Constructive Algorithmics (Part III)

Zhenjiang HU
University of Tokyo

Review

Part I: Basic Concepts

- Notations on Functions
- Algebraic View of Lists
- List functions as Compositions of Homomorphisms
- Basic Calculation Rules for Derivation of Homomorphisms: Promotion Rules
- Horner's Rule
- Maximum Segment Sum Problem

Part II: Homomorphisms

- Formalization of Homomorphism: Reduce after Map
- Program specification with Homomorphisms
- Point-free calculation rules for manipulating homomorphisms
- Longest All-P Segment Problem

Today's Lesson: Left Reductions

- General homomorphic equations and well-defined functions
- Left reduction: a sequential computation pattern
- Loops: implementation of left reduction
- Left-zeros
- The Minimax problem

A 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.

General Equations

So far we have mainly seen examples of homomorphisms. It is instructive to determine the conditions under which a general set of equations

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

determines a unique function h, not necessarily a homomorphism.

Consider the equations

$$
\begin{array}{lcl}
h'\ [\,] & = & ([], e) \\
h'\ [a] & = & ([a], f\ a) \\
h'\ (x \mathbin{+\!\!+} y) & = & h'\ x \oplus h'\ y \\
 & & \textbf{where } (x, u) \oplus (y, v) = (x \mathbin{+\!\!+} y, H(x, y, u, v))
\end{array}
$$

If h' is a well-defined function (a well-defined homomorphism), then so is h, because we have

$$h = \pi_2 \cdot h'$$

What is the condition for h' to be well-defined homomorphism?

Fact: h' defined by equations

$$\begin{array}{lcl} h'\ [\,] & = & ([], e) \\ h'\ [a] & = & ([a], f\ a) \\ h'\ (x \mathbin{+\!\!+} y) & = & h'\ x \oplus h'\ y \\ & & \textbf{where}\ (x, u) \oplus (y, v) = (x \mathbin{+\!\!+} y, H(x, y, u, v)) \end{array}$$

is well-defined if $(R, \oplus, ([], e))$ forms a monoid:

1. $([], e)$ is the unit of $\oplus$;
2. $\oplus$ is associative.

Translating the monoid condition into conditions on e and H gives the following three conditions.

1. $H(x, [], u, e) = u$
2. $H([], y, e, v) = v$
3. $H(x +\!\!+ y, z, H(x, y, u, v), w) = H(x, y +\!\!+ z, u, H(y, z, v, w))$

An Example: Longest All-Even Initial Segment

$$
\begin{aligned}
laei\ [] &= [] \\
laei\ [a] &= \textbf{if } even\ a \textbf{ then } [a] \textbf{ else } [] \\
laei\ (x +\!\!+ y) &= \textbf{if } laei\ x = x \textbf{ then } laei\ x +\!\!+ laei\ y \textbf{ else } laei\ x
\end{aligned}
$$

In this example,

$$
\begin{aligned}
e &= [] \\
H(x, y, u, v) &= \textbf{if } u = x \textbf{ then } u +\!\!+ v \textbf{ else } u
\end{aligned}
$$

Exercise: Prove that $\forall x.\ \#(laei\ x) \leq \#x$.

Exercise: Prove that *laei* is well-defined.
[Hint: Use the fact that $\#u \leq \#x$ and $\#v \leq \#y$.]

Lemma. *laei* is not a homomorphism

Proof. Suppose

$$laei(x ++ y) = laei\ x \oplus laei\ y$$

for some operator $\oplus$. Since $laei[2, 1] = 2$, $laei[4] = [4]$ and $laei[2] = [2]$, we have

$$\begin{aligned} laei\ [2, 1, 4] &= laei\ [2, 1] \oplus laei\ [4] \\ &= [2] \oplus [4] \\ &= laei\ [2] \oplus laei\ [4] \\ &= laei\ [2, 4] \end{aligned}$$

This is a contridition, since $laei\ [2, 1, 4] = [2]$ and $laei\ [2, 4] = [2, 4]$.

Left Reduction

$$\oplus \not\rightarrow_e [x_1, x_2, \ldots, x_n] = (((e \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n$$

In the monoid view of lists, the formal definition of $\oplus \not\rightarrow_e$ is as follows.

$$\begin{array}{lcl} \oplus \not\rightarrow_e [] & = & e \\ \oplus \not\rightarrow_e [a] & = & e \oplus a \\ \oplus \not\rightarrow_e (x +\!\!+ y) & = & \oplus \not\rightarrow_{e'} y \textbf{ where } e' = \oplus \not\rightarrow_e x \end{array}$$

Exercise: Prove that $\oplus \not\rightarrow_e$ is a well-defined function.

There is an instructive alternative way of seeing that $\oplus\!\not\to_e$ is well-defined. Define h by

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

Obviously, h is a homomorphism from $([a], +\!\!+, [])$ to $(\beta \to \beta, \cdot, id_\beta)$. Now we have

$$\oplus\!\not\to_e x = h\ x\ e$$

and so $\oplus\!\not\to_e$ is well-defined.

Left Reduction is Important

Every set of equations of the following form

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

can be defined in terms of a left reduction:

$$f = \pi_2 \cdot \oplus \not\rightarrow_{e'}$$

where

$$\begin{array}{lcl} e' & = & ([], e) \\ (x, u) \oplus a & = & (x +\!\!+ [a], F(a, x, u)) \end{array}$$

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

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

- *Cons View*: right reduction

Exercise: Give the definition for right reduction.

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 r;
   r := e;
   for b in x
      do r := r oplus b;
   return r
]|
```

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 left-zero of $\oplus$ then

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

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

Implementation of Left Reduction with Left-zero Check

From the fact that $\oplus \not\rightarrow_e (x +\!\!+\, y) = \oplus \not\rightarrow_{(\oplus \not\rightarrow_e x)}\, y$, we have the following program for left-reduction.

```
|[ var r;
   r := e;
   for b in x while not left-zero(r)
      do r := r oplus b;
   return r
]|
```

Specialization Lemma

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

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

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

$$a \odot x \;=\; \oplus_a \not\to_{-\infty} x$$
$$\text{where } b \oplus_a c \;=\; b \uparrow (a \downarrow c)$$

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
]|
```

The alpha-beta Algorithm

We now generalize the minimax problem to trees. Consider the tree data type defined by

$$\begin{array}{lcl} \mathit{Tree} & ::= & \mathit{Tip}\ \mathit{Int} \\ & \mid & \mathit{Fork}\ [\mathit{Tree}] \end{array}$$

we wish to calculate an efficient algorithm for computing a function

$$\begin{array}{lcl} \mathit{eval} & : & \mathit{Tree} \rightarrow \mathit{Int} \\ \mathit{eval}\ (\mathit{Tip}\ n) & = & n \\ \mathit{eval}\ (\mathit{Fork}\ ts) & = & \uparrow/(-\mathit{eval}) * ts \end{array}$$

Exercise: Calculate the value of the following expression.

$$\mathit{eval}\ (\mathit{Fork}\ [\mathit{Fork}\ [\mathit{Fork}\ [\mathit{Tip}\ 3, \mathit{Tip}\ 1, \mathit{Tip}\ 4], \mathit{Tip}\ 1], \mathit{Fork}\ [\mathit{Tip}\ 5, \mathit{Tip}\ 9]])$$

Homework

Exercise: Derive an efficient algorithm for computing *eval*.

Reference: Richard Bird and Jone Hughes, The alpha-beta algorithm: an exercise in program transformation. *Information Processing Letters*, Vol.24 (1987). 53–57.