

Research Report 189

Adding Second Order Functions to Kahn Data Flow

Steve Matthews

RR189

Gilles Kahn's elegant model of deterministic concurrent computation using sequential processes connected by Unix style pipes [Ka74] has been used by Ashcroft & Wadge as the basis for the functional data flow programming language Lucid [W & A85]. Kahn's data flow model is "first order", and so does not allow streams of processes for example. This has been reflected in Lucid by the restriction, until recently, to first order functions. In [Wa91] Wadge suggests a treatment for implementing higher order function definitions in Lucid using intensional multi-dimensional algebras. This has unfortunately placed the Lucid cart before the Kahn data flow horse. Is there a natural extension to Kahn's model which allows higher order functions, or can we only understand them in terms of a specific programming language such as Lucid? This report argues that higher order functions can be understood in terms of Kahn's data flow model by constructing a treatment therein for second order functions. This construction provides new insights into obtaining a generalisation of Kahn Data Flow which allows second order functions.

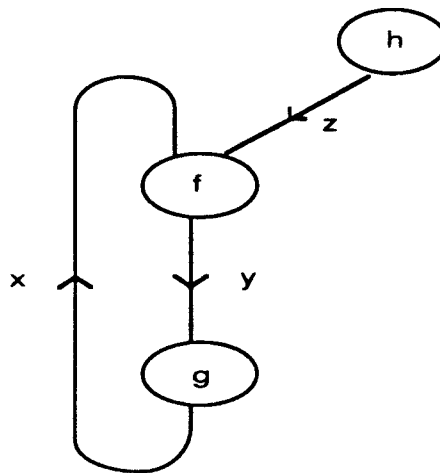
1) Introduction

Kahn Data Flow [Ka74] is a deterministic model of parallel computation in which processes in a network of processes communicate asynchronously by sending sequentially packages of data along uni-directional UNIX style "pipes". Each pipe is permanently connected to precisely one "producer" process and one "consumer" process. Packages sent through a pipe must be consumed by the pipe's consumer process in the same order in which they are produced by that pipe's producer process. Each process "waits" in its current state until the next package needed is available, upon which the process "fires" by consuming that package, possibly producing package(s), and finally updating its state ready for the next firing. Processes have no side effects, and can only communicate with other processes via pipes. Networks of processes may have either a finite or infinite number of processes.

Kahn conjectured correctly in [Ka74] that a network can be described denotationally as the least fixed point of a set of recursion equations over a chain complete partial order of "streams". The domain of streams Ka^D of streams of packages in D is the cpo of all finite and infinite sequences of members of D under the initial segment ordering. Processes are denoted by chain continuous functions of the form,

$$f : (Ka^D)^n \rightarrow (Ka^D)^m$$

and pipes by streams. For example, the denotation of the network,



is the least fixed point of the equations,

$$\begin{aligned} x &= g(y) \\ y &= f(x, z) \\ z &= h() \end{aligned}$$

2) Second Order Functions

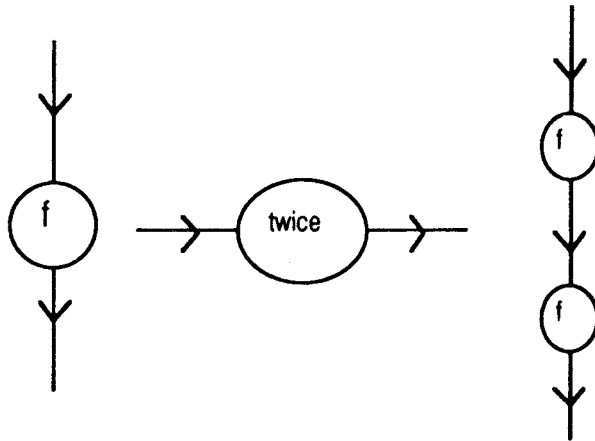
In any functional programming language second order functions such as,

$$twice(f)(x) ::= f(f(x))$$

are clearly preferable to writing multiple "instances" of the same definition, e.g.

$$\begin{aligned} twice-f(x) &::= f(f(x)) \\ twice-g(x) &::= g(g(x)) \end{aligned}$$

Second order definitions are clearly more expressive, and thus no functional language worth its salt can afford to be without them. Kahn's model does not, at first sight, suggest a natural extension which includes such second order functions. Interpreting the above example *twice* as a Kahn network would give us the following intuitive scenario.



Twice consumes a network with denotation f to produce a Kahn network with denotation $f \circ f$. However, if this scenario is to be formalised in Kahn Data Flow then both the input network for *twice* and its output network must be representable as streams. The first task in this report is thus to construct a stream representation for first order functions.

3) First Order Functions

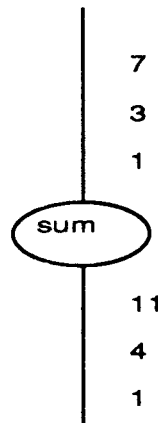
A first order Kahn process is both deterministic and sequential, thus to understand such a process as a stream it is necessary to understand the behaviour of that process. After a process with denotation $f: Ka^D \rightarrow Ka^D$ has consumed precisely the input $x \in Ka^D$ that process must have

produced precisely the output $f(x) \in Ka^D$ before consuming anymore input (if anymore exists). The "behaviour" of the process can thus be modelled by a finite or infinite sequence of "events". Each event is essentially the consumption of the next input $x_i \in D$ and the production of a finite or infinite output sequence in Ka^D as a response. This would suggest that we could model such behaviour by a stream of functions $f_n : D \rightarrow Ka^D$. For example, perhaps the stream function $inc : Ka^\omega \rightarrow Ka^\omega$ could be modelled by the stream of functions $inc_n : \omega \rightarrow Ka^\omega$, where,

$$\forall n \geq 0, \quad a \in \omega \quad . \quad inc_n(a) = \langle a+1 \rangle$$

Unfortunately not all stream functions can be modelled in this way, for example, the function $next : Ka^D \rightarrow Ka^D$ where $next(x)_i = x_{i+1}$. Here the i 'th output is not a function of the i 'th input, and so we cannot model it in the same way as we did inc .

Each event in the behaviour of a process thus cannot just be the production of the next output, but also the updating of the current "state" of the process. The information held in the state together with the next input will determine the next output. For example in the process *sum*,



in which each output is the sum of all inputs consumed so far the state is the running total of the sum so far. In this example we can imagine the state to be a register held inside the process which is updated with each event.

4) Function Streams

Definition 4.1

A "function stream" over a domain D is an infinite stream of function pairs $\langle f^+_n, f^-_n \rangle$ denoted $(\langle f^+, f^- \rangle)$ s.t.,

$$1) \quad \forall n \geq 0 \quad . \quad \begin{array}{lll} f^+_n & : & D \times ST \rightarrow Ka^D \\ f^-_n & : & D \times ST \rightarrow ST \end{array}$$

$$2) \quad f^+_0 \quad \text{and} \quad f^-_0 \quad \text{are constant functions.}$$

Here the set ST is intended to be the set of all possible states in which a process can be in during its execution. In the notion of state used here we have included (by definition) for technical convenience $f^-_n(a, s)$ as a state when $|f^+_n(a, s)| = \infty$. In other words we have included in our set ST of states "artificial" members which correspond to the cases in which a finite input can produce an infinite output.

Definition 4.2

Let $@ : Ka^{Ka^D} \rightarrow Ka^D$ be the function which appends a stream of streams together.

$$\text{Note that,} \quad \forall x, y \in Ka^D \quad . \quad \begin{array}{lll} x \leq y & \Rightarrow & x @ (y - x) = y \\ |x| = \infty & \Rightarrow & x @ y = x \end{array}$$

A function stream is a state transition model of the behaviour of a process. The next definition defines the function computed by a function stream.

Definition 4.3

The "value" of a function stream $\langle f^+, f^- \rangle$ is the function $f : Ka^D \rightarrow Ka^D$ s.t.,

$$\forall x \in Ka^D \quad . \quad \begin{array}{lll} |x| < \infty & \Rightarrow & f(x) = @_{0 \leq i \leq |x|} v_i \\ |x| = \infty & \Rightarrow & f(x) = @_{0 \leq i < \infty} v_i \end{array}$$

$$\text{where,} \quad v_0 ::= f^+_0()$$

$$\begin{aligned}
s_0 &::= f_0() \\
\forall i > 0 \quad . \quad & \begin{aligned} v_i &::= f_{+i}(x_{i-1}, s_{i-1}) \\ s_i &::= f_{-i}(x_{i-1}, s_{i-1}) \end{aligned}
\end{aligned}$$

Theorem 4.1

The value of a function stream is chain continuous.

Proof:

As @ is chain continuous.

□

The function $next : Ka^D \rightarrow Ka^D$ used above is an example of a process which needs at least two states in order to formulate it as a function stream.

Example 4.1

Let $\langle next+, next- \rangle$ be the function stream s.t.,

$$\begin{aligned}
1) \quad \forall n \geq 0 \quad . \quad & \begin{aligned} next+_n &: D \times \{\epsilon, \delta\} \rightarrow Ka^D \\ next-_n &: D \times \{\epsilon, \delta\} \rightarrow \{\epsilon, \delta\} \end{aligned}
\end{aligned}$$

$$2) \quad next+_0() = \langle \rangle$$

$$next-_0() = \epsilon$$

$$\forall n > 0, \quad a \in D \quad . \quad next+_n(a, \epsilon) = \langle \rangle$$

$$next+_n(a, \delta) = \langle a \rangle$$

$$next-_n(a, \epsilon) = \delta$$

$$next-_n(a, \delta) = \delta$$

Then the value of $\langle next+, next- \rangle$ is the function $next : Ka^D \rightarrow Ka^D$. In the next example the state is intended to be a "register" which holds the running total of the sum of all the inputs read so far.

Example 4.2

Let $\langle \text{sum}^+, \text{sum}^- \rangle$ be the function stream s.t. ,

$$1) \quad \forall n \geq 0 . \quad \text{sum}^+_n : \omega \times \omega \rightarrow \{ \langle \rangle \} \cup \{ \langle n \rangle \mid n \in \omega \}$$

$$\text{sum}^-_n : \omega \times \omega \rightarrow \omega$$

$$2) \quad \text{sum}^+_0() = \langle \rangle$$

$$\text{sum}^-_0() = 0$$

$$\forall n \geq 0 , \quad a , b \in \omega . \quad \text{sum}^+_n(a, b) = \langle a + b \rangle$$

$$\text{sum}^-_n(a, b) = a + b$$

Then the value of $\langle \text{sum}^+, \text{sum}^- \rangle$ is the function $\text{sum} : Ka^\omega \rightarrow Ka^\omega$ used in section 3. We now prove the most important result in this report, that our construction for turning first order functions into streams can be applied to any chain continuous function over Ka^D .

Theorem 4.2

Each chain continuous function $f : Ka^D \rightarrow Ka^D$ is the value of some function stream $\langle f^+, f^- \rangle$.

Proof:

Let $f : Ka^D \rightarrow Ka^D$ be a chain continuous function.

Let Ka^{D*} denote the subset of Ka^D of all finite sequences.

Let $\langle f^+, f^- \rangle$ be the unique function stream s.t. ,

$$1) \quad \forall n \geq 0 . \quad f^+_n : D \times Ka^{D*} \rightarrow Ka^D$$

$$f^-_n : D \times Ka^{D*} \rightarrow Ka^{D*}$$

$$2) \quad f_+(a) = f(<a>)$$

$$f_-(a) = <a>$$

$$\forall n > 0, a \in D, x \in Ka^{D*}$$

$$f_{+n}(a, x) = f(x @ <a>) - f(x)$$

$$f_{-n}(a, x) = x @ <a>$$

Then f is the value of $<f_+, f_->$.

□

While this last result proves that we can in principle turn first order functions into streams it is not a construction which could be considered in practice, as the number of states in Ka^{D*} is unnecessarily large for most functions likely to be programmed in a language such as *Lucid*. The function *next* used earlier is a simple example of this phenomenon as it needs only two states. We now introduce a refinement of this set of states which in general cuts down the number of states needed.

Definition 4.4

For each chain continuous function $f: Ka^D \rightarrow Ka^D$ let $\equiv^f$ be the unique equivalence relation on Ka^{D*} s.t. ,

$$\begin{aligned} \forall x, y \in Ka^{D*} \quad . \quad x \equiv^f y \quad \text{iff} \\ (|f(x)| = \infty \quad \wedge \quad |f(y)| = \infty) \quad \vee \\ (|f(x)| < \infty \quad \wedge \quad |f(y)| < \infty \quad \wedge \\ (\quad \forall z \in Ka^{D*} \quad . \quad f(x @ z) - f(x) \\ = \quad f(y @ z) - f(y) \quad) \quad) \end{aligned}$$

The next result expresses the fundamental deterministic property of Kahn processes that once two identical processes are in the same state they will forever produce the same results when given the same input.

Theorem 4.3

For each chain continuous function $f: Ka^D \rightarrow Ka^D$,

$$\forall x, y \in Ka^{D*}, a \in D \quad . \quad x \equiv^f y \quad \Rightarrow \quad x @ <a> \equiv^f y @ <a>$$

Proof:

Suppose that $x \equiv^f y$ and that $a \in D$ then ,

$|f(x)| = \infty \wedge |f(y)| = \infty \Rightarrow |f(x@<a>)| = \infty \wedge |f(y@<a>)| = \infty$
as f is continuous.

Thus, $x@<a> \equiv^f y@<a>$

Suppose now that $|f(x)| < \infty \wedge |f(y)| < \infty \wedge$

$$\begin{aligned} & (\quad \forall z \in Ka^{D^*} . \quad f(x@z) - f(x) \\ & \quad \quad \quad = f(y@z) - f(y) \quad) \end{aligned}$$

Then for each $z \in Ka^{D^*}$,

$$\begin{aligned} & f((x@<a>)@z) - f(x@<a>) \\ & = f(x@(<a>@z)) - f(x@<a>) \\ & = f(x)@ (f(x@(<a>@z)) - f(x)) - f(x@<a>) \\ & = f(x)@ (f(x@(<a>@z)) - f(x)) \\ & \quad \quad \quad - f(x)@ (f(x@<a>) - f(x)) \\ & = f(y)@ (f(x@(<a>@z)) - f(x)) \\ & \quad \quad \quad - f(y)@ (f(x@<a>) - f(x)) \\ & = f(y)@ (f(y@(<a>@z)) - f(y)) \\ & \quad \quad \quad - f(y)@ (f(y@<a>) - f(y)) \\ & = f(y@(<a>@z)) - f(y@<a>) \\ & = f((y@<a>)@z) - f(y@<a>) \end{aligned}$$

Thus, $x@<a> \equiv^f y@<a>$

The next result shows that we can use the quotient set $Ka^{D^*}/\equiv^f$ as a set of states.

Theorem 4.4

For each chain continuous function $f : Ka^D \rightarrow Ka^D$ there exists a function stream $\langle f^+, f^- \rangle$ with value f and state set $Ka^{D^*}/\equiv^f$.

Proof:

Let $f : Ka^D \rightarrow Ka^D$ be a chain continuous function.

Let $\langle f_+, f_- \rangle$ be the unique function stream s.t. ,

$$1) \quad \forall n \geq 0 \quad . \quad f_{+n} : D \times Ka^{D*} / \equiv^f \rightarrow Ka^D$$

$$f_{-n} : D \times Ka^{D*} / \equiv^f \rightarrow Ka^{D*} / \equiv^f$$

$$2) \quad f_{+0}() = f(\langle \rangle)$$

$$\langle \rangle \in f_{-0}()$$

$$\forall n > 0, \quad a \in D, \quad s \in Ka^{D*} / \equiv^f, \quad x \in s \quad .$$

$$f_{+n}(a, s) = f(x @ \langle a \rangle) - f(x)$$

$$x @ \langle a \rangle \in f_{-n}(a, s)$$

Then f is the value of $\langle f_+, f_- \rangle$.

□

Note that in the last theorem the equivalence class,

$$\{ \quad x \in Ka^{D*} \quad / \quad |f(x)| = \infty \quad \}$$

(if it exists) corresponds to the "artificial" state in which a process produces infinite output from a finite input. This particular class is included only for technical elegance & simplicity, when we actually could have done without it.

5) Multi-argument Functions

The function stream approach for modelling functions over streams as streams can be extended to multi-argument functions of the form,

$$f : Ka^D \times \dots \times Ka^D \rightarrow Ka^D$$

Example 5.1

Let fby (pronounced "followed by") [W&A85] be the function s.t.,

$$1) \quad fby : Ka^D \times Ka^D \rightarrow Ka^D$$

$$2) \quad \forall x, y \in Ka^D .$$

$$|x| = 0 \quad \Rightarrow \quad fby(x, y) = \langle \rangle$$

$$|x| > 0 \quad \Rightarrow \quad fby(x, y) = \langle x_0 \rangle @ y$$

Such functions can be modelled using the following generalisation of a function stream.

Definition 5.1

For each $m \geq 1$ an "m-argument function stream" over a domain D is an infinite stream of function pairs $\langle f_{+n}, f_{-n} \rangle$ (denoted $\langle f_{+}, f_{-} \rangle$) s.t. ,

$$1) \quad \forall n \geq 0 . \quad f_{+n} : D \times \{1, \dots, m\} \times ST \rightarrow Ka^D$$

$$f_{-n} : D \times \{1, \dots, m\} \times ST \rightarrow ST$$

$$2) \quad f_{+0} \text{ and } f_{-0} \text{ are constant functions.}$$

$$3) \quad \forall n \geq 0 , \quad a, b \in D , \quad 1 \leq i \neq j \leq m \quad s \in ST .$$

$$f_{+n}(a, i, s) @ f_{+n+1}(b, j, f_{-n}(a, i, s))$$

$$= f_{+n}(b, j, s) @ f_{+n+1}(a, i, f_{-n}(b, j, s))$$

Example 5.2

A generalised function stream to represent the function fby is,

$$1) \quad \forall n \geq 0 .$$

$$fby_{+n} : D \times \{1, 2\} \times (\{\varepsilon\} + Ka^{D*}) \rightarrow Ka^D$$

$$fby_{-n} : D \times \{1, 2\} \times (\{\varepsilon\} + Ka^{D*}) \rightarrow \{\varepsilon\} + Ka^{D*}$$

$$2) \quad fby_+(\epsilon) = \langle \rangle$$

$$fby_-(\epsilon) = \langle \rangle$$

$$\forall n > 0, \quad a \in D, \quad x \in Ka^{D^*} \quad . \quad fby_+(a, 1, \epsilon) = \langle \rangle$$

$$fby_-(a, 1, \epsilon) = \epsilon$$

$$fby_+(a, 2, \epsilon) = \langle a \rangle$$

$$fby_-(a, 2, \epsilon) = \epsilon$$

$$fby_+(a, 1, x) = \langle a \rangle @ x$$

$$fby_-(a, 1, x) = \epsilon$$

$$fby_+(a, 2, x) = \langle \rangle$$

$$fby_-(a, 2, x) = x @ \langle a \rangle$$

The third condition in Definition 5.1 is to ensure that m-argument function streams do in fact have a value. The problem in, for example, defining $fby(\langle a \rangle, \langle b \rangle)$ in terms of $\langle fby_+, fby_- \rangle$ is that there is in general more than one way to consume the input pair of sequences $(\langle a \rangle, \langle b \rangle)$, e.g.

$$(\langle \rangle, \langle \rangle) \leq (\langle a \rangle, \langle \rangle) \leq (\langle a \rangle, \langle b \rangle)$$

or,

$$(\langle \rangle, \langle \rangle) \leq (\langle \rangle, \langle b \rangle) \leq (\langle a \rangle, \langle b \rangle)$$

The third condition ensures that whichever way we consume the input the resulting output will always be the same. The definition of a "value" for m-argument function streams is straightforward but messy, and so is not included here. Instead consider the following example of how $fby(\langle a \rangle, \langle b \rangle)$ can be deduced from $\langle fby_+, fby_- \rangle$ in two different ways depending on whether we choose to consume a or b first.

$$\begin{aligned} & f_+(\epsilon) @ f_+(a, 1, f_-(\epsilon)) @ f_+(b, 2, f_-(a, 1, f_-(\epsilon))) \\ &= \langle \rangle @ f_+(a, 1, \langle \rangle) @ f_+(b, 2, f_-(a, 1, \langle \rangle)) \\ &= f_+(a, 1, \langle \rangle) @ f_+(b, 2, \epsilon) \\ &= (\langle a \rangle @ \langle \rangle) @ \langle b \rangle \\ &= \langle a, b \rangle \end{aligned}$$

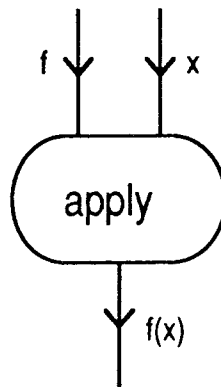
$$\begin{aligned}
& f_0() @ f_1(b, 2, f_0()) @ f_2(a, 1, f_1(b, 2, f_0())) \\
&= <> @ f_1(b, 2, <>) @ f_2(a, 1, f_1(b, 2, <>)) \\
&= f_1(b, 2, <>) @ f_2(a, 1, <> @) \\
&= <> @ f_2(a, 1,) \\
&= <a> @ \\
&= <a, b>
\end{aligned}$$

An analogue to Theorem 4.2 can be given, but not here, for multi-argument chain continuous functions.

6) Conclusions and Further Work

The method described in this report for constructing second order functions can clearly be applied in principle to even higher order functions. As a purely Kahn Data Flow construction it is questionable as to exactly how useful it is to existing Lucid ideas on implementation. It is worth noting here though that the Lucid community's informal notion of a "functor" appears to be exactly the "function stream" introduced in this report. The potential virtues of the ideas in this report really lie in future truly distributed parallel implementations of Lucid like stream based higher order functional programming languages. A critical point in such implementations will be an adequate understanding of programs to ensure a realistically small state set ST . Consequently this is both an interesting and necessary area for further work in the development of higher order asynchronous parallel computation.

The work in this report has another interesting application in developing a theory of functions based upon streams. Analogous to the lambda calculus an example of a basic process is,



Instead of evaluating f either eagerly or lazily, f flows into the *apply* process. The motivation for this is to construct a parallel model for the lambda calculus based upon a distributed

data flow approach rather than the more traditional reduction approach.

7) References

- [Ka74] *The Semantics of a Simple Language for Parallel Processing*,
Gilles Kahn , Proc. IFIP Congress 1974, pp. 471-475,
Elsevier North Holland, Amsterdam.

- [W&A85] *Lucid, the Dataflow Programming Language*
W.W. Wadge & E.A. Ashcroft, APIC Studies in Data Processing No.22,
Academic Press, 1985.

- [Wa91] *Higher Order Lucid*,
W.W. Wadge, Fourth International Symposium on Lucid
and Intensional Programming, SRI International, Menlo Park,
California, USA, April 29-30th, 1991.