms (generalised arrows). The *Wiring* GADT plays the same role here: it provides identity, swap, associativity, projection, copy, and discard without admitting arbitrary functions. The present work differs in that the host structure is a semiarrow (monoidal profunctor with sequential semigroup) rather than a generalised arrow; the constructors of *FreeSemi* are derived from the semiarrow operations; and *interpretFree* is a semiarrow homomorphism with a uniqueness guarantee.

Profunctor optics. Profunctor optics [14, 3] use profunctors whose *dimap* ranges over restricted morphism classes (Tambara modules, etc.). The restriction of *dimap* to *Wiring* in the free semiarrow is philosophically related: both restrict the morphisms available to the profunctor in order to capture exactly the desired computational structure.

7 Conclusion

We have presented the free semiarrow *FreeSemi p*, a GADT with five constructors corresponding exactly to the semiarrow operations, plus a lifting constructor for base-profunctor values. The *FreeSemi p* GADT carries semiarrow structure by construction. The function *interpretFree*, given any natural transformation

$p \Rightarrow q$ where q is a semiarrow, is the unique semiarrow homomorphism *FreeSemi p* $\Rightarrow q$ extending the natural transformation (Theorem 4.1). Two interpretations of the circuit DSL demonstrate the approach: a relational constraint-checking semantics (*interpretRel* over *FreeSemi RelGate*) and an R1CS cost model (*interpretFree costGate* over *FreeSemi Gate*). Restricting *dimap* to *Wiring* makes every circuit term structurally inspectable: no gate is hidden inside an opaque closure.

Future work. The *Wiring* constructors—*WId*, *WSwap*, *WAssocL*, *WAssocR*, *WFst*, *WSnd*, *WCopy*, *WDel*—are exactly the building blocks of cartesian structure: copying ($\Delta : A \to A \otimes A$) and discarding ($\varepsilon : A \to I$), on top of the usual symmetric monoidal isomorphisms. We would like to pin this down as a universal property, since that would say precisely what *SDimap* means in the semantics of *FreeSemi*. A related and promising route is Bernardy and Spiwack's trick of evaluating ordinary linear functions into a symmetric monoidal category [2], which could recover the same wiring without writing the GADT out by hand. Finally, many circuits have feedback—they are cyclic—and the right tool for that is a traced symmetric monoidal category; whether semiarrows with a trace are a useful structure is a natural next question.

A friendlier syntax. The *FreeSemi* terms of Section 5 are not pleasant to write by hand: the *SDimap*, *SPar*, and *WCopy* nodes pile up and hide the circuit one actually has in mind. What we would really like is to write a circuit with ordinary variables and let a translation produce the combinator term for us. This is exactly what Hughes' arrow notation does for arrows, and what the arrow calculus of Lindley, Wadler, and Yallop [7] does more cleanly. The catch is that their translation leans on *arr* everywhere, and *arr* is the one thing semiarrows do not have. So a "semiarrow calculus" would need a way to bind variables using only the *Wiring* generators—copy, discard, swap, projections—for all the plumbing. This is a nice problem in its own right, and leave it for future work.

Checking the laws by machine. One limitation is that Haskell cannot enforce the eight laws of Definition 2.1, nor quotient *FreeSemi p* by them; we present them on paper in Section 4. A proof assistant with good support for quotients, such as Cubical Agda, would let us take the quotient as a higher inductive type and check the homomorphism theorem by machine, putting the whole development on a formal setting.

References

[1] Kazuyuki Asada (2010): *Arrows Are Strong Monads*. In: *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*, MSFP '10, ACM, New York, NY, USA, pp. 33–42, doi:10.1145/1863597.1863607.

[2] Jean-Philippe Bernardy & Arnaud Spiwack (2021): *Evaluating Linear Functions to Symmetric Monoidal Categories*. In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (Haskell 2021)*, pp. 14–26, doi:10.1145/3471874.3472980.

[3] Bryce Clarke, Derek Elkins, Jeremy Gibbons, Fosco Loregiàn, Bartosz Milewski, Emily Pillmore & Mario Román (2020): *Profunctor optics, a categorical update*. *CoRR* abs/2001.07488. Available at `https://arxiv.org/abs/2001.07488`.

[4] Brian Day (1970): *On closed categories of functors*. In S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney & S. Swierczkowski, editors: *Reports of the Midwest Category Seminar IV*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–38, doi:10.1007/BFb0079385.

[5] John Hughes (2005): *Programming with Arrows*. In: *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP'04, Springer-Verlag, Berlin, Heidelberg, pp. 73–129, doi:10.1007/11546382_2.

[6] Bart Jacobs, Chris Heunen & Ichiro Hasuo (2009): *Categorical semantics for arrows*. *Journal of Functional Programming* 19(3-4), p. 403–438, doi:10.1017/S0956796809007308.

[7] Sam Lindley, Philip Wadler & Jeremy Yallop (2010): *The arrow calculus*. *Journal of Functional Programming* 20(1), pp. 51–69, doi:10.1017/S095679680999027X.

[8] Saunders MacLane (1971): *Categories for the Working Mathematician*. Springer-Verlag, New York. Graduate Texts in Mathematics, Vol. 5.

[9] Conor Mcbride & Ross Paterson (2008): *Applicative Programming with Effects*. *J. Funct. Program.* 18(1), pp. 1–13, doi:10.1017/S0956796807006326.

[10] Adam Megacz (2010): *Multi-Level Languages are Generalized Arrows*. In: *Third Workshop on Mathematically Structured Functional Programming (MSFP 2010)*. Also available as: *A Generalization of Arrows for Parallel Computation*.

[11] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Inf. Comput.* 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[12] Alexandre Garcia de Oliveira, Mauro Jaskelioff & Ana Cristina Vieira de Melo (2022): *On Structuring Functional Programs with Monoidal Profunctors*. *Electronic Proceedings in Theoretical Computer Science* 360, pp. 91–113, doi:10.4204/eptcs.360.7.

[13] Alexandre Garcia de Oliveira (2023): *Programming with monoidal profunctors and semiarrows*. Ph.D. thesis, Universidade de São Paulo, doi:10.11606/T.45.2023.tde-03112023-152323.

[14] Matthew Pickering, Jeremy Gibbons & Nicolas Wu (2017): *Profunctor Optics: Modular Data Accessors*. *The Art, Science, and Engineering of Programming* 1(2), doi:10.22152/programming-journal.org/2017/1/7.

[15] Exequiel Rivas & Mauro Jaskelioff (2017): *Notions of computation as monoids*. *Journal of Functional Programming* 27, p. e21, doi:10.1017/S0956796817000132.

[16] Amir Shpilka & Amir Yehudayoff (2010): *Arithmetic Circuits: A Survey of Recent Results and Open Questions*. *Foundations and Trends in Theoretical Computer Science* 5(3–4), pp. 207–388, doi:10.1561/0400000039.

[17] Philip Wadler (1992): *The Essence of Functional Programming*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, ACM, New York, NY, USA, p. 1–14, doi:10.1145/143165.143169.