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

```
data Item = Item {value :: Value, weight :: Weight}
knapsack :: Weight -> [Item] -> Value
knapsack maxW
  = maximum . map (sum . map value)
  . filter ((≤ maxW) . sum . map weight)
  . sublists
```

Generator

generates a bag of lists

Test

discards some of those lists

Aggregator

reduces every list to single value

reduces bag of values to final result

Complexity

sublists [1,2,3] =
{[ ], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]}

size of sublists l is $2^{\text{length } l}$

run time of knapsack maxW l is

$O(2^{\text{length } l})$

Generators

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

Test

by structural recursion on lists

Aggregator

by structural recursion on bag of lists

```
class Semiring s where
    zero, one :: s
    (⊕), (⊗) :: s -> s -> s
```

Laws

$$\text{zero} \oplus x = x \oplus \text{zero} = x \qquad \text{one} \otimes x = x \otimes \text{one} = x$$

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

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

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

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

$$\text{zero} \otimes x = x \otimes \text{zero} = \text{zero}$$

instance Semiring {[a]} where

$$\text{zero} = \{\}$$

$$\text{one} = \{[\,]\}$$

$$x \oplus y = x \uplus y \quad \text{-- bag union}$$

$$x \otimes y = \{ xs \mathbin{+\!\!+} ys \mid xs \leftarrow x, ys \leftarrow y \}$$

Example

$$(\{[1]\} \oplus \{[2]\}) \otimes \{[3]\}$$

$$= \{[1],[2]\} \otimes \{[3]\} = \{[1,3],[2,3]\}$$

Another Instance

```
instance Semiring Value where
  zero = -∞
  one  = 0

  x ⊕ y = max x y
  x ⊗ y = x + y
```

Type of Aggregator

```
maxValue :: [[Item]] -> Value
```

Restricted Generator

```
sublists :: [a] -> {[a]}
sublists l = genSublists (\x -> {[x]}) l

genSublists :: Semiring s
            => (a -> s) -> [a] -> s

genSublists _ []     = one
genSublists f (x:xs) =
    (one ⊕ f x) ⊗ genSublists f xs
```

Fusion

$$\begin{aligned}
& \text{maxValue} \;.\; \text{genSublists}\ (\backslash x \to \{[x]\}) \\
= \;& \text{genSublists}\ (\backslash x \to \text{maxValue}\ \{[x]\}) \\
= \;& \text{genSublists}\ (\backslash x \to \text{value}\ x)
\end{aligned}$$

Linear rather than exponential run-time complexity

Generate, Test, and Aggregate

- intuitive, inefficient specification
- certain conditions allow fusion
  (not shown: fuse aggregator and filter)
- efficient implementation
  can be automatically derived

Parallel Programming

Divide and Conquer

Structural recursion
over monoid structure of lists

$$
\begin{aligned}
\text{sum } [\,] &= 0 \\
\text{sum } [x] &= x \\
\text{sum } (xs +\!\!+ ys) &= \text{sum } xs + \text{sum } ys
\end{aligned}
$$

Parallel Sublist Generator

```
genSublists :: Semiring s
            => (a -> s) -> [a] -> s
genSublists _ []          = one
genSublists f [x]         = one ⊕ f x
genSublists f (xs ++ ys) =
    genSublists xs ⊗ genSublists ys
```

→ efficient, parallel knapsack function

How to obtain efficient implementation from specification automatically?

GHC RULES pragma?

more reliable way?

Haskell EDSL

```
generate :: (∀s. Semiring s => (a -> s) -> [a] -> s)
         -> [a] -> Gen a
```

```
test :: Monoid m
     => (m -> Bool) -> (a -> m) -> Gen a -> Gen a
```

```
aggregate :: Semiring s
          => (a -> s) -> Gen a -> s
```

Implementation

```
data Gen a where
  Generate :: (∀s. Semiring s => (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

```
aggregate :: Semiring s
          => (a -> s) -> Gen a -> s
aggregate f (Generate gen l) = gen f l
aggregate f (Test ok h gen) = ...
```

static guarantee:

no intermediate bags

Efficient knapsack function

```
knapsack :: Weight -> [Item] -> Value
knapsack maxW = aggregate value
                . test ((≤maxW).getSum) Sum
                . generate genSublists
```

looks like exponential specification

Summary

- certain Gen-Test-Agg Algorithms can be implemented efficiently
- parallelism orthogonal issue (depends on generators)
  → filters do not destroy parallelism
- implementation as Haskell EDSL strikingly simple

Test-case Generation ?

presented technique not restricted to lists

test can express (some) preconditions

aggregator can construct remaining test cases

complexity depends on number of <u>results</u>

not on number of discarded values (?)