A Haskell EDSL for (Parallel) Programming in Generate-Test-and-Aggregate Style

Sebastian Fischer
Outline

1. Generate, Test, and Aggregate
2. Parallel Programming
3. A Haskell EDSL
4. Test-case Generation?
Generate, Test, and Aggregate

- Like generate-and-test
  but with aggregate

```haskell
data Item = Item { value :: Value, weight :: Weight }

knapsack :: Weight -> [Item] -> Value

knapsack maxW
  = maximum . map (sum . map value)
  . filter ((\leq maxW). sum . map weight)
  . sublists
```
Generators
generates a bag of lists

Tests
discards some of those lists

Aggregators
reduces every list to single value
reduces bag of values to final result
Complexity

sublists \([1,2,3]\) = \(\{[3],[1,3],[2,3],[1,2,3]\}\)

size of sublists \(\ell\) is \(2^{\text{length } \ell}\)

run time of knapsack with \(\ell\) is \(O(2^{\text{length } \ell})\)
Generators

only allowed to use certain operations
(polymorphous over some type class)

Test

by structural recursion on lists

Aggregator

by structural recursion on bag of lists
class Semiring $s$ where

$$\text{two}, \text{one} :: s$$

$$(\oplus), (\otimes) :: s \rightarrow s \rightarrow s$$

Laws

$$\text{two} \oplus x = x \oplus \text{two} = x$$

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

$$x \otimes y = y \oplus x$$

$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$$

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$

$$\text{two} \otimes x = x \otimes \text{two} = \text{two}$$
instance Semiring $[[a]]S$ where

\[\begin{align*}
\text{zero} & = \emptyset \\
\text{one} & = \{\} \\
\end{align*}\]

\[x \oplus y = x \oplus y\] -- bag union

\[x \otimes y = \{xs ++ ys \mid xs \leftarrow x, \ ys \leftarrow y\} \]  

**Example**

\[
(\{[1],[2]\} \oplus \{[2],[3]\}) \otimes \{[3]\} = \{[1],[2],[3]\} \otimes \{[3]\} = \{[2,3], [1,3], [1,2,3]\}
\]
Another Instance

\[
\begin{align*}
\text{instance} & \text{ Semiring Value where} \\
\text{zero} & = -\infty \\
\text{one} & = 0 \\
\text{x} \odot \text{y} & = \max \text{x} \times \text{y} \\
\text{x} \ominus \text{y} & = \text{x} + \text{y}
\end{align*}
\]

Type of Aggregators

maxValue :: [[ItemJS] -> Value
**Restricted Generator**

\[
\text{sublists} :: [a] \rightarrow \mathcal{P}[a] \mathcal{S}
\]

\[
\text{sublists } l = \text{genSublists} \ (\lambda x \rightarrow \mathcal{P}[x] \mathcal{S}) \ l
\]

\[
\text{genSublists} :: \text{Semiring } s
\]

\[
\Rightarrow (a \rightarrow s) \rightarrow [a] \rightarrow s
\]

\[
\text{genSublists } \_ \ [ ] = \text{one}
\]

\[
\text{genSublists } \_ \ f \ (x : xs) =
\]

\[
(\text{one} \otimes f \ x) \otimes \text{genSublists } \_ \ f \ xs
\]
Fusion

\[ \text{maxValue . genSublists (lx -> [x][x])} \]

\[ = \text{genSublists (lx -> maxValue [x][x])} \]

\[ = \text{genSublists (lx -> value x)} \]

Linear rather than exponential run-time complexity
Generate, Test, and Aggregate
- intuitive, inefficient specification
- certain conditions allow fusion
  (not shown: fuse aggregators and filters)
- efficient implementation can be automatically derived
Parallel Programming

Divide and Conquer

Structural recursion over monoid structure of lists

\[ \text{sum } [x] = x \]
\[ \text{sum } (xs + ys) = \text{sum } xs + \text{sum } ys \]
Parallel Sublist Generator

\[ \text{genSublists} :: \text{Semiring} \ s \Rightarrow (a \to s) \to [a] \to s \]

\[ \text{genSublists} \ [] = \text{one} \]

\[ \text{genSublists} \ f \ [x] = \text{one} \oplus f \times \]

\[ \text{genSublists} \ f \ (xs ++ ys) = \text{genSublists} \ xs \oplus \text{genSublists} \ ys \]

\[ \Rightarrow \text{efficient, parallel knapsack function} \]
How to obtain efficient implementation from specification automatically?

GHC RULES pragma?

more reliable way?
Haskell EDSL

\[
\text{generate} :: (\forall s \cdot \text{Semiring} s \Rightarrow (a \to s) \to [a] \to s) \\
\Rightarrow [a] \to \text{Gen} a
\]

\[
\text{test} :: \text{Monoid} m \\
\Rightarrow (m \to \text{Bool}) \to (a \to m) \to \text{Gen} a \to \text{Gen} a
\]

\[
\text{aggregate} :: \text{Semiring} s \\
\Rightarrow (a \to s) \Rightarrow \text{Gen} a \Rightarrow s
\]
Implementation

data  Gen a where

  Generate :: (∀s. Semirings ⇒ (a ⇒ s) ⇒ [a] ⇒ s) ⇒ [a] ⇒ Gen a

  Test :: Monoid m ⇒ (m ⇒ Bool) ⇒ (a ⇒ m) ⇒ Gen a ⇒ Gen a

generate = Generate

test   = Test
Implementation of Fusion

\[
\text{aggregate} :: \text{Semiring } s \\
\Rightarrow (a \to s) \to \text{Gen } a \to s
\]

\[
\text{aggregate } f \text{ (Generate } \text{ gen } \text{ e) } = \text{ gen } f \text{ e}
\]

\[
\text{aggregate } f \text{ (Test ok h gen) } = \ldots
\]

**Static guarantee:**

no intermediate bags
Efficient knapsack function

\[ \text{knapsack} :: \text{Weight} \rightarrow [\text{Item}] \rightarrow \text{Value} \]

\[ \text{knapsack maxW} = \text{aggregate value} \]

. test \((\leq \text{maxW}), \text{getSum}) \text{Sum}\]

. generate genSubLists

Looks like exponential specification
Summary

- Certain Gen-Test-Log algorithms can be implemented efficiently
- Parallelism orthogonal issue (depends on generators)
  -> Siskus do not destroy parallelism
- Implementation as Haskell EDSL strikingly simple
Test-case Generation?
presented technique not restricted to lists
test can express (some) preconditions
aggregators can construct remaining test cases
complexity depends on number of results
not on number of discarded values (?)