Modular Causality in a Synchronous Stream Language

Pascal Cuoq, Marc Pouzet

INRIA, LIP6
This work has been partially supported by CNET-France Telecom

Abstract

This article presents a causality analysis for a synchronous stream language with higher-order functions. This analysis takes the shape of a type system with rows. Rows were originally designed to add extensible records to the ML type system (Didier Rémy, Mitchell Wand). We also restate briefly the coiterative semantics for synchronous streams (Paul Caspi, Marc Pouzet), and prove the correctness of our analysis with respect to this semantics.

1 Introduction

1.1 History

This part gives a quick overview, in chronological order, of existing languages allowing the manipulation of infinite streams, and the position of Lucid Synchrone[2] amongst them.

Lucid was the first language to propose the paradigm of streams as first-class citizens. Its purpose was to bring programming languages closer to logic languages in which properties can be expressed and proved. In Lucid, recursive equations on streams replaced imperative iteration.

Then appeared synchronous data-flow languages such as Lustre[8], Signal[1], on the one hand, and lazy functional languages (*e.g.* Haskell[9]) on the other hand. Lustre and Signal are used in the fields of signal processing and automatic control. Lazy functional languages can handle infinite data structures in general, and streams in particular.

These two families evolved quite independently. Lustre and Signal did not have higher-order functions, and the canonical representation of streams in a lazy functional language such as Haskell did not have the reactivity nor the efficiency required by the kind of applications in which the former two are used. Wadler proposed a set of techniques for functional languages, "listlessness"[19], to avoid building intermediate lists which are de-constructed immediately. The resulting optimized programs are in some cases very similar to the ones that are generated by Lustre or Signal.

Lucid Synchrone is an attempt at filling the gap between these two families of languages, by proposing as many ML-style features as possible (and in particular modular compilation) while retaining an efficient representation of streams [3].

1.2 Plan

In section 2, we describe informally the core language which will be the subject of this presentation. Section 3 describes the causality analysis which is the novel part of this article. Section 4 gives a formal semantics to our core language, by translating streams to their co-iterative representation. In section 5, we prove that the analysis described in section 3 is correct with respect to the semantics described in 4. Finally, our analysis is compared to other analyses in 6.

2 The Lucid Synchrone core

2.1 Language Description

Here is the description of the subset of Lucid Synchrone which will be the base of this presentation:

programs:

$t ::=$	(t_1, t_2)	pair
	$\mid$ `fst` t $\mid$ `snd` t	projections
	$\mid x$	variable
	$\mid \lambda x.t$	λ-abstraction
	$\mid t_1\ t_2$	application
	$\mid$ `let` $x = t_1$ `in` t_2	non-recursive let
	$\mid$ `rec` $x_1 = t_1$ `and` $x_2 = t_2$	recursive definition
	$\mid c$	constants
	$\mid p$	primitives
	$\mid$ `pre` $c\ t$	a delay, initialized with the constant c

The constants c are imported from an external scalar language, and lifted into the corresponding constant stream. Likewise, primitive functions imported from the scalar world are extended into pointwise functions on streams.

The abstraction, application, pair constructor, projections, and recursion combinator behave exactly as in the λ-calculus.

Finally, `pre` $c\ u$ catenates the scalar value c in front of the stream u. In the usual interpretation of streams as successive values of a node, `pre` introduces a delay of one instant.

u	$u_1\ u_2\ u_3\ u_4\ \ldots$
`pre` $c\ u$	$c\ \ u_1\ u_2\ u_3\ \ldots$

Synchronous stream languages (Lustre, Signal, and Lucid Synchrone) have in common another feature, which is not present here: a *clock* is associated with each expression. A clock system allows to extract a sub-stream from a stream by *sampling*. This feature is mostly orthogonal to the causality analysis, and this is why we do not put it in our core language.

2.2 A tutorial example

The domain of application of synchronous languages is the design of reactive systems. The mathematical concept corresponding to a reactive system is a Mealy automaton, that is, a finite state automaton that, given one set of input values, compute a set of output values that are determined by the inputs and the state in which the previous inputs had left the system. For instance, a coffee machine will, given the input "Button pressed", react by beeping or delivering some coffee depending on whether the input "Coin inserted" has been seen recently.

Since the current outputs depend not only on the current inputs but on the state of the system, which in turn depends on the previous inputs, it is convenient to see a reactive program as a computation transforming the stream of inputs into a stream of outputs, being understood that the nth value of the output stream is allowed to depend only on the n first values of the input stream.

Not only does this point of view allow to program very straightforwardly a lot of common behaviors, but since the notion of state disappears completely from the programs, it also makes it easier to prove properties about them.

Here is how the mechanism of the simplistic coffee machine could be expressed in Lucid Synchrone:

```
let rec number_coins = prev_number_coins + inserted_coins -
                         (if deliver_coffee then 1 else 0)
and prev_number_coins = pre 0 number_coins
and deliver_coffee = button_pressed and (prev_number_coins >= 1)
```

`button_pressed` and `inserted_coins` are the input streams. `button_pressed` is a boolean stream, and `inserted_coins` is a stream of integers that indicates the number of coins inserted during instant t.

The output stream is `deliver_coffee`; the other streams are only defined as computation intermediaries. Output streams can be be used in the computation of other streams, as is the case here for `number_coins`.

`(if deliver_coffee then 1 else 0)` is an integer stream whose value is `1` at instant t if coffee is delivered at instant t, and `0` otherwise (we suppose that delivering one beverage is instantaneous).

Another aspect of reactive systems on which we have not insisted enough yet is that they usually must be able to propose an output in a bounded time after they have been provided with an input. This aspect is not emphasized much in synchronous languages, in which it is assumed (and can be checked *a posteriori* if necessary) that computations are always fast enough. Still, an important consequence is that instantaneous recursion is absent from the language; the provided recursion operator is only intended to define streams recursively, using the value from instant t-1 to compute the value at instant t (`number_coins` is a typical example). In the kind of application synchronous languages are used for, this apparent lack of expressiveness is rarely a problem, whereas bounded time computations are a prime necessity.

3 Causality analysis of Lucid Synchrone programs

In real life, causality forbids that information from the future be used in the present. Such interferences cannot happen in the context of a clock-less synchronous stream language, because there are no operators to express the future. However, since computations are supposed to be instantaneous, causality can still be broken by using the current value of a variable within its definition.

Recursive definitions such as `rec x=x` (which is valid as an equation, but does not define a stream) and `rec x=x+1` (which has no solution) must thus be detected and rejected statically. This is the purpose of the causality analysis.

On the other hand, thanks to the delay introduced by `pre`, the expression `1 + pre 0 x` no longer depends instantaneously on `x`; the recursive definition `rec x = 1 + pre 0 x` that denotes the stream of strictly positive integers, is valid, and must be accepted.

In the case of mutually recursive definitions, it would be too restrictive to require each right-hand-side expression to feature a `pre`. For instance, `rec` $x = y + 1$ `and` $y =$ `pre` $0\ x$ is a valid definition of two streams x and y.

Finally, the function `rewind` defined by `let rewind =` λ`f . (rec x = f x)` cannot be rejected *a priori*, since the recursion will be causal if it is applied to a function that contains a `pre`. On the other hand if it is applied to a strict function such as the identity, the recursion in the body of `rewind` is not causal.

The causality analysis about to be presented was designed to accept this kind of programs, and to be compatible with higher-order functions.

This analysis is in fact a type system. It features let-polymorphism similar to the type systems of ML and Haskell. The main change is that there is only one base type $\{\varphi\}$, and that it carries a dependency annotation φ, which is an association list mapping variables to their dependence information. A recursion variable x can be marked as "present" (notation $x\!:\!p$), meaning that the expression e having type $\{x : p, \varphi\}$ might depend instantaneously on x, or "absent" ($x\!:\!a$), meaning that e does not depend instantaneously on x.

The association lists between variables and dependence information are represented as rows ([20],[14]). Rows were originally designed to introduce extensible records in ML. Generally speaking, rows allow in some cases (and in ours in particular) to use polymorphism to replace subtyping.

The grammar of types is as follows:
types:

$\tau ::=$	α	type variable
	$\mid \{\varphi\}$	dependency information (rows)
	$\mid \tau_1 * \tau_2$	pair
	$\mid \tau_1 \rightarrow \tau_2$	arrow type

rows:

$\varphi ::=$	ρ	row-variable
	$\mid x\!:\!\pi, \varphi$	one "field" concerning variable x

presence information:
π ::= a absent
| p present
| δ presence variable

type schemes:
σ ::= $\tau \mid \forall\alpha,\sigma \mid \forall\rho,\sigma \mid \forall\delta,\sigma$

Our equational theory on rows is the same as the one used in the context of extensible records:

$$x\!:\!\pi, y\!:\!\pi', \varphi = y\!:\!\pi', x\!:\!\pi, \varphi$$

Likewise, our notion of sorts is the same as the one for extensible records: the sort of a row φ is a set of labels (in our case, recursion variables) which intuitively are the fields which can not appear in φ, because they are already present on the left-hand side of a larger row of which φ is the tail. One can check easily that all the rules listed in this section are well-sorted. For all these reasons general theorems about rows([13, 14]) apply and our system has the principal typing property.

The binder $\forall$ plays its usual role in type schemes. There is also the classic relation "to be an instance of" between types and types schemes, noted $\tau < \sigma$, and inductively defined by:
$\tau < \tau$
If $\tau < \sigma$, then for any τ' and α, $\tau[{}^{\tau'}/_{\alpha}] < \forall\alpha,\sigma$
If $\tau < \sigma$, then for any row φ and row-variable ρ of the same sort, $\tau[{}^{\varphi}/_{\rho}] < \forall\rho,\sigma$
If $\tau < \sigma$, then for any π and δ, $\tau[{}^{\pi}/_{\delta}] < \forall\delta,\sigma$

We will assume that all the bound variables of the program being analyzed are distinct. If we had used a notion of reduction to define the semantics of streams programs, this assumption would be a cause of technical complications since it would not remain true after a reduction; but since the co-iterative semantics of stream programs is given by translation to a functional language instead it is a reasonable assumption to make.

Typing judgments are of the form $\Gamma \vdash t : \tau$.
Γ is a typing environment, associating type schemes to program variables.

The originality of this system resides in the types given to the operators `rec` and `pre`. This should not be surprising since `rec` is the construction that makes this analysis necessary, and `pre`, on the other hand, is the one that allows recursive definitions to be well-founded.

`pre` c is given the type scheme $\forall\rho,\rho',\{\rho\} \to \{\rho'\}$:

$$\frac{\Gamma \vdash t : \{\varphi\}}{\Gamma \vdash \texttt{pre}\ c\ t : \{\varphi'\}}$$

In this rule, whatever variables t has been found to depend on (the variables marked p in φ), the row $\{\varphi'\}$ of $\texttt{pre}\ c\ t$ is unconstrained. This may seem counter-intuitive: one would expect this rule to say in essence "$\texttt{pre}\ c\ t$ does not depend on anything", whereas its actual meaning is more like "The dependencies of $\texttt{pre}\ c\ t$ can be anything you like". But note that if you choose $\{\varphi'\}$ to be for instance $\{x : a, y : p, \rho\}$, you have only (correctly) lost the information that $\texttt{pre}\ c\ t$ does not depend on y.

This allows us to accept a term such as $\texttt{plus}\ x\ (\texttt{pre}\ 0\ x)$ where the pointwise addition on streams $\texttt{plus}$ has the type $\forall\rho, \{\rho\} \to \{\rho\} \to \{\rho\}$. In this case the rule for $\texttt{pre}$ is instantiated with identical rows for $\{\varphi\}$ and $\{\varphi'\}$. The result is found to have the same dependencies as x.

The rule for unary recursion is as follows:

$$\frac{\Gamma, x : \{x : p, \varphi\} \vdash t : \{x : a, \varphi\}}{\Gamma \vdash \texttt{rec}\ \ x = t : \{x : \pi, \varphi\}}$$

The recursion variable x is put in the environment, associated with the row $\{x : p, \varphi\}$. This way, any expression t depending on x (such as x itself, or $\texttt{plus}\ x\ (\texttt{pre}\ 0\ x)$) will have in its row a field x associated to p. This row won't be unifiable with the expected one of the form $\{x : a, \varphi\}$, and $\texttt{rec}\ \ x = t$ will be rejected.

Another, perhaps more intuitive conclusion for this rule would be $\Gamma \vdash \texttt{rec}\ \ x = t : \{\varphi\}$. Unfortunately, this would break the sorting rules on rows: φ should only be used with some $x : \pi$ on its left-hand side.

Leaving the instance of π open can help to accept contrived examples with nested recursions, such as $\texttt{rec}\ \ z = (\texttt{rec}\ \ x = \texttt{pre}\ 0\ (x + z))$.

Mutual recursion involves one row for each of the recursion variables:

$$\frac{\begin{array}{c}\Gamma, x : \{x : p, y : \pi_1, \varphi_1\}, y : \{x : \pi_2, y : p, \varphi_2\} \vdash t_1 : \{x : a, y : \pi_1, \varphi_1\} \\ \Gamma, x : \{x : p, y : \pi_1, \varphi_1\}, y : \{x : \pi_2, y : p, \varphi_2\} \vdash t_2 : \{x : \pi_2, y : a, \varphi_2\}\end{array}}{\Gamma \vdash \texttt{rec}\ \ x = t_1\ \ \texttt{and}\ \ y = t_2 : \{x : \pi_3, y : \pi_4, \varphi_1\} * \{x : \pi_5, y : \pi_6, \varphi_2\}}$$

This rule is a generalization of the one for unary recursion. x (resp. y) is associated in the environment to a row which carries the information "depends on x(resp. y)".

Of course there is nothing wrong with t_1 depending on y, as in the example $\texttt{rec}\ \ x = y\ \ \texttt{and}\ \ y = \texttt{pre}\ 0\ y$. Just instantiate π_1 with p.

We've seen that, if t_1 depends on y, we want to prevent t_2 from depending on x. Here is how this is achieved: if t_1 depends on y, t_2 is analyzed in an environment in which x is associated to $\{x : p, y : p, \rho\}$. In these circumstances, if t_2 depends on x, it too has $y : p$ in its dependencies, and the second premise can not be satisfied. Indeed, one can check that there is no derivation for the term $\texttt{rec}\ \ x = y\ \ \texttt{and}\ \ y = x$.

The key is that π_1 and π_2 are shared between both premises.

Constants are given the type scheme $\forall \rho, \{\rho\}$:

$$\overline{\Gamma \vdash c : \{\varphi\}}$$

All the remaining rules are identical to those of the Hindley-Milner type system:

$$\frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \texttt{fst}\ t : \tau_1} \qquad \frac{\Gamma \vdash t : \tau_1 * \tau_2}{\Gamma \vdash \texttt{snd}\ t : \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 * \tau_2}$$

$$\frac{\tau < \sigma}{\Gamma, x : \sigma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x.t : \tau \to \tau'} \qquad \frac{\Gamma \vdash t_1 : \tau \to \tau' \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ t_2 : \tau'}$$

$$\frac{\Gamma \vdash t : \tau \quad \Gamma, x : \forall(\mathcal{FV}(\tau) - \mathcal{FV}(\Gamma)), \tau \vdash t' : \tau'}{\Gamma \vdash \texttt{let}\ x = t\ \texttt{in}\ t' : \tau'}$$

The typing rules for `pre` and `rec` use row types instead of general types: the second argument of the `pre` operator can therefore only be a stream, and the only possible recursive definitions are definitions of streams. Being unable to recursively define a pair, we need the `rec` operator to explicitly handle mutual recursion, instead of encoding it as a recursion on a pair, as it is done in a traditional functional system.

These restrictions are not as arbitrary as it may at first seem. Firstly, we do not know how to give a semantics to (`pre` $c\ e$) where e is a function. Secondly, assuming for the sake of the argument that we knew how to define such a semantics, and assuming that (`pre` c) had the type $\forall \alpha \forall \alpha', \alpha \to \alpha'$ instead of $\forall \rho \forall \rho', \{\rho\} \to \{\rho'\}$, the type system would allow to infer that (`pre` $(\lambda x.$`pre` $0\ x)\ (\lambda x.x))\ y$ does not depend on y, whereas it would (again, if this program meant anything at all) after the first instant.

Recursively defined functions are a different problem. For higher-order recursion, the well-foundedness of the recursion depends on semantic conditions and can not be checked by this causality analysis. Although recursive functions are not allowed by this system, it is still possible, in an actual implementation, to have a different operator for functional recursion, and a compiler flag for either authorize the programmer to use it or to forbid its use (and guarantee reactiveness).

Examples:

– Constant streams
  Constant streams didn't really need to be integrated in the language, as the constant stream c can be defined as `rec z = pre c z`. The type scheme given to this expression, $\forall \rho, \delta,\ \{z\!:\!\delta, \rho\}$, is equivalent to the type scheme of constant streams $\forall \rho, \{\rho\}$ under the hypothesis we made that all the variables bound in a program are distinct.
– The "rewind" combinator λ`f.rec x = f(x)` is causal, of type scheme $\forall \rho, \delta,\ (\{x\!:\!p, \rho\} \to \{x\!:\!a, \rho\}) \to \{x\!:\!\delta, \rho\}$.

- Mutual recursion: the example `rec x = y and y = pre 0 x` is accepted. In the rule for the mutual recursion, π_1 (presence information of y in the row associated to x in the environment) is instantiated with p, and π_2, presence information of x in the row of y, is instantiated with a.
- Higher-order mutual recursion: the function
  $F \equiv \lambda$`f.`λ`g.rec x = g(y) and y = f(x)`
  is well-typed, and has the type scheme:
  $\forall \rho_1, \rho_2, \delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6, (\{x : \delta_2, y : p, \rho_2\} \to \{x : a, y : \delta_1, \rho_1\}) \to$
  $(\{x : p, y : \delta_1, \rho_1\} \to \{x : \delta_2, y : p, \rho_2\}) \to (\{x : \delta_3, y : \delta_4, \rho_1\} * \{x : \delta_5, y : \delta_6, \rho_2\})$
  Moreover, the partial application of F to a strict function such as the identity gives a function which can be applied to $\lambda x.$`pre` $c\ x$ but not to a strict function, whereas the partial application of F to $\lambda x.$`pre` $c\ x$ gives a function which can be applied to a strict function.
- The use of unification instead of polymorphism makes the computation of dependencies "bidirectional": for instance the example `rec x = pre 0 (x+y) and y = x` is rejected. What happens is that the rows for `x` and `y` become unified when `x` and `y` are added together, and these variables become interdependent. This is probably the biggest drawback of the system; Fortunately the types of variables and functions bound by a `let` construct can usually be generalized, which breaks this bidirectionality.

4 Co-iteration

In [3], synchronous streams are given a concrete representation made of an initial state and a transition function returning a value and a new state when applied to a state.

In this section, we will re-state the co-iterative representation of streams. The target language is a functional language, and it is best to think of it as lazily evaluated for now.

We choose to represent streams in the target language by the triple $([\![t]\!]_i, [\![t]\!]_v, [\![t]\!]_s)$, where:

- $[\![t]\!]_i$ is the initial state.
- $[\![t]\!]_v$ is a function returning a value when applied to the current state.
- $[\![t]\!]_s$ is a function returning the new state when applied to the current state.

Comparing this representation to the one of [3], we split the transition function into two separate ones, one returning the value and the other returning the new state. Of course, the two representations are equivalent, but this one allows for a simpler expression of the causality of programs.

The translations of constant streams, `pre`, pairs and projections are relatively straightforward. More details are available in the extended version of this article [5] and in [3].

A constant stream needs no memory, so its state is Nil.

$$\begin{aligned}
[\![c]\!]_i &= \texttt{Nil}\\
[\![c]\!]_v &= \lambda\texttt{Nil}.c\\
[\![c]\!]_s &= \lambda\texttt{Nil}.\texttt{Nil}
\end{aligned}$$

– **pre** c t:

$$\begin{aligned}
[\![\texttt{pre}\ c\ t]\!]_i &= (c, [\![t]\!]_i)\\
[\![\texttt{pre}\ c\ t]\!]_v &= \lambda(v,s).v\\
[\![\texttt{pre}\ c\ t]\!]_s &= \lambda(v,s).([\![t]\!]_v\ s, [\![t]\!]_s\ s)
\end{aligned}$$

– (t_1, t_2):

$$\begin{aligned}
[\![(t_1,t_2)]\!]_i &= ([\![t_1]\!]_i, [\![t_2]\!]_i)\\
[\![(t_1,t_2)]\!]_v &= \lambda(s_1,s_2).([\![t_1]\!]_v\ s_1, [\![t_2]\!]_v\ s_2)\\
[\![(t_1,t_2)]\!]_s &= \lambda(s_1,s_2).([\![t_1]\!]_s\ s_1, [\![t_2]\!]_s\ s_2)
\end{aligned}$$

– $\texttt{fst}\ t$:

$$\big([\![\texttt{fst}\ t]\!]_i,\ [\![\texttt{fst}\ t]\!]_v,\ [\![\texttt{fst}\ t]\!]_s\big) = \big([\![t]\!]_i,\ \lambda s.\texttt{fst}\ ([\![t]\!]_v\ s),\ [\![t]\!]_s\big)$$

– Recursion:
The definitions of $[\![t]\!]_i$, $[\![t]\!]_v$ and $[\![t]\!]_s$ use inductively the translations of the sub-terms of t. In the case of the translation of $\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2$, x and y are free variables in t_1 and t_2, and we need something to substitute $[\![x]\!]_i$, $[\![x]\!]_v$,... with in $[\![t_1]\!]$ and $[\![t_2]\!]$. In particular, in $[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_v$ we replace $[\![x]\!]_v$ by the function returning the value being recursively computed, and in $[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_s$ we replace it by the function returning the value computed by $[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_v$.

$$\begin{aligned}
[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_i &= ([\![t_1]\!]_i, [\![t_2]\!]_i)[^{\texttt{Nil}}/_{[\![x]\!]_i}][^{\texttt{Nil}}/_{[\![y]\!]_i}]\\
[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_v &= \lambda(s_1,s_2).\\
&\qquad \texttt{let rec}\ (x_v, y_v) =\\
&\qquad\qquad \big(([\![t_1]\!]_v\ s_1, [\![t_2]\!]_v\ s_2)[^{\lambda\texttt{Nil}.x_v,\lambda\texttt{Nil}.y_v}/_{[\![x]\!]_v,[\![y]\!]_v}]\big)\\
&\qquad \texttt{in}\ (x_v, y_v)\\
[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_s &= \lambda(s_1,s_2).\\
&\texttt{let rec}\ (x_v, y_v) =\\
&\qquad \big(([\![t_1]\!]_v\ s_1, [\![t_2]\!]_v\ s_2)[^{\lambda\texttt{Nil}.x_v,\lambda\texttt{Nil}.y_v}/_{[\![x]\!]_v,[\![y]\!]_v}]\big)\ \texttt{in}\\
&([\![t_1]\!]_s\ s_1, [\![t_2]\!]_s\ s_2)[^{\lambda\texttt{Nil}.x_v,\lambda\texttt{Nil}.y_v,\lambda\texttt{Nil}.\texttt{Nil},\lambda\texttt{Nil}.\texttt{Nil}}/_{[\![x]\!]_v,[\![y]\!]_v,[\![x]\!]_s,[\![y]\!]_s}]
\end{aligned}$$

5 Correctness proof

In this section, we give the correctness proof for the system presented in section 3.

Here, we consider the target language as a λ-calculus equipped with strong β-reduction. We will in particular make heavy use of the Church-Rosser property. The recursion construct is just syntactic sugar above the Y fixpoint combinator.

We will also use the property that terms of the target language that do not use the recursion operator have a normal form for strong β-reduction. This is due to the fact that all the terms we are dealing with can be typed.

The property that we intend to prove is essentially that if the term t has type $\{x:a,\ldots\}$, then t "does not depend" on x.

For a stream t, "not to depend" on x means that the instantaneous value $[\![t]\!]_v\ s$ of t can be computed without requiring the instantaneous value of x. Or, still more concretely, x does not occur in the normal form of $[\![t]\!]_v\ s$ [1].

Still informally, we will define the set of terms t that do not depend on x as the *interpretation* of the type $\{x:a,\ldots\}$. Obviously, with our definition of "not depending on", the membership of a term t to this interpretation only depends on the normal form of $[\![t]\!]_v\ s$.

The presence of polymorphism in the system requires us to define a notion of "candidate", as in [7]. A candidate is a set of terms that has the correct shape to be the interpretation of a type. Quite naturally, our notion of candidate will be "any set to which the membership of a term t only depends on a property of the normal form of $[\![t]\!]_v\ s$".

Definitions:

- A candidate C is a set of terms such that the property "$t \in C$" can be expressed under the form "$[\![t]\!]_v$ and $[\![t]\!]_s$ do not use recursion and, for all states s, t_1 being the normal form of $[\![t]\!]_v\ s$, $\mathcal{R}(t_1)$", for some arbitrary predicate $\mathcal{R}$.
- A row-candidate Q is a set of terms such that the property "$t \in Q$" can be expressed under the form:

  "$[\![t]\!]_v$ and $[\![t]\!]_s$ do not use recursion and
  For all states s, if t_1 is the normal form of $[\![t]\!]_v\ s$, then $\mathcal{R}(t_1)$"

  where $\mathcal{R}$ is a predicate that can be expressed under the form:

$$\mathcal{R}(t) \equiv \bigwedge_{x \in X} \big(x \text{ does not occur in } t\big)$$

These preliminary definitions deserve a couple of remarks:

- This definition is simpler that the one of a candidate in, for instance, the strong normalisation proof of the System F[7]. This is due to the fact that the property we want to prove is itself simpler.
- On the other hand, our system features rows, which need their own notion of candidate. This was to be expected, in the light of the fact that, as we pointed out previously, rules for `pre` and `rec` can only be instantiated with rows, and would become incorrect if instantiated with arbitrary types.

[1] In this informal explanation, we refer to "the" normal form of $[\![t]\!]_v\ s$ (for strong β-reduction). In fact, the existence of the normal form will be proved at the same time as the correctness of the dependency analysis. This will be done by proving that causal recursion on streams can be translated without recursion in the target language.

- Note that the definition of the interpretation on rows is compatible with the equational theory. That is, whichever particular syntactic representation of a given row yields the same interpretation.
- A row-candidate is a candidate. In fact, the last sentence is the image in the world of interpretations of the sentence "a row is a type".

We will now consider mappings from type variables to candidates, and row-variables and presence variables to row-candidates. Such a mapping is called a valuation and will be denoted V. Given a valuation V, we write $V \sqcup \alpha \mapsto C$ the valuation that maps α to C, and any other variable β to $V(\beta)$.

When referring to a valuation V, we will take for implicit the fact that the sets of terms associated to type variables (resp. row-variables and presence variables) are candidates (resp. row-candidates).

Definition of the interpretation $\mathcal{P}_V$ of a type (relative to a valuation V):

- $t \in \mathcal{P}_V(\{x:a,\varphi\})$ if $[\![t]\!]_v$ and $[\![t]\!]_s$ do not use recursion, and the normal form of its application to a state[2] does not contain x, and if $t \in \mathcal{P}_V(\{\varphi\})$.
- $\mathcal{P}_V(\{x:p,\varphi\}) = \mathcal{P}_V(\{\varphi\})$.
- $t \in \mathcal{P}_V(\{x:\delta,\varphi\})$ if $t \in V(\delta)$ and $t \in \mathcal{P}_V(\{\varphi\})$
- $\mathcal{P}_V(\rho) = V(\rho)$
- $\mathcal{P}_V(\tau_1 * \tau_2)$ is defined as the set of terms t such that `fst` $t \in \mathcal{P}_V(\tau_1)$, and `snd` $t \in \mathcal{P}_V(\tau_2)$
- $\mathcal{P}_V(\tau_1 \to \tau_2)$: the set of terms t such that $[\![t]\!]_v$ and $[\![t]\!]_s$ do not use recursion, and such that for all $t_1 \in \mathcal{P}_V(\tau_1)$, $(t\ t_1) \in \mathcal{P}_V(\tau_2)$
- $\mathcal{P}_V(\alpha) = V(\alpha)$

We need to generalize our interpretation to non-empty environments. Classically, this is done by requiring the term to satisfy the interpretation property under all possible substitutions of terms (satisfying the interpretation of the type they are assigned in the environment) to free variables.

More formally, we first extend the interpretation to type schemes:
$t \in \mathcal{P}_V(\forall\alpha,\sigma)$ if for all candidate C, $t \in \mathcal{P}_{V\sqcup\alpha\mapsto C}(\sigma)$.
$t \in \mathcal{P}_V(\forall\rho,\sigma)$ if for all row-candidate Q, $t \in \mathcal{P}_{V\sqcup\rho\mapsto Q}(\sigma)$.
$t \in \mathcal{P}_V(\forall\delta,\sigma)$ if for all row-candidate Q, $t \in \mathcal{P}_{V\sqcup\delta\mapsto Q}(\sigma)$.

Let us now list a few properties relative to interpretations and candidates:

Property 1. For any valuation V, $\mathcal{P}_V(\{\varphi\})$ is a row-candidate.

Proof: by induction on φ.

Property 2. For any valuation V, $\mathcal{P}_V(\tau)$ is a candidate.

[2] All successive states are normalizable because $[\![t]\!]_s$ does not use recursion. For this reason and because $[\![t]\!]_v$ does not use recursion "the" normal form of the application of $[\![t]\!]_v$ to a state exists

Proof: by induction on τ.

As expected, the interpretations of type schemes are included in the interpretations of their instances. This can be stated as:

Property 3. For all valuations V, if $t \in \mathcal{P}_V(\sigma)$ then for all $\tau < \sigma$, $t \in \mathcal{P}_V(\tau)$.

The proof is by induction on σ.

Theorem 1. *If $\Gamma \vdash t : \tau$, then for all V, for any substitution S adapted to Γ (under V)* [3], $S(t) \in \mathcal{P}_V(\tau)$.

The bulk of the proof is an induction on the inference tree of the causality analysis. We'll only present some interesting cases.

1. The case of `pre` is an interesting one, even though it is simple: $\mathcal{P}_V(\{\varphi'\})$ is a row-candidate. Since $[\![\texttt{pre}\ c\ t]\!]_v = \lambda(v,s).v$, obviously `pre` $c\ t$ is in this set.
2. The rule for `rec`:
   Take some fixed V and S.
   The interpretation $\mathcal{P}_V(\{y : \pi_1, \varphi_1\})$ is a row-candidate. Therefore, there exists X a set of variables such that t belongs to $\mathcal{P}_V(\{y : \pi_1, \varphi_1\})$ if and only if t does not depend on any variable of X.
   Let us distinguish two cases:
   (a) $y \in X$
      In this case, the induction hypothesis for the first premise yields that neither x not y occur in the normal form of $[\![(S \sqcup x \mapsto x \sqcup y \mapsto y)(t_1)]\!]_v\ s$ (i.e. $[\![S(t_1)]\!]_v\ s$).
      That means that the code for the "value" transition function $[\![\texttt{rec}\ x = t_1\ \texttt{and}\ y = t_2]\!]_v$ could simply have been implemented as [4]:

$$\begin{array}{l}\lambda(s_1, s_2).\\ \quad \texttt{let}\ x_v = [\![t_1]\!]_v\ s_1\ \texttt{in}\\ \quad \texttt{let}\ y_v = [\![t_2]\!]_v\ s_2\ \texttt{in}\\ \quad (x_v, y_v)\end{array}$$

   (b) $y \notin X$
      Claim: For all s, x does not appear in the normal form of $[\![t_2]\!]_v\ s$.
      Let us suppose it was otherwise, and build a contradiction: The idea is to substitute x with a term that depends on y (for instance, y itself) since it satisfies the interpretation of $\{x : p, \varphi_1\}$.

[3] That is, for any substitution S associating to each variable x of Γ a term t such that $t \in \mathcal{P}_V(\Gamma(x))$

[4] In fact, we are using this notation to lead the reader to believe that there is no recursive definition of values any longer, and that a strict functional language would be enough to implement this recursion. We are cheating: we only proved that x and y did not occur *in the normal form* of $[\![t_1]\!]_v\ s_1$, not that they didn't occur in $[\![t_1]\!]_v\ s_1$. Thanks for bearing our bad faith.

Applying the induction hypothesis for the second premise with $(S \sqcup x \mapsto y \sqcup y \mapsto y)(t_2)$ shows that if x appeared in $[\![t_2]\!]_v\ s$, then t_2 could not have the type $\{y : a, \varphi_2\}$.
Therefore, the "value" transition function for `rec` $x = t_1$ `and` $y = t_2$ is equivalent to:

$$
\begin{array}{l}
\lambda(s_1, s_2). \\
\ \ \texttt{let}\ y_v = [\![t_2]\!]_v\ s_2\ \texttt{in} \\
\ \ \texttt{let}\ x_v = [\![t_1]\!]_v\ s_1\ \texttt{in} \\
\ \ (x_v, y_v)
\end{array}
$$

In both cases, the LUCID SYNCHRONE `rec` could be represented without recursion in the target language, and the alternative representation clearly satisfies the induction invariant.
However, note that when, for instance, producing the code for the function $\lambda f.\lambda g.$`rec` $x = f(y)$ `and` $y = g(x)$, it is not yet possible to know which case will apply. Since the type scheme is polymorphic and the produced code is not, both versions of the code can be required, if this function is later applied several times.

6 Related work

There are quite a few systems whose purpose is to verify the productivity of recursive definitions. The closest to the one presented in this article is described in [18]: it is a causality analysis for an extension of Signal with higher-order functions. To the best of our understanding, although the rules for `pre` in both systems are very similar, the rule for recursion in [18] seems to be the classical one (if supposing $x : \tau$ one can prove $e : \tau, \ldots$), which would imply that non-causal programs have to be ruled out by side-conditions external to the system. In this sense, the novelty in the system presented here would be that these conditions are expressed inside the type system. Benefits of the absence of external side-conditions are that it allows us to have principal types, and that types, and typing errors, are easier to print.

Explicit handling of dependency graphs (as is done in LUSTRE, or in [15]) is more expressive than our system, but it isn't modular. Some information is lost when graphs are encoded in types (even with the expressivity granted by rows) but each piece of program can be rejected, or accepted and given a type, independently of the context in which it is used.

Most of other causality analyses are designed for languages in which streams can be de-constructed. This of course complicates the problem: the dependency on x in `pre 0 x` (or `Cons 0 x`) is only temporarily hidden, it can reappear if the stream is de-constructed.

In consequence, it is necessary to count how many `Cons` have been produced in each expression. This in turn makes subtyping almost unavoidable, if one wants to accept programs such as `x + (Cons 0 x)` (+ being the pointwise operator on streams). An instance of such a system is [6].

The system proposed by J.Hughes, L.Pareto and A.Sabry in [10] is much more expressive, but also much more complex. Their system makes use of subtyping. It allows to express linear relations between stream sizes, thus verifying at the same time causality and what would correspond to well-clockedness in a synchronous language.

In the case of *hardware* design, this analysis can be considered too restrictive. Circuits with cycles can be causal, as shown in [11]. The Esterel compiler has a causality analysis[16] aimed at accepting such cyclic programs. Other languages for hardware design, this time using Haskell streams, are LAVA[17] and HAWK[4]). It should be noted that their use of streams conforms to the synchronous hypothesis, and that, on the other hand, compilation to hardware has a different set of requirements than compilation to software. Compilation to hardware is usually not modular so it is less annoying if an analysis isn't either.

Analogous analyses are used in general programming languages to obtain information about the strictness of functions, or the information flow of programs. In general, in these analyses there is no special treatment of recursion, whereas in our case the recursion construct is the end, and expressing the relationships between functions arguments and results are only a mean to this end.

7 Conclusion and future work

We presented a causality analysis for a language of synchronous streams. Good properties such as principal typing and ease of implementation are inherited from the use of rows instead of subtyping. Although the correctness proof is specific to Lucid Synchrone's co-iterative semantics, the analysis itself should be generalizable to mutually recursive definitions where there is no operation (such as `tail`) to transform a guarded expression into a non-guarded one.

A fine-grained causality analysis imposes a different approach to code generation than is usually done in synchronous stream languages. We moreover expect that results from the causality analysis can be used for type-directed optimization of this code.

Acknowledgements

We are grateful to Michel Mauny and Didier Rémy for their help with early versions of this article. François Pessaux had used a type system with rows to statically detect uncaught exceptions[12], and his explanations were very beneficial. We would also like to thank the referees for their constructive suggestions.

References

1. A. Benveniste, P. LeGuernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
2. P. Caspi and M. Pouzet. The Lucid Synchrone distribution. `http://www-spi.lip6.fr/~pouzet/lucid-synchrone/`.

3. P. Caspi and M. Pouzet. A co-iterative characterisation of synchronous stream functions. Technical Report 07, VERIMAG, October 1997. Workshop on Coalgebraic Methods in Computer Science (CMCS'98), Electronic Lecture Notes in Theoretical Computer Science, Portugal, Lisbon (28-29 March 1998).
4. B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *1998 Workshop on Formal Techniques for Hardware*, 1998.
5. P. Cuoq and M. Pouzet. Modular causality in a stream language. `http://pauillac.inria.fr/~cuoq/streams/pl.dvi`.
6. E. Giménez. Structural recursive definitions in type theory. In *ICALP'98*, LNCS series no. 1443, July 1998.
7. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
8. 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.
9. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1990.
10. J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *ACM Principles of Programming Languages*, St Petersburg, Florida, January 1996.
11. S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer Aided Design*, 13(7):950–956, July 1994.
12. F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Trans. Prog. Lang. Syst.*, 22(2):340–377, 2000.
13. D. Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatique, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992. `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`.
14. D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz`.
15. J. Saraiva, D. Swiersrta, M. Kuiper, and M. Pennings. Strictification of lazy functions. Technical report, 1996. `http://www.cs.ruu.nl/docs/research/publication/TechList2.html`.
16. T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. International Design and Testing Conference*, Paris, 1996.
17. S. Singh and M. Sheeran. Designing FPGA circuits in Lava. Technical report. `http://www.dcs.gla.ac.uk/~satnam/lava/lava_intro.ps`.
18. J.-P. Talpin and D. Nowak. A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems. *Conference on Foundations of Software Technology and Theoretical Computer Science*, 1998.
19. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science, (Special issue of selected papers from 2'nd European Symposium on Programming)*, 73:231–248, 1990.
20. M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.