Exact Real Computation in AERN

Michal Konečný, Eike Neumann, *Aston University*

Pieter Collins, *Maastricht University*

Talk goals

- Overview of approaches to *exact real computation* (*ERC*)
- Some experimental comparisons of these approaches
- Why functional programming for ERC
- Why Arrows for ERC
- Overview of AERN (a Haskell ERC library)

Aston University

Goals of Exact Real Number Computation

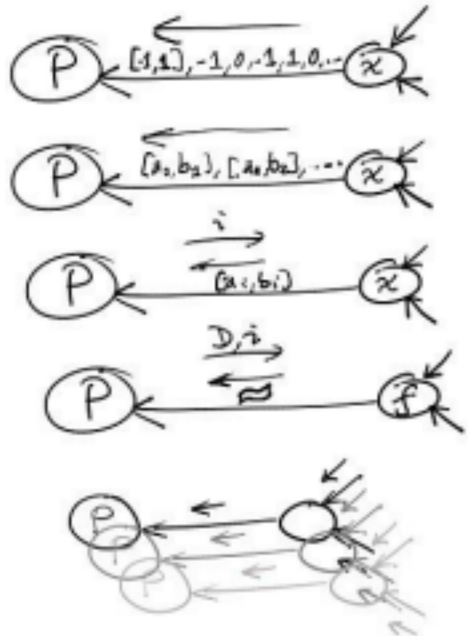
- first-class real numbers and functions
- close to familiar mathematical notation
- very reliable; ideally, verified
- as efficient as possible

    selecting execution strategy, hand-tuning

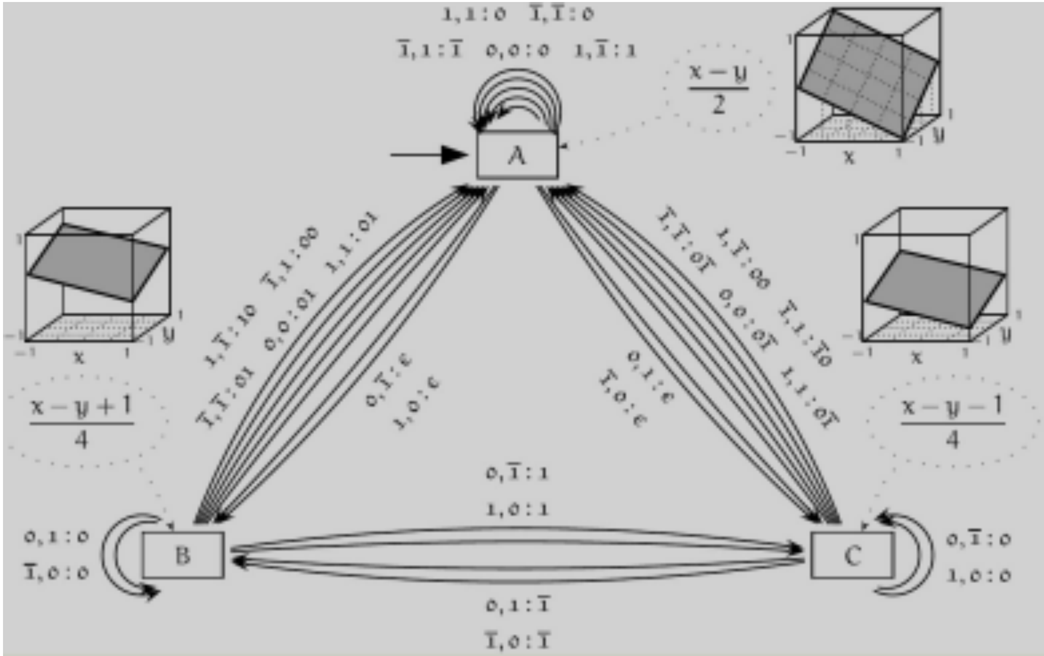
```
twiddle(k,n) = exp(-2*k*i*pi/n) -- mixing integers, reals and complex nums
myexp(x) = lim (\ n -> sum [ (x^k)/(k !) | k <- [1..n] ] ± errorBound(x,n))
              where errorBound(x,k) = ...
newton(f,f',iX_0) = iterateLim iX_0
                    (\ iX -> x - f(x)/f'(iX) where x = pickAny iX)

bad(x) = if (x == 0) then 1 else 0 -- disallow? or allow non-termination?
```

Execution strategies, approximation representations

- **Dedekind cuts + Abstract Stone Duality** (Marshall)
    - `sqrt = forall (\a -> ( (a>0) --> exists (\x -> x > 0 /\ x*x == a )))`
- **streams of enclosure refinements**
    - signed digits, continued fractions
- **(Fast converging) Cauchy sequences**
    - streams of enclosures, N -> E(X)
- **query-answer protocols** (cf Oracle machine)
    - represented as functions Q -> E(X)
        - eg, part of a function domain
- computation with **iterative increase of precision**



Streams of enclosure refinements

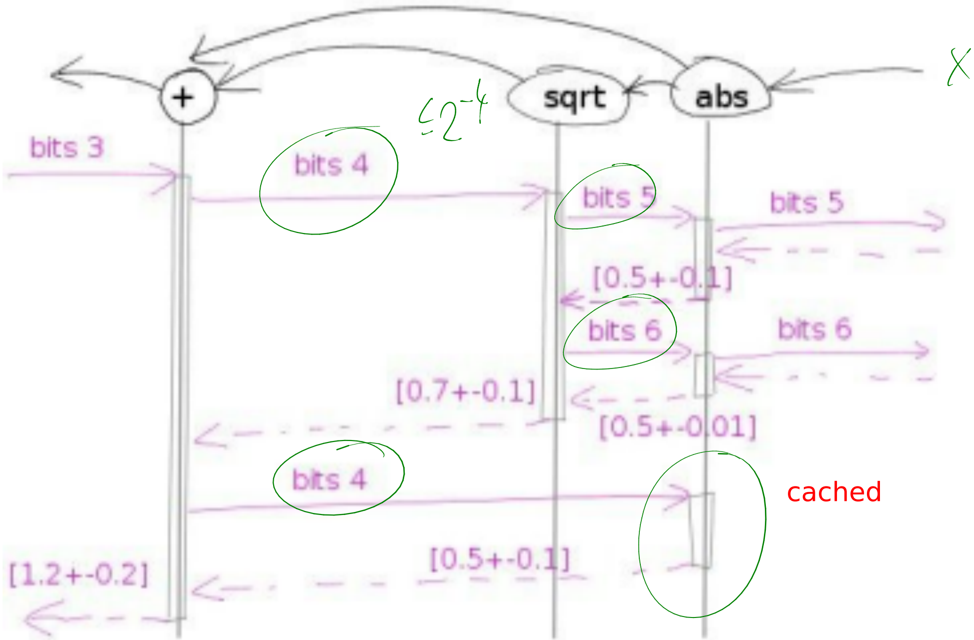


signed binary (x-y)/2 over [-1,1] by a **finite machine** with 3 states

(Fast-converging) Cauchy sequences

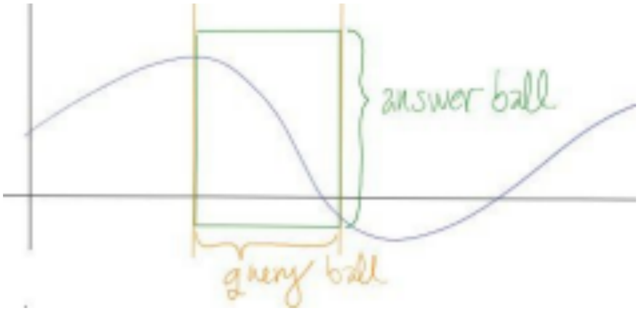
$\sqrt{abs(x)} + abs(x)$

```
type CReal = Accuracy -> MPBall

addCR r1 r2 = \ac -> r1 (ac + 1) + r2 (ac + 1)
```



Query-answer protocols

```
type ContFuncB = MPBall -> MPBall
```




```
type ContFuncP = (Interval Rational, Accuracy) -> PolynomialBall
```

Iterative increase of precision/effort

```
logistic c x0 n = tryPrecisions (\p -> logisticP p c x0 n)
```

Evaluated with $c = 3.82$, $n = 100$, $x_0 = 0.125$ and accuracy 2^{-100}:

```
Precision 55: result = Just [8.990933937732085e-1 ± Infinity]
Precision 89: result = Just [4.309357923818564e-1 ± Infinity]
Precision 144: result = Just [4.309357854522346e-1 ± 5.596784653385525e28918307]
Precision 233: result = Just [4.309357854522346e-1 ± 4.250486463172112e-13]
Precision 377: result = Just [4.309357854522346e-1 ± 2.613345252574047e-56]
```

Some recent ERC implementations

- streams of enclosure refinements
  - **IC Reals**, by L Errington, 2000, C and Haskell
  - not yet **ERCA** by WK Ho, 2013, Haskell
- (Fast converging) Cauchy sequences
  - **ERA**, by D Lester, 2003, Haskell (now CReals in package numbers)
  - **ireal** by Bjorn von Sydow, 2014, Haskell
  - **exact-real** by Joe Hermaszewski, 2015, Haskell
- query-answer protocols
  - **AERN**, by MK, 2007-ongoing
- enclosure (usually interval/ball) computation with iterative increase of precision/effort
  - **iRRAM** by N Mueller, 2000-ongoing, C++
  - **Ariadne** by P Collins, 2005-ongoing, C++
  - **haskell-fast-reals** by Ivo List, 2015-ongoing, Haskell
- representing Dedekind cuts using Abstract Stone Duality
  - **Marshall** by A Bauer, P Taylor, 2005-ongoing, OCaml, Haskell

AERN history

- 2005-2007: focus on multi-variate polynomial arithmetic in Chebyshev basis
  - $R^n \to R^m$ first-class values, approximated by piecewise polynomials
  - used in Polypaver
    - deciding real inequalities in Floating-point software verification
- 2008-2009: focus on parallel dataflow with general query-answer protocols
- 2010-2011: rewrite, focus on flexible effort specification and test coverage
- 2010-2014: focus on hybrid systems simulation
- 2015-ongoing: rewrite (https://github.com/michalkonecny/aern2)
  - focus on usability and multiple evaluation strategies

AERN current goals

- first-class real numbers and functions
- close to familiar mathematical notation
- very reliable, ideally, verified
- as efficient as possible
  - selecting execution strategy, hand-tuning

- **convenient** to use
- easy to write **composable & reusable** programs
- **safe to use** (static types), **reliable** (well tested), (eventually) **verified**
  - ERC *algorithm* ← separation → ERC *evaluation strategy*
  - based on **computable analysis**
- **multiple evaluation strategies** supported for ERC algorithms
  - facilitating comparisons and experimentation
- **efficient** (modulo a multiplicative constant 10-100) execution
  - can choose evaluation strategy, including parallel, distributed
  - can apply different strategies to different parts of the computation

Haskell laziness, type classes, purity

- `let numfrom n = n : (numfrom (n + 1))`
- `1 :: (Num a) => a -- ad hoc polymorphism`
  - `1 + (q :: Rational) -- 1 :: Rational here`
  - `1 + (Mod10 9) == Mod10 0 -- 1 :: Mod10 here`
- `instance (Num (Mod10)) where (Mod10 a)+(Mod10 b) = Mod10((a+b)%10)`
- `(+) :: (Num a) => a -> a -> a`
  - `no side effects allowed in (+)`
  - `the result depends only on the 2 operands`
    - cannot specify effort (eg precision, Taylor Model sweep limit)
  - side-effects allowed only in IO monad
    - eg `print :: (Show a) -> IO ()`
      - `(IO a)` build and executes an imperative program

AERN basic number types

Unambiguous literal types (unlike Haskell Prelude)

- `1/3` `:: Fractional t => t` `Rational`

Mixed-type operations (unlike Haskell Prelude)

- lesser need for explicit conversions
  - `\x -> x + 1` `:: Num t => t -> t` `CanAdd t Integer => t -> t`
  - `\x -> x + k` `:: Integer -> Integer` `CanAdd t Integer => t -> t`

  AddType t Integer

- allows a more efficient implementation of eg (polynomial + number)
  - `f + k`

AERN exact real numbers and intervals

- `MPBall`

  arbitrary precision centre, fixed-precision error bound

- `CauchyReal`

  encapsulates `Accuracy -> MPBall`

- `Interval MPBall`
- `Interval CauchyReal`
- `lim :: (Integer -> Interval r) -> r          -- (r ~ CauchyReal)`

  eg, `lim (\ n -> sum [ (x^k)/(k!) | k <- [1..n] ] ± errorBound(x,n))`

- `iterateLim :: (Interval r) -> (Interval r -> Interval r) -> r`

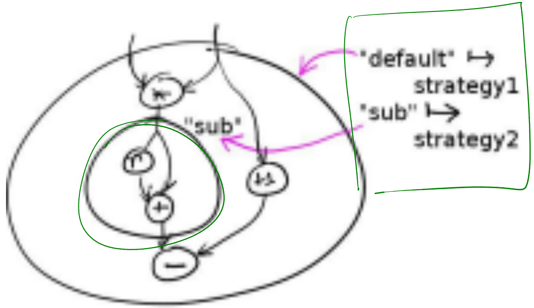
  eg, `iterateLim iX_0 (\ iX -> let x = pickAny iX in x - f(x)/f’(iX))`

- `pickAny :: Interval r -> r`

Real number algorithm vs evaluation strategy

Goal

```
myAlgorithm `evalWith` iterateForPrecisions

myAlgorithm `evalWith` cachedCauchyReal
```



How?

- **symbolic expressions**
  - ie. roll your own programming language
  - either very limited power or a huge amount of work
- **Arrow-generic expressions**
  - as used in various FRP libraries and circat by Connal Elliott
  - quite flexible and powerful

Haskell Arrows (see https://www.haskell.org/arrows/index.html)

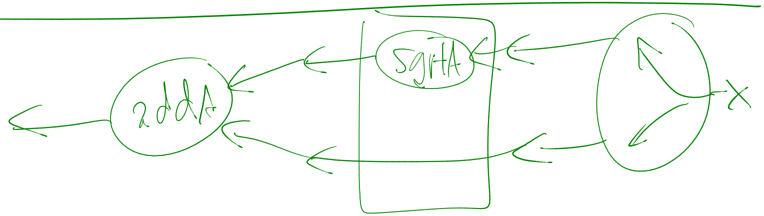
- Arrow API for algebraic representation of a “network”:

  (“network” = DAG-composition of (potentially) effectful computations)

```
arr :: (Arrow to) => (a -> b) -> (a `to` b) -- embed any Haskell code
first :: (Arrow to) => (a `to` b) -> (a,c) `to` (b,c) -- separate channels
(<<<) :: (Category to) => (b `to` c) -> (a `to` b) -> (a `to` c)
```

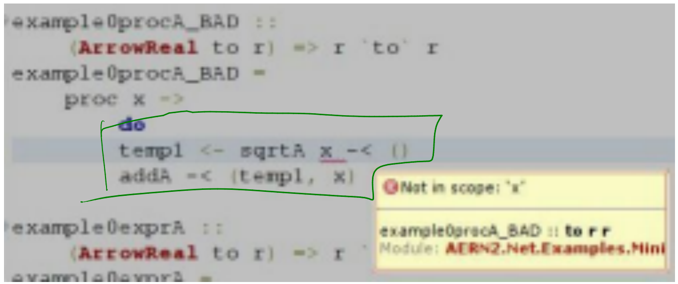
- example

```
example0A :: (ArrowReal to r) => r `to` r
example0A =
    addA   <<<   first sqrtA   <<<   arr (\x -> (x,x))
```



Convenient notation for arrow-generic expressions

```
example0A :: (ArrowReal to r) => r `to` r
example0A =
      addA <<< first sqrtA <<< arr (\x -> (x,x))
-- equivalently, with Haskell language extension “Arrows”:
      proc x ->
          do
          temp1 <- sqrtA -< x
          addA -< (temp1, x)
```



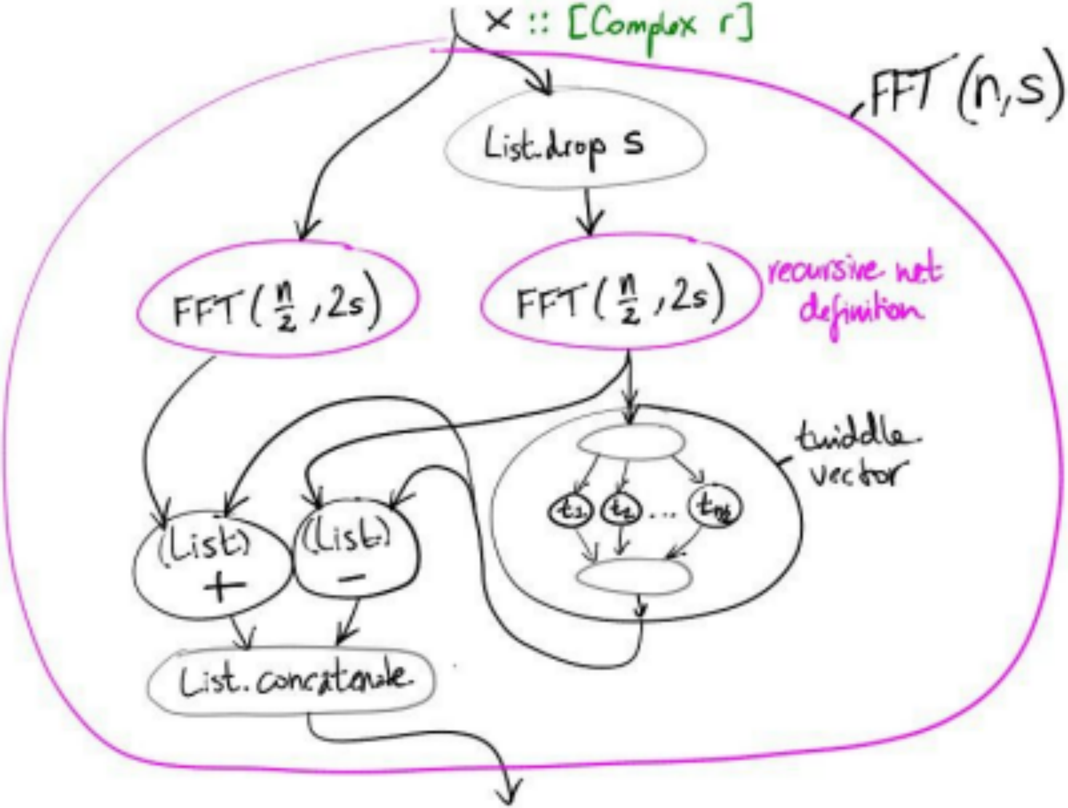

```
-- equivalently, using a macro (Template Haskell):
      $(exprA[| let [x] = vars in sqrt(x) + x |])
```

Combining arrow-generic expressions

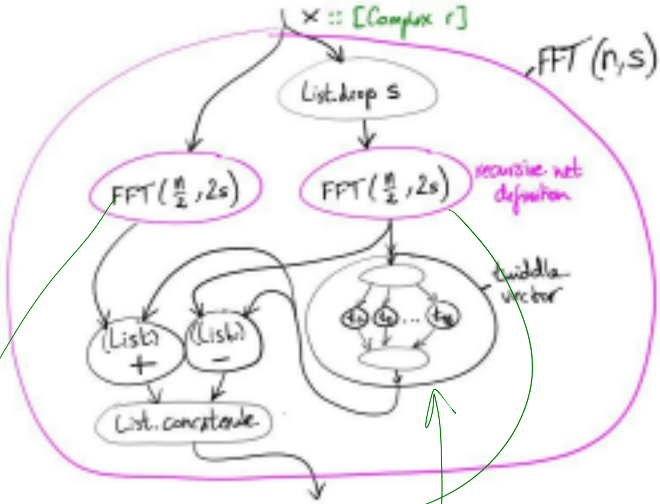
```
twiddleA ::
  (ArrowReal to r) => (Integer, Integer) -> () `to` (Complex r)
twiddleA (k,n) =
  $(exprA[| let [i]=vars in exp(-2*k*i*pi/n) |]) <<< complex_iA
```

- can use Integer, Rational, CauchyReal constants that are in scope
- can mix different ways of writing arrow-generic expressions

Family of networks - FFT



Family of networks - FFT




```
ditfft2 ::
    (ArrowReal to r)
    =>
    Integer {-^ @N@ -} ->
    Integer {-^ @s@ -} ->
    [Complex r] `to` [Complex r]
ditfft2 nN s
    | nN == 1 =
        proc (x0:_) ->
            returnA -< [x0]
    | otherwise =
        proc x ->
            do
            vTX0 <- ditfft2 nNhalf (2 * s) -< x
            vTXNhalf <- ditfft2 nNhalf (2 * s) -< drop (int s) x
            vTXNhalfTwiddled <- mapAwithPos twiddleA -< vTXNhalf
            vX0 <- zipWithA addA -< (vTX0, vTXNhalfTwiddled)
            vXNhalf <- zipWithA subA -< (vTX0, vTXNhalfTwiddled)
            returnA -< vX0 ++ vXNhalf
    where
    nNhalf = round (nN / 2)
    twiddleA k =
```

Limit of a sequence of intervals (Taylor expansion)

```
myExpA =
  limA $ \n -> proc x ->
    do
    terms <- mapA termA -< [(x,k) | k <- [0..n]]
    s <- sumA -< terms
    eb <- errorBoundA n <<< absA -< x
    plusMinusA -< (s,eb)
  where
  termA = proc (x,k) ->
      do
      temp1 <- powA -< (x,k)
      divA -< (temp1, (k!))
  errorBoundA n =
      $(exprA[| let [absx]=vars in (absx^(n + 1))*3/((n + 1)!)|])
```

Limit of a sequence of intervals (Newton iteration)

```
newtonA :: (...) =>
   (r `to` r) {-^ f -} ->
   (Interval r `to` Interval r) {-^ f' -} ->
   Interval r `to` (LimitTypeA to (Interval r))
newtonA f f' =
    iterateLimA $
        proc iX -> do
            x <- pickAnyA -< iX
            fx <- f -< x
            f'iX <- f' -< iX
            temp1 <- divA -< (singleton fx,f'iX)
            subA -< (singleton x,temp1)

iterateLimA :: (...) => (a `to` a) -> a `to` LimitTypeA to a
```

`iterateLimA fnA` repeats the network `fnA` infinitely many times

`fnA` contracts ⇒ only a finite portion is used for each query

Real functions in networks

```
newton2A :: (ArrowFunction fn, r~FnR fn, r ~ RnDomR fn) =>
  (fn, fn, Interval r) `to` (LimitTypeA to (Interval r))
newton2A =
    proc (f,f’,iX) ->
        iterateLimA $
            x <- pickAnyA -< iX
            fx <- evalAtPointA -< (f,x)
            f'iX <- evalOnIntervalA -< (f’,iX)
            temp1 <- divA -< (singleton fx,f'iX)
            subA -< (singleton x,temp1)
```

Tracing cached Cauchy real evaluation

```
$(exprA[| let [x] = vars in sqrt(x) + x |])
```

```
QANetLogCreate (ValueId 1) [] "1 % 3"
QANetLogCreate (ValueId 2) [ValueId 1] "sqrt"
QANetLogCreate (ValueId 3) [ValueId 2,ValueId 1] "+"
| QANetLog_Query (ValueId 3) "Bits 100"
| | QANetLog_Query (ValueId 2) "Bits 100"
| | | QANetLog_Query (ValueId 1) "Bits 100"
| | | QANetLogAnswer (ValueId 1) "cache was empty" "[3.333e-1 ±
  2.242e-44]"
| | QANetLogAnswer (ValueId 2) "cache was empty" "[5.773e-1 ± 4.484e-44]"
| | QANetLog_Query (ValueId 1) "Bits 100"
| | QANetLogAnswer (ValueId 1) "used cache" "[3.333e-1 ± 2.242e-44]"
| QANetLogAnswer (ValueId 3) "cache was empty" "[9.106e-1 ± 1.121e-43]"
```

Tracing cached Cauchy real evaluation: Limits

```
newtonA f f' =
    iterateLimA $
        proc iX -> do
            x <- pickAnyA -< iX
            fx <- f -< x
            f'iX <- f' -< iX
            temp1 <- divA -< (singleton fx,f'iX)
            subA -< (singleton x,temp1)
```

```
...
| QANetLog_Query (ValueId 3) "Bits 1"
...
| | QANetLogCreate (ValueId 22) [ValueId 5,ValueId 21] "-"
| | QANetLogCreate (ValueId 23) [ValueId 5,ValueId 18] "-"
| | | QANetLog_Query (ValueId 22) "Bits 3"
...
| | | QANetLogAnswer (ValueId 22) "cache was empty" "[1.375 ± 0]"
...
| QANetLogAnswer (ValueId 3) "cache was empty" "[1.40625 ± 3.125e-2]"
```

Compare strategies - Logistic map iteration

```
logisticA :: (RealExprA to r) => Rational -> Integer -> r `to` r
logisticA c n =
    (foldl1 (<<<) (replicate (int n) step))
    where
    step = $(exprA[|let [x]=vars in  c * x * (1 - x)|])
```

Evaluation Strategy	n = 10	n = 100	n = 1000	n = 10000
Direct CauchyReal	0.2s	-	-	-
Cached CauchyReal	0.02s	0.06s	0.54s	14s
Iterative precision increase	0.02s	0.06s	0.63s	15s
MPBall with manual precision	0.02s	0.03s	0.13s	2.6s
iRRAM using REAL (ie balls)				around 1s

Compare strategies - FFT

Evaluation Strategy	n = 16	n = 64	n = 512	n = 2048
Direct CauchyReal	2.3s	77s	-	-
Cached CauchyReal	0.18s	1.02s	12.5s	64s
Iterative precision increase	0.18s	0.82s	19.4s	101s
MPBall with manual precision	0.16s	0.89s	9.9s	49s

Evaluation of arrow-generic ERC

- **convenient** to use
- easy to write **composable & reusable** programs
- **safe to use** (static types), **reliable** (well tested), (eventually) **verified**
  - ERC *algorithm* ← separation → ERC *evaluation strategy*
  - based on **computable analysis**
- **multiple evaluation strategies** supported for ERC algorithms
  - facilitating comparisons and experimentation
- **efficient** (modulo a multiplicative constant 10-100) execution
  - can choose evaluation strategy, including parallel, distributed
  - can apply different strategies to different parts of the computation

Internals of arrow-generic ERC in AERN

- each strategy = different arrow to + different type r

  meets the constraint `ArrowReal to r`

- all Cauchy real operations are defined arrow-generically, eg:

```
instance (CanAsCauchyRealA to r) => CanNegA to (AsCauchyReal r) where
    negA = unaryOp "neg" neg (getInitQ1FromSimple id)

unaryOp name op getInitQ1 =
    proc r1 ->
        do
        r1Id <- getSenderIdA -< r1
        newCRA -< ([r1Id], Just name, ac2b r1)
    where
    ac2b r1 = proc ac ->
        do
        q1InitMB <- getInitQ1 -< (ac, r1)
        ensureAccuracyA1 getA1 op -< (ac, q1InitMB)
        where
        getA1 =
            proc q1 -> getAnswerCRA -< (r1,q1)
```

Summary

- Various ERC representations and evaluation strategies
- AERN2 provides safe, fairly convenient and relatively fast ERC
- Comparing different evaluation strategies for a single ERC algorithm

  Some comparisons for logistic map and FFT

Future work

- Study efficiency of different real number and function representations

  Eg rational vs polynomial enclosures vs CR $\to$ CR
- Parallel cached evaluation strategy, other strategies
- Many-valued operations (iRRAM-style)
- Taylor Model arithmetic (wrapping)
- Easier to use, eg, make `$(exprA[|...|])` more flexible