A Domain-Specific Language for Digital Signal Processing

Emil Axelsson, Karina Bunyik, Kálmán Karch

FP workshop

June 2009

Background

- Goal:
  - Raise abstraction level at which DSP algorithms are programmed
  - Increase code portability and maintainability
  - No compromise on efficiency (hopefully)
- High-level DSP:
  - Achieved using Haskell-style list-based programming (plus additional support for matrices etc.)
- Efficient execution
  - Achieved by making a domain-tailored embedded language in Haskell

Background

- Embedded language makes development much easier
  - No parser / type checker
  - Easy to experiment with new features
  - Reuse existing infrastructure (e.g. testing frameworks)
- Multi-stage compilation enables efficient execution
  - Haskell only used for code generation
  - *Haskell → core language → C (or perhaps assembler)*
  - Intermediate core language simple and compiler-friendly
- (In future, we may revise decision of using embedded approach)

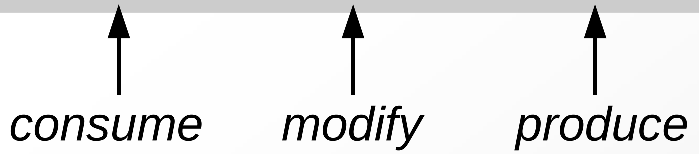
High-level data processing

- Functional programming paradigm encourages writing programs as a network of data structure manipulations
  - Data structures serve as “glue”
  - Data structures used both for data and control

- Classic example: Sum of squares

```
square x = x*x

sumSq n = sum (map square [1..n])
```



consume *modify* *produce*

High-level data processing

- Advantages of data manipulation style
  - Modularity
  - Clarity
  - No need for control structures
  - Equational reasoning
    - Fusion frameworks are able to transform programs like `sumSq` into a single constant-space loop
    - Only generally possible in pure languages
- Possible disadvantages
  - Intermediate structures not always possible to remove
  - Difficult to know what optimizations will happen

Data manipulation suitable for DSP

```
divBy x ys = map (`div` x) ys

corr xs ys = sum (zipWith (*) xs ys)

normalized l ys = find (\ys -> corr ys ys <= l) yss
  where
    yss = map (\x -> divBy (2^x) ys) [0..]

autoCorr m xs ws = take m (map (corr ys) (init (tails ys)))
  where
    ys = normalized 15 (zipWith (*) xs ws)
```

Current Disp

- Low-level core language
  - Like C but functional (pure, explicit state)
  - Special parallel tiling construct
  - Acts as interface to back-end compiler (ELTE University Budapest)
  - Programmers should rarely use core language directly
- High-level “vector” library
  - Inspired by Haskell's list library
  - Programs written as vector transformations
  - Completely separated from core language
  - Vector programs generally generate efficient core code
    - Optimization happens up front
  - Other high-level interfaces might be possible in future

Example: Sum of squares

```
square x = x*x

sumSq n = sum $ map square $ isPar $ enumFromTo 1 n
```

Example: Sum of squares

```
square x = x*x

sumSq n = sum $ map square $ isPar $ enumFromTo 1 n
```

Core output:

```
main v0 = v30_1
  where
    (v30_0,v30_1) = while cont body (0,0)
      where
        cont (v1_0,v1_1) = v13
          where
            v11 = v0 - 1
            v13 = v1_0 <= v11
        body (v14_0,v14_1) = (v18,v25)
          where
            v18 = v14_0 + 1
            v21 = v14_0 + 1
            v23 = v21 * v21
            v25 = v14_1 + v23
```

Core output is not DSL code!
(but it is runnable Haskell)

A single constant-space loop

Programmer's interface – core language

```
value :: (…) => a -> Data a
array :: (…) => Dimension [a] -> [a] -> Data [a]

getIx :: (…) => Data [a] -> Data Int -> Data a
setIx :: (…) => Data [a] -> Data Int -> Data a -> Data [a]
(!)   :: (…) => Data [a] -> Data Int -> Data a -> Data [a]

ifThenElse :: (…) => Data Bool -> (a -> c) -> (a -> c) -> (a -> c)

while :: (…) => (a -> Data Bool) -> (a -> a) -> (a -> a)

parallel :: (…) =>
    Int -> Data Int -> (Data Int -> Data a) -> Data [a]
```

`Data` implemented as a GADT with observable sharing at each node

Overloaded to work directly on Haskell structures (e.g. pairs)

While-loop

```
while :: (…) => (a -> Data Bool) -> (a -> a) -> (a -> a)
```

continue condition

“body” – current state to next state

entire loop – initial state to final state

```
whileExample = while (<3) (+1) (0 :: Data Int)
```

```
*Main> eval whileExample
3

*Main> whileExample
main v0 = v10
  where
    v10 = while cont body 0
      where
        cont v1 = v1 < 3
        body v5 = v5 + 1
```

For-loop

- The core language has no for-loop, but we can easily define our own:

```
for :: (…) => Data Int -> Data Int -> a -> (Data Int -> a -> a) -> a
for start end init body = snd $ while cont body' (start,init)
  where
    cont          = (<= end) . fst
    body' (i,s) = (i+1, body i s)
```

```
forExample = for 5 9 (0 :: Data Int) (\i s -> s + a!i)
  where
    a = array (10 :> IntArr) [1..10]
```

```
*Main> eval forExample
40
```

`a!5 + a!6 + … + a!9`

Parallel “loop” (tiling)

```
parallel :: (…) => Int -> Data Int -> (Data Int -> Data a) -> Data [a]
```

static size
(for allocation)

dynamic size
(tile size)

index function

result vector

```
parallelExample = parallel 10 10 (+5)
```

```
*Main> eval parallelExample
[5,6,7,8,9,10,11,12,13,14]

*Main> parallelExample
main v0 = v6
  where
    v6 = parallel 10 10 ixf
      where
        ixf v1 = v1 + 5
```

Symbolic vectors

```
data Vector t a where
    Indexed
      :: Data Size
      -> (Data Ix -> a) ←  index function (see parallel tiling)
      -> Vector Par a   ←  "parallel" vector

    Unfold
      :: (…)
      => Data Size
      -> (s -> (a,s))   ←  "step" function (computes element and next state)
      -> s              ←  initial state
      -> Vector Seq a   ←  "sequential" vector
```

- Parallel vectors are “simpler” and allow parallel execution
- Sequential vectors allow sharing results from previous elements

Example vectors

- The numbers 1, 3, 5 … 19 as parallel vector:

Example vectors

- The numbers 1, 3, 5 … 19 as parallel vector:
  - `Indexed 10 ((+1).(*2))`

Example vectors

- The numbers 1, 3, 5 … 19 as parallel vector:
  - `Indexed 10 ((+1).(*2))`

- The first 100 Fibonacci numbers as sequential vector:

Example vectors

- The numbers 1, 3, 5 … 19 as parallel vector:
  - `Indexed 10 ((+1).(*2))`
- The first 100 Fibonacci numbers as sequential:
  - `Unfold 100 (\(a,b) -> (a, (b,a+b))) (1,1)`

Symbolic to/from “hard” (=core) vectors

- Symbolic vectors do not necessarily “contain” elements; they contain *methods* for computing vector elements
- In some cases they do use actual memory:

```
toHard1 :: (…) => Size -> Vector t (Data a) -> Data [a]
toHard1 szs (Indexed szd ixf)   = parallel szs szd ixf
toHard1 szs (Unfold szd step s) =
    snd $ for 0 (value (szs-1)) (s, array (szs :> undefined) []) body
  where
    body i (s,vec) = (s', setIx vec i a)
      where (a,s') = step s

fromHard1 :: (…) => Size -> Data [a] -> Vector Par (Data a)
fromHard1 sz arr = Indexed (value sz) (getIx arr)
```

These functions convert *one* level of nesting. They are used internally by `toHard`/`fromHard`, which convert all levels at once.

Converting between symbolic vectors

```
toSeq :: Vector t a -> Vector Seq a
toSeq (Indexed sz ixf)   =
```

Converting between symbolic vectors

```
toSeq :: Vector t a -> Vector Seq a
toSeq (Indexed sz ixf)    = Unfold sz (\i -> (ixf i, i+1)) 0
toSeq (Unfold sz step s) = Unfold sz step s
```

Always cheap!

Converting between symbolic vectors

```
toSeq :: Vector t a -> Vector Seq a
toSeq (Indexed sz ixf)    = Unfold sz (\i -> (ixf i, i+1)) 0
toSeq (Unfold sz step s) = Unfold sz step s
```

Always cheap!

```
toPar :: (…) => Dimension [a] -> Vector t e -> Vector Par (Data a)
toPar szs vec = Indexed sz ixf
  where
    sz  = length vec
    ixf = getIx (toHard szs vec)
```

Puts the whole vector in memory.
Resulting parallel vector has cheap lookups.

Intermediate vectors, fusion

- Vector operations only manipulate the index/step functions:

```
map :: (a -> b) -> (Vector t a -> Vector t b)
map f (Indexed sz ixf)    = Indexed sz (f . ixf)
map f (Unfold sz step s) = Unfold sz ((f *** id) . step) s
```

- This gives us fusion by default!
  - Example: `map f . map g  ==  map (f . g)`

One loop instead of two.
No intermediate structure.

- The only vector operation that allocates memory is `toPar`
  - No need to rely on smart compiler for fusion
  - Programmer is in control

Avoiding fusion

```
sumSq n
    = sum $ map square
    $ toPar (10 :> IntArr) $ isPar $ enumFromTo 1 n
```

Avoiding fusion

```
sumSq n
    = sum $ map square
    $ toPar (10 :> IntArr) $ isPar $ enumFromTo 1 n
```

Puts the whole enumeration in memory.
Might be good if we want to reuse it.

Avoiding fusion

```
sumSq n
    = sum $ map square
    $ toPar (10 :> IntArr) $ isPar $ enumFromTo 1 n
```

Puts the whole enumeration in memory.
Might be good if we want to reuse it.

```
mul a b = map (\aRow -> map (scalarProd aRow) b') a
  where
    b' = toPar (10:>10:>IntArr) (transpose b)
```

Avoiding fusion

```
sumSq n
    = sum $ map square
    $ toPar (10 :> IntArr) $ isPar $ enumFromTo 1 n
```

Puts the whole enumeration in memory.
Might be good if we want to reuse it.

```
mul a b = map (\aRow -> map (scalarProd aRow) b') a
  where
    b' = toPar (10:>10:>IntArr) (transpose b)
```

Moves all elements of `b` before multiplication.

Operations on symbolic vectors

```
(++) :: (…) => Vector t a -> Vector t a -> Vector t a

Indexed sz1 ixf1 ++ Indexed sz2 ixf2 = Indexed (sz1+sz2) ixf
  where
    ixf i = ifThenElse (i < sz1) ixf1 (ixf2 . (+sz1)) i

Unfold sz1 step1 s1 ++ Unfold sz2 step2 s2 =
    Unfold (sz1+sz2) step (0, (s1,s2))
  where
    step (n, (s1',s2')) = n<sz1 ?
      ( let (a,s1'') = step1 s1' in (a, (n+1, (s1'', s2')))
      , let (a,s2'') = step2 s2' in (a, (n+1, (s1', s2'')))
      )
```

- No elements moved (use `toPar` to actually move data)
- Some small overhead in new index/step functions

Current Disp (summary)

- Low-level core language
  - Like C but functional (pure, explicit state)
  - Special parallel tiling construct
  - Acts as interface to ELTE compiler
  - Programmers should rarely use core language directly
- High-level “vector” library
  - Inspired by Haskell's list library
  - Programs written as vector transformations
  - Completely separated from core language
  - Vector programs generally generate efficient core code
    - Optimization happens up front
  - Other high-level interfaces might be possible in future

Future work

- Try out examples, add functionality on demand
  - Should not require (big) changes to core language
- Size inference
- Infinite vectors
- Tracing
- High-level optimizations
- …

Thanks...

… to Ericsson programmers and the “DSL group” at our department for valuable ideas and feedback.

Examples

Matrix muliplication

```
transpose a = Indexed (length $ head a) ixf
  where
    ixf y = Indexed (length a) (\x -> a ! x ! y)

scalarProd a b = sum (zipWith (*) a b)
```

```
mul a b = map (\aRow -> map (scalarProd aRow) b') a
  where
    b' = transpose b
```

Using vector lib.
No core
constructs.

```
testMul a b = toHard dim $
    mul (fromHard dim a) (fromHard dim b)
  where
    dim = 10 :> 10 :> IntArr
```

Wrapper to get
core in/output.

The result is a
pure *core*
program – all
vectors are
gone!

Matrix muliplication

```
main (v0_0,v0_1) = v47
  where
    v47 = parallel 10 10 ixf
     where
      ixf v1 = v45
        where
         v45 = parallel 10 10 ixf
          where
            ixf v2 = v42_1
              where
                (v42_0,v42_1) = while cont body (0,0)
                  where
                    cont (v3_0,v3_1) = v3_0 <= 9
                    body (v18_0,v18_1) = (v22,v37)
                      where
                        v22 = v18_0 + 1
                        v35 = ((v0_0 ! v1) ! v18_0) * ((v0_1 ! v18_0) ! v2)
                        v37 = v18_1 + v35
```

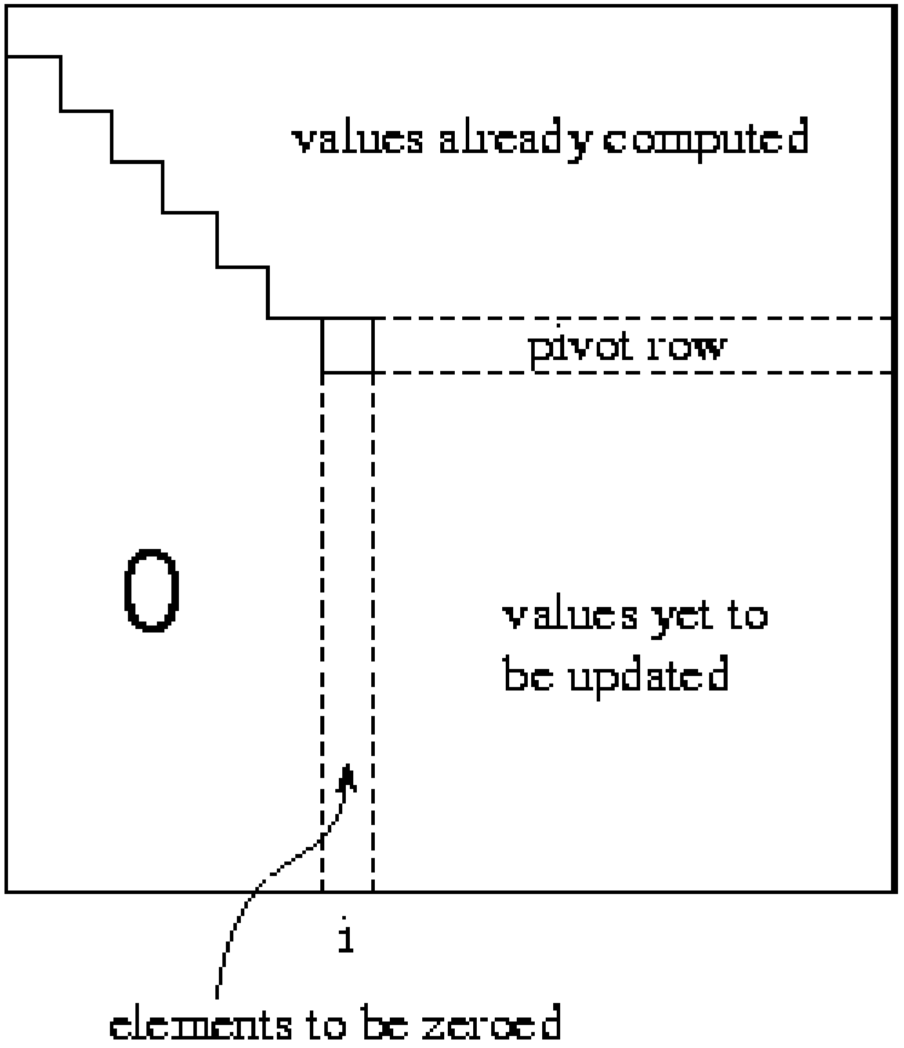
Transposition has been “fused” into loop

Examples in DISP

Gaussian elimination

Gaussian elimination I

- Why the Gaussian elimination?
- What kind of functions are needed?
- The algorithm briefly



Gaussian elimination II

```
gauss ::    Vector Par (Vector Par (Data Float))
          -> Vector Par (Vector Par (Data Float))
gauss m = valueEval $ cleanDiag $ upperTriang m
```

Gaussian elimination III

```
mLoop f [] = []
mLoop f as = b : mLoop f bs
  where
    b:bs = f as
```

Gaussian elimination IV

```
vecLoop :: (Vector Par (Vector Par (Data Float))
                                        -> Vector Par (Vector Par (Data Float)))
    -> Vector Par (Vector Par (Data Float))
    -> Vector Par (Vector Par (Data Float))
vecLoop f m = fHMf $ for 0 (length m - 1) (tHMf m) body
  where
    body i m = tHMf (m1 +++ f m2)
      where
        (m1,m2) = splitAt i (fHMf m)
```

Gaussian elimination V

```
cleanDiag :: Fractional (Data Float)
           => Vector Par (Vector Par (Data Float))
           -> Vector Par (Vector Par (Data Float))
cleanDiag m = map g m
  where
    g y = map f y
      where
        f x = (x/=0) ? (x  /  head (dropWhile (== (0::Data Float)) y),0.0)
```

Gaussian elimination VI

```
upperTriang m = vecLoop f m
  where
    f matr = take 1 matr +++ (map mapFn $ drop 1 matr)
      where
        mapFn x = zipWith zipFn x y
          where
            zipFn a b = a - b * t
              where
                t = diagElem x / diagElem y
               diagElem z = head $ dropWhile (==(0::Data Float)) z
        y = index matr 0
```

Gaussian elimination VII

```
valueEval m = reverse $ vecLoop f $ reverse m
 where
   f matr = take 1 matr +++ (map mapFn $ drop 1 matr)
     where
       mapFn x = zipWith zipFn x y
         where
           zipFn a b = a - b * t
             where
               t = index x j
           j = length y - (length $ dropWhile (==(0::Data Float)) y)
      y = index matr 0
```

Gaussian elimination VIII

- Future work