Program Transformation to Identify List-Based Parallel Skeletons

Venkatesh Kannan and G. W. Hamilton

Dublin City University, Ireland

02 - Apr - 2016

Background

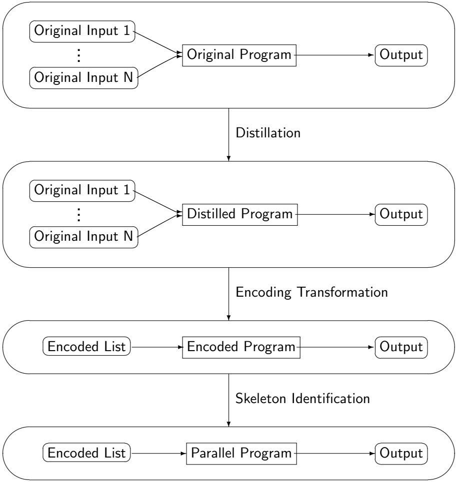
- Algorithmic skeletons used as building blocks in parallel program development.
- *Positives*.
  - Abstract away parallel implementation from developer.
- *Challenges*.
  - Requires intricate analysis of underlying algorithm.
  - Multiple skeletons may introduce inefficient intermediate data structures.
  - Potential mismatch in data structures and algorithms used by the skeletons and the program.
  - Most available skeleton libraries are defined over flat data types (list or arrays).

Existing Work

- Analytical Approaches.
  - Use static program analysis to rewrite recursive functions using skeletons.
  - *Positives*.
    - Minimum restriction on programs and inputs.
  - *Limitations*.
    - Use of inefficient intermediate data structures.
- Program Transformation Approaches.
  - Systematically transform/derive parallel functions in specific forms.
  - *Positives*.
    - Structured derivation of parallel programs.
  - *Limitations*.
    - Restrictions on programs and inputs.
    - Manual derivation of operators with desired properties.

Proposed Transformation Method

Desirable Solution: Automatic + Generic - Intermediate Data.



Functional Language

$$
\begin{array}{lll}
e ::= & x & \text{Variable} \\
\mid & c\ e_1 \ldots e_N & \text{Constructor Application} \\
\mid & e_0 & \text{Function Definition} \\
 & \textbf{where} & \\
 & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = e_1 & \\
 & \vdots & \\
 & f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = e_K & \\
\mid & f & \text{Function Call} \\
\mid & e_0\ e_1 & \text{Application} \\
\mid & \textbf{let}\ x_1 = e_1\ \ldots\ x_N = e_N\ \textbf{in}\ e_0 & \textbf{let}\text{–expression} \\
\mid & \lambda x.e & \lambda\text{–abstraction} \\
p ::= & x \mid c\ p_1 \ldots p_N & \text{Pattern} \\
\textbf{data}\ T\ \alpha_1 \ldots \alpha_M ::= & c_1\ t_1^1 \ldots t_N^1 \mid \ldots \mid c_K\ t_1^K \ldots t_N^K & \text{Data Type Declaration}
\end{array}
$$

Notation:

- Context expression – $E[e_1, \ldots, e_N]$

Example: Matrix Multiplication

$$
\begin{array}{lcl}
mMul & :: & [[a]] \rightarrow [[a]] \rightarrow [[a]] \\
\multicolumn{3}{l}{mMul\ xss\ yss} \\
\multicolumn{3}{l}{\textbf{where}} \\
mMul\ []\ yss & = & [] \\
mMul\ (xs : xss)\ yss & = & (map\ (dotp\ xs)\ \boxed{(transpose\ yss)}\ ) : (mMul\ xss\ yss) \\
dotp\ xs\ ys & = & foldr\ (+)\ 0\ \boxed{(zipWith\ (*)\ xs\ ys)} \\
transpose\ yss & = & transpose'\ yss\ [] \\
transpose'\ []\ yss & = & yss \\
transpose'\ (xs : xss)\ yss & = & transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ []\ yss & = & yss \\
rotate\ (x : xs)\ [] & = & [x] : (rotate\ xs\ yss) \\
rotate\ (x : xs)\ (ys : yss) & = & (ys +\!\!+ [x]) : (rotate\ xs\ yss)
\end{array}
$$

Distillation

- An unfold/fold-based program transformation method.
- Composes function definitions, reduces the number of intermediate data structures.
- Can potentially provide superlinear speedups.

$$
\begin{array}{rcll}
de^{\rho} & ::= & x\ de_1^{\rho} \ldots de_N^{\rho} & \text{Variable Application} \\
 & | & c\ de_1^{\rho} \ldots de_N^{\rho} & \text{Constructor Application} \\
 & | & de_0^{\rho} & \text{Function Definition} \\
 & & \textbf{where} & \\
 & & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = de_1^{\rho}\ \ldots\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = de_K^{\rho} & \\
 & | & \boxed{f\ x_1 \ldots x_N} & \text{Function Application} \\
 & & \quad \text{where } f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = de_1^{\rho}\ \ldots\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = de_K^{\rho} & \\
 & & \qquad \forall n \in \{1, \ldots, N\} \cdot \left(\boxed{x_n \in \rho\ \Rightarrow\ \forall k \in \{1, \ldots, K\} \cdot p_n^k = x_n^k} \right) & \\
 & | & \textbf{let}\ x = de_0^{\rho}\ \textbf{in}\ de_1^{\rho \cup \{x\}} & \textbf{let}\text{–expression} \\
 & | & \lambda x.de^{\rho} & \lambda\text{–abstraction} \\
p & ::= & x \mid c\ p_1 \ldots p_N & \text{Pattern}
\end{array}
$$

Example: Distilled Matrix Multiplication

$$
\begin{array}{lcl}
\multicolumn{3}{l}{mMul\ xss\ yss} \\
\multicolumn{3}{l}{\textbf{where}} \\
mMul\ xss\ yss & = & mMul_1\ xss\ yss\ yss \\[1ex]
mMul_1\ []\ zss\ yss & = & [] \\
mMul_1\ xss\ []\ yss & = & [] \\
mMul_1\ (xs : xss)\ (zs : zss)\ yss & = & \textbf{let}\ v = \lambda xs.g\ xs \\
 & & \qquad\qquad \textbf{where} \\
 & & \qquad\qquad g\ [] \qquad\ \ = 0 \\
 & & \qquad\qquad g\ (x : xs) = x \\
 & & \textbf{in}\ (mMul_2\ zs\ xs\ yss\ v) : (mMul_1\ xss\ zss\ yss) \\[1ex]
mMul_2\ []\ xs\ yss\ v & = & [] \\
mMul_2\ (z : zs)\ xs\ yss\ v & = & \textbf{let}\ v' = \lambda xs.g\ xs \\
 & & \qquad\qquad \textbf{where} \\
 & & \qquad\qquad g\ [] \qquad\ \ = 0 \\
 & & \qquad\qquad g\ (x : xs) = v\ xs \\
 & & \textbf{in}\ (mMul_3\ xs\ yss\ v) : (mMul_2\ zs\ xs\ yss\ v') \\[1ex]
mMul_3\ []\ yss\ v & = & 0 \\
mMul_3\ (x : xs)\ []\ v & = & 0 \\
mMul_3\ (x : xs)\ (ys : yss)\ v & = & (x + (v\ ys)) + (mMul_3\ xs\ yss\ v)
\end{array}
$$

Why Encode Inputs?

- Objective = Identify skeletons in distilled program.
- Potential mismatch in the data structures and algorithms used by the skeletons and the distilled program.
- Encode pattern-matched inputs of each recursive function into a list.

Steps to Encode:

1. Declare new data type (T_f) for the encoded input of recursive function f. Encoded list type is $[T_f]$.
2. Define function ($encode_f$) to encode the inputs of f.
3. Transform function f to operate over the encoded input.

1. Declare Encoded Input Data Type (T_f)

Recursive Function

$$\begin{array}{lcl} f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N & & \\ \textbf{where} & & \\ f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N & = & e_1 \\ \vdots & & \vdots \\ f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N & = & e_K \end{array}$$

where $\exists k \in \{1, \ldots, K\} \cdot e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k \right]$

- Declare new type T_f with constructors $c_1, \ldots, c_K$.
- For each pattern $p_1^k \ldots p_M^k$ of inputs $x_1 \ldots x_M$
  1. Use create fresh constructor c_k.
  2. Variables bound by constructor c_k correspond to variables in $p_1^k \ldots p_M^k$ that occur in
     - E_k, if e_k contains a recursive call to f
     - e_k, otherwise

2. Define Encode Function ($encode_f$)

Recursive Function

$$
\begin{array}{l}
f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N \\
\textbf{where} \\
\begin{array}{lcl}
f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N & = & e_1 \\
\vdots & & \vdots \\
f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N & = & e_K
\end{array} \\
\text{where } \exists k \in \{1, \ldots, K\} \cdot e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k \right]
\end{array}
$$

- Define function $encode_f$ to pattern-match and consume inputs $x_1, \ldots, x_M$ as in f.
- For each pattern $p_1^k \ldots p_M^k$ of inputs
  1. Create encoded list element using constructor c_k.
  2. Variables bound by constructor c_k correspond to variables in $p_1^k \ldots p_M^k$ that occur in E_k or e_k.
  3. Append encoding of recursive call arguments to this encoded list element.

Example: Encoded Data Type and Encode Function

$$
\begin{aligned}
mMul_3\ \boxed{[]\ yss}\ v &= 0\\
mMul_3\ \boxed{(x : xs)\ []}\ v &= 0\\
mMul_3\ \boxed{(x{:}xs)\ (ys{:}yss)}\ v &= (x * (v\ ys)) + (mMul_3\ xs\ yss\ v)
\end{aligned}
$$

$$
\begin{aligned}
\textbf{data}\ T_{mMul_3}\ a\ ::= &\ c_6\\
\mid &\ c_7\\
\mid &\ c_8\ a\ [a]
\end{aligned}
$$

$$
\begin{aligned}
encode_{mMul_3}\ []\ yss &= [c_6]\\
encode_{mMul_3}\ (x : xs)\ [] &= [c_7]\\
encode_{mMul_3}\ (x : xs)\ (ys : yss) &= [c_8\ x\ ys] +\!\!+ (encode_{mMul_3}\ xs\ yss)
\end{aligned}
$$

3. Transform Function f

Recursive Function

$f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$
where

$$\begin{array}{lcl} f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N & = & e_1 \\ \vdots & & \vdots \\ f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N & = & e_K \end{array}$$

where $\exists k \in \{1, \ldots, K\} \cdot e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k \right]$

- Transform function f to function f' where
  1. $f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N$ is transformed to $f'\ p^k\ x_{(M+1)} \ldots x_N$
     - Pattern p^k uses c_k to match the first element of encoded list.
  2. $f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$ is transformed to $f'\ x\ x_{(M+1)} \ldots x_N$
     - x is the encoding of pattern-matched inputs $x_1, \ldots, x_M$.

Example: Encoded Function

$$
\begin{aligned}
mMul_3 \; & \boxed{[] \; yss} \; v &&= 0 \\
mMul_3 \; & \boxed{(x : xs) \; []} \; v &&= 0 \\
mMul_3 \; & \boxed{(x : xs) \; (ys : yss)} \; v &&= (x * (v \; ys)) + (mMul_3 \; \boxed{xs \; yss} \; v)
\end{aligned}
$$

$$
\begin{aligned}
mMul_3' \; & \boxed{(c_6 : \overline{x})} \; v &&= 0 \\
mMul_3' \; & \boxed{(c_7 : \overline{x})} \; v &&= 0 \\
mMul_3' \; & \boxed{((c_8 \; x \; ys) : \overline{x})} \; v &&= (x * (v \; ys)) + (mMul_3' \; \boxed{\overline{x}} \; v)
\end{aligned}
$$

Skeleton Identification

- Transformed recursive functions are defined over encoded lists.
- Identify map- and reduce-based skeletons defined over list.
- Replace skeleton instance with call to library skeleton.
  - Eden – An extension of Haskell for parallel programming.
  - Skeletons include *parMap*, *parMapReduce* and other constructs.
  - Add skeletons such as *parMapReduce1* for non-empty lists[1].

[1] An encoded list is always non-empty. Proof available in our paper.

Example: Matrix Multiplication Defined Using Skeletons

$$
\begin{array}{lcl}
mMul'_1\ (c_1 : \overline{x})\ yss & = & [] \\
mMul'_1\ (c_2 : \overline{x})\ yss & = & [] \\
mMul_1\ \big((c_3\ xs\ zs) : \overline{x}\big)\ yss & = & \textbf{let}\ v = \lambda xs.g\ xs \\
 & & \qquad\qquad \textbf{where} \\
 & & \qquad\qquad g\ [] \qquad\ = 0 \\
 & & \qquad\qquad g\ (x : xs) = x \\
 & & \textbf{in}\ (mMul'_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v) : (mMul'_1\ \overline{x}\ yss)
\end{array}
$$

$$
\begin{array}{lcl}
map\ []\ f & = & [] \\
map\ (x : xs)\ f & = & (f\ x) : (map\ xs\ f)
\end{array}
$$

$$
\begin{array}{lcl}
mMul''_1\ \overline{x}\ yss & = & parMap\ f\ \overline{x} \\
 & & \textbf{where} \\
 & & f\ c_1 \qquad\qquad = [] \\
 & & f\ c_2 \qquad\qquad = [] \\
 & & f\ (c_3\ xs\ zs) = \textbf{let}\ v = \lambda xs.g\ xs \\
 & & \qquad\qquad\qquad\qquad \textbf{where} \\
 & & \qquad\qquad\qquad\qquad g\ [] \qquad\ = 0 \\
 & & \qquad\qquad\qquad\qquad g\ (x : xs) = x \\
 & & \qquad\qquad\qquad\ \ \textbf{in}\ mMul''_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v
\end{array}
$$

Example: Matrix Multiplication Defined Using Skeletons

$$\begin{array}{lcl} mMul'_3\ (c_6 : \overline{x})\ v & = & 0 \\ mMul'_3\ (c_7 : \overline{x})\ v & = & 0 \\ mMul'_3\ \left((c_8\ x\ ys) : \overline{x}\right)\ v & = & (x * (v\ ys)) + (mMul'_3\ \overline{x}\ v) \end{array}$$

$$\begin{array}{lcl} mapRedr\ []\ g\ v\ f & = & v \\ mapRedr\ (x : xs)\ g\ v\ f & = & g\ (f\ x)\ (mapRedr\ xs\ g\ v\ f) \end{array}$$

$$\begin{array}{lcl} mMul''_3\ \overline{x}\ v & = & parMapRedr1\ g\ f\ \overline{x} \\ & & \textbf{where} \\ & & \begin{array}{lcl} g\ x\ y & = & x + y \\ f\ c_6 & = & 0 \\ f\ c_7 & = & 0 \\ f\ (c_8\ x\ ys) & = & x * (v\ ys) \end{array} \end{array}$$

Example: Matrix Multiplication Defined Using Skeletons

$$
\begin{array}{l}
mMul''\ xss\ yss \\
\textbf{where} \\
mMul''\ xss\ yss\ =\ mMul''_1\ (encode_{mMul_1}\ xss\ yss)\ yss \\
mMul''_1\ \overline{x}\ yss\ =\ \boxed{parMap\ f\ \overline{x}} \\
\quad \textbf{where} \\
\quad f\ c_1 = [] \\
\quad f\ c_2 = [] \\
\quad f\ (c_3\ xs\ zs) = \textbf{let}\ v = \lambda xs.g\ xs \\
\qquad\qquad \textbf{where} \\
\qquad\qquad g\ [] = 0 \\
\qquad\qquad g\ (x:xs) = x \\
\qquad \textbf{in}\ mMul''_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v \\
mMul''_2\ (c_4 : \overline{x})\ xs\ yss\ v = [] \\
mMul''_2\ (c_5 : \overline{x})\ xs\ yss\ v = \textbf{let}\ v' = \lambda xs.g\ xs \\
\qquad\qquad \textbf{where} \\
\qquad\qquad g\ [] = 0 \\
\qquad\qquad g\ (x:xs) = v\ xs \\
\qquad \textbf{in}\ (mMul''_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul''_2\ \overline{x}\ xs\ yss\ v') \\
mMul''_3\ \overline{x}\ v\ =\ \boxed{parMapRedr1\ g\ f\ \overline{x}} \\
\quad \textbf{where} \\
\quad g\ x\ y = x + y \\
\quad f\ c_6 = 0 \\
\quad f\ c_7 = 0 \\
\quad f\ (c_8\ x\ ys) = x * (v\ ys)
\end{array}
$$

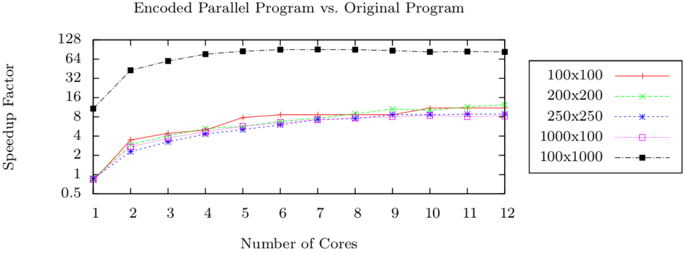
Parallel Evaluation of Skeletons

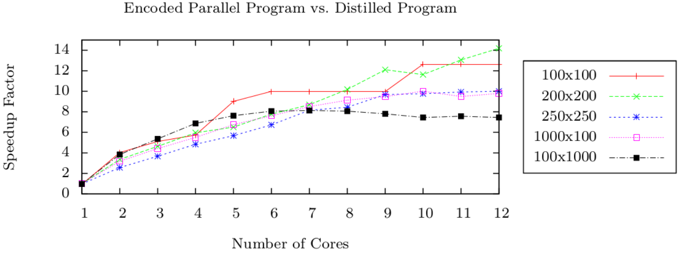
- Skeleton operators need to satisfy certain algebraic properties (such as associativity, distributivity) for parallel evaluation.
- Distillation can be used to automatically prove such properties for operators.
- For example, binary operator $\oplus :: T \rightarrow T \rightarrow T$ is associative if the following evaluates to *True*.

$$\forall x, y, z \cdot \mathcal{D}[\![\big(x \oplus (y \oplus z)\big) ==_T \big((x \oplus y) \oplus z\big)]\!]$$

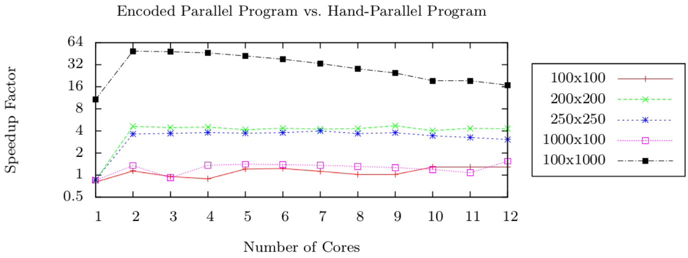
where
$\mathcal{D}$ is the distillation transformation
$==_T$ is the equality operator for type T

Evaluation of Matrix Multiplication Example





Evaluation of Matrix Multiplication Example



Summary

A transformation method with following attributes:

- Reduces inefficient intermediate data structures using *distillation*.
- Encodes all inputs into a *cons*-list.
- Facilitates matching with *map*- and *reduce*-based skeletons over list.
- *Improvements over existing work*.
  - No restrictions on programs or inputs.
  - Automatic identification of skeleton instances and operators.
  - Automatic verification of operator properties.
  - Parallel programs use fewer intermediate data structures.
- *Limitations*.
  - Potentially unbalanced encoded list in some cases.

Next Steps

- Efficient parallel execution with good load balancing.
- Potential solution
  - Encode inputs into new data structure to reflect recursive structure of function.
  - Transformed program potentially defined using skeletons over new data type.
  - Parallel implementations for polytypic skeletons.