Master's Thesis

Incremental Cata Computation for Generic Data Types

Jort van Gorkum
Computing Science - Programming Technology

Supervisors:
Dr. Wouter Swierstra, Dr. Trevor McDonell

April 5, 2022

1 Experiments

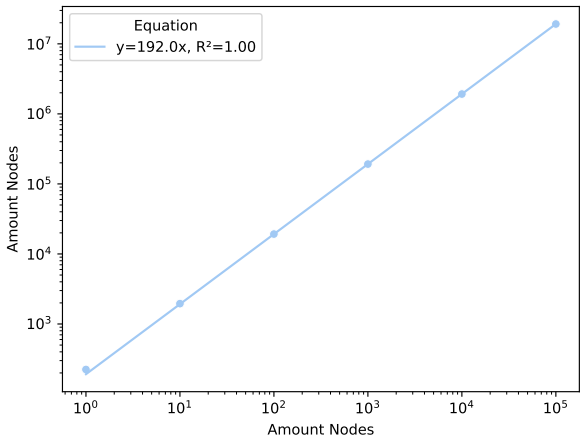

Figure 1: Bytes allocated for Cata Sum

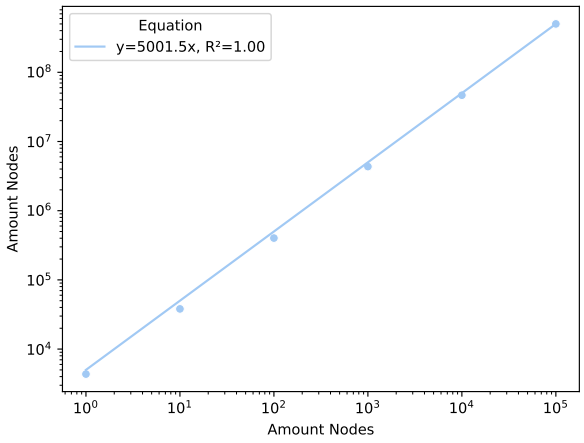

Figure 2: Bytes allocated for Generic Cata Sum

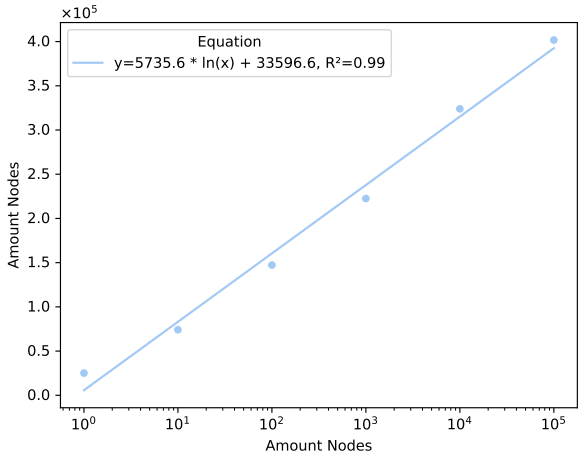

Figure 3: Bytes allocated for Incremental Cata Sum

2 Appendix

A Definition Generic Datatypes

```
data U r          = U
data I r          = I r
data K a r        = K a
data (:+:) f g r = L (f r) | R (g r)
data (:*:) f g r = (f r) :*: (g r)
data C c f r      = C (f r)

newtype Fix f = In { out :: f (Fix f) }
```

B Implementation Hashable

```
class Hashable f where
  hash :: f (Fix (g :*: K Digest)) -> Digest

instance Hashable U where
  hash _ = digest "U"

instance (Show a) => Hashable (K a) where
  hash (K x) = digestConcat [digest "K", digest x]

instance Hashable I where
  hash (I x) = digestConcat [digest "I", getDigest x]
    where
      getDigest :: Fix (f :*: K Digest) -> Digest
      getDigest (In (_ :*: K h)) = h

instance (Hashable f, Hashable g) => Hashable (f :+: g) where
  hash (L x) = digestConcat [digest "L", hash x]
  hash (R x) = digestConcat [digest "R", hash x]

instance (Hashable f, Hashable g) => Hashable (f :*: g) where
  hash (x :*: y) = digestConcat [digest "P", hash x, hash y]

instance (Hashable f) => Hashable (C c f) where
```

```
  hash (C x) = digestConcat [digest "C", hash x]
```

C Implementation Merkle

```
type Merkle f = Fix (f :*: K Digest)

merkleG :: Hashable f
        => f (Fix (g :*: K Digest))
        -> (f :*: K Digest) (Fix (g :*: K Digest))
merkleG f = f :*: K (hash f)

merkle :: (Regular a, Hashable (PF a), Functor (PF a))
       => a -> Merkle (PF a)
merkle = In . merkleG . fmap merkle . from
```

D Implementation Cata Merkle

```
cataMerkleState :: (Functor f, Traversable f)
                => (f a -> a) -> Fix (f :*: K Digest)
                -> State (M.Map Digest a) a
cataMerkleState alg (In (x :*: K h)) = do m <- get
  case M.lookup h m of
    Just a -> return a
    Nothing -> do y <- mapM (cataMerkleState alg) x
                  let r = alg y
                  modify (M.insert h r) >> return r

cataMerkle :: (Functor f, Traversable f)
           => (f a -> a) -> Fix (f :*: K Digest) -> (a, M.Map Digest a)
cataMerkle alg t = runState (cataMerkleState alg t) M.empty
```

E Implementation Zipper Merkle

```
data Loc :: * -> * where
  Loc :: (Zipper a) => Merkle a
                    -> [Ctx (a :*: K Digest) (Merkle a)]
                    -> Loc (Merkle a)
```

```
modify :: (a -> a) -> Loc a -> Loc a
modify f (Loc x cs) = Loc (f x) cs

updateDigest :: Hashable a => Merkle a -> Merkle a
updateDigest (In (x :*: _)) = In (merkleG x)

updateParents :: Hashable a => Loc (Merkle a) -> Loc (Merkle a)
updateParents (Loc x []) = Loc (updateDigest x) []
updateParents (Loc x cs) = updateParents
                         $ expectJust "Exception: Cannot go up"
                         $ up (Loc (updateDigest x) cs)

updateLoc :: Hashable a => (Merkle a -> Merkle a)
                        -> Loc (Merkle a) -> Loc (Merkle a)
updateLoc f loc = if   top loc'
                  then loc'
                  else updateParents
                       $ expectJust "Exception: Cannot go up" (up loc')
  where
    loc' = modify f loc
```