Fusion Power

Winterseminar 2012, Bled

Christian Höner zu Siederdissen

Institute for Theoretical Chemistry
University of Vienna

February 12, 2012 – February 19, 2012

universität wien

FWF Der Wissenschaftsfonds.

huh? physics?

The implementation available on the `RNAwolf` homepage is written in the high-level functional programming language Haskell. While this leads to an increase in running times (by a constant factor), the high-level notation and a library of special functions lead to very concise programs, and enable, e.g., the use of multiple cores.

A Folding Algorithm for Extended RNA Secondary Structures,
Höner zu Siederdissen, Bernhart, Stadler, Hofacker, 2011

Algebraic Dynamic Programming

- over sequence data
- separation of concerns:
  - grammar
  - algebra
  - asymptotic efficiency
- no explicit indices
- available in Haskell
- new idea: `GAP-L` and `GAP-C` (Java-like, compiles to `C++`)

Robert Giegerich *et al*, Practical Programming Group, Bielefeld University

ADP: the Nussinov'78 grammar

```
S -> e | lS | Sr | lSr | SS
```

```
s = (
  nil   <<< empty                         |||
  left  <<< base -~~ s                    |||
  right <<<          s ~~- base           |||
  pair  <<< base -~~ s ~~- base           |||
  split <<<      s +~+ s                  ... h
)
```

algebra

```
  nil   :: e -> S
  left  :: A -> S -> S
  right :: S -> A -> S
  pair  :: A -> S -> A -> S
  split :: S -> S -> S
  h     :: {S} -> {S}

pairmax:
   nil _        = 0
   left  _ x    = x
   right   x _  = x
   pair  l x r  = if pair l r then x + 1 else x
   split x y    = x + y
   h xs         = [maximum xs]
```

stream fusion basics

```
data Step s a = Yield a s
              | Skip    s
              | Done

data Stream a = forall s . Stream (s -> Step s a) s
```

- seed *s*
- element *a*
- *Yield* next element and seed
- *Skip* a step
- *Done*
- Coutts, Leshchinskiy, Stewart, 2007. Stream Fusion

Transformers

- `singleton x = Stream step True where`

- `map f (Stream next s) = Stream next' s where`

- `fold f z (Stream next s) = fold' z s where`

Transformers

- ```
  singleton x = Stream step True where
    step True  = Yield x False
    step False = Done
  ```

- ```
  map f (Stream next s) = Stream next' s where
  ```

- ```
  fold f z (Stream next s) = fold' z s where
  ```
  ```

Transformers

- `singleton x = Stream step True where`

- ```
  map f (Stream next s) = Stream next' s where
    next' s = case next s of
      Yield x s' -> Yield (f x) s'
      Skip    s' -> Skip s'
      Done       -> Done
  ```

- `fold f z (Stream next s) = fold' z s where`
  ```

Transformers

- `singleton x = Stream step True where`

- `map f (Stream next s) = Stream next' s where`

- `fold f z (Stream next s) = fold' z s where`

```
  fold' z' s = case next s of
    Yield x s' -> fold' (f z' x) s'
    Skip    s' -> fold' z' s'
    Done       -> z'
```

Transformers

- ```
  singleton x = Stream step True where
    step True  = Yield x False
    step False = Done
  ```
- ```
  map f (Stream next s) = Stream next’ s where
    next’ s = case next s of
      Yield x s’ -> Yield (f x) s’
      Skip    s’ -> Skip s’
      Done       -> Done
  ```
- ```
  fold f z (Stream next s) = fold’ z s where
    fold’ z’ s = case next s of
      Yield x s’ -> fold’ (f z’ x) s’
      Skip    s’ -> fold’ z’ s’
      Done       -> z’
  ```
  ```

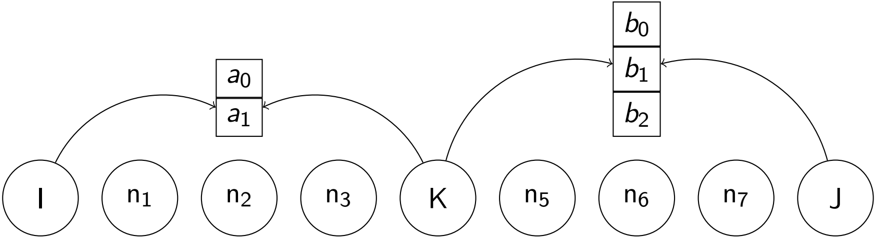
what is this good for?

- *almost all* functions are non-recursive
- *only final consumers* are recursive (fold)
- non-recursive functions can be *very well* optimized
- much easier to build “building blocks” that can be put together

- ADP creates *lists* of candidates for evaluation
- stream fusion works on "lists"
- need:
  - combinators
  - memoization (tables)
  - modern Haskell compiler

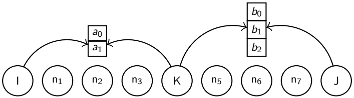
subword partitioning

- a *subword* is is substring of the input with index (i, j)
- need indices for subwords: i, k, j
- need sub-indices for sub-results: a_{ik}, b_{kj}
- need sub-results (arguments): a_0, a_1, b_0, b_1, b_2



the Tri-Stack

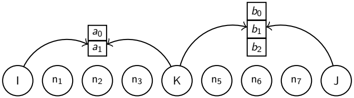
- triple of stacks defines single candidates
- subword indices, argument indices (sub-indices), arguments
- encode in Haskell:




```
( Z :.  i        :.  j
, Z
, Z
)
```

the Tri-Stack

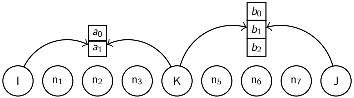
- triple of stacks defines single candidates
- subword indices, argument indices (sub-indices), arguments
- encode in Haskell:




```
( Z :.  i :.  k :.  j
, Z
, Z
)
```

the Tri-Stack

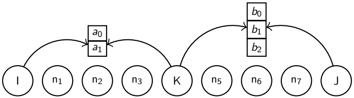
- triple of stacks defines single candidates
- subword indices, argument indices (sub-indices), arguments
- encode in Haskell:



(Z :. i :. k :. j
, Z :. a_{ik} :. b_{kj}
, Z
)

the Tri-Stack

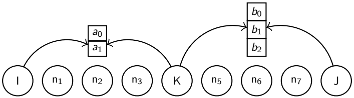
- triple of stacks defines single candidates
- subword indices, argument indices (sub-indices), arguments
- encode in Haskell:




```
( Z :.  i :.  k :.  j
, Z :.  a_ik :.  b_kj
, Z :.  a_0 :.  b_0
)
```

the Tri-Stack

- triple of stacks defines single candidates
- subword indices, argument indices (sub-indices), arguments
- encode in Haskell:



$$
\begin{array}{l}
(\ \mathtt{Z} :.\ \mathtt{i} :.\ \mathtt{k} :.\ \mathtt{j} \\
,\ \mathtt{Z} :.\ \mathtt{a}_{ik} :.\ \mathtt{b}_{kj} \\
,\ \mathtt{Z} :.\ \mathtt{a}_0 :.\ \mathtt{b}_1 \\
)
\end{array}
$$

chaining combinators

```
f <<< xs +∼+ ys

infixl 9 +~+
xs +~+ ys = Box xs mk step ys where
  mk (z:.i:.j,vs,as) = (z:.i:.i+1:.j,vs,as)
  step (z:.i:.k:.j,vs,as)
    | k+1<=j
    = Yield (z:.i:.k:.j,vs,as) (z:.i:.(k+1):.j,vs,as)
    | otherwise = Done
```

writing combinators is *still* hard?!

the dreaded interior loop: (i, k, l, j), $(k - i) + (j - l) \leq 30$

```
(#~~) = makeLeftCombinator 2 28

(~~#) xs ys = Box xs mk step ys where
  minT = 6
  minC = 2
  maxC = 30
```

writing combinators is *still* hard?!

the dreaded interior loop: (i, k, l, j), $(k - i) + (j - l) \leq 30$

```
(#~~) = makeLeftCombinator 2 28

(~~#) xs ys = Box xs mk step ys where
  minT = 6
  minC = 2
  maxC = 30
  mk (z:.k:.j,a,b) =
    let (_:.i) = z; cnsmd = k-i; l = max k (j-maxC+cnsmd)
    in return (z:.k:.l:.j,a,b)
  step (z:.k:.l:.j,a,b)
    | l<=j-(max 0 $ minT - cnsmd) && l+minC<=j
    = return Yield (z:.k:.l:.j,a,b) (z:.k:.(l+1):.j,a,b)
    | otherwise = return Done
    where cnsmd  = k-i; (_:.i) = z
```

QuickCheck to the rescue

```
f (i,j) = (,,) <<<
              fRegion #~~ fRegion ~~# fRegion
              ... id $ Z:.i:.j

g (i,j) = [ ( (i,k),(k,l),(l,j) )
                     | k <- [i..j]
                     , l <- [k..j]
                     , k-i >= 2
                     , j-l >= 2
                     , (k-i) + (j-l) >= 6
                     , (k-i) + (j-l) <= 30
                     ]

prop (i,j) = f (i,j) == g (i,j)
```

beautiful code generation

```
stream :: DIM2 -> Int
stream = f <<< xs +~+ ys +~+ zs ... h
xs i k = i + k +  23
ys k l = k + l +  42
zs l j = l + j + 123
h = sum
```

```
stream (I# i) (I# j) =
case <=# i j of _ {
  False -> I# 0;
  True -> I# (outerLoop 0 i (+# i 1) j (+# j 1) j)
}
```

. . . still beautiful

```
outerLoop acc i k j' jp1 j = case <=# (+# k 1) j' of _
    False -> case <=# jp1 j of _
      False -> acc;
      True  -> outerLoop acc jp1 (+# jp1 1) j (+# j 1) j
    True -> innerLoop acc i k (+# k 1) j'
                  (+# (+# i k) 23) i (+# k 1)
                  j' jp1 j

innerLoop acc i k l j' acc23 i' kp1 j' jp1 j =
  case <=# (+# l 1) j' of _
    False -> outerLoop acc i' kp1 j' jp1 j;
    True -> innerLoop
        (+# acc (+# (+# acc23 (+# (+# k l) 42))
              (+# (+# l j') 123)))
        i k (+# l 1) j' acc23 i' kp1 j' jp1 j
```

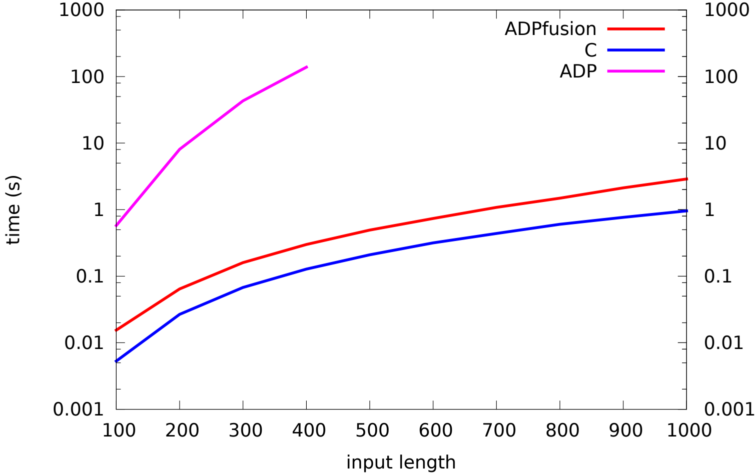
what is possible

- an ADP dialect:

```
pair  <<< base -~~ s ~~- base |||
split <<<       s +~+ s       ... h
```

- lazy & strict, immutable & mutable, scalar & vector data
- auto-adaptive code generation (no superfluous indices)
- full Haskell language available

RNAfold v2: `C` vs. `Haskell`



conclusion, future work

- DP framework for sequence data
- within $\times 2 - \times 3$ of well-optimized `C` for real-world problems
- graceful transition from legacy ADP (with all benefits) to ADPfusion
- warm & fuzzy: can have stochastic backtracking, all kinds of effects (monadic framework)

- sparse dynamic programming
- ADPfusion for 2d landscapes, co-folding, genome-wide scans
- RNAwolf, CMcompare are moving: gain speed, confidence in correctness
- faster than C
- long term goal: world domination

thanks and acknowledgments

- Ivo Hofacker, Robert Giegerich, Roman Leshchinskiy
- the Austrian FWF, SFB F43 RNA-SEQ

Höner zu Siederdissen, Christian. 2012.
Sneaking around *concatMap*
efficient combinators for dynamic programming

http://hackage.haskell.org/package/ADPfusion

universität wien

FWF Der Wissenschaftsfonds.

argument stack, functions & uncurrying

have
```
split :: Int → Int → Int
split x y = x+y
Z :. 3 :. 5
```

want
```
split :: (Z:.Int:.Int) → Int
split (Z:.x:.y) = x+y
```

Haskell magic

```
class Apply x where
  type Fun x :: *
  apply :: Fun x → x
instance Apply (Z:.a1···:.ak → res) where
  type Fun (Z:.a1···:.ak → res) = a1 →···→ ak → res
  apply fun (Z:.a1···:.ak) = fun a1 ... ak
```

stream generation: one argument

```
class StreamGen t r | t -> r where
  streamGen :: t -> DIM2 -> Stream r

instance (ExtractValue (DIM2 -> Scalar elm)
   , Asor (DIM2 -> Scalar elm) ~ k
   , Elem (DIM2 -> Scalar elm) ~ elm)
  => StreamGen (DIM2 -> Scalar elm) (DIM2,Z:.k,Z:.elm)
```

stream generation: many arguments

$+\sim+ \equiv$ Box ...

```
instance
( ExtractValue cntY, Asor cntY ~ cY, Elem cntY ~ eY
, cntY ~ ys
, StreamGen xs
, Idx2 _idx ~ idx
) => StreamGen
       (Box xs mk step ys)
       (idx:.Int,adx:.cX:.cY,arg:.eX:.eY)
```

value extraction

```
class ExtractValue cnt where
  type Asor cnt :: *
  type Elem cnt :: *
  extractStream :: cnt
    -> S.Stream (Idx3 z,astack,vstack)
    -> S.Stream (Idx3 z,astack:.Asor cnt,vstack:.Elem cnt)

instance ExtractValue (UZ.MArr0 DIM2 elm) where
  type Asor (UZ.MArr0 sh elm) = Z
  type Elem (UZ.MArr0 sh elm) = elm
  extractStream cnt stream = S.mapM addElm stream where
    addElm (z:.k:.x:.l, as, vs) = do
      vadd <- index cnt (Z:.k:.x)
      vadd `seq` return (z:.k:.x:.l, as:.Z, vs :. vadd)
```