Solving stream equation systems

Bruno Barbier[1]

1: Laboratoire d'Informatique de Franche-Comté
Route de Gray, 25000 Besançon, France
`barbier@lifc.univ-fcomte.fr`

Résumé

Many paradigms can be formally encoded with streams. One specifies streams in an intuitive and modular way by using mutually recursive equations. In this paper, we present a method that, given a stream defined by a system of stream equations, computes the solution of the system. The result is a simple automaton whose state contains only a finite number of stream values and the method succeeds under a syntactic, practical and intuitive condition. Moreover, the method relies only on parametricity and ensures laziness. The advantages are: the system is guaranteed to have one unique solution and this solution is a simple combinator application; the equations are fused; algorithmic calculations to derive efficient implementations or to draw proofs are possible.

1. Introduction

Streams, i.e. infinite sequence of values [1], have been used for a long time in many applications [25]. Examples of these domains are microprocessor specifications [19], reactive systems (for example, Lustre [11]), input/output encoded as streams in functional languages (for example, ML [20]), analytic functions [23], etc. Indeed, streams provide a natural formalization for many paradigms: the infinite sequence of values may be a discrete signal or the evolution of a variable through time; it can also be a sequence of (discrete) events that are received or sent by an operating system; mathematical sequences can also be modeled as streams, and, the successive state of a computer running forever is also a stream.

As any specification, stream specifications are easier to write in a divide and conquer manner; therefore, recursion and modularity play a crucial role. But these specifications have three main drawbacks.

Such systems of mutually recursive equations are executable in a lazy functional language like Haskell but their results are not always meaningful. For example, the expression `let` $x = x$ `in` x is executable; its execution may just die or may run forever depending on the underlying implementation. This kind of meaningless definition can be hidden in a set of mutually recursive definitions, and so, be hard to spot. Moreover, the straight execution of mutually recursive equations builds and destroys many intermediate interleaved structures; this is time-consuming. Finally, proofs are difficult because there are many intricate sub-cases to study and proof-assistants do not allow arbitrary recursion: we are restricted to syntactically guarded definitions (Coq [9]) or we have to give validity proofs (Isabelle [22]).

This contrasts with the fact that many interesting streams are definable coalgebraically in pure system F without relying on a fix-point ([29]); in this framework, a stream is the automaton that produces the stream values. Because there is no fix-point operator, expression equivalent to `let` $x = x$ `in` x can not be defined. As streams are defined with a simple combinator (the automaton constructor),

[1] The word *stream* is also used for *potentially* infinite sequence of values (see [16]) that have to be evaluated lazily.

simple calculational laws apply [5] and we can avoid building/destroying intermediate structure by fusion [26]. Finally, the combinatorial approach allows to define streams in a logical framework like the Calculus of Construction [4] and to draw proofs [6].

In this paper, we provide a method to compute the automaton solution of a system of mutually recursive equations. This is always possible under a syntactic condition which ensures that a system has one unique meaningful solution.

For example, consider the Fibonacci sequence defined as a stream $\widetilde{fib}$ (we will write $\widetilde{x}$ to emphasize that x is a stream): the first element of the stream $\widetilde{fib}$ is $fib_0 = 0$, the second is $fib_1 = 1$, and the kth value of the stream $\widetilde{fib}$ is $fib_k = fib_{k-2} + fib_{k-1}$. Using standard functions on lists, we get the following definition of the Fibonacci sequence in Haskell:

$$\begin{array}{ll} \text{let} & \widetilde{fib} = 0 : \widetilde{fib}' \\ & \widetilde{fib}' = 1 : \mathbf{zipWith}\ (+)\ \widetilde{fib}\ \widetilde{fib}' \\ \text{in}\ \widetilde{fib} & \end{array} \tag{1}$$

where $\mathbf{zipWith}\ f\ (u_1 : u_2 : u_3 : \ldots)\ (v_1 : v_2 : v_3 : \ldots) = f\, u_1\, v_1 : f\, u_2\, v_2 : f\, u_3\, v_3 : \ldots$.

Our method computes the automaton solution of this system :

$$\widetilde{fib} = \mathbf{str}\ (\pi_1,\ \lambda(x,y).(y, x+y))\ (0,1) \tag{2}$$

where **str** is the combinator that builds the stream (or automaton). Let f be the function $\lambda(x,y).(y, x+y)$, the automaton produces the following sequence:

$$\pi_1(0,1),\ \ \pi_1(f\,(0,1)),\ \ \pi_1(f\,(f\,(0,1))),\ \ \ldots,\ \ \pi_1(f^k\ (0,1)),\ \ \ldots$$

where $f^0 = \mathbf{id}$, $f^{n+1} = f^n \circ f$ and π_1 is the first projection on pairs.

The automaton (2) produces the unique solution of the system (1).

Our method is based on simple laws and strategic rewritings. We use a family of generic functions $\mathbf{fstr}_k$ to define all functions from k streams into one stream; all the fusions we apply are proved by the fusion laws of the generic functions $\mathbf{fstr}_k$.

The method derives the solution from the syntactic structure of the equations; stream values may be of any type like integers, functions or streams. Moreover, the method respects full-laziness and ensures that the state of the automaton solution contains only a finite number of stream values.

We choose to present the method for the lazy functional language Haskell; but the transformation would be valid in other frameworks. Indeed, the transformation relies only on parametricity. We use parametricity as described by Philip Wadler in *Theorems for free!* [28] and extended by John Launchbury and Ross Paterson in *Parametricity and Unboxing with Unpointed Types* [17] to work with bottom and a fix-point operator. In this framework, types are cpos, they may be pointed (have a least element) or not. Each type contains the least upper bound of any increasing chains of its elements. Functions are continuous functions between cpos and relations are relations between cpos. Parametricity allows to derive properties of a term by only looking at its type. For example, a term of type $\forall\alpha.\alpha \to \alpha$ can only be the identity function and a term of type $\forall\alpha.\alpha \to \mathbf{Int}$ can only be a constant function. Appendix A gives the parametricity schema for binary relations.

The three main contributions of this paper are:

- to give a generic function that allows to define stream functions: these functions can be fused;
- to present a transformation that, under a simple syntactic criteria, computes the unique solution of a system of mutually recursive stream equations: the solution is a simple automaton $\mathbf{str}\ (p, f)\ s$,

$$\begin{array}{lll} \mathbf{str} & :: & \forall\alpha.(\alpha \to \mathsf{a}, \alpha \to \alpha) \to \alpha \to \mathrm{Str}\ \mathsf{a} \\ \mathbf{shd} & :: & \mathrm{Str}\ \mathsf{a} \to \mathsf{a} \\ \mathbf{stl} & :: & \mathrm{Str}\ \mathsf{a} \to \mathrm{Str}\ \mathsf{a} \end{array}$$

Figure 1: Axiomatic Stream Definition: Types

$$\begin{array}{lrcl} \mathbf{Ax_1} & \mathbf{shd}\ (\mathbf{str}\ (p, f)\ s) & = & p\ s \\ \mathbf{Ax_2} & \mathbf{stl}\ (\mathbf{str}\ (p, f)\ s) & = & \mathbf{str}\ (p, f)\ (f\ s) \end{array}$$

Figure 2: Axiomatic Stream Definition: Laws

- and to handle *synchronous* as well as *asynchronous* stream functions, both kind being definable as **fstr** expressions.

The rest of the paper is organized as follows: the second section defines streams and how we prove equality between streams; the third defines the function $\mathbf{fstr}_k$, its fusion law and the definitions of the stream functions we will need with $\mathbf{fstr}_k$ combinators; the fourth section defines the kind of system of stream equations we allow; the fifth section illustrates the use of the method; the sixth section gives the syntactic condition under which our method succeeds and the kind of result we build; the seventh section gives an overview of the method; the eighth section compares the method with previous works and the ninth section concludes.

2. Streams

First, we give an axiomatic definition of streams. Then we show how to prove equality of streams.

2.1. Definition

We do not define a specific implementation for streams. Instead, we assume there is a type constructor Str such that Str a is the type of streams whose values are of type a.

To completely specify streams, we need some functions and some axioms on these functions. The types of the functions are defined in Figure 1. The function **str** builds a stream, the function **shd** takes the head of a stream and the function **stl** takes its tail. The two axioms given in Figure 2. The axiom (Ax_1) says that the head of the stream produced by the automaton $\mathbf{str}\ (p, f)\ s$ is $p\ s$; the axiom (Ax_2) says that its tail is $\mathbf{str}\ (p, f)\ (fs)$. All the laws that we will need come for free from the use of parametricity on their types and from these axioms.

Informally, the expression $\mathbf{str}\ (p, f)\ s :: \mathrm{Str}\ \mathsf{a}$ encodes the following stream ($t :: T$ is the Haskell notation for *the term t of type T*):

$$p\,s,\ p\,(f\,s),\ p\ (f^2\,s), \ldots,\ p\,(f^k\,s), \ldots$$

The expression $s :: \alpha$ defines the initial state; the function $f :: \alpha \to \alpha$ defines a transition from an internal state to the next one; and the observer $p :: \alpha \to \mathsf{a}$ observes the current value of an internal state. Given a stream expression $u :: \mathrm{Str}\ \mathsf{a}$, the only thing we can do is to apply the head function **shd** or the tail function **stl** to u and there is no way to observe the type of the internal state; this encapsulation ensures the validity of our method.

A possible implementation in Haskell is given in Appendix B.

2.2. Equality

Two streams are equal if they have the same values, that is, if we can not distinguish them by observing their values. One way to prove that two streams are equal is to use the take lemma ([1]) or the approximation lemma ([14]); one other way is to use coinduction ([10]).

In this paper, we will only use a simple fact that derives from parametricity to prove equality between streams: if the ways of building two streams are in a specific relation, then they are equal. Indeed, bisimulation is provable from parametricity as shown in [24] for any codatatype.

A *stream equality relation* (*SER*) is similar to a bisimulation relation for streams. Two streams are related if their *initial states* are related and, if their *next state* operations respect this relation and, if the observation of two related *states* yields the same value.

Definition 1. *Given two pairs of functions* $(p_u, f_u) :: (\mathtt{b} \to \mathtt{a}) \times (\mathtt{b} \to \mathtt{b})$ *and* $(p_v, f_v) :: (\mathtt{c} \to \mathtt{a}) \times (\mathtt{c} \to \mathtt{c})$*, a relation* $\mathcal{S} \subseteq \mathtt{b} \times \mathtt{c}$ *is a* stream equality relation *or* SER *if*

$$\mathcal{S}(x, y) \Rightarrow (p_u\, x = p_v\, y \wedge \mathcal{S}(f_u\, x, f_v\, y))$$

holds.

If there exists a *SER* between two streams, the parametricity theorem tells us that they are equal.

Theorem 1. *Given two streams* $\widetilde{u} = \mathbf{str}\ (p_u, f_u)\ \ s_u$ *and* $\widetilde{v} = \mathbf{str}\ (p_v, f_v)\ \ s_v$*; if there exists a* SER $\mathcal{S}$ *such that* $\mathcal{S}(s_u, s_v)$ *then* $\widetilde{u} = \widetilde{v}$*.*

Proof. see Appendix C □

3. A generic and fusible stream function

Using combinators to define functions has been a successful approach ([8], [26], [5], [7]). So, we define a family of combinators $\mathbf{fstr}_k$ that consumes k streams to produce one stream and we give its fusion law.

3.1. Definition

The definition of the function $\mathbf{fstr}_1$ is:

$$\begin{array}{lll}
\mathbf{fstr}_1 & :: & (\forall\alpha.(\alpha \to \mathtt{a}, \alpha \to \alpha) \to \phi\,\mathtt{a} \to \mathtt{b}) \to \\
& & (\forall\alpha.(\alpha \to \mathtt{a}, \alpha \to \alpha) \to \phi\,\mathtt{a} \to \phi\,\mathtt{a}) \to \\
& & (\forall\alpha.(\alpha \to \mathtt{a}, \alpha \to \alpha) \to \mathtt{a} \to \phi\,\mathtt{a}) \to \\
& & \mathrm{Str}\ \mathtt{a} \to \mathrm{Str}\ \mathtt{b} \\
\mathbf{fstr}_1\ \sigma_1\ \sigma_2\ \sigma_3\ s & := & \mathbf{str}\ \ (\sigma_1(\mathbf{shd}, \mathbf{stl}),\ \sigma_2(\mathbf{shd}, \mathbf{stl}))\ \ (\sigma_3(\mathbf{shd}, \mathbf{stl})s)
\end{array}$$

where ϕ is any type constructor (for example, ϕ can be the type constructor $\Lambda x.(\mathbf{Int}, x)$ yielding $\phi\,\mathtt{a} = (\mathbf{Int}, \mathtt{a})$.

The point is that σ_1, σ_2 and σ_3 are polymorphic over α and are parametrized on how to destroy the stream arguments; thus, fusion holds. We can define $\mathbf{fstr}_2$ that consumes 2 streams, and, $\mathbf{fstr}_k$ that consumes k streams in the same way.

3.2. Fusion

Functions like mapping, filtering and merging streams are definable with the combinator $\mathbf{fstr}_k$. The following generic theorem gives a fusion law for any suitable composition of these functions. Thus, instead of building a stream and applying to it a sequence of functions, we can directly build the expected result as a simple **str** expression.

Theorem 2.

$$\begin{array}{c}\mathbf{fstr}_1\ \ \sigma_1\ \sigma_2\ \sigma_3\ (\mathbf{str}\ (p,f)\ s)\\ =\\ \mathbf{str}\ (\sigma_1(p,f),\ \sigma_2(p,f))\ (\sigma_3(p,f)\ s)\end{array}$$

Proof. See Appendix D. □

Note that the only (implicit) hypothesis is that terms are well-typed; thus, the terms σ_1, σ_2 and σ_3 are polymorphic and parametricity allows to draw the proof of this theorem. Informally, the left-hand-side means: build the stream **str** $(p, f)\ s$; then, destroy it with σ_1, σ_2 and σ_3 using the stream destructors **shd** and **stl**. Now, because σ_1,σ_2 and σ_3 are suitably polymorphic, we can avoid building the stream **str** $(p, f)\ s$: we just need to use σ_1, σ_2 and σ_3 with the producers p and f instead of the destructors **shd** and **stl**.

There is an equivalent theorem for any function $\mathbf{fstr}_k$.

Let us see how the fusion works on an example. Let $\widetilde{u}$ and $\widetilde{v}$ be two streams of integers defined as

$$\begin{array}{lcl}\widetilde{u} & :: & \mathrm{Str}\ \mathbf{Int}\\ \widetilde{u} & := & \mathbf{str}\ ((*4),\ (+2))\ 0\\ \\ \widetilde{v} & :: & \mathrm{Str}\ \mathbf{Int}\\ \widetilde{v} & := & \mathbf{str}\ ((*5),\ (+3))\ 1\end{array}$$

So, the stream $\widetilde{u}$ is the sequence $0, 8, 16, 24, 32, 40, \ldots$ and the stream $\widetilde{v}$ is the sequence $5, 20, 35, 50, 65, 80, \ldots$.

$$\begin{array}{lcl}h & :: & (\mathrm{Str}\ \mathbf{Int} \times \mathrm{Str}\ \mathbf{Int}) \to \mathrm{Str}\ \mathbf{Int}\\ h & := & \mathbf{fstr}_2\ \ h_1\ h_2\ h_3\\ & & \text{where}\ \begin{array}{lcl} h_1 & := & \lambda((p_1,f_1),(p_2,f_2)).\ \lambda(x_1,x_2).\ p_1 x_1\ +\ p_2\ (f_2\ x_2)\\ h_2 & := & \lambda((p_1,f_1),(p_2,f_2)).\ \lambda(x_1,x_2).\ (f_1\ (f_1\ x_1),\ f_2\ x_2)\\ h_3 & := & \lambda((p_1,f_1),(p_2,f_2)).\ \lambda(x_1,x_2).\ (x_1,x_2)\end{array}\end{array}$$

We define $\widetilde{w}$ as the application of the stream function h to the streams $\widetilde{u}$ and $\widetilde{v}$. By applying the fusion theorem, we get a definition of $\widetilde{w}$ as a simple **str** expression:

$$\begin{array}{ll} & \widetilde{w}\\ = & h\ (\widetilde{u}, \widetilde{v})\\ & \{\text{inline definitions of } h,\ \widetilde{u},\ \widetilde{v}\ \}\\ = & \mathbf{fstr}_2\ h_1\ h_2\ h_3\ \begin{array}{ll}(& \mathbf{str}\ ((*4),\ (+2))\ 0\\ , & \mathbf{str}\ ((*5),\ (+3))\ 1\\ ) & \end{array}\\ & \{\text{fstr fusion (Theorem 2) and arithmetic simplifications}\ \}\\ = & \mathbf{str}\ (\lambda(x,y).4*x+5*(y+3),\ \lambda(x,y).(x+4,\ y+3))\ (0,1)\end{array}$$

So, the stream $\widetilde{w}$ is the sequence $20, 51, 82, 113, 144, 175, 206, 237, 268, 299, \ldots$.

$$
\begin{array}{lll}
(:) & :: & \mathtt{a} \to \mathrm{Str}\ \mathtt{a} \to \mathrm{Str}\ \mathtt{a} \\
(h\,:) & = & \mathbf{fstr}_1\ \sigma_1\ \sigma_2\ \sigma_3 \\
 & & \text{where}\ \begin{array}{lll} \sigma_1\ (p,f) & = & \lambda(v,s).v \\ \sigma_2\ (p,f) & = & \lambda(v,s).(p\ s, f\ s) \\ \sigma_3\ (p,f) & = & \lambda s.(h,s) \end{array}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{tl} & :: & \mathrm{Str}\ \mathtt{a} \to \mathrm{Str}\ \mathtt{a} \\
\mathbf{tl} & = & \mathbf{fstr}_1\ \sigma_1\ \sigma_2\ \sigma_3 \\
 & & \text{where}\ \begin{array}{lll} \sigma_1\ (p,f) & = & p \\ \sigma_2\ (p,f) & = & f \\ \sigma_3\ (p,f) & = & f \end{array}
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{map_2} & :: & (\mathtt{a} \to \mathtt{b} \to \mathtt{c}) \to (\mathrm{Str}\ \mathtt{a} \times \mathrm{Str}\ \mathtt{b}) \to \mathrm{Str}\ \mathtt{c} \\
\mathbf{map_2}\, g & = & \mathbf{fstr}_2\, \sigma_1\ \sigma_2\ \sigma_3 \\
 & & \text{where}\ \begin{array}{lll} \sigma_1\ ((p_1,f_1),(p_2,f_2)) & = & \lambda(x,y).\ g\ (p_1\ x)\ (p_2\ y) \\ \sigma_2\ ((p_1,f_1),(p_2,f_2)) & = & \lambda(x,y).(f_1\ x, f_2\ y) \\ \sigma_3\ ((p_1,f_1),(p_2,f_2)) & = & \mathbf{id} \end{array}
\end{array}
$$

Figure 3: Stream functions

3.3. Standard functions as fstr

We define stream functions that we will need using the $\mathbf{fstr}_k$ function, see Figure 3. The function (:) appends an element in front of a given stream (we will not use the Haskell function (:) on lists so that there will be no conflict); the expression $(h\,:)$ is the Haskell shortcut for $\lambda x.\ h : x$. The function $\mathbf{map_1}$ applies a function to each element of a stream; the function $\mathbf{map_k}$ takes a tuple of k stream arguments instead of one.

Other functions may be defined using $\mathbf{fstr}_k$: synchronous functions as well as asynchronous functions like **filter** and **merge**. Here, we call synchronous function a function from n streams into one stream that consumes one value of each of its stream arguments to produce one value of its stream output. The main benefit to define them using the $\mathbf{fstr}_k$ function is that their fusion laws come *for free* from Theorem 2.

4. Systems of Stream Equations

We consider only systems where equations define streams, that is, we do not allow equations that define functions or expressions of other types. Such a system, under a syntactic condition, that we will see later, has one unique solution.

We use a new syntactic construction *system ... where*.

Definition 2. *A stream equation system is an expression*

$$
\begin{array}{llll}
\mathit{system}\ \widetilde{y}_o & & & \\
\mathit{where} & \widetilde{y}_1 & = & \widetilde{u}_1 :: \mathrm{Str}\ A_1 \\
 & & \vdots & \\
 & \widetilde{y}_k & = & \widetilde{u}_k :: \mathrm{Str}\ A_k
\end{array}
$$

where $o \in [1 \ldots k]$ and $\widetilde{u}_i$ (for $i \in [1 \ldots k]$) is a valid stream.

The definitions inside the system have to be stream definitions. A stream is *valid* (in a given

$$
\begin{array}{lcl}
\widetilde{x} \in V_{system} & \vdash & \widetilde{x} \\
\widetilde{u} & \vdash & \mathbf{tl}\,\widetilde{u} \\
\widetilde{u} & \vdash & a : \widetilde{u} \\
\widetilde{u}_1, \ldots, \widetilde{u}_k & \vdash & \mathbf{map_k}\, f\ (\widetilde{u}_1, \ldots, \widetilde{u}_k) \\
 & \vdash & \mathbf{str}\ (p, f)\ s \\
\widetilde{u}_1, \ldots, \widetilde{u}_k & \vdash & \mathbf{fstr}_k\ \sigma_1\ \sigma_2\ \sigma_3\ (\widetilde{u_1}, \ldots, \widetilde{u_k})
\end{array}
$$

where V_{system} is the set of stream variables bound by the *system*,
the stream expressions $\widetilde{u}_k$ in the premises of the rules are *valid* stream,
the other expressions can not use stream variables defined in the *system*,
and expressions are assumed well-typed

Figure 4: inference rules for valid stream equations

system) if it can be built following the inference rules given in Figure 4. Valid streams are defined using only some stream combinators.

Note that a system defines only one stream output and the internal streams used inside the system are not visible outside.

Such a system can be turned into an executable specification (or a valid Haskell expression) by using a *let ... in* Haskell declaration.

Our way of defining stream systems is the same as in *Communication Lifting* (see [27]) except that we have added a new function **fstr** . Thus, our system is more expressive: we can, for example, encode filtering and merging of streams.

5. Illustration

We call **T** the algorithm that transforms a system of stream equations into the automaton solution: a simple **str** expression. Before we explain exactly the hypotheses under which **T** succeeds and which kind of solution is built, we motivate our work by an example.

We take the example of a reset counter that shows how the transformation may be used. The illustration contains three points:

1. From a system of stream equations, our transformation **T** is able to calculate the equivalent $\mathbf{fstr}_k$ expression: we define the reset counter as a function from one stream to one stream.

2. The application of a $\mathbf{fstr}_k$ stream function to **str** stream arguments yields a simple **str** expression by fusion: we apply the reset counter to a specific reset signal.

3. Properties of a stream can be proved by describing the expected observation ([12]) with a system of stream equations; thanks to our transformation (that rewrites the observer as a **str** expression), we only need to exhibit a suitable stream equality relation (*SER*) to complete the proof.

5.1. A stream to stream function: the reset counter

A reset counter is a function that takes a reset signal (a stream) of booleans and outputs, at each time, the value of the counter. At time $t = 0$, the counter is $c_0 = 0$. At time $t \geq 1$, if the reset signal is true at time $t - 1$ then $c_t = 0$; else, if the reset signal is false at time $t - 1$, then $c_t = c_{t-1} + 1$. We can translate this specification into Haskell:

$$
\begin{array}{lcl}
rc & :: & \text{Str } \mathbf{Bool} \to \text{Str } \mathbf{Int} \\
rc\; signal & = & \text{system } \widetilde{c} \\
& & \quad \text{where } \widetilde{c} \;=\; 0 : \widetilde{cn} \\
& & \qquad\quad\;\; \widetilde{cn} \;=\; \mathbf{map_2}\ (\lambda r\ i.\ \text{if } i \text{ then } 0 \text{ else } r+1)\ (\widetilde{c},\ \mathbf{str}\ (\mathbf{shd}, \mathbf{stl})\ \ signal)
\end{array}
$$

Using our transformation $\mathbf{T}$, we are going to compute the $\mathbf{fstr}_k$ expression for the reset counter rc.

Note that we wrote $\mathbf{str}\ (\mathbf{shd}, \mathbf{stl})\ \ signal$ instead of $signal$ to match the definition of a system (Law 1, see Appendix E, ensures that both expressions are equal). Applying the transformation $\mathbf{T}$, we get the following $\mathbf{str}$ definition of the reset counter rc:

$$
\begin{array}{l}
rc\; signal = \mathbf{str}\ (p_{rc}, f_{rc})\ (s_{rc}\; signal) \\
\text{where } \begin{array}{lcl}
p_{rc} & = & \pi_1 \\
f_{rc} & = & \lambda(r,s,i).\ \text{let } n = \mathbf{stl}\ s \text{ in } (\text{if } i \text{ then } 0 \text{ else } r+1,\ n,\ \mathbf{shd}\, n) \\
s_{rc} & = & \lambda x.(0,\ x,\ \mathbf{shd}\ \ x)
\end{array}
\end{array}
$$

We just need to abstract the pair $(\mathbf{shd}, \mathbf{stl})$ to get the appropriate $\mathbf{fstr}_1$ definition:

$$
\begin{array}{l}
rc = \mathbf{fstr}_1\ \ \sigma_1\ \sigma_2\ \sigma_3 \\
\text{where } \begin{array}{lcl}
\sigma_1(p,f) & = & \pi_1 \\
\sigma_2(p,f) & = & \lambda(r,s,i).\ \text{let } n = f\ s \text{ in } (\text{if } i \text{ then } 0 \text{ else } r+1,\ n,\ p\, n) \\
\sigma_3(p,f) & = & \lambda s.(0,\ s,\ p\, s)
\end{array}
\end{array} \tag{3}
$$

5.2. Application to a specific signal

Let us now assume that the reset signal is $\widetilde{sig} := \mathbf{false} : \mathbf{false} : \mathbf{false} : \mathbf{true} : \widetilde{sig}$. We want to find the automaton $\widetilde{rcsig} := \mathbf{str}\ (p,f)\ s$ such that

$$\widetilde{rcsig} = rc\ \ \widetilde{sig}$$

We first translate the reset signal $\widetilde{sig}$ with $\mathbf{T}$ to get a $\mathbf{str}$ definition:

$$
\begin{array}{l}
\widetilde{sig} = \mathbf{str}\ (p_{sig}, f_{sig})\ s_{sig} \\
\text{where } \begin{array}{lcl}
p_{sig} & = & \pi_1 \\
f_{sig} & = & \lambda(t0,t1,t2,t3).(t1,t2,t3,t0) \\
s_{sig} & = & (\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{true})
\end{array}
\end{array} \tag{4}
$$

We now apply rc to the signal $\widetilde{sig}$

$$
\begin{array}{rcl}
\widetilde{rcsig} & = & rc\ \ \widetilde{sig} \\
& & \{\text{equation 3 and equation 4}\} \\
& = & \mathbf{fstr}_1\ \ \sigma_1\ \sigma_2\ \sigma_3\ (\mathbf{str}\ (p_{sig}, f_{sig})\ s_{sig}) \\
& & \{\text{fusion}\} \\
& = & \mathbf{str}\ (\sigma_1(p_{sig}, f_{sig}),\ \sigma_2(p_{sig}, f_{sig}))\ (\sigma_3(p_{sig}, f_{sig})\ s_{sig}) \\
& & \{\text{inlining } (\sigma_k)_{k=1,2,3}\} \\
& = & \mathbf{str}\ (p_{rcsig}, f_{rcsig})\ s_{rcsig} \\
& & \text{where } \begin{array}{lcl}
p_{rcsig} & = & \pi_1 \\
f_{rcsig} & = & \lambda(v,(t0,t1,t2,t3),hs). \\
& & \quad \begin{array}{ll}
(& \text{if } hs \text{ then } 0 \text{ else } v+1 \\
, & (t1,t2,t3,t0) \\
, & t1 \\
) &
\end{array} \\
s_{rcsig} & = & (0,(\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{true}), \mathbf{false})
\end{array}
\end{array}
$$

5.3. Verifying stream properties

It is usually nice to prove that our program has the behaviour we intuitively expect; for example, we may want to prove that, at time t, $c_t = t\,\text{`}mod\text{`}\,4$ (where $t\,\text{`}mod\text{`}\,4$ is the Haskell expression that means the integer t modulo 4). It is not obvious from this point how to prove it directly, but, what we want to prove can be specified using streams: we first specify the time with the stream $\widetilde{ts}$; we then specify the value we expect at each instant with the stream $\widetilde{tm4}$ and then we compare all the values of $\widetilde{rcsig}$ with the ones of $\widetilde{tm4}$

$$
\begin{array}{lll}
\widetilde{check} & :: & \text{Str } \mathbf{Bool} \\
\widetilde{check} & := & \text{system } \widetilde{o} \\
 & & \quad \text{where } \begin{array}[t]{lll}
\widetilde{rcsig} & = & \mathbf{str}\ (p_{rcsig},\ f_{rcsig})\ s_{rcsig} \\
\widetilde{ts} & = & 0 : \mathbf{map_1}\ (+1)\ \widetilde{ts} \\
\widetilde{tm4} & = & \mathbf{map_1}\ (\text{`}mod\text{`}\,4)\ \widetilde{ts} \\
\widetilde{o} & = & \mathbf{map_2}\ (==)\ (\widetilde{rcsig}, \widetilde{tm4})
\end{array}
\end{array}
$$

We apply the transformation $\mathbf{T}$ to the definition of $\widetilde{check}$, and, after some reductions of the **str** ingredients, we get:

$$
\begin{array}{l}
\widetilde{check} = \mathbf{str}\ (p_{check}, f_{check})\ s_{check} \\
\text{where } \begin{array}[t]{lll}
p_{check} & = & \lambda(r, \_, \_, y).\ r == y\,\text{`}mod\text{`}\,4 \\
f_{check} & = & \lambda(r,\ (t0, t1, t2, t3), i, y). \\
 & & \quad \begin{array}{ll}
(& \text{if } i \text{ then } 0 \text{ else } r + 1 \\
, & (t1, t2, t3, t0) \\
, & t1 \\
, & y + 1 \\
) &
\end{array} \\
s_{check} & = & \begin{array}{ll}
(& 0 \\
, & (\mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{true}) \\
, & \mathbf{false} \\
, & 0 \\
) &
\end{array}
\end{array}
\end{array}
$$

Now, for our intuition to become a fact, we have to prove that $\widetilde{check} = \mathbf{str}\ (\mathbf{id}, \mathbf{id})\, \mathbf{true}$; that is, we want to prove that all the values of the stream $\widetilde{check}$ are true.

We just need to exhibit a *SER* between $\widetilde{check}$ and $\mathbf{str}\ (\mathbf{id}, \mathbf{id})\ \mathbf{true}$. This last step is the only one that relies on the user: we have to invent a relation.

Let $\mathcal{S} \subseteq (\mathbf{Int} \times \mathbf{Bool}^4 \times \mathbf{Bool} \times \mathbf{Int}) \times \mathbf{Bool}$ defined as:

$$\mathcal{S} := \{((t\,\text{`}mod\text{`}\,4, bs, bs_1, t), \mathbf{true}) \mid t \in \mathbf{Int},\ bs_n := (n == 4 - (t\,\text{`}mod\text{`}\,4))\}$$

We can check that:

- $(\mathcal{S} \to eq_A)(p_{check}, \mathbf{id})$,
- $(\mathcal{S} \to \mathcal{S})(f_{check}, \mathbf{id})$.

Thus, the relation $\mathcal{S}$ is a *SER*. Then, as the initial states are in this *SER* ($\mathcal{S}(s_{check}, \mathbf{true})$), by Theorem 1, we conclude that any value of $\widetilde{check}$ is **true**. So we effectively have:

$$c_t = t\,\text{`}mod\text{`}\,4$$

6. When has a system one unique solution?

In this section, we explain the kind of problems **T** can solve and the properties of its solution. The algorithm **T** itself will be explained in Section 7.

6.1. The kind of solution we are looking for

We define it formally as follows:

Definition 3. *Let* S :: Str $\mathtt{b}$ *be a system of stream equations, let* $\mathtt{a}_1, \ldots, \mathtt{a}_k$ *be the element types of the internal streams of* S*, we say that the system* S *is* solvable *if it has one unique solution and if there exists a n-ary polynomial type constructor* Γ *and three terms* $p : \Gamma\ \mathtt{a}_1 \ldots \mathtt{a}_k \to \mathtt{b}$*,* $f : \Gamma\ \mathtt{a}_1 \ldots \mathtt{a}_k \to \Gamma\ \mathtt{a}_1 \ldots \mathtt{a}_k$ *and,* $s : \Gamma\ \mathtt{a}_1 \ldots \mathtt{a}_k$ *such that* $S = \mathbf{str}\ (p, f)\ s$ *without duplicating redexes with* p*,* f *and* s *non-recursively defined.*

There are three constraints in this definition:

1. We forbid duplication of redexes, that is: we forbid to compute the same value several times.
2. We also forbid to build the solution using recursion: the solution is built using only simple language constructions like abstraction, application, tupling, etc. Thus, the method is valid for other languages and trivial meaningless solutions (like $S = \mathbf{str}\,(\mathbf{shd}, \mathbf{stl})\,S$) are forbidden (because S is defined recursively).
3. And we want the internal state to contain a *finite* number of stream values: thus, if stream values are simple (like integers, booleans), the solution can be run within bounded memory.

6.2. A sufficient syntactic condition

A simple syntactic condition characterizes the class of systems that do have a unique solution.

We focus only on the structure of the set of mutually recursive equations. Thus, a term that defines a stream is composed of two kinds of terms: stream terms and black box terms. This distinction is purely syntactic: a black box term may be a stream. The stream $h : u$ is the stream whose head (a black box) is h and whose tail is u. The stream $\mathbf{map_2}\, f\ (\widetilde{u}, \widetilde{v})$ is the stream whose values are the assembling of black boxes $f\, u_k\, v_k$.

We define two syntactic conditions $\mathcal{C}_1$ and $\mathcal{C}_2$; if these two conditions hold for a system, then the system is solvable, that is, the system has one unique solution and our method computes the solution: a simple **str** expression.

Remember that we restrict ourself to the study of the structure of the system of equations. Thus,

$$\begin{array}{lcl} \text{system } \widetilde{n} & & \\ \text{where } \widetilde{n} & = & \boxed{0} : \mathbf{map_1} \boxed{+1}\, \widetilde{n} \end{array}$$

and

$$\begin{array}{lcl} \text{system } \widetilde{n}' & & \\ \text{where } \widetilde{n}' & = & \boxed{2} : \mathbf{map_1} \boxed{*5}\, \widetilde{n}' \end{array}$$

have the same structure and, independently of the black boxes, we can prove that there is a unique solution for both systems. Now, consider the following system:

$$\begin{array}{lcl} \text{system } \widetilde{v} & & \\ \text{where } \widetilde{v} & = & \mathbf{map_1} \boxed{f}\, \widetilde{v} \end{array}$$

This system is not solvable because the existence of a unique solution depends on what the black box $\boxed{f}$ is.

- If $\boxed{f}$ is the identity function, any stream is a solution of this system or there is a unique solution $\bot{:}\bot{:}\bot{:}\ldots$; this depends on the underlying semantics.
- If $\boxed{f}$ is the constant function $\lambda x.1$, the unique solution is $1:1:1:1:1:1:\ldots$.

We define the condition $\mathcal{C}_1$ that forbids such bad systems. The equation dependencies form a graph. This graph has all the stream variables of a given system as nodes. Let $y = M$ be an equation in this system, there is a dependency edge from y to each occurrence of stream variables inside M. Let the stream variable z occurs in M, the value of the edge $y \to z$ is the number of (:) minus the number of **tl** that we encounter along the syntax tree to go from the root of M to the variable occurrence z. The value of a path is the sum of the values of its edges.

Definition 4. $\mathcal{C}_1$*: there is no cycle whose value is less or equal to* 0.

For example, the only cycle in the system

$$\begin{array}{lcl} \text{system } \widetilde{n} & & \\ \text{where } \widetilde{n} & = & \boxed{?} : \mathbf{map_1} \boxed{?}\ \widetilde{n} \end{array}$$

has for value 1, so, this system has a unique solution. And the only cycle in the system

$$\begin{array}{lcl} \text{system } \widetilde{v} & & \\ \text{where } \widetilde{v} & = & \mathbf{map_1} \boxed{?}\ (\mathbf{tl}\, \widetilde{v}) \end{array}$$

has for value -1, so, this system has no solution.

Let us consider the following trivial bad system:

$$\begin{array}{lcl} \text{system } \widetilde{v} & & \\ \text{where } \widetilde{v} & = & \widetilde{v} \end{array}$$

In Haskell, the reduction of the expression $\mathbf{shd}(\mathbf{stl}^n\ \widetilde{v})$, that is, the n$^{\text{th}}$ value of $\widetilde{v}$ is $\bot$ but this relies on the semantics of Haskell. As there is a cycle on $\widetilde{v}$ of value 0, we reject such a system because, if we don't choose a particular semantics, any stream is a solution of this system.

The criterion $\mathcal{C}_1$ does not handle occurrences of the generic $\mathbf{fstr}_k$ function. Indeed, this function can be used to encode asynchronous functions. By *synchronous function* we mean a function that consumes one value on each of its stream arguments to produce one value of its stream output. A function is *asynchronous* when it is not *synchronous*. For example, the function **filter** that filters stream values (that verifies a given predicate) is asynchronous. To explain the two kinds of problems that may arise with asynchronous functions, let us consider two examples.

We leave untold the exact definition of **filter** which uses $\mathbf{fstr}_1$: its formal definition requires to mix fix-point with parametricity in a consistent way (see [28] for troubles that arises when parametricity and fix-point meet themselves and see [17] for a framework where the stream function **filter** can be defined in Haskell).

First, let E_1 be the following system:

$$\begin{array}{lcl} \text{system } \widetilde{u} & & \\ \text{where } \widetilde{u} & = & 2 : \mathbf{filter}\ \boxed{p}\ \widetilde{u} \end{array} \tag{5}$$

If the black-box $\boxed{p}$ is the predicate that is always true, then the system E_1 has one unique solution: the constant stream of 2's. But, if $\boxed{p}$ is the predicate that is always false, then, in Haskell, the solution is $2:\bot{:}\bot{:}\ldots$. But, again, this relies on the particular semantics of Haskell so we reject such system.

Second, let E_2 be the following system:

$$\begin{array}{lcl} \text{system } \widetilde{v} & & \\ \text{where } \widetilde{w} & = & \mathbf{map_2}\ (\lambda xy.(x,y))\ (\mathbf{filter}\ \boxed{c}\ \widetilde{v})\ \widetilde{v} \\ \widetilde{v} & = & \mathbf{str}\ (\mathbf{id}, (+1))\ 0 \end{array} \tag{6}$$

Here, the stream $\widetilde{v}$ is consumed two times: once through a filter and once directly. If the filter predicate $\boxed{c}$ is the even predicate, then the stream $\widetilde{v}$ will be consumed at two different speeds; thus, we will have to store an unbounded number of values or we will have to duplicate the computation of the stream $\widetilde{v}$. As we do not want to take an arbitrary decision here, we reject such a system. The simple syntactic criterion $\mathcal{C}_2$ forbids such bad systems.

Definition 5. *$\mathcal{C}_2$: for each occurrence $\mathbf{fstr}_k\ \sigma_1\ \sigma_2\ \sigma_3\ (M_1, \ldots, M_k)$, each stream term M_i can be turned into an independent system without duplicating equations.*

To be able to rewrite each M_i as an independent system ensures that they can be replaced by an equivalent **str** expression. For the system (5), because the function **filter** is defined with $\mathbf{fstr}_1$, the stream term $\widetilde{u}$ should be turned into an independent system; this is not possible without duplicating the definition of $\widetilde{u}$. For the system (6), the term argument $\widetilde{v}$ of **filter** can not be inlined because it is used elsewhere. So, $\mathcal{C}_2$ does not hold for both systems. Intuitively, the criterion $\mathcal{C}_2$ forbids only systems where a stream may be consumed at two different speed and systems that may have no solution at all.

6.3. Main Result

Any system of stream equations, for which criteria $\mathcal{C}_1$ and $\mathcal{C}_2$ hold, is solvable and, the transformation $\mathbf{T}$ builds its automaton solution: a **str** expression.

Theorem 3. *A system is solvable if it verifies criteria $\mathcal{C}_1$ and $\mathcal{C}_2$.*

Proof. The transformation $\mathbf{T}$ (see next section) is the constructive proof of this theorem. ☐

Thus, any system for which $\mathcal{C}_1$ and $\mathcal{C}_2$ hold can be used as the definition of a stream $\widetilde{x}$; and, the definition of $\widetilde{x}$ as $\mathbf{str}\ (p, f)\ s$ is automatically extracted from such a system by our transformation $\mathbf{T}$.

In Haskell, that means that the system of equations has one meaningful solution, and, that we provide a way to compute the solution with no more intermediate streams. In general, it means that, if the black-boxes (stream values and functions from stream values to a stream value) are definable in a language $\mathcal{L}$, then the stream is also definable in $\mathcal{L}$ as an automaton.

7. Outline of the method

Here, we do not give a complete formalization of the method. We just sketch the algorithm [2].

Given a system S for which criteria $\mathcal{C}_1$ and $\mathcal{C}_2$ hold, the algorithm $\mathbf{T}$ finds the **str** expression solution of the system S.

The algorithm $\mathbf{T}$ works as follows: it identifies a recursive component rc, that is, a minimal subset of equations that we can not break down; then, it calculates the solution of this component rc and updates the original system accordingly; we repeat these two steps until we reach the solution.

7.1. Recursive component identification

Let G be the graph of dependencies between the equations of a system S: there is a node for each stream of the system and there is an edge from the stream (node) x to the stream (node) y for each occurrence of the term x in the equation that defines y. Let RG be the reduced graph of G, that is, the graph of the strongly connected components of G. A *trivial component* is a node of RG that

[2] A technical report is available online at `http://lifc.univ-fcomte.fr/~barbier/jfla-proofs.ps`. The document is written in French (it's a part of my PhD thesis).

contains only one stream (node) x with x not defined recursively (i.e. no edge in G from x to x). A *non-trivial* component is just a component that is not a trivial component.

Now, since the reduced graph RG is acyclic, there is a set of minimal non-trivial components. A *recursive component* contains the minimal non-trivial components that depend on the same trivial components and these latter trivial components. Formally, to build a set of streams rc that is a recursive component, choose one of the minimal non-trivial components and close it with the two following rules:

1. If c is a component such that $c \subseteq rc$, if c' is a component, and if there is an edge in RG from c' to c, then $c' \subseteq rc$.

2. If $\{x\}$ is a trivial component such that $\{x\} \subseteq rc$, if c is a component such that there is an edge in RG from $\{x\}$ to c , then $c \subseteq rc$.

7.2. Replacing a recursive component by its solution

Let say we have chosen a recursive component rc. Let E be the set of equations such that $S = E \cup rc$. Some streams defined in rc are used in E: let $\widetilde{i}_1$, …, $\widetilde{i}_k$ be the streams defined in rc that occur in E, we add the equation $\widetilde{o} = \mathbf{map_k}\ (\lambda x_1 \ldots x_k.(x_1, \ldots, x_k))\ (\widetilde{i}_1, \ldots, \widetilde{i}_k)$ to rc and we replace each occurrence of $\widetilde{i}_l$ in E by $\mathbf{map_1}\ (\lambda(x_1, \ldots, x_k).x_l)\ \widetilde{o}$.

As we will see after, rc has one solution $\mathbf{str}\ (p, f)\ s$. Thus, we can define S' as $E \cup \{\widetilde{o} = \mathbf{str}\ (p, f)\ s\}$.

7.3. Recursive component solution

We show now how to compute the solution of a recursive component as defined above (see 7.1). We reach the solution in several steps. We illustrate each step except the two first ones on the example (7).

$$\begin{array}{lcl} \text{system } \widetilde{u} & & \\ \text{where } \widetilde{u} & = & \mathbf{map_2}\ (+)\ (0 : \widetilde{u}, 0 : \mathbf{tl}^2\, \widetilde{v}) \\ \qquad\quad \widetilde{v} & = & 1 : 2 : \mathbf{map_2}\ (+)\ (\widetilde{u}, \mathbf{tl}\, \widetilde{v}) \end{array} \tag{7}$$

Remove **fstr**
By the hypothesis $\mathcal{C}_2$, each argument of the function **fstr** is defined as an independent graph. We can take this subgraph as a recursive component and solve it first; thus, we translate each **fstr** arguments into a **str** expression. Then, by fusion, we can get rid of the **fstr** function[3].

Remove **str**
We replace every expression $\mathbf{str}\ (p, f)\ s$ by $\mathbf{map_1}\ p\ \widetilde{x}$ where $\widetilde{x}$ is a new stream variable; and we add the equation $\widetilde{x} = s : \mathbf{map_1}\ f\ \widetilde{x}$ in the system. Law 9 (see Appendix E) ensures the equality of the two expressions.

Equations Ordering
To explain this step, we need some definitions.

[3] See `http://lifc.univ-fcomte.fr/~barbier/jfla-proofs.ps`for a more detailed explanation

We say that an equation is in a *head-map-tail* form if it has the following shape:

$$\widetilde{x} = h_1 : \cdots : h_k : \mathbf{map_l}\ f\ (\mathbf{tl}^m\ \ \widetilde{y}) \ldots\ (\mathbf{tl}^o\ \ \widetilde{w})$$

We call $h_1 : \cdots : h_k$ the prefix of $\widetilde{x}$ and we write $nbp(\widetilde{x})$ for the length of this prefix. We call $\mathbf{tl}^m\,\widetilde{y}$ a reference to the stream $\widetilde{y}$. We say that a reference $\mathbf{tl}^m\,\widetilde{y}$ is *computable* if the length of the prefix of $\widetilde{y}$ is strictly greater than m. We say that an head-map-tail equation is *computable* if for each of its references $\mathbf{tl}^m\,\widetilde{y}$, either $\mathbf{tl}^m\,\widetilde{y}$ is computable, or, $m = nbp(\widetilde{y})$ and the equation $\widetilde{y}$ is computable. Informally, an equation is computable if its next value depends only on computed values or on values that can be computed without unfolding.

We can now explain this step: its aim is to rewrite the system so that each equation is in head-map-tail form and is computable.

Since each equation contains only stream variables, the infix operator (:) and the functions **tl** and **map**, it is easy to rewrite them into head-map-tail forms by unfolding them if needed and by using laws like $\mathbf{tl}\ (\mathbf{map_1}\ \ f\ \widetilde{v}) = \mathbf{map_1}\ \ f\ (\mathbf{tl}\ \widetilde{v})$ and $\mathbf{map_1}\ \ f\ (h : \widetilde{v}) = f\,h : \mathbf{map_1}\ \ f\ \widetilde{v}$.

To unfold them in the right order so that each equation is computable is a more complicated story. But under the hypothesis $\mathcal{C}_1$, it is always possible[4].

For simplicity, we evaluate the value of the heads, but keep in mind that, since stream elements may be of any types (lists, functions, streams, etc.), sharing has to be preserved in a real implementation of this method.

$$\begin{array}{lrcl} \text{system } \widetilde{u} & & & \\ \text{where} & \widetilde{u} & = & 0 : \mathbf{map_2}\ (+)\ (\widetilde{u}, \mathbf{tl}^2\,\widetilde{v}) \\ & \widetilde{v} & = & 1 : 2 : \mathbf{map_2}\ (+)\ (\widetilde{u}, \mathbf{tl}\,\widetilde{v}) \end{array} \tag{8}$$

The Solution
To find the solution, we have to give three terms p, f and s such that $\widetilde{u} = \mathbf{str}\ \ p\ f\ s$. The term s is a tuple of tuples; each equation $\widetilde{x}$ yields a tuple of values: the values of its prefix. Here, we get:

$$s = ((0), (1, 2))$$

The term p is just the suitable projection on this tuple to get the first value of the output. Here, we get:

$$p = \lambda((u_0), (v_1, v_2)).u_0$$

The term f is a little more complicated. Imagine we unfold each equation of a system. Note that, since the equations are ordered and because the condition $\mathcal{C}_1$ holds, the new values are directly computable from old ones if we follow the right order (here, $v_2 = u_0 + v_1$, then $u_1 = u_0 + v_2$). Here, we get:

$$\begin{array}{lrcl} \text{system } \widetilde{u} & & & \\ \text{where} & \widetilde{u} & = & 0 : 2 : \mathbf{map_2}\ (+)\ (\mathbf{tl}\,\widetilde{u}, \mathbf{tl}^3\,\widetilde{v}) \\ & \widetilde{v} & = & 1 : 2 : 2 : \mathbf{map_2}\ (+)\ (\mathbf{tl}\,\widetilde{u}, \mathbf{tl}^2\,\widetilde{v}) \end{array} \tag{9}$$

Let say that we are now only interested by $\widetilde{u}' := \mathbf{tl}\,\widetilde{u}$; by construction, we can do the same for each equation. Here we get:

$$\begin{array}{lrcl} \text{system } \widetilde{u}' & & & \\ \text{where} & \widetilde{u}' & = & 2 : \mathbf{map_2}\ (+)\ (\widetilde{u}', \mathbf{tl}^2\,\widetilde{v}') \\ & \widetilde{v}' & = & 2 : 2 : \mathbf{map_2}\ (+)\ (\widetilde{u}', \mathbf{tl}\,\widetilde{v}') \end{array} \tag{10}$$

[4]The proof and an algorithm can be found in `http://lifc.univ-fcomte.fr/~barbier/jfla-proofs.ps`

This system has exactly the same shape as the original system except for the values of its prefixes. We could define a state s'; the observer remains the same p and yields the first value of $\widetilde{u}'$ which is also the second value of $\widetilde{u}$. The term f is just the function that computes directly s' from s. Here, we get:

$$f = \lambda((u_0), (v_0, v_1)).\ \mathrm{let}\ v_2 = u_0 + v_1\ \mathrm{in}\ ((u_0 + v_2), (v_1, u_0 + v_1))$$

8. Related Works

Our work relates to different topics such as stream transformation and compiling, fusion and deforestation of functional programs and, categorical or algorithmic calculations based on the coalgebraic definition of streams.

Much work has been done on stream compilation. Here we just consider two of them which are the closest from our work since they both relate to functional definitions of streams. First, our aim is very similar to the one of Willem G. Vree and Pieter P. Hartel in *Communication Lifting* [27]. We have used the same language on streams but added the function **fstr** which allows to define more stream functions (for example, in their framework, it was impossible to use asynchronous stream functions). We actually recast their transformation using a coalgebraic definition of streams. This allows us to have a modular transformation in which asynchronous functions integrate naturally. We also made the deliberate choice to fail on a kind of input (when there is a cycle whose value is less than one) whereas their transformation always succeeds but possibly leaving residual recursions at the end. Apart from that, their result has also the shape of an iteration that is observed at each step even if they do not capture this pattern in a stream combinator. Secondly, Paul Caspi and Marc Pouzet in *Synchronous Kahn Network* [2] and in *A Co-iterative Characterisation of Synchronous Stream Functions* [3] are interested in verifying and compiling programs of the specific domain language Lustre. Lustre is a higher-order stream language but the elements of the streams have to be simple (the value of an element of the stream may be used at compile time to compute the clock of a stream) whereas we start from a first-order specification and the element of our streams may range from integer to infinite trees or functions. They also do not have an equivalent for the function **tl**. They infer the synchronicity of the stream network by a clock-calculus and they use an other transformation for compiling the network. Their aim being to produce a proved synchronous and efficient program whereas we aim at extracting the automaton producing the stream solution of a system of equations. Our framework is simpler: it just requires parametricity, and, the automaton produced (similar to the result of the compilation) is defined in the language itself. The drawback of this simplicity is that we just do a *naïve* job about asynchronicity: streams have to be synchrone or completely independent for our method to succeed.

Marteen M. Fokkinga, in his PhD Thesis *Law and Order in Algorithmics* [5], presents a lot of laws that holds in a categorical setting. Among these laws, there is the *transformer of di-algebras*; the fusion (see Theorem 2) can be seen as an instantiation of this law; our merit is to recast it into the context of a real programming language where the categorical properties do not always hold.

Finally, the transformation may also be seen as an extraction of an anamorphism from a recursive definitions. Tim Sheard and John Launchbury in *Warm Fusion: Deriving Build-Catas from Recursive Definitions* [18] and, Zhenjiang Hu, Hideya Iwasaki and Masato Takeichi [13] in *Deriving Structural Hylomorphisms from Recursive Definitions* explain how to extract hylomorphisms from recursive definitions of functions. Our work may be seen as a trivial extraction because we only consider one type (streams). But, our work is concerned by the extraction of anamorphisms from a recursive value instead of a recursive function; furthermore, the non-existence of an initial algebra on streams forbids the use of hylomorphisms. More importantly, the condition under which our method succeeds is syntactic and human-readable.

Richard B. Kieburtz in [15] advocates the direct use of coalgebras for defining streams because they enjoy many nice properties; in Coq and Isabelle (see respectively [9] and [21]), streams are also required to be defined coinductively. Hutton also notices the under-use of the unfold combinators and explain the laws. Our work may be seen as complementary: our method allows to keep the specification intuitive and modular by using recursive equations; then, the transformation provide an equivalent definition as an automaton (a **str** expression) that fits these frameworks so that we can apply their method afterwards. Furthermore, proofs can be simplified by using our transformation as a rewriting strategy as shown in the reset counter example.

9. Conclusion

We gave a family of generic functions $\mathbf{fstr}_k$ from k streams to one stream; composition of such functions can be fused. All the functions we used and many others are definable following this pattern. We also have presented a transformation that computes the automaton solution of a system of mutually recursive stream equations. The transformation preserves laziness and succeeds under a intuitive and practical syntactic condition. Moreover, the state of the automaton is just a tuple of stream values. Thus, the transformation is a bridge between mutually recursive streams that are intuitive and modular and coiterative definitions, that are suitable for compilation or reasoning.

A first implementation of this transformation in Haskell allowed us to check the examples given in this paper; we are developing a second one: it uses F_ω to represent terms and will allow us to target different languages like ML, OCaml, Haskell, etc. All the examples of this article and many others like the Ackerman numbers, the microhawk architectures, etc. have been automatically transformed by our first implementation of the method.

We are currently working on two possible extensions of the method. The first is to study how the transformation is still useful when asynchronous functions appear anywhere; the result is then modulo simple equations. The second is to formalize the way the method interacts nicely with the ST monad of the functional language Haskell (the ST monad allows the user to manipulate references to the memory in a proven safe way).

Acknowledgment

I am grateful to John Launchbury who explained parametricity to me and improved our transformation by his suggestions and questions, Françoise Bellegarde for the idea and for proof-reading so many drafts, John Matthews, Sava Krštic, Magnus Carlsson, Richard Kieburtz for their discussions and explanations, the whole PacSoft team of the Oregon Graduate Institute (USA) for the exciting and fruitful environment they provide and Paul Caspi for helping me understand what our method can do (and what it can not do).

Bibliographie

[1] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988. Japanese translation, 1991. Dutch translation, 1991. German translation, 1992.

[2] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. *ACM SIGPLAN Notices*, 31(6):226–238, June 1996.

[3] Paul Caspi and Marc Pouzet. A co-iterative characterisation of synchronous stream functions. In *Workshop on Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Lecture Notes in

Theoretical Computer Science, Lisbon, Portugal, March 1998. Long version available as Technical Report 07, VERIMAG.

[4] Coquand and Huet. The calculus of constructions. *INFCTRL: Information and Computation (formerly Information and Control)*, 76, 1988.

[5] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[6] Jeremy Gibbons and Graham Hutton. Proof methods for structured corecursive programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, Stirling, Scotland, August 1999.

[7] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept 1998*, pages 273–279. ACM Press, New York, 1998.

[8] A. Gill, Launchbury J., and Peyton Jones S.L. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[9] E. Giménez. A tutorial on recursive types in Coq. Distribué comme partie de la documentation du système Coq., Mars 1996.

[10] Andrew Gordon. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pages 78–95. Springer Workshops in Computing, 1995.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96, 1993.

[13] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA., May 1996. ACM Press.

[14] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, August 2001.

[15] Richard B. Kieburtz. Reactive functional programming. Technical Report CSE-97-008, Oregon Graduate Institute of Science and Technology, 1997.

[16] Richard B. Kieburtz. Coalgebraic techniques for reactive functional programming. In *Journées Francophones des Languages Applicatifs*, pages 131–158. INRIA, 2000.

[17] John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *European Symposium of Programming*, volume 1058 of *Lecture Notes in Computer Science*. Springer, 1996.

[18] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*. ACM Press, 1995.

[19] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.

[20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[21] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report 304, University of Cambridge, Computer Laboratory, July 1993.

[22] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, April 1997.

[23] Dusko Pavlović and Martín Escardó. Calculus in coinductive form. In V. Pratt, editor, *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 408–417. IEEE Computer Society, 1998.

[24] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[25] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[26] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 306–316. ACM Press, New York, 1995.

[27] Willem G. Vree and Pieter H. Hartel. Communication lifting: fixed point computation for parallelism. *Journal of Functional Programming*, 5(4):549–581, October 1995.

[28] Philip Wadler. Theorems for free! In *FPCA'89, London, England*, pages 347–359. ACM Press, September 1989.

[29] Gavin C. Wraith. A note on categorical datatypes. In D. Pitts et al, editor, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

A. Parametricity

We just give the scheme to derive relations from types and the axiom of parametricity for binary relations. For an introduction to paramatricity, see *Theorems for free* [28]. For a way to make it works with Haskell, see *Parametricity and Unboxing with Unpointed Types* [17].

The scheme $\mathcal{P}$ builds a relation from a type. To define $\mathcal{P}$, we need an environment σ which, for each type variable $\mathtt{x}$ gives three things: a relation variable $\mathcal{X}$ and two type variables $\mathtt{x}_1$ and $\mathtt{x}_2$. We write $\sigma, \mathtt{x} \leftarrow (\mathcal{X}, \mathtt{x}_1, \mathtt{x}_2)$ for the environment that contains σ and the binding $\mathtt{x} \leftarrow (\mathcal{X}, \mathtt{x}_1, \mathtt{x}_2)$. We write M and N for arbitrary types, t, t_1 and t_2 for arbitrary terms.

Given M a type and σ an environment, the binary relation $\mathcal{P}\,[\![M, -, -]\!]_\sigma$ is a relation between a term of type $M\sigma[1]$ and a term of type $M\sigma[2]$ where $M\sigma[i]$ is defined as follows:

$$\begin{array}{ll}
\mathtt{x}\sigma[1] & := \mathtt{y}_1 \text{ if } \mathtt{x} \leftarrow (\mathcal{Y}, \mathtt{y}_1, \mathtt{y}_2) \in \sigma \\
\mathtt{x}\sigma[2] & := \mathtt{y}_2 \text{ if } \mathtt{x} \leftarrow (\mathcal{Y}, \mathtt{y}_1, \mathtt{y}_2) \in \sigma \\
\mathtt{x}\sigma[i] & := \mathtt{x} \text{ if } \mathtt{x} \notin \sigma \\
(M \to N)\sigma[i] & := M\sigma[i] \to N\sigma[i] \\
(\forall \mathtt{x}.M)\sigma[i] & := \forall \mathtt{x}.M\sigma[i]
\end{array}$$

We can now define the relation $\mathcal{P}\,[\![M, -, -]\!]_\sigma \subseteq M\sigma[1] \times M\sigma[2]$:

$$
\begin{array}{ll}
\mathcal{P}\,[\![\mathtt{x}, t_1, t_2]\!]_\sigma & := \mathcal{X}\ t_1\ t_2 \ \text{ if } \mathtt{x} \leftarrow (\mathcal{X}, \mathtt{x}_1, \mathtt{x}_2) \in \sigma \\
\mathcal{P}\,[\![\mathtt{x}, t_1, t_2]\!]_\sigma & := t_1 = t_2 \ \text{ if } \mathtt{x} \notin \sigma \\
\mathcal{P}\,[\![M \to N, t_1, t_2]\!]_\sigma & := \forall x_1 :: M_{\sigma[1]},\ x_2 :: M_{\sigma[2]}.\ \mathcal{P}\,[\![M, x_1, x_2]\!]_\sigma \Rightarrow \mathcal{P}\,[\![N, t_1\ x_1, t_2\ x_2]\!]_\sigma \\
\mathcal{P}\,[\![\forall \mathtt{x}.M, t_1, t_2]\!]_\sigma & := \forall \mathtt{x}_1, \mathtt{x}_2.\forall \mathcal{X} \subseteq \mathtt{x}_1 \times \mathtt{x}_2.\mathcal{P}\,[\![M, t_1\ \mathtt{x}_1, t_2\ \mathtt{x}_2]\!]_{\sigma, \mathtt{x} \leftarrow (\mathcal{X}, \mathtt{x}_1, \mathtt{x}_2)}
\end{array}
$$

The parametricity is the following axiom:

Axiom 1. *Given M a type and u a term of type M, then $\mathcal{P}\,[\![M, u, u]\!]_\emptyset$ is true.*

B. A possible implementation of streams

```
module Str_hlist (Str, str, shd, stl) where

type Str a = [a]
  -- specifications
str  :: (z->a, z->z) -> z -> Str a
shd  :: Str a -> a
stl  :: Str a -> Str a
  -- definitions
str (p,f) = rec where rec s = p s : rec (f s)
shd (h:_) = h
stl (_:t) = t
```

C. *SER* relations and equality

Proof. (Theorem 1) The type of the function **str** is

$$\forall \mathtt{a}.\forall \alpha.(\alpha \to \mathtt{a}) \to (\alpha \to \alpha) \to \alpha \to \mathrm{Str}\ \mathtt{a}$$

we deduce by applying parametricity that

$$
\begin{array}{l}
\forall \mathcal{X} \subseteq \alpha \times \alpha'.\forall p, p'. \\
\qquad (\mathcal{X} \to eq_{\mathtt{a}})(p, p') \Rightarrow \forall f, f'. \\
\qquad (\mathcal{X} \to \mathcal{X})(f, f') \Rightarrow \forall s, s'. \\
\qquad\qquad \mathcal{X}(s, s') \Rightarrow \mathbf{str}\, p f s = \mathbf{str}\, p' f' s'
\end{array}
\tag{11}
$$

Taking $\mathcal{X} = \mathcal{S}$, all the assumptions hold by the definition of a *SER* relation then

$$\mathbf{str}\ \ p_u\ \ f_u\ \ s_u = \mathbf{str}\ \ p_v\ \ f_v\ \ s_v$$

is true; thus, $\widetilde{u} = \widetilde{v}$. ◻

D. Fusion

From Theorem 1, we prove a more usual way to compare streams: two streams are equal if they have the same elements.

Proposition 1. *Given two streams $\widetilde{u}$ and $\widetilde{v}$ such that $\widetilde{u} = \mathbf{str}\ (p_u, f_u)\ \ s_u$ and $\widetilde{v} = \mathbf{str}\ (p_v, f_v)\ \ s_v$ then,*

$$(\forall n.\, p_u({f_u}^n\ s_u) = p_v({f_v}^n\ s_v)) \Rightarrow \widetilde{u} = \widetilde{v}$$

Proof. Take $\mathcal{X} \subseteq \gamma \times \gamma'$ (where $s_u :: \gamma$ and $s_v :: \gamma'$) to be the relation

$$\{(y, y') \mid \exists n. y = {f_u}^n\ s_u \wedge y' = {f_v}^n\ s_v\}$$

it is easy to check that $(\mathcal{X} \to eq_{\mathtt{a}})(p_u, p_v)$ and $(\mathcal{X} \to \mathcal{X})(f_u, f_v)$ hold; thus, $\mathcal{X}$ is a *SER* relation. Then, as the initial states are in the relation $\mathcal{X}(s_u, s_v)$, we deduce that both streams are equal by applying Theorem 1. □

Now, we can prove the fusion theorem.

Proof. (Theorem 2) Let n be a number, let σ^n be defined as:

$$\sigma^n(\mathbf{shd}, \mathbf{stl}) := \sigma_1(\mathbf{shd}, \mathbf{stl}) \circ (\sigma_2(\mathbf{shd}, \mathbf{stl}))^n \circ \sigma_3(\mathbf{shd}, \mathbf{stl})$$

the type of σ^n is

$$\forall \alpha. (\alpha \to \mathtt{a}) \times (\alpha \to \alpha) \to \alpha \to b$$

We show that

$$\forall i. \sigma^i(p, f) = \sigma^i(\mathbf{shd}, \mathbf{stl}) \circ \mathbf{str}\ (p, f)$$

This will prove the theorem by applying Proposition 1.

Let i be arbitrary fixed. Let $\mathcal{X} \subseteq \gamma \times \mathrm{Str}\ \mathtt{a}$ be the relation defined by

$$\{(s, \mathbf{str}\ (p, f)\ s) \mid \forall s : \gamma\}$$

It is easy to check that $(\mathcal{X} \to eq_{\mathtt{a}})(p, \mathbf{shd})$ and that $(\mathcal{X} \to \mathcal{X})(f, \mathbf{stl})$. By the type of σ^i and parametricity, we conclude that

$$\forall i.\ \sigma^i(p, f) = \sigma^i(\mathbf{shd}, \mathbf{stl}) \circ \mathbf{str}\ (p, f)$$

By the definition of σ^n and Proposition 1, the theorem holds. □

E. Laws

Law 1, Law 2, Law 3 and Law 4 define simple equivalences.

Law 1.

$$\mathbf{str}\ (\mathbf{shd}, \mathbf{stl})\ \widetilde{u} = \widetilde{u}$$

Proof. The only thing that we can apply on a stream are the functions **shd** and **stl**. By applying **shd** on the LHS and the axiom (Ax_1), we get $\mathbf{shd}\,\widetilde{u}$; and, by applying **stl** on the LHS and the axiom (Ax_2), we get $\mathbf{stl}\,\widetilde{u}$. □

Law 2.

$$\mathbf{tl}\,\widetilde{u} = \mathbf{stl}\,\widetilde{u}$$

Proof. By definition of **tl**, $\mathbf{tl}\,\widetilde{u} = \mathbf{str}(\mathbf{shd}, \mathbf{stl})(\mathbf{stl}\,\widetilde{u})$; then, Law 1 ends the proof. □

Law 3.

$$\widetilde{u} = \mathbf{shd}\ \widetilde{u} : \mathbf{tl}\ \widetilde{u}$$

Proof. By inlining the definitions of (:) and **tl**, the RHS becomes

$$\mathbf{str}\ (\lambda(v,s).\ v)\ (\lambda(v,s).\ (\mathbf{shd}\,v, \mathbf{stl}\,s))\ (\mathbf{shd}\,\widetilde{u}, \mathbf{stl}\,\widetilde{u})$$

and, by Law1, we know that $\widetilde{u} = \mathbf{str}\ (\mathbf{shd}, \mathbf{stl})\ \widetilde{u}$.

Let $\mathcal{S}$ be the following relation:

$$\mathcal{S} := \{((\mathbf{shd}\ \ x, \mathbf{stl}\ \ x), x\}$$

By axiom (Ax_1) and (Ax_2), S is a SER, and, because $S((\mathbf{shd}\ \ \widetilde{u}, \mathbf{stl}\ \ \widetilde{u}), \widetilde{u})$, both streams are equal. □

Law 4.

$$\mathbf{map_1}\,\mathbf{id}\ \ \widetilde{u} = \widetilde{u}$$

Proof. by Law 1 and inlining $\mathbf{map_1}$, the two expressions are equal. □

The following laws simplifies expressions by rewriting (from left to right). Law 5 removes unneeded heads. Law 6 move the **tl**(s) after the **map**(s). Law 7 extracts the heads. Law 8 fusions successive **map**. This set of laws can be turned into a rewrite system; and when the expression contains only stream variables, (:), **map** and **tl**, the result of the rewriting is an expression:

$$\widetilde{z} \equiv h_1 : \cdots : h_n : \mathbf{map_k}\ \ g\ (\mathbf{tl}^i\,\widetilde{x}, \ldots, \mathbf{tl}^j\,\widetilde{y})$$

Law 5.

$$\mathbf{tl}\ (a : \widetilde{u}) = \widetilde{u}$$

Proof. By applying fusions, the LHS becomes:

$$\mathbf{str}\ (p, f)\ s$$

where
$$\begin{array}{rcl} p &=& \lambda(r,\widetilde{s}).r \\ f &=& \lambda(r,\widetilde{s}).(\mathbf{shd}\,\widetilde{s}, \mathbf{stl}\,\widetilde{s}) \\ s &=& (a,\widetilde{u}) \end{array}$$

The Law 1 gives the **str** expression for the RHS. Then, the following *SER*

$$\mathcal{S} := \{((\mathbf{shd}\,\widetilde{s}, \mathbf{stl}\,\widetilde{s})\ ,\ \ \widetilde{s}\}$$

proves the law by using Theorem 1. □

Law 6.

$$\mathbf{tl}\ (\mathbf{map_k}\,f\ \ (\widetilde{u}_1, \ldots, \widetilde{u}_k)) = \mathbf{map_k}\,f\ \ (\mathbf{tl}\,\widetilde{u}_1, \ldots, \mathbf{tl}\,\widetilde{u}_k)$$

Proof. By applying fusions, the LHS becomes:

$$\mathbf{str}\ (p, f)\ s$$

where
$$\begin{array}{rcl} p &=& \lambda(s_1,\ldots,s_k).f\ (\mathbf{shd}\,s_1)\ldots(\mathbf{shd}\,s_k) \\ f &=& \lambda(s_1,\ldots,s_k).(\mathbf{stl}\,s_1,\ldots,\mathbf{stl}\,s_k) \\ s &=& (\mathbf{stl}\,\widetilde{u}_1,\ldots,\mathbf{stl}\,\widetilde{u}_k) \end{array}$$

and applying fusions, the RHS becomes:

$$\mathbf{str}\ (p', f')\ s'$$

where
$$\begin{array}{rcl} p' &=& p \\ f' &=& f \\ s' &=& s \end{array}$$
□

Law 7.

$$\mathbf{map_k}\, f\ (h_1 : \widetilde{u}_1, \ldots, h_k : \widetilde{u}_k)$$
$$=$$
$$f\ h_1 \ldots\ h_k : \mathbf{map_k}\, f\ (\widetilde{u}_1, \ldots, \widetilde{u}_k)$$

Proof. Let

$$l := \lambda(r, \widetilde{s}).r$$

and

$$g := \lambda(r, \widetilde{s}).(\mathbf{shd}\, \widetilde{s}, \mathbf{stl}\, \widetilde{s})$$

By applying fusions, the LHS becomes:

$$\mathbf{str}\ (p, f)\ s$$

where
$$\begin{aligned} p &= \lambda(s_1, \ldots, s_k).f\ (l\ s_1) \ldots (l\ s_k) \\ f &= \lambda(s_1, \ldots, s_k).(g\ s_1, \ldots, g\ s_k) \\ s &= ((h_1, \widetilde{u}_1), \ldots, (h_k, \widetilde{u}_k)) \end{aligned}$$

and by applying fusions, the RHS becomes:

$$\mathbf{str}\ (p', f')\ s'$$

where
$$\begin{aligned} p' &= \lambda(r, (s_1, \ldots, s_k)).r \\ f' &= \lambda(r, (s_1, \ldots, s_k)).(f\ (\mathbf{shd}\, s_1) \ldots (\mathbf{shd}\, s_k), (\mathbf{stl}\, s_1, \ldots, \mathbf{stl}\, s_k) \\ s' &= (f\ h_1 \ldots h_k), (\mathbf{stl}\, \widetilde{u}_1, \ldots, \mathbf{stl}\, \widetilde{u}_k)) \end{aligned}$$

The *SER* relation

$$\mathcal{S} := \begin{array}{ll} \{(& ((h_1, \widetilde{u}_1), \ldots, h_k, \widetilde{u}_k)) \\ , & ((h_1, \ldots, h_k), (\widetilde{u}_1, \ldots, \widetilde{u}_k)) \quad)\} \end{array}$$

proves the equality. □

Law 8.

$$\mathbf{map_1}\, f\ (\mathbf{map_2}\ \ g\ (\widetilde{u}, \widetilde{v}))$$
$$=$$
$$\mathbf{map_2}(\lambda(x, y).f(g\ x\ y))(\widetilde{u}, \widetilde{v})$$

Proof. by fusion (Theorem 2), the two streams are equal. □

The following law replaces **str** with an equivalent sytem.

Law 9.

$$\mathbf{str}\ (p, f)\ s$$
$$=$$
$$\begin{array}{ll} \mathbf{map_1}\ \ p\ (& \textit{system}\ \widetilde{o} \\ & \textit{where}\ \ \widetilde{o}\quad s : \mathbf{map_1}\ \ f\ \widetilde{o}) \end{array}$$

Proof. Let S be the following relation:

$$\left\{ \begin{array}{ll} x, \mathbf{map_1}\ \ p\ (& \text{system}\ \widetilde{y} \\ & \text{where}\ \ \widetilde{y} = x : \mathbf{map_1}\ \ f\ \widetilde{y}) \end{array} \right\}$$

We can prove, using preceding laws, that:

$$\mathbf{stl} \left(\begin{array}{ll} \mathbf{map_1}\ \ p\ (& \text{system}\ \widetilde{y} \\ & \text{where}\ \ \widetilde{y} = x : \mathbf{map_1}\ \ f\ \widetilde{y}) \end{array} \right) = \begin{array}{ll} \mathbf{map_1}\ \ p\ (& \text{system}\ \widetilde{y} \\ & \text{where}\ \ \widetilde{y} = fx : \mathbf{map_1}\ \ f\ \widetilde{y}) \end{array}$$

and, that:

$$\textbf{shd}\left(\textbf{map}_\mathbf{1}\ \ p\ (\begin{array}{ll}\text{system } \widetilde{y} & \\ \text{where} & \widetilde{y} = x : \textbf{map}_\mathbf{1}\ \ f\ \widetilde{y})\end{array}\right) = fx$$

Thus, by Law 1, S is a *SER* relation and by Theorem 1, both streams are equal. □