Parallel Prefix Sums In An Embedded Hardware Description Language

Yorick Sijsling

May 8, 2015

Supervised by Wouter Swierstra
Thanks to João Paulo Pizani Flor

1 Abstract

A scan (also known as a parallel prefix sum) takes a sequence of inputs $x_1, x_2, \ldots, x_n$ and produces the outputs $x_1, (x_1 \oplus x_2), \ldots, (x_1 \oplus x_2 \oplus \ldots \oplus x_n)$, where $\oplus$ is some binary associative operator. Scans are fundamental building blocks for many efficient parallel algorithms including integer addition, sorting and calculation of convex hulls.

We use Π-Ware, an Embedded Domain-Specific Language (EDSL) for hardware description, to build a representation of scans in the dependently-typed language Agda. We prove the correctness of this implementation and prove properties about how scan circuits can be combined to create new scan circuits.

2 Background

Π-Ware (pronounced piware) is a domain-specific language for hardware circuits embedded in Agda. It was created by João Paulo Pizani Flor's as part of his master's thesis[3] and is in active development.

In Π-Ware, we can describe and simulate circuits. The embedding in Agda can be used to do formal proofs about their properties.

2.1 Scans

One thing you might build hardware circuits for are *scans*.

Scans - also referred to as parallel prefix sums - are constructions which take a sequence of inputs $x_1, x_2, \ldots, x_n$ and produce the outputs $x_1, (x_1 \oplus x_2), \ldots, (x_1 \oplus x_2 \oplus \ldots \oplus x_n)$, where $\oplus$ is some binary associative operator.

The computation of such a scan can be modelled in a *scan circuit*, see figure 1 and 2.

Values flow from top to bottom. At the top are the inputs, at the bottom are outputs. The circles are operation nodes where the $\oplus$ operator is applied. For any number of inputs, different circuits can exist which implement a scan.

In An algebra of scans[1], Ralf Hinze describes how a scan circuit can be constructed from smaller components using tools like fans, stretching, horizontal (parallel)

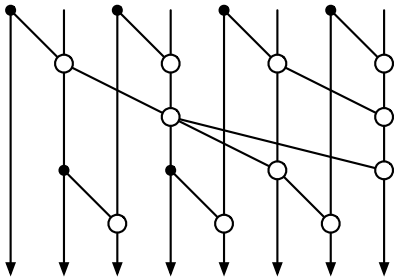

Figure 1: Depth-optimal scan circuit

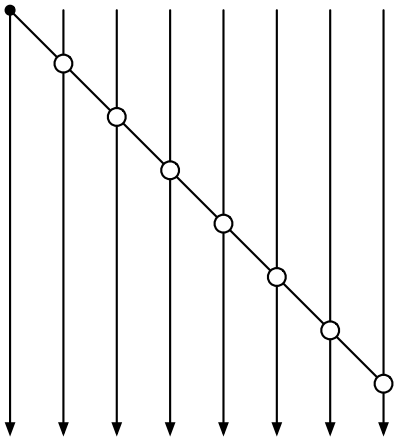

Figure 2: Serial scan circuit

composition and vertical composition (sequencing). The horizontal and vertical composition are also native constructions in Π-ware.

In particular, Hinze builds a few *scan combinators*. These are operators which combine two circuits into a bigger one, with the extra property that if both of the arguments are scan circuits, the result is also a scan circuit.

3 The project

This work was done as part of an experimentation project for 15 ECTS. The main goal was to provide a case study for Π-Ware, to see if it can be applied to implement the algebra of scans.

This report documents what we have learned during the project. While the biggest deliverable is the code itself, this report provides a rationale for some of the choices and serves as documentation for the delivered code.

4 Additions to Π-Ware

4.1 Combinationality

Before this project started, the circuit data type $\mathbb{C}$ looked something like this:

```
data ℂ : ℕ → ℕ → Set where
  Gate : ∀ g → ℂ (|in| g) (|out| g)
  Plug : ∀ {i o} → i ⨯ o → ℂ i o
  _⟫_ : ∀ {i m o} → ℂ i m → ℂ m o → ℂ i o
  _‖_ : ∀ {i₁ o₁ i₂ o₂} → ℂ i₁ o₁ → ℂ i₂ o₂ → ℂ (i₁ + i₂) (o₁ + o₂)
  DelayLoop : ∀ {i o l} (c : ℂ (i + l) (o + l)) {p : comb c} → ℂ i o
```

The two indices of the data type are the input and output size. `Gate` is used to create circuits which execute functions. `Plug` is used to make circuits which just do rewiring, mapping each output to one of the inputs. `⟫` and `‖` are vertical and horizontal composition respectively.

Purely combinational circuits have no internal state and can be simulated without regarding past inputs. Only the constructors Gate, Plug, ⟫ and ‖ are necessary to create purely combinational circuits. Scan circuits are purely combinational too.

The last constructor, DelayLoop, is the only way to create non-combinational circuits. The circuit passed to DelayLoop must however be purely combinational. To enforce this it takes a parameter p : comb c which is a predicate telling whether a circuit is *purely combinational*.

There are some practical issues with this way of handling combinationality. It is often annoying to have to pass the comb proof separately, and in case splits on $\mathbb{C}$ you always need a case for DelayLoop even when there is a comb proof for that circuit.

But most importantly, the proof of combinationality can not be automatically derived in many cases. There were some functions which created some big circuits, and separate proofs were needed to show that they were combinational.

4.1.1 combinationality in the type

We made two important observations:

- comb *restricts which circuits can be constructed*, just like the input and output size.
- Combinationality is something which you *require* for certain functions, but you usually don't need to calculate whether a circuit is combinational.

These observations prompted the new definition of circuits, where p : CombSeq is now an index of the data type.

```
data CombSeq : Set where
  σ : CombSeq -- Required to be combinational
  ω : CombSeq -- Allowed to be sequential
data ℂ′ : {p : CombSeq} → ℕ → ℕ → Set where
  Gate  : ∀ {p} g  → ℂ′ {p} (|in| g) (|out| g)
  Plug  : ∀ {i o p} → i ⨯ o → ℂ′ {p} i o
  _⟫_ : ∀ {i m o p}  → ℂ′ {p} i m  → ℂ′ {p} m o  → ℂ′ {p} i o
  _‖_ : ∀ {i₁ o₁ i₂ o₂ p} → ℂ′ {p} i₁ o₁ → ℂ′ {p} i₂ o₂ →
    ℂ′ {p} (i₁ + i₂) (o₁ + o₂)
  DelayLoop : ∀ {i o l} → ℂ′ {σ} (i + l) (o + l) → ℂ′ {ω} i o
```

Reading the type of _⟫_, we see that if the output circuit is required to be combinational both arguments must be combinational, and if the output circuit is allowed to be sequential the arguments are allowed to be sequential too.

For DelayLoop, the output circuit *must* be allowed to be sequential and the input circuit is required to be combinational.

This solution solves all the practical issues mentioned previously.

The approach is similar to what Conor McBride does in How to Keep Your Neighbours in Order[2].

4.2 Equality

Many properties we want to prove are about *equal behavior* of circuits. More formally, when you simulate two circuits they are equal if they give the same result for every possible input. We require some properties, the first three are needed to be an equivalence relationship:

- E1. Reflexivity, any circuit is equal to itself.
- E2. Symmetry, if f is equal to g then g is equal to f.
- E3. Transitivity, if f is equal to g and g is equal to h then f is equal to h.
- E4. Circuits can be equal if their sizes are not definitionally the same. For example in the property $f \parallel \mathsf{id}\!\!\times\!\!\!\times\, 0 \approx f$, where $\mathsf{id}\!\!\times\!\!\!\times\, n$ is the identity circuit with size n. The left hand side is of size $f + 0$, but the right hand side has size f. From the equality we should be able to obtain proofs that the sizes do match.

4.2.1 First try

A straightforward definition might be:

$$\_\approx_1\_ : \forall\ \{i\ o\}\ (f\ g : \mathbb{C}\ i\ o) \to \mathsf{Set}$$
$$f \approx_1 g = \forall\ w \to [\![\ f\ ]\!]\ w \equiv [\![\ g\ ]\!]\ w$$

To prove that two circuits are equal we must build a function which takes some input vector w of length i, and returns a proof that both circuits give the same result when we simulate them with that input. With this definition we get properties E1-3, but not E4. Both circuits use the same variables for their sizes, input size i and output size o.

4.2.2 Semi-heterogeneous equality

To get property E4 we need seperate variables for the sizes of the two circuits. This separation gives us two problems:

- The input sizes don't match, so we can't pick an input vector which fits both.
- The output sizes don't match so the types of the result are different. In Agda, the propositional equality $\equiv$ is homogeneous so it can't be used to compare the results.

One of the solutions we tried is to coerce one of the circuits so the sizes line up using a function of type $\forall\ \{i_1\ i_2\ o_1\ o_2\} \to i_1 \equiv i_2 \to o_1 \equiv o_2 \to \mathbb{C}\ i_1\ o_1 \to \mathbb{C}\ i_2\ o_2$. This works reasonably well, but the asymmetry in the definition feels awkward. A prettier solution is to not use propositional equality on the vectors. The standard library provides a semi-heterogeneous vector equality $\approx_v$ where the lengths of the vectors don't have to be the same. From a proof that two vectors are equal in this way, $xs \approx_v ys$, we can obtain a proof that the lengths of the two vectors are equal. When you are lucky you can rewrite your goals with this proof, then the lengths of the vectors become equal and you can convert the semi-heterogenous equality to a propositional equality $xs \equiv ys$. Using the semi-heterogeneous vector equality we arrive at the following definition:

$$\_\approx\_ : \forall\ \{i_1\ o_1\ i_2\ o_2\}\ (f : \mathbb{C}\ i_1\ o_1)\ (g : \mathbb{C}\ i_2\ o_2) \to \mathsf{Set}$$
$$f \approx g = \forall\ \{w_1\ w_2\} \to (w_1 \approx_v w_2) \to [\![\ f\ ]\!]\ w_1 \approx_v [\![\ g\ ]\!]\ w_2$$

This gives us property E4 quite elegantly. Reflexivity (E1) and symmetry (E2) are still easy to prove.

4.2.3 Empty domains

With this definition $\cong$ we got stuck in finding a proof for transitivity (E3). It took a while to find the root of this problem.

The essence is that if we have two circuits f and g with truly different sizes (contrary to definitionally different but equal sizes like $n + 0$ and n), the type $w_1 \approx_v w_2$ becomes empty. In that case we can always create a function which has the right return type $[\![f]\!] w_1 \approx_v [\![g]\!] w_2$, because it will never actually have to return anything. Here is a proof that the identity circuit of size 0 is equal to the identity circuit of size 1 (it should not be):

```
≅-inconsistent : id⤫ 0 ≅ id⤫ 1
≅-inconsistent ()
```

To solve this problem, we need to make sure that the domain can never be empty. The standard solution to do such a thing is to combine the function with a witness; some value of the correct type which is not really used, but it shows that the type is not empty. In our situation it is more fitting to require a proof that $i_1 \equiv i_2$. This is often easier to create and is sufficient.

This results in the final definition, we wrap the previous definition $\cong$ in a data type with a proof for $i_1 \equiv i_2$.

```
data _≅_ {i₁ o₁ i₂ o₂ : ℕ} : (f : ℂ i₁ o₁) (g : ℂ i₂ o₂) → Set where
  Mk≅ : {f : ℂ i₁ o₁} {g : ℂ i₂ o₂} (pi : i₁ ≡ i₂) (f ≈ g : f ≅ g) → f ≅ g
```

Now we have all the desired properties E1 to E4.

Note: We do not have to pass $o_1 \equiv o_2$, but we can create that proof in a roundabout way by creating a dummy input and passing it to the function. The resulting vector equality implies that the output sizes must be equal.

4.3 Compositionality

The behavior of a circuit is determined by the behavior of the parts. We can substitute any part of a circuit by another part with the same behavior, while the whole circuit keeps the same behavior.

4.3.1 Congruence under X

This notion was inmplemented by defining separate X-cong-functions for a lot of functions which combine or return circuits. A function X-cong tells us that X *preserves* behavioral equality.

```
_⟫-cong_ : ∀ {i₁ m₁ o₁} {c₁ : ℂ i₁ m₁} {d₁ : ℂ m₁ o₁} →
  ∀ {i₂ m₂ o₂} {c₂ : ℂ i₂ m₂} {d₂ : ℂ m₂ o₂} →
  c₁ ≅ c₂ → d₁ ≅ d₂ → c₁ ⟫ d₁ ≅ c₂ ⟫ d₂
_∥-cong_ : ∀ {i₁ o₁ j₁ p₁} {c₁ : ℂ i₁ o₁} {d₁ : ℂ j₁ p₁} →
  ∀ {i₂ o₂ j₂ p₂} {c₂ : ℂ i₂ o₂} {d₂ : ℂ j₂ p₂} →
  c₁ ≅ c₂ → d₁ ≅ d₂ → c₁ ∥ d₁ ≅ c₂ ∥ d₂
```

This pattern is also useful if you want to pass other kinds of equivalences like propositional equality or vector equality:

```
scan-cong : ∀ {m n} → scan m ≅ scan n
_⟿-cong_ : ∀ {m n} {f : ℂ m m} {g : ℂ n n}
```

$$\{as : \mathsf{Vec}\ \mathbb{N}\ m\}\ \{bs : \mathsf{Vec}\ \mathbb{N}\ n\} \to$$
$$f \approx g \to as \approx_{\mathsf{v}} bs \to f \prec as \approx g \prec bs$$

4.3.2 Contexts

An earlier approach was inspired by $\mathsf{cong} : \forall\ \{A\ B : \mathsf{Set}\} \to (f : A \to B) \to \{x\ y : A\} \to x \equiv y \to f\ x \equiv f\ y$.

To translate this function to circuits, we use contexts instead of the function f. A context is similar to a circuit, but one single hole in it (basically half of a Huet zipper). A circuit in the hole can be plugged with $\mathsf{plugCxt} : \forall\ \{i_x\ o_x\ i\ o\} \to \mathsf{Cxt}\ i_x\ o_x\ i\ o \to \mathbb{C}\ i_x\ o_x \to \mathbb{C}\ i\ o$. Now we can define a function similar to cong:

$$\approx\text{-cong} : \forall\ \{i_x\ o_x\ i\ o\} \to (cxt : \mathsf{Cxt}\ i_x\ o_x\ i\ o) \to \{f\ g : \mathbb{C}\ i_x\ o_x\} \to$$
$$f \approx g \to \mathsf{plugCxt}\ cxt\ f \approx \mathsf{plugCxt}\ cxt\ g$$

A disadvantage of this particular definition is that the sizes of f and g must be definitionally equal. We want to be able to give f and g definitionally different sizes. A context only works on circuits of one particular size, so we need a context for both f and g, and both must do the same thing (they are equal in a certain sense). Then we can plug one of the contexts with f and the other with g. If our context equality is $\approx_{\times}$, we can define:

$$\approx\text{-cong}_2 : \forall\ \{i_x^1\ o_x^1\ i^1\ o^1\ i_x^2\ o_x^2\ i^2\ o^2\}$$
$$\{cxt^1 : \mathsf{Cxt}\ i_x^1\ o_x^1\ i^1\ o^1\}\ \{cxt^2 : \mathsf{Cxt}\ i_x^2\ o_x^2\ i^2\ o^2\}$$
$$\{f : \mathbb{C}\ i_x^1\ o_x^1\}\ \{g : \mathbb{C}\ i_x^2\ o_x^2\} \to$$
$$cxt^1 \approx_{\times} cxt^2 \to f \approx g \to \mathsf{plugCxt}\ cxt^1\ f \approx \mathsf{plugCxt}\ cxt^2\ g$$

With some convenient constructors for the context equality $\approx_{\times}$ we can use it like this:

$$\approx\text{-cong}_2\text{-example} : \forall\ \{m\ n\}\ (f : \mathbb{C}\ m\ m)\ (g : \mathbb{C}\ n\ n) \to$$
$$((f \fatsemi \mathsf{id} \boxtimes m) \parallel g) \approx (f \parallel g)$$
$$\approx\text{-cong}_2\text{-example}\ f\ g = \approx\text{-cong}_2\ (\bullet \bullet\parallel \approx\text{-refl})\ (\fatsemi\text{-right-identity}\ \_)$$

4.3.3 Comparison of X-cong and contexts

Compare the terms using $\approx$-cong and X-cong:

$$\approx\text{-cong}_2\ (\bullet \bullet\parallel \approx\text{-refl})\ (\fatsemi\text{-right-identity}\ \_)$$

$$\fatsemi\text{-right-identity}\ \_ \parallel\text{-cong}\ \approx\text{-refl}$$

The X-cong-functions have some advantages:

- They follow the actual structure of the circuits more closely, which make them easier to use.
- It is possible to substitute both sides of an operator at the same time. This is not only shorter, but can circumvent problems when the sizes of the circuits are changed.
- They extend well to other user-made constructions, while the contexts are only defined on the native constructors of $\mathbb{C}$.

4.4 Basic properties

We proved a lot of basic structural properties mostly based on the structural laws of Hinze's scan algebra. These include left identity, right identity and associativity of ⟫ and ∥, distributivity of ∥ over ⟫ and merging of parallel id⤨'s.

4.5 Plugs

At the start of the project, plugs in Π-Ware were defined using rewiring function of type fin o → fin i. For any output wire, this function tells you which input wire it is connected to. The evaluation of a plug was basically ⟦ Plug p ⟧ w = tabulate (λ n → lookup (p n) w). Doing proofs with this higher order representation can get quite complicated, because the function is used as a higher order argument within the tabulate and lookup.

Later the plugs changed to a first-order representation Vec (Fin i) o. This makes them a bit easier to reason about, but it does not make the evaluation simpler - actually, it adds a lookup.

4.5.1 Plugs by morphism

To alleviate this problem, we use Vec-Morphism:

```
record Vec-Morphism (i : ℕ) (o : ℕ) : Set₁ where
  field
    op : {X : Set} → Vec X i → Vec X o
    op-map-commute : {X Y : Set} (f : X → Y) → -- Free property
      (w : Vec X i) → (op ∘ map f) w ≡ (map f ∘ op) w
```

And we can build plugs, in the current system, using these morphisms:

```
plug-by-morphism : ∀ {i o} → Vec-Morphism i o → ℂ i o
plug-by-morphism M = Plug (Vec-Morphism.op M (allFin _))
```

The fact that op is *parametrically polymorphic* guarantees that it commutes with map, as is shown in Theorems for free! by Philip Wadler. However, we can not get this theorem for free in Agda, so we need op-map-commute to be in the record. Using op-map-commute, we can prove a simple but powerful property:

```
plug-by-morphism-⟦⟧ : ∀ {i o} (M : Vec-Morphism i o) (w : W i) →
  ⟦ plug-by-morphism M ⟧ w ≡ Vec-Morphism.op M w
```

We just got rid of all the tabulate's and lookup's! This makes a lot of proofs simpler.

5 Patterns

When we use a function to generate some kind of circuit, we call it a pattern. Patterns usually have a few things in common, taking the fan pattern as an example:

- The pattern function: fan : ∀ n → ℂ n n
- An implementation: fan-impl : ∀ n → ℂ n n
- A specification: fan-spec : ∀ n → W n → W n
- Proof that the pattern function matches the implementation: reveal-fan : ∀ n → fan n ≋ fan-impl n

- Proof that the implementation matches the specification: fan-to-spec : ∀ n → (w : W n) → ⟦ fan n ⟧ w ≡ fan-spec n w

The reason for separating the pattern function and implementation is that we often mark the pattern function as abstract, so it wont reduce when we don't want it too. This is a good way to hide the complexity of patterns. We often do not need to know the actual implementation to be able to use them, so we can start viewing the problem at hand from a higher level of abstraction. In some cases it also leads to better performance.

For future work, it might be useful to make this structure more formal. During simulation this might be particularly useful, because you could choose to run the specification instead of simulating the implementation circuit. This might make simulation of very large circuits feasible.

5.1 Heterogeneous sequencing

To put two circuits in sequence with the native _⟫_ constructor, the sizes must be the same. To be able to put two circuits in sequence where the output size of the first is not the same, but known to be equal, to the input size of the second, the following *heterogeneous sequencing* pattern is defined:

```
_⟫[_]_ : ∀ {i m n o} (f : ℂ i m) (p : m ≡ n) (g : ℂ n o) → ℂ i o
```

Together with a variant where the proof is implicit:

```
_⟫[]_ : ∀ {i m n o} (f : ℂ i m) {p : m ≡ n} (g : ℂ n o) → ℂ i o
```

It is used like this:

```
hetseq-proof : ∀ {i m o} (f : ℂ i m) (g : ℂ m o) →
  (f ∥ id⨯ 0) ⟫ ((g ⟫ id⨯ o) ∥ id⨯ 0) ≋ f ⟫ g
hetseq-proof f g = begin
  (f ∥ id⨯ 0) ⟫ ((g ⟫ id⨯ _) ∥ id⨯ 0)
    ≋⟨ (∥-right-identity f) ⟫[-cong ≋-refl ⟩
  f ⟫[] ((g ⟫ id⨯ _) ∥ id⨯ 0)
    ≋⟨ ≋-refl ⟫[]-cong (∥-right-identity _) ⟩
  f ⟫[] (g ⟫ id⨯ _)
    ≋⟨ ≋-refl ⟫]-cong (⟫-right-identity g) ⟩
  f ⟫ g
    ∎
```

This is a great example where marking the pattern function as abstract is very useful: When m and n are the same, Agda reduces the implementation so far that it does not recognize it as a _⟫[]_ anymore. Without marking the pattern function as abstract, it might happen that a function which can be applied to f ⟫[p] g can not be applied to f ⟫[refl] g.

5.1.1 Congruence under ⟫[]-cong

Something similar to ⟫-cong can be defined for heterogeneous sequencing ⟫[]:

```
_⟫[]-cong_ : ∀ {i₁ m₁ n₁ o₁} {f₁ : ℂ i₁ m₁} {p₁ : m₁ ≡ n₁} {g₁ : ℂ n₁ o₁} →
  ∀ {i₂ m₂ n₂ o₂} {f₂ : ℂ i₂ m₂} {g₂ : ℂ n₂ o₂} →
  (ff : f₁ ≋ f₂) → (gg : g₁ ≋ g₂) →
  f₁ ⟫[ p₁ ] g₁ ≋ _⟫[]_ f₂ {⟫[]-cong-proof p₁ ff gg} g₂
```

Also, it can be useful to quickly convert from ⟫ to ⟫[], and back if the sizes allow it. Besides some simple conversion functions we can define two variants of ⟫[]-cong:

```
_⟫[-cong_ : ∀ {i₁ m₁ o₁} {f₁ : ℂ i₁ m₁} {g₁ : ℂ m₁ o₁} →
  ∀ {i₂ m₂ n₂ o₂} {f₂ : ℂ i₂ m₂} {g₂ : ℂ n₂ o₂} →
  (f ≈ f : f₁ ≋ f₂) → (g ≈ g : g₁ ≋ g₂) →
  f₁ ⟫ g₁ ≋ _⟫[]_ f₂ {⟫[-cong-proof f ≈ f g ≈ g} g₂
_⟫]-cong_ : ∀ {i₁ m₁ n₁ o₁} {f₁ : ℂ i₁ m₁} {p₁ : m₁ ≡ n₁} {g₁ : ℂ n₁ o₁} →
  ∀ {i₂ m₂ o₂} {f₂ : ℂ i₂ m₂} {g₂ : ℂ m₂ o₂} →
  (f ≈ f : f₁ ≋ f₂) → (g ≈ g : g₁ ≋ g₂) →
  f₁ ⟫[ p₁ ] g₁ ≋ f₂ ⟫ g₂
```

Notice that the convention here is that the proofs for the left hand side are given by the user, and the proofs on the right hand side are provided by the function. This convention makes it possible to automatically create the equality proofs in a lot of cases, for example:

```
hetseq-proof : ∀ {i m o} (f : ℂ i m) (g : ℂ m o) →
  (f ∥ id⤭ 0) ⟫ ((g ⟫ id⤭ o) ∥ id⤭ 0) ≋ f ⟫ g
hetseq-proof f g = begin
  (f ∥ id⤭ 0) ⟫ ((g ⟫ id⤭ _) ∥ id⤭ 0)
    ≋⟨ (∥-right-identity f) ⟫[-cong ≋-refl ⟩
  f ⟫[] ((g ⟫ id⤭ _) ∥ id⤭ 0)
    ≋⟨ ≋-refl ⟫[]-cong (∥-right-identity _) ⟩
  f ⟫[] (g ⟫ id⤭ _)
    ≋⟨ ≋-refl ⟫]-cong (⟫-right-identity g) ⟩
  f ⟫ g
    ∎
```

This convention is also followed by many of the properties regarding heterogeneous sequencing. The drawback of this is that it breaks when the proof is flipped with ≋-sym, resulting in forwards and backwards variants of some of the properties.

5.2 Stretching

The stretch operator ⇜ is straight from Hinze's paper and can be used to stretch a circuit by a list of widths a_1, a_2, ..., a_n where n is both the input and output size of the circuit. From the standpoint of the resulting circuit, its inputs and outputs are grouped such that every group i has length a_i. For each ith group, the first in-/output is connected to the ith wire of the inner circuit. We can implement it by first reordering the wires, passing the first n of them through the inner circuit and then reversing the original reordering:

```
_⇜_ : ∀ {n} → ℂ n n → (as : Vec ℕ n) → ℂ (size 1 as) (size 1 as)
c ⇜ as = in-⤭ as
  ⟫ c ∥ id⤭ (size 0 as)
  ⟫ out-⤭ as
```

Where size m as = sum (map (_+_ m) as).

The hard part is to come up with a definition for in-⤭ and out-⤭ such that it is easy to do proofs about them.

5.2.1 Permutations

Initially we wanted to work with the fact that both ⟦ in-⤨ ⟧ and ⟦ out-⤨ ⟧ must return a permutation of their input.

We formalized this using Lehmer codes, a way of uniquely encoding permutations. We did proofs about these Lehmer codes about their composibility, inversion, identities and application to vectors.

This method works fine if you want, for instance, the property that $\forall\ w \rightarrow$ ⟦ in-⤨ *as* ⟫ out-⤨ *as* ⟧ $w \equiv w$. It breaks down when you want to reason about where certain elements go and what is done to them, so we decided not to use it. We replaced it by plugs by morphism, as described in section 4.5.1.

5.2.2 MinGroups

The current version uses an explicit representation of groups. The data type MinGroups $A\ m\ as$, is basically a vector of vectors. If as is a list $a_1,\ a_2,\ ...,\ a_n$, the ith vector is of size $m + a_i$. We can convert a vector of the right size to a MinGroups and back, essentially splitting a vector into groups. We define a function extract-map : $\forall\ \{A\ i\ n\}$ $\{as : \text{Vec}\ \mathbb{N}\ n\} \rightarrow$
(Vec $A\ n \rightarrow$ Vec $A\ n) \rightarrow$ MinGroups A (suc i) $as \rightarrow$ MinGroups A (suc i) as, which performs the following steps:

1. Take the first element of each group and put them in a vector.
2. Apply a function to this vector.
3. Insert the elements of the resulting vector as the first element of each group.

This extract-map has some nice properties like composability and that it commutes with concatenation. There are several properties about MinGroups and extract-map which are obvious but still hard to prove, a few of those are simply postulated.

5.2.3 More stretching

We also define a ⤜ operator, which is the same as ⤙ except that it works on the last element of each group instead of the first. Many proofs about stretches are defined such that they work on both directions. There are also derived stretch operators ⤞ and ⤝, which use a list of circuits instead of the list of widths. They inherit most of the properties of the basic stretch operators.

5.3 Fans

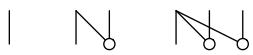

Figure 3: Fans of sizes 1, 2 and 3

In Ralf Hinze's scan algebra, fans are native constructions. Given some inputs $x_1,\ x_2,\ \ldots,\ x_n$, the circuit fan n has outputs $x_1,\ (x_1 \oplus x_2),\ (x_1 \oplus x_3),\ \ldots,\ (x_1 \oplus x_n)$ (figure 3).

In Π-Ware, fans have to be defined using smaller circuits. We can build a plusℂ circuit with two inputs and one output which applies the ⊕ operator, using the Gate constructor of ℂ.

Figure 4: Structure of fans. Left: first approach, right: current approach

At first, we defined fans as a Plug followed by a bunch of plusℂ's in parallel (figure 4, left). The definition itself is quite easy and elegant, but it was very hard to prove things about because there is no structure to recurse on.

In the current version we built them iteratively, using smaller fans to make the bigger ones (figure 4, right).

Note: Maybe fans can also be implemented by using an indexed variant of the Gate constructor, but this was not researched thoroughly.

5.4 Scans

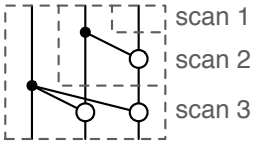

Figure 5: Structure of naive scans

When we have a scan circuit of size n, we can create a scan circuit of size $n + 1$ by using scan-succ.

```
scan-succ : ∀ {n} → ℂ n n → ℂ (suc n) (suc n)
scan-succ {n} f = id⤨ 1 ∥ f ⟫ fan (suc n)
```

From this definition we can derive a very simple method to create scan circuits of any size, as in figure 5:

```
scan : ∀ n → ℂ n n
scan zero = id⤨ 0
scan (suc n) = scan-succ (scan n)
```

This is the least efficient scan possible, both in depth and the number of operation nodes. It is, however, quite easy to work with.

Now we can make the statement ``*something* is a scan circuit" more formal: Given a circuit $f : ℂ\ n\ n$, we can say that f *is a scan circuit* if we have a proof that $f \approx$ scan n.

5.4.1 Scan combinators

Two of the scan combinators defined by Hinze are the vertical scan combinator ⍁ and horizontal scan combinator ⍃:

```
_⍁_ : ∀ {m n o p}
  (f : ℂ m (suc o)) (g : ℂ (suc n) p) → ℂ (m + n) (o + p)
_⍁_ {n = n} {o} f g = f ∥ id⤨ n
  ⟫[ sym (+-suc o n) ] id⤨ o ∥ g

_⍃_ : ∀ {m n}
  (f : ℂ (suc m) (suc m)) (g : ℂ n n) → ℂ (suc m + n) (m + suc n)
_⍃_ {m = m} {n} f g = f ∥ g
  ⟫[ sym (+-suc m n) ] id⤨ m ∥ fan (suc n)
```

We were able to prove for both combinators that if we use them to combine two scans, we get another scan:

```
⍁-combinator : ∀ m n → scan (suc m) ⍁ scan (suc n) ≋ scan (m + suc n)
⍃-combinator : ∀ m n → scan (suc m) ⍃ scan n ≋ scan (suc m + n)
```

5.4.2 Serial scans are scans

With all these tools we can do proofs about real scans. Take for instance the *serial scan*. This is a scan where all operations are done in sequence, see figure 2. It is the normal way to implement a scan without using parallelism.

Here we use id⤨ and _⟫[_]_ to make the size of serial-scan (suc n) ⍃ id⤨ 1 line up with the type.

```
serial-scan : ∀ n → ℂ n n
serial-scan 0 = id⤨ 0
serial-scan 1 = id⤨ 1
serial-scan (suc (suc n)) = id⤨ (2 + n)
  ⟫[ cong suc (+-comm 1 n) ] serial-scan (suc n) ⍃ id⤨ 1
  ⟫[ +-comm n 2 ] id⤨ (2 + n)
```

Like many scans, serial scans are built from smaller scans using scan combinators.

To prove that a serial scan is indeed a scan circuit (is equal to scan n), we simply have to prove that the parts are scans. In this case that is done by induction and with the fact that id⤨ 1 ≋ scan 1. When we have proved that the components are scans, we know that the whole circuit is a scan too. Here is a full proof:

```
serial-scan-is-scan : ∀ n → serial-scan n ≋ scan n
serial-scan-is-scan 0 = ≋-refl
serial-scan-is-scan 1 = id1=scan1
serial-scan-is-scan (suc (suc n)) = begin
  serial-scan (2 + n)
    ≋⟨⟩ -- Expand definition of serial-scan
  id⤨ _ ⟫[] (serial-scan (suc n) ⍃ id⤨ 1) ⟫[] id⤨ _
    ≋⟨ ⟫[]-right-identity _ ⟩ -- f ⟫[] id⤨ ≋ f
  id⤨ _ ⟫[] (serial-scan (suc n) ⍃ id⤨ 1)
    ≋⟨ ⟫[]-left-identity _ ⟩ -- id⤨ ⟫[] f ≋ f
  serial-scan (suc n) ⍃ id⤨ 1
```

```
    ≋⟨ (serial-scan-is-scan (suc n)) -- Induction
      □-cong id1=scan1
    ⟩
  scan (suc n) □ scan 1
    ≋⟨ □-combinator n 1 ⟩ -- □ is a scan combinator
  scan (suc n + 1)
    ≋⟨ scan-cong (cong suc (+-comm n 1)) ⟩ -- suc n + 1 ≡ 2 + n
  scan (2 + n)
    ∎
```

6 Future work

One example where the system does not work as easy as we would like, is in proving that `scan-succ (f ▽ scan-succ g ⟫[ p ] id⤫) ≋ scan-succ f ▽ scan-succ g`. Where Hinze uses 9 lines, we need 110 (!) lines. This is mainly because we need to prove a lot of boring things like $a \gg b \gg (c \gg d) \approxeq a \gg (b \gg c) \gg d$. It would be nice to have some kind of automated way of building proofs similar to the semiringsolver in the standard library. This might be done by normalizing circuits with `⟫`, `∥`, `id⤫` and `⟫[]` to some normal form.

Some things are still postulated, and should be proven formally. Most of these are quite 'obviously' true, or are a free theorem (see section 4.5.1.

7 Conclusion

We have shown that we can prove properties about whole families of circuits. These proofs can be written quite intuitively with equational reasoning. Many proofs follow exactly the same structure as Hinze's proofs.

In the course of this project, we have built several patterns (section 5) on top of Π-Ware. We have shown that it is possible to work with a high-level pattern like scan, without really noticing that they are actually composed of other things like fans and plugs.

Besides some left-over postulates, we have formally proven that certain circuits are scans, and that scans can be combined to create other scans.

References

[1] Ralf Hinze. An algebra of scans. In Dexter Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186--210. Springer Berlin Heidelberg, 2004.

[2] Conor Thomas McBride. How to keep your neighbours in order. *SIGPLAN Not.*, 49(9):297--309, August 2014.

[3] João Paulo Pizani Flor. Π-Ware: An embedded hardware description language using dependent types. Master's thesis, Utrecht University, 2014.