Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Type-based Initialization Analysis of a Synchronous Data-flow Language*

Jean-Louis Colaço[1], **Marc Pouzet**[2]

[1] LIP6, Université Pierre et Marie Curie, 8 rue du Capitaine Scott, 75015 Paris, France.
e-mail: Marc.Pouzet@lip6.fr
[2] ESTEREL Technologies, Park Avenue 9, Rue Michel Labrousse 31100 Toulouse, France
e-mail: Jean-Louis.Colaco@esterel-technologies.com

January 2004

Abstract. One of the appreciated features of the synchronous data-flow approach is that a program defines a perfectly deterministic behavior. But the use of the delay primitive leads to undefined values at the first cycle; thus a data-flow program is really deterministic only if it can be shown that such undefined values do not affect the behavior of the system.

This paper presents an *initialization analysis* that guarantees the deterministic behavior of programs. This property being undecidable in general, the paper proposes a safe approximation of the property, precise enough for most data-flow programs. This analysis is a *one-bit* analysis — expressions are either initialized or uninitialized — and is defined as an inference type system with sub-typing constraints. This analysis has been implemented in the Lucid Synchrone compiler and in a new Scade-Lustre prototype compiler at Esterel-Technologies. The analysis gives very good results in practice.

Key words: Synchronous data-flow languages – Lustre – Type systems with sub-typing – Program analysis.

1 Introduction

Since its definition in the early eighties, Lustre [8] has been successfully used by several industrial companies to implement safety critical systems in various domains like nuclear plants, civil aircrafts, transport and automotive systems. All these development have been done using Scade [14], a graphical environment based on Lustre and distributed successively by Verilog SA, Telelogic and now Esterel Technologies.

Lustre is a first-order *functional* language managing infinite sequences or *streams* as primitive values. These streams are used for representing input/outputs in a real-time system. Lustre is well suited for real-time critical systems constraints thanks to its well formalized semantics, its associated verification tools and its ability to be translated into simple imperative programs for which some fundamental properties can be ensured (e.g., bounded execution time and memory).

In order to "break" data-flow loops and then define a causally correct specification, explicit delays must be introduced. Because these delays are not initialized, they may introduce undefined values. These undefined values may be responsible for non deterministic behavior. The Scade-Lustre compiler analyses this risk. Nonetheless, the analysis is too much conservative and often forces the programmer to add extra initializations.

The purpose of this paper is to present a new initialization analysis which improves the existing one and is fast enough to be used in a development process of large programs. The analysis has been designed originally for Lucid Synchrone [4,5,13]. While keeping the fundamental properties of Lustre, Lucid Synchrone provides powerful extensions such as higher-order features, data-types, type and clock inference. The compilation also performs a causality analysis and an initialization analysis which is the subject of the present paper. Lucid Synchrone is used today by the Scade team at Esterel Technologies for testing extension of Scade and the design of a new compiler.

1.1 Contribution

This paper presents an initialization analysis for a synchronous data-flow language providing uninitialized unary delay and a separate initialization operator. We express it as a standard typing problem with sub-typing con-

* A preliminary version of this paper appears in [6].

straints and we base it on classical resolution techniques of the field.

The paper does not present new theoretical result in the field of type systems with sub-typing. Rather, it shows the adequacy of a type-based formulation of the initialization problem and the use of conventional techniques to solve it.

The analysis has been implemented in the industrial Lustre compiler prototype called ReLuC (*Retargetable Lustre Compiler*) and in the Lucid Synchrone compiler. Compared to the actual Scade implementation, the analysis is more accurate: most of the time, rejected programs do produce undefined results. The analysis is also faster: this is particularly important in an industrial setting where the analysis should be able to check real size programs (several thousand lines of code) before any simulation start. The analysis is modular in the sense that the initialization information of a node reduces to the initialization information of its definition. This initialization information is a type, i.e., an abstraction of value with respect to the initialization problem. Modularity is partly responsible for efficiency. Finally, it appears to give good diagnostics in practice in the sense that it is often enough to insert an initialization where the error occurs in order to obtain a correct program that will be analyzed successfully.

The paper is organized as follows. Section 2 gives the main intuitions of our initialization analysis and motivation. Section 3 formalizes the analysis on a higher-order data-flow synchronous kernel in which Lustre and Lucid Synchrone can be easily translated. It then establishes the correctness theorem stating that *"well typed programs are well initialized"*. Section 4 discuss implementation approaches taken by both compilers. In section 5, we come back to the need of un-initialized delays in data-flow languages and alternative or existing approaches for the initialization problem. Section 6 is the conclusion.

2 Initialization Analysis: intuitions

Because this work has been done for two different languages with their own syntax, we base our presentation on the more abstract syntax presented in section 3.

Synchronous data-flow languages manage infinite sequences or *streams* as primitive values. For example, `1` stands for an infinite constant sequence and primitives imported from a host language are applied point-wise to their argument. These languages provide also a unary delay `pre` (for *p*revious) and an initialization operator `->` (for *f*ollowed-by).

`x`	$x_0\ x_1\ x_2\ x_3\ \ldots$
`pre x`	$nil\ x_0\ x_1\ x_2\ \ldots$
`y`	$y_0\ y_1\ y_2\ y_3\ \ldots$
`y -> x`	$y_0\ x_1\ x_2\ x_3\ \ldots$
y `-> pre` x	$y_0\ x_0\ x_1\ x_2\ \ldots$
y `-> pre (pre` x`)`	$y_0\ nil\ x_0\ x_1\ \ldots$

The initial value of `pre x` is undefined and noted *nil*. These undefined values can be eliminated by inserting an initialization operator (`->`). It is easy to see that combining these two primitives leads to an always defined delay `->pre`. In the context of critical system programming, we need to check that these undefined values will not affect the behavior of the program.

A trivial analysis consists in verifying that each delay is syntactically preceded by an initialization. Nonetheless, an equation is often defined by separating the invariant part (written with `pre`) from its initialization part (written with `->`) as in the following example:

```
node switch c = o
  with o = c -> if c then not (pre o)
                else pre o
```

The syntactic criteria is unable to verify that this function is indeed correct, that is, `o` does not contain any *nil* values. Moreover, it is often useful to define a function that does not necessarily initialize all its output and does not need its input to be initialized, leaving the initialization problem to the calling context of the function. Thus, intermediate *nil* values should be accepted.

Moreover, a too severe criteria can lead to redundancies and useless code due to over initialized expressions. At least, these useless initializations make the code less clear and there is little evidence that they will be eliminated by the compiler.

Another simple approach could consist in giving a default initial value to delays (e.g., `false` in the case of boolean delays) in pretty much the same way as C compilers give initial values to memories. This is by no means satisfactory in the context of critical systems programming since programs would rely on an implicit information. Moreover, when considering an implementation into circuits, forcing the initialization of delays (i.e, *latches*) cost extra wires which must be avoided as much as possible.

An ambitious approach could be to use verification tools (e.g., NP-tools, Lesar) by translating the initialization problem into a boolean problem. Though theoretically possible, this approach may be quite expensive; it is not modular (at least for higher-order) and certainly useless for many programs. Moreover, keeping precise and comprehensive diagnostics — which is crucial — is far from being easy.

These considerations have led us to design a specific initialization analysis. We adopt a very modest *one-bit*

approach where streams are considered to be either always defined or maybe undefined at the very first instant only. This leads to a natural formulation in terms of a type-system with sub-typing. In this system, an expression which is always defined receives the type $\mathbf{0}$ whereas an expressions whose first value may be undefined receives the type $\mathbf{1}$. Then we use sub-typing to express the natural assumption: *"an initialized stream can be used where an uninitialized one is expected"*, that is, $\mathbf{0} \leq \mathbf{1}$. Consider, for example:

```
node deriv x = s with
  s = x - pre x
```

The function gets type $\mathbf{0} \to \mathbf{1}$ meaning that its input `x` must be initialized and its output `s` is not. Indeed, `pre x` has type $\mathbf{1}$ if `x` has type $\mathbf{0}$. The (`-`) operator needs its two arguments to share the same type, thus, after weakening the type of `x`, `s` receives the type $\mathbf{1}$. Thus, the following program is rejected.

```
node deriv2 x = s with
  s = deriv (deriv (x))
            ^^^^^^^^^^
```

Indeed, the (underlined) expression `deriv x` has type $\mathbf{1}$ whereas `deriv` expects a value with type $\mathbf{0}$. Thanks to the modularity of the analysis, the compiler is able to give a precise diagnostic of the error by pointing out the expression whose type is different from what is expected.

Type variables are naturally introduced for expressing type constraints between program variables. Consider, for example:

```
node min (x,y) = z
  with z = if x <= y then x else y
```

`min` receives the polymorphic type scheme $\forall\delta.\delta \times \delta \to \delta$ stating that for any type δ ($\delta \in \{\mathbf{0}, \mathbf{1}\}$), if the input `x` and `y` have the type δ, then the output `z` receives the same type δ. Thus, `min` can be used with two different type instances $\mathbf{0} \times \mathbf{0} \to \mathbf{0}$ or $\mathbf{1} \times \mathbf{1} \to \mathbf{1}$.

```
node low (x,y) = l
  with m = min (x,y)
  and  l = x -> min (m, pre l)
```

The first instance of `min` is used with type $\mathbf{0} \times \mathbf{0} \to \mathbf{0}$ whereas the second one is used with type $\mathbf{1} \times \mathbf{1} \to \mathbf{1}$. Finally, the function `low` receives the type $\mathbf{0} \times \mathbf{0} \to \mathbf{0}$ since `m` has type $\mathbf{0}$ which can be weakened into type $\mathbf{1}$ and `pre l` has type $\mathbf{1}$.

The *one-bit* abstraction comes from the observation that equations are often written by separating the initialization part from the invariant part and from the charateristic of the initialization operator. Indeed, an initialization operator cannot eliminate the *nil* value appearing at the second instant in an expression `pre(pre x)` [1]. This explains why such expressions are rarely used in common programs. Of course, using a `pre(pre x)` does not necessarily lead to an incorrect program (see section 3.5). Accepting them should rely on boolean properties of programs and we reject expressions like `pre(pre x)`.

In practice, the *one-bit* abstraction, while being modest, gives remarkably good results. The initialization analysis is modular, it addresses Lucid Synchrone as well as Lustre programs and can be implemented using standard techniques.

3 Formalization

This section presents a synchronous data-flow kernel, defines its data-flow semantics. It gives also its associated initialization analysis presented as a standard type system with sub-typing constraints.

3.1 A Synchronous Kernel and its Data-flow Semantics

An expression e may be an immediate constant (i), a variable (x), the point-wise application of an operator to a tuple of inputs ($op(e_1, e_2)$) [2], the application of a delay (`pre`) or initialization (`->`), a conditional (`if` e `then` e `else` e), an application ($e(e)$), a local definition of streams (e `with` D), a function definition (`node` $f\,x = e$ `in` e) or a pair (e, e) and its access functions.

A set of definitions (D) contains equations between streams ($x = e$). These equations are considered to be mutually recursive.

The kernel may import immediate constants (i) from a host language (e.g., a boolean or an integer) or functional values (op) which are applied point-wisely to their inputs.

$$
\begin{array}{rl}
e & ::= i \mid x \mid op(e,e) \mid \texttt{pre}\ e \\
 & \quad \mid e\ \texttt{->}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e(e) \\
 & \quad \mid e\ \texttt{with}\ D \mid \texttt{node}\ f\,x = e\ \texttt{in}\ e \\
 & \quad \mid (e,e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
D & ::= x = e\ \texttt{and}\ D \mid x = e \\
i & ::= \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid ... \\
op & ::= + \mid ...
\end{array}
$$

This kernel is essentially an ML kernel managing sequences as primitive values. We give it a classical Kahn semantics [9] that we remind here shortly [3]. Let T^ω be the set of finite or infinite sequences of elements over the set T ($T^\omega = T^* + T^\infty$). The empty sequence is noted ϵ and $x.s$ denotes the sequence whose head is x and tail is s. We also consider $T_{nil} = T + nil$ as the set T complemented with a special value *nil*. If s is a sequence,

[1] This is in contrast with classical imperative languages where, once a variable is assigned, it keeps its value until its next modification.

[2] For simplicity, we only consider binary operators in this presentation.

[3] We keep here the original notion of the paper.

$$
\begin{aligned}
i^{\#} &= i.i^{\#} \\
\\
op^{\#}(s_1, s_2) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(v_1.s_1, v_2.s_2) &= (v_1\, op\, v_2).op^{\#}(s_1, s_2) \\
&\quad \text{if } v_1 \neq nil \text{ and } v_2 \neq nil \\
op^{\#}(v_1.s_1, v_2.s_2) &= nil.op^{\#}(s_1, s_2) \\
\\
\texttt{if}^{\#}(s_1, s_2, s_3) &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\texttt{if}^{\#}(nil.s, v_2.s_2, v_3.s_3) &= nil.\texttt{if}^{\#}(s, s_2, s_3) \\
\texttt{if}^{\#}(true.s, v_2.s_2, v_3.s_3) &= v_2.\texttt{if}^{\#}(s, s_2, s_3) \\
\texttt{if}^{\#}(false.s, v_2.s_2, v_3.s_3) &= v_3.\texttt{if}^{\#}(s, s_2, s_3) \\
\\
\texttt{pre}^{\#}(s) &= nil.s \\
\\
s_1 \texttt{->}^{\#} s_2 &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
(v_1.s_1) \texttt{->}^{\#} (v_2.s_2) &= v_1.s_2
\end{aligned}
$$

Fig. 1. Data-flow semantics for primitives

$$
\begin{aligned}
S_\rho(op(e_1, e_2)) &= op^{\#}(S_\rho(e_1))(S_\rho(e_2)) \\
S_\rho(\texttt{pre}\ e) &= \texttt{pre}^{\#}(S_\rho(e)) \\
S_\rho(e_1\ \texttt{->}\ e_2) &= (S_\rho(e_1))\ \texttt{->}^{\#}\ (S_\rho(e_2)) \\
S_\rho(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \texttt{if}^{\#}(S_\rho(e_1), S_\rho(e_2), S_\rho(e_3)) \\
S_\rho(x) &= \rho(x) \\
S_\rho(i) &= i^{\#} \\
S_\rho(\texttt{node}\ f\ x = e_1\ \texttt{in}\ e_2) &= S_{\rho[f \leftarrow \lambda y.S_{\rho[x \leftarrow y]}(e_1)]}(e_2) \\
&\quad \text{where } y \notin Dom(\rho) \\
S_\rho(e_1(e_2)) &= (S_\rho(e_1))(S_\rho(e_2)) \\
S_\rho(e\ \texttt{with}\ D) &= S_{S_\rho(D)}(e) \\
S_\rho(e_1, e_2) &= (S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(\texttt{fst}\ e) &= v_1 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(\texttt{snd}\ e) &= v_2 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(x = e) &= \rho[x \leftarrow x^\infty] \text{ where} \\
&\quad x^\infty = fix\ \lambda y.S_{\rho[x \leftarrow y]}(e) \\
&\quad \text{with } y \notin Dom(\rho) \\
S_\rho(x = e\ \texttt{and}\ D) &= S_{\rho[x \leftarrow x^\infty]}(D) \text{ where} \\
&\quad x^\infty = fix\ \lambda y.S_{(S_{\rho[x \leftarrow y]}(D))}(e) \\
&\quad \text{with } y \notin Dom(\rho)
\end{aligned}
$$

Fig. 2. Data-flow semantics

$s(i)$ denotes the i-th element of s if it exists. If $s = x.s'$ then $s(0) = x$ and $s(i) = s'(i-1)$. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if x is a prefix of y. The ordered set $(T^\omega, \leq)$ is a cpo. If f is a continuous mapping from sequences to sequences, we shall write $fix\ f = lim_{n \to \infty} f^n(\epsilon)$ for the smallest fix point of f.

If $T_1, T_2, ...$ are set of scalar values (typically values imported from a host language), we define the domain V as the smallest set containing $T_i{}^\infty_{nil}$ and closed by product and exponentiation.

For any assignment ρ (mapping values to variable names) and expressions e, we define the denotation of an expression e by $S_\rho(e)$. We overload $S(.)$ such that $S_\rho(D)$ defines the denotation of stream equations. We first give in figure 1 the interpretation over sequences for every data-flow primitive. We use the mark # to give the semantics of primitives over infinite sequences. For example, $i^{\#}$ stands for the infinite sequence made of an immediate value; $op^{\#}$ stands for the interpretation of the imported primitive op. A primitive returns ϵ as soon as one of its inputs is ϵ. Otherwise, it is applied point-wise to its inputs. When one input item is *nil*, the current output is also *nil*. The delay prefixes the value *nil* to its inputs and the initialization operator prefixes the head of its first input with the tail of its second input.

We can easily check that the above primitives are continuous. The denotational semantics for other constructions is given in figure 2. The denotational semantics is defined as usual [10]. The `node/in` construct introduces stream functions. The semantics of a set of equations is the least fix point of the associated stream function.

Definition 1 (Well initialized) *An expression e is well initialized from k ($k \in \mathbb{N}$) in an environment ρ, noted $\rho \models e : k$ if $\mathrm{S}_\rho(e) \neq \epsilon$ implies that for all $n \geq k$, if $\mathrm{S}_\rho(e)(n)$ exists then $\mathrm{S}_\rho(e)(n) \neq \mathrm{nil}$. An expression is well initialized from k if for any ρ, $\rho \models e : k$. This is noted $\models e : k$.*

Notice that causality loops (such as `x = x + 1`) define streams with the value ϵ (for which the n-th element is not defined). Since the paper focuses on the initialization problem only, causality loops are considered well initialized. In practice, these loops are rejected by another analysis [7] which is applied before the initialization analysis.

The initialization analysis will be restricted to a *one-bit* analysis, i.e., a stream expression always verify either $\models e : 0$ or $\models e : 1$.

3.2 The Type System

We use polymorphism to express the relationship between input and output initialization facts in operators and nodes, and sub-typing to express the natural assumption : *"an initialized flow can be used where an uninitialized one is expected"*.

3.2.1 The Initialization Type Language

Types are separated into type schemes (σ) and regular types (t). A type scheme is a type quantified over type variables (α) and initialization type variables (δ), together with a set of type constraints (C).

A type (t) may be a type variable (α), a function type ($t \to t$), a product type ($t \times t$) or an initialization type (d) for a stream value. An initialization type (d) may be **0** meaning that the stream is always defined; **1** meaning that the stream is well defined at least after the

first instant or a variable (δ). C is a set of constraints between initialization type variables.

$$\begin{aligned}\sigma &::= \forall\alpha_1,...,\alpha_k.\forall\delta_1,...,\delta_n : C.t \text{ with } k \geq 0,\ n \geq 0\\ t &::= \alpha \mid t \to t \mid t \times t \mid d\\ d &::= \mathbf{0} \mid \mathbf{1} \mid \delta\\ H &::= [x_1 : \sigma_1, ..., x_n : \sigma_n]\\ C &::= [\delta_1 \leq \delta'_1, ..., \delta_n \leq \delta'_n]\end{aligned}$$

We define the predicate $C \Rightarrow t_1 \leq t_2$ stating that under the hypothesis C, t_1 is a sub-type of t_2. Its definition is given in figure 3. The sub-type relation is built from the relation between base types (rules (TRIV)), extends it to function and product types (with a contra-variant rule for function types) and adds a transitivity rule. The relation is lifted to sets of inequations (rules (SET) and (EMPTY)).

3.2.2 Initial conditions, Instantiation and Generalization

Expressions are typed in an initial environment H_0:

$$\begin{aligned}H_0 = [&\texttt{pre} : \mathbf{0} \to \mathbf{1},\\ &(\texttt{ -> }) : \forall\delta.\delta \to \mathbf{1} \to \delta,\\ &\texttt{if . then . else .} : \forall\delta.\delta \to \delta \to \delta \to \delta,\\ &\texttt{fst} : \forall\alpha_1,\alpha_2.\alpha_1 \times \alpha_2 \to \alpha_1,\\ &\texttt{snd} : \forall\alpha_1,\alpha_2.\alpha_1 \times \alpha_2 \to \alpha_2]\end{aligned}$$

The delay operator (`pre`) imposes its inputs to be always initialized whereas the initialization operator (`->`) does not impose any constraints.

We define $FV(t)$ as the set of free type variables (α, δ) and lift it to type scheme and environments. The definition classical and reminded in appendix A.

Types can be instantiated or generalized. A type scheme $\forall\alpha_1,...,\alpha_n.\forall\delta_1,...,\delta_k : C.t$ may be instantiated by applying a substitution to its bound type variables and to its initialization type variables. The resulting set of constraints $C[d_1/\delta_1, ..., d_k/\delta_k]$ must be correct with respect to the sub-type relation, i.e, derivable from the definition of $\Rightarrow$. Every type variable or initialization type variable may be generalized if it is free in the environment H. Since these variables may appear in constraints, constraints are added to type schemes. Type instantiation is such that:

1. $(t, C) \in inst(t)$
2. $(t[t_1/\alpha_1, ..., t_n/\alpha_n][d_1/\delta_1, ..., d_k/\delta_k], C')$
   $\in inst(\forall\alpha_1, ..., \alpha_n.\forall\delta_1, ..., \delta_k : C.t)$
   if $C' \Rightarrow C[d_1/\delta_1, ..., d_k/\delta_k]$

Type generalization is such that:

1. $gen_{H|C}(t) = \forall\alpha_1, ..., \alpha_k.\forall\delta_1, ..., \delta_n : C.t$
   where $\{\alpha_1, ..., \alpha_k, \delta_1, ..., \delta_n\} = FV(t) - FV(H)$

3.2.3 The Type System

The initialization analysis of an expression is obtained by stating judgments of the form:

$$H \mid C \vdash e : t \qquad H \mid C \vdash D$$

The first judgment means that an expression e has initialization type t with sub-typing constraints C in an environment H. The second one means that the recursive definition D is well initialized in the environment H and under the set of constraints C.

The definition of these two predicates is given in figure 4. This is a classical type-system with sub-typing constraints [1,11,12]. We adopt a simpler (and less general) presentation adapted to the initialization problem.

- a constant receives the type $\mathbf{0}$, saying that it is always defined;
- primitive operators need their arguments to have the same type;
- when typing a node declaration, we first type its body, generalize its type and then type the expression where it may be used. Polymorphism is only introduced at node declarations;
- applications are typed in the usual way;
- `with` declarations are considered to be recursive. This is why H_0 appears on the left of the typing judgment. Note also that equations contained in declarations are necessarily stream equations (rule (EQ)) since they must have an initialization type d;
- scheme types can be instantiated (rule (INST));
- finally, a type may be weakened (rule (SUB)).

Theorem 1 (Correctness). *For all expressions e, if $\vdash e : d$ then $\models e : d$*

The proof follows a technique relating typing and evaluation semantics. It is given in appendix B.

3.3 An Example

We illustrate the analysis on the following example.

```
node sum (x, y, z) = o where
  o = (x -> y) + z
```

`sum` receives the initialization type $\forall\delta.\delta \times \mathbf{1} \times \delta \to \delta$. Let us see how this type is obtained.

Variables are first introduced in the typing environment. We have $H = [\texttt{x} : \delta_1, \texttt{y} : \delta_2, \texttt{z} : \delta_3]$. As usual in type systems with sub-typing, sub-typing rules must be applied at every application point thus, `(x -> y)` receives some type δ_4 under the constraint $C_1 = \{\delta_1 \leq \delta_4\}$ (the constraint $\delta_2 \leq \mathbf{1}$ is verified by definition). Then `(x -> y) + z` receives the type δ_5 and the new set of constraints is $C_2 = \{\delta_1 \leq \delta_4, \delta_4 \leq \delta_5, \delta_3 \leq \delta_5\}$. Thus, the final type could be $\forall\delta_1, \delta_2, \delta_3, \delta_4, \delta_5 : C_2.\delta_1 \times \delta_2 \times \delta_3 \to \delta_5$. Nonetheless, this type is overly complex and can be replaced by a simpler one, without loss in generality [2,12].

$$\text{(TRIV-1)}\ C \Rightarrow d \leq \mathbf{1} \quad \text{(TRIV-2)}\ C \Rightarrow \mathbf{0} \leq d \quad \text{(TRIV-3)}\ C \Rightarrow \delta \leq \delta$$

$$\text{(FUN)}\frac{C \Rightarrow t_3 \leq t_1 \quad C \Rightarrow t_2 \leq t_4}{C \Rightarrow t_1 \to t_2 \leq t_3 \to t_4} \quad \text{(PROD)}\frac{C \Rightarrow t_1 \leq t_3 \quad C \Rightarrow t_2 \leq t_4}{C \Rightarrow t_1 \times t_2 \leq t_3 \times t_4}$$

$$\text{(TRANS)}\frac{C \Rightarrow d_1 \leq d_2 \quad C \Rightarrow d_2 \leq d_3}{C \Rightarrow d_1 \leq d_3} \quad \text{(SET)}\frac{C \Rightarrow C_1 \quad C \Rightarrow C_2}{C \Rightarrow C_1, C_2}$$

$$\text{(TAUT)}\ C, \delta_1 \leq \delta_2 \Rightarrow \delta_1 \leq \delta_2 \quad \text{(EMPTY)}\ C \Rightarrow \emptyset$$

Fig. 3. Subtype relation

$$\text{(IM)}\ H \mid C \vdash i : \mathbf{0} \qquad \text{(OP)}\frac{H \mid C \vdash e_1 : d \quad H \mid C \vdash e_2 : d}{H \mid C \vdash op(e_1, e_2) : d}$$

$$\text{(NODE)}\frac{H, x : t \mid C \vdash e_1 : t_1 \quad H, f : gen_{H|C}(t \to t_1) \mid C \vdash e_2 : t_2}{H \mid C \vdash \texttt{node}\ f\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

$$\text{(APP)}\frac{H \mid C \vdash e_1 : t_2 \to t_1 \quad H \mid C \vdash e_2 : t_2}{H \mid C \vdash e_1\,(e_2) : t_1}$$

$$\text{(DEF)}\frac{H, H_0 \mid C \vdash D \quad H, H_0 \mid C \vdash e : t}{H \mid C \vdash e\ \texttt{with}\ D : t}$$

$$\text{(PAIR)}\frac{H \mid C \vdash e_1 : t_1 \quad H \mid C \vdash e_2 : t_2}{H \mid C \vdash (e_1, e_2) : t_1 \times t_2} \quad \text{(EQ)}\frac{H[x, d] \mid C \vdash e : d}{H[x, d] \mid C \vdash x = e}$$

$$\text{(AND)}\frac{H \mid C \vdash D_1 \quad H \mid C \vdash D_2}{H \mid C \vdash D_1\ \texttt{and}\ D_2}$$

$$\text{(INST)}\frac{C, t \in inst(H(x))}{H \mid C \vdash x : t} \qquad \text{(SUB)}\frac{H \mid C \vdash e : t \quad C \Rightarrow t \leq t'}{H \mid C \vdash e : t'}$$

Fig. 4. The type system

The technique consists in identifying the variables that are in a co-variant position (δ_5) and those in a contra-variant position ($\delta_1, \delta_2, \delta_3$). If a variable is exclusively present in a co-variant (monotonic) position, it can be replaced by its lower bound; if it is exclusively in a contra-variant (anti-monotonic) it can be replaced by its upper bound. These substitution can be done without loss of generality of the type thanks to the sub-typing relation. Those that appear in both position cannot disappear without loosing some generality. Applying this to the previous type leads to: $\forall\delta_5.\delta_5 \times \mathbf{1} \times \delta_5 \to \delta_5$ since $\delta_3 \leq \mathbf{1}$.

3.4 The First Order Case

The previous presentation introduces the most general case with higher-order constructions. Because Lustre is a first order language, it can be specified inside our synchronous kernel by imposing some syntactic constraints overs expressions and types.

The type language is a subset of the previous one, by taking:

$$\begin{aligned} \sigma &::= \forall\delta_1, ..., \delta_n : C.b \to b \\ t &::= b \to b \mid b \\ b &::= d \mid b \times b \\ d &::= \mathbf{0} \mid \mathbf{1} \mid \delta \end{aligned}$$

Up to syntactic details, valid LUSTRE expressions are characterized in the following way.

$$\begin{array}{lll} node & ::= & \texttt{node}\ f\ x = y\ \texttt{with}\ D \\ e & ::= & i \mid x \mid op(e,e) \mid \texttt{pre}\ e \mid e\ \texttt{->}\ e \mid x(e) \\ & & \mid (e,e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \end{array}$$

In LUSTRE, functions (named nodes) must be defined at top level and cannot be nested.

3.5 Limitations

Taking a modest *one-bit* abstraction will clearly reject valid programs. We illustrate the limitation of the analysis on programs computing the *Fibonacci* sequences. At least two versions can be considered. For example:

```
node fib dummy = x
  with x = 1 -> pre (1 -> x + pre x)
```

`dummy` is a useless parameter. The program is accepted by the compiler. The following program also defines the *Fibonacci* sequence:

```
node repeat n = c
  with c = true -> (count >= 1) & pre c
   and count = n -> if pre count >= 0
                    then pre count - 1 else 0

node fib dummy = x
  with x = if repeat 2 then 1
           else pre x + pre (pre x)
```

`repeat n` is true during `n` instants and then false forever. This program has a perfectly valid semantics. Nonetheless, it is rejected by our analysis since the fact that the second values of `pre(pre x)` is never accessed depends on semantics conditions which are not checked.

3.6 Clocks and initialization analysis

LUSTRE and LUCID SYNCHRONE are based on the notion of *clocks* as a way to combine slow and fast processes. LUSTRE provides an operator `when` for filtering a stream according to a boolean condition and an operator `current` projecting a stream on the immediately faster clock. LUCID SYNCHRONE does not provide `current` but instead an operator `merge` which combines two complementary streams. Their behavior is given in figure 5.

The initialization analysis checks that all the boolean flows used as clocks are always defined (they receive the type $\mathbf{0}$). Indeed, a *nil* value in a clock could lead to an undetermined control-flow. Thus, a safe approximation consists in giving to the `when` operator the initialization type $\forall\delta.\delta \to \mathbf{0} \to \delta$. For the same reason, `merge` receives type $\forall\delta.\mathbf{0} \to \delta \to \delta \to \delta$.

The present analysis is applied to the whole LUCID SYNCHRONE language with no restriction. In particular, sampling operators receive the initialization types given above.

`x`	$x_0\ x_1\ x_2\ x_3\ x_4\ x_5\ \ldots$
`y`	$y_0\ y_1\ y_2\ y_3\ y_4\ y_5\ \ldots$
`h`	$f\ f\ t\ t\ f\ t\ \ldots$
`x when h`	$\quad\quad x_2\ x_3\quad x_5\ \ldots$
`y whenot h`	$y_0\ y_1\quad\quad y_4\quad\ \ldots$
`current (x when h)`	$nil\ nil\ x_3\ x_4\ x_4\ x_6\ \ldots$
`merge h (x when h) (y whenot h)`	$y_0\ y_1\ x_2\ x_3\ y_4\ x_5\ \ldots$

Fig. 5. Clock operators

The `current` of LUSTRE raises specific difficulties. Indeed, the operator may introduce *nil* values as long as its input clock is not true at the first instant. Since boolean values (and in particular clocks) are not interpreted specifically, our method fails on analyzing the `current` operator.

The `current` operator is not a primitive operator in LUCID SYNCHRONE and should be programmed. Following the syntax of the paper, we should write:

```
node current (init, clk, i) = o with
      po = init -> pre o
  and o = merge clk i (po whenot clk)
```

and the initialization analysis will force to give an initial value `init` to `po`.

We did not find any real application in SCADE using the `current` operator: sampling is made through the *activation condition* operator `condact` which imposes to give an initial value. `condact(clk,f,i,init)` takes a clock `clk`, a node `f`, an input `i` and an initial value `init` and it applies the node `f` to the input sampled on clock `clk`. It emits the previous value initialized with `init` when the clock is false. In our system, `condact` receive the initialization type: $\mathbf{0} \times (\mathbf{0} \to \mathbf{0}) \times \mathbf{0} \times \mathbf{0} \to \mathbf{0}$.

Extending the analysis in order to take the `current` operator into account has not been considered so far and is a matter of future work.

3.7 Partial functions

Up to now, we have focused on the presence of *nil* values in memories or in clocks. Once the analysis have been performed, it is proven that the behavior of the program does not depend on these *nil* values. In both SCADE and LUCID SYNCHRONE, there is no explicit representation of *nil* at run-time and the compilers may choose any value of the correct type.

Partial functions must be taken into account when dealing with the initialisation problem. For example, several imported primitives (e.g., floating-point division /,

integer division, modulus) are not defined if their second argument is zero. This means that a computation a/b where b is *nil* at the very first instant, may lead to a division by zero just because the compiler may choose to represent this *nil* value by zero.

Imported partial primitives are taken into account by choosing an appropriate type for them. For example, the floating-point division / gets the type $\forall\delta.\delta \times \mathbf{0} \to \delta$, stating that the second argument must always be well initialized [4]. The treatment of partial primitives is done in both SCADE and LUCID SYNCHRONE compilers.

Note that we do not claim to statically detect division by zero, but to avoid to have the user puzzled by a division by zero that doesn't appear explicitely in the code.

4 Implementation Issues

Being a classical typing problem with sub-typing, standard implementation techniques as proposed in [2,12] has been considered. The analysis has been implemented both in the LUCID SYNCHRONE compiler and in the RELUC compiler. Nonetheless, we have experimented quite different techniques in the representation of types and the way type simplification is performed.

In both compilers, the analysis is performed after typing and infered types are used for generating valid initialization type skeletons. This approach leads to a strong simplification of the implementations.

4.1 Implementation in the Lucid Synchrone Compiler

In this implementation, we have adopted a relative unusual representation of type and constraints, taking advantage of the simple form of types.

- sub-typing constraints are directly annotated on types instead of being gathered in a global table. The principle is the following: a dependence variable δ points to its list of predecessors and successors. Thus, internally, the type language becomes:
$$d ::= \mathbf{0} \mid \mathbf{1} \mid \delta^{\leq\rho}_{\geq\rho} \quad \rho ::= d, \rho \mid \emptyset$$
- when a variable is unified with $\mathbf{0}$, its predecessors are unified with $\mathbf{0}$; when a variable is unified with $\mathbf{1}$, its successors also.
- the sub-typing rule is only done at instantiation points (i.e, variable occurrences), using type skeletons computed during the type inference phase.
- the efficiency of a type system with sub-typing relies on the simplification of sub-typing constraints. We based our implementation on the computation of polarities as studied by Pottier [12]. Type simplification is performed at generalization points only (i.e, function definitions).

[4] On the contrary, a total function like the integer addition + will get the type $\forall\delta.\delta \times \delta \to \delta$.

Application domain	nb. lines	analysis time
transport	$\sim$ 17000	$\leq$ 2 sec.
helicopter syst.	$\sim$ 30000	$\leq$ 2 sec.
aircraft syst.	$\sim$ 11000	$\leq$ 1 sec.
automotive syst.	$\sim$ 1000	$\leq$ 0.1 sec.

Fig. 6. Benchmarks on RELUC

These optimizations lead to an efficient implementation which is an order of magnitude faster than the implementation in the RELUC compiler.

4.2 Implementation in the ReLuC Lustre Compiler

As in the case of LUCID SYNCHRONE, a type variable is introduced for all streams and the set of constraint is solved at each generalization point (i.e., at node definitions). The type is simplified by saturation of monotonic type variables; note that in a first order language, all the introduced type variables are monotonic. This allows for a very simple representation of the constraint in the types by adding a type union $\sqcup$. With this extension, the type of a `add` node is $\forall\delta_1, \delta_2.\delta_1 \times \delta_2 \to \delta_1 \sqcup \delta_2$.

The figure 6 gives some benchmarks of the RELUC implementation on some real applications [5]. These applications contains a lot of nodes having several hundreds of inputs and outputs.

As the analysis may fail on correct programs (consider a well initialized program not to be initialized), the choice in SCADE is to prevent the user with a warning and let him prove by other means that it is effectively correct or use the diagnosis to write a checkable specification by adding an extra initialization.

The RELUC initialization analyzer has been used to validate the SCADE library and it has been applied to several big models. These experiments confirm the accuracy and the applicability of the approach in an industrial context. This is a great improvement of the available SCADE semantic check which is safe but remains at the level of the trivial analysis discussed in 2. It should be integrated in a future version of the SCADE compiler.

5 Discussion

In the introduction, we have motivated the interest of un-initialized delays in synchronous data-flow languages which call for a dedicated initialization analysis. In this section, we elaborate further on this point, discuss alternative solutions for the initialization problem and compare the analysis to the existing one for SCADE.

In the early days of LUCID SYNCHRONE there was no un-initialized delays and thus a need for any initialization analysis. One had to write `pre` i e for the delay initialized with the static value i. Then, we introduced the

[5] The compiler was running on a Pentium 3 (800Mhz) processor.

more general initialized delay operator `fby` of LUCID [3] where the delay is initialized with the first element of a stream ($x.xs$ `fby` $b = x.b$, that is, a `fby` $b = a$ `->` `pre` b). Nonetheless, it appeared that program writing was less natural and elegant than in a language providing also un-initialized delays such as LUSTRE. This is why we finally incorporate the LUSTRE `pre` operator. Indeed, data-flow equations are often defined by *separating* the invariant part (written with `pre`) from the initialization part (written with `->`). Using only initialized delays tends to create extra redundancies and forces to find an initial value for some streams whereas it is clear that their value is useless at the very first instant.

To illustrate this, consider the discrete derivative function that computes the difference of its current input with the previous one:

```
let deriv x = x - pre i x
```

what would be a correct value for `i`? Is-it `0` or some initial value `i` given as an extra parameter? If the user of this code is not satisfied with some value `i`, he may write in turn `dx0 -> deriv x` but then, `i` becomes useless! The only good program point to put this initial condition is out of the minus operator and then far away from the delay. When `deriv` is a library operator, only the user of the operator knows what is the good initial value.

Providing only initialized delays may lead to an unnatural programming discipline which could be compared to the necessity to declare a variable with an initial value in an imperative language. The risk with this discipline is that it is not always relevant to do this and a lot of algorithms use variables whose first value cannot be given at the beginning. As a consequence the user would sometimes give a dummy value to satisfy the discipline. The intention of the programmer would, as a consequence, be less evident because of the presence of these dummy values. In a language that claims to address safety critical applications we believe this must be avoided as much as possible. In some data-flow tools (e.g., SIMULINK), delays and statefull primitives must always receive an extra initialization value. Thus, systems are well initialized by construction but with the weakness mentioned previously. We think that providing initialized and uninitialized delays together with a simple initialization check leads to a more natural writing of programs.

An ambitious approach to solve the initialization problem could be to rely on boolean solvers (e.g., NP-tools, LESAR). This approach consists in generating an invariant stating that the output of a program does not contain any *nil* value. If such a proof fails a special tool to trace back to the initialization problem on the source code may be necessary in order to obtain a useful diagnostic. This method is certainly more powerful in the sense that more complicated situation can be analyzed (when general boolean properties are responsible for the correct initialization). Nonetheless, the verification is not modular and has to be done on the whole program. On the contrary, a dedicated type system is modular in essence. Programs are analyzed structurally making diagnostics easier when an error is encountered. When the type discipline is clear, the user knows how to write programs that will pass the type checking and is able to fix, at least locally, initialization errors. Moreover, initialization types are part of the specification of a node, as it is usually admitted for conventional types.

The academic LUSTRE compiler doesn't check at all the good initialization of streams and the SCADE/LUSTRE compiler do it with a too simple algorithm that is not able to go through equations nor nodes. Thanks to its modularity the initialization analysis is strictly more accurate than the one implemented in SCADE. It is a satisfactory answer to SCADE users that sometimes complain about the amount of false initialization alarms raised by the available SCADE code generator. All the user programs that were bench-marked with the RELUC compiler were analyzed successfully: no initialization problem was detected except one and it was not a false alarm. These experiments confirm that the program discipline imposed by the present type system is finally less restrictive than the one SCADE users apply usually. We believe that the proposed analysis offers a good compromise between complexity and usability.

6 Conclusion

In this paper we have presented a type-based initialization analysis for a synchronous data-flow language providing uninitialized unary delay and a separated initialization operator. Originally developed in a higher-order setting, the system has been implemented in the prototype SCADE-LUSTRE compiler and in the LUCID SYNCHRONE compiler.

Being a classical type system with sub-typing, it can benefit from conventional implementation techniques and some extra optimizations due to the *one-bit* abstraction. The implementation is light (less than one thousand lines of code).

Although the *one-bit* abstraction may appear too coarse, it gives surprisingly good result on real size programs: the analysis is fast and gives good diagnostics. Most of the time, rejected programs do produce undefined results. This is mainly due to the very nature of synchronous data-flow languages which do not provide control structures and where streams are defined by equations, delays and initialization operators.

Many improvements of the analysis can be considered, the most interesting being to use clock informations synthesized during the clock calculus. This is a matter of future work.

References

1. Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Seventh Conference on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993. ACM.
2. Alexander Aiken, Edward Wimmers, and Jens Palsberg. Optimal Representations of Polymorphic Types with Subtyping. In *Theoretical Aspects of Computer Software (TACS)*, September 1997.
3. E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language*. Academic Press, 1985.
4. Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.
5. Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Submitted to publication, 2001.
6. Jean-Louis Colaço and Marc Pouzet. Type-based Initialization of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.
7. Pascal Cuoq and Marc Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.
8. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
9. G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.
10. John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
11. François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 122–133, May 1996.
12. François Pottier. Simplifying subtyping constraints: a theory. Information & Computation, November 2001.
13. Marc Pouzet. *Lucid Synchrone, version 2. Tutorial and reference manual*. Université Pierre et Marie Curie, LIP6, Mai 2001. Distribution available at: `www-spi.lip6.fr/lucid-synchrone`.
14. SCADE. `http://www.esterel-technologies.com/scade/`.

A Auxiliary definitions

We define the set $FV(t)$ as the disjoint union of free type variables (α) and free initialisation type variables (δ): $FV(t) = FT(t) + FI(t)$. The function $FV.$ is lifted to type schemes and environments.

$$
\begin{array}{ll}
FT(\forall\alpha_1...\alpha_m.\sigma) & = FT(\sigma) - \{\alpha_1, ..., \alpha_m\} \\
FV(\forall\delta_1...\delta_k : C.t) & = FT(t) \\
FT(t_1 \to t_2) & = FT(t_1) \cup FT(t_2) \\
FT(t_1 \times t_2) & = FT(t_1) \cup FT(t_2) \\
FT(\alpha) & = \{\alpha\} \\
FV(d) & = \emptyset \\
FI(\forall\alpha_1...\alpha_m.\sigma) & = FI(\sigma) \\
FI(\forall\delta_1...\delta_k : C.t) & = (FI(t) \cup FI(C)) - \{\delta_1, ..., \delta_k\} \\
FI([\delta'_1 \le \delta_1, ..., \delta'_n \le \delta_n]) & = \{\delta'_1, ..., \delta'_n, \delta_1, ..., \delta_n\} \\
FI(t_1 \to t_2) & = FI(t_1) \cup FI(t_2) \\
FI(t_1 \times t_2) & = FI(t_1) \cup FI(t_2) \\
FI(\alpha) & = \emptyset \\
FI(d) & = FI(d) \\
FI(\delta) & = \{\delta\} \\
FI(\mathbf{0}) & = \emptyset \\
FI(\mathbf{1}) & = \emptyset \\
FV([\sigma_1/x_1, \ldots, \sigma_n/x_n]) & = \cup_{i=1}^{n} FV(\sigma_i)
\end{array}
$$

B Proof of the Initialisation Analysis

We define a valuation $\mathcal{V}$ from variables (δ) to the set $\{\mathbf{0}, \mathbf{1}\}$ and lift it to types in the usual way. We shall interpret the basic type $\mathbf{0}$ with the integer value 0 and the basic type $\mathbf{1}$ with the integer value 1.

We define interpretation functions $\mathcal{I}_.(.)$ relating type schemes and set of values. We overload the notation for evaluation environments and for constraints. An interpretation $\mathcal{I}_.(.)$ is such that:

$$
\begin{array}{ll}
v \in \mathcal{I}_{\mathcal{V}}(d) & \text{iff for all } k \ge \mathcal{V}(d), \\
 & \quad \text{if } v(k) \text{ is defined} \\
 & \quad \text{then } v(k) \neq nil \\
v \in \mathcal{I}_{\mathcal{V}}(t_1 \to t_2) & \text{iff for all } v_1 \text{ such that} \\
 & \quad v_1 \in \mathcal{I}_{\mathcal{V}}(t_1), v(v_1) \in \mathcal{I}_{\mathcal{V}}(t_2) \\
(v_1, v_2) \in \mathcal{I}_{\mathcal{V}}(t_1 \times t_2) & \text{iff } v_1 \in \mathcal{I}_{\mathcal{V}}(t_1) \text{ and } v_2 \in \mathcal{I}_{\mathcal{V}}(t_2) \\
v \in \mathcal{I}_{\mathcal{V}}(\forall\alpha_1, ..., \alpha_k.\sigma) & \text{iff for all } t_1, ..., t_k, \\
 & \quad v \in \mathcal{I}_{\mathcal{V}}(\sigma[t_1/\alpha_1, ..., t_k/\alpha_k]) \\
v \in \mathcal{I}_{\mathcal{V}}(\forall\delta_1, ..., \delta_n : C.t) & \text{iff for all } d_1, ..., d_n, \text{such that} \\
 & \quad \mathcal{I}_{\mathcal{V}}(C[d_1/\delta_1, ..., d_n/\delta_n]), \\
 & \quad v \in \mathcal{I}_{\mathcal{V}}(t[d_1/\delta_1, ..., d_n/\delta_n]) \\
\rho \in \mathcal{I}_{\mathcal{V}}(\emptyset) & \\
\rho[x \leftarrow v] \in \mathcal{I}_{\mathcal{V}}(H[x : \sigma]) & \text{iff } \rho \in \mathcal{I}_{\mathcal{V}}(H) \wedge v \in \mathcal{I}_{\mathcal{V}}(H) \\
\mathcal{I}_{\mathcal{V}}([d_1 \le d'_1, ..., d_n \le d'_n]) & \text{iff } \mathcal{I}_{\mathcal{V}}(d_1) \subseteq \mathcal{I}_{\mathcal{V}}(d'_1) \wedge ... \wedge \\
 & \quad \mathcal{I}_{\mathcal{V}}(d_n) \subseteq \mathcal{I}_{\mathcal{V}}(d'_n)
\end{array}
$$

$\mathcal{V}(d)$ returns an integer equal to 0 or 1 and $\mathcal{I}_{\mathcal{V}}(d)$ defines a set of streams. For example, $\mathcal{I}_{\mathcal{V}}(\mathbf{0})$ defines streams which are always well initialized whereas $\mathcal{I}_{\mathcal{V}}(\mathbf{1})$ may contain streams with a *nil* value at their first instant. Then, we associate a set of functional values to function types $t_1 \to t_2$, Cartesian products of sets to product types. Then, a value belongs to the interpretation of a universally quantified type when it belong to all its instances verifying the sub-typing constraints.

Remark 1 (Causality). For any continuous stream function f from $(D^\omega, \leq)$ to $(D^\omega, \leq)$ (where $\leq$ stands for the prefix order overs streams), for any initialisation type d and valuation $\mathcal{V}$, if $f(\epsilon) \in \mathcal{I}_\mathcal{V}(d)$ then $\mathit{fix}\ f \in \mathcal{I}_\mathcal{V}(d)$. This means that the initialisation information of a stream recursion depends only on one iteration.

Indeed, $\epsilon \in \mathcal{I}_\mathcal{V}(d)$. Then, either $f(\epsilon) = \epsilon$ or $f(\epsilon) = x.s$. In the first case, $\mathit{fix}\ f = lim_{n\to\infty}(f^n(\epsilon)) = \epsilon \in \mathcal{I}_\mathcal{V}(d)$ and it corresponds to a *causality loop*. In the later case, f is strictly increasing and $f(\epsilon) = x.s$ and for all n, $f^n(\epsilon)(0) = (\mathit{fix}\ f)(0) = x$. Thus, $f(\epsilon) \in \mathcal{I}_\mathcal{V}(d)$ implies that $\mathit{fix}\ f \in \mathcal{I}_\mathcal{V}(d)$.

The theorem 1 is based on the two following lemmas. The first one states the correctness of the relation between types.

Lemma 1 (Relation). *The two properties hold.*

1. *If $C \Rightarrow t_1 \leq t_2$ then for all valuation $\mathcal{V}$ such that $\mathcal{I}_\mathcal{V}(C)$, we have $\mathcal{I}_\mathcal{V}(t_1) \subseteq \mathcal{I}_\mathcal{V}(t_2)$.*
2. *if $C' \Rightarrow C$ then for all valuation $\mathcal{V}$, if $\mathcal{I}_\mathcal{V}(C')$ then $\mathcal{I}_\mathcal{V}(C)$.*

Proof: The lemma is proved by recurrence on the proof structure of $C \Rightarrow t_1 \leq t_2$. □

The following one states a more general property than theorem 1. The theorem is obtained by instantiating t with d and considering empty environments and constraints.

Lemma 2 (Correctness). *The two properties hold.*

1. *for all environment H, expression e, type t and constraints C, if $H \mid C \vdash e : t$ then for all valuation $\mathcal{V}$ and evaluation environment ρ, if $\mathcal{I}_\mathcal{V}(C)$ and $\rho \in \mathcal{I}_\mathcal{V}(H)$ then $\mathrm{S}_\rho(e) \in \mathcal{I}_\mathcal{V}(t)$.*
2. *for all environments H, constraint C, definition D, if $H \mid C \vdash D$ holds then for all valuation $\mathcal{V}$ such that $\mathcal{I}_\mathcal{V}(C)$, for all ρ, if $\rho \in \mathcal{I}_\mathcal{V}(H)$ then $\mathrm{S}_\rho(D) \in \mathcal{I}_\mathcal{V}(H)$.*

Proof: The lemma is proved by recurrence on the structure of the proof tree.

Rule (IM) $e = i$. Let $\mathcal{V}$ and ρ such that $\mathcal{I}_\mathcal{V}(C)$ and $\rho(x) \in \mathcal{I}_\mathcal{V}(H(x))$. We have $S_\emptyset(i) = S_\rho(i) = i^\# \in \mathcal{I}_\emptyset(\mathbf{0}) = \mathcal{I}_\mathcal{V}(\mathbf{0})$.

Rule (INST) $e = x$. Let $H(x) = \sigma$. Let $t, C' \in \mathit{inst}(\sigma)$. Let $\mathcal{V}$ such that $\mathcal{I}_\mathcal{V}(C')$. Let ρ such that for all $x \in Dom(H)$, $\rho(x) \in \mathcal{I}_\mathcal{V}(\sigma)$. Either $\sigma = t$ or some variables are universally quantified.

- If $\sigma = t$ then $H, x : t \mid C \vdash x : t$ and the property holds trivially.
- According to the definition of $\mathcal{I}_.(.)$, $\rho(x) \in \mathcal{I}_\mathcal{V}(\forall\alpha_1, ..., \alpha_n.\forall\delta_1, ..., \delta_k : C.t)$ iff for all $t_1, ..., t_n$ and for all $d_1, ..., d_k$ such that $\mathcal{I}_\mathcal{V}(C[d_1/\delta_1, ..., d_k/\delta_k])$, $\rho(x) \in \mathcal{I}_\mathcal{V}(t[t_1/\alpha_1, ..., t_n/\alpha_n][d_1/\delta_1, ..., d_k/\delta_k])$. According to the definition of $\mathit{inst}(.)$, $C' \Rightarrow C[d_1/\delta_1, .../\delta_k]$. Since $\mathcal{I}_\mathcal{V}(C')$, according to lemma 1 (prop. 2), we have $\mathcal{I}_\mathcal{V}(C[d_1/\delta_1, ..., d_k/\delta_k])$. Thus, $\rho(x) \in \mathcal{I}_\mathcal{V}(t[t_1/\alpha_1, ..., t_n/\alpha_n][d_1/\delta_1, ..., d_k/\delta_k])$ which is the expected result.

Rule (OP) Direct recurrence.

Rule (APP) $e = e_1(e_2)$ Let H be the typing environment and ρ, the corresponding evaluation environment.

Suppose the property holds for e_1 and e_2, that is $S_\rho(e_1) \in \mathcal{I}_\mathcal{V}(t_1 \to t_2)$ and $S_\rho(e_2) \in \mathcal{I}_\mathcal{V}(t_1)$.
According to the property of $\mathcal{I}_.(.)$, for any $v_1 \in \mathcal{I}_\mathcal{V}(t_1)$, $(S_\rho(e_1))(v_1) \in \mathcal{I}_\mathcal{V}(t_2)$.
Thus, $(S_\mathcal{V}(e_1))(S_\mathcal{V}(e_2)) = S_\mathcal{V}(e_1(e_2)) \in \mathcal{I}_\mathcal{V}(t_2)$ which is the expected result.

Rule (NODE) $e = \texttt{node}\ f\, x = e_1\ \texttt{in}\ e_2$. There are two cases.

Either $gen_{H|C}(t \to t_1) = t \to t_1$ or some variables can be generalized.

- Case 1. Let $\mathcal{V}$ such that $\mathcal{I}_\mathcal{V}(C)$. Let $v_x \in \mathcal{I}_\mathcal{V}(t)$ and $\rho \in \mathcal{I}_\mathcal{V}(H)$. We have $\rho[x \leftarrow v_x] \in \mathcal{I}_\mathcal{V}(H[x : t])$.
  Applying the recurrence hypothesis to the left premise, $S_{\rho[x\leftarrow v_x]}(e_1) \in \mathcal{I}_\mathcal{V}(t_1)$. This is true for any $v_x \in \mathcal{I}_\mathcal{V}(t)$ thus, according to the definition of $\mathcal{I}_.(.)$, $\lambda y.S_{\rho[x\leftarrow y]}(e_1) \in \mathcal{I}_\mathcal{V}(t \to t_1)$ (with $y \notin Dom(\rho)$).
  Applying the recurrence hypothesis to the second premise, we can state that for all $v_f \in \mathcal{I}_\mathcal{V}(t \to t_1)$, $S_{\rho[f\leftarrow v_f]}(e_2) \in \mathcal{I}_\mathcal{V}(t_2)$. Thus, $S_\rho(\texttt{node}\ f\, x = e_1\ \texttt{in}\ e_2)$ equals $S_{\rho[f\leftarrow \lambda y.S_{\rho[x\leftarrow y]}(e_1)]}(e_2)$ (according to the definition of `node`) which belongs to $\mathcal{I}_\mathcal{V}(t_2)$ and this is the expected result.
- Case 2. Half of the proof stays the same and differs for the first premise. Let $\mathcal{V}$ such that $\mathcal{I}_\mathcal{V}(C)$. Let $v_x \in \mathcal{I}_\mathcal{V}(t)$ and $\rho \in \mathcal{I}_\mathcal{V}(H)$.
  We have $\rho[x \leftarrow v_x] \in \mathcal{I}_\mathcal{V}(H[x : t])$. Applying the recurrence hypothesis to the left premise, we can state that $S_{\rho[x\leftarrow v_x]}(e_1) \in \mathcal{I}_\mathcal{V}(t')$.
  This is true for any $v_x \in \mathcal{I}_\mathcal{V}(t)$ thus, according to the definition of $\mathcal{I}_.(.)$, $\lambda y.S_{\rho[x\leftarrow y]}(e_1) \in \mathcal{I}_\mathcal{V}(t \to t')$ (with $y \notin Dom(\rho)$). Now, let $\alpha_1, ..., \alpha_n \notin FV(H)$.
  Then $\lambda y.S_{\rho[x\leftarrow y]}(e_1) \in \mathcal{I}_\mathcal{V}(\forall\alpha_1, ..., \alpha_n.t \to t')$. Let $\delta_1, ..., \delta_k \notin (FV(t \to t') \cup FV(C))$. $\lambda y.S_{\rho[x\leftarrow y]}(e_1) \in \mathcal{I}_\mathcal{V}(\forall\alpha_1, ..., \alpha_n.\forall\delta_1, ..., \delta_k : C.t \to t')$.
  Now, applying the recurrence hypothesis to the second premise, we can state that for all v_f, if $v_f \in \mathcal{I}_\mathcal{V}(\forall\alpha_1, ..., \alpha_n.\forall\delta_1, ..., \delta_k : C.t \to t')$ then $S_{\rho[f\leftarrow v_f]}(e_2) \in \mathcal{I}_\mathcal{V}(t_2)$. $S_{\rho[f\leftarrow \lambda y.S_{\rho[x\leftarrow y]}(e_1)]}(e_2) = S_\rho(\texttt{node}\ f\, x = e_1\ \texttt{in}\ e_2) \in \mathcal{I}_\mathcal{V}(t_2)$ which is the expected result.

Rule (DEF) By recurrence, using property 2.

Rule (PAIR) Direct recurrence.

Rule (EQ) Let $\mathcal{V}$ such that $\mathcal{I}_{\mathcal{V}}(C)$. For all $v_x \in \mathcal{I}_{\mathcal{V}}(d)$ and $\rho \in \mathcal{I}_{\mathcal{V}}(H)$, that is, $\rho[x \leftarrow v_x] \in \mathcal{I}_{\mathcal{V}}(H[x : d])$, we can state that $S_{\rho[x \leftarrow v_x]}(e) \in \mathcal{I}_{\mathcal{V}}(d)$ (recurrence hypothesis). Now, $\epsilon \in \mathcal{I}_{\mathcal{V}}(d)$ and $S_{\rho[x \leftarrow \epsilon]}(e) \in \mathcal{I}_{\mathcal{V}}(d)$. Using remark 1, then $x^{\infty} = \mathit{fix}\ \lambda y.(S_{\rho[x \leftarrow y]}(e)) \in \mathcal{I}_{\mathcal{V}}(d)$ which is the expected result.

Rule (AND) $D = (x = e)$ `and` D'. Let $\mathcal{V}$ such that $\mathcal{I}_{\mathcal{V}}(C)$. For all ρ, if $\rho \in \mathcal{I}_{\mathcal{V}}(H)$ then $S_{\rho}(x = e) = \rho[x \leftarrow x^{\infty}] \in \mathcal{I}_{\mathcal{V}}(H)$ applying the recurrence hypothesis on the first premise. Moreover, $H(x) = d$ (according to rule (EQ)). Applying the recurrence on the second premise, we can state that for all $v_x \in \mathcal{I}_{\mathcal{V}}(d)$, if $\rho[x \leftarrow v_x] \in \mathcal{I}_{\mathcal{V}}(H[x : d])$ then $S_{\rho[x \leftarrow v_x]}(D') \in \mathcal{I}_{\mathcal{V}}(H[x : d])$. Since $x^{\infty} \in \mathcal{I}_{\mathcal{V}}(d)$, $S_{\rho}(D) = S_{\rho[x \leftarrow x^{\infty}]}(D') \in \mathcal{I}_{\mathcal{V}}(H)$ which is the expected result.

Rule (SUB) Direct application of lemma 1. □