Mathematical Structures of Programs
– Specification and Implementation –

Zhenjiang Hu

National Institue of Informatics

May - June, 2011

Outline

Specification and Implementation

- A specification
  - describes what task an algorithm is to perform,
  - expresses the programmers' intent,
  - should be as clear as possible.

Specification and Implementation

- A specification
  - describes what task an algorithm is to perform,
  - expresses the programmers' intent,
  - should be as clear as possible.

- An implementation
  - describes how task is to perform,
  - expresses an algorithm (an execution),
  - should be efficiently done within the time and space available.

Specification and Implementation

- A specification
  - describes what task an algorithm is to perform,
  - expresses the programmers' intent,
  - should be as clear as possible.

- An implementation
  - describes how task is to perform,
  - expresses an algorithm (an execution),
  - should be efficiently done within the time and space available.

The link is that the implementation should be proved to satisfy the specification.

How to write a specification?

- By predicates: describing intended relationship between input and output of an algorithm.

How to write a specification?

- By predicates: describing intended relationship between input and output of an algorithm.
- By functions: describing straightforward functional mapping from input to output of an algorithm, which is executable but could be terribly inefficient.

Specifying Algorithms by Predicates (1/3)

Specification: describing intended relationship between input and output of an algorithm.

Specifying Algorithms by Predicates (1/3)

Specification: describing intended relationship between input and output of an algorithm.

Example: *increase*

The specification

$$increase :: Int \rightarrow Int$$
$$increase\ x > square\ x$$

says that the result of *increase* should be strictly greater than the square of its input, where $square\ x = x * x$.

Specifying Algorithms by Predicates (2/3)

In this case, an Implementation is first given and then proved to satisfy the specification.

Specifying Algorithms by Predicates (2/3)

In this case, an Implementation is first given and then proved to satisfy the specification.

Example: *increase* (continue)

One implementation is

$$increase\ x = square\ x + 1$$

which can be proved by the following simple calculation.

$$\begin{array}{ll} & increase\ x \\ = & \quad \{ \text{ definition of } increase \ \} \\ & square\ x + 1 \\ > & \quad \{ \text{ arithmetic property } \} \\ & square\ x \end{array}$$

Specifying Algorithms by Predicates (3/3)

Exercise S1

Give another implementation of *increase* and prove that your implementation meets its specification.

Specifying Algorithms by Functions (1/3)

Specification: describing straightforward functional mapping from input to output of an algorithm, which is executable but could be terribly inefficient.

Specifying Algorithms by Functions (1/3)

Specification: describing straightforward functional mapping from input to output of an algorithm, which is executable but could be terribly inefficient.

Example: *quad*

The specification for computing quadruple of a number can be described straightforwardly by

$$quad\ x = x * x * x * x$$

which is not efficient in the sense that multiplications are used three times.

Specifying Algorithms by Functions (2/3)

With functional specification, we do not need to invent the implementation; just to improve specification via calculation.

Specifying Algorithms by Functions (2/3)

With functional specification, we do not need to invent the implementation; just to improve specification via calculation.

Example: *quad* (continue)

We derive (develop) an efficient algorithm with only two multiplications by the following calcualtion.

$$
\begin{array}{ll}
 & quad\ x \\
= & \quad \{ \text{ specification } \} \\
 & x * x * x * x \\
= & \quad \{ \text{ since } x \text{ is associative } \} \\
 & (x * x) * (x * x) \\
= & \quad \{ \text{ definition of } square \ \} \\
 & square\ x * square\ x \\
= & \quad \{ \text{ definition of } square \ \} \\
 & square\ (aquare\ x)
\end{array}
$$

Specifying Algorithms by Functions (3/3)

Exercise S2

Extend the idea in the derivation of efficient *quad* to develop an efficient algorithm for computing *exp* defined by

$$exp(x, n) = x^n.$$

Advantages of Functional Specification

- Functional specification is executable.

Advantages of Functional Specification

- Functional specification is executable.
- Functional specification is powerful to express intended mappings directly by functions or through their composition.

Advantages of Functional Specification

- Functional specification is executable.
- Functional specification is powerful to express intended mappings directly by functions or through their composition.
- Functional specification is suitable for reasoning, when functions used are well-structured with good algebraic properties.

Advantages of Functional Specification

- Functional specification is executable.
- Functional specification is powerful to express intended mappings directly by functions or through their composition.
- Functional specification is suitable for reasoning, when functions used are well-structured with good algebraic properties.

In this course, we will consider functional specification.

Mathematical Structures in Programs – Homomorphism –

Zhenjiang Hu

National Institute of Informatics

May 30, 2010

Outline

Longest Even Segment Problem

Given is a sequence x and a predicate p. Required is an efficient algorithm for computing some longest segment of x, all of whose elements satisfy p.

$$\mathit{lsp}\ \mathit{even}\ [3,1,4,1,5,9,2,6,5] = [2,6]$$

Homomorphisms

A homomorphism from a monoid $(\alpha, \oplus, id_\oplus)$ to a monoid $(\beta, \otimes, id_\otimes)$ is a function h satisfying the two equations:

$$
\begin{array}{lcl}
h\ id_\oplus & = & id_\otimes \\
h\ (x \oplus y) & = & h\ x \otimes h\ y
\end{array}
$$

Lemma (Promotion)

h is a homomorphism if and only if the following holds.

$$h \cdot \oplus/ = \otimes/ \cdot h*$$

Lemma (Promotion)

h is a homomorphism if and only if the following holds.

$$h \cdot \oplus/ = \otimes/ \cdot h*$$

Proof Sketch.

- $\Leftarrow$: *simple.*
- $\Rightarrow$: *by induction.*

Lemma (Promotion)

h is a homomorphism if and only if the following holds.

$$h \cdot \oplus/ = \otimes/ \cdot h*$$

Proof Sketch.

- $\Leftarrow$: *simple.*
- $\Rightarrow$: *by induction.*

So we have

$$\begin{array}{rcl} f * \cdot +\!\!+ / & = & +\!\!+ / \cdot f ** \\ \oplus/ \cdot +\!\!+ / & = & \oplus/ \cdot (\oplus/)* \end{array}$$

Characterization of Homomorphisms

Lemma

h is a homomorphism from the list monoid if and only if there exists f and $\oplus$ such that

$$h = \oplus/ \cdot f*$$

Proof

$\Rightarrow$:

$$
\begin{array}{ll}
 & h \\
= & \quad \{ \text{ definition of } id \text{ } \} \\
 & h \cdot id \\
= & \quad \{ \text{ identity lemma (can you prove it?) } \} \\
 & h \cdot +\!\!\!+ / \cdot [\cdot]* \\
= & \quad \{ \text{ } h \text{ is a homomorphism } \} \\
 & \oplus / \cdot h * \cdot [\cdot]* \\
= & \quad \{ \text{ map distributivity } \} \\
 & \oplus / \cdot (h \cdot [\cdot])* \\
= & \quad \{ \text{ definition of } h \text{ on singleton } \} \\
 & \oplus / \cdot f*
\end{array}
$$

Proof (Cont.)

$\Leftarrow$: We reason that $h = \oplus/ \cdot f*$ is a homomorphism by calculating

$$
\begin{array}{cl}
 & h \cdot +\!\!+\,/ \\
= & \quad \{ \text{ given form for } h \ \} \\
 & \oplus/ \cdot f * \cdot +\!\!+\,/ \\
= & \quad \{ \text{ map and reduce promotion } \} \\
 & \oplus/ \cdot (\oplus/ \cdot f*)* \\
= & \quad \{ \text{ hypothesis } \} \\
 & \oplus/ \cdot h*
\end{array}
$$

Examples of Homomorphisms

- #: compute the length of a list.

$$\# = +/ \cdot K_1 *$$

Examples of Homomorphisms

- $\#$: compute the length of a list.

$$\# = +/ \cdot K_1 *$$

- *reverse*: reverses the order of the elements in a list.

$$reverse = \tilde{+\!\!+} / \cdot [\cdot]*$$

Here, $x \tilde{\oplus} y = y \oplus x$.

- *sort*: reorders the elements of a list into ascending order.

$$sort = \wedge\!\!\!\wedge\!\!\!\wedge\, / \cdot [\cdot]*$$

Here, $\wedge\!\!\!\wedge\!\!\!\wedge$ (pronounced merge) is defined by the equations:

$$\begin{array}{lcll} x \wedge\!\!\!\wedge\!\!\!\wedge [\,] & = & x & \\ [\,] \wedge\!\!\!\wedge\!\!\!\wedge y & = & y & \\ ([a] +\!\!\!+ x) \wedge\!\!\!\wedge\!\!\!\wedge ([b] +\!\!\!+ y) & = & [a] +\!\!\!+ (x \wedge\!\!\!\wedge\!\!\!\wedge ([b] +\!\!\!+ y)), & \text{if } a \leq b \\ & = & [b] +\!\!\!+ (([a] +\!\!\!+ x) \wedge\!\!\!\wedge\!\!\!\wedge y), & \text{otherwise} \end{array}$$

- *all p*: returns True if every element of the input list satisfies the predicate p.

$$all\ p = \wedge/ \cdot p*$$

- *all p*: returns True if every element of the input list satisfies the predicate p.

$$all\ p = \wedge/ \cdot p*$$

- *some p*: returns True if at least one element of the input list satisfies the predicate p.

$$some\ p = \vee/ \cdot p*$$

- *split*: splits a non-empty list into its last element and the remainder.

$$
\begin{array}{lcl}
\mathit{split} & : & [\alpha]^{+} \to ([\alpha], \alpha) \\
\mathit{split}\ [a] & = & ([\,], a) \\
\mathit{split}\ (x +\!\!+\ y) & = & \mathit{split}\ x \oplus \mathit{split}\ y \\
 & & \text{where } (x, a) \oplus (y, b) = (x +\!\!+ [a] +\!\!+ y, b)
\end{array}
$$

- *split*: splits a non-empty list into its last element and the remainder.

$$
\begin{array}{lcl}
\mathit{split} & : & [\alpha]^+ \rightarrow ([\alpha], \alpha) \\
\mathit{split}\ [a] & = & ([\,], a) \\
\mathit{split}\ (x \mathbin{+\!\!+} y) & = & \mathit{split}\ x \oplus \mathit{split}\ y \\
 & & \text{where } (x, a) \oplus (y, b) = (x \mathbin{+\!\!+} [a] \mathbin{+\!\!+} y, b)
\end{array}
$$

Note: *split* is a homomorphism on the semigroup $([\alpha]^+, +\!\!+)$.

- *split*: splits a non-empty list into its last element and the remainder.

$$
\begin{array}{lcl}
split & : & [\alpha]^+ \rightarrow ([\alpha], \alpha) \\
split\ [a] & = & ([\,], a) \\
split\ (x +\!\!+\, y) & = & split\ x \oplus split\ y \\
 & & \text{where } (x, a) \oplus (y, b) = (x +\!\!+ [a] +\!\!+\, y, b)
\end{array}
$$

Note: *split* is a homomorphism on the semigroup $([\alpha]^+, +\!\!+)$.

Exercise: Let $init = \pi_1 \cdot split$, where $\pi_1\ (a, b) = a$. Show that *init* is not a homomorphism.

All applied to

The operator ^{o} (pronounced all applied to) takes a sequence of functions and a value and returns the result of applying each function to the value.

$$[f, g, \ldots, h]^o a = [f\ a, g\ a, \ldots, h\ a]$$

Formally, we have

$$\begin{array}{lcl} [\,]^o\ a & = & [\,] \\ [f]^o\ a & = & [f\ a] \\ (fs +\!\!+\ gs)^o\ a & = & (fs^o\ a) +\!\!+ (gs^o\ a) \end{array}$$

so, $(^o\ a)$ is a homomorphism.

Exercise: Show that $[\cdot] = [id]^o$.

Conditional Expressions

The conditional notation

$$\begin{array}{lcll} h\ x & = & f\ x, & \text{if } p\ x \\ & = & g\ x, & \text{otherwise} \end{array}$$

will be written by the McCarthy conditional form:

$$h = (p \rightarrow f, g)$$

Conditional Expressions

The conditional notation

$$\begin{array}{lcll} h\ x & = & f\ x, & \text{if } p\ x \\ & = & g\ x, & \text{otherwise} \end{array}$$

will be written by the McCarthy conditional form:

$$h = (p \rightarrow f, g)$$

Laws on Conditional Forms

$$\begin{array}{lcl} h \cdot (p \rightarrow f, g) & = & (p \rightarrow h \cdot f, h \cdot g) \\ (p \rightarrow f, g) \cdot h & = & (p \cdot h \rightarrow f \cdot h, g \cdot h) \\ (p \rightarrow f, f) & = & f \end{array}$$

(Note: all functions are assumed to be total.)

Filter

The operator $\triangleleft$ (pronounced filter) takes a predicate p and a list x and returns the sublist of x consisting, in order, of all those elements of x that satisfy p.

$$p\triangleleft = +\!\!+ \;/ \cdot (p \rightarrow [id]^o, [\,]^o)*$$

Filter

The operator $\triangleleft$ (pronounced filter) takes a predicate p and a list x and returns the sublist of x consisting, in order, of all those elements of x that satisfy p.

$$p\triangleleft = +\!\!+ / \cdot (p \rightarrow [id]^o, []^o)*$$

Exercise: Prove that the filter satisfies the filter promotion property:

$$(p\triangleleft) \cdot +\!\!+ / = +\!\!+ / \cdot (p\triangleleft)*$$

Filter

The operator $\triangleleft$ (pronounced filter) takes a predicate p and a list x and returns the sublist of x consisting, in order, of all those elements of x that satisfy p.

$$p\triangleleft = +\!\!+ \ / \cdot (p \to [id]^o, []^o)*$$

Exercise: Prove that the filter satisfies the filter promotion property:

$$(p\triangleleft) \cdot +\!\!+ \ / = +\!\!+ \ / \cdot (p\triangleleft)*$$

Exercise: Prove that the filter satisfies the map-filter swap property:

$$(p\triangleleft) \cdot f* = f * \cdot (p \cdot f)\triangleleft$$

Cross-product

$X_\oplus$ is a binary operator that takes two lists x and y and returns a list of values of the form $a \oplus b$ for all a in x and b in y.

$$[a, b] X_\oplus [c, d, e] = [a \oplus c, b \oplus c, a \oplus d, b \oplus d, a \oplus e, b \oplus e]$$

Formally, we define $X_\oplus$ by three equations:

$$\begin{array}{lcl} x X_\oplus [\,] & = & [\,] \\ x X_\oplus [a] & = & (\oplus a) * x \\ x X_\oplus (y +\!\!\!+ z) & = & (x X_\oplus y) +\!\!\!+ (x X_\oplus z) \end{array}$$

Thus $x X_\oplus$is a homomorphism.

Properties

$[\,]$ is the zero element of $X_{\oplus}$:

$$[\,]X_{\oplus}x \;=\; xX_{\oplus}[\,] \;=\; [\,]$$

We have cross promotion rules:

$$\begin{array}{lcl} f** \cdot X_{++}/ & = & X_{++}/ \cdot f*** \\ \oplus/ \cdot X_{++}/ & = & X_{\oplus}/ \cdot (X_{\oplus}/)* \end{array}$$

Example Uses of Cross-product

- *cp*: takes a list of lists and returns a list of lists of elements, one from each component.

$$cp\ [[a,b],[c],[d,e]] = [[a,c,d],[b,c,d],[a,c,e],[b,c,e]]$$

$$cp = X_{++} \,/ \cdot ([id]^{o} *)*$$

- *subs*: computes all subsequences of a list.

$$subs\ [a, b, c] = [[], [a], [b], [a, b], [c], [a, c], [b, c], [a, b, c]]$$

$$subs = X_{+\!\!+} / \cdot [[\,]^o, [id]^o]^o *$$

- *subs*: computes all subsequences of a list.

$$subs\ [a, b, c] = [[], [a], [b], [a, b], [c], [a, c], [b, c], [a, b, c]]$$

$$subs = X_{++} / \cdot [[\,]^o, [id]^o]^o *$$

Note: $subs = cp \cdot [[]^o, [id]^o]*$.

- $(all\ \ p)\triangleleft$:

$$(all\ \ even) \triangleleft [[1,3],[2,4],[1,2,3]] = [[2,4]]$$

$$(all\ \ p)\triangleleft = +\!\!+\ / \cdot (X_{+\!\!+} / \cdot (p \rightarrow [[id]^{o}]^{o}, [\,]^{o})*)*$$

Selection Operators

Suppose f is a numeric valued function. We want to define an operator $\uparrow_f$ such that

1. $\uparrow_f$ is associative, commutative and idempotent;
2. $\uparrow_f$ is selective in that

$$x \uparrow_f y = x \quad \text{or} \quad x \uparrow_f y = y$$

3. $\uparrow_f$ is maximizing in that

$$f(x \uparrow_f y) = f\ x \uparrow f\ y$$

Selection Operators

Suppose f is a numeric valued function. We want to define an operator $\uparrow_f$ such that

1. $\uparrow_f$ is associative, commutative and idempotent;
2. $\uparrow_f$ is selective in that

$$x \uparrow_f y = x \quad \text{or} \quad x \uparrow_f y = y$$

3. $\uparrow_f$ is maximizing in that

$$f(x \uparrow_f y) = f\ x \uparrow f\ y$$

Condition: f should be injective.

An Example: $\uparrow_{\#}$

But if f is not injective, then $x \uparrow_f y$ is not specified when $x \neq y$ but $f\ x = f\ y$.

$$[1,2] \uparrow_{\#} [3,4]$$

An Example: $\uparrow_{\#}$

But if f is not injective, then $x \uparrow_f y$ is not specified when $x \neq y$ but $f\ x = f\ y$.

$$[1, 2] \uparrow_{\#} [3, 4]$$

To solve this problem, we may *refine* f to an injective function f' such that

$$f\ x < f\ y \Rightarrow f'\ x < f'\ y.$$

An Example: $\uparrow_\#$

But if f is not injective, then $x \uparrow_f y$ is not specified when $x \neq y$ but $f\ x = f\ y$.

$$[1,2] \uparrow_\# [3,4]$$

To solve this problem, we may *refine* f to an injective function f' such that

$$f\ x < f\ y \Rightarrow f'\ x < f'\ y.$$

So we may select the *lexicographically* least sequence as the value of $x \uparrow_\# y$ when $\#x = \#y$.

In this case, $+\!\!+$ distributes through $\uparrow_\#$:

$$
\begin{array}{rcl}
x +\!\!+ (y \uparrow_\# z) & = & (x +\!\!+ y) \uparrow_\# (x +\!\!+ z) \\
(x \uparrow_\# y) +\!\!+ z & = & (x +\!\!+ z) \uparrow_\# (y +\!\!+ z)
\end{array}
$$

That is,

$$
\begin{array}{rcl}
(x +\!\!+) \cdot \uparrow_\# / & = & \uparrow_\# / \cdot (x +\!\!+) * \\
(+\!\!+ \ x) \cdot \uparrow_\# / & = & \uparrow_\# / \cdot (+\!\!+ \ x) * .
\end{array}
$$

We assume $\omega = \uparrow_\# /[\,]$, satisfying $\#\omega = -\infty$. (ω is the zero element of $+\!\!+$)

A short calculation

$$\begin{array}{ll}
& \uparrow_{\#} / \cdot (all\ \ p)\triangleleft \\
= & \quad \{ \text{ definition before } \} \\
& \uparrow_{\#} / \cdot +\!\!+ / \cdot (X_{+\!\!+} / \cdot (p \rightarrow [[id]^{o}]^{o}, [\,]^{o})*)* \\
= & \quad \{ \text{ reduce promotion } \} \\
& \uparrow_{\#} / \cdot (\uparrow_{\#} / \cdot X_{+\!\!+} / \cdot (p \rightarrow [[id]^{o}]^{o}, [\,]^{o})*)* \\
= & \quad \{ +\!\!+ \text{ distributes over } \uparrow_{\#} \} \\
& \uparrow_{\#} / \cdot (+\!\!+ / \cdot (\uparrow_{\#} /) * \cdot (p \rightarrow [[id]^{o}]^{o}, [\,]^{o})*)* \\
= & \quad \{ \text{ many steps ... } \} \\
& \uparrow_{\#} / \cdot (+\!\!+ / \cdot (p \rightarrow [id]^{o}, K_{\omega})*)*
\end{array}$$

Existence of Homomorphism

Existence Lemma

The list function h is a homomorphism iff the implication

$$h\ v = h\ x\ \wedge\ h\ w = h\ y\ \Rightarrow\ h\ (v +\!\!\!+\ w) = h\ (x +\!\!\!+\ y)$$

holds for all lists v, w, x, y.

Existence of Homomorphism

Existence Lemma

The list function h is a homomorphism iff the implication

$$h\ v = h\ x\ \wedge\ h\ w = h\ y\ \Rightarrow\ h\ (v +\!\!+\, w) = h\ (x +\!\!+\, y)$$

holds for all lists v, w, x, y.

Proof Sketch.

- $\Rightarrow$: obvious by assuming $h = \odot/ \cdot f*$.

Existence of Homomorphism

Existence Lemma

The list function h is a homomorphism iff the implication

$$h\ v = h\ x\ \wedge\ h\ w = h\ y\ \Rightarrow\ h\ (v \mathbin{+\!\!+} w) = h\ (x \mathbin{+\!\!+} y)$$

holds for all lists v, w, x, y.

Proof Sketch.

- $\Rightarrow$: obvious by assuming $h = \odot/ \cdot f*$.
- $\Leftarrow$: Define $\odot$ by $t \odot u = h\ (g\ \ t \mathbin{+\!\!+} g\ \ u)$.
  for some g such that $h = h \cdot g \cdot h$ (such a g exisits!). Thus

$$h\ (x \mathbin{+\!\!+} y)\ \ =\ \ h\ x \odot h\ y.$$

Reference

Lemma

For every computatble total function h with enumerable domain, there is a computatble (but possibly partial) function g such that $h \cdot g \cdot h = h$.

Reference

Lemma

For every computatble total function h with enumerable domain, there is a computatble (but possibly partial) function g such that $h \cdot g \cdot h = h$.

Proof. Here is one suitable definition of g.

$$g\ t = head\ [x \mid h\ x = t]$$

If t is in the range of h then this process terminates.

Specification of the Problem

Recall the problem of computing the longest segment of a list, all of whose elements satisfied some given property p.

$$lsp = \uparrow_{\#} / \cdot (all\ \ p) \triangleleft \cdot segs$$

Specification of the Problem

Recall the problem of computing the longest segment of a list, all of whose elements satisfied some given property p.

$$lsp = \uparrow_{\#} / \cdot (all\ \ p) \triangleleft \cdot segs$$

Property: *lsp* is not a homomorphism.

Specification of the Problem

Recall the problem of computing the longest segment of a list, all of whose elements satisfied some given property p.

$$lsp = \uparrow_{\#} / \cdot (all\ \ p) \triangleleft \cdot segs$$

Property: *lsp* is not a homomorphism.

This is because:

$$\begin{array}{lll} lsp\ [2,1] & = lsp\ [2] & = [2] \\ lsp\ [4] & = lsp[4] & = [4] \end{array}$$

does not imply

$$lsp\ ([2,1] +\!\!+ [4]) = lsp\ ([2] +\!\!+ [4]).$$

Calculating a Solution to the Problem

$$
\begin{array}{ll}
 & \uparrow_{\#} / \cdot (\mathit{all}\ p) \triangleleft \cdot \mathit{segs} \\
= & \quad \{ \text{ segment decomposition } \} \\
 & \uparrow_{\#} / \cdot (\uparrow_{\#} / \cdot (\mathit{all}\ p) \triangleleft \cdot \mathit{tails}) * \cdot \mathit{inits} \\
= & \quad \{ \text{ result before } \} \\
 & \uparrow_{\#} / \cdot (\uparrow_{\#} / \cdot (+\!\!+ / \cdot (p \rightarrow [\mathit{id}]^{o}, K_{\omega}) *) * \cdot \mathit{tails}) * \cdot \mathit{inits} \\
= & \quad \{ \text{ Horner's rule with } x \odot a = (x +\!\!+ (p\ a \rightarrow [a], \omega) \uparrow_{\#} [\,] \ \} \\
 & \uparrow_{\#} \cdot \odot \not\rightarrow_{[\,]} * \cdot \mathit{inits} \\
= & \quad \{ \text{ accumulation lemma } \} \\
 & \uparrow_{\#} \cdot \odot \not\!\!\!\#\!\!\rightarrow_{[\,]}
\end{array}
$$

Calculating a Solution to the Problem

$$
\begin{array}{ll}
 & \uparrow_{\#} / \cdot (all\ \ p) \triangleleft \cdot segs \\
= & \quad \{ \text{ segment decomposition } \} \\
 & \uparrow_{\#} / \cdot (\uparrow_{\#} / \cdot (all\ \ p) \triangleleft \cdot tails) * \cdot inits \\
= & \quad \{ \text{ result before } \} \\
 & \uparrow_{\#} / \cdot (\uparrow_{\#} / \cdot (+\!\!+ / \cdot (p \rightarrow [id]^o, K_\omega)*) * \cdot tails) * \cdot inits \\
= & \quad \{ \text{ Horner's rule with } x \odot a = (x +\!\!+ (p\ a \rightarrow [a], \omega) \uparrow_{\#} [\,]\ \} \\
 & \uparrow_{\#} \cdot \odot \not\rightarrow_{[\,]} * \cdot inits \\
= & \quad \{ \text{ accumulation lemma } \} \\
 & \uparrow_{\#} \cdot \odot \not\!\!\!\!/\!\!\rightarrow_{[\,]}
\end{array}
$$

Exercise: Show the final program is linear in the number of calculation of p.

Outline

1. Specification and Implementation
2. Homomorphism
3. Foldr
   - Definition
   - Fusion
   - Tupling
4. Application: Parallelization

Mathematical Structures in Programs
– Foldr (Catamorphism, Right Reduction) –

Zhenjiang Hu

National Institue of Informatics

June 6, 2011

Foldr

Foldr is the most essential and simplest computation pattern on the cons lists.

$$[\alpha] = [\,] \mid \alpha : [\alpha]$$

Foldr

Foldr is the most essential and simplest computation pattern on the cons lists.

$$[\alpha] = [\,] \mid \alpha : [\alpha]$$

Given e and $\oplus$, the following computation pattern

$$\begin{array}{lcl} h\,[\,] & = & e \\ h\,(x : xs) & = & x \oplus h\,xs \end{array}$$

takes a lists, replaces $[\,]$ by e and $(:)$ by $\oplus$, and evaluates the result.

Foldr

Foldr is the most essential and simplest computation pattern on the cons lists.

$$[\alpha] = [\,] \mid \alpha : [\alpha]$$

Given e and $\oplus$, the following computation pattern

$$\begin{array}{lcl} h\,[\,] & = & e \\ h\,(x : xs) & = & x \oplus h\,xs \end{array}$$

takes a lists, replaces $[\,]$ by e and $(:)$ by $\oplus$, and evaluates the result.

Example

$$x_1 : (x_2 : (x_3 : (x_4 : [\,]))) \xrightarrow[h]{[\,]\Rightarrow e \;\; :\Rightarrow\oplus} x_1 \oplus (x_2 \oplus (x_3 \oplus (x_4 \oplus e)))$$

foldr

The pattern of computation by h is captured by a higher order function *foldr*.

$$
\begin{array}{lcl}
foldr & :: & (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta \\
foldr\ f\ e\ [\,] & = & e \\
foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs)
\end{array}
$$

foldr

The pattern of computation by h is captured by a higher order function *foldr*.

$$
\begin{array}{lcl}
foldr & :: & (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta \\
foldr\ f\ \ e\ [\,] & = & e \\
foldr\ f\ \ e\ (x : xs) & = & f\ \ x\ (foldr\ f\ \ e\ xs)
\end{array}
$$

Example: h in *foldr*

$$h \quad = \quad foldr\ (\oplus)\ e$$

Specification with *foldr*

Many list functions can be specified by a single *foldr*.

Specification with *foldr*

Many list functions can be specified by a single *foldr*.

Functions in *foldr*

$$
\begin{array}{lcl}
sum & = & foldr\ (+)\ 0 \\
product & = & foldr\ (*)\ 1 \\
and & = & foldr\ (\wedge)\ True \\
or & = & foldr\ (\vee)\ False \\
maximum & = & foldr\ max\ (-\infty) \\
minimum & = & foldr\ min\ \infty \\
length & = & foldr\ oneplus\ 0 \quad \textbf{where}\ oneplus\ x\ r = 1 + r \\
concat & = & foldr\ (+\!\!+)\ [\,]
\end{array}
$$

Specification with *foldr*: *reverse*

Problem

Write a *foldr* specification for reversing a list.

$$reverse\ [x_1, x_2, \ldots, x_n] \rightarrow [x_n, \ldots, x_2, x_1]$$

Specification with *foldr*: *reverse*

Problem

Write a *foldr* specification for reversing a list.

$$reverse\ [x_1, x_2, \ldots, x_n] \rightarrow [x_n, \ldots, x_2, x_1]$$

Idea: Assuming $reverse = foldr\ (\oplus)\ e$, we obtain e and $\oplus$ by considering the following two questions.

- What is the result of $reverse\ [\,]$? This helps derive e.
- How to obtain the result of $reverse\ (x : xs)$ from x and r, given that r is the result of $reverse\ xs$. This helps derive $\oplus$.

Specification with *foldr*: *reverse*

Problem

Write a *foldr* specification for reversing a list.

$$reverse\ [x_1, x_2, \ldots, x_n] \rightarrow [x_n, \ldots, x_2, x_1]$$

Idea: Assuming $reverse = foldr\ (\oplus)\ e$, we obtain e and $\oplus$ by considering the following two questions.

- What is the result of $reverse$ [ ]? [ ]
- How to obtain the result of $reverse\ (x : xs)$ from x and r, given that r is the result of $reverse\ xs$. This helps derive $\oplus$.

Specification with *foldr*: *reverse*

Problem

Write a *foldr* specification for reversing a list.

$$reverse\ [x_1, x_2, \ldots, x_n] \rightarrow [x_n, \ldots, x_2, x_1]$$

Idea: Assuming $reverse = foldr\ (\oplus)\ e$, we obtain e and $\oplus$ by considering the following two questions.

- What is the result of $reverse\ []$? $[]$
- How to obtain the result of $reverse\ (x : xs)$ from x and r, given that r is the result of $reverse\ xs$. $r +\!\!+ [x]$

Specification with *foldr*: *reverse*

Problem

Write a *foldr* specification for reversing a list.

$$reverse\ [x_1, x_2, \ldots, x_n] \rightarrow [x_n, \ldots, x_2, x_1]$$

Idea: Assuming $reverse = foldr\ (\oplus)\ e$, we obtain e and $\oplus$ by considering the following two questions.

- What is the result of $reverse\ [\,]$? $[\,]$
- How to obtain the result of $reverse\ (x : xs)$ from x and r, given that r is the result of $reverse\ xs$. $r +\!\!+ [x]$

Answer

$$reverse = foldr\ (\oplus)\ [\,] \quad \textbf{where}\ x \oplus r = r +\!\!+ [x]$$

How expressive is a *foldr*?

Many list functions can be described by a single *foldr*, but not all list functions can be described by a single foldr.

How expressive is a *foldr*?

Many list functions can be described by a single *foldr*, but not all list functions can be described by a single foldr.

Example

The *decimal* function cannot be described by a single foldr.

$$decimal\ [x_0, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{n-k}$$

How expressive is a *foldr*?

Many list functions can be described by a single *foldr*, but not all list functions can be described by a single foldr.

Example

The *decimal* function cannot be described by a single foldr.

$$decimal\ [x_0, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{n-k}$$

Proof Sketch. Suppose $decimal = foldr\ (\oplus)\ e$. Then contradiction happens.

$$\begin{array}{rcl} 123 & = & decimal\ [1,2,3] \\ & = & 1 \oplus decimal\ [2,3] \\ & = & 1 \oplus 23 \\ & = & 1 \oplus decimal\ [0,2,3] \\ & = & decimal\ [1,0,2,3] \\ & = & 1023 \end{array}$$

When can a function be described by a *foldr*?

Theorem (Gibbons&Hutton:2001)

A list function h can be described by a *foldr*, if and only if for all x, xs, ys,

$$h\ xs = h\ ys \;\Rightarrow\; h(x : xs) = h(x : ys).$$

When can a function be described by a *foldr*?

Theorem (Gibbons&Hutton:2001)

A list function h can be described by a *foldr*, if and only if for all x, xs, ys,

$$h\ xs = h\ ys \;\Rightarrow\; h(x : xs) = h(x : ys).$$

Exercise L3-6

Use the above theorem to prove that *sum* can be described by a fold, whereas *decimal* cannot.

When can a function be described in terms of *foldr*?

Theoretically, we have the following theorem.

Theorem

Any list function can be described by a foldr followed by a project function.

When can a function be described in terms of *foldr*?

Theoretically, we have the following theorem.

Theorem

Any list function can be described by a foldr followed by a project function.

Proof Sketch. Let h be a list function. It is always possible to define it as follows.

$$\begin{array}{l} fst \cdot foldr\ (\oplus)\ (h\ [\,], [\,]) \\ \quad \textbf{where}\ x \oplus (r_1, r_2) = (h\ (x : r_2), x : r_2) \end{array}$$

When can a function be described in terms of *foldr*?

Theoretically, we have the following theorem.

Theorem

Any list function can be described by a foldr followed by a project function.

Proof Sketch. Let h be a list function. It is always possible to define it as follows.

$$\begin{array}{l} fst \cdot foldr\ (\oplus)\ (h\ [\,], [\,]) \\ \quad \textbf{where}\ x \oplus (r_1, r_2) = (h\ (x : r_2), x : r_2) \end{array}$$

Practically, this is not useful because it never reuses the recursive results at all.

Mathematical Structures in Programming – Fusion –

Zhenjiang Hu

National Institute of Informatics

June 6, 2011

max

Consider the function to compute the maximum of a list:

$$\begin{array}{rcl} max & : & [Int] \rightarrow Int \\ max & = & head \cdot sort \end{array}$$

where *sort* is defined by

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

How to eliminate all intermediate results in computing *max*?

reverse

Consider the following function to reverse a list:

$$
\begin{array}{lcl}
rev\ x & = & fastrev'\ x\ [\,] \\
fastrev'\ x\ y & = & reverse\ x \mathbin{+\!\!+} y \\
 & & \textbf{where} \\
 & & \quad reverse = foldr\ (\lambda a\, r\, .\, r \mathbin{+\!\!+} [a])\ [\,]
\end{array}
$$

How to eliminate the intermediate list in computing *fastrev'*?

sumsq

Consider the function to compute the sum of squares of numbers from one number to the other.

$$
\begin{array}{lcl}
sumsq & : & (Int, Int) \rightarrow Int \\
sumsq & = & sum \cdot map\ square \cdot upto \\
& & \textbf{where} \\
& & \quad sum = foldr\ (+)\ 0 \\
& & \quad upto = unfold\ \ fstGreater\ \ fst\ \ succFst
\end{array}
$$

How to eliminate all intermediate results in computing *sumsq*?

Fusion Law for Foldr

Lemma (Foldr Fusion)

$$\frac{f\,(a \oplus r) = a \otimes f\ r}{f \circ foldr\ (\oplus)\ e = foldr\ (\otimes)\ (f\ e)}$$

Fusion: max

Consider the fusion for *max*:

$$max = head \circ foldr\ insert\ [\,]$$

where we assume that $head\ [\,] = -\infty$.

Fusion: max

Consider the fusion for *max*:

$$max = head \circ foldr\ insert\ [\,]$$

where we assume that $head\ [\,] = -\infty$.

To apply the foldr fusion lemma, we consider calculation of $head\ (insert\ a\ r)$.

We calculate as follows.

- For the case of $r = [\,]$, we have:

$$
\begin{array}{ll}
 & \mathit{head}\ (\mathit{insert}\ a\ [\,]) \\
= & \quad \{\ \text{def. of } \mathit{insert}\ \} \\
 & \mathit{head}\ [a] \\
= & \quad \{\ \text{def. of } \mathit{head}\ \} \\
 & a
\end{array}
$$

Fusion Example: max

- For the case of $r = b : x$, we have:

$$
\begin{array}{ll}
 & head\ (insert\ a\ (b : x)) \\
= & \quad \{\ \text{def. of } insert\ \} \\
 & head\ (\textbf{if}\ a \geq b\ \textbf{then}\ a : (b : x)\ \textbf{else}\ b : insert\ a\ x) \\
= & \quad \{\ \text{distribute } head \text{ over } if\ \} \\
 & \textbf{if}\ a \geq b\ \textbf{then}\ head\ (a : (b : x))\ \textbf{else}\ head\ (b : insert\ a\ x) \\
= & \quad \{\ \text{def. of } head\ \} \\
 & \textbf{if}\ a \geq b\ \textbf{then}\ a\ \textbf{else}\ b \\
= & \quad \{\ \text{assumption: } r = b : x\ \} \\
 & \textbf{if}\ a \geq head\ r\ \textbf{then}\ a\ \textbf{else}\ head\ r
\end{array}
$$

In summary, we have

$$\begin{array}{l} head\ (insert\ a\ r) = a \otimes head\ r \\ \quad \textbf{where}\ a \otimes r = \textbf{if}\ a \geq r\ \textbf{then}\ a\ \textbf{else}\ r \end{array}$$

It follows from the foldr fusion lemma that we get the following new definition for *max*.

$$max = foldr\ (\otimes)\ (-\infty)$$

A linear time program!

Fusion Example: Fast Reverse

Consider fusion of the following program:

$$\mathit{fastrev}'\ x\ y = \mathit{reverse}\ x \mathbin{+\!\!+} y$$

Fusion Example: Fast Reverse

Consider fusion of the following program:

$$fastrev'\ x\ y = reverse\ x \mathbin{+\!\!+} y$$

Exercise

What is the intermediate list in the above computation?

Fusion Example: Fast Reverse

Consider fusion of the following program:

$$fastrev'\ x\ y = reverse\ x \mathbin{+\!\!+} y$$

Exercise

What is the intermediate list in the above computation?

We can see where fusion calculation is application if we rewrite the definition.

$$\begin{array}{lcl} fastrev'\ x & = & (\mathbin{+\!\!+})\ (reverse\ x) \\ & = & (\underline{(\mathbin{+\!\!+})} \circ foldr\ \underline{(\lambda a\, r.\, r \mathbin{+\!\!+} [a])}\ [\,])\ x \end{array}$$

Let us calculate the fusion condition:

$$
\begin{array}{ll}
 & (+\!\!+)\ (r +\!\!+ [a]) \\
= & \quad \{\ \eta \text{ expansion }\ \} \\
 & \lambda y.\,(+\!\!+)\ (r +\!\!+ [a])\ y \\
= & \quad \{\ \text{section notation }\ \} \\
 & \lambda y.\,(r +\!\!+ [a]) +\!\!+ y \\
= & \quad \{\ \text{associativity of } +\!\!+\ \} \\
 & \lambda y.\,r +\!\!+ ([a] +\!\!+ y)
\end{array}
$$

Marching it with $a \otimes ((+\!\!+)\ r)$ gives

$$a \otimes r' = \lambda y.\,r'\ ([a] +\!\!+ y)$$

So we get

$$\begin{array}{l} \mathit{fastrev}'\ x = \mathit{foldr}\ (\otimes)\ ((+\!\!+)\ [\,])\ x \\ \quad \textbf{where}\ a \otimes r' = \lambda y.\, r'\ ([a] +\!\!+ y) \end{array}$$

So we get

$$\begin{array}{c} \mathit{fastrev}'\ x = \mathit{foldr}\ (\otimes)\ ((+\!\!+)\ [\,])\ x \\ \textbf{where}\ a \otimes r' = \lambda y.\, r'\ ([a] +\!\!+ y) \end{array}$$

That is,

$$\begin{array}{lcl} \mathit{fastrev}'\ [\,]\ y & = & y \\ \mathit{fastrev}'\ (a : r)\ y & = & \mathit{fastrev}'\ r\ (a : y) \end{array}$$

So we get

$$\begin{array}{c} \mathit{fastrev}'\ x = \mathit{foldr}\ (\otimes)\ ((+\!\!+)\ [\,])\ x \\ \textbf{where}\ a \otimes r' = \lambda y.\, r'\ ([a] +\!\!+ y) \end{array}$$

That is,

$$\begin{array}{lcl} \mathit{fastrev}'\ [\,]\ y & = & y \\ \mathit{fastrev}'\ (a : r)\ y & = & \mathit{fastrev}'\ r\ (a : y) \end{array}$$

A linear time algorithm!

Fusion Example: *sumsq*

$$
\begin{array}{ll}
 & sumsq \\
= & \quad \{ \text{ def. of } sumsq \ \} \\
 & sum \circ map\ square \circ upto \\
= & \quad \{ \text{ map is a foldr } \} \\
 & sum \circ foldr\ (\lambda a\, r.\, square\ a : r)\ []\circ upto \\
= & \quad \{ \text{ foldr fusion } \} \\
 & foldr\ \ (\lambda a\, r.\, square\ a + r)\ 0 \circ upto \\
= & \quad \{ \text{ upto is an unfold } \} \\
 & hylo\ (\lambda a\, r.\, square\ a + r, 0)\ (fstGreater, fst, succFst)
\end{array}
$$

Fusion of *sumsq*

Unfolding the definition of *hylo* yields the following recursive definition.

$$
\begin{array}{lcl}
\mathit{sumsq}\ (m, n) & = & \textbf{if}\ m > n\ \textbf{then}\ 0 \\
 & & \textbf{else}\ \mathit{square}\ m + \mathit{sumsq}\ (m+1, n)
\end{array}
$$

Fusion of *sumsq*

Unfolding the definition of *hylo* yields the following recursive definition.

$$\begin{array}{lcl} \mathit{sumsq}\ (m, n) & = & \textbf{if}\ m > n\ \textbf{then}\ 0 \\ & & \textbf{else}\ \mathit{square}\ m + \mathit{sumsq}\ (m+1, n) \end{array}$$

No intermediate list exisits now!

Mathematical Structures in Programming
– Tupling –

Zhenjiang Hu

National Institute of Informatics

June 6, 2011

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$\mathit{biggers}\ [3, 10, 4, -2, 1, 3] = [10, 4, -3]$$

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$biggers\ [3, 10, 4, -2, 1, 3] = [10, 4, -3]$$

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

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$biggers\ [3, 10, 4, -2, 1, 3] = [10, 4, -3]$$

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

How can we optimize this program?

Definition (Mutumorphism)

Functions $f_1, \ldots, f_n$ are said to form a mutumorphism 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 the function $f \; x = (f_1 \; x, \ldots, f_n \; x)$ as follows.

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

Expressive Power of Mutumorphism

- *foldr* is a special case:

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

- It covers all primitive recursive functions on lists.

$$\begin{array}{lcl} prim\ [\,] & = & e \\ prim\ (a : x) & = & F(a, x, prim\ x) \end{array}$$

This is because we can consider *prim* is mutually defined with the identity function on lists.

biggers as a Mutumorphism

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

Lemma (Mutu-Tupling)

$$
\begin{aligned}
&[\![(e_1, e_2, \ldots, e_n), (\oplus_1, \oplus_2, \ldots, \oplus_n)]\!] \\
&\quad = \mathit{foldr}\ (\oplus)\ (e_1, e_2, \ldots, e_n) \\
&\qquad \textbf{where}\ a \oplus r = (a \oplus_1 r, a \oplus_2 r, \ldots, a \oplus_n r)
\end{aligned}
$$

Consider, as an example, to apply the mutu-tupling lemma to *biggers*.

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

Inlining *foldr* in the derived program gives the following readable recursive program:

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

Lemma (Foldr-Tupling)

$$(foldr\ (\oplus_1)\ e_1\ x,\ \ foldr\ (\oplus_2)\ e_2\ x) = foldr\ (\oplus)\ (e_1, e_2)\ x$$
$$\textbf{where}\ a \oplus (r_1, r_2) = (a \oplus_1 r_1, a \oplus_2 r_2)$$

For example, the following program for computing the average of a list:

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

can be transformed into the following with the foldr-tupling lemma.

$$\begin{array}{l} average\ x = \textbf{let}\ (s, l) = tup\ x\ \textbf{in}\ s/l \\ \qquad \textbf{where}\ tup = foldr\ (\lambda a\ (s, l).\ (a + s, 1 + l))\ (0, 0) \end{array}$$

Outline

Mathematical Structures in Programming – Calculational Parallelization –

Zhenjiang Hu

National Institute of Informatics

Februry 9, 2010

Maximum Prefix Sum Problem

Design a D&C parallel program that computes the maximum of all the prefix sums of a list.

$$mps\ [\underline{1, -2, 3, -9, 5, 7}, -10, 8, -9, 10] = 5$$

List Homomorphism

Function h on lists is a list homomorphism, if

$$\begin{array}{lcl} h\ [\,] & = & e \\ h\ [a] & = & f\ \ a \\ h\ (x +\!\!+\ y) & = & h\ x \odot h\ y \end{array}$$

for some $\odot$.

Properties

- Suitable for parallel computation in the D&C style
- Basic concept for skeletal parallel programming
- Enjoy many nice algebraic properties (1st, 2nd, 3rd Homomorphism theorems)

The Third Homomorphism Theorem (Gibbons:JFP95)

A function f can be described as a foldl and a foldr

$$
\begin{array}{lcl}
f & = & foldr\ (\oplus)\ e \\
f & = & foldl\ (\otimes)\ e
\end{array}
$$

that is,

$$
\begin{array}{lcl}
f\ (a : x) & = & a \oplus f\ x \\
f\ (x +\!\!+ [a]) & = & f\ x \otimes a
\end{array}
$$

iff there exists an associative operator $\odot$ such that

$$f(x +\!\!+ y) = f\ x \odot f\ y.$$

The Third Homomorphism Theorem (Gibbons:JFP95)

A function f can be described as a foldl and a foldr

$$
\begin{array}{lcl}
f & = & \mathit{foldr}\ (\oplus)\ e \\
f & = & \mathit{foldl}\ (\otimes)\ e
\end{array}
$$

that is,

$$
\begin{array}{lcl}
f\ (a : x) & = & a \oplus f\ x \\
f\ (x +\!\!+ [a]) & = & f\ x \otimes a
\end{array}
$$

iff there exists an associative operator $\odot$ such that

$$f(x +\!\!+ y) = f\ x \odot f\ y.$$

Two sequential programs guarantee existence of a parallel program!

The Third Homomorphism Theorem (Gibbons:JFP95)

A function f can be described as a foldl and a foldr

$$\begin{array}{rcl} f & = & \mathit{foldr}\ (\oplus)\ e \\ f & = & \mathit{foldl}\ (\otimes)\ e \end{array}$$

that is,

$$\begin{array}{rcl} f\ (a : x) & = & a \oplus f\ x \\ f\ (x +\!\!+ [a]) & = & f\ x \otimes a \end{array}$$

iff there exists an associative operator $\odot$ such that

$$f(x +\!\!+ y) = f\ x \odot f\ y.$$

Two sequential programs guarantee existence of a parallel program!

A cons-list cata + A snoc-list cata $\Leftrightarrow$ A join-list cata

Existence of Homomorphism (Review)

Existence Lemma

The list function h is a homomorphism iff the implication

$$h\ v = h\ x\ \wedge\ h\ w = h\ y\ \Rightarrow\ h\ (v +\!\!+\, w) = h\ (x +\!\!+\, y)$$

holds for all lists v, w, x, y.

Proof of the Third Homomorphism Theorem

Proof. Let $h\ v = h\ x$ and $h\ w = h\ y$. Then:

$$
\begin{array}{ll}
 & h\ (v \mathbin{+\!\!+} w) \\
= & \quad \{\ h = \mathit{foldr}\ (\oplus)\ e\ \} \\
 & \mathit{foldr}\ (\oplus)\ e\ (v \mathbin{+\!\!+} w) \\
= & \quad \{\ \text{property of foldr}\ \} \\
 & \mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ (\oplus)\ e\ w)\ v \\
= & \quad \{\ h\ w = h\ y\ \} \\
 & \mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ (\oplus)\ e\ y)\ v \\
= & \quad \{\ \text{property of foldr}\ \} \\
 & \mathit{foldr}\ (\oplus)\ e\ (v \mathbin{+\!\!+} y) \\
= & \quad \{\ h = \mathit{foldr}\ (\oplus)\ e\ \} \\
 & h\ (v \mathbin{+\!\!+} y) \\
= & \quad \{\ \text{symmetrically, since } h = \mathit{foldl}\ (\otimes)\ e\ \} \\
 & h\ (x \mathbin{+\!\!+} y)
\end{array}
$$

Here by the Existence Lemma, h is a homomorphism.

Examples

- $sum\ [1,2,3] = 6$

$$\begin{array}{lcl} sum\ (a : x) & = & a + sum\ x \\ sum\ (x +\!\!+ [a]) & = & sum\ x + a \end{array}$$

Examples

- $sum\ [1,2,3] = 6$

$$\begin{array}{lcl} sum\ (a : x) & = & a + sum\ x \\ sum\ (x +\!\!+ [a]) & = & sum\ x + a \end{array}$$

- $sort\ [1,3,2] = [1,2,3]$

$$\begin{array}{lcl} sort\ (a : x) & = & insert\ a\ (sort\ x) \\ sort\ (x +\!\!+ [b]) & = & insert\ b\ (sort\ x) \end{array}$$

Examples

- $sum\ [1,2,3] = 6$

$$\begin{array}{lcl} sum\ (a : x) & = & a + sum\ x \\ sum\ (x +\!\!+ [a]) & = & sum\ x + a \end{array}$$

- $sort\ [1,3,2] = [1,2,3]$

$$\begin{array}{lcl} sort\ (a : x) & = & insert\ a\ (sort\ x) \\ sort\ (x +\!\!+ [b]) & = & insert\ b\ (sort\ x) \end{array}$$

- $psums\ [1,2,3] = [1, 1+2, 1+2+3]$

$$\begin{array}{lcl} psums\ (a : x) & = & a : map\ (a+)\ (psums\ x) \\ psums\ (x +\!\!+ [b]) & = & psums\ x +\!\!+ [last\ (psums\ x) + b] \end{array}$$

A Challenge Problem

It remains as a challenge to automatically derive *efficient* an associative operator $\odot$ from $\oplus$ and $\otimes$.

Parallelization Theorem

Let f° denote a weak right inverse of f.

$$\frac{\begin{array}{lcl} f(a : x) & = & a \oplus f\ x \\ f(x +\!\!+ [b]) & = & f\ x \otimes b \end{array}}{\begin{array}{l} f(x +\!\!+ y) = f\ x \odot f\ y \\ \quad \textbf{where}\ \ a \odot b = f(f^\circ\ a +\!\!+ f^\circ\ b) \end{array}}$$

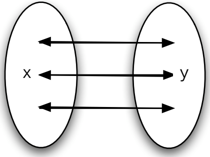
Weak (Right) Inverse

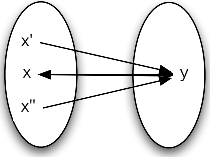
- g is an inverse of f, if

$$g\ y = x \ \Leftrightarrow\ f\ x = y$$

- g is a weak (right) inverse of f, if for $y \in \mathrm{image}(f)$

$$g\ y = x \ \Rightarrow\ f\ x = y$$





Properties of Weak Inverse

- Weak inverse always exists but may not be unique.

  Example: Function *sum*

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

  can have infinite number of weak inverse:

Properties of Weak Inverse

- Weak inverse always exists but may not be unique.

  Example: Function *sum*

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

  can have infinite number of weak inverse:

  $$\begin{array}{lcl} g_1\ y & = & [y] \\ g_2\ y & = & [0, y] \\ & \cdots & \end{array}$$

Parallelizing *sum*

From

1. $sum\ (a : x) = a + sum\ x$
2. $sum\ (x ++ [b]) = sum\ x + b$
3. $sum^{\circ}\ y = [y]$

we soon obtain

$$sum\ (x ++ y) = sum\ x \odot sum\ y$$

where

$$\begin{aligned} a \odot b &= sum\ (sum^{\circ}\ a ++ sum^{\circ}\ b) \\ &= sum\ ([a] ++ [b]) \\ &= a + b \end{aligned}$$

Parallelizing *sum*

From

1. $sum\ (a : x) = a + sum\ x$
2. $sum\ (x +\!\!+ [b]) = sum\ x + b$
3. $sum^\circ\ y = [y]$

we soon obtain

$$
\begin{array}{l}
sum\ (x +\!\!+ y) = sum\ x \odot sum\ y \\
\quad \text{where} \\
\qquad \begin{array}{rcl}
a \odot b & = & sum\ (sum^\circ\ a +\!\!+ sum^\circ\ b) \\
& = & sum\ ([a] +\!\!+ [b]) \\
& = & a + b
\end{array}
\end{array}
$$

That is,

$$sum\ (x +\!\!+ y) = sum\ x + sum\ y.$$

Weak inversion is not easy!

- What is a weak inverse for *sum*?

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

Weak inversion is not easy!

- What is a weak inverse for *sum*? $\underline{sum^\circ\ y = [y]}$

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

Weak inversion is not easy!

- What is a weak inverse for *sum*? $\underline{sum^\circ\ y = [y]}$

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

- What is it for *mps*?

$$\begin{array}{l} mps\ [\,] = 0 \\ mps\ (a : x) = 0 \uparrow a \uparrow (a + mps\ x) \end{array}$$

Weak inversion is not easy!

- What is a weak inverse for *sum*? $\underline{sum^\circ\ y = [y]}$

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

- What is it for *mps*? $\underline{mps^\circ\ y = [y]}$

$$\begin{array}{l} mps\ [\,] = 0 \\ mps\ (a:x) = 0 \uparrow a \uparrow (a + mps\ x) \end{array}$$

Weak inversion is not easy!

- What is a weak inverse for *sum*? $\underline{sum^{\circ}\ y = [y]}$

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

- What is it for *mps*? $\underline{mps^{\circ}\ y = [y]}$

$$\begin{array}{l} mps\ [\,] = 0 \\ mps\ (a : x) = 0 \uparrow a \uparrow (a + mps\ x) \end{array}$$

- What is it for $f = mps \vartriangle sum$?

$$f\ x = (mps\ x, sum\ x)$$

Weak inversion is not easy!

- What is a weak inverse for *sum*? $\underline{sum^{\circ}\ y = [y]}$

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

- What is it for *mps*? $\underline{mps^{\circ}\ y = [y]}$

$$\begin{array}{l} mps\ [\,] = 0 \\ mps\ (a : x) = 0 \uparrow a \uparrow (a + mps\ x) \end{array}$$

- What is it for $f = mps \vartriangle sum$? $\underline{f^{\circ}\ (p, s) = [p, s - p]}$

$$f\ x = (mps\ x, sum\ x)$$

Weak inversion is challenging

Can you find a weak inverse for f?

$$f\ x = (mss\ x, mps\ x, mts\ x, sum\ x)$$

where

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

Weak inversion is challenging

Can you find a weak inverse for f?

$$f\ x = (mss\ x, mps\ x, mts\ x, sum\ x)$$

where

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

$$\underline{f^{\circ}\ (m, p, t, s) = [p, s - p - t, m, t - m]}$$

Derivation of Weak Right Inverse

- Idea:

  deriving a weak right inverse

  $\Downarrow$

  solving conditional linear equations

Derivation of Weak Right Inverse

- Idea:

  deriving a weak right inverse
  $$\Downarrow$$
  solving conditional linear equations

- Consider to find a weak right inverse for f defined by

  $$f\ x = (mps\ x, sum\ x)$$

  Let x_1, x_2 be a solution to the following equations:

  $$\begin{array}{rcl} mps\ [x_1, x_2] & = & p \\ sum\ [x_1, x_2] & = & s \end{array}$$

  then

  $$f^{\circ}\ (p, s) = [x_1, x_2]$$

Derivation of Weak Right Inverse

- Idea:

deriving a weak right inverse

$\Downarrow$

solving conditional linear equations

- Consider to find a weak right inverse for f defined by

$$f\ x = (mps\ x, sum\ x)$$

Let x_1, x_2 be a solution to the following equations:

$$\begin{array}{lcl} 0 \uparrow x_1 \uparrow (x_1 + x_2) & = & p \\ x_1 + x_2 & = & s \end{array}$$

then

$$f^{\circ}\ (p, s) = [x_1, x_2]$$

Derivation of Weak Right Inverse

- Idea:

  deriving a weak right inverse
  $$\Downarrow$$
  solving conditional linear equations

- Consider to find a weak right inverse for f defined by

$$f\ x = (mps\ x, sum\ x)$$

Let x_1, x_2 be a solution to the following equations:

$$\begin{array}{rcl} x_1 & = & p \\ x_2 & = & s - p \end{array}$$

then

$$f^\circ\ (p, s) = [x_1, x_2]$$

Derivation of Weak Right Inverse

- Idea:

  deriving a weak right inverse
  $$\Downarrow$$
  solving conditional linear equations

- Consider to find a weak right inverse for f defined by

  $$f\ x = (mps\ x, sum\ x)$$

  Let x_1, x_2 be a solution to the following equations:

  $$\begin{array}{lll} x_1 & = & p \\ x_2 & = & s - p \end{array}$$

  then

  $$f^\circ\ (p, s) = [p, s - p]$$

Conditional Linear Equations

$$
\begin{aligned}
t_1(x_1, x_2, \ldots, x_m) &= c_1 \\
t_2(x_1, x_2, \ldots, x_m) &= c_2 \\
&\vdots \\
t_m(x_1, x_2, \ldots, x_m) &= c_m
\end{aligned}
$$

Conditional Linear Equations

$$
\begin{array}{rcl}
t_1(x_1, x_2, \ldots, x_m) & = & c_1 \\
t_2(x_1, x_2, \ldots, x_m) & = & c_2 \\
& \vdots & \\
t_m(x_1, x_2, \ldots, x_m) & = & c_m
\end{array}
$$

$$
\begin{array}{lcl}
t & ::= & n \mid x \mid n\ x \mid t_1 + t_2 \mid p \rightarrow t_1; t_2 \\
p & ::= & t_1 < t_2 \mid t_1 = t_2 \mid \neg p \mid p_1 \wedge p_2 \mid p_1 \vee p_2
\end{array}
$$

Conditional Linear Equations

$$
\begin{array}{rcl}
t_1(x_1, x_2, \ldots, x_m) & = & c_1 \\
t_2(x_1, x_2, \ldots, x_m) & = & c_2 \\
& \vdots & \\
t_m(x_1, x_2, \ldots, x_m) & = & c_m
\end{array}
$$

Conditional linear equations can be efficiently solved by using Mathematica. [PLDI'07]

Can we generalize the idea from lists to trees?

$$\frac{\begin{array}{lcl} f(a:x) & = & a \oplus f\ x \\ f(x +\!\!+ [b]) & = & f\ x \otimes b \end{array}}{\begin{array}{l} f(x +\!\!+ y) = f\ x \odot f\ y \\ \quad \textbf{where}\ \ a \odot b = f(f^\circ\ a +\!\!+ f^\circ\ b) \end{array}}$$

$$\Downarrow$$

$$\frac{\begin{array}{l} \text{f is a bottom-up tree reduction} \\ \text{f is a top-down tree reduction} \end{array}}{\begin{array}{l} f(t_1 \triangleleft t_2) = f\ t_1 \odot f\ t_2 \\ \quad \textbf{where}\ \ a \odot b = f(f^\circ\ a \triangleleft f^\circ\ b) \end{array}}$$

(Binary) Trees

- Definition:

$$\begin{array}{lcl} \mathit{Tree}\ a & = & N\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\ & \mid & E \\ & \mid & \bullet \end{array}$$

We assume that every tree contains one and only one hole $\bullet$.

Hole Filling Operator $\triangleleft$

- Definition:

$$
\begin{array}{lcl}
(\triangleleft) & : & Tree\ a \rightarrow Tree\ a \rightarrow Tree\ a \\
E \triangleleft t & = & E \\
(N\ n\ l\ r) \triangleleft t & = & N\ n\ (l \triangleleft t)\ (r \triangleleft t) \\
\bullet \triangleleft t & = & t
\end{array}
$$

Hole Filling Operator $\triangleleft$

- Definition:

$$
\begin{array}{lcl}
(\triangleleft) & : & Tree\ a \to Tree\ a \to Tree\ a \\
E \triangleleft t & = & E \\
(N\ n\ l\ r) \triangleleft t & = & N\ n\ (l \triangleleft t)\ (r \triangleleft t) \\
\bullet \triangleleft t & = & t
\end{array}
$$

- $(\triangleleft, \bullet)$ forms a monoid.

$$
(t_1 \triangleleft t_2) \triangleleft t_3 = t_1 \triangleleft (t_2 \triangleleft t_3)
$$
$$
\bullet \triangleleft t = t \triangleleft \bullet = t
$$

Bottom-up Tree Reduction

- Definition: f is a bottom-up tree reduction, if there exists a function k such that

$$f\ (N\ a\ l\ r)\ \ =\ \ k\ a\ (f\ \ l)\ (f\ \ r).$$

Top-Down Tree Reduction

- Definition: f is a top-down tree reduction, if there exist two functions k_l and k_r such that:

$$\begin{array}{lcl} f\ (t \triangleleft (N\ a\ l\ \bullet)) & = & k_l\ a\ (f\ t)\ (f\ l), \\ f\ (t \triangleleft (N\ a\ \bullet\ r)) & = & k_r\ a\ (f\ t)\ (f\ r). \end{array}$$

Tree Homomorphism

- Definition: h is a tree homomorphism if there exists an associative operator $\oplus$ such that:

$$h\ (t_1 \triangleleft t_2) \quad = \quad h\ t_1 \oplus h\ t_2.$$

The Tree Homomorphism Theorem

There exist k, k_l and k_r such that the function f can be defined in the following form

$$\begin{array}{lcl} f\ (N\ a\ l\ r) & = & k\ a\ (f\ l)\ (f\ r) \\ f\ (t \triangleleft (N\ a\ l\ \bullet)) & = & k_l\ a\ (f\ t)\ (f\ l) \\ f\ (t \triangleleft (N\ a\ \bullet\ r)) & = & k_r\ a\ (f\ t)\ (f\ r) \end{array}$$

if and only if

$$\begin{array}{l} f\ (t_1 \triangleleft t_2) = f\ t_1 \odot f\ t_2 \\ \quad \textbf{where}\ a \odot b = f\ (f^o\ a \triangleleft f^o\ b) \end{array}$$

where f^o is a weak (right) inverse of f.