TYPES FOR EXACT REAL COMPUTATION

(USING AERN2/HASKELL)

Michal Konečný, Eike Neumann

Aston University, Birmingham, UK

CCC 2017, Nancy, Loria

INTRODUCTION

- Why Haskell/ML/...?
  - Conveniently supports new abstractions
    - Can express a lot of maths almost literally
  - Powerful type system (not full dependent types)
  - Runs quite fast (not as fast as C++)
  - Easy parallelisation
- Why Haskell/ML/... for exact real computation?
  - Prototyping of Comput. Analysis ideas, algorithms
  - Practical reliable numerical computation

INTRODUCTION

- AERN2 - Haskell package for exact real computation
  - since 2008, several rewrites
  - exact reals, continuous real functions
  - multiple evaluation strategies, including:
    - iRRAM-style
    - lazy Cauchy reals
    - parallel execution
    - (distributed execution via MPI etc)
- Here we focus on AERN2 exact real types

WHAT DO WE WANT?

- *exact* real numbers, functions...
  - **easy to use** (types help)
  - **reliable** (types help a lot)
  - not terribly slow
    - eg FFT ($n = 1024$)
      - Double arithmetic: 0.04s
      - Exact 1000 digits: 0.7s
- scalable to solving ODE, PDE, hybrid systems,...

WHAT DO WE WANT? - SAFETY

- partial functions, eg

$$\sqrt{1-x^2} \quad \text{Type: Warning!}$$

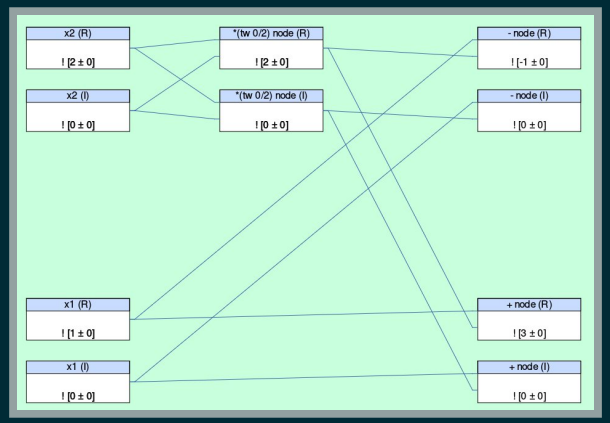
$$\sqrt{x^2-2xy+y^2} \quad \text{OK even for } x=y\in\mathbb{R}$$

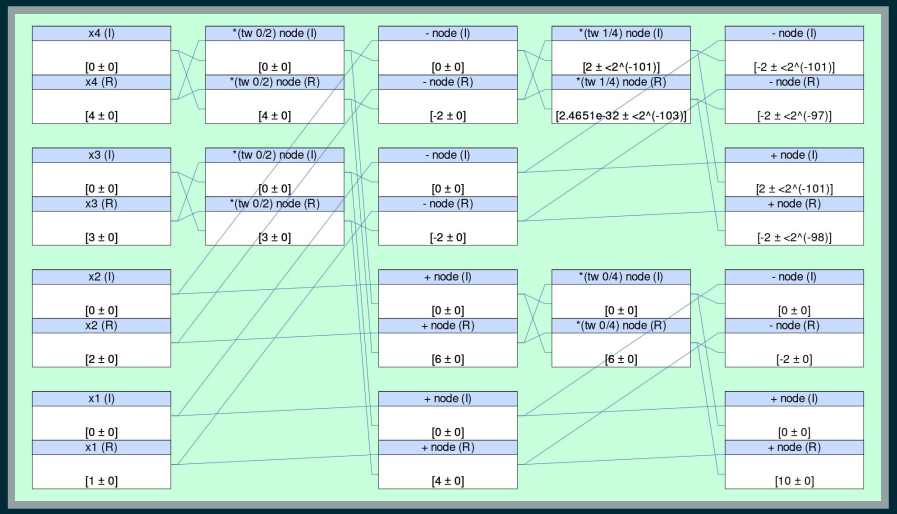
WHAT DO WE WANT? - CHOICE

- parallel if
  - eg $a(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$
  - our types should not exclude this
- multi-valued functions, eg trisection, pivoting

WHAT DO WE WANT? - STRATEGY

one code, multiple evaluation strategies





...

- FFT(n=1024), 1000 decimal digits:
  - CR: N1: 8.5s N2: 5.2s N3: 3.8s N4: 3.2s
  - MP: N1: 0.7s N2: ? N3: ? N4: ?

EXACT VALUES IN AERN2

ARBITRARY-ACCURACY BALLS

```
data MPBall = MPBall MPFloat ErrorBound
data ErrorBound = ... -- ~ Double
```

```
> :t pi100
pi100 :: MPBall

> pi100
[3.141592653589793 ± <2^(-100)]
> (getPrecision pi100, getAccuracy pi100)
(Precision 103,bits 100)
```

```
> pi100 > 3.14
Just True

> pi100 == pi100
Nothing
```

EXACT VALUES IN AERN2

CAUCHY REAL NUMBERS

```
type CauchyReal = FastConvSeq MPBall
type FastConvSeq enclosure = -- ~ AccuracySG -> enclosure
data AccuracySG = -- strict + guide
```

```
> :t pi
pi :: CauchyReal

> pi ? (bitsSG 500000 600000)
[3.141592653589793 ± <2^(-600001)]
```

```
type EffortConvSeq enclosure = -- ~ Effort -> enclosure
type Effort = AccuracySG -- (for convenience)
```

```
> :t pi == pi
pi == pi :: EffortConvSeq (Maybe Bool)

> (pi == pi) ? (bitsSG 100 100)
Nothing
```

MIXING TYPES IN EXPRESSIONS

We want flexibility:

```
twiddle :: Integer -> Integer -> Complex CauchyReal
twiddle k n = exp (-2*(k/n)*pi*i)
    where i = 0:+1 -- :: Complex Integer
```

```
(/) :: Integer    -> Integer          -> Rational
(*) :: Integer    -> Rational         -> Rational
(*) :: Rational   -> CauchyReal       -> CauchyReal
(*) :: CauchyReal -> Complex Integer -> Complex CauchyReal
exp :: Complex CauchyReal             -> Complex CauchyReal
```

- also vectors, matrices, functions, ...
- flexibility $\leftrightarrow$ type safety & inference

MIXING TYPES IN EXPRESSIONS

Numeric types in normal Haskell:

```
(+) :: (Num t) => t -> t -> t
1   :: (Num t) => t
1.5 :: (Fractional t) => t
pi  :: (RealFloat t) => t
```

Numeric types in AERN2:

```
(+) :: (CanAdd t1 t2) => t1 -> t2 -> (AddType t1 t2)
1   :: Integer
1.5 :: Rational
pi  :: CauchyReal
```

co-developed with Pieter Collins

MIXING TYPES IN EXPRESSIONS

EXPRESSIONS WITH GENERIC TYPES, INFERENCE

```
-- non-AERN Haskell:
average xs = (sum xs) / (convert $ length xs)
-- inferred type:
average :: (Fractional t, Convertible Int t) => [t] -> t
```

```
-- AERN2:
average xs = (sum xs) / (length xs)
-- inferred type:
average :: (ConvertibleExactly Integer t, CanDiv t Int,
      CanAdd t t, AddType t t ~ t) => [t] -> DivType t Int
-- manually specified type:
average :: (Field t) => [t] -> t
-- aliases provided by AERN2:
type CanAddSameType t = (CanAdd t t, AddType t t ~ t)
type Ring t = (HasIntegers t, CanAddSameType t, ...)
type Field t = (Ring t, CanDivBy t Int, ...)
```

EXPRESSIONS WITH PARTIAL FUNCTIONS

ERROR-COLLECTING MONAD

```
type CN t = CollectErrors NumErrors t

sqrt :: MPBall -> CN MPBall
sqrt :: CN MPBall -> CN MPBall

type CauchyRealCN = FastConvSeq (CN MPBall)

sqrt :: CauchyReal -> CauchyRealCN
sqrt :: CauchyRealCN -> CauchyRealCN
sqrt :: FastConvSeq (CN a) -> FastConvSeq (EnsureCN (SqrtType a))
 -- EnsureCN is a type function, adds CN unless already there
```

EXPRESSIONS WITH PARTIAL FUNCTIONS

PARTIAL FUNCTIONS IN PRACTICE (1/2)

```
> sqrt (-1)
{[(ERROR,out of range: sqrt: argument must be >= 0: [-1 ± 0])]}

> sqrt 0
[0 ± 0]
```

```
> let x = real((1/3) ~!); y=x in sqrt(x^2-2*x*y+y^2)
{[(POTENTIAL ERROR,out of range: sqrt: argument must be >= 0:
  [3.1115076389305 71e-61 ± <2^(-198)])]}

> let x = real((1/3) ~!); y=x in (sqrt(x^2-2*x*y+y^2) ~!)
[7.509732082812 05e-31 ± <2^(-100)]
```

EXPRESSIONS WITH PARTIAL FUNCTIONS

PARTIAL FUNCTIONS IN PRACTICE (2/2)

```
average :: (Field t) => [t] -> CN t
average xs = (sum xs) / (length xs)

... case ensureNoCN (average list) of
      Right (avg :: t) -> ...
      Left err -> ...
```

```
f1 :: CauchyReal -> CauchyReal
f1 x = x/!(log 2)

f2 :: CauchyReal -> CauchyRealCN
f2 x = (x^x)/!(log 2)

f2 :: CauchyReal -> CauchyReal
f2 x = (x^!x)/!(log 2)
```

PARALLEL BRANCHING - IF

```
myAbs :: CauchyReal -> CauchyRealCN
myAbs r = if r < 0 then -r else r
```

redefined Haskell's if-then-else:

```
ifThenElse :: Bool -> t -> t -> t -- original
-- redefined, most generic type:
ifThenElse :: (HasIfThenElse b t) =>
              b -> t -> t -> (IfThenElseType b t)
-- "parallel if" instance used above:
ifThenElse ::
  (CanUnionCNSameType t) =>
  EffortConvSeq (Maybe Bool) ->
  FastConvSeq t -> FastConvSeq t -> FastConvSeq (EnsureCN t)
```

PARALLEL BRANCHING - PICK

```
trisection ::
  (Rational -> CauchyReal) -> (Rational,Rational) -> CauchyReal
trisection f (l,r) =
  ... (withAccuracy :: AccuracySG -> MPBall)
  where
  withAccuracy ac = onSegment (l,r)
    where
    onSegment (a,b) =
      let ab = mpBall ((a+b)/2, (b-a)/2) in
      if getAccuracy ab >= ac
        then ab
        else pick (map withSegment subsegments)
        -- pick :: [EffortConvSeq (Maybe t)] -> t,  here t=MPBall
    withSegment (c,d) = ... (withAcc :: AccuracySG -> Maybe MPBall)
      where
      withAcc acF
          | ((f c)?acF) * ((f d)?acF) !<! 0 = Just $ onSegment (c, d)
          | otherwise = Nothing
```

EVALUATION STRATEGIES AND EFFECTS

One algorithm, compute iRRAM-style or CauchyReals

How to parallelise?

How to add caching?

How to log the computation?

EVALUATION STRATEGIES

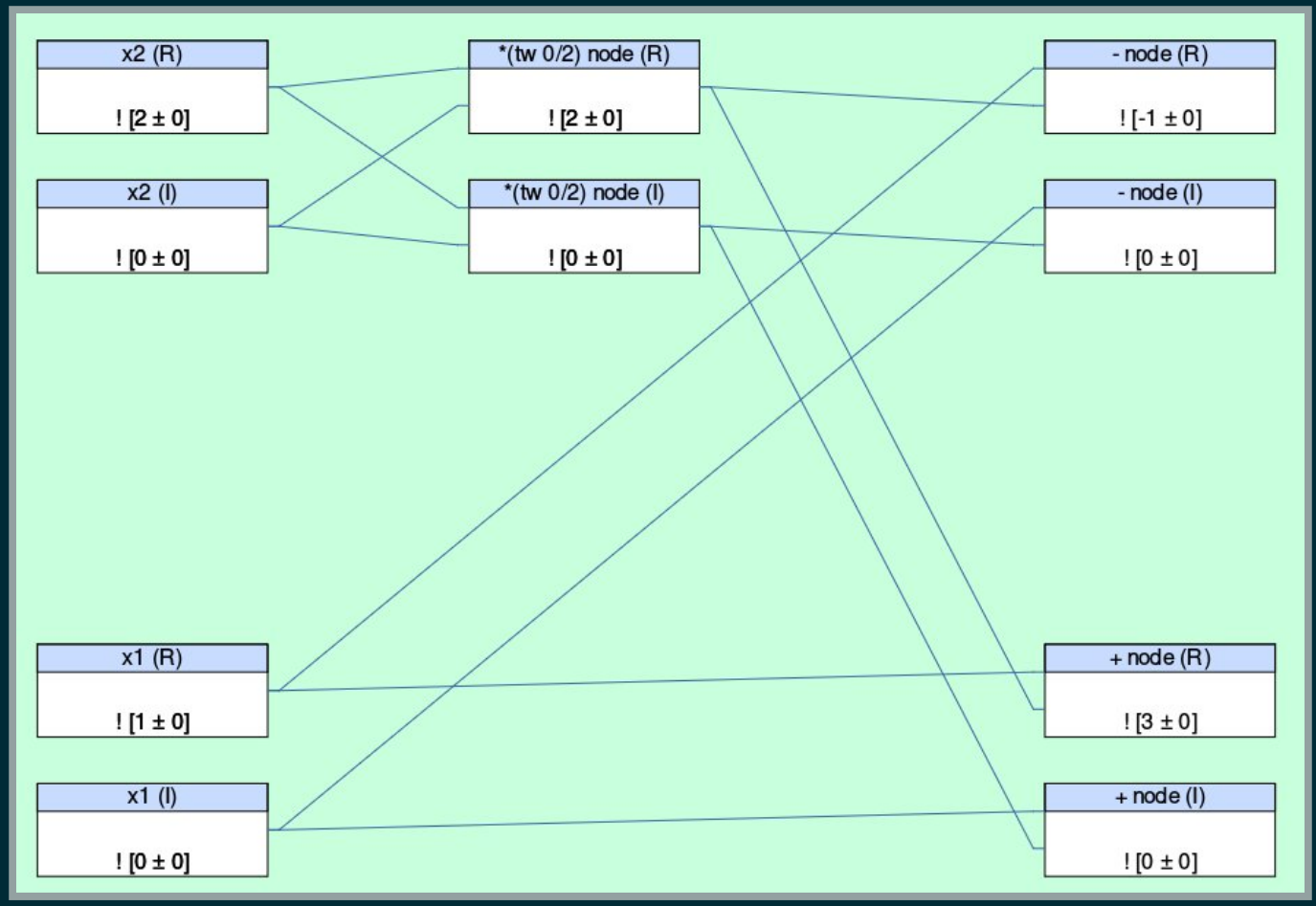
DISCRETE/FAST FOURIER TRANSFORM (DFT/FFT)

$$X_k = \sum_{\ell=0}^{n} x_\ell e^{-2\pi i k \ell / n}$$

```
fft i n s (x :: [Complex CauchyReal])
  | n == 1 = [x !! i]                    -- base case
  | otherwise =
    let
      nHalf = n `div` 2
      yEven = fft i nHalf (2*s) x     -- recursion 1 (divide)
      yOdd = fft (i+s) nHalf (2*s) x -- recursion 2
      yOddTw = zipWith (*) yOdd ...   -- multiply by constants (tw k n)
      yL = zipWith (+) yEven yOddTw  -- vector addition
      yR = zipWith (-) yEven yOddTw  -- vector subtraction
    in
    (yL <> yR) :: [Complex CauchyReal]
```

EVALUATION STRATEGIES

FFT LOGGING, CACHING, PARALLELISATION



EVALUATION STRATEGIES

ARROWS

- Arrow ~ a kind of category in Haskell
- `(->)` arrow: category of Haskell functions
- `QACached` arrow: `(->)` +
  - caching answers in sequences
  - query-answer logging
- `QAPar` arrow: `QACached` + parallel queries

EVALUATION STRATEGIES

ARROW-GENERIC EXPRESSIONS

```
-- multiplication in any QAArrow:
(*) :: (QAArrow to) =>
  (SequenceA to a) -> (SequenceA to b) -> (SequenceA to (AddType a b))
  -- does NOT automatically register the result sequence
```

```
-- "register" a sequence, eg to set up answer caching:
(-:-) :: (QAArrow to) => (SequenceA to a) `to` (SequenceA to a)

-- "register" a sequence + prefer parallel query evaluation:
(-:-||) :: (QAArrow to) => (SequenceA to a) `to` (SequenceA to a)
```

Caching occurs automatically for `to = (->)`.

EVALUATION STRATEGIES

LARGER EXAMPLE

```
regComplex
  | isParallel =
    proc (a:+b) -> do
      a' <- (-:-||) -< a
      b' <- (-:-||) -< b
      returnA -< (a':+b')
  | otherwise =
    proc (a:+b) -> do
      a' <- (-:-) -< a
      b' <- (-:-) -< b
      returnA -< (a':+b')
```

EVALUATION STRATEGIES

LARGER EXAMPLE

```
aux :: Integer -> Integer -> Integer ->
  [Complex (CauchyRealA to)] `to` [Complex (CauchyRealA to)]
aux i n s = ... convLR
  where
  convLR =
    proc x -> do



      let yL = zipWith (+) yEven yOddTw
      let yR = zipWith (-) yEven yOddTw
      mapA (regComplex (nI == n)) -< yL ++ yR
regComplex
  | isParallel =                | otherwise =
    proc (a:+b) -> do             proc (a:+b) -> do
      a' <- (-:-||) -< a            a' <- (-:-) -< a
      b' <- (-:-||) -< b            b' <- (-:-) -< b
      returnA -< (a':+b')           returnA -< (a':+b')
```

EVALUATION STRATEGIES

LARGER EXAMPLE

```
aux :: Integer -> Integer -> Integer ->
  [Complex (CauchyRealA to)] `to` [Complex (CauchyRealA to)]
aux i n s = ... convLR
  where
  convLR =
    proc x -> do
      yEven <- aux i nHalf (2*s) -< x
      yOdd <- aux (i+s) nHalf (2*s) -< x
      yOddTw <- mapWithIndexA twiddleA -< yOdd
      let yL = zipWith (+) yEven yOddTw
      let yR = zipWith (-) yEven yOddTw
      mapA (regComplex (nI == n)) -< yL ++ yR
regComplex
  | isParallel =                | otherwise =
    proc (a:+b) -> do             proc (a:+b) -> do
      a' <- (-:-||) -< a            a' <- (-:-) -< a
      b' <- (-:-||) -< b            b' <- (-:-) -< b
      returnA -< (a':+b')           returnA -< (a':+b')
```

CONCLUSION

- Haskell types help reliability & ease of use
  - check compatibility of mixed-type operands
  - highlight use of partial operations
  - support safe parallel if and parallel pick
  - support strategy-generic programs using Arrow
- Haskell programs are easy to parallelise

FUTURE WORK

- Keep AERN2 in sync with Ariadne and iRRAM
- Multi-variate real functions, ODE, PDE, HS solving
- Taylor models; SMT proofs over reals