Transformational Derivation of Higher-Order Process Compositions

Norihiko Yoshida
Department of Computer and Information Sciences,
Nagasaki University, Nagasaki 852-8521, Japan
E-mail: yoshida@cis.nagasaki-u.ac.jp

We present the current status of our research on a higher-order abstraction framework for process compositions and transformational derivation in functional programming. The aim is to aid formal treatment of system-level design for highly-parallel systems and, in particular, VLSI architectures. Starting from a well-known technique for higher-order abstraction of process compostions, we investigate higher-order abstraction of compostion transformation rules, which abstracts out concrete compostions. This abstract transformation is regarded as transformation of interpreters, which is meta-level transformation. We mention our investigation towards monad-based interpretater transformation.

1 Introduction

Generally speaking, application systems are implemented in one of the forms of: (1) custom software on general-purpose hardware, (2) custom software on custom hardware, or (3) custom hardware. Software design and hardware design have been separated, and most design technologies have focused their attentions on (1) or (3). Today, recent advance of VLSI design technologies is facilitating (2) especially for embedded systems.

"Hardware/software design" is ne approach towards designing hardware-software-mixed systems. In this framework, hardware and software in a system are designed in a cooperative manner, however they are designed still separately. There has emerged a more revolutionary approach, *system-level design*, in which hardware and software are designed in an unified and integrated manner [1]. In other words, system-level design focuses its attention to early stages in a design process where a system is abstract before getting separated into hardware and software.

Towards the system-level design, our research is aiming at *system-level evolution* [a]. This term follows "software evolution" which is vigorously investigated in a joint research project funded by the Ministry of Education in Japan. In software evolution, a system is first specified in a very abstract form such as a mathematical equation, and then an concrete implementation is (or possibly more than one are) derived using formal rules from this specification. At several stages in this design process, pre-defined components of various abstraction levels may be utilized and included. System-level evolution inherits the same framework in principle, however differs in that the outcome is not

necessarily only software, but possibly also hardware or hardware+software.

One of the most crucial in such a design technique is abstraction. There are already abstraction schemes for system components. Abstraction of system compositions (or system topologies) is vigorously investigated in such studies as "Skeletons" by Cole, Kelly, Darlington and many other researchers [3], and "Software Architecture" proposed in Software Engineering. This abstraction is to enable us to build libraries of compositions as reusable "meta-components."

Our research is trying to go further. Formal rules for system derivation (or system evolution) should be abstracted as well, so that we build libraries of derivation rules and procedures. Also, this abstraction is supposed to aid invention or investigation of the rules and procedures. This paper presents the current status of our research.

Abstraction of a process composition yields a higher-order abstract process. It specifies connections and relations of first-order component processes. Abstraction of a composition derivation rule yield a yet higher-order abstract process. It specifies a relation between a given composition and a derived composition. To realize and represent this hierarchy of higher-order abstractions, we use functional programming [b].

The targets of our design technique are parallel systems made of perpetual processes working on data streams, which are repetitive, synchronous, and deterministic (Nondeterministic concurrent systems are out of our concern). Similar systems have also been studied as "reactive systems" [4]. There are some programming languages for them named "synchronous languages" [5,6], and in particular, "Lucid Synchrone" is actually a functional programming language [7]. However, formal derivation of reactive systems has not been well investigated yet.

Our approach is also relevant to the researches of "Ruby" [8] and "Lava" [9] on VLSI hardware design using functional programming. Lava uses monads to switch program manipulation among simulation, verification and logic synthesis, however the detail of Lava monads has not been presented as far as we know. We focus more on system-level design.

Section 2 summarizes some basic (well-known) definitions which will be used in further discussions. Section 3 describes derivation of compositions, or system-level evolution, including an example. Then, Section 4 presents our trial towards abstraction of derivation rules using monads. Section 5 contains

[a] There is a research topic of a similar term, "evolvable hardware", which studies probabilistic optimization of hardware systems [2]. A typical example is hardware design using genetic algorithms. This topic is out of our concern here.

[b] Programs are presented in Gofer (Actually, we use MacGofer developed by Dr. Kevin Hammond).

some concluding remarks.

2 Basic Definitions

Perpetual Process

A perpetual process is a fixed point in system state transition. A process applying a function `f` to a stream argument `xs` is defined in the form of a recursive function as:

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

or just simply

```
proc :: (a -> b) -> [a] -> [b]
proc = map
```

Pipeline Composition of Processes

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

```
infix 3 >>
(p >> q) xs = q (p xs)
```

and a pipeline function `pipe` for a list of processes as:

```
pipe :: [a -> a] -> a -> a
pipe = foldr (>>) id
```

This definition is essentially the same as the one in Skeletons.

Parallel Composition of Processes

A function `para` which represents a parallel composition of processes (possibly identical in a SIMD style) to a bunch of streams is defined as:

```
para :: [a -> b] -> [a] -> [b]
para = curry ((map eval) . zip2)

zip2 (s1, s2) = zip s1 s2
```

The pipeline and parallel compositions are the two most primary ones. More complex compositions such as a bidirectional pipeline and an orthogonal grid are defined using these. Other compositions include such as a binary tree and a hexagonal grid. Our investigation to formalize them in higher-order abstraction are found elsewhere [10].

Regarding formal treatment of our framework, O'Donnell's Parallel Abstract Machine (PAM) [11] has a motivation similar to ours, and can be a good

basis. To apply PAM to our framework, PAM should be extended with stream-based data-flow computation. "Synchronous Automata" [12], originally proposed for synchronous languages, is used for this extension.

3 Transformation of Compositions

Transformational derivation from (mathematical) specifications is one of the most promising approach to formal system-level design.

We already achieved some formal derivation of highly-parallel systems and VLSI architectures based on transformation of process compositions [13]. Below is an example (The original version was presented in a logic programming language). This is an outline of derivation of an hardware algorithm, or a systolic array, for a finite impulse response filter defined as:

$$y_i = \sum_{j=0}^{k-1} w_j x_{i+j}$$

where w_j, x_i, y_i are weights, an input sequence, an output sequence respectively. This recurrence equation is mapped on the below program (A), where `pp` is interpreted as a process.

```
fir ws xs = pp ws xs [0,0..]
pp ws (x : xs) (y : ys) = p ws (x : xs) y : pp ws xs ys
p [] xs y = y
p (w : ws) (x : xs) y = p ws xs (f w x y)
```

It is transformed to program (B):

```
fir ws xs = pp ws xs [0,0..]
pp [] xs ys = ys
pp (w : ws) (x : xs) ys = pp ws xs (p w (x : xs) ys)
p w (x : xs) (y : ys) = f w x y : p w xs ys
```

and then to program (C), where `p` is interpreted as a process, and `pp` is interpreted as a pipeline.

```
fir ws xs = ys' where (_, ys') = pp ws xs [0,0..]
pp [] xs ys = (xs, ys)
pp (w : ws) xs ys = pp ws xs' ys'
        where (_ : xs' ys') = p w xs ys
p w (x : xs) (y : ys) = (x : xs', f w x y : ys')
        where (xs', ys') = p w xs ys
```

This transformation is illustrated in Fig. 1.

The principle behind the technique is to decompose a single complex process into a composition of some simple processes. A pipeline composition is

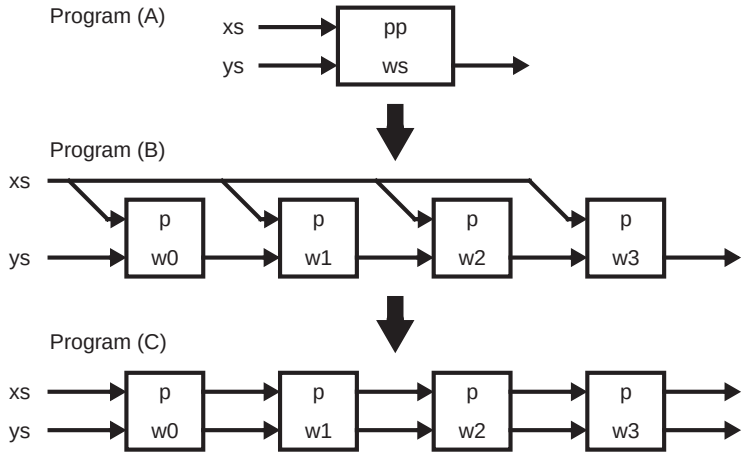

Figure 1: Transforamtional Derivation of Systolic Array.

derived from a single process applying functions consecutively. Likewise, an SIMD composition is derived from a single process applying a set of functions to a bunch of streams.

Now we investigate higher-order abstraction of transformation rules based on the higher-order abstraction of compositions. The transformation rule to derive a pipeline composition is specified using program templates:

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

Here, `seq` is identical to `pipe`, and this rule is abstracted as:

```
proc_seq :: [a -> a] -> [a] -> [a]
proc_seq = (proc . pipe)
      ↓
pipe_proc :: [a -> a] -> [a] -> [a]
pipe_proc = (pipe . map proc)
```

The correctness of this rule is shown in an inductive manner as follows. We assume that:

```
pipe (map proc fs) xs == proc (seq fs) xs
```

and show:

```
pipe (map proc fs) (x : xs) == proc (seq fs) (x: xs)
```

as follows (-- indicates a comment):

```
pipe (map proc [f1, f2, ..., fn]) (x : xs)
 == pipe [proc f1, proc f2, ..., proc fn] (x : xs)
                          -- unfold map
 == proc fn (... (proc f2 (proc f1 (x : xs))))
                          -- unfold pipe
 == proc fn (... (proc f2 (f1 x : proc f1 xs)))
                          -- unfold the innermost proc
 == proc fn (... (f2 (f1 x) : proc f2 (proc f1 xs)))
                          -- once more
 == ...
 == fn (... (f2 (f1 x))) :
        proc fn (... (proc f2 (proc f1 xs)))
 == seq [f1, f2, ..., fn] x :
        pipe [proc f1, proc f2, ..., proc fn] xs
                          -- fold seq and pipe
 == seq [f1, f2, ..., fn] x :
        pipe (map proc [f1, f2, ..., fn]]) xs
                          -- fold map
 == seq [f1, f2, ..., fn] x :
        proc (seq [f1, f2, ..., fn]) xs
                          -- apply the assumption
 == proc (seq [f1, f2, ..., fn]) (x : xs)
                          -- fold proc
```

The transformation rule to derive an SIMD composition is specified as:

```
proc_resp fs xss = proc (resp fs) xss
resp [] [] = []
resp (f : fs) (x : xs) = f x : resp fs xs
```

↓

```
para_proc [] [] = []
para_proc (f : fs) (xs : xss) =
        proc f xs : para_proc fs xss
```

This rule is abstracted as:

```
proc_resp :: [a -> b] -> [[a]] -> [[b]]
proc_resp = (proc . para)
      ↓
para_proc :: [a -> b] -> [[a]] -> [[b]]
para_proc = transpose /| (para . map proc) |/ transpose

infix 2 |/
infix 1 /|
(f |/ g) x y = f x (g y)
(f /| g) x y = f (g x y)
```

Inductively, we have an insight that (at least, some) rules for composition derivation can be generalized and abstracted using a composition variable `z` as:

```
proc_comp :: (a -> b -> c) -> a -> [b] -> [c]
proc_comp = \z -> (proc . z)                    -- (D)
      ↓
comp_proc :: ([[a] -> [b]] -> c) -> [a -> b] -> c
comp_proc = \z -> (z . map proc)                -- (E)
```

where "`\<pat> -> <expr>`" is a lambda expression.

This looks natural also from the intuitive standpoint of view in that a process of a composition of functions is transformed to a composition of processes. We can invent a new composition-derivation rule by assigning a composition to `z`. The generality of this highly abstracted transformation rule is being investigated.

4 Towards Monad-based Interpretation

Now the composition is abstracted out as a higher-order variable from the transformation rules. Provided a value (actually a composition) for this variable, The above program (D) yields a process containing the composition, and (E) yields a composition of processes which is equivalent to (D). The transformation is specified regardless of the value of the variable. This scheme can be formalized also as that we have two interpreters for a composition, each of which corresponds to (D) and (E) respectively, and the transformation is specified as transformation of the interpreters. This transformation of interpreters is considered as a meta-level transformation, as the composition which they interpret is at the base-level.

In order to build a framework for interpreter transformation, or meta-level transformation, we are investigating an applicability of "monads" [14]. Monads

are category-theoretic constructs that allows a program to manipulate computations as values, which is a kind of reflection, or meta-level computation. Monads are to use introduce, for example, states, I/O's, continuations, concurrencies, nondeterminism, and so on to purely functional programming.

In a (possible) monad-based framework, we will define two composition monads, `S` and `C`, such that, given a composition `z`, for example `pipe` or `para`, its interpretation under the monad `S` yields a single process of the composition of functions, while the interpretation under the monad `C` yields the composition of processes. The transformation will be specified as transformation from the monad `S` to the monad `C`.

Towards this target, we are investigating in particular the monadic imperative streams [15] and the vectorisation monads [16]. The former introduces an stream extension to I/O monads; the latter investigates transformation from an imperative `for` loop to purely functional `fold`, `map` and `scan`. An inverse of this transformation will help to build an monadic interpreter for our framework.

5 Concluding Remarks

We presented a higher-order abstraction framework for process compositions and their transformational derivation rules in functional programming. It is to aid reuse and formal design in system-level design of highly-parallel systems and VLSI architecture.

As the readers can easily seen, we are still at the starting point of our research. Monad-based interpretation for this framework is our ongoing task. The next crucial issue to address is, from the system-level design standpoint of view, criterions on when and where to separate hardware and software.

Acknowledgments

We are grateful to Dr. Kevin Hammond and Dr. Phil Trinder for their helpful comments and suggestions at the Workshop. This research is partly supported by a joint research project on "Software Evolution" funded by the Ministry of Education in Japan.

References

1. `http://www.intermetrics.com/SLDL/`
2. `http://www.etl.go.jp/~ehw/`
3. `http://www.dcs.ed.ac.uk/home/mic/skeletons.html`

4. N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer (1993)
5. G. Berry, "The Foundations of Esterel", *Proof, Language and Interaction: Essays in Honour of Robin Milner* (G. Plotkin, C. Stirling and M. Tofte, eds), MIT Press (1998)
6. N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre", Proc. IEEE, 79:9 (1991)
7. `http://www-spi.lip6.fr/softs/lucid-synchrone.html/`
8. G. Jones and M. Sheeran, "Circuit Design in Ruby", in *Formal Methods for VLSI Design* (J. Staunstrup ed.), Elsevier, 13-70 (1990)
9. P. Bjesse, K. Claessen, M. Sheeran and S. Singh, "Lava: Hardware Design in Haskell", Proc. ICFP'98 (1998)
10. N. Yoshida, "Higher-Order Abstraction of Process Compositions and Their Transformation", Reports of the Faculty of Engineering, Nagasaki Univ., 29:52, 67-71 (1999)
11. J. O'Donnell and G. Runger, "A Methodology for Deriving Parallel Programs with a Family of Parallel Abstract Machines", Proc. 3rd Int'l Euro-Par Conf., 662-669 (1997)
12. O. Maffeis and A. Poigne, "Synchronous Automata for Reactive, Real-Time or Embedded Systems", Technical Report No. 967, German National Research Center in Information Technology (GMD) (1996)
13. N. Yoshida, "Transformational Derivation of Systolic Arrays", in *Concurrency: Theory, Language and Architecture* (T. Ito and A. Yonezawa eds.), Lecture Notes in Computer Science 491, 297-311 (1991)
14. P. Wadler, "The Essence of Functional Programming", Proc. 19th POPL, 1-14 (1992)
15. E. Scholz, "Imperative Streams – A Monadic Combinator Library for Synchronous Programming", Proc. ICFP'98 (1998)
16. J. M. D. Hill, K. M. Clarke and R. Bornat, "The Vectorisation Monad", Proc. PASCO'94 (1994)