A Short Introduction to Constructive Algorithmics

Kiminori MATSUZAKI

(Kochi University of Technology)
Constructive Algorithmics

- Constructive Algorithmics:
  “Methods for calculating programs from their specifications, and the design of notations for such calculation; also, the investigation of software support for the calculational process.”

- [http://www.cs.auckland.ac.nz/research/groups/CDMTCS/docs/ca.html](http://www.cs.auckland.ac.nz/research/groups/CDMTCS/docs/ca.html)
Outline

- Notations
  - BMF (Bird-Meertens Formalism)
  - Lists and their operations

- Program Calculation (Program Transformation)
  - Homomorphisms
  - Transformation rules
  - Calculation Example

- Extended Topics
Notations
Functions

- Function application is written without brackets
  - $f \ a$ means $f(a)$

- Function are curried and associate to the left
  - $f \ a \ b$ means $(f \ a) \ b$

- Function composition
  - $(f \cdot g) \ x = f(g \ x)$

- Binary operators can be sectioned
  - $(\oplus) \ a \ b = (a \oplus) \ b = (\oplus b) \ a = a \oplus b$
List

- Lists are finite sequence of values of the same type.

- Any list can be constructed with the following two operations:
  - Nil: An empty list
    ✓ Denoted by \[ \]
  - Cons: Put an element before a list
    ✓ Denoted by \( a : as \)

- Example: \([1, 2, 3] = 1 : (2 : (3 : [ ]))\)

- We may call such a list as “cons list”
Join List

- Another way to construct a list

- We use three operations to construct a list
  - Nil: An empty list
    - Denoted by \[ \[] \]
  - Singleton: A list with a single element
    - Denoted by \([a]\) or \([.]\ a\)
  - Join: Concatenation of two smaller lists
    - Denoted by \(x ++ y\)
    - Concatenation is associative
Map

- Map (denoted by * ) takes a function and a list, and applies the function to each element of the list.

- Informally:

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

• Reduce (denoted by / ) takes an associative operator and a list, and collapse the list into a value with the operator.

• Informally:

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

- Left-to-right reduce (denoted by $\rightarrow$) is a reduce-like operation, but it can take non-associative operator.

- Informally:

$$\oplus \rightarrow_e [a_1, a_2, \ldots, a_n] = ((e \oplus a_1) \oplus \cdots \oplus a_n$$

- We can consider right-to-left reduce $\leftarrow$
Accumulation (Scan)

- Left-accumulate (denoted by $\nLeftAccumulate$) takes an operator and a list, and computes on the list by accumulating the values with the operator.

- Informally:

$$\nLeftAccumulate_e[a_1, a_2, \ldots, a_n] = [e, e \oplus a, \ldots, ((e \oplus a_1) \oplus \cdots \oplus a_n]$$
Program Calculation
Idea of Program Calculation

- Starting from a (mathematical) specification, we derive a program by applying transformation rules:
  - Concise and clear specification
  - Transformation rules proved to be correct
  - Systematic / mechanical / automatic derivation

```
P0  ⇔  P1  ⇔  ...  ⇔  Pk
```

**Specification**
- Concise
- Clear
- Correct

**Transformation Rules**

**Derived Program**
- Efficient
- Correct
Homomorphism

- A function $h$ defined in the following form is called homomorphism.

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

- It maps a monoid $([\alpha], \++, [\,])$ to $(\beta, \oplus, id_\oplus)$

- Property: $h$ is uniquely determined by $f$ and $\oplus$
Homomorphism Theorems

- First homomorphism theorem:
  - Every homomorphism can be written as the composition of a map and a reduce.
  - \( \text{hom} (\oplus) \ f = (\oplus/) \ . \ (f^*) \)

- Third homomorphism theorem:
  - If a function \( h \) can be defined as
    \[
    \begin{align*}
    h ([a] ++ x) &= a \oplus h x \\
    h (y ++ [b]) &= h y \otimes b
    \end{align*}
    \]
    then, \( h \) is a homomorphism.

  - This theorem also works as a parallelization theorem.
    [Morita et al., PLDI 2007]
Rules (1)

- Horner’s rule:
  - E.g. \((a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1\)
  - \(\oplus / \cdot \otimes / \ast \cdot \text{tails} = \ominus \rightarrow_e \)
  - Distributivity over \(\otimes\) and \(\oplus\) is important

- Promotion:
  - \(f \ast \cdot \ast / = \ast / \cdot (f\ast)\ast\)
  - \(\oplus / \cdot \ast / = \oplus / . (\oplus /)\ast\)
Rules (2)

\[ \text{init} [a_1, a_2, \ldots, a_n] = [[], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]] \]

- Accumulation:

\[ (\oplus \not\rightarrow_e) = (\oplus \not\rightarrow_e) \ast \text{init} \]
Example (1)

- The maximum segment sum problem:
  Compute the maximum of sums of all segments of a given sequence of numbers

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

- Concise and clear solution (specification)

  \[ mss = \uparrow / \cdot + / \ast \cdot \text{segs} \]

  - But, slow: \( O(n^3) \)
Example (2)

- Calculating a linear-time algorithm

\[
\uparrow / \cdot + / \ast \ast \text{segs} \\
= \{ \text{definition of segs} \} \\
\uparrow / \cdot + / \ast \ast \ast \text{tails} \ast \ast \text{inits} \\
= \{ \text{map and reduce promotion} \} \\
\uparrow / \cdot (\uparrow / \cdot + / \ast \ast \text{tails}) \ast \ast \text{inits} \\
= \{ \text{Horner’s rule with } a \odot b = (a + b) \uparrow 0 \} \\
\uparrow / \cdot \odot \odot \ast \ast \text{inits} \\
= \{ \text{accumulation lemma} \} \\
\uparrow / \cdot \odot \odot \odot 0
\]
Extended Topics
List → Other data structures

- Matrix (2D array)
  - Data is constructed with Singleton, JoinX, JoinY
  - Some property is needed on JoinX/JoinY

- Binary tree
  - Data is constructed with Leaf and Node
    Node :: a -> (Tree a) -> (Tree a) → Tree a

- We can apply generic-programming technique (category-theoretic approach) to formalize other data structures (e.g., rose trees)
Relations

- Extend functions into relations
  - Studied by the group of Richard Bird (Oxford Univ.)
- Formalization of non-deterministic computations
- Example: Thinning theorem
  - Formalization of optimization problems (dynamic programming / approximation)
  - [Mu et al., WGP 2010.]
Optimization Problems

- Derive a dynamic-programming algorithm from the problem specification
  - Use algebraic properties of operators

- Examples
  - Maximum marking problem (Extension of mss)
    - Sasano et al. ICFP 2000
  - Generic maximum marking problem
    - Emoto et al. ESOP 2012
Parallel Programming

- Use of patterns as algorithmic skeletons
  - SkeTo parallel skeleton library
    - Data structures: lists, matrices, and trees
    - Skeletons: map, reduce, scan (and others)
- Third homomorphism theorem for parallelization
  - A tool for parallelization [Morita et al. PLDI 2007]
  - Parallelization for tree problems [Morihata et al. POPL 2009]
Tools

- Fusion optimization
  - Strong optimization in Haskell
  - Warm fusion: a powerful fusion rule based on list homomorphism
  - Stream fusion: a structure-generic fusion mechanism

- Yicho system
  - A tool for program calculation
  - Hylomorphism (extension of homomorphism)
  - [Yokoyama et al. APLAS 2002]
Conclusion

3 Important Aspects of Constructive Algorithmics

- Notations and patterns
  - Bird-Meertens Formalism (BMF)
  - Functional modeling of

- Program Calculations
  - Systematically deriving programs from the specification
  - Theorems for homomorphisms

- Supporting Tools
  - For systematic/automatic derivation of programs
  - Domain-specific tools for program derivation
References

- Lecture notes about constructive algorithmics by Zhenjiang Hu (NII Japan)
  - http://research.nii.ac.jp/~hu/pub/teach/pm06/CA[1-4].pdf