An EDSL for KDB/Q

rationale, techniques and lessons learned

Tim Williams | October 2017

Standard Chartered

An EDSL for KDB/Q

What is KDB/Q?

KDB/Q is an array processing language used for programming the proprietary KDB+ columnar database by Kx systems

- KDB is commonly used in the finance industry for time-series applications
- Q is dynamically typed, famously terse

An EDSL for KDB/Q

Problem

We have a significant amount of Haskell logic that needs porting to KDB/Q, which is made especially difficult by incompatible syntax and semantics*

*We will spare you from having to read much KDB/Q code in this talk!

An EDSL for KDB/Q

Solution

- Haskell is expressive enough to enable the composition of Q programs within Haskell itself, using a (deeply) embedded domain specific language (EDSL)
- EDSLs should be cheaper to build and maintain than more traditional approaches to code generation.

We will also apply some Category Theory!

Standard Chartered

EDSL Rationale

- Haskell syntax
  - lexical scoping
  - standard operator precedence rules
- Choice of semantics
  - static types
  - referential transparency
  - null safety
  - IEEE-754 compliant operators
  - no expression size limits

EDSL Rationale

- The EDSL uses types to document interfaces and machine-check correctness
- Evaluate Q programs using Haskell or using KDB
  - KDB requires a license per machine
- Mix Q programs with Haskell code inside the same file
  - invaluable for testing
- A safe and restricted subset of Q
  - For example, we can offer termination guarantees

EDSL Rationale

An (easy) subset of Q

- The EDSL here is only concerned with composing *scalar* operations, which may or may not be applied to bulk data within KDB.
- Giving static types to bulk operations or queries, is a much harder problem and still an area of ongoing research†

---

† Modern Haskell is certainly capable of tackling this. For example, giving types to the relational algebra [1] and implicit lifting of scalar operations into bulk operations using rank polymorphism [2].

Standard Chartered

Key Features

- The front end syntax has both expressions and statements
  - side-effecting primitives are primitive monadic instructions
  - differentiate between pure functions and procedures
  - pure functions exploited during optimisation
- Both explicit sharing and implicit (recovered) sharing
  - affords some manual control
  - non-trivial to preserve evaluation semantics in the presence of side-effects
- No attempt at overloading syntax for shallow/deep polymorphism

Examples

The EDSL inherits Haskell's syntax and operator precedence rules, which can significantly simplify mathematical expressions:

EDSL

```
f (x, y, z) = 2*x + 3*y < 4*z
```

Q

```
f:{[x; y; z] ((2*x) + (3*y)) < (4*z)};
```

Examples

Haskell's record syntax makes it easier to construct composite data:

EDSL

```
toQ Params
    { pCcy    = KRW
    , pSpread = 0.5
    , pLo     = 50
    , pHi     = 80
    }
```

Q

```
‘pCcy‘pSpread‘pLo‘pHi!(‘KRW;0.5;10f;20f);
```

Standard Chartered

Examples

Records are declared, which document and guarantee the presence of fields:

```
data Result = Result
  { rPrice :: Double
  , rDate  :: Datetime
  }
$deriveView ’’Result

scalePrice :: Q Double -> Q Result -> Q Result
scalePrice x = modL rPriceL (*x) -- Note: x is captured
```

Standard Chartered

Examples

Sum-types are useful to document and guarantee the handling of options.
Enums are a special-case, which are handled and represented separately:

EDSL

```
data ABC = A | B | C

f :: Q ABC -> Q Int
f x = switch x [ A --> 1
               , B --> 2
               , C --> 3
               ]
```

Q

```
f:{[x] $[ x~‘A; 1; x~‘B; 2; x~‘C; 4; ’impossible]};
```

Standard Chartered

Examples

Arbitrary sum types are embedded using fold functions generated using Template Haskell:

```
data Either a b = Left a | Right b
$deriveElim ''Either

either
    :: (QTy a, QTy b, QTy r)
    => (Q a -> Q r)
    -> (Q b -> Q r)
    -> Q (Either a b)
    -> Q r
either f g e = elim e f g
```

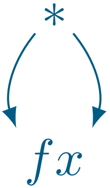
Standard Chartered

Examples

Sharing can be made explicit, using the `letQ` primitive:

```
letQ :: (QTy a, QTy b) => Q a -> (Q a -> Q b) -> Q b
```

```
letQ (f x) $ \y ->
    y*y
```



Examples

Impure code, such as code that use mutable references, has a monad:

```
-- | returns 6
impure :: QProg Int
impure = do
    r <- newRef 0
    mapM_ (f r) [1, 2, 3]
    readRef r
  where
    f :: Q (Ref Int) -> Q Int -> QProg ()
    f r x = modifyRef r (+x)
```

Techniques

Deep Embeddings

- A deeply embedded DSL yields an abstract-syntax-tree (AST) upon evaluation
- We can then analyse, optimise and compile the AST as is necessary

```
{-# LANGUAGE GADTs #-}
data Q :: * -> * where
  QVar  :: QTy a => Var    -> Q a
  QAtom :: QTy a => Atom a -> Q a
  QLam  :: (QTy a, QTy b) => (Q a -> Q b) -> Q (a -> b)
  QApp  :: (QTy a, QTy b) => Q (a -> b) -> Q a -> Q b
  ...
```

Standard Chartered

Overloading

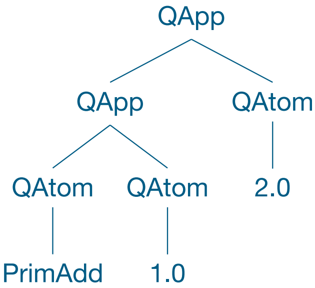
Haskell's type classes permit expressive adhoc overloading, making it possible to achieve a deep embedding without too much syntactic noise

```
instance Num a => Num (Q a) where
  (+) x y = QApp (QApp (QAtom PrimAdd) x) y
  fromInteger = QAtom . ADbl . fromInteger

instance Fractional a => Fractional (Q a) where
  fromRational = QAtom . ADbl . fromRational
```

Overloading

```
λ> 1 + 2 :: Q Double
QApp (QApp (QAtom PrimAdd) (QAtom 1.0)) (QAtom 2.0)
```



Higher-order abstract syntax

- Re-uses abstraction and binding from the host language
- HOAS is useful to reify functions in embedded programs
- GADTs can be used to preserve type information
- Beware of *exotic terms*‡

```
{-# LANGUAGE GADTs #-}
data Q :: * -> * where
    QLam  :: (QTy a, QTy b) => (Q a -> Q b) -> Q (a -> b)
    QVar  :: QTy a => Id -> Q a  -- ^ to convert out of HOAS
    ...
```

---

‡We must not perform case analysis on types used as inputs to a binding function!

Standard Chartered

Sequencing effects

We use a Monad in the EDSL in order to sequence side effects and support mutable references

```
type QProg a = Prog Stmt (Q a)

data Stmt :: * -> * where
    -- References
    NewRef   :: Q a -> Stmt (Q (Ref a))
    ReadRef  :: Q (Ref a) -> Stmt (Q a)
    WriteRef :: Q (Ref a) -> Q a -> Stmt (Q ())
    ...
```

Operational Monad

The *Operational* package allows us to reify monads, similarly to a Free Monad, but with better asymptotics [3]

```
data Prog ins a where
  Return :: a -> Prog ins a
  (:>>=) :: Prog ins a -> (a -> Prog ins b) -> Prog ins b
  instr  :: ins (Prog ins) a -> Prog ins a

instance Monad (Prog ins) where
  return = Return
  (>>=)  = :>>=
```

Standard Chartered

Meta-programming

Meta-programming in the EDSL is achieved just by using functions in the host language

```
Q (a -> b)  -- ^ embedded function
Q a -> Q b  -- ^ meta-function
```

Meta-programming

Lenses derived using template haskell

```
priceBidL    :: Q Price :-> Q Double
resultPriceL :: Q Result :-> Q Price
```

Lens computations are meta-programs which are computed at staging-time

```
getL :: (f :-> a) -> f -> a
setL :: (f :-> a) -> a -> f -> f
compose :: (b :-> c) -> (a :-> b) -> (a :-> c)
```

Meta-programming

The Reader monad can be used as a meta-program to thread values through without any runtime cost

```haskell
type QProgR r a = ReaderT (Q r) (Prog Stmt) (Q a)

runReaderT :: ReaderT r m a -> r -> m a
```

Standard Chartered

Dynamic types

- Often need to deal with untyped data at the interface boundaries
- Use a *Dynamic* wrapper type to contain these untrusted values
- Unpacking the dynamic value forces a runtime type check

```
data Dynamic

class QTy a => HasDynamic a where
    pack   :: Q a -> Q Dynamic
    unpack :: Q Dynamic -> Q (Maybe a)
```

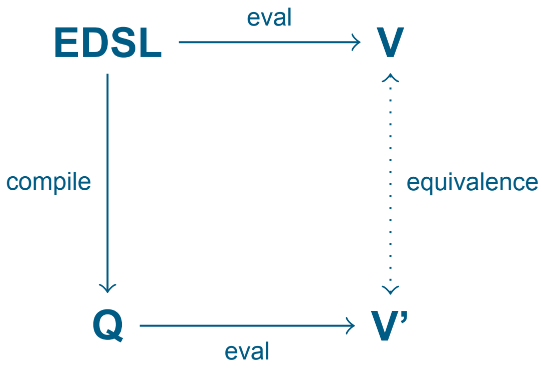
QuickCheck

- Use QuickCheck to generate and interpret random expressions
- Test for properties that must hold over the results
- Build an evaluator for the DSL and use it to verify the assumed semantics and compilation output

Standard Chartered

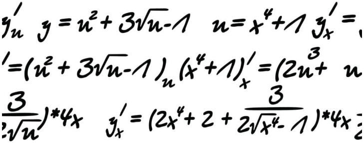
QuickCheck

Using an evaluator and the compiled output, we perform a 2-way comparison:



Generating test expressions

- Generating expressions of arbitrary type difficult
  - requires constraint solving
- But very easy to do if we limit the types. For example:
  - double arithmetic (with infinities, NaNs and zeros)
  - boolean algebra
  - list operations
  - dictionary operations



Embedding Algebraic Data Types

A type class defines which types can be embedded into a Q expression:

```
class QTy a where
  toQ  :: a -> Q a

-- An example Q encoding for a sum type
instance QTy a => QTy (Maybe a) where
    toQ (Just x) = variant ”Just”    (toQ x)
    toQ Nothing  = variant ”Nothing” unit

-- An example encoding for a record
instance QTy Point where
    toQ (Point x y) = record [ (”x”, toQ d1)
                             , (”y”, toQ d2)
                             ]
```

Standard Chartered

Views

A “View” type class allows us to use pattern matching for product types [4]:

```
-- | for pattern-matching on tuples and records
class QTy a => View a where
    type Rep a
    toView   :: Q a -> Rep a
    fromView :: Rep a -> Q a
```

This works well when combined with the “ViewPatterns” GHC extension:

```
swap :: Q (a, b) -> Q (b, a)
swap (toView -> (a, b)) = fromView (b, a)
```

Template Haskell is used to generate instances for arbitrary records.

Eliminators

An "Elim" type class allows us to eliminate sum-types, as one normally would using case analysis [4]:

```
-- | for folding/eliminating data-types
class QTy a => Elim a r where
    type Eliminator a r
    elim :: Q a -> Eliminator a r
```

The instance for `forall a. Maybe a` is as follows:

```
instance (QTy a, QCond r) => Elim (Maybe a) r where
    type Eliminator (Maybe a) r = r -> (Q a -> r) -> r
    elim ma b f = cond (isNothing ma) b $ f (fromJust ma)
```

Template Haskell is used to generate instances for arbitrary sum types

Standard Chartered

Closure conversion

Problems

- We need to port a significant amount of Haskell code to the EDSL that makes heavy use of lexical scoping and closures, which Q does not support
- Q has expression size limits for branches of a conditional, which is most easily worked around by *eta-expansion* and lambda-lifting

Solution

- Transform the AST to remove any lexically captured variables

Closure conversion

Luckily, Q does support partial application, so we can employ a very simple conversion to close all “open” lambdas containing free-variables:

- calculate the free variables bottom-up
- add the captured variables to the parameter lists and partially apply the additional arguments

Closure conversion

We have

```
f = \x -> \y -> x + y         -- ^ not supported in Q
```

We want

```
f = \x -> (\x y -> x + y) x  -- ^ supported in Q
```

Closure conversion

Problem

- How can we achieve separation-of-concerns without nested folds?
- How can we avoid specifying every case?

```
-- WARNING: This has quadratic complexity!
closeExpr :: QExpr -> QExpr
closeExpr (QLam vs e) =
    let e' = closeExpr e
        vs' = Set.toList $ freeVars e' \\ (Set.fromList vs)
    in QApply (QLam (vs' ++ vs) e') vs'
...
freeVars :: QExpr -> Set Var
```

Standard Chartered

Solution

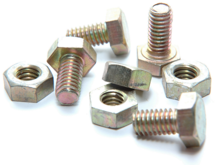
Use Functor fixed-points and recursion schemes!

- Add principled structure to our traversals
- Achieve compositional data-types and traversal code
- Avoid boilerplate traversal code using Foldable and Traversable

Fixed points of Functors

An idea from category theory which gives:

- data-type generic traversals
- compositional data-types
- especially useful for annotations and recovering sharing



```
-- | the least fixpoint of functor f
newtype Fix f = Fix { unFix :: f (Fix f) }
```

A functor `f` is a data-type of kind `* -> *` together with an `fmap` function.

$$Fix\ f \cong f(f(f(f(f...etc$$

Catamorphism

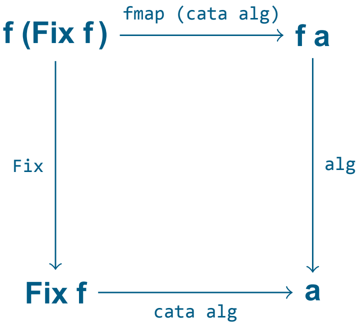
A *catamorphism* (cata meaning “downwards”) is a generalisation of the concept of a fold [5,6]

- models the fundamental pattern of (internal) *iteration*
- a catamorphism will traverse bottom-up, however top-down or a combination is possible using a function codomain
- category theory shows us how to define it data-type generically for a functor fixed-point

```
cata :: Functor f => (f a -> a) -> Fix f -> a
```

Catamorphism

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```



Closure conversion

Pattern Functor AST

```
type QExpr = Fix QExprF

data QExprF r
    = QVar  Var
    | QPrim PrimOp
    | QAtom Atom
    | QLam  [Name] r
    | QApp  r r
    | ...
```

Closure conversion

We will use a *zygomorphism* to factor out the free variable calculation as an auxiliary algebra

```
closeExpr :: QExpr -> QExpr
closeExpr = zygo fvsAlg mainAlg

mainAlg :: QExprF (QExpr, Set Var) -> QExpr
fvsAlg  :: QExprF (Set Var) -> Set Var

-- | semi-mutual recursion
zygo :: Functor f =>
        (f b -> b) -> (f (a, b) -> a) -> Fix f -> a
```

Standard Chartered

Zygomorphism

A zygomorphism just adds additional structure to a catamorphism

```
-- | semi-mutual recursion
zygo :: Functor f =>
        (f b -> b) -> (f (a, b) -> a) -> Fix f -> a
zygo f g = fst . cata (algZygo f g)

algZygo :: Functor f =>
    (f  b     -> b) ->
    (f (a, b) -> a) ->
    f (a, b) -> (a, b)
algZygo f g = g &&& f . fmap snd
```

Standard Chartered

Closure conversion

We have $O(n)$ complexity, separation of concerns and minimal boilerplate

```
-- | close all lambdas
mainAlg :: QExprF (QExpr, Set Var) -> QExpr
mainAlg (QLam vs (e, fvs)) =
    let vs’ = Set.toList $ fvs \\ (Set.fromList vs)
    in Fix $ QApply (Fix $ QLam (vs’ ++ vs) e) vs’
mainAlg e = Fix e

-- | gather free variables
fvsAlg :: QExprF (Set Var) -> Set Var
fvsAlg (QVar v)     = Set.singleton v
fvsAlg (QLam vs e) = (fold e) \\ (Set.fromList vs)
fvsAlg e            = fold e
```

Standard Chartered

Closure conversion

Problem

- Q has a limit of only 8 function parameters.
  Therefore we cannot simply add each captured variable as a new parameter, we will soon hit this limit

Solution

- Pass and extend a single environment, a linked-list of frames
- Add an environment identifier to each parameter list and partially apply the functions with an appropriately extended environment
- Rewrite any free variable references to index into this environment

Closure conversion

The main algebra now needs to produce a function, which when called with an initial environment, will traverse top-down passing and extending it as necessary

```
type Env  = Map Id Path

mainAlg :: QExprF (Env -> QExpr, Set Var) -> Env -> QExpr
mainAlg (QLam vs (ef, fvs)) env =
    let (e, envArg) = envExtend vs ef fvs env
    in Fix $ QApply (Fix $ QLam (EnvId : vs) e) [envArg]
mainAlg (QVar idn) env
    | Just path <- Map.lookup idn env = envElem path
mainAlg e env = Fix $ fmap (($ env) . fst) e
```

Standard Chartered

Conclusions

- EDSLs are quick to build relative to other code generation techniques
- EDSLs let us take back some control over syntax and semantics
- Model and test any assumed semantics with an evaluator
  - quickcheck is invaluable
- Recursion schemes are a principled and effective way to structure traversals and lessen boilerplate
- It's very difficult to generate readable code
  - especially since most names are generated

References

[1] L. Augustsson and M. Agren, “Experience Report: Types for a Relational Algebra Library”, Proc. 9th Symposium on Haskell, pp. 127-132, 2016.

[2] J. Gibbons, “APLicative Programming with Naperian Functors”, Proc. Work. Type-Driven Development, pp 13-14, 2016.

[3] https://wiki.haskell.org/Operational

[4] G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers, “Algebraic data types for language-integrated queries”, Proc. 2013 Work. Data driven Funct. Program. - DDFP ’13, p. 5, 2013.

[5] J. Gibbons, “Origami programming.”, The Fun of Programming, Palgrave, 2003.

[6] E. Meijer, “Functional Programming with Bananas , Lenses , Envelopes and Barbed Wire”, 1991.

This presentation will soon be available on the conference website at the following link:

https://skillsmatter.com/conferences/8522-haskell-exchange-2017#skillscasts

The slides will be available here:

http://www.timphilipwilliams.com/slides/AnEDSLForKDBQ.pdf