Efficient Implementation of Dynamic Programming Algorithms

Christian Höner zu Siederdissen

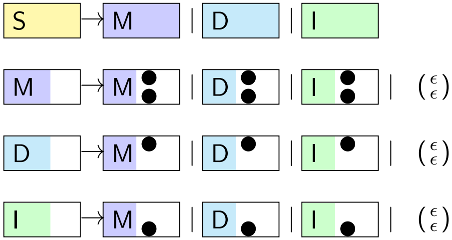
Bioinformatics Group, Dept. of Computer Science, University of Leipzig

May 23th, 2019

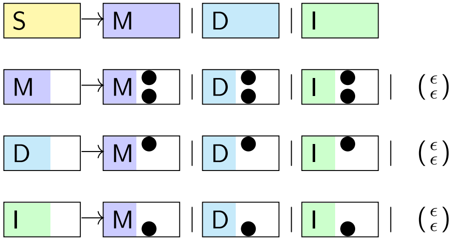
Functions all the Way Down

- from *domain-specific languages* to assembly
- multi-tape algorithms: “alignments”
- interlocking symbols: “pseudoknots”
- β-reduction, type-class resolution, and purity
- a comment on performance: “just a tiny bit more”
- developments in progress

The Maximum Likelihood & Forward Algorithm



The Maximum Likelihood & Forward Algorithm



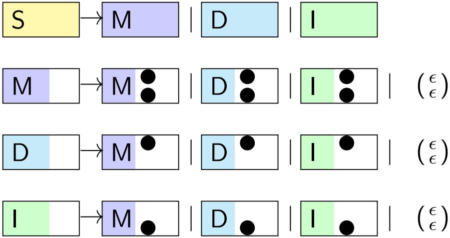
$$\begin{array}{lcccccccc}
S & \rightarrow & M & \mid & D & \mid & I & & \\
M & \rightarrow & M\binom{c}{d} & \mid & D\binom{c}{d} & \mid & I\binom{c}{d} & \mid & \binom{\varepsilon}{\varepsilon} \\
D & \rightarrow & M\binom{c}{-} & \mid & D\binom{c}{\cdot} & \mid & I\binom{c}{-} & \mid & \binom{\varepsilon}{\varepsilon} \\
I & \rightarrow & M\binom{-}{d} & \mid & D\binom{-}{d} & \mid & I\binom{\cdot}{d} & \mid & \binom{\varepsilon}{\varepsilon}
\end{array}$$

The Maximum Likelihood & Forward Algorithm

$$\begin{array}{lcl|c|c|c}
S_{m,n} & \rightarrow & M_{m,n} & D_{m,n} & I_{m,n} & \\
M_{i,j} & \rightarrow & M_{i-1,j-1}\left(\begin{smallmatrix} c_i \\ d_j \end{smallmatrix}\right) & D_{i-1,j-1}\left(\begin{smallmatrix} c_i \\ d_j \end{smallmatrix}\right) & I_{i-1,j-1}\left(\begin{smallmatrix} c_i \\ d_j \end{smallmatrix}\right) & \left(\begin{smallmatrix} \varepsilon_0 \\ \varepsilon_0 \end{smallmatrix}\right) \\
D_{i,j} & \rightarrow & M_{i-1,j}\left(\begin{smallmatrix} c_i \\ - \end{smallmatrix}\right) & D_{i-1,j}\left(\begin{smallmatrix} c_i \\ . \end{smallmatrix}\right) & I_{i-1,j}\left(\begin{smallmatrix} c_i \\ - \end{smallmatrix}\right) & \left(\begin{smallmatrix} \varepsilon_0 \\ \varepsilon_0 \end{smallmatrix}\right) \\
I_{i,j} & \rightarrow & M_{i,j-1}\left(\begin{smallmatrix} - \\ d_j \end{smallmatrix}\right) & D_{i,j-1}\left(\begin{smallmatrix} - \\ d_j \end{smallmatrix}\right) & I_{i,j-1}\left(\begin{smallmatrix} . \\ d_j \end{smallmatrix}\right) & \left(\begin{smallmatrix} \varepsilon_0 \\ \varepsilon_0 \end{smallmatrix}\right)
\end{array}$$

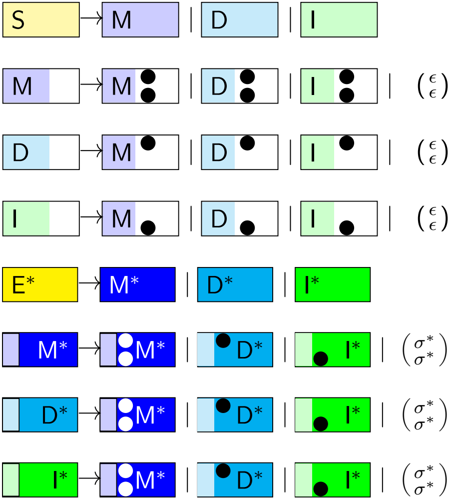
$$\begin{array}{lcl|c|c|c}
S & \rightarrow & M & D & I & \\
M & \rightarrow & M\left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right) & D\left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right) & I\left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right) & \left(\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}\right) \\
D & \rightarrow & M\left(\begin{smallmatrix} c \\ - \end{smallmatrix}\right) & D\left(\begin{smallmatrix} c \\ . \end{smallmatrix}\right) & I\left(\begin{smallmatrix} c \\ - \end{smallmatrix}\right) & \left(\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}\right) \\
I & \rightarrow & M\left(\begin{smallmatrix} - \\ d \end{smallmatrix}\right) & D\left(\begin{smallmatrix} - \\ d \end{smallmatrix}\right) & I\left(\begin{smallmatrix} . \\ d \end{smallmatrix}\right) & \left(\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}\right)
\end{array}$$

The Maximum Likelihood & Forward Algorithm

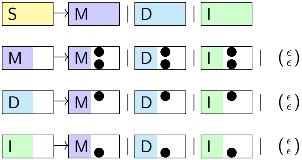


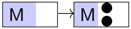
$$
\begin{array}{ccccccccc}
S & \rightarrow & M & | & D & | & I & & \\
M & \rightarrow & M\binom{c}{d} & | & D\binom{c}{d} & | & I\binom{c}{d} & | & \binom{\varepsilon}{\varepsilon} \\
D & \rightarrow & M\binom{c}{-} & | & D\binom{c}{\cdot} & | & I\binom{c}{-} & | & \binom{\varepsilon}{\varepsilon} \\
I & \rightarrow & M\binom{-}{d} & | & D\binom{-}{d} & | & I\binom{\cdot}{d} & | & \binom{\varepsilon}{\varepsilon}
\end{array}
$$

Inside-Outside: Probability of s_i and t_j Being Matched



Implementation Considerations



- order of tables S, M, D, I: (i) serial or parallel *table* filling, (ii) order of *cell* filling (M_{ij}, D_{ij} or D_{ij}, M_{ij})
- rule semantics: how to score 

- how to keep concerns (rules, semantics, behaviour/implementation of symbols) separate?
- how to write the grammar and algebra/semantics?

concern (i): the grammar

- have a simple domain-specific language
- for single- and multi-tape production rules
- is input-structure (e.g. strings, graphs, trees, . . .) generic
- can be extended: i.e. inclusion of grammar products is easy

(i): grammar DSL

```
Grammar: Gotoh
N: S M D I                decl. of syn.vars
T: c                      decl. of terminals
S: S                      start symbol

M -> match <<< M [c,c]    match rules
M -> match <<< D [c,c]
M -> match <<< I [c,c]
M -> empty <<<   [ε,ε]
D -> indel <<< M [c,-]    in/del rules
D -> idcnt <<< D [c,-]
D -> indel <<< I [c,-]
...
S -> M | D | I
//
```

(i): grammar function from DSL

```
1: monad to compute in
2: intermediate result type
3: final result type
4: input (T: c) type

                 1 2 3 4
gGotoh :: SigGotoh m x r c
       -> tM -> tD -> tI       table structures to fill
       -> v1 c                 first input sequence
       -> v2 c                 second input sequence
       -> Z :. Tabled tM (LimitType i -> i -> m r)
            :. Tabled tD (LimitType i -> i -> m r)
            :. Tabled tI (LimitType i -> i -> m r)
```

(i): grammar function weirdness

- `gGotoh` is (mostly) type-independent
- domain to optimize over is free: `x,r`
- input type `c` can be anything that fits into a vector `v1,v2`
- the DP tables can be freely chosen, using different memoization techniques (dense, sparse, none)
- even the index structure is free (`i`)

(ii): signatures combine grammars and algebras

```
data SigGotoh m x r c = SigGotoh
{ match :: x -> (Z:.c:.c) -> x
, indel :: x -> (Z:.c:.()) -> x
, idcnt :: x -> (Z:.c:.()) -> x
, ...
, h :: Stream m x -> m r
}
```

- automatically generated from the grammar DSL
- `h` collects all parses and chooses an optimal one
- parses are `Stream`s
- `Stream` is the `list` co-structure

concern (ii): the algebra

```
score :: Monad m => SigGotoh m Int Int Char
score = SigGotoh
{ match = \x (Z:.a:.b) -> if a==b then x+1 else x-2
, indel = \x (Z:.a:.()) -> x-5
, idcnt = \x (Z:.a:.()) -> x-1
, ...
, h = Stream.foldl' max minBound
}
```

- one just needs to describe the semantics, here maximally scoring alignments
- no indication of the recursive nature of the DP algorithm
- `Streams` behave just like lists of parses

concern (ii): semiring-generic algebras

```
semiringed :: (Monad m, Semiring s) => SigGotoh m s s Char
semiringed = SigGotoh
{ match = \x (Z:.a:.b) -> x ⊗ similarity a b
, indel = \x (Z:.a:.()) -> x ⊗ open a
, idcnt = \x (Z:.a:.()) -> x ⊗ cont a
, ...
, h = Stream.foldl' ⊕ mzero
}
```

- we can easily generalize to semiring-generic algebras
- *specialization* is then easy:

  ```
  score :: SigGotoh m (MaxPlus Int) (MaxPlus Int) Char
  score = semiringed
  ```

- `MaxPlus, Viterbi, ...` from *SciBaseTypes*

Semirings make it trivial to generate variants

Inside/Outside for posterior alignment probabilities

- requires probabilities: `Probability n k`
- on `Double`s for performance: `Probability n Double`
- without normalization: `Probability NotNorm Double`
- in the log-domain: `Log (Probability NotNorm Double)`
- fast and numerically stable!

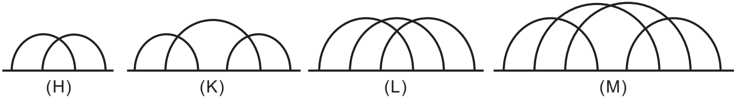
Discretized scoring systems (like in `RNAfold`):

- log-odds for scoring: `DiscLogOdds (k :% l)`
- in 1/100: `DiscLogOdds (1 :% 100)`
- choose the semiring: `MaxPlus (DiscLogOdds (1 :% 100))`
- fast and uses `Int` internally, not `Double`

Full Forward Algorithm

```
forward :: V Char -> V Char -> Z:.Tbl ... s:.Tbl ... s
forward inp1 inp2
  let n = length inp1
  let m = length inp2
  arrM, arrD, arrI <- newArray (Z:.n:.m) mzero
  fillTables $ gGotoh semiringed
    (ITbl @Id @Unboxed @0 @0 arrM)      bind table
    (ITbl @Id @Unboxed @0 @1 arrD)      set order of
    (ITbl @Id @Unboxed @0 @2 arrI)      filling
    (Chr inp1) (Chr inp2)               bind inputs
```

RNA Pseudoknots



$$
\begin{aligned}
I &\rightarrow S \mid T \\
S &\rightarrow (S)S \mid .S \mid \epsilon \\
T &\rightarrow I(T)S \\
T &\rightarrow IA_1 IB_1 IA_2 IB_2 S \\
T &\rightarrow IA_1 IB_1 IA_2 IC_1 IB_2 IC_2 S \\
T &\rightarrow IA_1 IB_1 IC_1 IA_2 IB_2 IC_2 S \\
T &\rightarrow IA_1 IB_1 IC_1 IA_2 ID_1 IB_2 IC_2 ID_2 S \\
\vec{X} &\rightarrow \begin{pmatrix} (_X I X_1 \\ X_2 I)_X \end{pmatrix} \mid \begin{pmatrix} (_X \\ )_X \end{pmatrix}
\end{aligned}
$$

(Reidys et al, 2011. Topology and prediction of RNA pseudoknots)

The Implementation

- Interleaved syntactic variables define alignment-style rules:

$$U \to \begin{pmatrix} S\,(\,U_1 \\ U_2\,S\,) \end{pmatrix}$$
$$U \to \begin{pmatrix} S \\ - \end{pmatrix}\begin{pmatrix} (\\ - \end{pmatrix}\begin{pmatrix} U_1 \\ U_2 \end{pmatrix}\begin{pmatrix} - \\ S \end{pmatrix}\begin{pmatrix} - \\ ) \end{pmatrix}$$

- They can be used atom-by-atom until all parts have been used:

$$S \to U_1 V_1 U_2 V_2$$

- This requires one new type constructor for partial symbols: $U_1 \dots U_{k-1}$
- and one for the final symbol: U_k
- ...a total of two new symbols to enable mcfgs

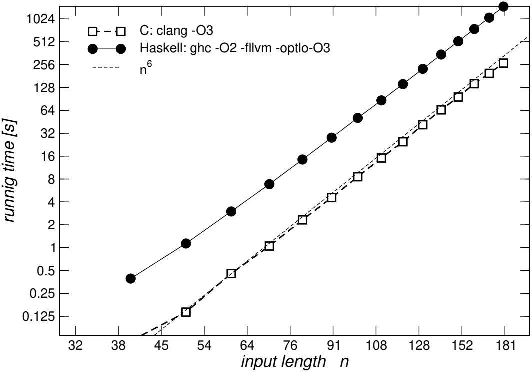
Performance

$$S \rightarrow (S)S \mid .S \mid \epsilon$$

$$S \rightarrow U_1 V_1 U_2 V_2$$

$$U \rightarrow \begin{pmatrix} S(U_1 \\ U_2 S) \end{pmatrix} \mid \begin{pmatrix} \epsilon \\ \epsilon \end{pmatrix}$$

$$V \rightarrow \begin{pmatrix} S[V_1 \\ V_2 S] \end{pmatrix} \mid \begin{pmatrix} \epsilon \\ \epsilon \end{pmatrix}$$



Algebraic Operations

$+$ Monoid $P_1^n + P_2^n = P_1^n \cup P_2^n$

$-$ Magma $P_1^n - P_2^n = \{p \in P_1^n | p \notin P_2^n\}$

$*$ power $G * n, n \in \mathbb{Z}$: symbols $\alpha \rightarrow \binom{\alpha}{\alpha}$

$\otimes$ Monoid $P_1^m \otimes P_2^n = \{(p_1 \otimes p_2)^{m+n} | p_1^m \in P_1^m, p_2^n \in P_2^n\}$

$p_1 \otimes p_2$

- $\binom{A_1}{A_2} \rightarrow \binom{B_1}{B_2}\binom{x_1}{x_2}$
- $\binom{A_1}{A_2} \rightarrow \binom{B_1}{\epsilon}\binom{x_1}{y_2}$
- $\binom{A_1}{A_2} \rightarrow \binom{y_1}{y_2}$
- $\binom{A_1}{\epsilon} \rightarrow \binom{B_1}{\epsilon}\binom{x_1}{\epsilon}$

Product Construction

$$\mathcal{S} = \{X \rightarrow Xa \mid X\}$$
$$\mathcal{N} = \{X \rightarrow \varepsilon\}$$
$$\mathcal{L} = \{X \rightarrow X\}$$

Product Construction

$$\begin{aligned}\mathcal{S} &= \{X \rightarrow Xa \mid X\}\\ \mathcal{N} &= \{X \rightarrow \varepsilon\}\\ \mathcal{L} &= \{X \rightarrow X\}\end{aligned}$$

$$\mathcal{NW} = \mathcal{S} \otimes \mathcal{S} + \mathcal{N} \otimes \mathcal{N} - \mathcal{L} \otimes \mathcal{L}$$

Product Construction

$$
\begin{aligned}
\mathcal{S} &= \{X \rightarrow Xa \mid X\} \\
\mathcal{N} &= \{X \rightarrow \varepsilon\} \\
\mathcal{L} &= \{X \rightarrow X\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{NW} &= \mathcal{S} \otimes \mathcal{S} + \mathcal{N} \otimes \mathcal{N} - \mathcal{L} \otimes \mathcal{L} \\
&= \binom{X}{X} \rightarrow \binom{X}{X}\binom{a}{a} \;\Big|\; \binom{X}{X}\binom{a}{-} \;\Big|\; \binom{X}{X}\binom{-}{a} \;\Big|\; \binom{X}{X} \\
&+ \binom{X}{X} \rightarrow \binom{\varepsilon}{\varepsilon} \\
&- \binom{X}{X} \rightarrow \binom{X}{X}
\end{aligned}
$$

Product Construction

$$
\begin{aligned}
\mathcal{S} &= \{X \rightarrow Xa \mid X\} \\
\mathcal{N} &= \{X \rightarrow \varepsilon\} \\
\mathcal{L} &= \{X \rightarrow X\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{NW} &= \mathcal{S} \otimes \mathcal{S} + \mathcal{N} \otimes \mathcal{N} - \mathcal{L} \otimes \mathcal{L} \\
&= \binom{X}{X} \rightarrow \binom{X}{X}\binom{a}{a} \mid \binom{X}{X}\binom{a}{-} \mid \binom{X}{X}\binom{-}{a} \mid \binom{X}{X} \\
&+ \binom{X}{X} \rightarrow \binom{\varepsilon}{\varepsilon} \\
&- \binom{X}{X} \rightarrow \binom{X}{X} \\
&= \binom{X}{X} \rightarrow \binom{X}{X}\binom{a}{a} \mid \binom{X}{X}\binom{a}{-} \mid \binom{X}{X}\binom{-}{a} \mid \binom{\varepsilon}{\varepsilon}
\end{aligned}
$$

Efficiency Concerns: Layers Upon Layers

- how efficient are DP algorithms with so many layers?
- optimizer example:

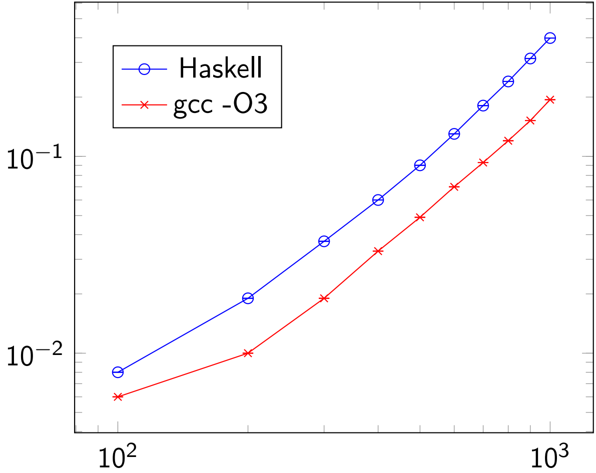
```
data Log a = Exp { ln :: a }
  data Prob norm a = Prob { unProb :: a }
    data Double = Double D#
```

- DP cell example:

```
foldl' ⊕ mzero
  (  match x (Z:.a:.b) <> indel x (Z:.a:.())
  <> delin x (Z:.():.b) <> epsilon (Z:.():.()) )
```

with `x`, say, the above `Log (Prob Notnorm Double)`

Performance



the *Haskell performance tax* is ≈ 2.0 in Needleman-Wunsch measuring tight-loop performance

Why?

- the best code is no code
- (i) move static things from the *value level* to the *type level*
- *type level* code is erased during compilation (no code!)
- (ii) use equational reasoning to replace code with equivalent, faster code

- *Stream Fusion: From Lists to Streams to Nothing at All*, Coutts, Leshchinskiy, Stewart
- *The HERMIT in the Stream*, Farmer, Höner zu Siederdissen, Gill
- *Sneaking Around `concatMap`*, Höner zu Siederdissen

β reduction

- application of function to an expression
- $f\ x = x * x$
- $f\ 7$ is replaced by $7 * 7$

Type Erasure

- Haskell removes all types during compilation
- minimal and maximal size of symbols known during compile time: e.g.

ADP	`region3 = region ‘with‘ (minSize 3)` where `minSize` *filters out* small regions
fusion	`region3 = String @0 @3 input` where `@3` is a type-level annotation (using `@`-syntax)

- type-level hints are compiled into constants that modify loop indices

Type Class Resolution

- type class instances are chosen based on types
- type-based recursion is completely unrolled during compilation

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a

instance Num Int where
  a + b = intPlus a b
  a * b = intMult a b

instance Num (Log a) where
  Exp a + Exp b = let h = max a b
                      l = min a b
      in Exp $ h + log1pexp (l - h)
  Exp a * Exp b = Exp $ a + b
```

Index constructions

- indices (and fusioned DP in general) make extensive use of inductive structures for multi-tape inputs
- an index of dimension 0 is denoted `Z` and operations in this dimension are trivial: they are *constant "null"* and removed during compilation
- a program with $i + 1$ dimensions is constructed using the inductive tuple (where `is` has dim. i):
  `data is :. i = is :. i`
- given an index structure for left-linear languages
  `PointL k = PointL Int`
- Needleman-Wunsch: `Z:.PointL k:.PointL k`

Index constructions

- Needleman-Wunsch `Z:.PointL k:.PointL k`
- HMMER-like: `Z:.StateIx k:.PointL k`
- ViennaRNA: `Subword Inside` for the usual RNAfold, and `Subword Outside` for McCaskill-like
- Infernal-like: `Z:.StateIx k:.Subword k`

```
ij = Z :. PointL 3 :. PointL 5
gh = Z :. LtPointL 8 :. LtPointL 9

(1) idx Z Z = 0
(2) size Z = 0
(3) idx (Lt g) (Pl i) = i
(4) size (Lt g) = g+1
(5) idx (g:.h) (i:.j) = idx g i * size h + idx h j
(6) size (g:.h) = size g * size h

    idx gh ij =
    idx (Z:.LtPL g:.Lt h) (Z:.PL i:.PL j) =
(5) idx (Z:.Lt g) (Z:.Pl i) * size (Lt h) + idx (Lt h) (Pl j) =
(3) idx (Z:.Lt g) (Z:.Pl i) * size (Lt h) + j =
(4) idx (Z:.Lt g) (Z:.Pl i) * (h+1) + j =
(5) (idx Z Z * size (Lt g) + idx (Lt g) (Pl i)) * (h+1) + j =
(3) (idx Z Z * size (Lt g) + i) * (h+1) + j =
(4) (idx Z Z * (g+1) + i) * (h+1) + j =
(1) (0 * (g+1) + i) * (h+1) + j =
(c) i * (h+1) + j
```

Importance in tight loops

```
streamUp tbl graAlg (Z:.g:.h) = loop (Z:.0:.0) where
  loop (Z:.i:.j) | i <= g && j <= h = do
    write tbl (Z:.i:.j) $ graAlg (Z:.i:.j)
    loop (Z:.i:.(j+1))
  loop (Z:.i:.j) | i <= g = loop (Z:.(i+1):.0)
  loop (Z:.i:.j) = pure ()
```

```
streamUp t g h = loop 0# 0# where
  loop i j | 1# <- i <=# g &&# j <=# h = do
    write t (i *# (h +# 1)) +# j (case
      ( the full body of grammar+algebra
        is completely inlined
        and applied to i j ) )
...
```

ADPfusion: Implementing a Parser Combinator

- choose or create a data type:
  `data Chr r x = Chr (f ::  V x → Int → r) (V x)`
- create an instance of `MkStream` parameterized over the data type, index type, and Inside/Outside grammar type:

```
instance MkStream (IStatic d) (ls:!:Chr r x) (PointL I)
  mkStream (ls:!:Chr f xs) u (PointL i) =
    map (\(S s ii ee) -> S s (ii:.i) (ee:.f xs i)
```

- Complexity depends on the chosen functionality, index type, and grammar type

Interlocking Symbols

- `sizeOf` is a type class function to find symbols with the same identifier
- the recursion is resolved during *compile time*
- as a result, during runtime, access is immediate (typically two register accesses)

```
...
IL -> iloop <<< ss P ss
...
iloop = \ls p rs -> p + log (length $ ls <> rs)
...
fillTables $ grammarVienna
  (AlgVienna ... iloop ...)
  (String @LeftRight input)     <- ls
  (String @LeftRight input)     <- rs
```

```
data String (ident :: String) xs = Str (V xs)

instance MkStream (sv:.IStatic d)
                  (ls:!:String ident xs)
                  (Subword I)
  mkStream Proxy (ls:!:Str xs) (us:.u) (is:.i)
    = flatten mk step . mkStream where
      mk = let sz = sizeOf (Proxy @ident) ls
           in  return (sz, ...)
```

For Robert

```
type instance LeftPosTy (OFirstLeft d)
                        (TwITbl b s m arr EmptyOk (PointL O)
                        (PointL O)
  = TypeError
  ( Text "OFirstLeft is illegal for outside tables.
         Check your grammars for multiple Outside syntactic
         variable on the r.h.s!")
```

Conclusion

- Common language for dynamic programming over strings, sets, trees, position-specific languages, and more
- Open: extension to new data types is easy!
- easy grammar construction using grammar products, and a shallow DSL
- Free (as in: no user work) outside algorithms for every CFG
- reasonably fast (slower by a factor of × 2–5 compared to *well optimized* `C`.)
- the *full host language* is available!

more general:

- Embedded DSL for (M)CFG's
- Compiler optimization based on rewriting rules

Future Work

large scale:

- Automatic derivation of rules for new data types
- Parameter estimation
- new compilation targets: fpga's, and gpu's