Program Calculus
– Calculational Programming –

Zhenjiang Hu

National Institute of Informatics

June 21 / June 28 / July 5, 2010

What we will learn?

- Discussing the mathematical structures in programs, and

What we will learn?

- Discussing the mathematical structures in programs, and
- explaining how mathematical reasoning plays an important role in designing efficient and correct algorithms (programs).

What we will learn?

- Discussing the mathematical structures in programs, and
- explaining how mathematical reasoning plays an important role in designing efficient and correct algorithms (programs).

$$\Uparrow$$

A new programming style: calculational programming

Calculation is widely used in solving our daily problems, but its importance in programming has not been fully recognized.

Tsuru-Kame-Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

Tsuru-Kame-Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- The kindergarten approach: plain simple enumeration!
  - Crane 0, Tortoise 5 . . . No.
  - Crane 1, Tortoise 4 . . . No.
  - Crane 2, Tortoise 3 . . . No.
  - Crane 3, Tortoise 2 . . . Yes!
  - Crane 4, Tortoise 1 . . . No.
  - Crane 5, Tortoise 0 . . . No.

Tsuru-Kame-Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- Elementary school: let's do some reasoning ...
  - If all 5 animals were cranes, there ought to be $5 \times 2 = 10$ legs.
  - However, there are in fact 14 legs. The extra 4 legs must belong to some tortoises. There must be $(14 - 10)/2 = 2$ tortoises.
  - So there must be $5 - 2 = 3$ cranes.
- It generalises to larger numbers of heads and legs.

Tsuru-Kame-Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- Junior high school: algebra (theory of equation)!

$$\begin{array}{rcl} x + y & = & 5 \\ 2x + 4y & = & 14 \end{array}$$

- It's a general approach applicable to many other problems and perhaps easier.

Tsuru-Kame-Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

rules and theories are useful for solving problems easily and systematically.

- Problems are declaratively specified.
- Implementation is hidden.

Can we have a set of useful rules and theories in programming for developing correct and efficient algorithms (programs)?

A Programming Problem

Can you develop a correct linear-time program for solving the following problem?

Maximum Segment Sum Problem

Given a list of numbers, find the maximum of sums of all *consecutive* sublists.

- $[-1, 3, 3, -4, -1, 4, 2, -1] \implies 7$
- $[-1, 3, 1, -4, -1, 4, 2, -1] \implies 6$
- $[-1, 3, 1, -4, -1, 1, 2, -1] \implies 4$

A Simple Solution

1. Enumerating all segments (*segs*);

A Simple Solution

1. Enumerating all segments (*segs*);
2. Computing sum for each segment(*sums*);

A Simple Solution

1. Enumerating all segments (*segs*);
2. Computing sum for each segment(*sums*);
3. Calculating the maximum of all the sums (*max*).

A Simple Solution

1. Enumerating all segments (*segs*);
2. Computing sum for each segment(*sums*);
3. Calculating the maximum of all the sums (*max*).

A Simple Solution

1. Enumerating all segments (*segs*);
2. Computing sum for each segment(*sums*);
3. Calculating the maximum of all the sums (*max*).

Exercise

How many segments does a list of length n have?

A Simple Solution

1. Enumerating all segments (*segs*);
2. Computing sum for each segment(*sums*);
3. Calculating the maximum of all the sums (*max*).

Exercise

How many segments does a list of length n have?

Exercise

What is the time complexity of this simple solution?

There indeed exists a clever solution!

```
mss=0; s=0;
 for(i=0;i<n;i++){
      s += x[i];
      if(s<0) s=0;
      if(mss<s) mss= s;
 }
```

$x[i]$		3	1	-4	-1	1	2	-1
s	0	3	4	0	0	1	3	2
mss	0	3	4	4	4	4	4	4

There is a big gap between the simple and clever solutions!

There is a big gap between the simple and clever solutions!

- Can we calculate the clever solution from the simple solution?

There is a big gap between the simple and clever solutions!

- Can we calculate the clever solution from the simple solution?
- What rules and theorems are necessary to do so?

There is a big gap between the simple and clever solutions!

- Can we calculate the clever solution from the simple solution?
- What rules and theorems are necessary to do so?
- How to apply the rules and theorems to do so?

There is a big gap between the simple and clever solutions!

- Can we calculate the clever solution from the simple solution?
- What rules and theorems are necessary to do so?
- How to apply the rules and theorems to do so?

Transformational Programming

One starts by writing clean and correct programs, and then use *program transformation* techniques to transform them step-by-step to more efficient equivalents.

Specificaiton: Clean and Correct programs

$\Downarrow$

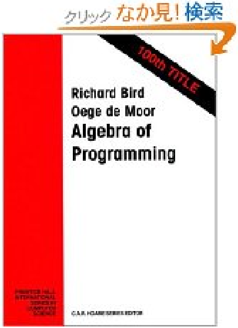
Folding/Unfolding Program Transformation

$\Downarrow$

Efficient Programs

Program Calculation

Program calculation is a kind of program transformation based on Constructive Algorithmics, a framework for developing laws/rules/theories for manipulating programs.



Program Calculation

Program calculation is a kind of program transformation based on Constructive Algorithmics, a framework for developing laws/rules/theories for manipulating programs.

Specificaiton: Clean and Correct programs

$\Downarrow$

Folding-free Program Transformation

$\Downarrow$

Efficient Programs

References

- Richard Bird, *Lecture Notes on Constructive Functional Programming*, Technical Monograph PRG-69, Oxford University, 1988.
- Anne Kaldewaij, Programming: *The Derivation of Algorithms*, Prentice Hall, 1990.
- Richard Bird and Oege de Moor, *The Algebra of Programming*, Prentice-Hall, 1996.
- Roland Backhouse, *Program Construction: Calculating Implementation from Specification*, Wiley, 2003.

Specification and Implementation

Zhenjiang Hu

National Institute of Informatics

June 21 / June 28 / July 5, 2010

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

$$\begin{aligned} & \mathit{increase} :: \mathit{Int} \rightarrow \mathit{Int} \\ & \mathit{increase}\ x > \mathit{square}\ x \end{aligned}$$

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

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.

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

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.

Introduction to Bird Meertens Formalisms

Zhenjiang Hu

National Institute of Informatics

June 21 / June 28 / July 5, 2010

Bird Meentens Formalisms (BMF)

BMF is a calculus of functions for *people* to derive programs from specifications:

- a range of concepts and notations for defining functions over lists;
- a set of algebraic laws for manipulating functions.

A Problem

Consider the following simple identity:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1$$

This equation generalizes in the obvious way to n variables $a_1, a_2, \ldots, a_n$, and we will refer to it as Horner'e rule.

A Problem

Consider the following simple identity:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1$$

This equation generalizes in the obvious way to n variables $a_1, a_2, \ldots, a_n$, and we will refer to it as Horner'e rule.

- How many $\times$ are used in each side?

A Problem

Consider the following simple identity:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1$$

This equation generalizes in the obvious way to n variables $a_1, a_2, \ldots, a_n$, and we will refer to it as Horner'e rule.

- How many $\times$ are used in each side?
- Can we generalize $\times$ to $\otimes$, $+$ to $\oplus$? What are the essential constraints for $\otimes$ and $\oplus$?

A Problem

Consider the following simple identity:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1$$

This equation generalizes in the obvious way to n variables $a_1, a_2, \ldots, a_n$, and we will refer to it as Horner'e rule.

- How many $\times$ are used in each side?
- Can we generalize $\times$ to $\otimes$, $+$ to $\oplus$? What are the essential constraints for $\otimes$ and $\oplus$?
- Do you have suitable notation for expressing the Horner's rule concisely?

Functions

- A function f that has source type α and target type β is denoted by

  $$f : \alpha \rightarrow \beta$$

  We shall say that f takes arguments in α and returns results in β.
- Function application is written without brackets; thus $f\ a$ means $f(a)$. Function application is more binding than any other operation, so $f\ a \otimes b$ means $(f\ a) \otimes b$.
- Functions are curried and applications associates to the left, so $f\ a\ b$ means $(f\ a)\ b$ (sometimes written as $f_a\ b$.

- Function composition is denoted by a centralized dot $(\cdot)$. We have

$$(f \cdot g)\ x = f(g\ x)$$

Exercise: Show the following equation state that functional compositon is associative.

$$(f\cdot) \cdot (g\cdot) = ((f \cdot g)\cdot)$$

- Binary operators will be denoted by $\oplus$, $\otimes$, $\odot$, etc. Binary operators can be sectioned. This means that $(\oplus)$, $(a\oplus)$ and $(\oplus a)$ all denote functions. The definitions are:

$$
\begin{aligned}
(\oplus)\ a\ b &= a \oplus b \\
(a\oplus)\ b &= a \oplus b \\
(\oplus b)\ a &= a \oplus b
\end{aligned}
$$

Exercise: If $\oplus$ has type $\oplus : \alpha \times \beta \rightarrow \gamma$, then what are the types for $(\oplus)$, $(a\oplus)$ and $(\oplus b)$ for all a in α and b in β?

- The identity element of $\oplus : \alpha \times \alpha \to \alpha$, if it exists, will be denoted by $id_{\oplus}$. Thus,

$$a \oplus id_{\oplus} = id_{\oplus} \oplus a = a$$

  Exericise: What is the identity element of functional composition?

- The constant values function $K : \alpha \to \beta \to \alpha$ is defined by the equation

$$K\ a\ b = a$$

Lists

- Lists are finite sequence of values of the same type. We use the notation $[\alpha]$ to describe the type of lists whose elements have type α.
  - Examples:
    $[1, 2, 1] : [\mathit{Int}]$
    $[[1], [1, 2], [1, 2, 1]] : [[\mathit{Int}]]$
    $[\,] : [\alpha]$

List Data Constructors

- $[\,] : [\alpha]$ constructs an empty list.
- $[.] : \alpha \rightarrow [\alpha]$ maps elements of α into singleton lists.

$$[.]\ a = [a]$$

- The primitive operator on lists is concatenation, denoted by $+\!\!+$.

$$[1] +\!\!+ [2] +\!\!+ [1] = [1, 2, 1]$$

Concatenation is associative:

$$x +\!\!+ (y +\!\!+ z) = (x +\!\!+ y) +\!\!+ z$$

Exercise: What is the identity for concatenation?

Algebraic View of Lists

- $([\alpha], +\!\!+, [\,])$ is a monoid.
- $([\alpha], +\!\!+, [\,])$ is a free monoid generated by α under the assignment $[.] : \alpha \to [\alpha]$.
- $([\alpha]^+, +\!\!+)$ is a semigroup.

List Functions: Homomorphisms

A function h defined in the following form is called homomorphism:

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

It defines a map from the monoid $([\alpha], +\!\!+, [\,])$ to the monoid $(\beta, \oplus : \beta \to \beta \to \beta, id_{\oplus} : \beta)$.

List Functions: Homomorphisms

A function h defined in the following form is called homomorphism:

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

It defines a map from the monoid $([\alpha], +\!\!+, [\,])$ to the monoid $(\beta, \oplus : \beta \to \beta \to \beta, id_{\oplus} : \beta)$.

Property: h is uniquely determined by f and $\oplus$.

An Example: the function returning the length of a list.

$$
\begin{array}{lcl}
\# \, [\,] & = & 0 \\
\# \, [a] & = & 1 \\
\# \, (x \mathbin{+\!\!+} y) & = & \# \, x + \# \, y
\end{array}
$$

Note that $(Int, +, 0)$ is a monoid.

Bags and Sets

- A bag is a list in which the order of the elements is ignored. Bags are constructed by adding the rule that $+\!\!+$ is commutative (as well as associative):

$$x +\!\!+ y = y +\!\!+ x$$

- A set is a bag in which repetitions of elements are ignored. Sets are constructed by adding the rule that $+\!\!+$ is idempotent (as well as commutative and associative):

$$x +\!\!+ x = x$$

Map

The operator $*$ (pronounced map) takes a function on Its left and a list on its right. Informally, we have

$$f * [a_1, a_2, \ldots, a_n] = [f\ a_1, f\ a_2, \ldots, f\ a_n]$$

Formally, $(f*)$ (or sometimes simply written as $f*$) is a homomorphism:

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

Map Distributivity: $(f \cdot g)* = (f*) \cdot (g*)$

Reduce

The operator / (pronounced reduce) takes an associative binary operator on Its left and a list on its right. Informally, we have

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

Formally, $\oplus/$ is a homomorphism:

$$\begin{array}{lcll} \oplus/[\,] & = & id_\oplus & \{ \text{ if } id_\oplus \text{ exists } \} \\ \oplus/[a] & = & a & \\ \oplus/(x +\!\!+\, y) & = & (\oplus/x) \oplus (\oplus/y) & \end{array}$$

Examples:

$$
\begin{array}{lcl}
max & : & [Int] \to Int \\
max & = & \uparrow / \\
 & & \text{where } a \uparrow b = \text{if } a \leq b \text{ then } b \text{ else } a \\
head & : & [\alpha]^{+} \to \alpha \\
head & = & \lessdot / \\
 & & \text{where } a \lessdot b = a \\
last & : & [\alpha]^{+} \to \alpha \\
last & = & \gtrdot / \\
 & & \text{where } a \gtrdot b = b
\end{array}
$$

Promotion

$f*$ and $\oplus/$ can be expressed as identities between functions.

Empty Rules

$$\begin{aligned} f * \cdot K\,[\,] &= K\,[\,] \\ \oplus/ \cdot K\,[\,] &= id_{\oplus} \end{aligned}$$

One-Point Rules

$$\begin{aligned} f * \cdot [\cdot] &= [\cdot] \cdot f \\ \oplus/ \cdot [\cdot] &= id \end{aligned}$$

Join Rules

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

An Example of Calculation

$$
\begin{array}{ll}
 & \oplus/ \cdot f * \cdot +\!\!+/ \cdot g* \\
= & \quad \{ \text{ map promotion } \} \\
 & \oplus/ \cdot +\!\!+/ \cdot f ** \cdot g* \\
= & \quad \{ \text{ reduce promotion } \} \\
 & \oplus/ \cdot (\oplus/) * \cdot f ** \cdot g* \\
= & \quad \{ \text{ map distribution } \} \\
 & \oplus/ \cdot (\oplus/ \cdot f * \cdot g)*
\end{array}
$$

Directed Reductions

We introduce two more computation patterns $\not\to$ (pronounced left-to-right reduce) and $\not\leftarrow$ (right-to-left reduce) which are closely related to $/$. Informally, we have

$$\begin{array}{rcl} \oplus \not\to_e [a_1, a_2, \ldots, a_n] & = & ((e \oplus a_1) \oplus \cdots) \oplus a_n \\ \oplus \not\leftarrow_e [a_1, a_2, \ldots, a_n] & = & a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e)) \end{array}$$

Formally, we can define $\oplus \not\to_e$ on lists by two equations.

$$\begin{array}{rcl} \oplus \not\to_e [\,] & = & e \\ \oplus \not\to_e (x +\!\!+ [a]) & = & (\oplus \not\to_e x) \oplus a \end{array}$$

Exercise: Give a formal definition for $\oplus \not\leftarrow_e$.

Directed Reductions without Seeds

$$
\begin{array}{rcl}
\oplus \not\rightarrow [a_1, a_2, \ldots, a_n] & = & ((a_1 \oplus a_2) \oplus \cdots) \oplus a_n \\
\oplus \not\leftarrow [a_1, a_2, \ldots, a_n] & = & a_1 \oplus (a_2 \oplus \cdots \oplus (a_{n-1} \oplus a_n))
\end{array}
$$

Properties:

$$
\begin{array}{rcl}
(\oplus \not\rightarrow) \cdot ([a] +\!\!+) & = & \oplus \not\rightarrow_a \\
(\oplus \not\leftarrow) \cdot (+\!\!+ [a]) & = & \oplus \not\leftarrow_a
\end{array}
$$

An Example Use of Left-Reduce

Consider the right-hand side of Horner's rule:

$$(((1 \times a_1 + 1) \times a_2 + 1) \times \cdots + 1) \times a_n + 1$$

This expression can be written using a left-reduce:

$$\odot \not\rightarrow_1 [a_1, a_2, \ldots, a_n]$$
$$\text{where } a \odot b = (a \times b) + 1$$

Exercise: Give the definition of $\ominus$ such that the following holds.

$$\ominus \not\rightarrow [a_1, a_2, \ldots, a_n] = (((a_1 \times a_2 + a_2) \times a_3 + a_3) \times \cdots + a_{n-1}) \times a_n + a_n$$

Accumulations

With each form of directed reduction over lists there corresponds a form of computation called an accumulation. These forms are expressed with the operators $\not\!\!\!\!/\!\!\!\rightarrow$ (pronounced left-accumualte) and $\leftarrow\!\!\!\!\!\not\!\!\!\!/$ (right-accumulate) and are defined informally by

$$
\begin{array}{lcl}
\oplus \not\!\!\!\!/\!\!\!\rightarrow_e [a_1, a_2, \ldots, a_n] & = & [e, e \oplus a_1, \ldots, ((e \oplus a_1) \oplus) \cdots \oplus a_n] \\
\oplus \leftarrow\!\!\!\!\!\not\!\!\!\!/_e [a_1, a_2, \ldots, a_n] & = & [a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e)), \ldots, a_n \oplus e, e]
\end{array}
$$

Formally, we can define $\oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e$ on lists by two equations by

$$\begin{array}{lcl} \oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e [\,] & = & [e] \\ \oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e ([a] +\!\!\!+\, x) & = & [e] +\!\!\!+ (\oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_{e \oplus a} x), \end{array}$$

or

$$\begin{array}{lcl} \oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e [\,] & = & [e] \\ \oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e (x +\!\!\!+ [a]) & = & (\oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e x) +\!\!\!+ [b \oplus a] \\ & & \text{where } b = \mathit{last}(\oplus\!\!\not\!\!\!\!/\!\!\!\rightarrow_e x). \end{array}$$

Efficiency in Accumulate

$\oplus\not\!\!\!\!\!/\!\!\!\rightarrow_e [a_1, a_2, \ldots, a_n]$: can be evaluated with $n-1$ calculations of $\oplus$.

Exercise: Consider computation of first $n+1$ factorial numbers: $[0!, 1!, \ldots, n!]$. How many calculations of $\times$ are required for the following two programs?

1. $\times\not\!\!\!\!\!/\!\!\!\rightarrow_1 [1, 2, \ldots, n]$
2. $\mathit{fact} * [0, 1, 2, \cdots, n]$ where $\mathit{fact}\ 0 = 1$ and $\mathit{fact}\ k = 1 \times 2 \times \cdots \times k$.

Relation between Reduce and Accumulate

$$
\begin{array}{l}
\oplus \not\rightarrow_e = last \cdot \oplus \not\!\!\!/\!\!\!\rightarrow_e \\
\oplus \not\!\!\!/\!\!\!\rightarrow_e = \otimes \not\rightarrow_{[e]} \\
\qquad \text{where } x \otimes a = x +\!\!\!+ [last\ x \oplus a]
\end{array}
$$

Segments

A list y is a segment of x if there exists u and v such that

$$x = u +\!\!+ y +\!\!+ v.$$

If $u = []$, then y is called an initial segment.
If $v = []$, then y is called an final segment.

An Example:

$$segs\ [1, 2, 3] = [[], [1], [1, 2], [2], [1, 2, 3], [2, 3], [3]]$$

Exercise: How many segments for a list $[a_1, a_2, \ldots, a_n]$?

inits

The function inits returns the list of initial segments of a list, in increasing order of a list.

$$inits\ [a_1, a_2, \ldots, a_n] = [[\,], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]]$$

$$inits = (+\!\!+ \not\!\!\!\!\rightarrow\!\!\!\!/_{[]}) \cdot [\cdot]*$$

tails

The function tails returns the list of final segments of a list, in decreasing order of a list.

$$tails\ [a_1, a_2, \ldots, a_n] = [[a_1, a_2, \ldots, a_n], [a_2, \ldots, a_n], \ldots, [a_n], [\,]]$$

$$tails = (+\!\!+ \not\leftarrow\!\!\!/\!\!/_{[]}) \cdot [\cdot]*$$

segs

$$segs = +\!\!+ / \cdot tails * \cdot inits$$

Exercise: Show the result of $segs\ [1, 2]$.

Accumulation Lemma

$$(\oplus\not\!\!\!\!/\!\!\rightarrow_e) = (\oplus \not\!\rightarrow_e) * \cdot inits$$
$$(\oplus\not\!\!\!\!/\!\!\rightarrow) = (\oplus \not\!\rightarrow) * \cdot inits^+$$

The accumulation lemma is used frequently in the derivation of efficient algorithms for problems about segments.

> *On lists of length n, evaluation of the LHS requires $O(n)$ computations involving $\oplus$, while the RHS requires $O(n^2)$ computations.*

The Problem: Revisit

Consider the following simple identity:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1$$

This equation generalizes in the obvious way to n variables $a_1, a_2, \ldots, a_2$, and we will refer to it as Horner'e rule.

- Can we generalize $\times$ to $\otimes$, $+$ to $\oplus$? What are the essential constraints for $\otimes$ and $\oplus$?
- Do you have suitable notation for expressing the Horner's rule concisely?

Horner's Rule

The following equation

$$\oplus/ \cdot \otimes/* \cdot tails = \odot \not\rightarrow_e$$

$$\begin{array}{l} \text{where} \\ \quad e = id_{\otimes} \\ \quad a \odot b = (a \otimes b) \oplus e \end{array}$$

holds, provided that $\otimes$ distributes (backwards) over $\oplus$:

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

for all a, b, and c.

Exercise: Prove the correctness of the Horner's rule.

Hints:

- Show that

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

  is equivalent to

$$(\otimes c) \cdot \oplus/ = \oplus/ \cdot (\otimes c) * .$$

- Show that

$$f = \oplus/ \cdot \otimes/ * \cdot tails$$

  satisfies the equations

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

Generalizations of Horner's Rule

Generalization 1:

$$\oplus/ \cdot \otimes/ * \cdot tails^{+} = \odot \not\rightarrow$$

where

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

Generalizations of Horner's Rule

Generalization 1:

$$\oplus/ \cdot \otimes/ * \cdot tails^{+} = \odot \not\rightarrow$$

where

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

Generalization 2:

$$\oplus/ \cdot (\otimes/ \cdot f*) * \cdot tails = \odot \not\rightarrow_{e}$$

where

$$e = id_{\otimes}$$

$$a \odot b = (a \otimes f\ b) \oplus e$$

The Maximum Segment Sum (mss) Problem

Compute the maximum of the sums of all segments of a given sequence of numbers, positive, negative, or zero.

$$mss\ [3, 1, -4, 1, 5, -9, 2] = 6$$

A Direct Solution

$$mss = \uparrow / \cdot +/ * \cdot segs$$

Calculating a Linear Algorithm using Horner's Rule

$$
\begin{array}{ll}
 & mss \\
= & \quad \{ \text{ definition of } mss \ \} \\
 & \uparrow / \cdot +/ * \cdot segs \\
= & \quad \{ \text{ definition of } segs \ \} \\
 & \uparrow / \cdot +/ * \cdot +\!\!+ / \cdot tails * \cdot inits \\
= & \quad \{ \text{ map and reduce promotion } \} \\
 & \uparrow / \cdot (\uparrow / \cdot +/ * \cdot tails) * \cdot inits \\
= & \quad \{ \text{ Horner's rule with } a \odot b = (a+b) \uparrow 0 \ \} \\
 & \uparrow / \cdot \odot \not\to_0 * \cdot inits \\
= & \quad \{ \text{ accumulation lemma } \} \\
 & \uparrow / \cdot \odot /\!\!/\!\!\to_0
\end{array}
$$

A Program in Haskell

Exercise: Code the derived linear algorithm for *mss* in your favorite programming language.

Segment Decomposition

The sequence of calculation steps given in the derivation of the *mss* problem arises frequently. The essential idea can be summarized as a general theorem.

Theorem (Segment Decomposition)

Suppose S and T are defined by

$$
\begin{aligned}
S &= \oplus/ \cdot f * \cdot segs \\
T &= \oplus/ \cdot f * \cdot tails
\end{aligned}
$$

If T can be expressed in the form $T = h \cdot \odot \not\rightarrow_e$, then we have

$$S = \oplus/ \cdot h * \cdot \odot /\!\!\!/\!\!\!\rightarrow_e$$

Homomorphism

Zhenjiang Hu

National Institute of Informatics

June 21 / June 28 / July 5, 2010

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

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

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.

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

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.

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

- *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 $([\alpha]^+, \mathbin{+\!\!+})$.

- *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

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

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.

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

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:

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

Thus $xX_\oplus$is a homomorphism.

Properties

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

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

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

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

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

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

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

Proof Sketch.

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

$$h\ (x +\!\!\!+ y) \quad = \quad h\ x \odot h\ y.$$

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

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

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.

Left Reduction (Foldl)

Zhenjiang Hu

National Institute of Informatics

June 21 / June 28 / July 5, 2010

The Minimax Problem

Given is a list of lists of numbers. Required is an efficient algorithm for computing the minimum of the maximum numbers in each list. More succinctly, we want to compute

$$minimax \;=\; \downarrow / \cdot \uparrow / *$$

as efficiently as possible.

Three Views of Lists

- Monoid View: every list is either
  - (i) the empty list;
  - (ii) a singleton list; or
  - (iii) the concatenation of two (non-empty) lists.
- Snoc View: every list is either
  - (i) the empty list; or
  - (ii) of the form $x +\!\!+ [a]$ for some list x and value a.
- Cons View: every list is either
  - (i) the empty list; or
  - (ii) of the form $[a] +\!\!+ [x]$ for some list x and value a.

Three General Computation Forms

- Monoid View: homomorphism
- Snoc View: left reduction (foldl)

$$\begin{array}{lcl} \oplus \not\rightarrow_e [\,] & = & e \\ \oplus \not\rightarrow_e (x +\!\!+ [a]) & = & (\oplus \not\rightarrow_e x) \oplus a \end{array}$$

- Cons View: right reduction (foldr)

$$\begin{array}{lcl} \oplus \not\leftarrow_e [\,] & = & e \\ \oplus \not\leftarrow_e ([a] +\!\!+ x) & = & a \oplus (\oplus \not\leftarrow_e x) \end{array}$$

Loops and Left Reductions

A left reduction $\oplus \not\rightarrow_e x$ can be translated into the following program in a conventional imperative language.

```
|[ var a;
   a := e;
   for b in x
      do a := a oplus b;
   return a
]|
```

Left Zeros

Left reductions require that the argument list be traversed in its entirety. Such a traversal can be cut short if we recognize the possibility that an operator may have left zeros.

ω is a left zero of $\oplus$ if

$$\omega \oplus a = \omega$$

for all a.

Left Zeros

Left reductions require that the argument list be traversed in its entirety. Such a traversal can be cut short if we recognize the possibility that an operator may have left zeros.

ω is a left zero of $\oplus$ if

$$\omega \oplus a = \omega$$

for all a.

Exercise: Prove that if ω is a zero left of $\oplus$ then

$$\oplus \not\rightarrow_{\omega} x = \omega$$

for all x. (by induction on snoc list x.)

Specialization Lemma

Lemma

Specialization Every homomorphism on lists can be expressed as a left (or also a right) reduction. More precisely,

$$\begin{array}{rcl} \oplus/\cdot f* & = & \odot \not\to_e \\ \textit{where} & & \\ e & = & id_{\oplus} \\ a \odot b & = & a \oplus f\ \ b \end{array}$$

Specialization Lemma

Lemma

Specialization Every homomorphism on lists can be expressed as a left (or also a right) reduction. More precisely,

$$\begin{array}{rcl} \oplus/\cdot f* & = & \odot \not\rightarrow_e \\ \multicolumn{3}{l}{\mathit{where}} \\ e & = & id_\oplus \\ a \odot b & = & a \oplus f\ b \end{array}$$

Exercise: Prove the specialization lemma.

Specialization Lemma

Every homomorphism on lists can be expressed as a left (or also a right) reduction. More precisely,

$$\begin{aligned} \oplus/ \cdot f* &= \odot \not\rightarrow_e \\ \text{where} & \\ e &= id_\oplus \\ a \odot b &= a \oplus f\ b \end{aligned}$$

Exercise: Prove the specialization lemma.

Minimax

Let us return to the problem of computing

$$minimax \;=\; \downarrow/ \cdot \uparrow/*$$

efficiently. Using the specialization lemma, we can write

$$minimax \;=\; \odot \not\to_{\infty}$$

where ∞ is the identity element of $\downarrow/$, and

$$a \odot x \;=\; a \downarrow (\uparrow/x)$$

Since $\downarrow$ distributes through $\uparrow$ we have

$$a \odot x \;=\; \uparrow / ((a \downarrow) * x)$$

Using the specialization lemma a second time, we have

$$\begin{array}{rcl} a \odot x & = & \oplus_a \not\rightarrow_{-\infty} x \\ \text{where } b \oplus_a c & = & b \uparrow (a \downarrow c) \end{array}$$

Since $\downarrow$ distributes through $\uparrow$ we have

$$a \odot x \;=\; \uparrow / ((a \downarrow) * x)$$

Using the specialization lemma a second time, we have

$$\begin{array}{rcl} a \odot x & = & \oplus_a \not\rightarrow_{-\infty} x \\ \text{where } b \oplus_a c & = & b \uparrow (a \downarrow c) \end{array}$$

Exercise: What are left zeros for $\oplus_a$ and $\odot$?

An Efficient Implementation of `minimax xs`

```
|[ var a; a := infinity;
   for x in xs while a <> \infinity
      do a := a odot x;
   return a
]|
```

where the assignment `a := a odot x` can be implemented by the loop:

```
|[ var b; b := -infinity;
   for c in x while c <> a
      do b := b max (a min c);
   a := b
]|
```