Dynamic programming using histomorphisms

Jevgeni Kabanov
Viinistu, 2005
CATAMORPHISM (FOLD)

• Structural recursion combinator

• Generic foldr (Haskell)

• Eats (folds) trees from bottom-up, producing combined result

• Similar to Visitor pattern in OOP, but doesn’t update structures
Sum fold animation

Let’s count this tree sum...
Sum fold animation
Sum fold animation
Sum fold animation
SUM FOLD ANIMATION
SUM FOLD ANIMATION
Sum fold animation
Sum fold animation
Sum fold animation
Sum fold animation
Sum fold animation
Sum fold animation
SUM FOLD ANIMATION
Sum fold animation
Sum fold animation

And the result is $16 = 1 + 5 + 3 + 7$
Histomorphism

- Introduced by Varmo & Tarmo in 1999
- Course-of-value structural recursion combinator
- Inspired by dynamic programming technique
- Moves bottom-up annotating the tree with results
- Allows to reuse sub(-sub)* node results
- Finally collapses the tree producing the end result
Funny sum histo animation

Let’s count this tree (funny) sum...
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation

```
1 -- 5
\|/ \|/ \|/
1 -- 3 -- 7
```

Funny sum histo animation

1 5 3 7
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation
Funny sum histo animation

$24$

And the result is $24 = 1 \times 2 + 5 + 3 + 7 \times 2$
Generic hylomorphism

- General recursion combinator
- 2 stages:
  1. Build an intermediate structure using unfold
  2. Collapse the intermediate structure using fold
- The intermediate structure corresponds to the implicit call tree
- The intermediate structure does not really have to be built
**Dynamic Hylomorphism**

- Dynamic recursion combinator
- The *fold* is replaced by the histomorphism

```
\[ F(A) \leftarrow \varphi \rightarrow A \]
\[ F(\mu F) \quad \text{in} \rightarrow \mu F \]
\[ F(\{()\psi\}) \rightarrow \psi \rightarrow B \]
CHALLENGES

• Histomorphism expressive power
• Dynamic hylomorphism expressive power
• Properties of transformation to dynamic recursion
• Deriving dynamic definition
Case study

• Fibonacci numbers
• Binary partition number
• Levenshtein (Edit) distance
• Longest common subsequence

• Only first two can be defined as pure histomorphisms
• General recursion is needed
Inspiration

Fibonacci dependency tree

Collapsed dependency graph
Inspiration (2)

Levenshtein (Edit) distance dependency tree

\[
\begin{align*}
D_{ij} & \\
D_{i-1j} & D_{i-1j-1} & D_{ij-1} \\
D_{i-2j} & D_{i-2j-1} & D_{i-1j-1} & D_{i-2j-2} & D_{i-1j-2} & D_{i-1j-1} & D_{i-1j-2} & D_{ij-2}
\end{align*}
\]
Inspiration (3)

Levenshtein (Edit) distance collapsed dependency graph
**Transformation**

- Original definition: \( f = \psi \circ Tf \circ \varphi \)
- Dynamic definition: \( f = \psi \circ \sigma \circ T'[\langle f, \text{in}^{-1}\rangle] \circ \varphi' \)
  - \( \varphi' \) generates more compact intermediate structure
  - \( T' \) defines the structure recursive pattern
  - \( \sigma \) restores one level of the old structure
  - \( \sigma \) and \( T' \) are uniquely determined by \( \varphi' \)
- The consumer (algebra) part is preserved
- The producer (coalgebra) part is consistently updated


**DEPENDENCY ALGEBRA**

Let

- Original dependency producers: \( h_i : A \rightarrow A \)
- Dynamic dependency producers: \( h'_j : A \rightarrow A \)
- Projections: \( \pi_i : T^\nu (C) \rightarrow T^\nu (C) \),
  \[
  \pi_i = [\text{in}, \text{out}_i \circ \text{outr}] \circ \text{in}^{-1}
  \]
- Deep projections:
  \[
  \pi_i^* = \text{outl} \circ \pi'_{k_l} \circ \pi'_{k_{l-1}} \circ \cdots \circ \pi'_{k_2} \circ \text{out}_{k_1}
  \]
- Induction indicator: \( p : A \rightarrow \text{Bool} \)
Dependency algebra (2)

Then

- \( \varphi = (id + \langle id, h_1, h_2, \ldots, h_n \rangle) \circ p? \)
- \( \varphi' = (id + \langle id, h'_1, h'_2, \ldots, h'_m \rangle) \circ p'? \)
- \( \sigma = [\text{inl}, (out_0 + \langle out_0, \pi^*_1, \pi^*_2, \ldots, \pi^*_n \rangle) \circ (p \circ out_0)?] \)

And \( \varphi' \) has to satisfy following for each \( i \in I \), each \( s \in S' \):

\[
P(s, i) = \langle k_1, k_2, \ldots, k_l \rangle \in J^*
\]

\[
\text{outl} \circ \pi_i \circ [(\langle id, \varphi \rangle)] = \text{outl} \circ \pi'_{k_l} \circ \pi'_{k_{l-1}} \circ \cdots \circ \pi'_{k_1} \circ [(\langle id, \varphi' \rangle)]
\]

\[
h_i(s) = h'_{k_l} \circ h'_{k_{l-1}} \circ \cdots \circ h'_{k_1}(s)
\]
Future work

• More categorical approach to transformation

• A solid proof for dependency algebra

• (Semi)-automatical derivation for restricted cases