Static query optimization

Giuseppe Maggiore Renzo Orsini Michele Bugliesi
Università Ca' Foscari Venezia
Dipartimento di Informatica
{maggiore,orsini,bugliesi}@dsi.unive.it

Abstract

In this paper we show how the results coming from the field of query optimization can be adapted as a step of a modern, high-level optimizing compiler. Thanks to advances in meta-programming we discuss how any functional program performing list processing with the usual higher-order library functions such as List.map, List.filter, etc. could greatly benefit from employing the same techniques and algorithms used to optimize and execute queries in a relational database. This is possible given the close similarity between these widely used libraries for manipulating lists or sequences and the relational operators. The query optimization and execution algorithms can either be encoded as compile-time operations if the language meta-programming facilities are advanced enough, or can be encoded with more conventional runtime type-level operators such as reflection.

1 Introduction

List processing libraries are ubiquitous [] in functional languages. Virtually any functional program will employ either lists, sets, maps, lazy sequences or some variations and with those the accompanying higher-order-functions such as map, filter, reduce, etc. []

These libraries allow for shorter, cleaner and more readable code []. Unfortunately, imperative code which "condensates" various chained operations can very easily be turned into a faster equivalent than its functional counterpart, even though this comes at the cost of conciseness and readability []. This leads to a very undesirable dicotomy between good code and fast code []: no such choice should ever exist!

In this paper we discuss how list processing libraries can be augmented with static information about the shape of the chain of operations performed on the original data source. We use these augmentations to build a static optimizer capable of automatically turning chains of list processing operations into faster equivalents.

To achieve this result we will use the most historically renowned [] set of knowledge in the field of accessing and transforming data: relational algebra. We observe the strict correspondence between most list processing higher order functions and well-known, studied and understood relational operators.

Our technique can be described as follows: list processing operators do not simply return lists; rather, list processing operators return values that can be converted into lists but which type can be used to describe exactly what kind of operation will be performed on the source data. Static operators will then build appropriate optimization and execution functions specifically tailored at compile time for the shape of each operation.

We will use a Haskell-like syntax for describing, optimizing and executing queries. Overlapping instances are so common in our case that we do not claim that our code is actually Haskell, if not for the apparently identical syntax when describing type classes. The semantics of our "imaginary" compiler and runtime are best described in [Mark Jones]: type classes are logical clauses that are resolved at compile-time through back-tracking. This makes our type system capable of computing essentially any transformation that we may wish for on our code.

We argue that this kind of use of types and metaprogramming is the future of many important compiler optimizations. Until now compiler optimizations have been monolithically embedded in compilers and have offered very little customization opportunities (most of the time just a few compiler switches are offered: hardly any customization at all). Thanks to the emergence of powerful metaprogramming mechanism such as type classes (with overlapping instances handled gracefully) we can finally start building libraries that can be executed completely at compile-time and which realize lots of poweful optimizations, from memory management and reference counting [] to algorithmic optimizations such as the one presented in this paper and even up to analysis of one's source code with abstract interpretation and similar approaches.

2 Motivating example

To better understand the problem, we will refer to a typical situation encountered when trying to build videogames in a functional language. Let us suppose that in our game the player must stop the asteroids that come toward the bottom of the screen by shooting them down with plasma projectiles. We will have to update the simulation at fixed (small) intervals

exactly as we would with any physical simulation. The update function takes the amount of time dt that elapsed between this and the previous call to update, a description of the state of the game at time T ($game\_state$) and returns the new state of the game at time $T + dt$. The code that achieves this is the following:

```
type GameState = { asteroids : [Asteroid]; plasma : [Plasma] }

update_state dt (game_state:GameState) =
  let asteroids' = game_state.asteroids
                   |> map (update_asteroid dt)
                   |> filter (fun a -> plasma
                                       |> map (fun p -> p.location)
                                       |> exists (collides a.location))
  let plasma' = game_state.plasma
                |> map (update_plasma dt)
                |> map (fun p -> p.location)
                |> filter (fun p -> asteroids
                                    |> map (fun a -> a.location)
                                    |> exists (collides p.location))
  in { asteroids = asteroids'; plasma = plasma' }
```

Where

```
x |> f = f x
```

The above code can be quite optimized by:

- removing and inlining the unnecessary maps that extract the location of each asteroid and plasma
- grouping the asteroids and the projectiles in spatial clusters (for example with a Quad Tree []) so that the two joins are faster

While this is not the "definitive example" that shows all possible optimizations in action (there are many more optimizations that we will discuss) it is nevertheless quite useful. This code allows us to understand how easy it is to write clean, readable and concise code that uses list processing HOFs but which executes slower than it could and which is quite hard to optimize. Optimizing the above code would most likely result in a total rewriting: hardly convenient. Moreover, in our example the $update\_state$ function will be called at intervals of $\frac{1}{60}^{th}$ of a second: this notion only serves to stress the importance of being able to optimize said code in such scenarios.

In the rest of the paper we will discuss how these and many other optimizations can be achieved statically through the smart use of types.

3 Query definition and execution

Query operators are not defined as a variant type. Rather, we define query operators as all those types which can be converted to a list. The type predicate that characterizes queries is the following:

```
class Query q where
  type Elem q :: *
  run_query :: q -> [Elem q]
```

A type q that represents a query that returns elements of type $Elem\ q$ can be converted ("executed" or "run") by producing a list of elements of type $Elem\ q$. $Elem$ is a type function (a function from types to types []) while run_query is an actual function that is defined for all those types q for which the predicate $Query\ q$ holds.

The first query operators that we support are:

- projection ("select")
- filtering ("filter")
- joining ("join")

.

All these operators will have their instances of the $Query$ predicate. An additional instance will be given for relations, that is for simple lists:

```
instance Query [a] where
  type Elem [a] = a
  run_query l = l
```

A projection simply takes a source query and the fields to be selected.

```
type Select source fields = Select source fields
```

A projection is a query when its source is a query and its fields represent a proper subset of the element of the source query:

```
instance (Query source, Subset fields (Elem source)) =>
          Query (Select source fields) where
  type Elem (Select source fields) =
       Project fields (Elem source)
  run_query (Select source fields) =
       run_query source |> map (project fields)
```

The projection operation requires a valid fields parameter; such a field must be a tuple of type-level naturals. When the fields tuple is empty, then we say that the projection of no fields of any tuple is the empty tuple:

```
instance Tuple value => Subset () value where
  type Project () value = ()
  project () value = ()
```

When we have at least one field index to pick (the inductive case of projecting just one element), then we can say:

```
instance (Natural n, Tuple value,
          n < Length value, Subset ns value) =>
          Subset (n, ns) value where
  type Project (n, ns) value =
       Nth n value , Project ns value
  project (n, ns) value =
       nth n value, project ns value
```

Notice that we are assuming the existence of a series of predicates for determining whether or not a type variable is bound to:

- a type-level numeral (*Natural* predicate)
- a tuple (*Tuple* predicate)

. We also assume a predicate for comparing two type-level naturals ($<$); the *Length* type function is available on *value* since it is true that *Tuple value*. Finally, we freely use the comma operator for building and decomposing inductively defined tuples.

Filtering queries require a source to be filtered and a statically known condition:

```
type Filter source pred = Filter source pred
```

A filter is a proper query when its source is a query and its condition is a valid predicate on the elements of the source query:

```
instance (Query source,
          Predicate pred (Elem source)) =>
          Query (Filter source pred) where
  type Elem (Filter source pred) =
       Elem source
  run_query (Filter source pred) =
       run_query source |> filter (predicate pred)
```

A predicate is a boolean expression where each possible node has a different type; let us consider a small set of operators which can be easily extended:

```
type True = True
type And a b = And a b
type Or a b = Or a b
type Equals a b = Equals a b
```

The simplest instances that we give are for combining together the *True*, *Or* and *And* predicates:

```
instance Predicate True value where
  predicate True value = true

instance (Predicate a value,
          Predicate b value) =>
          Predicate (And a b) value where
  predicate (And a b) value =
       predicate a value && predicate b value

instance (Predicate a value, Predicate b value) =>
          Predicate (Or a b) value where
  predicate (Or a b) value =
       predicate a value || predicate b value
```

Another important type that we define is the item lookup; this type allows us to statically represent the lookup of the n^{th} item of a tuple for comparing it with a specific value in the *Equals* predicate:

```
type Item n = Item n

instance (Natural n, Tuple value,
          n < Length value, Nth n value == b,
          Predicate b value) =>
          Predicate (Equals (Item n) b) value where
  predicate (Equals (Item n) b) value =
       nth n value == b
```

Now we can give predicates such as:

```
And(Equals(Item(0), "Joe"), Equals(Item(2), 21))
```

which can be applied to any tuple where the first item is a string and the third item is an integer.

To make it easier writing conditions, we could even define type synonyms for our various predicates, such as:

```
:&&: = And
:||: = Or
:==: = Equals
```

These synonyms would allow us to rewrite the above example as:

```
(Item(0) :=: "Joe") :&&: (Item(2) :=: 21)
```

Which is quite more readable.

The final query operator we define is the join operator. A join simply is constructed with two queries q_1 and q_2:

```
type Join q1 q1 = Join q1 q2
```

A *Join* is a query when both its parameters are queries and the elements of its sub-queries can be flattened into just one element:

```
instance (Query source1, Query source2,
          CanFlatten (Elem source1) (Elem source2)) =>
          Query (Join source1 source2) where
  type Elem (Join source1 source2) =
       Flatten (Elem source1) (Elem source2)
  run_query (Join source1 source2) =
       [flatten x y | x <- run_query source1,
                      y <- run_query source2]
```

The *CanFlatten* predicate allows us to flatten the two tuples coming from the queries *source*1 and *source*2 into a single tuple:

```
class CanFlatten value1 value2 where
  type Flatten value1 value2 :: *
  flatten :: value1 -> value2 -> Flatten value1 value2
```

We need just two predicates for flattening tuples: one for the base case of flattening an empty tuple with a non-empty tuple, and one for the inductive case:

```
instance (Tuple value2) => CanFlatten () value2 where
  type Flatten () value2 = value2
  flatten () value2 = value2

instance CanFlatten vs value2 => CanFlatten (v,vs) value2 where
  type Flatten (v,vs) value2 = (v, Flatten vs value2)
  flatten (v,vs) value2 = (v, flatten vs value2)
```

4 Query optimization

The optimization of our queries is a simple greedy algorithm that recursively traverses a query tree in search of certain local patterns. We use a subset of the equations coming from relational algebra [].

Filtering is idempotent:

$\sigma_A(R) = \sigma_A\sigma_A(R)$

Subsequent filters can be merged into just one:

$\sigma_{A\wedge B}(R) = \sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$

The filtering of a cartesian product can be decomposed into three sub-filters on the branches of the cartesian product and the resulting cartesian product:

$\sigma_A(R \times P) = \sigma_{B\wedge C\wedge D}(R \times P) = \sigma_D(\sigma_B(R) \times \sigma_C(P))$

As long as the required labels are maintained, a filtering and a projection can be swapped:

$\pi_{a_1,...,a_n}(\sigma_A(R)) = \sigma_A(\pi_{a_1,...,a_n}(R))$ where fields in $A \subseteq \{a_1, ..., a_n\}$

The main task of our optimizer thus becomes that of applying these rules and pushing filter operations as deep as possible in the optimized query. This way filtering happens as early as possible and all subsequent operations have less elements to process.

The first definition we give is a type class that describes a query which can be optimized. This type class has a type function, *Optimized*, which returns the shape (the type) of the optimized query and a function *optimize* which takes the original query and returns its optimized counterpart:

```
class OptimizableQuery q where
  type Optimized q :: *
  optimize :: q -> Optimized q
```

The simplest instance of an optimizable query is, of course, the basic relation. In this case the optimization is simply the identity function:

```
instance OptimizableQuery [a] where
  type Optimized [a] = [a]
  optimize q = q
```

Now we give a series of instances of optimizable queries which are only used to recursively traverse the query in search for other places where we can perform the optimization. Projections of optimizable queries are optimized by building the projection of the optimized argument:

```
instance OptimizableQuery q =>
         OptimizableQuery (Project fields q) where
  type Optimized (Project fields q) =
          Project fields (Optimized q)
  optimize (Project fields q) =
          Project fields (optimize q)
```

Cartesian products of optimizable queries are optimized by building the join of the optimized arguments:

```
instance (OptimizableQuery q1, OptimizableQuery q2) =>
          OptimizableQuery (Join q1 q2) where
  type Optimized (Join q1 q2) =
       Join (Optimized q1) (Optimized q2)
  optimize (Join q1 q2) =
       Join (optimize q1) (optimize q2)
```

The optimization of a filtering is the filtering of the optimized inner query:

```
instance OptimizableQuery q => OptimizableQuery (Filter cond q) where
  type Optimized (Filter cond q) = Filter cond (Optimized q)
  optimize (Filter cond q) = Filter cond (optimize q)
```

The optimization of two nested filterings where both conditions are identical simply removes the redundant filtering and optimizes the resulting query:

```
instance (StaticallyEquals cond1 cond2,
          OptimizableQuery (Filter cond1 q)) =>
          OptimizableQuery (Filter cond1 (Filter cond2 q)) where
  type Optimized (Filter cond1 (Filter cond2 q)) =
         Optimized (Filter cond1 q)
  optimize (Filter cond1 (Filter cond2 q)) =
         optimize (Filter cond1 q)
```

The *StaticallyEquals* predicate checks whether two conditions are exactly the same *at compile-time*; this requires a mechanism for encoding constants as types (as we are doing with integers).

```
class StaticallyEquals cond1 cond2
```

The constants used and their encoding depend on the application and the elements of our queries. This means that the *StaticallyEquals* predicate will potentially need many domain-specific instances. What we can define about this predicate is the traversal of conditions:

```
instance StaticallyEquals True True

instance (StaticallyEquals a a', StaticallyEquals b b') =>
          StaticallyEquals (And a a') (And b b')

instance (StaticallyEquals a a', StaticallyEquals b b') =>
          StaticallyEquals (Or a a') (Or b b')

instance (StaticallyEquals a a', StaticallyEquals b b') =>
          StaticallyEquals (Equals a a') (Equals b b')

instance StaticallyEquals (Item n) (Item n)
```

Notice that there is an obvious overlapping between the last two instances: this is not really a problem since we can either use a most specific match (if the second instance can be matched, then the first is ignored) or we can order the instances so that the least specific instances are checked only if the most specific fail.

Two nested filterings are merged into a single filtering with the two original conditions merged with an *And* and the result is optimized (if the two conditions are not the same, in which case the previous rule is used):

```
instance (not(StaticallyEquals cond1 cond2),
          OptimizableQuery (Filter (And cond1 cond2) q)) =>
          OptimizableQuery (Filter cond1 (Filter cond2 q)) where
  type Optimized (Filter cond1 (Filter cond2 q)) =
          Optimized (Filter (And cond1 cond2) q)
  optimize (Filter cond1 (Filter cond2 q)) =
          optimize (Filter (And cond1 cond2) q)
```

We use the type-level $not(X)$ operator to express failure in proving the X predicate.

A project within a filtering can be swapped and the result optimized, provided that the filtering condition is correctly offset since the indices of the various fields are now different:

```
instance (Remappable cond fields,
          OptimizableQuery (Filter (Remap cond fields) q)) =>
          OptimizableQuery (Filter cond (Project fields q)) where
  type Optimized (Filter cond (Project fields q)) =
          Project fields (Optimized (Filter (Remap cond fields) q)))
  optimize (Filter cond (Project fields q)) =
          Project fields (optimze (Filter (remap cond fields) q)))
```

Remapping a condition so that it looks up the correct fields in a value before the remapping performed by the projection:

```
class Remappable cond fields where
  type Remap cond fields :: *
  remap :: cond -> fields -> Remap cond fields
```

To instance this predicate we will iterate the various *Items* in *fields* and for the n^{th} *Item* i we replace its index n with i. We define another predicate (which is identical to *Remappable* but with an explicit accumulator argument to track the index of each item):

```
class RemappableAux cond fields acc where
  type RemapAux cond fields acc :: *
  remap_aux :: cond -> fields -> acc -> RemapAux cond fields acc
```

The conversion rule states that remapping can be computed with our auxiliary predicate provided that the starting index is zero:

```
instance RemappableAux cond fields 0 => Remappable cond fields where
  type Remap cond fields = RemapAux cond fields 0
  remap cond fields = remap_aux cond fields 0
```

When all fields are processed (*fields* is empty), then no mapping is necessary:

```
instance RemappableAux cond () n where
  type RemapAux cond fields n = cond
  remap_aux cond fields = cond
```

When *fields* is not empty, then we replace the current item with its index in the condition and then we iterate:

```
instance (Natural n, Natural i,
          RemappableAux cond fields (n+1),
          MappableItem n i cond) =>
          RemappableAux cond (Item i, fields) n where
  type RemapAux cond (Item i, fields) n =
          RemapAux (MapItem n i cond) fields (n+1)
  remap_aux cond (Item i, fields) n =
          remap_aux (map_item n i cond) fields (n+1)
```

The mapping of conditions is relatively straightforward. The mapping predicate is defined as:

```
class MappableItem i i' cond where
  type MapItem i i' cond :: *
  map_item :: i -> i' -> cond -> MapItem i i' cond
```

While most instances simply traverse the condition tree (in a fashion similar to []):

```
instance MappableItem i i' True where
  type MapItem i i' True = True
  map_item i i' True = True

instance (MappableItem i i' a,
          MappableItem i i' b) =>
          MappableItem i i' (And a b) where
  type MapItem i i' (And a b) = And (MapItem i i' a) (MapItem i i' b)
  map_item i i' (And a b) = And (map_item i i' a) (map_item i i' b)

instance (MappableItem i i' a,
          MappableItem i i' b) =>
          MappableItem i i' (Or a b) where
  type MapItem i i' (Or a b) = Or (MapItem i i' a) (MapItem i i' b)
  map_item i i' (Or a b) = Or (map_item i i' a) (map_item i i' b)

instance (MappableItem i i' a,
          MappableItem i i' b) =>
          MappableItem i i' (Equals a b) where
  type MapItem i i' (Equals a b) = Equals (MapItem i i' a) (MapItem i i' b)
  map_item i i' (Equals a b) = Equals (map_item i i' a) (map_item i i' b)
```

When we encounter an item we check if its parameter (n) has the value we are looking for (i). If the answer is positive, then we replace n with i'. Otherwise we leave the item intact:

```
instance i == n =>
         MappableItem i i' (Item n) where
```

```
  type MapItem i i' (Item n) = Item i'
  map_item i i' (Item n) = Item i'

instance i /= n =>
         MappableItem i i' (Item n) where
  type MapItem i i' (Item n) = Item n
  map_item i i' (Item n) = Item n
```

Possibly the most complex instance of an optimizable query is that of a filtering with a cartesian product inside. In this case we split the condition *cond* into three sub-conditions:

- $cond_{12}$, which contains those portions of *cond* that lookup elements from both queries
- $cond_1$, which contains those portions of *cond* that only lookup elements from the query q_1
- $cond_2$, which contains those portions of *cond* that only lookup elements from the query q_2

The split is such that $cond \equiv cond_{12} \wedge cond_1 \wedge cond_2$; otherwise, we would risk creating an optimized query that yields more elements than the original one.

The optimized query is taken by creating three filterings; one with $cond_{12}$ is applied to the result of the join, while the other two respectively apply the conditions $cond_1$ and $cond_2$ to the sub-queries q_1 and q_2. The resulting join (not the outermost filter) is further optimized:

```
instance (e1 = Elem q1,
          e2 = Elem q2,
          Splittable cond e1 e2,
          cond12 = Shared cond e1 e2,
          cond1 = Cond1 cond e1 e2,
          cond2 = Cond2 cond e1 e2,
          OptimizableQuery (Join (Filter cond1 q1) (Filter cond2 q2))) =>
          OptimizableQuery (Filter cond (Join q1 q2)) where
  type Optimized (Filter cond (Join q1 q2)) = Filter cond12 (Optimized (Join (Filter cond1
  optimize (Filter cond (Join q1 q2)) =
    let cond12, cond1, cond2 = split cond
    Filter cond12 (optimize (Join (Filter cond1 q1) (Filter cond2 q2))))
```

Splitting a condition requires checking its shape. We look for nested sequences of *And*s and analyze the domain of each branch; we can group together those branches which lookup values from only one of the two query elements. The type class that splits a condition with respect to two query elements contains:

- three type functions *Shared*, *Cond*1 and *Cond*2 which compute the type of the condition split into three parts: those that access values from both elements of the sub-queries, those that access values only from the first element and those that access values only from the second element;
- a split function that given a condition *cond* splits its into a triplet containing its three subconditions.

```
class Splittable cond e1 e2 where
  type Shared cond e1 e2 :: *
  type Cond1 cond e1 e2 :: *
  type Cond1 cond e1 e2 :: *
  split :: cond -> (Shared cond e1 e2, Cond1 cond e1 e2, Cond2 cond e1 e2)
```

The first instance of the splitting predicate is the fallback rule; this rule will not be matched unless all the other, more specific rules fail. This rule is required because any condition that is not an *And* simply cannot be safely split:

```
instance Splittable cond e1 e2 where
  type Shared (And a b) e1 e2 = cond
  type Cond1 (And a b) e1 e2 = True
  type Cond2 (And a b) e1 e2 = True
  split cond = (cond,True,True)
```

If the condition is an *And* and both its branches access values from the first element of the query, then the entire condition will only be tested on the first sub-query:

```
instance (Domain a e1 e2 = First, Domain b e1 e2 = First) =>
         Splittable (And a b) e1 e2 where
  type Shared (And a b) e1 e2 = True
  type Cond1 (And a b) e1 e2 = And a b
  type Cond2 (And a b) e1 e2 = True
  split (And a b) = (True, And a b, True)
```

The *Domain* predicate is relatively straightforward, in that it simply checks whether all the *Item*s of a condition fall within the first element (e_1), if they access indices all greater than *Length* e_1 or if it access both kind of indices.

Similarly, if both branches of the *And* only access values from the only one element of the query, then each branch needs only to be tested on the corresponding sub-query; in this case care must be taken to remap the domain of the condition b_2, because now it will not be applied on merged pairs of type *Flatten* e_1 e_2, but rather it will be applied on values of type e_2:

```
instance (Domain a e1 e2 = First,
          Domain b e1 e2 = Second,
          RemappableDomain b e1 e2) =>
          Splittable (And a b) e1 e2 where
  type Shared (And a b) e1 e2 = True
  type Cond1 (And a b) e1 e2 = a
  type Cond2 (And a b) e1 e2 = RemapDomain b e1 e2
  split (And a b) = (True, a, b)
```

The *RemappableDomain cond* e_1 e_2 predicate is very similar to the *MappableItem* predicate seen above; and simply turns every instance of *Item* i found inside *cond* into an instance of *Item* $(i - \mathit{Length}\ e_1)$.

In case the first branch a only refers to the first sub-query but the second branch b also refers to the second sub-query, then we recursively split the second branch into three sub-queries; the sub-query of b that refers to the first sub-query is then put in conjunction with a:

```
instance (Domain a e1 e2 = First, Domain b e1 e2 = Both,
          Splittable b e1, e2) =>
          Splittable (And a b) e1 e2 where
  type Shared (And a b) e1 e2 = Shared b e1 e2
  type Cond1 (And a b) e1 e2 = And a (Cond1 b e1 e2)
  type Cond2 (And a b) e1 e2 = Cond2 b e1 e2
  split (And a b) =
    let (b_shared,b1,b2) = split b
    (b_shared, And a b1, b2)
```

The remaining cases cover all the other possible combinations of the various domains of the branches of an *And* condition; we omit them since they are very similar to the ones already presented.

5 Optimized execution

6 Additional operators

7 Conclusions and future work

[0]