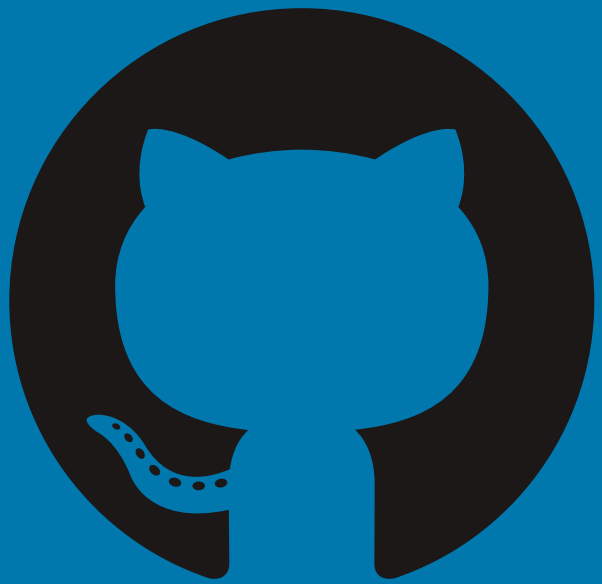
On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits

Craig Ramsay, University of Strathclyde, Scotland

June 25, 2021



github.com/cramsay/
idris_dsp_circuits

But first, what's Type Checking?

We're used to type systems catching simple bugs in code which is syntactically correct...

But first, what's Type Checking?

We're used to type systems catching simple bugs in code which is syntactically correct...

```
char f(int x) {
  ...
  return y;
}
```

And Dependent Types...

Let us perform computations during type checking

e.g. the ***type*** of an output can *depend* on the ***value*** of an input

And Dependent Types...

Let us perform computations during type checking

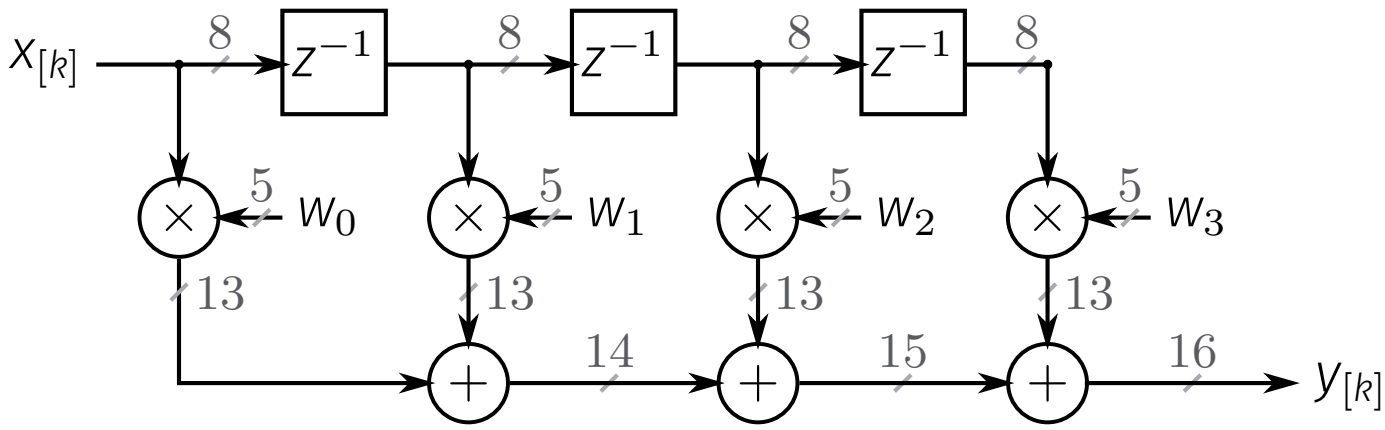
e.g. the ***type*** of an output can *depend* on the ***value*** of an input

...or *prove* certain properties of a function at compile time

Applications for generating DSP circuits?

First Steps: FIR filtering...

Worst-case growth

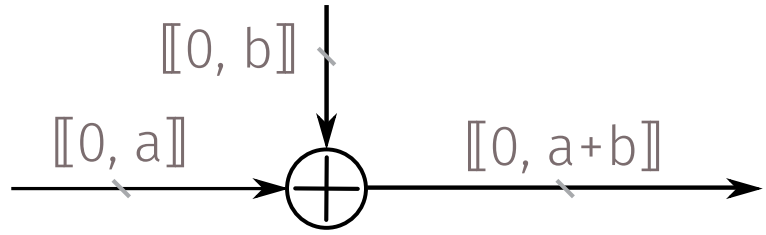


First Steps: FIR filtering...

Track ranges with `Bounded a`
(natural number in range $[\![0, a]\!]$)

First Steps: FIR filtering...

Track ranges with `Bounded a`
(natural number in range $[\![0, a]\!]$)

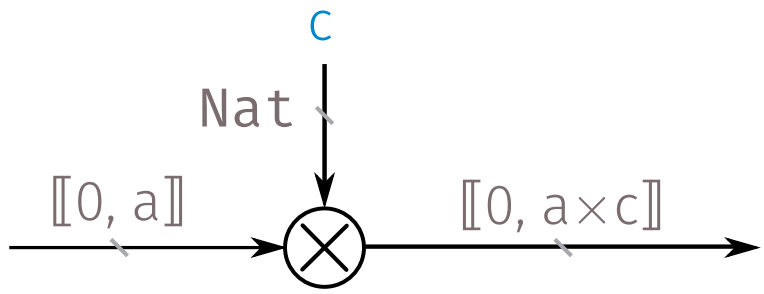



```
add : Bounded a -> Bounded b ->
      Bounded (a+b)

add (B x) (B y) = B (x+y)
```

First Steps: FIR filtering...

Track ranges with `Bounded a`
(natural number in range $[\![0, a]\!]$)




```
mulConst : Bounded a -> (c: Nat) ->
           Bounded (a*c)

mulConst (B x) c = B (x*c)
```

First Steps: FIR filtering...

What should the output range be in general?

$$y_{[k]} = \sum_{i=0}^{j-1} w_i \cdot x_{[k-i]}$$

First Steps: FIR filtering...

What should the output range be in general?

$$y_{[k]} = \sum_{i=0}^{j-1} w_i \cdot x_{[k-i]}$$

$$|y_{[k]}| = |x_{[k]}| \sum_{i=0}^{j-1} w_i$$

First Steps: FIR filtering...

Translating our equation to a useful type:

$$|y_{[k]}| = |x_{[k]}| \sum_{i=0}^{j-1} w_i$$

First Steps: FIR filtering...

Translating our equation to a useful type:

$$|y_{[k]}| = |x_{[k]}| \sum_{i=0}^{j-1} w_i$$

```
dotProd : (ws : Vect j Nat)     ->
          Vect j (Bounded n)    ->
          Bounded (n * sum ws)

dotProd ws xs = ...?
```

First Steps: FIR filtering...

Translating our equation to a useful type:

$$|y_{[k]}| = |x_{[k]}| \sum_{i=0}^{j-1} w_i$$

```
dotProd : (ws : Vect j Nat)     ->
          Vect j (Bounded n)    ->
          Bounded (n * sum ws)

dotProd {j=Z}          _          _          = zeros
dotProd {j=S l} {n} (w :: ws) (x :: xs) =
  let y = add (mulConst x w) (dotProd ws xs)
  in  rewrite dotProdDistrib n w l ws in y
```

What do we gain from this?

```
fir : (ws : Vect j Nat)  ->
      Stream (Bounded n) ->
      Stream (Bounded (n * sum ws))
```

- Expressed our intent (for wordlengths) precisely

What do we gain from this?

```
fir : (ws : Vect j Nat)  ->
      Stream (Bounded n) ->
      Stream (Bounded (n * sum ws))
```

- Expressed our intent (for wordlengths) precisely
  - A contract between `fir`'s dev and its user...
  - Mechanically checked by the type checker

What do we gain from this?

```
fir : (ws : Vect j Nat)  ->
      Stream (Bounded n) ->
      Stream (Bounded (n * sum ws))
```

- Expressed our intent (for wordlengths) precisely
  - A contract between `fir`'s dev and its user…
    - Mechanically checked by the type checker
  - Ensuring wordlength is theoretical minimum
  - For *all parameter sets*, not just one instance

...And what happens when things go wrong?

- If the output range is incorrect for *any set of parameters*, the whole function won't compile (type error)

...And what happens when things go wrong?

- If the output range is incorrect for *any set of parameters*, the whole function won't compile (type error)
  - ...Probably indicates arithmetic mistake or "off-by-one"

...And what happens when things go wrong?

- If the output range is incorrect for *any set of parameters*, the whole function won't compile (type error)

- ...Probably indicates arithmetic mistake or "off-by-one"

- Might always give correct wordlength but via a different expression (not `n * sum ws`).
  Dev must prove equivalence to type checker

...And what happens when things go wrong?

- If the output range is incorrect for *any set of parameters*, the whole function won't compile (type error)

- ...Probably indicates arithmetic mistake or "off-by-one"

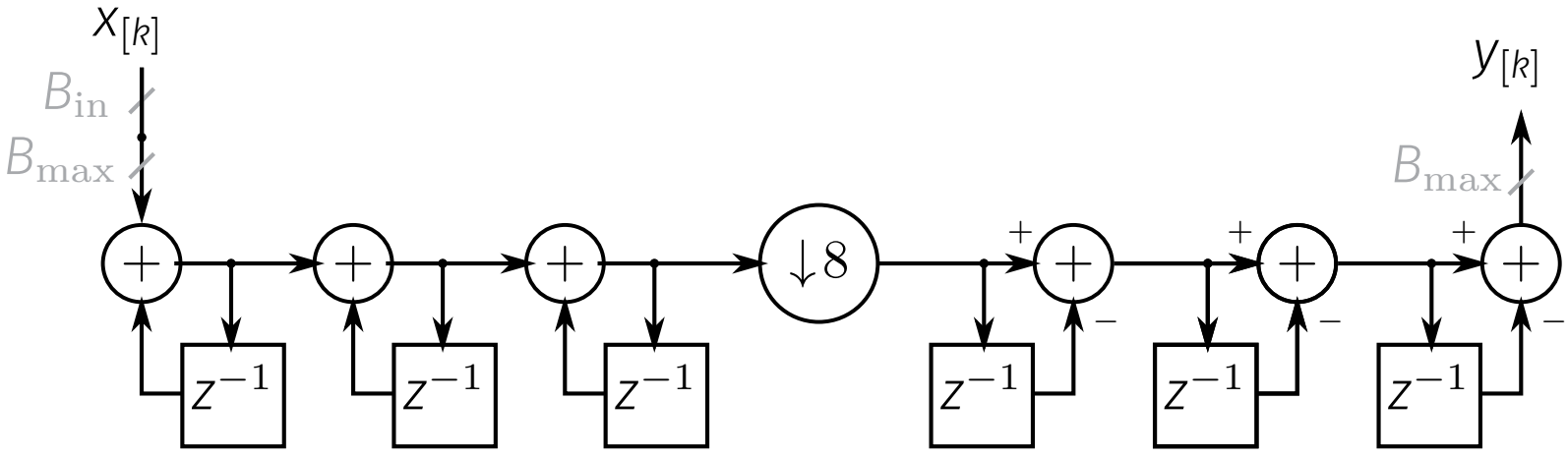
- Might always give correct wordlength but via a different expression (not `n * sum ws`).
Dev must prove equivalence to type checker

- Bad faith implementation would just resize...

We could encode arithmetic meaning with same tools, not just wordlengths.

Going Further: CIC pruning...

Without Hogenauer pruning
($R = 8$, $N = 3$ and $M = 1$)



Going Further: CIC pruning...

$$B_j = \left\lfloor -\log_2 F_j + \log_2 \sigma_{T_{2N+1}} + \frac{1}{2}\log_2 \frac{6}{N} \right\rfloor$$

where F_j is the variance error for the j^{th} stage, $\sigma_{T_{2N+1}}$ is the total variance at the output due to truncation, N is the number of stages.

Going Further: CIC pruning...

$$B_j = \left\lfloor -\log_2 F_j + \log_2 \sigma_{T_{2N+1}} + \frac{1}{2}\log_2 \frac{6}{N} \right\rfloor$$

Note that the first two terms have complicated definitions of their own, including cases, sums, exponentials, and binomial coefficients.

Going Further: CIC pruning...

```
parameters (r, n, m, b_in, b_out : Nat)

  cicStage : (j : Nat) ->
             Stream (Unsigned (hogenauer  j    )) ->
             Stream (Unsigned (hogenauer (j+1)))

  cicStage = ...?
```

Application Speculation

Application Speculation

- These examples have type-level wordlengths, but could be...

[1] E. Brady et al, “Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types”

Application Speculation

- These examples have type-level wordlengths, but could be...

- Type-level pipeline depth, arithmetic intent[1], or maybe even full *z*-transform

[1] E. Brady et al, "Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types"

Application Speculation

- These examples have type-level wordlengths, but could be...
- Type-level pipeline depth, arithmetic intent[1], or maybe even full *z*-transform
- Proving equivalence between whole circuit families...

(Enhances peer-reviewed proofs for optimisations — the implementation is computer-checked too)

[1] E. Brady et al, "Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types"

Application Speculation

- These examples have type-level wordlengths, but could be...
- Type-level pipeline depth, arithmetic intent[1], or maybe even full z-transform
- Proving equivalence between whole circuit families...
  (Enhances peer-reviewed proofs for optimisations — the implementation is computer-checked too)
- User can do all of this — we just need to support dependent types

[1] E. Brady et al, "Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types"

Implementation Speculation

Implementation Speculation

- We have been simulating circuits in Idris, but synthesis from high-level descriptions is promising.
  Inspirations including:

Implementation Speculation

- We have been simulating circuits in Idris, but synthesis from high-level descriptions is promising.
  Inspirations including:

  - CλaSH — Compiler for Haskell → circuits
    (Extremely good functional HDL *without* dependent types)

Implementation Speculation

- We have been simulating circuits in Idris, but synthesis from high-level descriptions is promising.
Inspirations including:

  - CλaSH — Compiler for Haskell → circuits
(Extremely good functional HDL *without* dependent types)

  - Π-ware — EDSL for circuits with dependent types
(Interesting project with extremely low-level circuit descriptions)

Implementation Speculation

- We have been simulating circuits in Idris, but synthesis from high-level descriptions is promising.
  Inspirations including:

  - CλaSH — Compiler for Haskell → circuits
    (Extremely good functional HDL *without* dependent types)

  - Π-ware — EDSL for circuits with dependent types
    (Interesting project with extremely low-level circuit descriptions)

- Proto-Quipper-D — Dependently typed tool for quantum circuits
  (Great tool from a different domain with similar challenges)

Thanks! Q&A?