Higher-Order Abstraction of Process Compositions and Their Transformation

by

Norihiko YOSHIDA *

This paper presents a higher-order abstraction framework for process compositions which are common to highly-parallel systems and VLSI architectures. It is to aid reuse and formal design of such compositions. We construct the framework on functional programming, since it has a facility for process modeling and higher-order abstraction. Our achievements include a basis for inheritance and aggregation relationship in a collection of compositions, and analysis and design of transformation rules for deriving compositions.

1. Introduction

Our research is aiming at building a design framework for highly-parallel systems, in particular VLSI hardware algorithms. This paper presents a framework for abstraction of system compositions. Abstraction is crucial to aid system design. There are already abstraction schemes for components using objects or processes, but no scheme for composition topologies or configurations yet. So, for example, we can only build a library of components, but not of compositions yet. Abstraction of compositions enables us to build a library for compositions as meta-components. In this point of view, our motivation is similar to the one for design patterns or frameworks introduced recently in the area of object-oriented software design.

In our previous research, we made some achievements regarding formal design of VLSI architectures from their mathematical specifications using transformational derivation techniques which had been originally introduced for software design[1,2]. Abstraction of compositions will help greatly such formal design techniques by enabling us abstraction of transformation rules. This is our another motivation.

Strictly speaking, abstraction of a process composition yields a higher-order abstract process. It is abstract that the composition is encapsulated, and it is higher-order that concrete component processes of the composition are of no concern. For example, a pipeline composition of processes is defined as a higher-order abstract process of just the pipeline topology. With a set of concrete processes, it is instantiated as a concrete pipeline (We will show this details later.)

Here we concern ourselves with process compositions common to many parallel systems and hardware algorithms, which are repetitive, synchronous and deterministic. Partly based on our preliminary work[3], we investigate the ability and potential of a framework for higher-order abstraction through formalizing some basic compositions such as pipeline, parallel and tree processes. We construct the framework on functional programming, since functional programming has a facility for process modeling and higher-order abstraction. And more than that, we also examine inheritance and aggregation relationship in a collection of compositions represented in our higher-order abstraction framework, since they are important when we consider libraries of compositions.

Section 2 briefly overviews functional programming and some related works as the background. Section 3 introduces some functions to represent processes, and also examine an inheritance-like hierarchy among them. Section 4 formalizes some process compositions, and then introduces higher-order functions for them. This section also examines their aggregation. Section 5 discusses transformation of compositions represented in higher-order functions. Section 6 contains some concluding remarks.

Received on October, 1998

**Department of Computer and Information Sciences

2. Background

2.1 Functional Programming

We write programs in this paper using a functional programming language Gofer. It has a syntax and semantics quite common to most functional programming languages such as Miranda and Haskell. More details of functional programming are left to the reference[4].

A function call is described as "`f x y`", against the mathematical convention to describe it as "f (x, y)". In fact, "`f x y`" may be interpreted also as a function "`f x`" applied to an argument "y". A function's argument and/or return value may be another function, in which case the function is called higher-order. A function definition is described as "`f x y = x + y`".

One of the basic data structures is a list. The empty list is "`[]`", a list of 1, 2 and 3 is "`[1,2,3]`", and CONS (a Lisp term) of 1 and "`[2,3]`" is "`1:[2,3]`", which yields "`[1,2,3]`". It is important that a list may be infinite, namely without an end, and such an infinite list is interpreted as a data stream fed to a process. We describe a list of "x"es by "`xl`" and a stream of "x"es by "`xs`" so as to distinguish them which are syntactically identical. Another basic data structure is a tuple. A tuple of 1, 2 and 3 is "`(1,2,3)`". Unlike a list, the size of a tuple is fixed. "`()`" is also used to manage the order of function application.

2.2 Related Works

While there are many researches on parallel execution of functional programs including ours[5], researches on modeling of parallel systems in functional programming were originated rather recently, and are rapidly attracting attentions. They were derived from researches on formal design of concurrent systems and data-flow computing. Some prominent works are summarized below.

Kelly formalized and defined a couple of basic compositions including "`pipeline`"[6]:

```
pipeline fl xs =
  (foldr (.) id (map map fl)) xs
```

where "`(.)`", "`id`", "`map`" and "`foldr`" are system functions defined as :

```
(f . g) x = f (g x)
id x = x
map f [] = []
map f (x : xs) = f x : map f xs
foldr f z [] = z
foldr f z (x : xs) =
  f x (foldr f z xs)
```

He was aware that a process could be represented using "`map`", but did not formalize a function for a process. We will define a process function based on "`map`" later in this paper.

Darlington, partly together with Kelly, extended this Kelly's work, and are now building a composition framework named Skeletons[7]. Their research somewhat shares motivations and approaches with ours. They defined principal skeletons for pipeline and parallel compositions as :

```
PIPE = foldr1 (.)
FARM f env = map . (f env)
foldr1 f [x] = x
foldr1 f (x : xs) = f x (foldr1 f xs)
```

where "`foldr1`" is a system function. "`PIPE`" is almost identical to Kelly's original. "`FARM`" is the skeleton for a simple parallel composition in which a single function "`f`" with a parameter "`env`" is applied to all the component processes in the composition. We will define a little more general one later.

Regarding design support for parallel systems and VLSI, the Ruby project around Oxford University applied functional programming to VLSI design, in particular at the gate-transfer level[8]. But this research is not concerned much with higher-order abstraction of compositions.

3. A Perpetual Process

3.1 Definition

A perpetual process is a fixed point of state transition. A process "`process`" applying a function "`f`" on its stream argument "`xs`" is defined to be a recursive function[9,10] as:

```
process (x : xs) = f x : process xs
```

In fact, this "`process`" is to map "`f`" over every item in "`xs`", and thus a process function "`proc`" is defined to be equivalent to "`map`" :

```
proc = map
```

For example, a process which squares every integers in its input stream is described as :

```
sq x = x * x
p0 = proc sq
p0 [1, 2, 3, ..] == [1, 4, 9, ..]
```

This "`p0`" is an instantiation of the abstract process "`proc`" with a function "`sq`".

3.2 Consideration for Inheritance

In our higher-order abstraction framework, we can build inheritance-like hierarchies among compositions, which could not be formalized before. We show this using an example.

Processes are classified according to arities : 1-in 1-out process, 1-in 2-out process, 2-in 1-out process, 2-in 2-out process and so on. An M-in N-out process "procMN" in general has M input streams and N output streams. They all can be defined as subclass-like functions of the generic process "proc". Before going further, we should introduce some assistant operators and functions :

```
(f |/ g) x y = f x (g y)
(f /| g) x y = f (g x y)
zip2 (s1, s2) = zip s1 s2
unzip2 = unzip
```

where "zip" and "unzip" are system functions. This "zip2" converts a tuple of lists into a list of tuples, while "unzip" does inversely. Now we define "procMN" for M = 1 . . . 2 and N = 1 . . . 2 respectively as :

```
proc11 = super
  where super = proc
proc21 = super |/ zip2
  where super = proc
proc12 = unzip2 /| super
  where super = proc
proc22 = unzip2 /| (super |/ zip2)
  where super = proc
```

Just as in usual inheritance hierarchies, this "proc22" may be defined also as a subclass of "proc21" which in turn is a subclass of "proc":

```
proc22 = unzip2 /| super
  where super = proc21
```

This inheritance-like relationship of compositions can be represented only in higher-order abstraction. On the base level, namely without abstraction, we may describe "procMN" just as :

```
proc11 f (x : xs) =
  f x : proc11 f xs
proc21 f (x : xs, y : ys) =
  f (x, y) : proc21 f (xs, ys)
```

where their relations are not easy to understand.

But we do not claim yet this to be true inheritance exactly, since we have not investigated thoroughly its correspondence with type polymorphism, in particular type overloading. This should be of further study.

4. Process Compositions

4.1 Pipeline Processes

A pipeline of two processes "p" and "q" connected by a stream channel is represented as "q (p xs)". So we introduce a pipeline operator ">>":

```
(p >> q) xs = q (p xs)
```

It is important that a pipeline "p >> q" itself is a higher-order process, and we can declare ">>" as right-associative. Hence, we introduce another pipeline function "pipe" for a list of processes, which satisfies :

```
p1 >> .. >> pn == pipe [p1, .., pn]
```

We define this, following Kelly and Darlington's work :

```
pipe = foldr (>>) id
```

For example, a pipeline which computes $out_k = (2in_k + 1)^2$ is described as :

```
p1 = proc ((*) 2)
p2 = proc ((+) 1)
p3 = proc sq
pipe1 = pipe [p1, p2, p3]
pipe1 [1, 2, 3, ..] == [9, 25, 49, ..]
```

This "pipe1" is an instantiation of the abstract pipeline "pipe" with a list of concrete processes "[p1, p2, p3]".

4.2 Parallel Processes

Following the same consideration as above, we introduce a parallel function "para" which represents a parallel composition of processes applying a set of functions (possibly identical in a SIMD style) to a bunch of streams :

```
para [] [] = []
para (p : pl) (xs : xsl) =
  p xs : para pl xsl
```

This is a little more general than Darlington's, since component processes need not be identical. We define this in a more abstract manner :

```
para = curry ((map eval) . zip2)
eval (f, x) = f x
curry f a b = f (a, b)
```

where "curry" is a system function.

4.3 Tree Processes

Pipeline and parallel compositions are the two commonest of parallel systems, but there are some others such as trees and hexagonal arrays, which were not formalized in higher-order abstraction before. A tree composition of processes is for funneling a bunch of

streams. It is mostly used with associative functions like addition or comparison, because consecutive applications of such an function, which takes $O(n)$ time, can be transformed into bi-recursive applications, which takes only $O(\log n)$ time.

For simplicity, here we consider a binary tree composition in which all the component processes are identical. First, we should introduce a split-at-center function "split2" which splits a list at its center :

```
split2 xl = splitAt (length xl / 2) xl
```

where "splitAt" and "length" are system functions. Then we define a binary tree function "tree" which consists of bi-recursion :

```
tree p [xs] = xs
tree p xsl  =
  p (tree p (fst (split2 xsl)),
     tree p (snd (split2 xsl)))
fst (x, y) = x
snd (x, y) = y
```

where "fst" and "snd" are system functions.

4.4 Consideration for Aggregation

Aggregation is whole-part relationship. The word as well as inheritance is from object-oriented programming. A composition is an aggregate of its component processes. It is important that the composition itself is a higher-order abstract process, and can be a component of another composition. This scheme leads to an aggregation hierarchy among compositions represented in our higher-order abstract framework.

For example, a parallel composition of pipeline compositions is described as :

```
[ys1, ys2, ys3] =
  para (copy 3 (pipe [p1, p2, p3]))
       [xs1, xs2, xs3]
```

where "copy" is a system function to make a list of N copies of its second argument.

5. Transformation of compositions

Transformational derivation from mathematical specifications is one of the most promising approach to formal design of system implementations. It is performed by applying transformation (rewriting) rules to program representations in a step-wise manner.

Formal design of highly-parallel systems and VLSI architectures are achieved by transformational derivation of process compositions[1,2]. The principle behind transformational derivation of a process composition is decomposing a single complex process into a composition of some simple processes. A pipeline composition is derived from a process of successive functions, and a parallel composition is derived from a process of respective functions.

First, we investigate derivation of a pipeline composition. The transformation rule to derive a pipeline composition has been specified so far using program patterns :

```
proc_seq f1 xs = proc (seq f1) xs
seq [] x = x
seq (f : f1) x = seq f1 (f x)
      ↓
pipe_proc [] xs = xs
pipe_proc (f : f1) xs =
  pipe_proc f1 (proc f xs)
```

The function "seq" to apply successive functions is, in fact, equivalent to "pipe". So, this pipeline-derivation rule is described also as :

```
proc (pipe f1) xs
      ↓
pipe (map proc f1) xs
```

We specify the rule shown above in an abstract manner as :

```
(proc . pipe)
      ↓
(pipe . map proc)
```

We see that this representation of the derivation rule is more concise than the first one, and we get rid of program patterns like "proc_seq" and "pipe_proc".

Following the same consideration as above, we investigate derivation of a parallel composition as well next. The transformation rule to derive a parallel composition has been specified so far using program patterns :

```
proc_resp f1 xls = proc (resp f1) xls
resp [] [] = []
resp (f : f1) (x : xl) =
  f x : resp f1 xl
      ↓
para_proc [] [] = []
para_proc (f : f1) (xs : xsl) =
  proc f xs : para_proc f1 xsl
```

The function "resp" to apply functions respectively is, in fact, equivalent to "para". So, we specify the parallel-

derivation rule in an abstract manner as :

```
(proc . para)
      ↓
 transpose /|
  ((para . map proc) |/ transpose)
```

where "transpose" is a system function to transpose a list of lists.

In this representation, we have an insight, which could not be noticed before, that the two rules for pipeline derivation and parallel derivation have significant resemblance that we may generalize their essence using a composition variable "c" as :

```
(proc . c)
      ↓
(c . map proc)
```

This kind of insight helps greatly when trying to design new derivation rules. This is another benefit of higher-order abstraction as well as making derivation rules concise and easy to understand. We are now studying transformational derivation of other class of compositions.

6. Concluding Remarks

In this paper, we presented a higher-order abstraction framework for process compositions in functional programming. It is to aid reuse and formal design of repetitive, synchronous and deterministic compositions common to highly-parallel systems and VLSI architectures. It is known that functional programming has some weakness for modeling nondeterministic systems, but this is not the case in our research, since we concern ourselves with deterministic systems.

Through formalizing some basic and common compositions such as pipeline, parallel and tree processes, we investigated the ability and potential of higher-order abstraction. Our achievements include a basis for inheritance and aggregation relationship in a collection of compositions, and analysis and design of transformation rules for deriving compositions.

Our ongoing research to extend this work includes :

- Collecting and formalizing more class of compositions,
- Studying composition inheritance and composition derivation further (as mentioned earlier,)
- Connecting to an object-oriented hardware description language which we are building elsewhere so as to aid VLSI design in a more thorough manner.

It also must be of interest to estimate how many classes of compositions are practically necessary and sufficient from the pragmatic point of view.

Acknowledgments

We are grateful to Dr. Mark P. Jones and Dr. Kevin Hammond for offering Gofer and MacGofer to public ; this research would be more difficult without them.

References

[1] N. Yoshida, "A transformational approach to the derivation of hardware algorithms from recurrence equations", Proc. Supercomputing '88, pp. 433-440, 1988.

[2] N. Yoshida, "Transformational derivation of systolic arrays", Lecture Notes in Computer Science, Vol. 491, pp. 297-311, 1991.

[3] N. Yoshida, "Higher-order abstraction and derivation of process group topologies in functional programming", Lecture Notes in Software Science, Vol. 15, pp. 131-139, 1995 (in Japanese).

[4] R. Bird and P. Wadler, "Introduction to Functional Programming", Prentice-Hall, 1988.

[5] C. Howson, N. Yoshida and K. Araki, "Implementing functional languages on a network of transputers", Proc. Third Transputer/Occam Int'l Conf., pp. 21-33, 1990.

[6] P. Kelly, "Functional Programming for Loosely-Coupled Multiprocessors", Pitman, 1989.

[7] J. Darlington, et al., "Parallel programming using skeleton functions", Proc. Parallel Architectures and Languages Europe (PARLE '93), 1993.

[8] G. Jones and M. Sheeran, "Circuit design in Ruby", in Formal Methods for VLSI Design, ed. J. Staunstrup, pp. 13-70, Elsevier, 1990.

[9] T. Ida and J. Tanaka, "Functional Programming with Streams", Proc. IFIP '83, pp. 265-270, 1983.

[10] T. Ida and J. Tanaka, "Functional Programming with Streams — Part II", New Generation Computing, Vol. 2, No. 3, pp. 261-275, 1984.