Tupling via Constructive Algorithmics

Kate Verbitskaia

JetBrains Programming Languages and Tools Lab

20.09.2021

What is Tupling

Program transformation technique which groups functions with same arguments together

Objectives:

- Eliminate multiple traversals of the same data structure
- Eliminate redundant recursive calls

Example: Maximum and Length

Compute both maximum value of the list and its length

Example: Average of the List

Compute the average of the list of numbers

Tupling via Fold/Unfold

A Transformation System for Developing Recursive Programs.
R.M. Burstall and John Darlington. (1977)

Objective: Transform a "very simple, lucid and hopefully correct program" into a more efficient one

How: using a combination of the following transformations[1]

- Definition
- Instantiation
- Unfolding
- Folding
- Abstraction
- Laws

[1] And some eureka tuples

Transformations in the Fold/Unfold Framework

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

- Definition
  - Introduce a new recursive equation
  - LHS should not be an instance of any other equation
  - `g n = (fib (n+1), fib n)`
- Instantiation
  - Introduce a substitution instance of an existing equation
  - `g 0 = (fib (0+1), fib 0)`
- Unfolding
  - Replace LHS of an equation with the corresponding instance of its RHS within some other expression
  - `g 0 = (fib (0+1), fib 0) = (fib 1, fib 0) = (1, 1)`

Transformations in the Fold/Unfold Framework

- Abstraction
  - Add a `where` clause
  - `g (n+1) = (fib (n+2), fib (n+1))`
  - `g (n+1) = (fib (n+1) + fib (n), fib (n+1))`
  - `g (n+1) = (u + v, u) where (u, v) = (fib (n+1), fib n)`
- Folding
  - Replace RHS of some equation with the instance of the LHS
  - `g (n+1) = (u + v, u) where (u, v) = (fib (n+1), fib n)`
  - `g (n+1) = (u + v, u) where (u, v) = g n`
- Laws
  - Rewrite equations using some laws valid in the domain
  - `0 + 1 = 1`
  - `x + y = y + x`
  - `(x + y) * z = x * z + y * z`

Eureka Tuples

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

Eureka Tuples

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

`g n = (fib (n+1), fib n)` — Eureka tuple

Eureka Tuples

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

`g n = (fib (n+1), fib n)` — Eureka tuple

A few transformations later...

```
g 0 = (1, 1)
g (n+1) = (u + v, u) where (u, v) = g n

fib 0 = 1
fib 1 = 1
fib (n+2) = u + v where (u, v) = g n)
```

Tupling Strategy

A Powerful Strategy for Deriving Efficient Programs by Transformations.
Alberto Pettorossi. (1984)

Objective: find a way to derive eureka steps

How: find a *progressive sequence of cuts* in a dependency graph of a function with the *same number* of nodes and make a tuple out of it.

Progressive Sequence of Cuts

Cut: set of nodes in a dependency graph, s.t. if we remove them along with their edges, we are left with 2 disconnected graphs g_1 and g_2, and $\forall m$ — node in g_1, $\forall n$ — node in g_2: $m > n$ [2]

Progressive sequence of cuts:
$\{c_i \mid 0 \leq i \leq k\}$,
$\forall i.\ c_i \cap c_{i-1} \neq c_i \neq c_{i-1}$,
$\forall m \in c_i \setminus (c_i \cap c_{i-1}).\ \exists n \in c_{i-1}.\ m > n$, and
$\forall n \in c_{i-1} \setminus (c_i \cap c_{i-1}\ \exists m \in c_i.\ m > n)$

[2] $>$ is an ancestor-descendent relation

Progressive Sequence of Cuts: Example

```
f n a b c =
  if n == 0
  then skip
  else f (n-1) a c b ++ ab ++ f (n-1) c b a
```

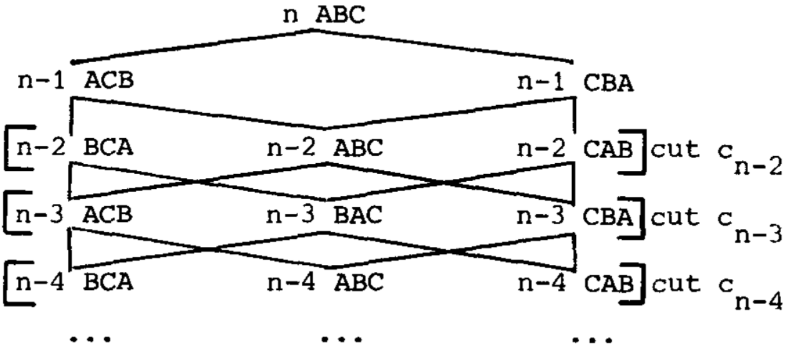

fig.2 The m-dag for f(n,A,B,C). n xyz ≡ f(n,x,y,z).

Tupling Strategy: Limitations

- Not fully automatic
- Possible heuristic 1: search for repeated computations while building dependency graph
- Possible heuristic 2: not unfold those recursive calls which can be derived in constant time from calls already in the cut

Mechanizing Tupling Further

Towards an Automated Tupling Strategy.
Wei-Ngan Chin. (1993)

Objective: develop a fully automatic tupling algorithm

How:

- Remove the most senior nodes in a cut and replace them with their children
- Check if there are ancestors which match the cut
- If they match, it is a candidate for tupling
- If they do not, check the next cut

Towards an Automated Tupling Strategy: Limitations

- Extension of the method:
  - Tree of cuts instead of sequences
  - Recursion parameter ordering
- Termination is only guaranteed if:
  - There is a single recursive parameter
  - The recursive parameter is strictly decreasing
  - No other parameters are accumulating
  - All variables in recursive calls are taken from the recursive parameters
- Good news: sometimes preprocessing can be used to transform the function into this form
- Still: Needs a clever control to avoid infinite unfolding and is not easy to implement in a real compiler

Tupling via Constructive Algorithmics

Tupling Calculation Eliminates Multiple Data Traversals.
Z. Hu, H. Iwasaki, M. Takeichi, A. Takano. (1997)

Objective: create a fully automatic tupling algorithm suitable to be used in a real compiler

How: ~~throw some category theory at the problem~~

Tupling via Constructive Algorithmics

Tupling Calculation Eliminates Multiple Data Traversals.
Z. Hu, H. Iwasaki, M. Takeichi, A. Takano. (1997)

Objective: create a fully automatic tupling algorithm suitable to be used in a real compiler

How: ~~throw some category theory at the problem~~

How: use Constructive Algorithmics and Mutu theorem

Constructive Algorithmics

- Represent data types as *polynomial endofunctors*
- Represent recursive functions as *catamorphisms*
- Use *Mutu theorem* and other *laws* to transform recursive functions

Constructive Algorithmics: Polynomial Endofunctors

- Identity
  - $I\ X = X$
  - $I\ f = f$
- Constant
  - $!A\ X = A$
  - $!A\ f = id$
- Product $X \times Y$
  - $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$
  - $\pi_1(a, b) = a$
  - $\pi_2(a, b) = b$
  - $(f \times g)(x, y) = (f\ x, g\ y)$
  - $(f \triangle g) a = (f\ a, g\ a)$
- Separated sum $X + Y$
  - $X + Y = \{1\} \times X \cup \{2\} \times Y$
  - $(f + g)(1, x) = (1, f\ x)$
  - $(f + g)(2, y) = (2, g\ y)$
  - $(f \triangledown g)(1, x) = f\ x$
  - $(f \triangledown g)(2, y) = g\ y$

Polynomial Functors: List

```
data List a = Nil | Cons a (List a)
```

$F_{L_A} = !1 + !A \times I$

$in_{F_{L_A}} = Nil \triangledown Cons$

```
out = \xs . case xs of
              Nil -> (1, ())
              Cons a as -> (2, (a, as))
```

Polynomial Functors: Binary Tree

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

$F_{T_A} = !A + I \times I$

$in_{F_{T_A}} = Leaf \triangledown Node$

```
out = \t . case t of
            Leaf a -> (1, a)
            Node l r -> (2, (l, r))
```

Catamorphisms: List

```
cata [] = e
cata (x : xs) = x + (cata xs)
```

Here e and + uniquely determine a catamorphism over lists

Can be rewritten: $cata = ([e \triangledown +])_{F_{L_A}}$

Catamorphisms over a data type captured by functor F is characterized by:

$$h = ([\phi])_F \;\equiv\; h \circ in_F = \phi \circ F\,h$$

$$sum = (\![0 \triangledown plus]\!)$$

$$\begin{array}{ll}
& sum = (\![0 \triangledown plus]\!) \\
\equiv & \quad \{ \text{ catamorphism charaterization } \} \\
& sum \circ in_{F_{L_A}} = (0 \triangledown plus) \circ F_{L_A}\, sum \\
\equiv & \quad \{ \ in_{F_{L_A}} = (Nil \triangledown Cons),\ F_{L_A} f = id + id \times f \ \} \\
& sum \circ (Nil \triangledown Cons) = (0 \triangledown plus) \circ (id + id \times sum) \\
\equiv & \quad \{ \text{ Laws for } \triangledown, + \text{ and } \circ \} \\
& (sum\ Nil) \triangledown (sum \circ Cons) = 0 \triangledown (plus \circ (id \times sum)) \\
\equiv & \quad \{ \text{ by laws of } \triangledown \ \} \\
& sum\ Nil = 0; \quad sum \circ Cons = plus \circ (id \times sum)
\end{array}$$

That is,

$$\begin{array}{lcl}
sum\ Nil & = & 0 \\
sum\ (Cons(x, xs)) & = & plus(x, sum\ xs).
\end{array}$$

The Mutu Tupling Theorem

Theorem 1 (Mutu Tupling)

$$\frac{f \circ in_F = \phi \circ F(f \vartriangle g), \;\; g \circ in_F = \psi \circ F(f \vartriangle g)}{f \vartriangle g = (\!|\phi \vartriangle \psi|\!)_F}$$

- Functions which traverse over the same data structure (in a specific regular way) should be tupled
- Tupling should be done with a catamorphism

Example: Deepest Leaves

$$
\begin{array}{lcl}
deepest\ (Leaf\,(a)) & = & [a] \\
deepest\ (Node(l,r)) & = & deepest(l),\ \ depth(l) > depth(r) \\
 & = & deepest(l) +\!\!+\ deepest(r), \\
 & & \qquad\qquad depth(l) = depth(r) \\
 & = & deepest(r),\ \ \text{otherwise} \\
depth\ (Leaf\,(a)) & = & 0 \\
depth\ (Node(l,r)) & = & 1 + max(depth(l), depth(r))
\end{array}
$$

Deepest Leaves: Mutu Theorem Application

$$
\begin{array}{l}
deepest \circ in_{F_{T_{Int}}} = \phi \circ F_{T_{Int}}(deepest \vartriangle depth) \\
\quad \textbf{where} \;\; \phi = \phi_1 \triangledown \phi_2 \\
\qquad\qquad \phi_1 \; a = [a] \\
\qquad\qquad \phi_2 \; ((tl, hl), (tr, hr)) = tl, \qquad\quad \textbf{if } hl > hr \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad = tl \mathbin{+\!\!+} tr, \;\; \textbf{if } hl = hr \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad = tr, \qquad\quad \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{l}
depth \circ in_{F_{T_{Int}}} = \psi \circ F_{T_{Int}}(deepest \vartriangle depth) \\
\quad \textbf{where} \;\; \psi = \psi_1 \triangledown \psi_2 \\
\qquad\qquad \psi_1 \; a = 0 \\
\qquad\qquad \psi_2 \; ((tl, hl), (tr, hr)) = 1 + max(hl, hr)
\end{array}
$$

$$
deepest = \pi_1 \circ (deepest \vartriangle depth) = \pi_1 \circ (\![\phi \vartriangle \psi]\!)_{F_{T_{Int}}}
$$

Main Property of the Approach

All multiple data traversals by tuplable functions in a program can be eliminated by tuple calculation

Multiple Data Traversal

Multiple data traversal: if there exists two calls $f\ p$ and $f'\ p'$, in which p is equal or is a sub-pattern of p'

Tuplable Functions

Mutually recursive functions:

$$f_1, \ldots, f_m$$

Defined by equations:

$$f_i \;\; p_{ij} \;\; v_{s_1} \ldots v_{s_{n_i}} = e_{ij}$$

$f_1, \ldots, f_m$ are called *tuplable* if for every occurance of recursive calls to $f_1, \ldots, f_m$ in all e_{ij}, say $f_k \;\; e' \;\; e_1 \ldots e_{n_k}$, e' is a sub-pattern of p_{ij}

Tuplable Functions: Examples

Example:

$$\begin{aligned} rep\ \underline{(Node\,(\mathrm{l},\mathrm{r}))}\ ms = Node\,(&rep\ \underline{\mathrm{l}}\ (take\ (size\ l)\ ms), \\ &rep\ \underline{\mathrm{r}}\ (drop\ (size\ l)\ ms)) \end{aligned}$$

Non-Example:

$$foo\,(x_1 : x_2 : xs) = x_1 + foo\,\underline{(2 * x_2 : xs)} + foo\,\underline{(x_1 : xs)}$$

Standardizing

$$f = \phi \circ (Fh \circ out_F \vartriangle F^2 h \circ out_F^2 \vartriangle \cdots \vartriangle F^l h \circ out_F^l) \quad (1)$$

where

(i) $h = f_1 \vartriangle \cdots \vartriangle f_n \vartriangle g_1 \vartriangle \cdots \vartriangle g_m$, where $f_1, \cdots, f_n$ denote functions mutually defined with f and one of them is f, and $g_1, \cdots, g_m$ denote tuplable functions in the definition of f while traversing over the same recursive data as f;

(ii) $F^n = F^{n-1} \circ F$ and $out_F^n = F^{n-1} out_F \circ out_F^{n-1}$;

(iii) l is a finite natural number.

Manipulation of Functions in Standard Form

$$\frac{f = \phi \circ (Fh \circ out_F \vartriangle \cdots \vartriangle F^l h \circ out_F^l)}{f = (\phi \circ (\pi_1 \vartriangle \cdots \vartriangle \pi_l)) \circ (Fh \circ out_F \vartriangle \cdots \vartriangle F^l h \circ out_F^l \vartriangle F^{l+1} h \circ out_F^{l+1})} \tag{R1}$$

$$\frac{f = \phi \circ (F(h_1 \vartriangle \cdots \vartriangle h_n) \circ out_F \vartriangle \cdots \vartriangle F^l(h_1 \vartriangle \cdots \vartriangle h_n) \circ out_F^l)}{\begin{array}{lll} f & = & (\phi \circ (F\Pi \times \cdots \times F^l\Pi)) \circ (FH \circ out_F \vartriangle \cdots \vartriangle F^l H \circ out_F^l) \\ & & \textbf{where } \Pi = \pi_1 \vartriangle \cdots \vartriangle \pi_n,\ H = h_1 \vartriangle \cdots \vartriangle h_n \vartriangle h_{n+1} \end{array}} \tag{R2}$$

$$\frac{f = \phi \circ (F(h_1 \vartriangle h_2) \circ out_F \vartriangle \cdots \vartriangle F^l(h_1 \vartriangle h_2) \circ out_F^l)}{\begin{array}{lll} f & = & (\phi \circ (Fex \times \cdots \times F^l ex)) \circ (F(h_2 \vartriangle h_1) \circ out_F \vartriangle \cdots \vartriangle F^l(h_2 \vartriangle h_1) \circ out_F^l) \\ & & \textbf{where } \ ex\,(x, y) = (y, x) \end{array}} \tag{R3}$$

$$\frac{\begin{array}{lll} f & = & \phi \circ (F(f \vartriangle g) \circ out_F \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^l(f \vartriangle g) \circ out_F^l) \\ g & = & \psi \circ (F(f \vartriangle g) \circ out_F \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^l(f \vartriangle g) \circ out_F^l) \end{array}}{f \vartriangle g = (\phi \vartriangle \psi) \circ (F(f \vartriangle g) \circ out_F \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^l(f \vartriangle g) \circ out_F^l)} \tag{R4}$$

Tupling of Tuplable Functions

Theorem 5 (Tupling Tuplable Functions) Let

$$f_i = \Pi_i \circ (\![\phi_i]\!)_F, \; i = 1, \cdots, n$$

be n tuplable functions where Π_i stands for a projection function. Then,

$$f_1 \vartriangle \cdots \vartriangle f_n = \Pi \circ (\![\phi]\!)_F$$

where $\Pi = \Pi_1 \times \cdots \times \Pi_n$ and $\phi = \phi_1 \circ F\pi_1 \vartriangle \cdots \vartriangle \phi_n \circ F\pi_n$.

Example: Fibonacci

$$
\begin{array}{lcl}
\mathit{fib}\ \mathit{Zero} & = & \mathit{Zero} \\
\mathit{fib}\ (\mathit{Succ}(\mathit{Zero})) & = & \mathit{Succ}(\mathit{Zero}) \\
\mathit{fib}\ (\mathit{Succ}(\mathit{Succ}(n))) & = & \mathit{plus}(\mathit{fib}(\mathit{Succ}(n)), \mathit{fib}(n))
\end{array}
$$

$$
\mathit{fib} = \phi \circ (F_N \mathit{fib} \circ \mathit{out}_{F_N} \vartriangle F_N^2 \mathit{fib} \circ \mathit{out}_{F_N}^2)
$$

$$
\begin{array}{lcl}
F_N & = & !\mathbf{1}\ +\ I \\
F_N^2 & = & !\mathbf{1}\ +\ (!\mathbf{1}\ +\ I) \\
\mathit{out}_{F_N} & = & \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to (1, ()) \\
 & & \qquad\qquad\quad \mathit{Succ}(n) \to (2, n) \\
\mathit{out}_{F_N}^2 & = & \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to (1, ()) \\
 & & \qquad\qquad\quad \mathit{Succ}(n) \to \\
 & & \quad (2, \mathbf{case}\ n\ \mathbf{of}\ \mathit{Zero} \to (1, ()) \\
 & & \qquad\qquad\qquad\quad \mathit{Succ}(n') \to (2, n'))
\end{array}
$$

Example: Fibonacci

$$
\begin{array}{l}
\mathit{fib} = \phi \circ (\lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \to ((1,()),(1,())) \\
\qquad\qquad\qquad\qquad\quad \mathit{Succ}(n) \to \\
\qquad\qquad ((2, \mathit{fib}\, n), \\
\qquad\qquad\ (2, \mathbf{case}\ n\ \mathbf{of}\ \mathit{Zero} \to (1,()) \\
\qquad\qquad\qquad \mathit{Succ}(n') \to (2, \mathit{fib}\, n')))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{fib} & = & \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{Zero} \ \to \mathit{Zero} \\
 & & \qquad\qquad\qquad \mathit{Succ}(n) \to \\
 & & \mathbf{case}\ n\ \mathbf{of}\ \mathit{Zero} \to \mathit{Succ(Zero)} \\
 & & \qquad\qquad\quad \mathit{Succ}(n') \to \mathit{plus}(\mathit{fib}\, n, \mathit{fib}\, n').
\end{array}
$$

$$
\begin{array}{rcl}
\phi & = & \lambda(x,y).\ \mathbf{case}\ (x,y)\ \mathbf{of}\ ((1,()),(1,())) \to \mathit{Zero} \\
 & & \qquad\qquad\qquad\qquad\quad ((2,f_1),(2,y')) \to \\
 & & \quad \mathbf{case}\ y'\ \mathbf{of}\ (1,()) \to \mathit{Succ(Zero)} \\
 & & \qquad\qquad\qquad (2,f_2) \to \mathit{plus}(f_1,f_2).
\end{array}
$$

$$
\mathit{fib} = \pi_1 \circ (\![\phi \circ (F_N \pi_1 \vartriangle F_N \pi_2) \vartriangle F_N \pi_1]\!)_{F_N}.
$$

Fibonacci: Result

$$
\begin{array}{lcl}
\mathit{fib}\ n & = & x \\
 & & \quad \textbf{where}\ (x, y) = f'\ n \\
f'\ \mathit{Zero} & = & (\mathit{Zero}, (1, ())) \\
f'\ (\mathit{Succ}(n)) & = & (\textbf{case}\ y'\ \textbf{of}\ (1, ()) \to \mathit{Succ}(\mathit{Zero}) \\
 & & \qquad\qquad\qquad (2, f_2) \to \mathit{plus}(f_1, f_2), \\
 & & \ (2, f_1)) \\
 & & \textbf{where}\ (f_1, y') = f'\ n
\end{array}
$$

Tupling Calculation: Properies and Limitations

+ The algorithm is correct
+ The algorithm terminates
+ All multiple data traversals by tuplable functions can be eliminated
– Tupled functions require extra memory
– Needs an efficient tuple implementation

Literature

- *A Transformation System for Developing Recursive Programs.* R.M. Burstall and John Darlington. (1977)
- *A Powerful Strategy for Deriving Efficient Programs by Transformations.* Alberto Pettorossi. (1984)
- *Towards an Automated Tupling Strategy.* Wei-Ngan Chin. (1993)
- *Tupling Calculation Eliminates Multiple Data Traversals.* Z. Hu, H. Iwasaki, M. Takeichi, A. Takano. (1997)