Chapter 16: Introduction to Calculational Programming

Zhenjiang Hu, Wei Zhang

School of Computer Science
Peking University

November 12, 2025

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

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

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

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

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

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.

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

Specifying Algorithms by Functions (3/3)

Exercise

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.

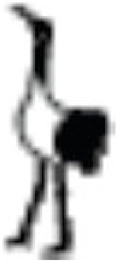
In this course, we consider functional specification.

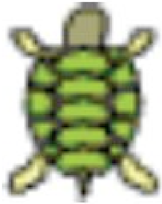
Outline

Tsuru-Kame-Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 6 heads and 20 legs. Find out the numbers of cranes and tortoises.





A Kindergarten Approach

A Kindergarten Approach

- A simple enumeration

A Kindergarten Approach

- A simple enumeration

Primary School

Primary School

- Reasoning

Primary School

- Reasoning

  if all 6 animals were cranes, there ought to be $6 \times 2 = 12$ legs.

Primary School

- Reasoning

  if all 6 animals were cranes, there ought to be $6 \times 2 = 12$ legs.

  However, there are in fact 20 legs, the extra $20 - 12 = 8$ legs must belong to some tortoises.

Primary School

- Reasoning

  if all 6 animals were cranes, there ought to be $6 \times 2 = 12$ legs.

  However, there are in fact 20 legs, the extra $20 - 12 = 8$ legs must belong to some tortoises.

  Since one tortoise can add 2 legs, we have $8/2 = 4$ tortoises.

Primary School

- Reasoning

  if all 6 animals were cranes, there ought to be $6 \times 2 = 12$ legs.

  However, there are in fact 20 legs, the extra $20 - 12 = 8$ legs must belong to some tortoises.

  Since one tortoise can add 2 legs, we have $8/2 = 4$ tortoises.

  So there must be $6 - 4 = 2$ cranes.

Middle School

Middle School

- Algebra (Equation Theory)

Middle School

- Algebra (Equation Theory)

$$\begin{aligned} x + y &= 6 \\ 2x + 4y &= 20 \end{aligned}$$

Middle School

- Algebra (Equation Theory)

$$\begin{aligned} x + y &= 6 \\ 2x + 4y &= 20 \end{aligned}$$

which gives

$$\begin{aligned} x &= 2 \\ y &= 4 \end{aligned}$$

- The same problem may have different difficulties depending on what weapons we have in hand.

Many arithmetic problems can be easily solved if we use the equation theory.

- The same problem may have different difficulties depending on what weapons we have in hand.

  Many arithmetic problems can be easily solved if we use the equation theory.

- What are weapons for solving programming problems? Do we have an "equation theory" for constructing correct and efficient programs?

- The same problem may have different difficulties depending on what weapons we have in hand.

Many arithmetic problems can be easily solved
if we use the equation theory.

- What are weapons for solving programming problems? Do we have an "equation theory" for constructing correct and efficient programs?

$$\Downarrow$$

Calculational Programming

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?
- Can we reuse the derivation procedure to solve similar problems, say maximum increasing segment sum problme?

Outline

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.

Specification: Clean and Correct programs

$\Downarrow$

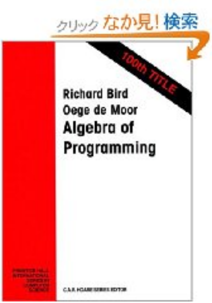
Folding/Unfolding Program Transformation

$\Downarrow$

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.

Specification: Clean and Correct programs

$\Downarrow$

$\boxed{\text{Folding-free Program Transformation}}$

$\Downarrow$

Efficient Programs

Work on Program Calculation

- Algorithm Derivation
  - Fold/Unfold-based Transformational Programming (Darlington&Burstall:77)
  - Bird-Meertens Formalism (BMF) (Bird:87)
  - Algebra of Programming (Bird&de Moor:96)

Work on Program Calculation

- Algorithm Derivation
  - Fold/Unfold-based Transformational Programming (Darlington&Burstall:77)
  - Bird-Meertens Formalism (BMF) (Bird:87)
  - Algebra of Programming (Bird&de Moor:96)
- Our Work on Program Transformation in Calculation Form
  - Fusion (ICFP'96)
  - Tupling (ICFP'97)
  - Accumulation (NGC'99)
  - Inversion/Bidirectionalization (MPC'04, PEPM'07, ICFP'07, MPC'10, ICFP'10)
  - Dynamic Programming (ICFP'00, ICFP'03, ICFP'08)
  - Parallelization (POPL'98, ESOP'02, PLDI'07, POPL'09, ESOP'12)

What I will talk in this course?

- Algorithm Derivation
  - Fold/Unfold-based Transformational Programming (Darlington&Burstall:77)
  - Bird-Meertens Formalism (BMF) (Bird:87)
  - Algebra of Programming (Bird&de Moor:96)
- Our Work on Program Transformation in Calculation Form
  - Fusion (ICFP'96)
  - Tupling (ICFP'97)
  - Accumulation (NGC'99)
  - Inversion/Bidirectionalization (MPC'04, PEPM'07, ICFP'07, MPC'10, ICFP'10)
  - Dynamic Programming (ICFP'00, ICFP'03, ICFP'08)
  - Parallelization (POPL'98, ESOP'02, PLDI'07, POPL'09, ESOP'12)

What I will talk in this course?

- Algorithm Derivation
  - Fold/Unfold-based Transformational Programming (Darlington&Burstall:77)
  - Bird-Meertens Formalism (BMF) (Bird:87)
  - Algebra of Programming (Bird&de Moor:96)
- Our Work on Program Transformation in Calculation Form
  - Fusion (ICFP'96)
  - Tupling (ICFP'97)
  - Accumulation (NGC'99)
  - Inversion/Bidirectionalization (MPC'04, PEPM'07, ICFP'07, MPC'10, ICFP'10)
  - Dynamic Programming (ICFP'00, ICFP'03, ICFP'08)
  - Parallelization (POPL'98, ESOP'02, PLDI'07, POPL'09, ESOP'12)

$\Uparrow$

Functional Programming

(basic concepts of algorithmic languages, program specification and reasoning)

Plan

1. Tool for Calculation: Agda (about 3 lectures)
    - Learn functional programming in Agda
    - Learn program reasoning in Agda
2. Program Calculus: BMF (about 4 lectures)
    - Learn basic programming theory for calculating programs from problem specifications
    - Learn basic techniques for calculating programs
3. Applications of Calculational Programming (about 1 lectures)
    - Learn how to solve a wide class of optimization problems
    - Learn how to automatic parallelize sequential programs

References

- Aaron Stump, *Verified Functional Programming in Agda*. ACM Book, 2016.
- Ulf Norell, *Dependently Typed Programming in Agda*. Advanced Functional Programming 2008: 230-266.
- Richard Bird, *Lecture Notes on Constructive Functional Programming*, Technical Monograph PRG-69, Oxford University, 1988.
- Richard Bird and Oege de Moor, *The Algebra of Programming*, Prentice-Hall, 1996.
- Roland Backhouse, *Program Construction: Calculating Implementation from Specification*, Wiley, 2003.

Homework

16-1 Write a Haskell program to solve the maximum segment sum problem, following the three steps in the slides.

16-2 Write a Haskell program to solve the maximum segment sum problem, using the smart algorithm in the slides.