# 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:

$$[\mathbf{31}, 4, 1, 5, \mathbf{9}, 2, \mathbf{6}] \;\Rightarrow\; 46$$

## 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;
```

# Transformational Programming

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

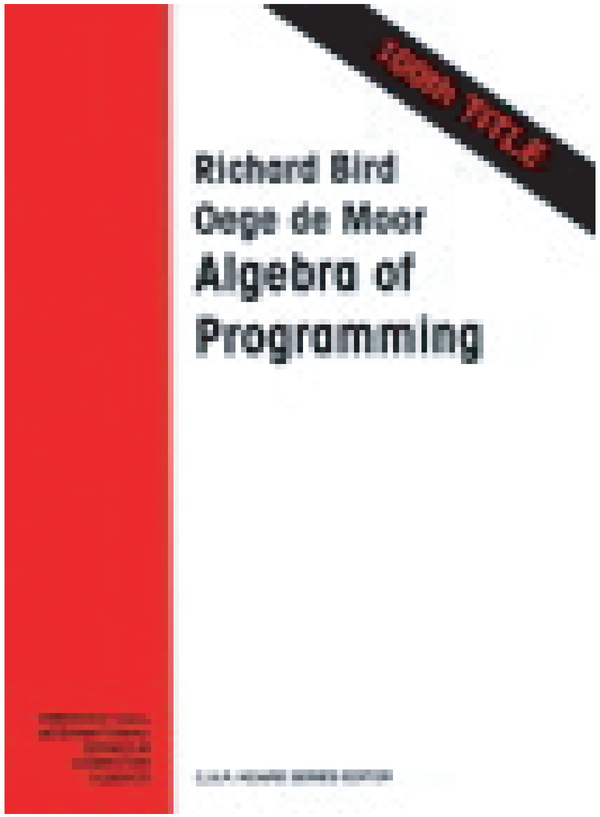
$\Downarrow$

魚和熊掌**可不**同時兼得
(One **can** have both fishes and bear palms **not** at the same time.)

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

$$
\begin{array}{ll}
 & x^2 - c^2 = 0 \\
\equiv & \quad \{ \text{ by identity: } a^2 - b^2 = (a-b)(a+b) \ \} \\
 & (x-c)(x+c) = 0 \\
\equiv & \quad \{ \text{ by law: } ab = 0 \Leftrightarrow a = 0 \text{ or } b = 0 \ \} \\
 & x - c = 0 \text{ or } x + c = 0 \\
\equiv & \quad \{ \text{ by law: } a = b \Leftrightarrow a \pm d = b \pm d \ \} \\
 & x = c \text{ or } x = -c
\end{array}
$$

## 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*.

$$\begin{array}{lcl} \mathit{square}\ x & = & x * x \\ \mathit{larger}\ x\ y & = & \textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y \end{array}$$

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

$$\lambda x.\, x * x$$

*Functional application* is denoted by a space and the argument.

$$\begin{array}{lcl} \mathit{square}\ 5 & \Rightarrow & 25 \\ \mathit{larger}\ 3\ 2 & \Rightarrow & 3 \\ (\lambda x.\ x * x)\ 5 & \Rightarrow & 25 \end{array}$$

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

$$\mathit{square}\ 5 + 3 \ = \ (\mathit{square}\ 5) + 3 \ \neq \ \mathit{square}\ (5 + 3)$$

*Functional composition* is denoted by a centralized circle $\circ$.

$$(f \circ g)\, x = f\, (g\, x)$$

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.

$$\textbf{data}\ [a] = [\,]\ \mid\ a : [a]$$

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.

$$
\begin{array}{lcl}
\mathit{sort}\ [\,] & = & [\,] \\
\mathit{sort}\ (a : x) & = & \mathit{insert}\ a\ (\mathit{sort}\ x) \\
\\
\mathit{insert}\ a\ [\,] & = & [a] \\
\mathit{insert}\ a\ (b : x) & = & \textbf{if}\ a \geq b\ \textbf{then}\ a : (b : x) \\
 & & \textbf{else}\ b : \mathit{insert}\ a\ x
\end{array}
$$

## Higher-order Functions

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

$$map\ (1+)\ [1,2,3,4,5] = [2,3,4,5,6]$$

Can you understand the following Haskell program?

$$
\begin{array}{l}
sumBiggers = sum \circ biggers \\
\quad \textbf{where} \\
\qquad biggers\ [\,] = [\,] \\
\qquad biggers\ (a : x) = \textbf{if}\ a > sum\ x\ \textbf{then}\ a : biggers\ x\ \textbf{else}\ biggers\ x \\
\qquad sum\ [\,] = 0 \\
\qquad sum\ (a : x) = a + sum\ x
\end{array}
$$

How about this?

$$
\begin{aligned}
& \mathit{sumBiggers}\ x = \mathbf{let}\ (b, c) = \mathit{sumBiggers'}\ x\ \mathbf{in}\ a \\
& \quad \mathbf{where} \\
& \qquad \mathit{sumBiggers'}\ [] = (0, 0) \\
& \qquad \mathit{sumBiggers'}\ (a : x) = \ \mathbf{let}\ (b, c) = \mathit{sumBiggers'}\ x \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathbf{if}\ a > c\ \mathbf{then}\ (a + b, a + c)\ \mathbf{else}\ (b, a + c)
\end{aligned}
$$

# 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 *sort*. Then, a direct solution is to sort the input and to return the first element:

$$max\ x \;=\; hd\ (sort\ x)$$

where

$$\begin{array}{lcl} hd\ [] & = & -\infty \\ hd\ (a : x) & = & a. \end{array}$$

## Optimization by Fold/Unfold Transformations

We aim to derive a new recursive definition for $max$.
For the base case, we have:

$$
\begin{array}{ll}
 & max\ [] \\
= & \quad \{ \text{ unfold } max \ \} \\
 & hd\ (sort\ []) \\
= & \quad \{ \text{ unfold } sort \ \} \\
 & hd\ [] \\
= & \quad \{ \text{ unfold } hd \ \} \\
 & -\infty
\end{array}
$$

For the recursive case, we do unfolding similarly.

$$
\begin{array}{ll}
 & max\ (a : x) \\
= & \quad \{\ \text{unfold}\ max\ \} \\
 & hd\ (sort\ (a : x)) \\
= & \quad \{\ \text{unfold}\ sort\ \} \\
 & hd\ (insert\ a\ (sort\ x))
\end{array}
$$

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

$\Longrightarrow$ 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 *insert*, by assuming $b : x' = sort\ x$, that is

$$\begin{array}{lcl} b & = & hd\ (sort\ x) \\ x' & = & tail\ (sort\ x) \end{array}$$

Here is the detailed transformation.

$$
\begin{array}{cl}
& hd\ (insert\ a\ (b : x')) \\
= & \quad \{ \text{unfold } insert \} \\
& hd\ (\textbf{if}\ a \geq b\ \textbf{then}\ a : (b : x)'\ \textbf{else}\ b : insert\ a\ x') \\
= & \quad \{ \text{law: } f\ (\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = \textbf{if}\ b\ \textbf{then}\ f\ e_1\ \textbf{else}\ f\ e_2 \} \\
& \textbf{if}\ a \geq b\ \textbf{then}\ hd\ (a : (b : x'))\ \textbf{else}\ hd\ (b : insert\ a\ x') \\
= & \quad \{ \text{unfold } hd \} \\
& \textbf{if}\ a \geq b\ \textbf{then}\ a\ \textbf{else}\ b \\
= & \quad \{ \text{unfold } b \} \\
& \textbf{if}\ a \geq hd\ (sort\ x)\ \textbf{then}\ a\ \textbf{else}\ hd\ (sort\ x) \\
= & \quad \{ \textbf{fold}\ max \} \\
& \textbf{if}\ a \geq max\ x\ \textbf{then}\ a\ \textbf{else}\ max\ x
\end{array}
$$

## Derived Efficient Program

$$
\begin{aligned}
max\ [] &= -\infty \\
max\ [a] &= a \\
max\ (a : x) &= \textbf{if}\ a \geq max\ x\ \textbf{then}\ a\ \textbf{else}\ max\ x
\end{aligned}
$$

Or it is simple as follow:

$$
\begin{aligned}
max\ [] &= -\infty \\
max\ (a : x) &= \textbf{if}\ a \geq max\ x\ \textbf{then}\ a\ \textbf{else}\ max\ x
\end{aligned}
$$

## 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!

$$
\begin{array}{lcl}
f\ [\,] & = & \cdots \\
f\ (a : x) & = & \cdots f\ x \cdots f\ (g\ x) \cdots f\ (f\ x) \cdots
\end{array}
$$

$$\Downarrow$$

Composition of recursive functions in simpler form.

$$
\begin{array}{lcl}
hom_l\ [\,] & = & e \\
hom_l\ (a : x) & = & a \oplus hom_l\ x.
\end{array}
$$

$$\boxed{hom_l = (\![e, \oplus]\!)_l}$$

## Examples of (List) Homomorphisms

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

*Compositions of homomorphisms* can describe complicated computation concisely.

$$mis = maxlist \circ (map\ sum) \circ inits$$

# Promotion: A Generic Calculation Law

promotion: $$\frac{f\ (a \oplus x) = a \otimes f\ x}{f \circ (\![e, \oplus]\!) = (\![f\ e, \otimes]\!)}$$

## Revisit $max$: Program Calculation without Folding Steps

$$max = hd \circ sort$$

We may calculate as follows.

$$\begin{array}{ll} & max \\ = & \quad \{ \text{ define max in terms of hom } \} \\ & hd \circ (\![ [], insert ]\!) \\ = & \quad \{ \text{ promotion: } \forall a, x.\ hd\ (insert\ a\ x) = a \otimes hd\ x \ \} \\ & (\![ hd\ [], \otimes ]\!) \end{array}$$

The $\otimes$ that satisfies

$$\forall a, x.\ hd\ (insert\ a\ x) = a \otimes hd\ x$$

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

$$\begin{array}{ll}
a \otimes b & = \quad \{ \text{ let } x \text{ be any list; by } \textbf{inversion} \ \} \\
 & a \otimes hd\ (b : x) \\
 & = \quad \{ \text{ the condition in the promotion rule } \} \\
 & hd\ (insert\ a\ (b : x)) \\
 & = \quad \{ \text{ definition of } insert \ \} \\
 & hd\ (\textbf{if}\ a \geq b\ \textbf{then}\ a : (b : x)\ \textbf{else}\ b : insert\ a\ x) \\
 & = \quad \{ \text{ if property } \} \\
 & \textbf{if}\ a \geq b\ \textbf{then}\ hd\ (a : (b : x))\ \textbf{else}\ hd\ (b : insert\ a\ x) \\
 & = \quad \{ \text{ definition of } hd \ \} \\
 & \textbf{if}\ a \geq b\ \textbf{then}\ a\ \textbf{else}\ b
\end{array}$$

## 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

```
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

$$
\begin{array}{lcl}
\mathit{sumBiggers} & = & \mathit{sum} \circ \mathit{biggers} \\
\mathit{biggers}\ [] & = & [] \\
\mathit{biggers}\ (a : x) & = & \mathbf{if}\ a \geq \mathit{sum}\ x\ \mathbf{then}\ a : \mathit{biggers}\ x\ \mathbf{else}\ \mathit{biggers}\ x \\
\mathit{sum}\ [] & = & [] \\
\mathit{sum}\ (a : x) & = & a + \mathit{sum}\ x
\end{array}
$$

## An Efficient Haskell Program after Loop Fusion

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

# Outline

# 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:

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

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.

$$f_1 = [\![(e_1, \ldots, e_n), (\oplus_1, \ldots, \oplus_n)]\!].$$

Note:

$$(\![e, \oplus]\!) = [\![(e), (\oplus)]\!]$$

## An Example

From

$$
\begin{array}{lcl}
biggers\ [] & = & [] \\
biggers\ (a : x) & = & \textbf{if}\ a \geq sum\ x\ \textbf{then}\ a : biggers\ x\ \textbf{else}\ biggers\ x \\
sum\ [] & = & [] \\
sum\ (a : x) & = & a + sum\ x
\end{array}
$$

we have

$$
\begin{array}{l}
biggers = [\![([], 0), (\oplus_1, \oplus_2)]\!] \\
\quad \textbf{where}\ a \oplus_1 (r, s) = \textbf{if}\ a \geq s\ \textbf{then}\ a : r\ \textbf{else}\ r \\
\qquad\qquad a \oplus_2 (r, s) = a + s
\end{array}
$$

# 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

$$[\![(e_1, e_2, \ldots, e_n), (\oplus_1, \oplus_2, \ldots, \oplus_n)]\!] = \mathit{fst} \circ (\![(e_1, e_2, \ldots, e_n), \oplus]\!)$$
$$\textbf{where } a \oplus r = (a \oplus_1 r, a \oplus_2 r, \ldots, a \oplus_n r)$$

Here, *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.

$$
\begin{array}{ll}
 & \mathit{biggers} \\
= & \quad \{ \text{ mutumorphism for } \mathit{biggers} \} \\
 & [\![ ([\,], 0), (\oplus_1, \oplus_2) ]\!] \\
= & \quad \{ \text{ flattening rule } \} \\
 & \mathit{fst} \circ (\![ ([\,], 0), \oplus ]\!) \\
 & \quad \textbf{where } a \oplus (r, s) = (\textbf{if } a \geq s \textbf{ then } a : r \textbf{ else } r, a + s)
\end{array}
$$

Inlining the homomorphism in the derived program gives the following readable recursive program, which consists of a single loop.

$$
\begin{array}{l}
\mathit{biggers}\ x = \mathbf{let}\ (r, s) = \mathit{hom}\ x\ \mathbf{in}\ r \\
\quad \mathbf{where}\ \mathit{hom}\ [] = ([], 0) \\
\qquad\qquad \mathit{hom}\ (a : x) = \mathbf{let}\ (r, s) = \mathit{hom}\ x \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\mathbf{if}\ a \geq s\ \mathbf{then}\ a : r\ \mathbf{else}\ r, a + s)
\end{array}
$$

## Tupling: dealing with adjacent independent loops

$$f((\![e_1, \oplus_1]\!)\ x,\ (\![e_2, \oplus_2]\!)\ x) = f((\![(e_1, e_2), \oplus]\!)\ x)$$
$$\textbf{where}\ a \oplus (r_1, r_2) = (a \oplus_1 r_1, a \oplus_2 r_2)$$

## An Example

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

$$average\ x = sum\ x / length\ x$$

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

$$\begin{aligned} average\ x = &\ \mathbf{let}\ (s,l) = tup\ x\ \mathbf{in}\ s/l \\ &\ \mathbf{where}\ tup = (\![(0,0), \lambda a\ (s,l).\ (a+s, 1+l)]\!) \end{aligned}$$

## Fusion: dealing with adjacent dependent loops

$$(\![e, \oplus]\!) \circ \mathit{build}\ g = g\ (e, \oplus)$$
$$\quad \textbf{where}\ \mathit{build}\ g = g\ ([\,], (:))$$

Here the build-form can be obtained by *promotion*:

$$(\![d, \otimes]\!) = \mathit{build}\ (\lambda(c, \odot).\ (\![c, \odot]\!) \circ (\![d, \otimes]\!))$$

## An Example

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

$$\begin{array}{l} biggers = fst \circ (\![([\,], 0), \oplus]\!) \\ \quad \textbf{where } a \oplus (r, s) = (\textbf{if } a \geq s \textbf{ then } a : r \textbf{ else } r, a + s) \end{array}$$

We can obtain the following build form:

$$\begin{array}{l} biggers = build\ (\lambda(c, \odot).\ fst \circ (\![(c, 0), \oplus']\!)) \\ \quad \textbf{where } a \oplus' (r, s) = (\textbf{if } a \geq s \textbf{ then } a \odot r \textbf{ else } r, a + s) \end{array}$$

Now applying the shortcut fusion rule to

$$sumBiggers = (\![0, +]\!) \circ bigger$$

soon yields the following single-loop program for *sumBiggers*:

$$\begin{array}{l} sumBiggers = fst \circ (\![(0,0), \otimes]\!) \\ \quad \textbf{where } a \otimes (r, s) = (\textbf{if } a \geq s \textbf{ then } a + r \textbf{ else } r, a + s) \end{array}$$

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:

$$
\begin{array}{lll}
sumBiggers & = & sum \circ biggers \\
& = & \quad \{ \text{ represent list functions by mutumorphism/homomorphism } \} \\
& & (\![0, +]\!) \circ [\![([], 0), (\oplus_1, \oplus_2)]\!] \\
& & \quad \textbf{where } a \oplus_1 (r, s) = \textbf{if } a \geq s \textbf{ then } a : r \textbf{ else } r \\
& & \qquad\qquad a \oplus_2 (r, s) = a + s \\
& = & \quad \{ \text{ flatten: } a \otimes (r, s) = (\textbf{if } a \geq s \textbf{ then } a + r \textbf{ else } r, a + s) \ \} \\
& & (\![0, +]\!) \circ \mathit{fst} \circ (\![(0, 0), \otimes]\!) \\
& = & \quad \{ \text{ make ``build'' form } \} \\
& & (\![0, +]\!) \circ \mathit{build}\ (\lambda(c, \odot).\ \mathit{fst} \circ (\![(c, 0), \oplus']\!)) \\
& & \quad \textbf{where } a \oplus' (r, s) = (\textbf{if } a \geq s \textbf{ then } a \odot r \textbf{ else } r, a + s) \\
& = & \quad \{ \text{ fusion } \} \\
& & \mathit{fst} \circ (\![(0, 0), \otimes]\!) \\
& & \quad \textbf{where } a \otimes (r, s) = (\textbf{if } a \geq s \textbf{ then } a + r \textbf{ else } r, a + s)
\end{array}
$$

# 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.

| A Sequential Program | $\Longrightarrow$ | 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 | | A Parallel Program |
|---|---|---|
| $f :: [a] \to R$ | $\Longrightarrow$ | $f :: [a] \to R$ |

## 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 | | A Parallel Program |
|---|---|---|
| $f :: \mathit{SeqList}\ a \rightarrow R$ | $\Rightarrow$ | $f' :: \mathit{ParaList}\ a \rightarrow R$ |

## Data Refinement

A *sequential* view of lists:

$$ConsList\ a \;=\; [\,] \mid a : ConsList\ a$$

A *parallel* view of lists:

$$JoinList\ a \;=\; [\,] \mid [.]\ a \mid JoinList\ a +\!\!+ JoinList\ a$$

**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*.

$$\begin{aligned} sumS\ [] &= 0 \\ sumS\ (a : x) &= a + sumS\ x \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} sumP\ [] &= 0 \\ sumP\ [a] &= a \\ sumP\ (x +\!\!+\ y) &= sumP\ x + sumP\ y \end{aligned}$$

## Running Example: the Maximum Segment Sum Problem

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

$$mss\ [3, -4, \underline{2, -1, 6}, -3] = 7$$

*A Sequential Program:*

$$\begin{array}{lcl} mss\ [] & = & 0 \\ mss\ (a : x) & = & a \uparrow (a + mis\ x) \uparrow mss\ x \\ mis\ [] & = & 0 \\ mis\ (a : x) & = & a \uparrow (a + mis\ x) \end{array}$$

# 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

*J-homomorphisms* (*Homomorphisms on JoinList*) are functions defined in the following form:

$$h\,(x +\!\!+\, y) \quad = \quad h\,x \underline{\oplus}\, h\,y$$

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

$$f\ \overline{x_i}_1^n\ (f\ \overline{y_i}_1^n\ r) = f\ \overline{(g_i\ \overline{x_i}_1^n\ \overline{y_i}_1^n)}_1^n\ r$$

## Example: a composition-closed function

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

because

$$\begin{array}{ll} & f\ x_1\ x_2\ (f\ y_1\ y_2\ r) \\ = & \quad \{\ \text{definition of } f\ \} \\ & x_1 \uparrow (x_2 + (y_1 \uparrow (y_2 + r))) \\ = & \quad \{\ \ \text{since } a + (b \uparrow c) = (a + b) \uparrow (a + c)\ \} \\ & x_1 \uparrow ((x_2 + y_1) \uparrow (x_2 + (y_2 + r))) \\ = & \quad \{\ \text{associativity of } + \text{ and } \uparrow\ \} \\ & (x_1 \uparrow (x_2 + y_1)) \uparrow ((x_2 + y_2) + r) \\ = & \quad \{\ \text{define } g_1\ x_1\ x_2\ y_1\ y_2 = (x_1 \uparrow (x_2 + y_1)),\ \ g_2\ x_1\ x_2\ y_1\ y_2 = x_2 + y_2\ \} \\ & (g_1\ x_1\ x_2\ y_1\ y_2) \uparrow (g_2\ x_1\ x_2\ y_1\ y_2 + r) \\ = & \quad \{\ \text{definition of } f\ \} \\ & f\ (g_1\ x_1\ x_2\ y_1\ y_2)\ (g_2\ x_1\ x_2\ y_1\ y_2)\ r \end{array}$$

## 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

$$a \oplus r = f\ \overline{e_i}_1^n\ r$$

then

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

$$\begin{array}{lcl} h\ [a] & = & (e_1, e_2, \ldots, e_n) \\ h(x +\!\!+\, y) & = & h\ x \otimes h\ y \\ & & \textbf{where}\ \overline{x_i}_1^n \otimes \overline{y_i}_1^n = \overline{g_i\ \overline{x_1}_1^n\ \overline{y_i}_1^n}_1^n) \end{array}$$

## Example: parallelization of $mis$

The initial program:

$$\begin{array}{lcl} mis\ [] & = & 0 \\ mis\ (a : x) & = & a \uparrow (a + mis\ x) \end{array}$$

which is in fact a homomorphism:

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

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:

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

where

$$\begin{array}{lcl} h\ [a] & = & (a, a) \\ h\ (x +\!\!+\ y) & = & h\ x \otimes h\ y \\ & & \mathbf{where}\ (x_1, x_2) \otimes (y_1, y_2) = (x_1 \uparrow (x_2 + y_1), x_2 + y_2). \end{array}$$

## 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 *loop fusion calculation* to the program to obtain a compact program defined in terms of *homomorphisms*.
2. Derive *composition-closed functions* from homomorphisms [APLAS04].
3. Apply the *parallelizing rule* to map homomorphisms to J-homomorphisms.

## Example: parallelizing $mss$

$$
\begin{array}{lcl}
mss\ [] & = & 0 \\
mss\ (a:x) & = & a \uparrow (a + mis\ x) \uparrow mss\ x \\
mis\ [] & = & 0 \\
mis\ (a:x) & = & a \uparrow (a + mis\ x)
\end{array}
$$

**Step 1: Loop fusion calculation**

$$mss = fst \circ mss\_mis$$

where $mss\_mis$ is the homomorphism defined below:

$$\begin{array}{l} mss\_mis = (\![ (0,0), \oplus ]\!) \\ \quad \textbf{where } a \oplus (s,i) = (a \uparrow (a+i) \uparrow s, a \uparrow (a+i)). \end{array}$$

**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 $g_1, g_2, g_3, g_4, g_5$:

$$\begin{array}{lcl}
g_1\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = & x_1 \uparrow (x_2 + y_4) \uparrow (x_3 + y_1) \\
g_2\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = & (x_2 + y_5) \uparrow (x_3 + y_2) \\
g_3\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = & x_3 + y_3 \\
g_4\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = & x_4 \uparrow (x_5 + y_4) \\
g_5\ x_1\ x_2\ x_3\ x_4\ x_5\ y_1\ y_2\ y_3\ y_4\ y_5 & = & x_5 + y_5
\end{array}$$

## Step 3: Application of the parallelization rule

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

where $h$ is a J-homomorphism defined as follows.

$$\begin{aligned}
&h\ [a] = (a, a, 0, a, a)\\
&h(x +\!\!+\, y) = h\ x \otimes h\ y\\
&\quad \mathbf{where}\ (x_1, x_2.x_3.x_4.x_5) \otimes (y_1, y_2, y_3, y_4, y_5)\\
&\qquad\qquad = (x_1 \uparrow (x_2 + y_4) \uparrow (x_3 + y_1),\\
&\qquad\qquad\quad (x_2 + y_5) \uparrow (x_3 + y_2),\\
&\qquad\qquad\quad x_3 + y_3,\\
&\qquad\qquad\quad x_4 \uparrow (x_5 + y_4),\\
&\qquad\qquad\quad x_5 + y_5)
\end{aligned}$$

# 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

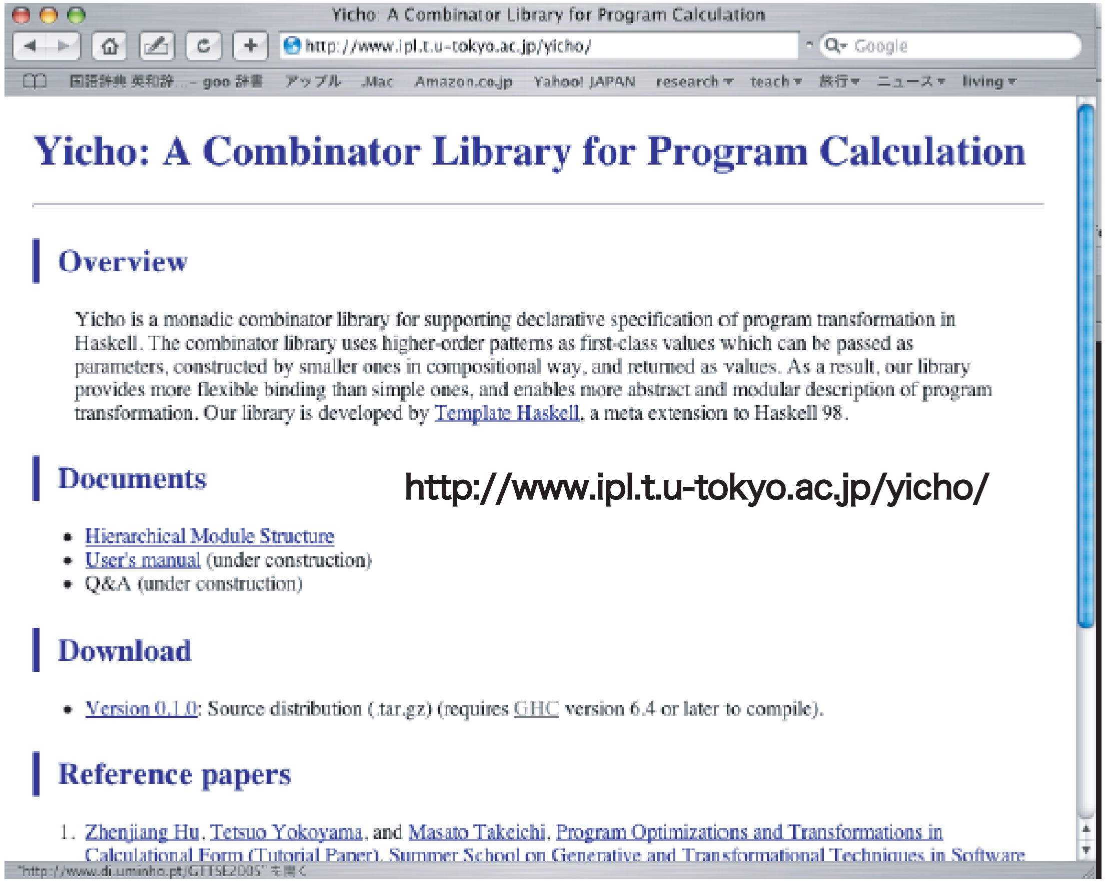
**direct and efficient implementation of calculation rules in Haskell**

with

deterministic higher-order patterns .

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

# Yicho Website:



# Program Representation in Template Haskell

## Quote and Unquote

```
sum :: [Int] -> Int
[| sum |] :: Q Exp
$ ([| sum |]) :: [Int] -> Int
```

## Representation of Function Definitions

```
def =
  [d|
      max = hd . sort

      sort [] = []
      sort (a:x) = insert a (sort x)

      insert a [] = b
      insert a (b:x) = if a >= b then a : (b : x)
                                  else b : insert a x
  |]
```

# 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.

```
ret  :: Q Exp → Y (Q Exp)
runY :: Y (Q Exp) → Q Exp
```

Note: `ExpQ` = `Q Exp`, `ExpY` = `Y ExpQ`.

## Useful Combinators for Coding Calculation

| | | |
|---|---|---|
| Match | `(<==)` | `:: ExpQ -> ExpQ -> Y ()` |
| Rule | `(==>)` | `:: ExpQ -> ExpQ -> RuleY` |
| Sequence | `(>>)` | `:: Y () -> Y () -> Y ()` |
| Choice | `(<+)` | `:: ExpY -> ExpY -> ExpY` |
| Case | `casem` | `:: ExpQ -> [RuleY] -> ExpY` |

# Match

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

**An Example**

```
[| \a x -> $oplus a (biggers x, sum x) |]
    <== [| \a x -> if a >= sum x then a : biggers x
                   else biggers x
        |]
```

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

```
{ $oplus := \x (b,s) ->
            if x >= s then x : b else b }.
```

## Rule

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

### An Example

```
[| hom $e $oplus . build $g |] ==> [| g $e $oplus |]
```

Note: Rule can be defined by Match.

```
(==>) :: ExpQ -> ExpQ -> RuleY
(lhs ==> rhs) term = do lhs <== term
                        ret rhs
```

## Choise & Casem

Used to express deterministic choice.

```
(rule1 e) <+ (rule2 e)
```

```
casem :: ExpQ -> [RuleY] -> ExpY
casem sel (r:rs) = r sel <+ casem sel rs
```

# Code Calculation Rules in Yicho

## Code the promotion rule

promotion: $$\frac{f(a \oplus x) = a \otimes f\ x}{f \circ \mathit{foldr}\ (\oplus)\ e = \mathit{foldr}\ (\otimes)\ (f\ e)}$$

$$\Downarrow$$

```
promotion :: ExpQ -> ExpY
promotion exp = do
    [f,oplus,e,otimes] <- pvars ["f","oplus","e","otimes"]
    [| $f . foldr $oplus $e |] <== exp
    [| \a x -> $otimes a ($f x) |]
        <== [| \a x -> $f ($oplus a x) |]
    ret [| foldr $otimes ($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.

```
> 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!