Functional Hardware Description with Dependent Types

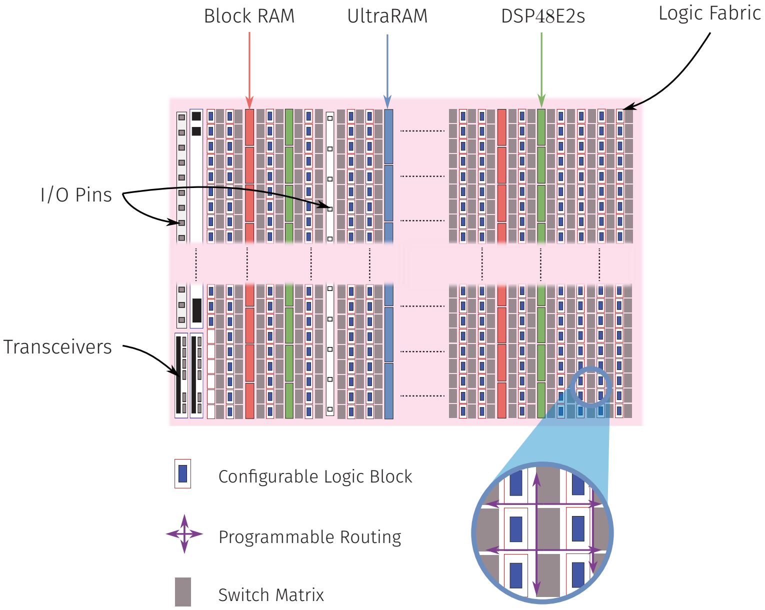
Craig Ramsay

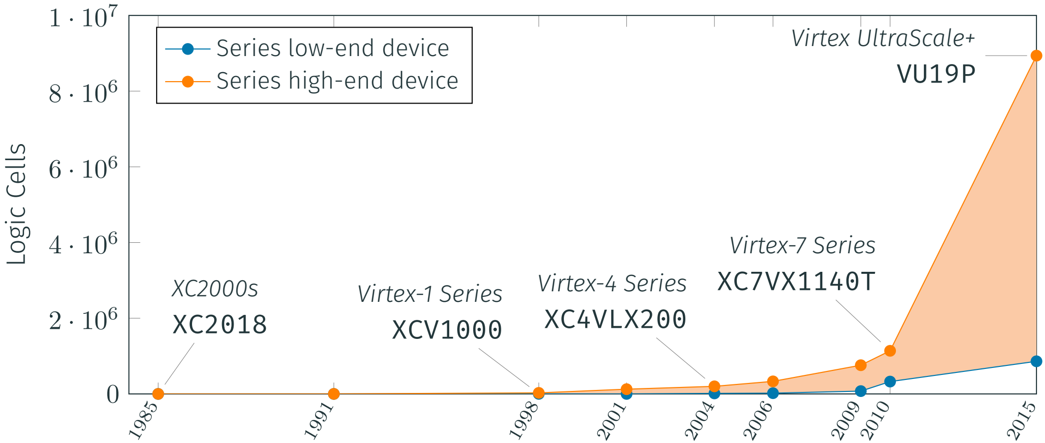
November 2023

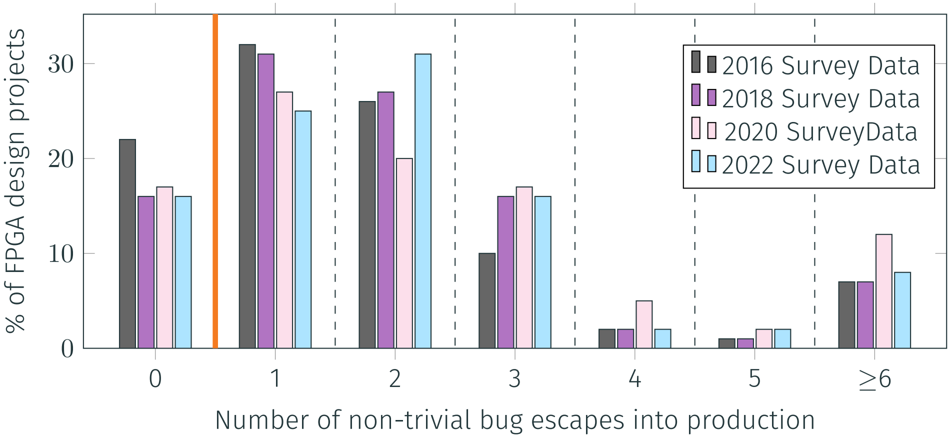
University of Strathclyde

```
"Y88888888888888888888888888888888888888888888888888888888Y"
  "Y8888888888888888888888888888888888888888888888888888Y"
   888                                                 888
   888   888                     888    d8b            888
   888   888                     888    Y8P            888
   888   888                     888                   888
   888   888888 .d88b.   8888b.  888888 888  .d88b.    888
   888   888   d88""88b     "88b 888    888 d8P  Y8b   888
   888   888   888  888 .d888888 888    888 88888888   888
   888   Y88b. Y88..88P 888  888 Y88b.  888 Y8b.       888
 .e888e   "Y888 "Y88P"  "Y888888  "Y888 888  "Y8888   e888e.
```

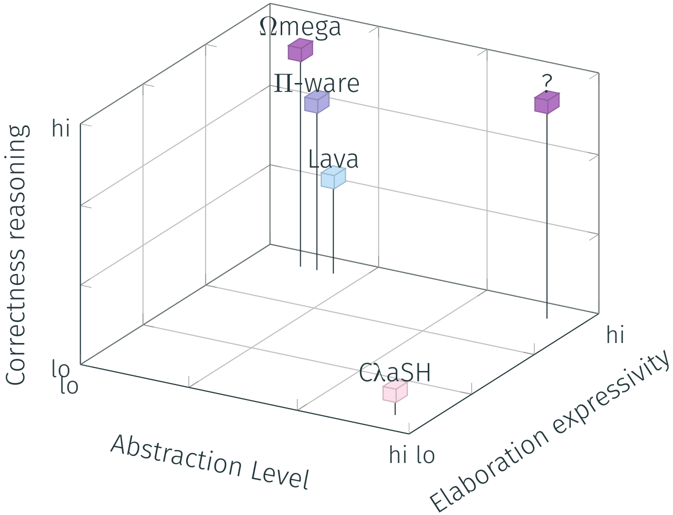
The *motivation* (the problem & novelty)

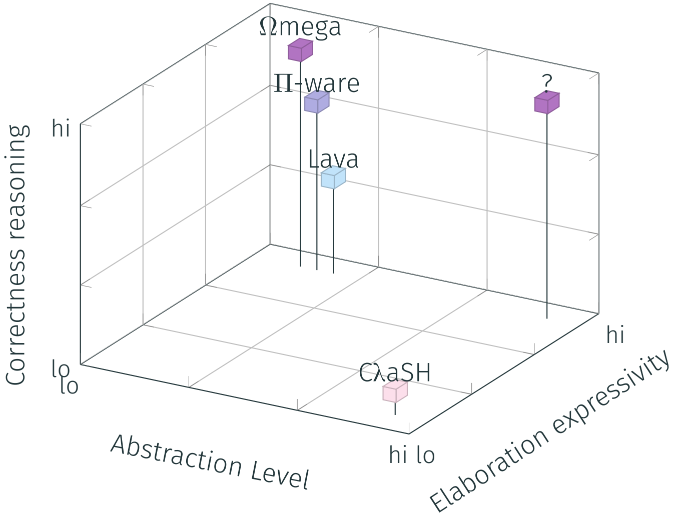

Block RAM
UltraRAM
DSP48E2s
Logic Fabric
I/O Pins
Transceivers
Configurable Logic Block
Programmable Routing
Switch Matrix


Logic Cells
$1 \cdot 10^7$
$8 \cdot 10^6$
$6 \cdot 10^6$
$4 \cdot 10^6$
$2 \cdot 10^6$
0
1985
1991
1998
2001
2004
2006
2009
2010
2015
Series low-end device
Series high-end device
XC2000s
XC2018
Virtex-1 Series
XCV1000
Virtex-4 Series
XC4VLX200
Virtex-7 Series
XC7VX1140T
Virtex UltraScale+
VU19P


% of FPGA design projects
0
10
20
30
0
1
2
3
4
5
≥6
Number of non-trivial bug escapes into production
2016 Survey Data
2018 Survey Data
2020 SurveyData
2022 Survey Data

	Paradigm	Typing Discipline	Abstraction Level	Hosting Style
				Traditional HDLs
VHDL	Mixed / Synchronous	Strong Typing	RTL	Stand-Alone
Verilog	Mixed / Synchronous	Weak Typing	RTL	Stand-Alone
SystemVerilog	Mixed / Synchronous	Strong Typing	RTL	Stand-Alone
				High-Level Synthesis Languages
Vivado HLS	Imperative	Strong Typing	Behavioural	Stand-Alone
				Functional HDLs
Lava	Functional	Stronger Typing + Hindley–Milner	Gate	Embedded (Haskell)
CλaSH	Functional	Stronger Typing + Hindley–Milner	RTL	Stand-Alone
Π-ware	Functional	Stronger Typing + Dependent Types	Gate	Embedded (Adga)
`toatie`	Functional	Stronger Typing + Dependent Types	RTL	Stand-Alone


Ωmega
Π-ware
?
Lava
CλaSH
Correctness reasoning
hi
lo
lo
Abstraction Level
hi
lo
hi
Elaboration expressivity



Chapter 4 contribution

A CλaSH case study of an application well-suited for EDSLs:

- Motivates first-class staging
- Motivates dependent types for ergonomics and verification
- Open source low-cost, high-speed, parallel filters for direct RF sampling

The *what*

(technical discussion)

We explore an HDL that can:

Represent circuits as plain functions
(needs a stand-alone *compiler*)

Ascribe meaning to synthesisable data types
(needs a language with dependent types)

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
          ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
              let rec = addU _ _ _ _ xbs ybs cin'
                  ans = UCons _ _ _ rec lsb
              in eqInd2 _ _ _
                        prfAddU c xn yn a b xsn ysn prf
                        (λh ⇒ Unsigned (S (S w)) h)
                        ans
```

Similar to Brady’s proposal in “Constructing correct circuits“, 2007.

But what about synthesis?

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

Goal is to safely reason about:

circuit runtime

vs

elaboration-time

vs

typechecking only?

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

Chapter 5 contribution

Further investigation of use cases for dependently typed HDLs, representing circuits as functions:

- Minimal type complexity — enjoy a single language for entire circuit lifetime
- Moderate type complexity — enjoy tracking and informing non-functional circuit properties at compile-time
- Full functional verification — Scales well for combinatorial DSP implementations , and shows promise for synchronous circuits.

Take a small dependently typed software language, TinyIdris, then layer our experimental features on top.

Our features:

Erasure

Distinguish typechecking time vs rest

Also applies to *data*

Staging

Distinguish elaboration vs circuit run-times

Synthesis

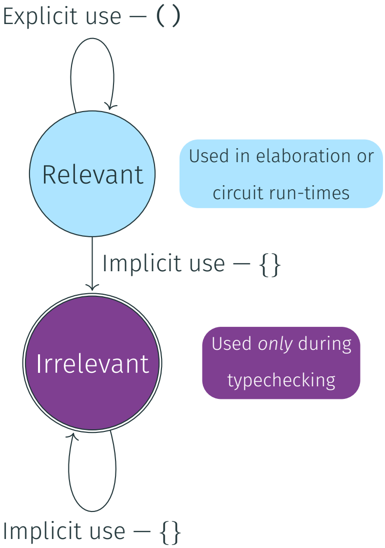
Derive bit representations for user types

Perform elaboration

Chapter 6 contribution

- `toatie`: an open source implementation for combinatorial circuits
- Phases of circuit lifetime are the challenge
- Two features used as software optimisation become necessary for an HDL
- Synthesis can lean on two existing parts of dependently typed compilers

Erasure


Explicit use — ( )
Relevant
Used in elaboration or circuit run-times
Implicit use — {}
Irrelevant
Used *only* during typechecking
Implicit use — {}

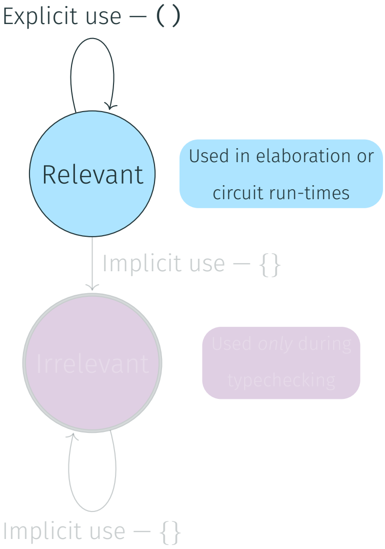
Discard terms only needed during typechecking

We use irrelevance (with ICC*) to direct erasure absolutely

Typechecker prevents path from *Irrelevant* back to *Relevant*

$$\frac{\Gamma \vdash (\Pi\{x : S\} \to T) : \mathsf{Type} \qquad \Gamma; \lambda\{x : S\} \vdash e : T \qquad x \notin \mathrm{FV}(\mathcal{E}[\![e]\!])}{\Gamma \vdash (\lambda\{x : S\}.\ e) : \Pi\{x : S\} \to T}$$

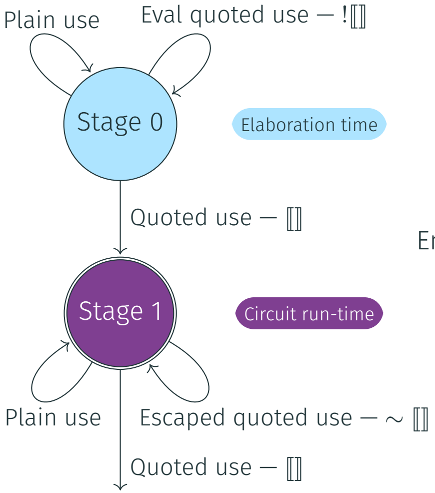
Erasure


Explicit use — ( )
Relevant
Used in elaboration or circuit run-times
Implicit use — {}
Irrelevant
Used *only* during typechecking
Implicit use — {}

After typechecking, ICC*'s extraction performs erasure

$$
\begin{aligned}
&\mathcal{E}[\![x]\!] &&= x &&\text{(variables)}\\
&\mathcal{E}[\![\Pi(x:S)\to T]\!] &&= \Pi(x:\mathcal{E}[\![S]\!])\to\mathcal{E}[\![T]\!] &&\text{(Explicit }\Pi\text{)}\\
&\mathcal{E}[\![\Pi\{x:S\}\to T]\!] &&= \forall(x:\mathcal{E}[\![S]\!])\to\mathcal{E}[\![T]\!] &&\text{(Implicit }\Pi\text{)}\\
&\mathcal{E}[\![\lambda(x:S).\,e]\!] &&= \lambda(x:\mathcal{E}[\![S]\!]).\,\mathcal{E}[\![e]\!] &&\text{(Explicit }\lambda\text{)}\\
&\mathcal{E}[\![\lambda\{x:S\}.\,e]\!] &&= \mathcal{E}[\![e]\!] &&\text{(Implicit }\lambda\text{)}\\
&\mathcal{E}[\![e\;u]\!] &&= \mathcal{E}[\![e]\!]\;\mathcal{E}[\![u]\!] &&\text{(Explicit application)}\\
&\mathcal{E}[\![e\;\{u\}]\!] &&= \mathcal{E}[\![e]\!] &&\text{(Implicit application)}
\end{aligned}
$$

Staging



Plain use

Eval quoted use — !⟦⟧

Stage 0

Elaboration time

Quoted use — ⟦⟧

Stage 1

Circuit run-time

Plain use

Escaped quoted use — ∼ ⟦⟧

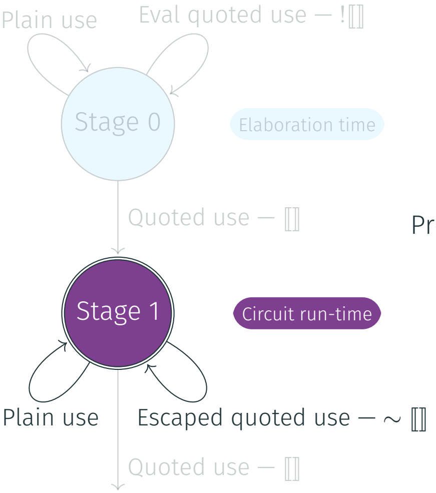
Quoted use — ⟦⟧

Staging distinguishes elaboration and circuit run-time

Ensures elaboration can complete *without* inspecting any circuit run-time values

Uses the ⟦. . .⟧, ∼, !, and ⟨. . .⟩ syntax

Staging



Plain use

Eval quoted use — $!\llbracket\,\rrbracket$

Stage 0

Elaboration time

Quoted use — $\llbracket\,\rrbracket$

Stage 1

Circuit run-time

Plain use

Escaped quoted use — $\sim\llbracket\,\rrbracket$

Quoted use — $\llbracket\,\rrbracket$

Typechecker extensions ensure consistient use

Prevents values known only at circuit run-time from being used during elaboration time

$$\frac{(\lambda x :_n S) \in \Gamma \qquad n \leq m}{\Gamma_m \vdash x : S}\ (\mathrm{var}_\lambda)$$

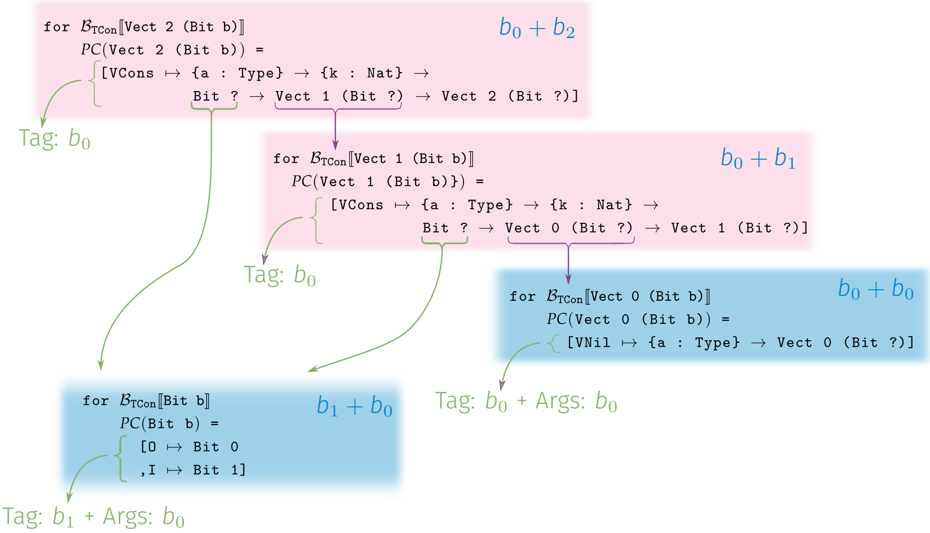
Automatic bit representations

Since we represent circuits as plain functions,
we need a way to synthesise user data types into bit representations

```
simple Vect : Nat → Type → Type where
  VNil  : {a : Type}                            → Vect Z     a
  VCons : {a : Type} → {k : Nat} → a → Vect k a → Vect (S k) a
```

Let’s reuse the already required unification engine for help

Automatic bit representations



for $\mathcal{B}_{\text{TCon}}[\![$Vect 2 (Bit b)$]\!]$ $b_0 + b_2$

PC(Vect 2 (Bit b)) =

{[VCons ↦ {a : Type} → {k : Nat} →
Bit ? → Vect 1 (Bit ?) → Vect 2 (Bit ?)]

Tag: b_0

for $\mathcal{B}_{\text{TCon}}[\![$Vect 1 (Bit b)$]\!]$ $b_0 + b_1$

PC(Vect 1 (Bit b)}) =

{[VCons ↦ {a : Type} → {k : Nat} →
Bit ? → Vect 0 (Bit ?) → Vect 1 (Bit ?)]

Tag: b_0

for $\mathcal{B}_{\text{TCon}}[\![$Vect 0 (Bit b)$]\!]$ $b_0 + b_0$

PC(Vect 0 (Bit b)) =

{[VNil ↦ {a : Type} → Vect 0 (Bit ?)]

Tag: b_0 + Args: b_0

for $\mathcal{B}_{\text{TCon}}[\![$Bit b$]\!]$ $b_1 + b_0$

PC(Bit b) =

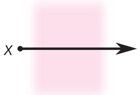
{[O ↦ Bit 0
,I ↦ Bit 1]

Tag: b_1 + Args: b_0

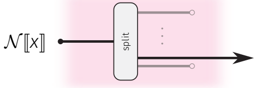
Normalisation to a netlist

We reuse the normalisation (by evaluation) system also required in typechecking to normalise down to a tiny language which is circuit-friendly:

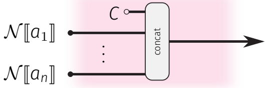
$\mathcal{N}\llbracket x\rrbracket$



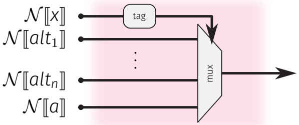
$\mathcal{N}\llbracket \pi_i^C x\rrbracket$



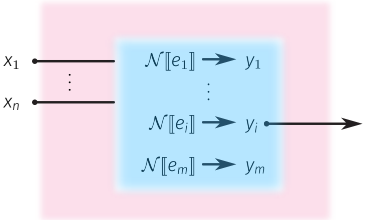
$\mathcal{N}\llbracket C_{\texttt{simple}}\ \overline{a}\rrbracket$


C
$\mathcal{N}\llbracket a_1\rrbracket$
$\mathcal{N}\llbracket a_n\rrbracket$
concat

$\mathcal{N}\llbracket \texttt{case}\ x\ \texttt{of}\ \overline{alt}\ [\texttt{default}\ a]\rrbracket$


$\mathcal{N}\llbracket x\rrbracket$
tag
$\mathcal{N}\llbracket alt_1\rrbracket$
$\mathcal{N}\llbracket alt_n\rrbracket$
$\mathcal{N}\llbracket a\rrbracket$
mux

$\mathcal{N}\llbracket \lambda\,\overline{x:\tau_1}\,.\,\texttt{let}\ \overline{y:\tau_2 \mapsto e}\ \texttt{in}\ y_i\rrbracket$


x_1
x_n
$\mathcal{N}\llbracket e_1\rrbracket$
y_1
$\mathcal{N}\llbracket e_i\rrbracket$
y_i
$\mathcal{N}\llbracket e_m\rrbracket$
y_m

The *future*

(what's left to do?)

Further work

- Support for synchronous logic
- ...and its place in our correct-by-construction verification
- A fully-typed synthesis scheme
- A formalisation of synthesisability requirements
- A rebase on Idris 2
- Netlist optimisations for vendor tools

The *impact* (outputs and more contributions)

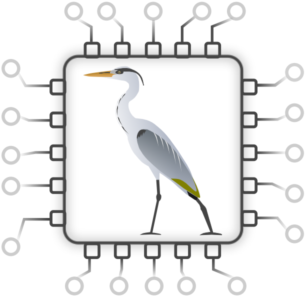
Outputs

- Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications
  L. Crockett, D. Northcote, C. Ramsay, F. Robinson, and R. Stewart
  Strathclyde Academic Media, Book — 2019
- Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework
  J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. Crockett, and R. Stewart
  IEEE Open Access, Journal paper — 2020
- On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits
  C. Ramsay, L. Crockett, and R. Stewart
  2021 IEEE MWSCAS, Conference paper — 2021
- Low-cost, High-speed Parallel FIR Filters for RFSoC Front-Ends Enabled by CλaSH
  C. Ramsay, L. Crockett, and R. Stewart
  IEEE Asilomar, Conference paper — 2021
- Data for `toatie`— A Hardware Description Language With Dependent Types
  C. Ramsay, L. Crockett, and R. Stewart
  Self-published, Digital artefact — 2022

HAFLANG

an EPSRC project for the Hardware Acceleration of Functional Languagues

`haflang.github.io`



Appendix

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

Let’s subdue the “noise”

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

What do the types guarantee?

What do they *not* guarantee?

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

Data constructors for synthesisable types have non-synthesisable arguments!

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

How do we know which function arguments must be applied before the function becomes synthesisable?

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
          ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
              let rec = addU _ _ _ _ xbs ybs cin'
                  ans = UCons _ _ _ rec lsb
              in eqInd2 _ _ _
                      prfAddU c xn yn a b xsn ysn prf
                      (λh ⇒ Unsigned (S (S w)) h)
                      ans
```

We pattern match on circuit runtime values...

Is this is irrefutably OK? Can elaboration complete?

Dangers in unsigned binary addition

```
-- Unsigned binary addition in simulation
addU : (w,x,y,c : Nat)
       Unsigned w x → Unsigned w y → Bit c →
       Unsigned (S w) (plus c (plus x y))

pat c, cin ⇒
  addU 0 0 0 c UNil UNil cin
    = UCons _ 0 c UNil cin

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) _ _ c (UCons w xsn xn xbs xb)
                   (UCons w ysn yn ybs yb)
                   cin
    = case (addBit _ _ _ cin xb yb) of
        pat a, b, prf, cin', lsb
           ⇒ (MkBitPair a b _ prf cin' lsb) ⟹
                let rec = addU _ _ _ _ xbs ybs cin'
                    ans = UCons _ _ _ rec lsb
                in eqInd2 _ _ _
                          prfAddU c xn yn a b xsn ysn prf
                          (λh ⇒ Unsigned (S (S w)) h)
                          ans
```

Goal is to safely reason about:

circuit runtime

vs

elaboration-time

vs

typechecking only?

Dangers in unsigned binary addition — reprise

```
-- Unsigned binary addition (in toatie)
addU : (w : Nat) → {x,y,c : Nat} →
       ⟨ Unsigned w x ⟩ → ⟨ Unsigned w y ⟩ → ⟨ Bit c ⟩ →
       ⟨ Unsigned (S w) (plus c (plus x y)) ⟩

pat c, cin ⇒
  addU 0 {0} {0} {c} ⟦ UNil ⟧ ⟦ UNil ⟧ cin
    = ⟦ UCons {_} {0} {c} UNil ~cin ⟧

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) {_} {_} {c} ⟦ UCons {w} {xsn} {xn} xbs xb ⟧
                         ⟦ UCons {w} {ysn} {yn} ybs yb ⟧
                         cin
    = ⟦ case (addBit {_} {_} {_} ~cin xb yb) of
          pat a, b, prf, cin', lsb
          ⇒ (MkBitPair {a} {b} {_} {prf} cin' lsb) ⟹
             let rec = ~(addU _ {_} {_} {_} ⟦ xbs ⟧ ⟦ ybs ⟧ ⟦ cin' ⟧ )
                 ans = UCons {_} {_} {_} rec lsb
              in eqInd2 {_} {_} {_}
                        {prfAddU c xn yn a b xsn ysn prf}
                        {λh ⇒ Unsigned (S (S w)) h} ans
      ⟧
```

Data constructors for synthesisable types have non-synthesisable arguments!

How do we know which function arguments must be applied before the function becomes synthesisable?

We pattern match on circuit runtime values...

Dangers in unsigned binary addition — reprise

```
-- Unsigned binary addition (in toatie)
addU : (w : Nat) → {x,y,c : Nat} →
       ⟨ Unsigned w x ⟩ → ⟨ Unsigned w y ⟩ → ⟨ Bit c ⟩ →
       ⟨ Unsigned (S w) (plus c (plus x y)) ⟩

pat c, cin ⇒
  addU 0 {0} {0} {c} ⟦ UNil ⟧  ⟦ UNil ⟧  cin
    = ⟦  UCons {_} {0} {c} UNil ~cin  ⟧

pat w, c, xsn, xn,  xbs, xb, ysn, yn,  ybs, yb, cin ⇒
  addU (S w) {_} {_} {c} ⟦  UCons {w} {xsn} {xn}  xbs xb  ⟧
                         ⟦  UCons {w} {ysn} {yn}  ybs yb  ⟧
                         cin
    = ⟦  case (addBit {_} {_} {_} ~cin xb yb) of
           pat a, b, prf, cin', lsb
           ⇒ (MkBitPair {a} {b} {_} {prf} cin' lsb) ⟹
              let rec = ~(addU _ {_} {_} {_} ⟦ xbs ⟧ ⟦ ybs ⟧ ⟦ cin' ⟧ )
                    ans = UCons {_} {_} {_} rec lsb
                in eqInd2 {_} {_} {_}
                          {prfAddU c xn yn a b xsn ysn prf}
                          {λh ⇒ Unsigned (S (S w)) h} ans
      ⟧
```

Data constructors for synthesisable types have non-synthesisable arguments!

How do we know which function arguments must be applied before the function becomes synthesisable?

We pattern match on circuit runtime values…

Normalisation to a netlist

We reuse the normalisation (by evaluation) system also required in typechecking

Our irrelevance and staging annotations mean we can hopefully normalise down to a tiny language which is circuit-friendly:

$\tau,\ \sigma\ ::=$	Types
$D_{\mathsf{S}}\ \overline{\tau}$	Fully applied simple type
$a\ ::=$	Argument expressions
x	Local variable
$\mid\ C_{\mathsf{S}}\ \overline{a}$	Fully applied simple data constructor
$e\ ::=$	Subexpressions
x	Local variable
$\mid\ C_{\mathsf{S}}\ \overline{a}$	Fully applied simple data constructor
$\mid\ \mathtt{case}\ x\ \mathtt{of}\ \overline{alt}\ [\mathtt{default}\ a]$	Case with optional default
$\mid\ \pi_i^C\, x$	Projection
$alt\ ::=\ C_{\mathsf{S}}\ \overline{x} \longrightarrow a$	Alternatives
$g\ ::=\ \lambda\ \overline{x : \tau}\ .\ \mathtt{let}\ \overline{y : \sigma \mapsto e}\ \mathtt{in}\ z$	Top-level circuit