Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 14 pages

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

Jean-Louis Colaço [1]

ESTEREL Technologies
Park Avenue 9, Rue Michel Labrousse 31100 Toulouse, France.

Marc Pouzet [2]

Laboratoire LIP6
Université Paris 6, 8 rue du Capitaine Scott, 75015 Paris, France.

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. It gives very good results in practice.

1 Introduction

Since its definition in the early eighties, Lustre [6] 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 [11], a graphical environment based on Lustre and distributed successively by Verilog SA, Telelogic and now Esterel Technologies.

[1] E-mail: Jean-Louis.Colaco@esterel-technologies.com
[2] Email: Marc.Pouzet@lip6.fr

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 initialization.

The purpose of this paper is to present a new initialization analysis which improves the existing one without using general proof systems. The analysis has been designed for Lucid Synchrone [4,3,10]. While keeping the fundamental properties of Lustre, Lucid Synchrone provides powerfull 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.

The language is used today by the Scade team for defining and prototyping extensions of Lustre in collaboration with VERIMAG and LIP6.

1.1 Contribution

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

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. These initialization informations are types, i.e., abstractions of value with respect to the initialization problem. Modularity is partly responsible for efficiency. Finally, it appears to give *good diagnostic* 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 analysed successfully.

The paper is organized as follows. Section 2 gives the main intuitions of our initialization analysis. 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 correction theorem stating that *"well typed programs are well initialized"*. Section 4 discuss implementation approaches taken by both compilers. Section 5 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 sequences and primitives imported from a host language are applied point-wisely to their argument. These languages provide also a unary delay `pre` (for *previous*) and an initialization operator `->` (for *followed-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. Moreover, it is often useful to define a function that does not necessarily initialize all its outputs and does not need its inputs 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.

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, and 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 $\mathbf{0} \rightarrow \mathbf{1}$ 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, `deriv x` has type $\mathbf{1}$ whereas `deriv` expects a value with type $\mathbf{0}$.

The *one-bit* abstraction comes from the observation that equations are often written by separating the initialization part from the invariant part. Notice that an initialization operator cannot eliminate the *nil* value appearing at the second instant in an expression `pre(pre x)` [3]. 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 3.4). 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, give remarkably good results. Thanks to the type based formulation, the initialization analysis is modular, it addresses LUCID SYNCHRONE as well as LUSTRE programs and can be implemented using standard techniques.

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

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 ($op(e_1, e_2)$) [4], 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.

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

This kernel is essentially an ML kernel managing sequences as primitive values. We give it a classical Kahn semantics [7] that we remind here shortly. Let T^∞ be the set of finite or infinite sequences of elements over the set T ($T^\infty = T^* + T^\omega$). 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 non empty sequence, $s(i)$ defines the i-th element of s. Let $\leq$ be the prefix order over sequences, i.e., $x \leq y$ if x is a prefix of y. The ordered set $(T^\infty, \leq)$ is a cpo. If f is a continuous mapping from sequences to sequences, we shall write $fix\ f$ 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 $T_{i\,nil}^{\infty}$ 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 $S(.)$

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

such that $S_\rho(D)$ defines the denotation of stream equations. We first give an interpretation on sequences to every data-flow primitive.

$$\begin{aligned}
op^{\#}(s_1, s_2) &= \epsilon && \text{if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
op^{\#}(v_1.s_1, v_2.s_2) &= (v_1\, op\, v_2).op^{\#}(s_1, s_2) && \text{if } v_1 \neq nil \text{ and } v_2 \neq nil \\
op^{\#}(v_1.s_1, v_2.s_2) &= nil.op^{\#}(s_1, s_2) && \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathtt{if}^{\#}(s_1, s_2, s_3) &= \epsilon && \text{if } s_1 = \epsilon \text{ or } s_2 = \epsilon \text{ or } s_3 = \epsilon \\
\mathtt{if}^{\#}(nil.s, v_2.s_2, v_3.s_3) &= nil.\mathtt{if}^{\#}(s, s_2, s_3) \\
\mathtt{if}^{\#}(true.s, v_2.s_2, v_3.s_3) &= v_2.\mathtt{if}^{\#}(s, s_2, s_3) \\
\mathtt{if}^{\#}(false.s, v_2.s_2, v_3.s_3) &= v_3.\mathtt{if}^{\#}(s, s_2, s_3)
\end{aligned}$$

$$\begin{aligned}
\mathtt{pre}^{\#}(s) &= nil.s \\
s_1 \;\mathtt{->}^{\#} s_2 &= \epsilon \qquad \text{if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
(v_1.s_1) \;\mathtt{->}^{\#} (v_2.s_2) &= v_1.s_2
\end{aligned}$$

$$\begin{aligned}
S_\rho(op(e_1, e_2)) &= op^{\#}(S_\rho(e_1))(S_\rho(e_2)) \\
S_\rho(\mathtt{pre}\ e) &= \mathtt{pre}^{\#}(S_\rho(e)) \\
S_\rho(e_1 \;\mathtt{->}\; e_2) &= (S_\rho(e_1)) \;\mathtt{->}^{\#}\; (S_\rho(e_2)) \\
S_\rho(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) &= \mathtt{if}^{\#}(S_\rho(e_1), S_\rho(e_2), S_\rho(e_3)) \\
S_\rho(x) &= \rho(x) \\
S_\rho(i) &= i.S_\rho(i)
\end{aligned}$$

We can easily check that the above primitives are continuous. The denotational semantics for other constructions is defined below [5]:

$$\begin{aligned}
S_\rho(\mathtt{node}\ f\ x = e_1\ \mathtt{in}\ e_2) &= S_{\rho[f \leftarrow \lambda y. S_{\rho[x \leftarrow y]}(e_1)]}(e_2) \\
S_\rho(e_1(e_2)) &= S_\rho(e_1)(S_\rho(e_2)) \\
S_\rho(e\ \mathtt{with}\ D) &= S_{S_\rho(D)}(e) \\
S_\rho(e_1, e_2) &= (S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(\mathtt{fst}\ e) &= v_1 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(\mathtt{snd}\ e) &= v_2 \text{ if } S_\rho(e) = (v_1, v_2) \\
S_\rho(x = e) &= \rho[x \leftarrow x^\infty] \text{ where } x^\infty = \mathit{fix}\ \lambda y. S_{\rho[x \leftarrow y]}(e) \\
S_\rho(x_1 = e_1\ \mathtt{and}\ x_2 = e_2) &= S_{S_\rho(x_1 = e_1[x_2\ \mathtt{with}\ e_2/x_2])}(x_2 = e_2)
\end{aligned}$$

Definition 1 (Well initialized) *An expression e is well initialized from k*

[5] For simplicity, we only give the semantics for one or two recursive equations.

in an environment ρ*, noted* $\rho \models e : k$ *if* $\mathrm{S}_\rho(e) \neq \epsilon$ *implies that for all* $n \geq k$*,* $\mathrm{S}_\rho(e)(n) \neq \mathrm{nil}$. *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. Since the paper focuses on the initialization problem only, causality loops are considered well initialized. In practice, these loops are rejected by another analysis [5].

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 ($t \rightarrow t$), a product type ($t \times t$) or an initialization type (d) for a stream value. An initialization type (d) may be $\mathbf{0}$ meaning that the stream is always defined; $\mathbf{1}$ 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.

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

3.2.2 Initial conditions, Instanciation and Generalization

Expressions are typed in an initial environment H_0:

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

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

Types can be instanciated or generalized. A type scheme may be instanciated by applying a substitution to its bound type variables and to its initialization type variables. 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.

$$\begin{array}{ll} \mathit{inst}(t) & = t, \emptyset \\ \mathit{inst}(\forall \alpha_1, ..., \alpha_k.\forall \delta_1, ..., \delta_n : C.t) & = t[t_1/\alpha_1, ..., t_k/\alpha_k][d_1/\delta_1, ..., d_n/\delta_n], \\ & \quad C[d_1/\delta_1, ..., d_n/\delta_n] \\ \mathit{gen}_{H|C}(t) & = \forall \alpha_1, ..., \alpha_k.\forall \delta_1, ..., \delta_n : C.t \\ & \quad \text{where for all } \alpha_i,\ \delta_i \notin FV(H) \end{array}$$

3.2.3 The Type System

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

$$H \vdash e : t \mid C \qquad H \vdash D : H_0 \mid C$$

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 produces an environment H_0 and a set of constraints C.

The definition of these two predicates is given in figure 1. This is a classical type-system with sub-typing constraints [1,8,9]. We adopt a simpler (and less general) presentation adapted to the initialization problem.

- constant receive type $\langle \mathbf{0} \rangle$, saying that they are 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 and on the right of the typing judgment. Note also that equations contained in declarations are necessarily stream equations (rule (EQ)) since they must have an initialization type $\langle d \rangle$.
- scheme types can be instanciated (rule (INST)).
- finally, a type may be weakened (rule (SUB)).

The type system uses an auxiliary predicate $\Rightarrow$ expressing the sub-type relation. Its definition is given in figure 2. The sub-type relation is built from the relation between base types (rules (TRIV)), extends it to function

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

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

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

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

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

$$\text{(AND)}\ \frac{H \vdash D_1 : H_1 \mid C \quad H \vdash D_2 : H_2 \mid C}{H \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2 \mid C}$$

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

Fig. 1. The type system

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

$$\text{(ARROW)}\ \frac{C \Rightarrow t_3 \leq t_1 \quad C \Rightarrow t_2 \leq t_4}{C \Rightarrow t_1 \to t_2 \leq t_3 \to t_4} \qquad \text{(PRODUCT)}\ \frac{C \Rightarrow t_1 \leq t_3 \quad C \Rightarrow t_2 \leq t_4}{C \Rightarrow t_1 \times t_2 \leq t_3 \times t_4}$$

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

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

Fig. 2. Subtype relation

and product types (with a contra-variant rule for function types) and adds a transitivity rule. The relation is lifted to sets of type relations (rules (SET) and (EMPTY)).

Proposition 3.1 (Correction) *For all expressions* e*, if* $\vdash e : \langle d \rangle$ *then* $\models e : d$

The proof follows the usual techniques relating typing and evaluation semantics. It is given in appendix A.

3.3 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:

$$\sigma ::= \forall \delta_1, ..., \delta_n : C.b \to b$$

$$t ::= b \to b \mid b$$

$$b ::= d \mid b \times b$$

$$d ::= \mathbf{0} \mid \mathbf{1} \mid \delta$$

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

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

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

3.4 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

`x`	x_0 x_1 x_2 x_3 x_4 x_5 ...
`y`	y_0 y_1 y_2 y_3 y_4 y_5 ...
`h`	f f t t f t ...
`x when h`	x_2 x_3 x_5 ...
`y when not h`	y_0 y_1 y_4 ...
`current (x when h)`	*nil* *nil* x_3 x_4 x_4 x_6 ...

Fig. 3. Clock operators

semantics conditions which are not checked.

3.5 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. Their behavior is given in figure 3. This 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 analysing the `current` operator.

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 $\forall\delta.\langle\delta\rangle \rightarrow \langle\mathbf{0}\rangle \rightarrow \langle\delta\rangle$.

Extending the analysis in order to take clocks into account is a matter of future work.

4 Implementation Issues

4.1 Implementation in the Lucid Synchrone Compiler

Being a classical typing problem with sub-typing, standard implementation techniques apply. However, being a very simple type system, many ad-hoc optimizations apply that largely improve the efficiency. They are briefly presented in the following.

Since the analysis is performed after typing, infered types are used for generating valid initialization type skeletons. This lead to a strong simplification of the implementation. The efficiency of a type system with sub-typing relies deeply on the simplification of sub-typing constraints [2,9]. We apply classical techniques for eliminating intermediate variables in constraints. Moreover, due to the simple type language, we apply some extra optimizations:

- sub-typing constraints are directly annotated on types. The principle is

Application domain	nb. lines of Lustre	analysis time
transport	~ 17000	≤ 55 sec.
helicopter syst.	~ 45000	≤ 11 sec.
aircraft syst.	~ 3000	≤ 1 sec.
automotive syst.	~ 1000	≤ 0.1 sec.

Fig. 4. Benchmarks on ReLuC

the following: a dependence variable δ points to its list of predecessors and successors. Thus, internally, the type language becomes:

$$d ::= \mathbf{0} \mid \mathbf{1} \mid \delta_{\geq\rho}^{\leq\rho} \quad \rho ::= d, \rho \mid \emptyset$$

- 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.

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

4.2 Implementation in the ReLuC Lustre Compiler

In this implementation, we chose the usual approach for solving type inference in presence of sub-typing. i.e. 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 $\forall \delta_1, \delta_2 . \delta_1 \times \delta_2 \rightarrow \delta_1 \sqcup \delta_2$.

The figure 4 gives some benchmarks of the ReLuC implementation on real applications. The first application contains a lot of nodes having several hundreds inputs and outputs. This is partly responsible for this execution time which remains acceptable. For example, the analysis is five time faster on the second program.

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 initialization.

The ReLuC initialization analyser has been used to validate the Scade library and 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.

5 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 many standard implementation techniques and some extra optimisations 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 useful 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] P. Caspi and M. Pouzet. Lucid Synchrone, a functional extension of Lustre. submitted to publication, 2000.

[4] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[5] P. Cuoq and M. Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

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

[7] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[8] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 122–133, May 1996.

[9] François Pottier. Simplifying subtyping constraints: a theory. To appear in *Information & Computation*, August 2000.

[10] 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`.

[11] SCADE. `http://www.esterel-technologies.com/scade/`

A Proof Sketch of the Initialisation Analysis

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

$$
\begin{array}{ll}
v \in \mathcal{I}(\langle d \rangle) & \text{iff for all } k, k \geq d, v(k) \neq nil \\
v \in \mathcal{I}(t_1 \to t_2) & \text{iff for all } v_1 \text{ such that } v_1 \in \mathcal{I}(t_1), v(v_1) \in \mathcal{I}(t_2) \\
(v_1, v_2) \in \mathcal{I}(t_1 \times t_2) & \text{iff } v_1 \in \mathcal{I}(t_1) \text{ and } v_2 \in \mathcal{I}(t_2) \\
v \in \mathcal{I}(\forall \alpha_1, ..., \alpha_k.\sigma) & \text{iff for all } t_1, ..., t_k, v \in \mathcal{I}(\sigma[t_1/\alpha_1, ..., t_k/\alpha_k]) \\
v \in \mathcal{I}(\forall \delta_1, ..., \delta_n : C.t) & \text{iff for all } d_1, ..., d_n, \\
& \quad \text{such that } \mathcal{I}(C[d_1/\delta_1, ..., d_n/\delta_n]), \\
& \quad v \in \mathcal{I}(t[d_1/\delta_1, ..., d_n/\delta_n])
\end{array}
$$

$$\mathcal{I}([d_1 \leq d_1', ..., \mathrm{d}_n \leq \mathrm{d}_n']) \text{ iff } \mathcal{I}(d_1) \subseteq \mathcal{I}(d_1') \wedge ... \wedge \mathcal{I}(d_n) \subseteq \mathcal{I}(d_n')$$

The following lemma is proved by recurrence on the type structure.

Proposition A.1 (Relation) *For all t_1 and t_2, if $t_1 \leq t_2$ then $\mathcal{I}(t_1) \subseteq \mathcal{I}(t_2)$.*

We then prove the following property about typing judgments, by recurrence on the structure of the proof tree.

Proposition A.2 (Correction) *If $[x_1 : \sigma_1, ..., x_n : \sigma_n] \vdash e : t \mid C$ then for all $v_1, ..., v_n$ such that $v_1 \in \mathcal{I}(\sigma_1), ..., v_n \in \mathcal{I}(\sigma_n)$ and $\mathcal{I}(C)$, $\mathrm{S}_{[v_1/x_1, ..., v_n/x_n]}(e) \in \mathcal{I}(t)$.*