Program Optimization and Transformation in Calculational Form

Zhenjiang Hu
University of Tokyo
July 4-5, 2005
Clarity and Efficiency

A Chinese Proverb

魚和熊掌不可同時兼得
(One cannot have both fishes and bear palms at the same time.)

- In Programming
  Clearly written programs have the desirable properties of being easier to understand, show correct, and modify, but they are often (extremely) inefficient.

- In Software Engineering
  Software with high modularity can lead to inefficiency, because of the overhead of communication between components, and because it may preclude potential optimizations across component boundaries.
A Simple Programming Problem

**Problem:** Sum up all the bigger elements in an array. An element is *bigger* if it is greater than the sum of the elements that follow it till the end of the array.

An Example:

$$[31, 4, 1, 5, 9, 2, 6] \Rightarrow 46$$
A Clear Solution in C:

/* copy all bigger elements from A[0..n-1] into B[] */
count = 0;
for (i=0; i<n; i++) {
    sumAfter = 0;
    for (j=i+1; j<n; j++) {
        sumAfter += A[j];
    }
    if (A[i] > sumAfter)
        B[count++] = A[i];
}

/* compute the sum of all elements in B[] */
sumBiggers = 0;
for (i=0; i<count; i++) {
    sumBiggers += B[i];
}
return sumBiggers;
A More Efficient Solution in C:

sumBiggers = 0;
sumAfter = 0;
for (i=n-1; i>=0; i--) {
    if (A[i] > sumAfter)
        sumBiggers += A[i];
    sumAfter += A[i];
}
return sumBiggers;
We start by writing clean and correct programs, and then use program transformation techniques to transform them step-by-step to more efficient equivalents.
Program Calculation

*Program calculation* is a kind of program transformation based on the theory of *Constructive Algorithmics*. (Bird:87, de Moor:91, Meijer:91, Fokkinga:92, Johan:93, Hu:96)
What does it mean by calculation?

Recall the manipulation of formulas as in high school algebra.

The following example shows a calculation of the solution of $x$ for the equation $x^2 - c^2 = 0$.

$$x^2 - c^2 = 0$$

$\equiv$ \{ by identity: $a^2 - b^2 = (a - b)(a + b)$ \}

$$(x - c)(x + c) = 0$$

$\equiv$ \{ by law: $ab = 0 \Leftrightarrow a = 0$ or $b = 0$ \}

$$x - c = 0 \text{ or } x + c = 0$$

$\equiv$ \{ by law: $a = b \Leftrightarrow a \pm d = b \pm d$ \}

$$x = c \text{ or } x = -c$$
Bird-Meertens Formalism (Bird:87)

A program calculus designed for

- developing identities/laws/rules for calculating programs;
- deriving correct and efficient algorithms from specification based on developed identities/laws/rules.

Proved to be Useful for Algorithm Derivation
Calculational approach is useful for automatic program optimization and transformation

- **Fusion Transformation in Calculational Form**
  Gill & Peyton Jones & Launchbury: FPCA93, Takano & Meijer: FPCA95, Hu & Iwasaki & Takeichi: ICFP96

- **Tupling Transformation in Calculational Form**
  Hu & Iwasaki & Takeichi: ICFP97, TOPLAS (97)

- **Accumulation Transformation in Calculational Form**
  Hu & Iwasaki & Takeichi: New Generation Computing (99)

- **Parallelization Transformation in Calculational Form**
  Hu & Takeichi & Chin: POPL98, Hu & Takeichi & Iwasaki: ESOP02

- **Bidirectional Transformation in Calculational Form**
  Hu & Mu & Takeichi: PEPM04, MPC04
About this Tutorial

We demonstrate how to formalize program optimizations and transformations in calculational form, with two examples:

- program optimization by loop fusion
- parallelizing program transformation

to show that program transformation in calculational form

- has higher modularity;
- is more suitable for efficient implementation.
Outline

• Introduction
• Program Calculation vs Fold/Unfold Program Transformation
• Loop Fusion in Calculational Form
• Parallelization in Calculational Form
• Implementing Program Calculation in Yicho
• Conclusion

Yicho’s Home Page:

http://www.ipl.t.u-tokyo.ac.jp/yicho/
(by Tetsuro Yokoyama)
Outline

• Introduction

• Program Calculation vs Fold/Unfold Program Transformation
  – Notation for writing programs and specifying calculation rules
  – Fold/Unfold approach to program transformation
  – Program calculation and program transformation in calculational form

• Loop Fusion in Calculational Form

• Parallelization in Calculational Form

• Implementing Program Calculation in Yicho

• Conclusion
Notation

Haskell is a popular functional language, which will be used for writing programs and specifying transformation laws/rules.

- It is *good for writing clear and modular programs*, because it supports a powerful and elegant programming style.
- It is *good for performing transformation*, because of its nice mathematical properties.
Functions

Programs are a list of *function definitions*.

\[
\begin{align*}
\text{square } x &= x \times x \\
\text{larger } x \ y &= \text{if } x > y \text{ then } x \text{ else } y
\end{align*}
\]

*Lambda expressions* are used to define a function without giving its name.

\[
\lambda x. x \times x
\]
**Functional application** is denoted by a space and the argument.

\[
\begin{align*}
\text{square } 5 & \quad \Rightarrow \quad 25 \\
\text{larger } 3 2 & \quad \Rightarrow \quad 3 \\
(\lambda x. x \times x) \ 5 & \quad \Rightarrow \quad 25
\end{align*}
\]

Functional application is regarded as *stronger binding* than any other operator.

\[
\text{square } 5 + 3 \quad = \quad (\text{square } 5) + 3 \quad \neq \quad \text{square } (5 + 3)
\]
Functional composition is denoted by a centralized circle $\circ$.

\[(f \circ g)x = f(gx)\]

Functional composition is an associative operator, and the identity function, denoted by $id$, is its unit.
Infix binary operators will often be denoted by $\oplus$, $\otimes$ and can be sectioned; an infix binary operator like $\oplus$ can be turned into unary functions as follows.

$$(a \oplus) \ b = a \oplus b = (\oplus b) \ a$$

What do the following functions denote?

$$(1+) \ (]/2) \ (== 9) \circ (1+) \circ (*2)$$
List (Array)

Lists are finite sequences of values of the same type. The type of the *cons lists* with elements of type \( a \) is defined as follows.

\[
\text{data } [a] = [] \mid a : [a]
\]

Abbreviation:

\[
[x_1, x_2, \ldots, x_n] = x_1 : (x_2 : (\ldots : (x_n : []))))
\]

List concatenation function \( + \) :

\[
[1, 2, 3] ++ [4, 5, 6] = [1, 2, 3, 4, 5, 6]
\]
Recursion

Functions may be defined recursively.

\[
\begin{align*}
\text{sort } [] & = [] \\
\text{sort } (a : x) & = \text{insert } a \ (\text{sort } x) \\
\text{insert } a \ [] & = [a] \\
\text{insert } a \ (b : x) & = \begin{cases} 
\text{if } a \geq b & \text{then } a : (b : x) \\
\text{else } b : \text{insert } a \ x & \end{cases}
\end{align*}
\]
Higher-order Functions

*Higher-order functions* are functions which can take other functions as arguments, and may also return functions as results.

\[
\text{map (1+)} \ [1, 2, 3, 4, 5] = [2, 3, 4, 5, 6]
\]
Can you understand the following Haskell program?

\[
\text{sumBiggers} = \text{sum} \circ \text{biggers}
\]

where

\[
\text{biggers } \text{[]} = \text{[]}
\]

\[
\text{biggers } (a : x) = \text{if } a > \text{sum } x \text{ then } a : \text{biggers } x \text{ else } \text{biggers } x
\]

\[
\text{sum } \text{[]} = 0
\]

\[
\text{sum } (a : x) = a + \text{sum } x
\]
How about this?

\[
\text{sumBiggers} \ x = \textbf{let } (b, c) = \text{sumBiggers'} \ x \ \textbf{in } a \\
\text{where} \\
\text{sumBiggers'} \ [] = (0, 0) \\
\text{sumBiggers'} \ (a : x) = \textbf{let } (b, c) = \text{sumBiggers'} \ x \\
\quad \quad \textbf{in } \textbf{if } a > c \ \textbf{then } (a + b, a + c) \ \textbf{else } (b, a + c)
\]
Outline

- **Introduction**
- **Program Calculation vs Fold/Unfold Program Transformation**
  - *Notation for writing programs and specifying calculation rules*
  - Fold/Unfold approach to program transformation
  - Program calculation and program transformation in calculational form
- **Loop Fusion in Calculational Form**
- **Parallelization in Calculational Form**
- **Implementing Program Calculation in Yicho**
- **Conclusion**
### Fold/Unfold Approach to Program Transformation

Transform programs (basically) by repeatedly applying unfolding rules or folding rules.

For any function definition of a program:

\[ f \ x_1 \ldots \ x_n = e \]

we have a unfolding rule:

\[ f \ x_1 \ldots \ x_n \Rightarrow e \]

and a folding rule:

\[ f \ x_1 \ldots \ x_n \Leftarrow e. \]
An Example of Fold/Unfold Transformations

A Programming Problem

Find a maximum element in a list.

A Naive Solution

Suppose that we already have \( \text{sort} \). Then, a direct solution is to sort the input and to return the first element:

\[
\text{max } x = \text{hd} (\text{sort } x)
\]

where

\[
\text{hd } [] = -\infty \\
\text{hd } (a : x) = a.
\]
Optimization by Fold/Unfold Transformations

We aim to derive a new recursive definition for \( \text{max} \).
For the base case, we have:

\[
\begin{align*}
\text{max} & \quad [] \\
= & \quad \{ \text{unfold} \ \text{max} \} \\
\text{hd} \ (\text{sort} \ [])) & \quad = \quad \{ \text{unfold} \ \text{sort} \} \\
\text{hd} \ [] & \quad = \quad \{ \text{unfold} \ \text{hd} \} \\
& \quad = \quad -\infty
\end{align*}
\]
For the recursive case, we do unfolding similarly.

\[
\begin{align*}
\text{max} (a : x) \\
&= \{ \text{unfold max} \} \\
&\text{hd (sort (a : x))} \\
&= \{ \text{unfold sort} \} \\
&\text{hd (insert a (sort x))}
\end{align*}
\]

We get stuck; we can neither unfold \text{insert} because we do not know whether \text{sort x} is empty or not, nor perform folding to get a recursive definition.

\[\Rightarrow\] To instantiate \(x\).
For the case where \( x = [] \), we can easily obtain \( \max [a] = a \).

For the case where \( x \) is not empty, we unfold \( \text{insert} \), by assuming \( b : x' = \text{sort} \ x \), that is

\[
\begin{align*}
b &= \text{hd} (\text{sort} \ x) \\
x' &= \text{tail} (\text{sort} \ x)
\end{align*}
\]
Here is the detailed transformation.

\[
\text{hd (insert } a \ (b : x')\text{)}
\]

\[
= \begin{cases} 
\text{unfold insert} \\
\quad \text{if } a \geq b \text{ then } a : (b : x')' \text{ else } b : \text{insert } a \ x'
\end{cases}
\]

\[
= \begin{cases} 
\text{law: } f (\text{if } b \text{ then } e_1 \text{ else } e_2) = \text{if } b \text{ then } f \ e_1 \text{ else } f \ e_2 \\
\quad \text{if } a \geq b \text{ then } \text{hd } (a : (b : x')) \text{ else } \text{hd } (b : \text{insert } a \ x')
\end{cases}
\]

\[
= \begin{cases} 
\text{unfold hd} \\
\quad \text{if } a \geq b \text{ then } a \text{ else } b
\end{cases}
\]

\[
= \begin{cases} 
\text{unfold } b \\
\quad \text{if } a \geq \text{hd } (\text{sort } x) \text{ then } a \text{ else } \text{hd } (\text{sort } x)
\end{cases}
\]

\[
= \begin{cases} 
\text{fold } \text{max} \\
\quad \text{if } a \geq \text{max } x \text{ then } a \text{ else } \text{max } x
\end{cases}
\]
Derived Efficient Program

\[
\begin{align*}
\quad \max [\ ] &= -\infty \\
\quad \max [a] &= a \\
\quad \max (a : x) &= \text{if } a \geq \max x \text{ then } a \text{ else } \max x \\
\end{align*}
\]

Or it is simple as follow:

\[
\begin{align*}
\quad \max [\ ] &= -\infty \\
\quad \max (a : x) &= \text{if } a \geq \max x \text{ then } a \text{ else } \max x \\
\end{align*}
\]
Limitations of Fold/Unfold Transformations

It is *general and powerful*, but suffers from several problems which often prevent it from being used in practice.

- It is *difficult to decide when unfolding steps should stop* while guaranteeing exposition of enough information for later folding steps.
- It is *expensive to implement*, because it requires keeping records of all possible folding patterns and have them checked upon any new subexpressions produced during transformation.
- Each transformation step is very small, but an effective way is *lacking to group and/or structure them into bigger steps.*
Outline

• **Introduction**

• **Program Calculation vs Fold/Unfold Program Transformation**
  – *Notation for writing programs and specifying calculation rules*
  – *Fold/Unfold approach to program transformation*
  – Program calculation and program transformation in calculational form

• Loop Fusion in Calculational Form

• Parallelization in Calculational Form

• Implementing Program Calculation in Yicho

• Conclusion
Program Transformations in Calculational Form

Fold-free Program Transformations

Transformations are based on a set of calculation laws but exclude the use of folding steps.

The challenge is how to formalize necessary folding steps by means of calculation laws.
Three-Step Formalization Procedure

1. Define a specific form of programs that are best suitable for the transformation and can be used to describe a class of interesting computations.

2. Develop calculational rules (laws) for implementing the transformation on programs in the specific form.

3. Show how to turn more general programs into those in the specific form and how to apply the newly developed calculational rules systematically.
Homomorphisms: A Generic Recursive Form

It is known that goto is considered harmful to write clear programs and to optimize programs.

Loop (recursion) should be structured for efficient manipulation!

\[
f [] = \cdots \\
f (a : x) = \cdots f x \cdots f (g x) \cdots f (f x) \cdots \\
\downarrow
\]

Composition of recursive functions in simpler form.

\[
hom_\ell [] = e \\
hom_\ell (a : x) = a \oplus hom_\ell x.
\]

\[
hom_\ell = (e, \oplus)_\ell
\]
Examples of (List) Homomorphisms

\[
\begin{align*}
    \text{sum} &= (0, +) \\
    \text{prod} &= (1, \times) \\
    \text{maxlist} &= (-\infty, \uparrow) \quad \text{where } a \uparrow r = \text{if } a \geq r \text{ then } a \text{ else } r \\
    \text{reverse} &= ([], \oplus) \quad \text{where } a \oplus r = r \oplus [a] \\
    \text{inits} &= ([[]], \oplus) \quad \text{where } a \oplus r = [] : \text{map } (a :) r \\
    \text{map } f &= ([], \oplus) \quad \text{where } a \oplus r = f a : r \\
    \text{sort} &= ([], \text{insert})
\end{align*}
\]

Compositions of homomorphisms can describe complicated computation concisely.

\[\text{mis} = \text{maxlist} \circ (\text{map } \text{sum}) \circ \text{inits}\]
Promotion: A Generic Calculation Law

\[
promotion: \quad \frac{f (a \oplus x) = a \otimes f x}{f \circ ([e, \oplus]) = ([f e, \otimes])}
\]
Revisit \(\text{max}:\) Program Calculation without Folding Steps

\[
\text{max} = \text{hd} \circ \text{sort}
\]

We may calculate as follows.

\[
\begin{align*}
\text{max} &= \{ \text{define max in terms of hom } \} \\
\text{hd} \circ ([], \text{insert}) &= \{ \text{promotion: } \forall a, x. \text{ hd (insert } a \ x) = a \otimes \text{hd } x \} \\
([\text{hd } [], \otimes])
\end{align*}
\]
The ⊗ that satisfies

\[ \forall a, x. \, \text{hd} (\text{insert} \ a \ x) = a \otimes \text{hd} \ x \]

may be obtained via a higher order matching algorithm. Here, we show another concise calculation.

\[
\begin{align*}
a \otimes b &= \{ \text{let } x \text{ be any list; by inversion } \} \\
a \otimes \text{hd} \ (b : x) &= \{ \text{the condition in the promotion rule } \} \\
\text{hd} (\text{insert} \ a \ (b : x)) &= \{ \text{definition of } \text{insert} \} \\
\text{hd} (\text{if } a \geq b \text{ then } a : (b : x) \text{ else } b : \text{insert} \ a \ x) &= \{ \text{if property } \} \\
\text{if } a \geq b \text{ then } \text{hd} \ (a : (b : x)) \text{ else } \text{hd} \ (b : \text{insert} \ a \ x) &= \{ \text{definition of } \text{hd} \} \\
\text{if } a \geq b \text{ then } a \text{ else } b
\end{align*}
\]
How to Obtain Homomorphisms?

Generally, the promotion rule can do this.

\[ f = f \circ id = f \circ ([], (:)) \]

In practice, we may need to find more efficient and systematic way.

- Warm fusion (Sheard&Launchbury:FPCA95)
- Deriving Hylomorphisms (Hu&Iwasaki&Takeichi:ICFP96)
A Note on Genericity

The framework discussed so far applies to any algebraic data types like lists and trees. We focus on lists in this tutorial.
Outline

- **Introduction**
- **Program Calculation vs Fold/Unfold Program Transformation**
  - Notation for writing programs and specifying calculation rules
  - Fold/Unfold approach to program transformation
  - Program calculation and program transformation in calculational form
- Loop Fusion in Calculational Form
- Parallelization in Calculational Form
- Implementing Program Calculation in Yicho
- Conclusion
Outline

- Introduction
- Program Calculation vs Fold/Unfold Program Transformation
- Loop Fusion in Calculational Form
  - Loop Fusion
  - Structured Recursive Form for Loop Fusion
  - Calculational Rules for Loop Fusion
  - A Calculational Algorithm for Loop Fusion
- Parallelization in Calculational Form
- Implementing Program Calculation in Yicho
- Conclusion
Loop Fusion

Loop fusion, a well-known *optimization technique in compiler construction*, is to fuse some adjacent loops into one loop to *reduce loop overhead and improve run-time performance*.

There are basically three cases for two adjacent loops:

1. *one loop is put after another* and the result computed by the first is used by the second;

2. one loop is put after another and the result computed by the first is not used by the second;

3. *one loop is used inside another*. 
A C Program with Multiple Loops

/* copy all bigger elements from A[0..n-1] into B[] */
count = 0;
for (i=0; i<n; i++) {
    sumAfter = 0;
    for (j=i+1; j<n; j++) {
        sumAfter += A[j];
    }
    if (A[i] > sumAfter)
        B[count++] = A[i];
}

/* compute the sum of all elements in B[] */
sumBiggers = 0;
for (i=0; i<count; i++) {
    sumBiggers += B[i];
}
return sumBiggers;
An Efficient C Program after Loop Fusion

```c
sumBiggers = 0;
sumAfter = 0;
for (i=n-1; i>=0; i--) {
    if (A[i] > sumAfter)
        sumBiggers += A[i];
    sumAfter += A[i];
}
return sumBiggers;
```
Multiple Loops (Recursion) in Haskell

\[
\begin{align*}
  \text{sumBiggers} &= \text{sum} \circ \text{biggers} \\
  \text{biggers } [] &= [] \\
  \text{biggers} (a : x) &= \text{if } a \geq \text{sum } x \text{ then } a : \text{biggers } x \text{ else } \text{biggers } x \\
  \text{sum } [] &= [] \\
  \text{sum} (a : x) &= a + \text{sum } x
\end{align*}
\]
An Efficient Haskell Program after Loop Fusion

\[ \text{sumBiggers } x = \text{let } (b, c) = \text{sumBiggers'} x \text{ in } a \]

\text{where}

\[ \text{sumBiggers'} [ ] = (0, 0) \]

\[ \text{sumBiggers'} (a : x) = \text{let } (b, c) = \text{sumBiggers'} x \]

\[ \text{in } \text{if } a > c \text{ then } (a + b, a + c) \text{ else } (b, a + c) \]
Outline

- **Introduction**
- **Program Calculation vs Fold/Unfold Program Transformation**
- **Loop Fusion in Calculational Form**
  - *Loop Fusion*
  - Structured Recursive Form for Loop Fusion
  - Calculational Rules for Loop Fusion
  - A Calculational Algorithm for Loop Fusion
- **Parallelization in Calculational Form**
- **Implementing Program Calculation in Yicho**
- **Conclusion**
Mutumorphism: A Structured Form for Loop Fusion

A function $f_1$ is said to be a list mutumorphism with respect to other functions $f_2, \ldots, f_n$ if each $f_i \ (i = 1, 2, \ldots, n)$ is defined in the following form:

$$
\begin{align*}
    f_i [] &= e_i \\
    f_i (a : x) &= a \oplus_i (f_1 x, f_2 x, \ldots, f_n x)
\end{align*}
$$

where $e_i \ (i = 1, 2, \ldots, n)$ are given constants and $\oplus_i \ (i = 1, 2, \ldots, n)$ are given binary functions. We represent $f_1$ as follows.

$$
    f_1 = \llbracket (e_1, \ldots, e_n), (\oplus_1, \ldots, \oplus_n) \rrbracket.
$$

Note:

$$
    \llbracket (e, \oplus) \rrbracket = \llbracket (e), (\oplus) \rrbracket
$$
An Example

From

\[
\begin{align*}
\text{biggers} & \quad = \quad [] \\
\text{biggers} (a : x) & \quad = \quad \text{if } a \geq \text{sum } x \text{ then } a : \text{biggers } x \text{ else } \text{biggers } x \\
\text{sum} & \quad = \quad [] \\
\text{sum} (a : x) & \quad = \quad a + \text{sum } x
\end{align*}
\]

we have

\[
\text{biggers} = [(\emptyset, 0), (\oplus_1, \oplus_2)]
\]

\[
\begin{align*}
\text{where } a \oplus_1 (r, s) & \quad = \quad \text{if } a \geq s \text{ then } a : r \text{ else } r \\
a \oplus_2 (r, s) & \quad = \quad a + s
\end{align*}
\]
Outline

• Introduction

• Program Calculation vs Fold/Unfold Program Transformation

• Loop Fusion in Calculational Form
  – Loop Fusion
  – Structured Recursive Form for Loop Fusion
  – Calculational Rules for Loop Fusion
  – A Calculational Algorithm for Loop Fusion

• Parallelization in Calculational Form

• Implementing Program Calculation in Yicho

• Conclusion
Calculational Rules for Loop Fusion

Flatten: dealing with nested loops

\[ [(e_1, e_2, \ldots, e_n), (\oplus_1, \oplus_2, \ldots, \oplus_n)] = \text{fst} \circ [(e_1, e_2, \ldots, e_n), \oplus] \]

where \( a \oplus r = (a \oplus_1 r, a \oplus_2 r, \ldots, a \oplus_n r) \)

Here, \( \text{fst} \) is a projection function returning the first element of a tuple.
An Example

Consider to apply the flattening rule to `biggers` to flatten the nested loop.

\[
\text{biggers} = \{ \text{mutumorphism for } \text{biggers} \} \\
\llbracket ([], 0), (\oplus_1, \oplus_2) \rrbracket \\
= \{ \text{flattening rule} \} \\
fst \circ \llbracket ([], 0), \oplus \rrbracket \\
\text{where } a \oplus (r, s) = (\text{if } a \geq s \text{ then } a : r \text{ else } r, a + s)
\]
Inlining the homomorphism in the derived program gives the following readable recursive program, which consists of a single loop.

\[
\text{biggers } x = \text{let } (r, s) = \text{hom } x \text{ in } r \\
\text{\hspace{1cm} \textbf{where} hom } [] = ([], 0) \\
\text{\hspace{1cm} hom } (a : x) = \text{let } (r, s) = \text{hom } x \\
\text{\hspace{1cm} \hspace{1cm} in } (\text{if } a \geq s \text{ then } a : r \text{ else } r, a + s)
\]
Tupling: dealing with adjacent independent loops

\[ f([e_1, \oplus_1] \ x , \ [e_2, \oplus_2] \ x) = f([((e_1, e_2), \oplus)] \ x) \]

where \( a \oplus (r_1, r_2) = (a \oplus_1 r_1, a \oplus_2 r_2) \)
An Example

The following program is to compute the average of a list:

\[
\text{average } x = \text{sum } x / \text{length } x
\]

which has two loops can be merged into a single loop by applying the tupling rule.

\[
\text{average } x = \text{let } (s, l) = \text{tup } x \text{ in } s/l
\]

\[
\text{where } \text{tup } = \{ (0, 0), \lambda a \ (s, l). \ (a + s, 1 + l) \}\]
Fusion: dealing with adjacent dependent loops

\[(e, \oplus) \circ build \; g = g \; (e, \oplus)\]

where

\[build \; g = g \; ([], (:))\]

Here the build-form can be obtained by \textit{promotion}:

\[[d, \otimes] = build \; (\lambda (c, \circ). \; ([c, \circ]) \circ ([d, \otimes]))\]
An Example

Recall that we have obtained the following definition for `biggers`.

\[
\text{bigger} = \text{fst} \circ \left( \left[\left(\left[\right], 0\right), \oplus\right] \right)
\]

where

\[
a \oplus (r, s) = \begin{cases} 
a : r & \text{if } a \geq s \\
r, a + s & \text{else}
\end{cases}
\]

We can obtain the following build form:

\[
\text{bigger} = \text{build} \left( \lambda (c, \odot). \text{fst} \circ \left[\left(c, 0\right), \oplus'\right] \right)
\]

where

\[
a \oplus' (r, s) = \begin{cases} 
a \odot r & \text{if } a \geq s \\
r, a + s & \text{else}
\end{cases}
\]
Now applying the shortcut fusion rule to

\[ \text{sumBiggers} = ([0, +]) \circ \text{bigger} \]

soon yields the following single-loop program for \text{sumBiggers}:

\[ \text{sumBiggers} = \text{fst} \circ [(0, 0), \otimes] \]

\textbf{where} \( a \otimes (r, s) = (\text{if } a \geq s \text{ then } a + r \text{ else } r, a + s) \)

which is actually the same as that in the introduction if we inline it.
Outline

- Introduction
- Program Calculation vs Fold/Unfold Program Transformation
- Loop Fusion in Calculational Form
  - Loop Fusion
  - Structured Recursive Form for Loop Fusion
  - Calculational Rules for Loop Fusion
  - A Calculational Algorithm for Loop Fusion
- Parallelization in Calculational Form
- Implementing Program Calculation in Yicho
- Conclusion
A Calculational Algorithm for Loop Fusion

1. **Represent** as many recursive functions on lists by mutumorphisms as possible.

2. Apply the **flattening rule** to transform all mutumorphism to homomorphisms.

3. Apply the **promotion rule and shortcut fusion rule** as much as possible.

4. Apply the **tupling rule** to merge independent homomorphisms.

5. **Inline** homomorphism/mutumorphism to output transformed program in a friendly manner.

*Note:* A similar algorithm was implemented in Glasgow Haskell Compiler (The Hylo System by Onoue, 1997); References: ICFP’96, ICFP’97.
Example:

\[ \text{sumBiggers} \quad = \quad \text{sum} \circ \text{biggers} \]
\[ = \quad \{ \text{represent list functions by mutumorphism/homomorphism} \} \]
\[ ([0, +]) \circ [(\emptyset, 0), (\oplus_1, \oplus_2)] \]
\[ \textbf{where} \quad a \oplus_1 (r, s) = \text{if} \ a \geq s \ \text{then} \ a : r \ \text{else} \ r \]
\[ a \oplus_2 (r, s) = a + s \]
\[ = \quad \{ \text{flatten: } a \otimes (r, s) = (\text{if} \ a \geq s \ \text{then} \ a + r \ \text{else} \ r, a + s) \} \]
\[ ([0, +]) \circ \text{fst} \circ [(0, 0), \otimes] \]
\[ = \quad \{ \text{make "build" form} \} \]
\[ ([0, +]) \circ \text{build} (\lambda(c, \odot). \text{fst} \circ [(c, 0), \oplus']) \]
\[ \textbf{where} \quad a \oplus' (r, s) = (\text{if} \ a \geq s \ \text{then} \ a \odot r \ \text{else} \ r, a + s) \]
\[ = \quad \{ \text{fusion} \} \]
\[ \text{fst} \circ [(0, 0), \otimes] \]
\[ \textbf{where} \quad a \otimes (r, s) = (\text{if} \ a \geq s \ \text{then} \ a + r \ \text{else} \ r, a + s) \]
Outline

- Introduction
- Program Calculation vs Fold/Unfold Program Transformation
- Loop Fusion in Calculational Form
  - Loop Fusion
  - Structured Recursive Form for Loop Fusion
  - Calculational Rules for Loop Fusion
  - A Calculational Algorithm for Loop Fusion
- Parallelization in Calculational Form
- Implementing Program Calculation in Yicho
- Conclusion
Outline

• Introduction
• Program Calculation vs Fold/Unfold Program Transformation
• Loop Fusion in Calculational Form
• Parallelization in Calculational Form
  – Parallelization Transformation
  – J-Homomorphism: A Parallel Form for List Functions
  – A Parallelizing Rule
  – A Calculation Algorithm for Parallelization
• Implementing Program Calculation in Yicho
• Conclusion
Parallelization

Parallelization is a transformation for automatically generating parallel code from high level sequential description.

\[
\text{A Sequential Program} \implies \text{A Parallel Program}
\]

It is a big challenge to clarify

- what kind of sequential programs can be parallelized
- how they can be systematically parallelized.
Parallelization of List Functions

Parallelization is a transformation for automatically generating parallel code from high level sequential description manipulating lists.

A Sequential Program
\[ f :: [a] \rightarrow R \]

\[ \Rightarrow \]

A Parallel Program
\[ f :: [a] \rightarrow R \]
Parallelization of List Functions (Cont)

A hint from Constructive Algorithmics:
The control structure of a program should be determined by the data structure the program is to manipulate.

A Sequential Program
\[ f :: \text{SeqList} \ a \rightarrow R \]
⇒
A Parallel Program
\[ f' :: \text{ParaList} \ a \rightarrow R \]
Data Refinement

A *sequential* view of lists:

\[ \text{ConsList } a = [ ] | a : \text{ConsList } a \]

A *parallel* view of lists:

\[ \text{JoinList } a = [ ] | [. ] a | \text{JoinList } a ++ \text{JoinList } a \]

An Example

Given a list \([1, 2, 3, 4, 5, 6]\), we may represent it in the following two ways:

\[
1 : (2 : (3 : (4 : (5 : (6 : [ ])))))
\]
\[
([1] ++ [2] ++ [3]) ++ ([4] ++ [5] ++ [6])
\]
A Simple Example of Parallelization

Programs defined on cons lists *inherit sequentiality from cons lists*, while programs defined on join lists *gain parallelism from join lists*.

\[
\begin{align*}
\text{sumS} \; [] & = 0 \\
\text{sumS} \; (a : x) & = a + \text{sumS} \; x \\
\end{align*}
\]

\[
\downarrow
\]

\[
\begin{align*}
\text{sumP} \; [] & = 0 \\
\text{sumP} \; [a] & = a \\
\text{sumP} \; (x + + y) & = \text{sumP} \; x + \text{sumP} \; y \\
\end{align*}
\]
Running Example: the Maximum Segment Sum Problem

Compute the maximum of the sums of contiguous segments within a list of integers. For example,

\[ mss [3, -4, 2, -1, 6, -3] = 7 \]

A Sequential Program:

\[
\begin{align*}
    mss [] & = 0 \\
    mss (a : x) & = a \uparrow (a + mis x) \uparrow mss x \\
    mis [] & = 0 \\
    mis (a : x) & = a \uparrow (a + mis x)
\end{align*}
\]
Outline

- Introduction
- Program Calculation vs Fold/Unfold Program Transformation
- Loop Fusion in Calculational Form
- Parallelization in Calculational Form
  - Parallelization Transformation
  - J-Homomorphism: A Parallel Form for List Functions
  - A Parallelizing Rule
  - A Calculation Algorithm for Parallelization
- Implementing Program Calculation in Yicho
- Conclusion
J-Homomorphism: A Parallel Form for List Functions

\textit{J-homomorphisms (Homomorphisms on JoinList)} are functions defined in the following form:

\[ h(x ++ y) = h x \oplus h y \]

where \( \oplus \) is an associative operator.
A Calculation Rule for Parallelization

We aim at a way of expressing a homomorphism in terms of J-homomorphisms. The challenge is how to obtain an associative operator required in J-homomorphism.

Composition-closed Functions

Let $\overline{x_{i_1}^n}$ denote a sequence $x_1 \ x_2 \ \cdots \ x_n$. A function $f \ \overline{x_{i_1}^n} \ r$ is said to be composition-closed if there exist $n$ functions $g_i \ (i = 1, \cdots, n)$, so that

$$f \ \overline{x_{i_1}^n} \ (f \ \overline{y_{i_1}^n} \ r) = f \ (g_i \ \overline{x_{i_1}^n} \ \overline{y_{i_1}^n})^n_1 \ r$$
Example: a composition-closed function

\[ f \ x_1 \ x_2 \ r = x_1 \uparrow (x_2 + r) \]

because

\[
\begin{align*}
& f \ x_1 \ x_2 \ (f \ y_1 \ y_2 \ r) \\
= & \quad \{\text{definition of } f\}\ 
\ x_1 \uparrow (x_2 + (y_1 \uparrow (y_2 + r))) \\
= & \quad \{\text{since } a + (b \uparrow c) = (a + b) \uparrow (a + c)\}\ 
\ x_1 \uparrow ((x_2 + y_1) \uparrow (x_2 + (y_2 + r))) \\
= & \quad \{\text{associativity of } \uparrow \text{ and } +\}\ 
(x_1 \uparrow (x_2 + y_1)) \uparrow ((x_2 + y_2) + r) \\
= & \quad \{\text{define } g_1 \ x_1 \ x_2 \ y_1 \ y_2 = (x_1 \uparrow (x_2 + y_1), \ g_2 \ x_1 \ x_2 \ y_1 \ y_2 = x_2 + y_2\}\ 
(g_1 \ x_1 \ x_2 \ y_1 \ y_2) \uparrow (g_2 \ x_1 \ x_2 \ y_1 \ y_2 + r) \\
= & \quad \{\text{definition of } f\}\ 
\ f \ (g_1 \ x_1 \ x_2 \ y_1 \ y_2) \ (g_2 \ x_1 \ x_2 \ y_1 \ y_2) \ r
\end{align*}
\]
A Parallelization Rule [POPL98]

Given a homomorphism $([e, \oplus])$, if there exists a composition-closed function $f$ with respect to $g_1, g_2, \ldots, g_n$, such that

$$ a \oplus r = f \ e_1^n r $$

then

$$ ([e, \oplus]) x = \text{let } (a_1, a_2, \ldots, a_n) = h x \text{ in } f a_1 a_2 \cdots a_n e $$

$$ h[a] = (e_1, e_2, \ldots, e_n) $$

$$ h(x \oplus y) = h x \otimes h y $$

where $x_1^n \otimes y_1^n = g_i \ x_1^n y_1^n$
Example: parallelization of $mis$

The initial program:

$$mis \[] = 0 \quad mis \(a : x\) = a \uparrow (a + mis \ x)$$

which is in fact a homomorphism:

$$mis = ([0, \oplus]) \text{ where } a \oplus r = a \uparrow (a + r)$$

The difficulty is to find a composition-closed function from $\oplus$. In fact, such function $f$ is

$$f \ x_1 \ x_2 \ r = x_1 \uparrow (x_2 + r)$$

whose composition-closed property has been shown. Now we have

$$a \oplus r = f \ a \ a \ r.$$
Applying the parallelization rule to $mis$ gives the following parallel program:

$$mis \ x = \textbf{let} \ (a_1, a_2) = h \ x \ \textbf{in} \ a_1 \uparrow (a_2 + e)$$

where

$$h [a] = (a, a)$$
$$h (x \oplus y) = h x \otimes h y$$

where $$(x_1, x_2) \otimes (y_1, y_2) = (x_1 \uparrow (x_2 + y_1), x_2 + y_2).$$
Outline

• Introduction

• Program Calculation vs Fold/Unfold Program Transformation

• Loop Fusion in Calculational Form

• Parallelization in Calculational Form
  – Parallelization Transformation
  – J-Homomorphism: A Parallel Form for List Functions
  – A Parallelizing Rule
  – A Calculation Algorithm for Parallelization

• Implementing Program Calculation in Yicho

• Conclusion
A Calculation Algorithm for Parallelization

1. Apply the \textit{loop fusion calculation} to the program to obtain a compact program defined in terms of \textit{homomorphisms}.

2. Derive \textit{composition-closed functions} from homomorphisms [APLAS04].

3. Apply the \textit{parallelizing rule} to map homomorphisms to J-homomorphisms.
Example: parallelizing $mss$

\[
\begin{align*}
    mss & = 0 \\
    mss (a : x) & = a \uparrow (a + mis x) \uparrow mss x \\
    mis & = 0 \\
    mis (a : x) & = a \uparrow (a + mis x)
\end{align*}
\]
Step 1: Loop fusion calculation

\[ mss = \text{fst} \circ mss\_mis \]

where \( mss\_mis \) is the homomorphism defined below:

\[ mss\_mis = \left[ (0,0), \oplus \right] \]

where \( a \oplus (s, i) = (a \uparrow (a + i) \uparrow s, a \uparrow (a + i)) \).
Step 2: Derivation of composition-closed functions [APLAS04]

\[ a \oplus (s, i) = f \ a\ a\ 0\ a\ a\ (i, s) \]

where \( f \) is a composition-closed function defined by

\[ f\ x_1\ x_2\ x_3\ x_4\ x_5\ (s, i) = (x_1 \uparrow (x_2 + i) \uparrow (x_3 + s), x_4 \uparrow (x_5 + i)) \]

with respect to \( g_1, g_2, g_3, g_4, g_5 \):

\[
\begin{align*}
  g_1\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = x_1 \uparrow (x_2 + y_4) \uparrow (x_3 + y_1) \\
  g_2\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = (x_2 + y_5) \uparrow (x_3 + y_2) \\
  g_3\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = x_3 + y_3 \\
  g_4\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = x_4 \uparrow (x_5 + y_4) \\
  g_5\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = x_5 + y_5
\end{align*}
\]
Step 3: Application of the parallelization rule

\( mss\_mis \ x = \textbf{let} \ (a_1, a_2, a_3, a_4, a_5) = h \ x \ \textbf{in} \ f \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ (0, 0) \)

where \( h \) is a J-homomorphism defined as follows.

\[
\begin{align*}
    h[\mathbf{a}] &= (a, a, 0, a, a) \\
    h(x \oplus y) &= h\ x \otimes h\ y \\
    \textbf{where} \quad & (x_1, x_2, x_3, x_4, x_5) \otimes (y_1, y_2, y_3, y_4, y_5) \\
    &= (x_1 \uparrow (x_2 + y_4) \uparrow (x_3 + y_1), \\
    & \quad (x_2 + y_5) \uparrow (x_3 + y_2), \\
    & \quad x_3 + y_3, \\
    & \quad x_4 \uparrow (x_5 + y_4), \\
    & \quad x_5 + y_5)
\end{align*}
\]
Outline

• Introduction

• Program Calculation vs Fold/Unfold Program Transformation

• Loop Fusion in Calculational Form

• Parallelization in Calculational Form
  – Parallelization Transformation
  – J-Homomorphism: A Parallel Form for List Functions
  – A Parallelizing Rule
  – A Calculation Algorithm for Parallelization

• Implementing Program Calculation in Yicho

• Conclusion
Outline

- Introduction
- Program Calculation vs Fold/Unfold Program Transformation
- Loop Fusion in Calculational Form
- Parallelization in Calculational Form
- Implementing Program Calculation in Yicho
  - Overview of Yicho
  - Program Representation
  - Basic Combinators for Programming Calculations
  - Programming Calculation Rules in Yicho
- Conclusion
Yicho

Yicho is designed and implemented for supporting

**direct and efficient implementation of calculation rules in Haskell**

with

![](deterministic_higher-order_patterns)

It is built upon *Template Haskell*, and implemented by *Tetsuro Yokoyama*. 
Yicho Website:

Yicho: A Combinator Library for Program Calculation

Overview

Yicho is a monadic combinator library for supporting declarative specification of program transformation in Haskell. The combinator library uses higher-order patterns as first-class values which can be passed as parameters, constructed by smaller ones in compositional way, and returned as values. As a result, our library provides more flexible binding than simple ones, and enables more abstract and modular description of program transformation. Our library is developed by Template Haskell, a new extension to Haskell 98.

Documents

- Hierarchical Module Structure
- User’s manual (under construction)
- Q&A (under construction)

Download

- Version 0.1.6: Source distribution (.tgz) (requires GHC version 6.4 or later to compile).

Reference papers

1. Zhenjiang Fu, Tetsuo Yokoyama, and Masato Takeuchi, Program Optimizations and Transformations in Calculational Form (Tutorial Paper), Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2003), Japan.
Program Representation in Template Haskell

Quote and Unquote

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int} \\
[| \text{sum} |] :: \text{Q Exp} \\
\$ (\[[| \text{sum} |]\]) :: [\text{Int}] \rightarrow \text{Int}
\]
Representation of Function Definitions

\[
def =
[\text{d}]
\]

\[
\text{max} = \text{hd} \cdot \text{sort}
\]

\[
\text{sort} \; [] = []
\]
\[
\text{sort} \; (a:x) = \text{insert} \; a \; (\text{sort} \; x)
\]

\[
\text{insert} \; a \; [] = b
\]
\[
\text{insert} \; a \; (b:x) = \begin{cases} 
\text{if} \; a \; \geq \; b \; \text{then} \; a : (b : x) \\
\text{else} \; b : \text{insert} \; a \; x
\end{cases}
\]

\]
Outline

• Introduction
• Program Calculation vs Fold/Unfold Program Transformation
• Loop Fusion in Calculational Form
• Parallelization in Calculational Form
• Implementing Program Calculation in Yicho
  – Overview of Yicho
  – Program Representation
  – Basic Combinators for Programming Calculations
  – Programming Calculation Rules in Yicho
• Conclusion
Basic Combinators for Programming Calculations

Calculation Monad $Y$

To capture updating of transformation environments and to handle exceptions that occur during transformation.

\[
\begin{align*}
ret & : Q \text{ Exp} \rightarrow Y (Q \text{ Exp}) \\
runY & : Y (Q \text{ Exp}) \rightarrow Q \text{ Exp}
\end{align*}
\]

Note: $\text{Exp}Q = Q \text{ Exp}$, $\text{Exp}Y = Y \text{ Exp}Q$. 
Useful Combinators for Coding Calculation

- **Match** \( (<==) :: \text{ExpQ} \rightarrow \text{ExpQ} \rightarrow \text{Y} () \)
- **Rule** \( (==>) :: \text{ExpQ} \rightarrow \text{ExpQ} \rightarrow \text{RuleY} \)
- **Sequence** \( (>>>) :: \text{Y} () \rightarrow \text{Y} () \rightarrow \text{Y} () \)
- **Choice** \( (<+) :: \text{ExpY} \rightarrow \text{ExpY} \rightarrow \text{ExpY} \)
- **Case** \( \text{casem} :: \text{ExpQ} \rightarrow [\text{RuleY}] \rightarrow \text{ExpY} \)
Match

The most essential combinator used to *match a pattern with a term* and produce a substitution (embedded in monadic $Y$).

An Example

\[
\lambda a \ x \rightarrow \oplus a \ (bigger x, \sum x) \ \ |
\]

\[
\leq \rightarrow \lambda a \ x \rightarrow \begin{cases} 
  a \geq \sum x \text{ then } a : bigger x \\
  \text{else } bigger x 
\end{cases} 
\]

This will yield the following substitution embedded in $Y$.

\[
\{ \oplus := \lambda x (b, s) \rightarrow \\
\quad \begin{cases} 
  x \geq s \text{ then } x : b \\
  \text{else } b 
\end{cases} \}. 
\]
Rule

Used to create a calculation rule mapping from one program pattern to another.

An Example

\[
[| \text{hom } e \; $oplus \; . \; \text{build } g |] \implies [| g \; e \; $oplus |]
\]

Note: Rule can be defined by Match.

\[
(\implies) :: \text{ExpQ} \to \text{ExpQ} \to \text{RuleY}
\]

(\text{lhs} \implies \text{rhs}) \text{ term } = \text{do } \text{lhs} \leftarrow \text{term}

\hspace{1cm} \text{ret rhs}

Choise & Casem

Used to express deterministic choice.

\[(\text{rule1 } e) \leftarrow (\text{rule2 } e)\]

\[
\text{casem} :: \text{ExpQ} \to [\text{RuleY}] \to \text{ExpY}
\]

\[
\text{casem } \text{sel} \ (r:rs) = r \ \text{sel} \leftarrow \text{casem sel rs}
\]
Code Calculation Rules in Yicho

Code the promotion rule

\[
promotion: \quad f(a \oplus x) = a \otimes f(x)
\]

\[
f \circ \text{foldr} (\oplus) e = \text{foldr} (\otimes) (f \ e)
\]

\[
promotion :: \text{ExpQ} \rightarrow \text{ExpY}
promotion \ exp = \ do
\[\ [f,oplus,e,otimes] <- \ pvars \ ["f","oplus","e","otimes"]\]
\[\ [| \ f . \ \text{foldr} \ \oplus \ e \ |] <= \ exp\]
\[\ [| \ a \ x -> \ \text{otimes} \ a \ (f \ x) \ |] \]
\[\ <= \ [| \ a \ x -> \ f \ (\oplus a \ x) \ |] \]
ret [\ [| \ \text{foldr} \ \text{otimes} \ (f \ z) \ |] \]
Enhance the promotion with an additional rule

```
promotionWithRule :: RuleY -> ExpQ -> ExpY
promotionWithRule rule exp = do
    [f,oplus,e,otimes] <- pvars ["f","oplus","e","otimes"]
    [| $f . foldr $oplus $e |] <=< rule exp
    [| \a x -> $otimes a ($f x) |]
    <=< rule [| \a x -> $f ($oplus a x) |]
    ret [| foldr $otimes ($f $z) |]
```
Run it!

oldExp = [\ | sum . foldr (\x y -> 2 * x : y) [] |]  
newExp = runY (promotionWithRule rule oldExp)

⇒

GHCi> prettyExpQ newExp
foldr (\x_1 -> (+) (2 * x_1)) 0

GHCi> $oldExp (take 100000 [1..])
10000100000
(0.33 secs, 21243136 bytes)

GHCi> $newExp (take 100000 [1..])
10000100000
(0.27 secs, 19581216 bytes)
Try it!

- Step 1: Download Yicho
- Step 2: Uncompress the source
- Step 3: Add to your module `Import Yicho`

All the calculations in this tutorial has been implemented in Yicho.

```bash
> ghci -fglasgow-exts Examples/Main.hs
...
GHCi> all_examples
```
Outline

• Introduction
• Program Calculation vs Fold/Unfold Program Transformation
• Loop Fusion in Calculational Form
• Parallelization in Calculational Form
• Implementing Program Calculation in Yicho
  – Overview of Yicho
  – Program Representation
  – Basic Combinators for Programming Calculations
  – Programming Calculation Rules in Yicho
• Conclusion
Conclusion

Important Points

- Program calculation is a *fold free* program transformation.
- To formalize a program transformation in calculational form, one may first define a suitable form for the program, then develop calculation rules to capture the essence of the transformation, and finally construct a calculation algorithm.
- Program calculation can be implemented directly and efficiently.
Advantages of Program Transformations in Calculational Form

- **Modularity**: local analysis, local rule application
- **Generality**: polytypic, extendability
- **Cheap Implementation**: simple rule application
- **Compatibility**: all based on constructive algorithmics

*We believe that more optimizations and transformations can be formalized in calculational form to gain the advantages discussed above, and we are looking forward to see more practical applications.*
Thank You!