Bird Meertens Formalism

– Directed Reduction (Foldl) –

Zhenjiang Hu, Wei Zhang

Department of Computer Science and Technology
Peking University

December 22, 2021

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.

Specialization Lemma

Lemma (Specialization)

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

$$\oplus/ \cdot f* \;=\; \odot\not\rightarrow_e$$

$$\text{where}$$

$$e \;=\; id_\oplus$$

$$a \odot b \;=\; a \oplus f\ b$$

Specialization Lemma

Lemma (Specialization)

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

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

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

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

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

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

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

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

Using the specialization lemma a second time, we have

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

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 b<a
      do b := b max (a min c);
   a := b
]|
```

Homework BMF 2-3

Code the efficient implementation of *minimax* in Haskell.