Normalising Lustre Preserves Security

Sanjiva Prasad and R. Madhukar Yerraguntla

Indian Institute of Technology Delhi, INDIA
{sanjiva,madhukar.yr}@cse.iitd.ac.in

Abstract. The synchronous reactive data flow language Lustre is an expressive language, equipped with a suite of tools for modelling, simulating and model-checking a wide variety of safety-critical systems. A critical intermediate step in the formally certified compilation of Lustre involves translation to a well-behaved sub-language called "Normalised Lustre" (NLustre). Recently, we proposed a simple Denning-style lattice-based secure information flow type system for NLustre, and proved its soundness by establishing that security-typed programs are non-interfering with respect to the co-inductive stream semantics.
In this paper, we propose a similar security type system for unrestricted Lustre, and show that Bourke *et al.*'s semantics-preserving normalisation transformations from Lustre to NLustre are security-preserving as well. A novelty is the use of refinement security types for node calls. The main result is the preservation of security types by the normalisation transformations. The soundness of our security typing rules is shown by establishing that well-security-typed programs are non-interfering, via a reduction to type-preservation (here), semantics-preservation (Bourke *et al.*) and our previous result of non-interference for NLustre.

Keywords: Synchronous reactive data flow, Lustre, Compiler transformation, Security type system, Non-interference, Security preservation.

1 Introduction

The synchronous reactive data flow language Lustre [6,10] is an expressive language with an elegant formal semantics. Its underlying deterministic, clocked model makes it a versatile programming paradigm, with diverse applications such as distributed embedded controllers, numerical computations, and complex Scade 6 [7] safety-critical systems. It is also equipped with a suite of tools, comprising: (a) a certified compilation framework from the high-level model into lower-level imperative languages [2,3]; (b) model-checkers [16,12] (c) simulation tools [11] for program development.

The development of a formally certified compiler from Lustre to an imperative language is the subject of active research [2,3]. A critical intermediate step involves the translation from Lustre to a well-behaved sub-language called "Normalised Lustre" (NLustre), presented in [3]. A recent paper (in French) defines the normalisation transformations from Lustre to NLustre, and establishes formally that they are semantics-preserving with respect to the stream semantics [4] (see Theorems 2 and 3).

Recently we proposed a Denning-style lattice-based secure information flow (SIF) type system for NLustre, and proved its soundness by establishing that securely-typed programs are *non-interfering* with respect to the co-inductive stream semantics [15]. The main ideas underlying the security type system are (i) that each stream is assigned (w.r.t. assumptions on variables) a *symbolic* security type, (ii) equations induce constraints on security types of the defined variables and of the defining expressions, and (iii) the output streams from a node have security levels at least as high as those of the input streams on which they depend. The symbolic constraint-based formulations allows us to *infer* constraints that suffice to ensure security. The rules are simple, intuitive and amenable to being incorporated into the mechanised certified compilation [14] already developed for Lustre [5]. In this paper, we propose a similar secure-information-flow type system for unrestricted Lustre (§3). The main innovation is formulating symbolic constraint-based *refinement* (sub)types. These are necessitated by the presence in Lustre of nested node calls (in NLustre, directed nesting is disallowed).

The security type system is shown by reduction to be *sound* with respect to Lustre's *co-inductive* stream semantics. While it is possible to do so directly by establishing that well-security-typed programs exhibit non-interference [9] using exactly the approach in [15], here we do so via a sound *compiler transformation*: We show that the semantics-preserving normalisation transformations (de-nesting and distribution, and explicit initialisation of `fby`) from Lustre to NLustre proposed in [4] *preserve security types* as well (Theorem 1 in §4). In particular, there is a strong correspondence at the level of *node definitions*. The preservation of *security signatures* of node definitions is established via Lemma 3 in §3. The main idea is to remove local variable types via a substitution procedure simplify (Figure 9), showing that this maintains satisfiability of type constraints. Since these transformations preserve operational behaviour *as well as security types of nodes*, and since we have already established non-interference for the target language NLustre [15, Theorem 5], non-interference holds for source Lustre as well (Theorem 5 in §5).

Although the paper is intimately dependent on results of earlier work, we have endeavoured to keep it self-contained. The reader interested in the complete stream semantics of Lustre may refer to the appendices.

Related Work. We mention only the immediately relevant work here; a fuller discussion on related work can be found in [15]. The formalisation of Lustre semantics and its certified compilation are discussed in detail in [1,2,3]. The normalisation transformations examined here are proposed in [4]. Our lattice-based SIF framework harks back to Denning's seminal work [8]. The idea of type systems for SIF can be found in *e.g.*, [18]. That work also expressed soundness of a SIF type system in terms of the notion of non-interference [9]. Our previous work [15] adapted that framework to a declarative data flow setting, showing that it is possible to *infer* minimal *partial-ordering constraints* between *symbolic types*. The idea of type-preservation under the rewriting of programs is commonplace in logic and proof systems ("subject reduction").

2 LUSTRE and NLUSTRE

$$
\begin{array}{lr}
e := & \textbf{(expr)} \\
\quad | \ c & \text{(cnst)} \\
\quad | \ x & \text{(var)} \\
\quad | \ \diamond \ e & \text{(unop)} \\
\quad | \ e \oplus e & \text{(binop)} \\
\quad | \ \vec{e} \ \texttt{when} \ x = k & \text{(whn)} \\
\quad | \ \texttt{merge} \ x \ \vec{e} \ \vec{e} & \text{(mrg)} \\
\quad | \ \texttt{if} \ e \ \texttt{then} \ \vec{e} \ \texttt{else} \ \vec{e} & \text{(ite)} \\
\quad | \ \vec{e} \ \texttt{fby} \ \vec{e} & \text{(fby)} \\
\quad | \ f(\vec{e}) & \text{(ncall)} \\
eq := & \textbf{(equation)} \\
\quad | \vec{x} = \vec{e} & \text{(eq)}
\end{array}
$$

Fig. 1. LUSTRE syntax

$$
\begin{array}{lr}
e := & \textbf{(expr')} \\
\quad | \ c & \text{(cnst)} \\
\quad | \ x & \text{(var)} \\
\quad | \ \diamond \ e & \text{(unop)} \\
\quad | \ e \oplus e & \text{(binop)} \\
\quad | \ e \ \texttt{when} \ x = k & \text{(whn')} \\
ce := & \textbf{(cntrl expr)} \\
\quad | \ e & \text{(expr')} \\
\quad | \ \texttt{merge} \ x \ ce \ ce & \text{(mrg')} \\
\quad | \ \texttt{if} \ e \ \texttt{then} \ ce \ \texttt{else} \ ce & \text{(ite')} \\
eq := & \textbf{(equation)} \\
\quad | \ x =_{ck} ce & \text{(eq')} \\
\quad | \ x =_{ck} c \ \texttt{fby} \ e & \text{(fby')} \\
\quad | \ \vec{x} =_{ck} f(\vec{e}) & \text{(ncall')}
\end{array}
$$

Fig. 2. NLUSTRE syntax

$$
\begin{array}{lr}
ck := & \textbf{(clock)} \\
\quad | \ \texttt{base} & \text{(base)} \\
\quad | \ ck \ \texttt{on} \ (x = k) & \text{(on)}
\end{array}
\qquad
\begin{array}{lr}
d := & \textbf{(node declr)} \\
\quad | \ \texttt{node} \ f(\overrightarrow{x^{ck}}) \ \texttt{returns} \ \overrightarrow{y^{ck}} & \\
\quad\quad \texttt{var} \ \vec{z} \ \texttt{let} \ \overrightarrow{eqn} \ \texttt{tel} & \\
G := \vec{d} & \textbf{(program)}
\end{array}
$$

Fig. 3. Common syntax of nodes and clocks

A LUSTRE program describes a synchronous network with *clocked* streams of data flowing between operators and *nodes*. A program consists of a set of *node definitions*, each parameterised by clocked input and output flows. A clock is a boolean stream – either a `base` clock or one *derived* from another clock when a variable takes a specific (boolean) value (`on` $x = k$, where $k \in \{\texttt{T}, \texttt{F}\}$).

Each node comprises a set of (possibly mutually recursive) *equations*, which define local variables and output flows in terms of flow *expressions*. Such definitions are unique, and may appear in any order. LUSTRE satisfies the *definition* and *substitution* principles, namely that the context does not determine the meaning of an expression and that referential transparency holds. Nodes do not have free variables. Nodes cannot make recursive calls; therefore, the dependency

order on nodes forms a DAG. All expressions and equations can be *annotated* with a clock, following a static analysis to determine clock dependencies.

Figures 1–3 present the syntax of LUSTRE and NLUSTRE.

LUSTRE expressions (Figure 1) include flows described by constants, variables, unary and binary operations on flows, as well as the flows obtained by sampling when a variable takes a particular boolean value (`when`), interpolation based on a boolean variable flow (`merge`), and conditional combinations of flows (`if_then_else`). Of particular interest are flows involving guarded delays (`fby`) and those involving *node calls*.

NLUSTRE is a sub-language into which LUSTRE can be translated, from which subsequent compilation is easier. The main differences between between LUSTRE and NLUSTRE are (i) the former supports *lists* of flows (written $\vec{e}$) for conciseness, whereas in the latter all flows are single streams; (ii) NLUSTRE requires that conditional and `merge` "control" expressions are not nested below unary and binary operators or sampling; (iii) node call and delayed flows (`fby`) are treated as first-class expressions, whereas in NLUSTRE, they can appear only in the context of equations; (iv) LUSTRE permits nested node calls, whereas there is no nesting in NLUSTRE; (v) finally, the first argument of `fby` expressions in NLUSTRE must be a *constant*, to enable a well-defined initialisation that can be easily implemented.

The *translation* from LUSTRE to NLUSTRE [4] involves *distributing* constructs over the individual components of lists of expressions, and *de-nesting* expressions by introducing fresh local variables (See §4). The reader can see an example, adapted from [4], of a LUSTRE program and its translation into NLUSTRE in Figure 12 (ignoring for the moment the security type annotations therein).

2.1 Stream Semantics

The semantics of LUSTRE and NLUSTRE programs are *synchronous*: Each variable and expression defines a data stream which pulses with respect to a *clock*. A clock is a stream of booleans (CompCert/Coq's true and false in Vélus). A flow takes its n^{th} value on the n^{th} clock tick, *i.e.*, some value, written $\langle v \rangle$, is present at instants when the clock value is true, and none (written $\langle\rangle$) when it is false. The *temporal operators* `when`, `merge` and `fby` are used to express the complex clock-changing and clock-dependent behaviours of sampling, interpolation and delay respectively.

Formally the stream semantics is defined using predicates over the program graph G, a (co-inductive) stream *history* ($H_* : Ident \to value\ Stream$) that associates value streams to variables, and a clock bs [3,15,4]. Semantic operations on (lists of) streams are written in blue sans serif typeface. Streams are written in red, with lists of streams usually written in **bold face**. All these stream operators, defined co-inductively, enforce the clocking regime, ensuring the presence of a value when the clock is true, and absence when false.

The predicate $G, H_*, bs \vdash e \Downarrow_{\mathbf{e}} \mathbf{es}$ relates an *expression* e to a *list* of streams, written **es**. A list consisting of only a single stream es is explicitly denoted as $[es]$. The semantics of *equations* are expressed using the predicate $G, H_*, bs \vdash \overrightarrow{eq_i}$,

which requires *consistency* between the assumed and defined stream histories in H_* for the program variables, as induced by the equations. Finally, the semantics of *nodes* is given as a stream history transformer predicate $G \vdash \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}$.

We discuss here only some constructs which relate to the normalisation transformations. Appendices B and C present a complete account of the stream semantics for LUSTRE and NLUSTRE, consistent with [5].

$$\frac{H_*(x) = xs}{G, H_*, bs \vdash x \Downarrow_{\mathsf{e}} [xs]} \text{ (LSvar)}$$

Rule (LSvar) associates a variable x to the stream given by $H_*(x)$.

$$\frac{\forall i: \;\; G, H_*, bs \vdash e0_i \Downarrow_{\mathsf{e}} \mathbf{e0s}_i \quad \forall j: \; G, H_*, bs \vdash e_j \Downarrow_{\mathsf{e}} \mathbf{es}_j \qquad \widehat{\mathsf{fby}_L}\ (\flat(\overrightarrow{\mathbf{e0s}_i}))\ (\flat(\overrightarrow{\mathbf{es}_j})) = \mathbf{os}}{G, H_*, bs \vdash \overrightarrow{e0_i}\ \mathtt{fby}\ \overrightarrow{e_j} \Downarrow_{\mathsf{e}} \mathbf{os}} \text{ (LSfby)}$$

A delay operation is implemented by $e0$ `fby` e. The rule (LSfby) is to be read as follows. Let each expression $e0_i$ denote a list of streams $\mathbf{e0s_i}$, and each expression e_j denote a list of streams $\mathbf{es_j}$. The predicate $\widehat{\mathsf{fby}_L}$ *maps* the predicate fby_L to act on the corresponding components of *lists* of streams, *i.e.*,

$$\widehat{\mathsf{fby}_L}\ \mathbf{xs}\ \mathbf{ys} = \mathbf{zs} \text{ abbreviates } \bigwedge_{i \in [1,m]} \mathsf{fby}_L\ xs_i\ ys_i = zs_i$$

(Similarly for the predicates $\widehat{\mathsf{when}}$, $\widehat{\mathsf{merge}}$, and $\widehat{\mathsf{ite}}$.) The operation $\flat(\_)$ flattens a list of lists (of possibly different lengths) into a single list. Flattening is required since expression e_i may in general denote a *list* of streams $\mathbf{es}_i$. The output list of streams $\mathbf{os}$ consists of streams whose first elements are taken from each stream in $\flat(\overrightarrow{\mathbf{e0s}_i})$ with the rest taken from the corresponding component of $\flat(\overrightarrow{\mathbf{es}_j})$.

$$\frac{\forall i \in [1,..,k]\ G, H_*, bs \vdash e_i \Downarrow_{\mathsf{e}} \mathbf{es}_i \quad [H_*(x_1), \ldots, H_*(x_n)] = \flat(\overrightarrow{\mathbf{es}_i})}{G, H_*, bs \vdash \overrightarrow{x_j} = \overrightarrow{e_i}} \text{ (LSeq)}$$

The rule (LSeq) for equations checks the consistency between the assumed meanings for the defined variables x_j according to the history H_* with the corresponding components of the tuple of streams $\flat(\overrightarrow{\mathbf{es}_i})$ to which a tuple of right-hand side expressions evaluates.

$$\frac{\left\{\begin{array}{rl} \mathsf{name} = & \mathtt{f};\ \mathsf{in} = \overrightarrow{x};\ \mathsf{var} = \overrightarrow{z};\\ \mathsf{out} = & \overrightarrow{y};\ \mathsf{eqs} = \overrightarrow{eqn} \end{array}\right\} \in G \qquad H_*(g.\mathbf{in}) = \mathbf{xs}}{H_*(g.\mathbf{out}) = \mathbf{ys} \quad \mathsf{base\text{-}of}\ \mathbf{xs} = bs \quad \forall eq \in \overrightarrow{eq}: \; G, H_*, bs \vdash eq} \text{ (LSndef)}$$

The rule (LSndef) presents the meaning given to the definition named f of a node $g \in G$ as a stream list transformer. If history H_* assigns lists of streams to

the input and output variables for a node in a manner such that the semantics of the equations in the node are satisfied, then the semantic function $\widehat{f}$ transforms input stream list **xs** to output stream list **ys**. The operation base-of finds an appropriate base clock with respect to which a given list of value streams pulse.

$$\frac{G, H_*, bs \vdash \vec{e_i} \Downarrow_{\mathsf{e}} \mathbf{xs} \quad G \models \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}}{G, H_*, bs \vdash f(\vec{e_i}) \Downarrow_{\mathsf{e}} \mathbf{ys}} \text{ (LSncall)}$$

The rule (LSncall) applies the stream transformer semantic function $\widehat{f}$ to the stream list **xs** corresponding to the tuple of arguments $\vec{e_i}$, and returns the stream list **ys**.

Stream semantics for NLustre. The semantic relations for NLustre are either identical to (as in constants, variables, unary and binary operations) or else the singleton cases of the rules for Lustre (as in `merge`, `ite`, `when`). The main differences are in the occurrence of `fby` (now in a restricted form) and node call only in the context of equations, (which are clock-annotated).

$$\frac{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} [vs] \quad \mathsf{fby}_{NL}\ c\ vs = H_*(x)}{G, H_*, bs \vdash x =_{ck} c\ \texttt{fby}\ e} \text{ (NSfby')}$$

$$\frac{\left\{\begin{array}{l}\mathsf{name} = \texttt{f};\ \mathsf{in} = \vec{x};\ \mathsf{var} = \vec{z};\\ \mathsf{out} = \vec{y};\ \mathsf{eqs} = \vec{eqn}\end{array}\right\} \in G \quad H_*(n.\mathbf{in}) = \mathbf{xs} \quad \text{base-of}\ \mathbf{xs} = bs \quad \text{respects-clock}\ H_*\ bs \quad H_*(n.\mathbf{out}) = \mathbf{ys} \quad \forall eq \in \vec{eq} :\ G, H_*, bs \vdash eq}{G \models \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}} \text{ (NSndef')}$$

$$\frac{H_*, bs \vdash \vec{e} \Downarrow_{\mathsf{e}} \mathbf{vs} \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \text{base-of}\ \mathbf{vs} \quad G \models \widehat{f}(\mathbf{vs}) \searrow \overrightarrow{H_*(x_i)}}{G, H_*, bs \vdash \vec{x} =_{ck} f(\vec{e})} \text{ (NSncall')}$$

Fig. 4. Stream semantics of NLustre nodes and equations

The (NSfby') rule for `fby` in an equational context uses the semantic operation fby_{NL}, which differs from fby_L in that it requires its first argument to be a constant rather than a stream. The (NSndef') rule only differs from (LSndef) in that after clock alignment during *transcription*, we have an additional requirement of H_* being in accordance with the base clock bs, enforced by respects-clock. Finally, the rule rule (NSncall') for node call, now in an equational context, is similar to (LSncall) except that it constrains the clock modulating the equation to be the base clock of the input flows.

3 A Security Type System for LUSTRE

We define a symbolic secure information flow type system, where under security-level type assumptions for program variables, LUSTRE expressions are given a *security type*, and LUSTRE equations induce a set of *ordering constraints* over security types.

Security type expressions (α, β) for LUSTRE are either (i) *type variables* (written δ) drawn from a set STV, or (ii) constructed using a *join* operator (written $\alpha \sqcup \beta$). (iii) The identity element of the associative, commutative and idempotent monoid operation $\sqcup$ is $\bot$. While the above suffice for NLUSTRE, for LUSTREwe introduce (iv) *refinement types* $\alpha\{\!|\rho|\!\}$, where symbolic constraint ρ modulates type expression α. Constraints on security types, typically ρ, are (conjunctions of) relations of the form $\alpha \sqsubseteq \beta$. Since security types are to be interpreted with respect to a complete lattice, we have $\alpha \sqsubseteq \beta$ exactly when $\alpha \sqcup \beta = \beta$. Our proposed security types and their equational theory are presented in Figure 5. The security types for NLUSTRE and their equational theory [15] are highlighted in grey within the diagram. This congruence on NLUSTRE types (henceforth $\equiv_{NL}$), which is given in the highlighted second line of Figure 5, is significantly simpler since it does not involve refinement types!

Types: $\alpha, \beta, \gamma, \theta ::= \bot \mid \delta \in STV \mid \alpha \sqcup \beta \mid \alpha\{\!|\rho|\!\}$ Constraints: $\rho ::= (\theta \sqsubseteq \alpha)^*$

$(\alpha \sqcup \beta) \sqcup \theta = \alpha \sqcup (\beta \sqcup \theta)$, $\alpha \sqcup \alpha = \alpha$, $\alpha \sqcup \beta = \beta \sqcup \alpha$, $\alpha \sqcup \bot = \alpha = \bot \sqcup \alpha$,

$\alpha\{\!|\}\!| = \alpha$, $\alpha_1\{\!|\rho_1|\!\} \sqcup \alpha_2\{\!|\rho_2|\!\} = (\alpha_1 \sqcup \alpha_2)\{\!|\rho_1 \cup \rho_2|\!\}$, $\alpha\{\!|\rho_1|\!\}\{\!|\rho_2|\!\} = \alpha\{\!|\rho_1 \cup \rho_2|\!\}$,

$\overrightarrow{\alpha_i}\{\!|\rho|\!\} = \overrightarrow{\alpha_i\{\!|\rho|\!\}}$, $\{\alpha\{\!|\rho_1|\!\} \sqsubseteq \beta\{\!|\rho_2|\!\}\} = \{\alpha \sqsubseteq \beta\} \cup \rho_1 \cup \rho_2$, $\overrightarrow{\alpha_j}[\theta_i/\delta_i] = \overrightarrow{\alpha_j[\theta_i/\delta_i]}$,

$\alpha\{\!|\rho|\!\}[\theta_i/\delta_i] = \alpha[\theta_i/\delta_i]\{\!|\rho[\theta_i/\delta_i]|\!\}$, $(\alpha \sqsubseteq \beta)[\theta_i/\delta_i] = \alpha[\theta_i/\delta_i] \sqsubseteq \beta[\theta_i/\delta_i]$.

Fig. 5. Security types, constraints and their properties

We write $\alpha[\theta_i/\delta_i]$ for $i = 1, \ldots, k$ to denote the (simultaneous) substitution of security types θ_i for security type variables δ_i in security type α. The notation extends to substitutions on tuples ($\vec{\alpha}$) and constraints ($\alpha \sqsubseteq \beta$).

Constraints are interpreted in a security class lattice SC by the homomorphic extension of a ground instantiation $s : STV \to SC$, such that $s(\bot) = \bot$, $s(\alpha \sqcup \beta) = s(\alpha) \sqcup s(\beta)$, $s(\vec{\alpha}) = \overrightarrow{s(\alpha_i)}$, $s(\alpha \sqsubseteq \beta) = s(\alpha) \sqsubseteq s(\beta)$, and $s(\alpha\{\!|\rho|\!\}) = s(\alpha)$ if $s(\rho)$ holds in SC, *i.e.*, if "s satisfies ρ" (else undefined). To make sense, refinement type $\alpha\{\!|\rho|\!\}$ requires the satisfaction of constraint ρ.

Lemma 1 (Soundness). *The equational theory induced by the equalities in Figure 5 is sound with respect to any ground instantiation s, i.e., (i) $\alpha = \beta$ implies $s(\alpha) = s(\beta)$, and (ii) $\rho_1 = \rho_2$ implies $s(\rho_1)$ is satisfied iff $s(\rho_2)$ is.*

Lemma 2 (Confluence). *All equations other than those of associativity and commutativity (AC) can be oriented left-to-right into rewriting rules. The rewriting system is confluent modulo AC. Equal types (respectively, equal constraints) can be rewritten to a common form modulo AC.*

PROOF SKETCH. The equational theory $\equiv_{NL}$ is decidable, since it is a convergent rewriting system modulo AC. The rules in lines 3 and 4 can all be oriented left to right. We use completion [13] to introduce rule $\alpha_1\{\!|\rho_1|\!\} \sqcup \alpha_2 \longrightarrow (\alpha_1 \sqcup \alpha_2)\{\!|\rho_1|\!\}$, when α_2 is not a refinement type. We use the theory of strongly coherent rewriting modulo AC [17], to efficiently decide type equality. □.

3.1 Security Typing Rules

Assume typing environment $\Gamma : \mathit{Ident} \to \mathit{ST}$, a partial function associating a security type to each free variable x in a LUSTRE program phrase. Expressions and clocks are type-checked with the predicates: $\Gamma \overset{e}{\vdash} e : \vec{\alpha}$ and $\Gamma \overset{ck}{\vdash} ck : \alpha$ respectively. These are read as "under the context Γ mapping variables to security types, e, ck have security type(s) α". The types for tupled expressions are tuples of the types of the component expressions. For equations, we use the predicate: $\Gamma \overset{eqn}{\vdash} eq$:> ρ, which states that under the context Γ, equation eq when type-elaborated generates constraints ρ. Elementary constraints for equations are of the form $\alpha \sqsubseteq \beta$, where β is the security type of the defined variable, and α the security type obtained from that of the defining expression joined with the clock's security type. Since every flow in LUSTRE is defined *exactly once*, by the Definition Principle, no further security constraints apply.

The security typing rules for LUSTRE are presented in Figures 6 – 8, plus the rules for node definition and node call. These rules generalise those in [15] to handle expressions representing lists of flows, and nested node calls. The rules for NLUSTRE expressions other than node call and `fby` are just the singleton cases. Node call and `fby` are handled by the rule for equations.

$$\frac{\Gamma(\texttt{base}) = \gamma}{\Gamma \overset{ck}{\vdash} \texttt{base} : \gamma}\ (\text{LTbase}) \qquad \frac{\Gamma(x) = \gamma_1 \quad \Gamma \overset{ck}{\vdash} ck : \gamma_2}{\Gamma \overset{ck}{\vdash} ck\ \texttt{on}\ x = k : \gamma_1 \sqcup \gamma_2}\ (\text{LTon})$$

Fig. 6. LUSTRE security typing rules for clocks

In (LTbase), we assume Γ maps the base clock `base` to some security variable (γ by convention). In (LTon), the security type of the derived clock is the join of the security types of the clock ck and that of the variable x.

Constants have security type $\bot$, irrespective of the context (rule (LTcnst)). For variables, in rule (LTvar), we look up their security type in the context Γ. Unary operations preserve the type of their arguments (rule (LTunop)). Binary

$$\frac{\Gamma(x) = \alpha}{\Gamma \overset{e}{\vdash} x : \alpha}\ (\text{LTvar}) \qquad \frac{\Gamma \overset{e}{\vdash} e : \alpha}{\Gamma \overset{e}{\vdash} \diamond\, e : \alpha}\ (\text{LTunop}) \qquad \frac{\Gamma \overset{e}{\vdash} e_1 : \alpha_1 \quad \Gamma \overset{e}{\vdash} e_2 : \alpha_2}{\Gamma \overset{e}{\vdash} e_1 \oplus e_2 : \alpha_1 \sqcup \alpha_2}\ (\text{LTbinop})$$

$$\frac{\theta = \Gamma(x) \quad \Gamma \overset{e}{\vdash} \vec{e_t} : \vec{\alpha} \quad \Gamma \overset{e}{\vdash} \vec{e_f} : \vec{\beta}}{\Gamma \overset{e}{\vdash} \texttt{merge}\ x\ \vec{e_t}\ \vec{e_f} : \overrightarrow{(\theta \sqcup \alpha_i \sqcup \beta_i)_i}}\ (\text{LTmrg}) \qquad \frac{}{\Gamma \overset{e}{\vdash} c : \bot}\ (\text{LTcnst})$$

$$\frac{\Gamma \overset{e}{\vdash} e : \theta \quad \Gamma \overset{e}{\vdash} \vec{e_t} : \vec{\alpha} \quad \Gamma \overset{e}{\vdash} \vec{e_f} : \vec{\beta}}{\Gamma \overset{ce}{\vdash} \texttt{if}\ e\ \texttt{then}\ \vec{e_t}\ \texttt{else}\ \vec{e_f} : \overrightarrow{(\theta \sqcup \alpha_i \sqcup \beta_i)_i}}\ (\text{LTite})$$

$$\frac{\Gamma \overset{e}{\vdash} \vec{e_0} : \vec{\alpha} \quad \Gamma \overset{e}{\vdash} \vec{e} : \vec{\beta}}{\Gamma \overset{e}{\vdash} \vec{e_0}\ \texttt{fby}_l\ \vec{e} : \overrightarrow{(\alpha_i \sqcup \beta_i)_i}}\ (\text{LTfby}) \qquad \frac{\Gamma \overset{e}{\vdash} e_1 : \alpha_1 \ldots \Gamma \overset{e}{\vdash} e_n : \alpha_n \quad \Gamma(x) = \gamma}{\Gamma \overset{e}{\vdash} \vec{e}\ \texttt{when}\ x = k : \overrightarrow{(\alpha_i \sqcup \gamma)_i}}\ (\text{LTwh})$$

Fig. 7. Lustre Security Typing Rules for Expressions

$$\frac{\vec{\beta} = \Gamma(\vec{x}) \quad \Gamma \overset{e}{\vdash} \vec{e} : \vec{\alpha} \quad \Gamma \overset{ck}{\vdash} ck : \gamma}{\Gamma \overset{eqn}{\vdash} \vec{x}^{ck} = \vec{e}\ \texttt{:>}\ \{(\gamma \sqcup \alpha_i \sqsubseteq \beta_i)_i\}}\ (\text{LTeq}) \qquad \frac{\Gamma \overset{eqn}{\vdash} eq\ \texttt{:>}\ \rho \quad \Gamma \overset{eqn}{\vdash} eqs\ \texttt{:>}\ \rho'}{\Gamma \overset{eqn}{\vdash} eq; eqs\ \texttt{:>}\ \rho \cup \rho'}\ (\text{LTeqs})$$

Fig. 8. Lustre security typing rules for equations

($\oplus$,`when` and `fby`) and ternary (`if-then-else` and `merge`) operations on flows generate a flow with a security type that is the join of the types of the operand flows (rules (LTbinop), (LTwhn), (LTmrg), (LTite), and (LTfby). In operations on *lists of flows*, the security types are computed component-wise. There is an implicit dependency on the security level of the common clock of the operand flows for these operators. This dependence on the security level of the clock is made explicit in the rule for equations. In general, the security type for any constructed expression is the join of those of its components (and of the clock).

Node call. Node calls assume that we have a security signature for the node definition (described below). We can then securely type node calls by instantiating the security signature with the types of the actual arguments (and that of the base clock). Note the rule (LTncall) creates refinement types consisting of the output types β_i modulated with ρ', the instantiated set of constraints ρ taken from the node signature:

$$\frac{\overset{Node}{\vdash}\ \texttt{Node f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta} \quad \Gamma \overset{e}{\vdash} \vec{e} : \vec{\alpha'} \quad \Gamma(\texttt{base}) = \gamma' \quad \rho' = \rho[\gamma'/\gamma][\vec{\alpha'}/\vec{\alpha}]}{\Gamma \overset{e}{\vdash} f(\vec{e}) : \vec{\beta}\{\!|\rho'|\!\}}\ (\text{LTncall})$$

Node definition. A node definition is given a signature $\overset{Node}{\vdash}\ \texttt{Node f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}$, which is to be read as saying that the node named f relates the security types $\vec{\alpha}$

of the input variables (and γ, that of the base clock) to the types of the output variables $\vec{\beta}$, via the constraints ρ.

Let $\alpha_1, \ldots, \alpha_n, \delta_1, \ldots \delta_k, \beta_1, \ldots \beta_m, \gamma$ be distinct *fresh type variables*. Assume these to be the types of the input, local and output variables, and that of the base clock. We compute the constraints over these variables induced by the nodes equations. Finally, we eliminate, via substitution, the type variables given to the local program variables, since these should not appear in the node's interface. The security signature of a node definition is thus given as:

$$\frac{\begin{array}{c} G(f) = n : \{\texttt{in} = \vec{x}, \texttt{out} = \vec{y}, \texttt{var} = \vec{z}, \texttt{eqn} = \vec{eq}\} \\ \Gamma_F := \{\vec{x} \mapsto \vec{\alpha}, \vec{y} \mapsto \vec{\beta}, \texttt{base} \mapsto \gamma\} \quad \Gamma_L := \{\vec{z} \mapsto \vec{\delta}\} \\ \Gamma_F \cup \Gamma_L \overset{eqn}{\vdash} \vec{eq} \ :> \rho' \ \ (\_, \rho) = \mathsf{simplify}\ (\_, \rho')\ \vec{\delta} \end{array}}{\overset{Node}{\vdash}\ \texttt{Node f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}} \text{(LTndef)}$$

The node signature (and call) rules can be formulated in this step-wise and modular manner since Lustre does not allow recursive node calls and cyclic dependencies. Further, all variables in a node definition are explicitly accounted for as input and output parameters or local variables, so no extra contextual information is required.

$$\frac{}{(\vec{\alpha}, \rho) = \mathsf{simplify}\ (\vec{\alpha}, \rho)\ [\,]} \qquad \frac{(\vec{\alpha'}, \rho') = \mathsf{simplify}\ (\vec{\alpha}[\nu/\delta], \rho[\nu/\delta])\ \vec{\delta}}{(\vec{\alpha'}, \rho') = \mathsf{simplify}\ (\vec{\alpha}, \rho \cup \{\nu \sqsubseteq \delta\})\ (\delta :: \vec{\delta})}\ \delta \text{ not in } \nu$$

$$\frac{(\vec{\alpha'}, \rho') = \mathsf{simplify}\ (\vec{\alpha}[\nu/\delta], \rho[\nu/\delta])\ \vec{\delta}}{(\vec{\alpha'}, \rho') = \mathsf{simplify}\ (\vec{\alpha}, \rho \cup \{\nu \sqcup \delta \ \sqsubseteq\ \delta\})\ (\delta :: \vec{\delta})}\ \delta \text{ not in } \nu$$

Fig. 9. Eliminating local variables' security type constraints

Observe that in the (LTndef) rule, δ_i are fresh security type variables assigned to the local variables in a node. Since there will be exactly one defining equation for any local variable z_i, note that in constraints ρ', there will be exactly one constraint in which δ_i is on the right, and this is of the form $\nu_i \sqsubseteq \delta_i$. Procedure simplify (Figure 9) serially (in some arbitrary but fixed order) eliminates such type variables via substitution in the types and type constraints. Our definition of simplify here generalises that given for the types of NLustre in [15].

Lemma 3 (Correctness of simplify $(\vec{\alpha}, \rho)\ \vec{\delta}$). *Let ρ be a set of constraints such that for a security type variable δ, there is at most one constraint of the form $\mu \sqsubseteq \delta$. Let s be a ground instantiation of security type variables in a security class lattice SC such that ρ is satisfied by s.*

1. *If* $\rho = \rho_1 \cup \{\nu \sqsubseteq \delta\}$*, where variable* δ *is not in* ν*, then* $\rho_1[\nu/\delta]$ *is satisfied by* s*. (Assume disjoint union.)*
2. *If* $\rho = \rho_1 \cup \{\nu \sqcup \delta \sqsubseteq \delta\}$*, where variable* δ *is not in* ν*, then* $\rho_1[\nu/\delta]$ *is satisfied by* s*. (Assume disjoint union.)*

Lemma 3 is central to establishing that the type signature of a node does not change in the normalisation transformations of §4, which introduce equations involving fresh local program variables.

Revisiting Figure 12, the reader can see the type system at work, with the security types and constraints annotated. Also shown is the simplification of constraints using simplify.

4 Normalisation

We now present Bourke *et al.*'s "normalisation transformations" , which de-nest and distribute operators over lists (tuples) of expressions, and finally transform `fby` expressions to a form where the first argument is a constant.

Normalising an n-tuple of Lustre expressions yields an m-tuple of Lustre expressions without tupling and nesting, and a set of equations, defining fresh local variables (Figure 10). We denote the transformation as

$$([e'_1, \ldots, e'_m]^{\alpha_1,\ldots,\alpha_m}, eqs^{\rho}) \leftarrow \lfloor e_1, ..., e_n \rfloor$$

where we have additionally decorated the transformations of [4] with security types for each member of the tuple of expressions, and with a set of type constraints for the generated equations. We show that the normalisation transformations are indeed *typed transformations*. Our type annotations indicate why security types and constraints of well-security-typed Lustre programs are preserved (modulo satisfaction), as in Theorem 1.

The rules *(Xcnst)-(Xbinop)* for constants, variables, unary and binary operators are obvious, generating no new equations. In rule *(Xwhn)*, where the sampling condition is distributed over the members of the tuple, the security type for each expression is obtained by taking a join of the security type α_i of the expression e'_i with γ, *i.e.*, that of the variable x.

Of primary interest are the rules *(Xfby)* for `fby` and *(Xncall)* for *node call*, where fresh variables x_i and their defining equations are introduced. In these cases, we introduce *fresh* security type variables δ_i for the x_i, and add appropriate constraints. The rules *(Xite)* and *(Xmrg)* resemble *(Xfby)* in most respects. In rule *(Xncall)*, the constraints are obtained from the node signature via substitution.

The rules *(Xbase)* and *(Xon)* for clocks also introduce no equations. The rules *(Xtup)* for tuples (lists) of expressions and *(Xeqs)* for equations regroup the resulting expressions appropriately. The translation of node definitions involves translating the equations, and adding the fresh local variables.

Theorem 1 (Preservation of security types). *Let* $g \in G$ *be a node in* Lustre *program* G*. If the node signature for* g *in* G *is* $\overset{Node}{\vdash}$ `Node f` $(\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}$,

$$\lfloor c \rfloor = ([c^{\bot}], [\,]^{\emptyset}) \qquad \textit{Xcnst}$$
$$\lfloor x^{\alpha} \rfloor = ([x]^{\alpha}, [\,]^{\emptyset}) \qquad \textit{Xvar}$$

$$\lfloor \diamond\ e_1 \rfloor = \text{let } ([e']^{\alpha}, eqs^{\rho}) \leftarrow \lfloor e \rfloor \text{ in } ([\diamond\ e'_1]^{\alpha}, eqs^{\rho}) \qquad \textit{Xunop}$$

$$\lfloor e_1 \oplus e_2 \rfloor = \text{let } ([e'_1]^{\alpha_1}, eqs_1^{\rho_1}) \leftarrow \lfloor e_1 \rfloor \text{ and } ([e'_2]^{\alpha_2}, eqs_2^{\rho_2}) \leftarrow \lfloor e_2 \rfloor \text{ in } ([e'_1 \oplus e'_2]^{\alpha_1 \sqcup \alpha_2}, (eqs_1 \cup eqs_2)^{\rho_1 \cup \rho_2}) \qquad \textit{Xbinop}$$

$$\lfloor \vec{e}\ \texttt{when}\ x^{\gamma} = k \rfloor = \text{let } ([e'_1{}^{\alpha_1}, \ldots, e'_m{}^{\alpha_m}], eqs^{\rho}) \leftarrow \lfloor \vec{e} \rfloor \text{ in } ([e'_1\ \texttt{when}\ x = k^{\alpha_1 \sqcup \gamma}, \ldots, e'_m\ \texttt{when}\ x = k^{\alpha_m \sqcup \gamma}], eqs^{\rho}) \qquad \textit{Xwhn}$$

$$\lfloor \vec{e_0}\ \texttt{fby}\ \vec{e_1} \rfloor = \text{let } (\vec{e'_0}^{\vec{\alpha}}, eqs_0^{\rho_0}) \leftarrow \lfloor \vec{e_0} \rfloor \text{ and } (\vec{e'_1}^{\vec{\beta}}, eqs_1^{\rho_1}) \leftarrow \lfloor \vec{e_1} \rfloor \text{ in } (\vec{x}^{\vec{\delta}}, (\{(x_i = e'_{0i}\ \texttt{fby}\ e'_{1i})_{i=1}^{k}\} \cup eqs_0 \cup eqs_1)^{\rho}) \text{ where } \rho = \{(\alpha_i \sqcup \beta_i \sqsubseteq \delta_i)_{i=1}^{k}\} \cup \rho_0 \cup \rho_1 \qquad \textit{Xfby}$$

$$\lfloor \texttt{merge}\ x^{\gamma}\ \vec{e_1}\ \vec{e_2} \rfloor = \text{let } (\vec{e'_1}^{\vec{\alpha}}, eqs_1^{\rho_1}) \leftarrow \lfloor \vec{e_1} \rfloor \text{ and } (\vec{e'_2}^{\vec{\beta}}, eqs_2^{\rho_2}) \leftarrow \lfloor \vec{e_2} \rfloor \text{ in } (\vec{x}^{\vec{\delta}}, (\{(x_i = \texttt{merge}\ x\ e'_{1i}\ e'_{2i})_{i=1}^{k}\} \cup eqs_1 \cup eqs_2)^{\rho}) \text{ where } \rho = \{(\gamma \sqcup \alpha_i \sqcup \beta_i \sqsubseteq \delta_i)_{i=1}^{k}\} \cup \rho_1 \cup \rho_2 \qquad \textit{Xmrg}$$

$$\lfloor \texttt{if}\ e\ \texttt{then}\ \vec{e_t}\ \texttt{else}\ \vec{e_f} \rfloor = \text{let } (e'^{\kappa}, eqs_c^{\rho_c}) \leftarrow \lfloor e \rfloor \text{ and } (\vec{e'_t}^{\vec{\alpha}}, eqs_t^{\rho_t}) \leftarrow \lfloor \vec{e_t} \rfloor \text{ and } (\vec{e'_f}^{\vec{\beta}}, eqs_f^{\rho_f}) \leftarrow \lfloor \vec{e_f} \rfloor \text{ in } (\vec{x}^{\vec{\delta}}, (\{(x_i = \texttt{if}\ e'\ \texttt{then}\ e'_{ti}\ \texttt{else}\ e'_{fi})_{i=1}^{k}\} \cup eqs)^{\rho}) \text{ where } eqs = eqs_c \cup eqs_t \cup eqs_f \quad \rho = (\kappa \sqcup \alpha_i \sqcup \beta_i \sqsubseteq \delta_i)_{i=1}^{k} \cup \rho_c \cup \rho_t \cup \rho_f \qquad \textit{Xite}$$

$$\lfloor f(e_1, \ldots, e_n) \rfloor = \text{let } ([e'_1, \ldots, e'_m]^{\vec{\alpha'}}, eqs^{\rho_1}) \leftarrow \lfloor e_1, \ldots, e_n \rfloor \text{ in } ([x_1{}^{\delta_1}, \ldots, x_k{}^{\delta_k}], (\{(x_1, \ldots, x_k) = f(e'_1, \ldots, e'_m)\} \cup eqs)^{\rho_2}) \text{ where } \rho_2 = \rho[\vec{\alpha'}/\vec{\alpha}][\vec{\delta}/\vec{\beta}][\gamma'/\gamma] \cup \rho_1 \text{ given } \overset{Node}{\vdash}\ \texttt{Node f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta} \text{ and } \gamma' = \Gamma(\texttt{base}) \qquad \textit{Xncall}$$

$$\lfloor [e_1, \ldots, e_n] \rfloor = \text{let for } i \in \{1, \ldots, n\} : ([e'_{i1}{}^{\alpha_{i1}}, \ldots, e'_{im_i}{}^{\alpha_{im_i}}], eqs_i^{\rho_i}) \leftarrow \lfloor e_i \rfloor \text{ in } ([e'_{11}{}^{\alpha_{11}}, \ldots, e'_{1m_1}{}^{\alpha_{1m_1}}, \ldots, e'_{k1}{}^{\alpha_{k1}} \ldots, e'_{km_k}{}^{\alpha_{km_k}}], (\textstyle\bigcup_{i=1..k} eqs_i)^{\cup_i \rho_i}) \qquad \textit{Xtup}$$

$$\lfloor \texttt{base} \rfloor = \texttt{base} \qquad \textit{Xbase}$$
$$\lfloor ck\ \texttt{on}\ x = k \rfloor = \lfloor ck \rfloor\ \texttt{on}\ x = k \qquad \textit{Xon}$$

$$\lfloor \vec{x}^{\vec{\beta}} =_{ck^{\gamma}} \vec{e} \rfloor = \text{let } (\vec{e'}^{\vec{\alpha}}, eqs^{\rho}) \leftarrow \lfloor \vec{e} \rfloor \text{ in } (\{(\vec{x}_j =_{ck}\ e'_j)_{j=1}^{m}\} \cup eqs)^{\{(\gamma \sqcup \alpha_i \sqsubseteq \beta_i)_{i=1}^{k}\} \cup \rho} \qquad \textit{Xeqs}$$

Fig. 10. Lustre to NLustre normalisation

$$\lfloor x^{\theta} =_{ck^{\gamma}} e_0^{\alpha}\ \texttt{fby}_l\ e^{\beta} \rfloor_{fby} = \begin{cases} xinit^{\delta_1} =_{ck^{\gamma}} \texttt{true}^{\perp}\ \texttt{fby}_{nl}\ \texttt{false}^{\perp} & \perp \sqcup \gamma \sqsubseteq \delta_1 \\ px^{\delta_2} =_{ck^{\gamma}} c^{\perp}\ \texttt{fby}_{nl}\ e^{\beta} & \gamma \sqcup \beta \sqsubseteq \delta_2 \\ x^{\theta} =_{ck^{\gamma}} \texttt{if}\ xinit^{\delta_1}\ \texttt{then}\ e_0^{\alpha} & \gamma \sqcup \delta_1 \sqcup \alpha \sqcup \delta_2 \sqsubseteq \theta \\ \qquad\qquad \texttt{else}\ px^{\delta_2} & \end{cases}$$

Fig. 11. Explicit `fby` initialisation

correspondingly in $\lfloor G \rfloor$ *it is* $\overset{Node}{\vdash}$ `Node` $f\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho'} \vec{\beta}$, *and for any ground instantiation* s, $s(\rho)$ *implies* $s(\rho')$.

The proof is on the DAG structure of G. Here we rely on the modularity of nodes, and the correctness of simplify (Lemma 3). The proof employs induction on the structure of expressions. For the further explicit initialisation of `fby` (Figure 11), the preservation of security via simplify is easy to see.

Semantics preservation. We recall the important results from [4], which establish the preservation of stream semantics by the transformations.

Theorem 2 (Preservation of semantics. Theorem 2 of [4]). *De-nesting and distribution preserve the semantics of* Lustre *programs. (La passe de désimbrication et distributivité préserve la sémantique des programmes.)*

$$\forall G\ f\ \mathbf{xs}\ \mathbf{ys}: \quad G \vdash_{\mathrm{s}} \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys} \implies \lfloor G \rfloor \vdash_{\mathrm{s}} \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}$$

Theorem 3 (Preservation of semantics. Theorem 3 of [4]). *The explicit initialisations of* `fby` *preserve the semantics of the programs. (L'explicitation des initialisations préserve la sémantique des programmes.)*

$$\forall G\ f\ \mathbf{xs}\ \mathbf{ys}: \quad G \vdash_{\mathrm{s}} \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys} \implies \lfloor G \rfloor_{fby} \vdash_{\mathrm{s}} \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}$$

4.1 Example

We adapt an example from [4] to illustrate the translation and security-type preservation. The `re_trig` node in Figure 12 uses the `cnt_dn` node (Figure 21) to implement a count-down timer that is explicitly triggered whenever there is a rising edge (represented by `edge`) on `i`. If the count `v` expires to 0 before a `T` on `i`, the counter isn't allowed restart the count. Output `o` represents an active count in progress.

We annotate the program with security types (superscripts) and constraints for each equation (as comments), according to the typing rules. `cnt_dn` is assumed to have security signature $\overset{Node}{\vdash}$ `Node cnt_dn` $(\alpha_1, \alpha_2)^{\gamma} \xrightarrow{\{\gamma \sqcup \alpha_1 \sqcup \alpha_2 \sqsubseteq \beta\}} \beta$.

Eliminating the security types $\delta_1', \delta_2', \delta_3'$, and δ_6', of the local variables `edge`, `c`, `v` and nested call to `cnt_dn` respectively, we get the constraint $\{\gamma' \sqcup \alpha_1' \sqcup \alpha_2' \sqsubseteq \beta'\}$.

Normalisation introduces local variables (`v21,v22,v24`) with security types $\delta'_4, \delta'_5, \delta'_6$. (Identical names have been used to show the correspondence.) The δ'_i are eliminated by simplify, and the refinement type $\delta'_6\{|\rho'|\}$ for the node call in the LUSTRE version becomes an explicit constraint ρ_5 in NLUSTRE. Observe that the security signature remains the same across the translation.

```
node re_trig(i^α'1 :bool; n^α'2 :int)
  returns (o^β' : bool)
  var edge^δ'1, c^δ'2 :bool,
   v^δ'3 :int;
let
  (edge^ck)^δ'1γ' = i^α'1 and
    (false^⊥ fby (not i^α'1));
-- ρ1L = {⊥ ⊔ α'1 ⊔ ⊥ ⊔ α'1 ⊑ δ'1}
  (c^ck)^δ'2γ' = edge^δ'1 or
    (false^⊥ fby o^β');
-- ρ2L = {γ' ⊔ δ'1 ⊔ ⊥ ⊔ β' ⊑ δ'2}
  (v^c)^δ'3δ'2 = merge c^δ'2
   (cnt_dn((edge^δ'1, n^α'2)
    when c^δ'2))^δ'6{|ρ'|}δ'2
   (0 when not c^δ'2);
-- ρ' = {δ'2 ⊔ (δ'1 ⊔ δ'2) ⊔ (α'2 ⊔ δ'2) ⊑ δ'6}
-- ρ3L = {δ'2 ⊔ δ'2 ⊔ δ'6 ⊔ ⊥ ⊔ δ'2 ⊑ δ'3} ∪ ρ'
   (o^c)^β'δ'2 = v^δ'3 > 0^⊥;
-- ρ4L = {δ'2 ⊔ δ'3 ⊔ ⊥ ⊑ β'}
tel
```

```
node re_trig(i^α'1 :bool; n^α'2 :int)
 returns (o^β' : bool)
 var edge^δ'1, ck^δ'2 :bool, v^δ'3 :int,
  v22^δ'4 :bool, v21^δ'5 :bool,
  v24^δ'6 :int when ck;
let
  v22^δ4 =_δ'2 false^⊥ fby
    (not i^α'1);
-- ρ1 = {δ'2 ⊔ ⊥ ⊔ α'1 ⊑ δ'4}
  edge^δ'1 =_⊥ i^α'1 and v22^δ'4;
-- ρ2 = {⊥ ⊔ α'1 ⊔ δ'4 ⊑ δ'1}
  v21^δ'5 =_⊥ false^⊥ fby o^β';
-- ρ3 = {⊥ ⊔ ⊥ ⊔ β' ⊑ δ'5}
  ck^δ'2 =_γ' edge^δ'1 or v21^δ'5;
-- ρ4 = {⊥ ⊔ δ'1 ⊔ δ'5 ⊑ δ'2}
  v24^δ'6 =_δ'2 cnt_dn(
    edge^δ'1 when ck^δ'2,
   n^α'2 when ck^δ'2);
-- ρ5 = {δ'2 ⊔ (δ'1 ⊔ δ'2) ⊔ (α'2 ⊔ δ'2) ⊑ δ'6}
  v^δ'3 =_δ'2 merge ck^δ'2 v24^δ'6
    (0^⊥ when not ck^δ'2);
-- ρ6 = {δ'2 ⊔ δ'2 ⊔ δ'6 ⊔ ⊥ ⊔ δ'2 ⊑ δ'3}
  o^β' =^δ'2 v^δ'3>0^⊥;
-- ρ7 = {δ'2 ⊔ δ'3 ⊔ ⊥ ⊑ β'}
tel
```

$$\mathsf{simplify}_L\ (\beta', \{\rho_{1L} \cup \rho_{2L} \cup \rho_{3L} \cup \rho_{4L}\})\ \{\delta'_1, \delta'_2, \delta'_3, \delta'_6\} = (\beta', \{\gamma' \sqcup \alpha'_1 \sqcup \alpha'_2 \sqsubseteq \beta'\})$$

$$\mathsf{simplify}_{NL}\ (\beta', \{\rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4 \cup \rho_5 \cup \rho_6 \cup \rho_7\})\ \{\delta'_1, \delta'_2, \delta'_3, \delta'_4, \delta'_5, \delta'_6\} = (\beta', \{\gamma' \sqcup \alpha'_1 \sqcup \alpha'_2 \sqsubseteq \beta'\})$$

Fig. 12. Example: Security analysis and normalisation. `when c` and `when not c` abbreviate `when c = T` and `when c = F`.

5 Security and Non-Interference

We first recall and adapt concepts from our previous work [15].

Lemma 4 (Security of Node Calls; *cf.* Lemma 3 in [15]). *Assume the following, for a call to a node with the given security signature*

$$\overset{Node}{\vdash} \texttt{Node}\ f\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta} \qquad \Gamma \overset{e}{\vdash} \vec{e} : \vec{\alpha'} \qquad \Gamma \overset{e}{\vdash} f(\vec{e}) : \vec{\beta'} \qquad \Gamma \overset{ck}{\vdash} ck : \gamma$$

where ck is the base clock underlying the argument streams $\vec{e}$. Let s be a ground instantiation of type variables such that for some security classes $\vec{t}, w \in SC$: $s(\vec{\alpha'}) = \vec{t}$ and $s(\gamma) = w$.

Now, if ρ is satisfied by the ground instantiation $\{\vec{\alpha} \mapsto \vec{t}, \vec{\beta} \mapsto \vec{u}, \gamma \mapsto w\}$, then the $s(\vec{\beta'})$ are defined, and $s(\vec{\beta'}) \sqsubseteq s(\vec{\beta}\{\!|\rho|\!\})$.

Lemma 4 relates the satisfaction of constraints on security types generated during a node call to satisfaction in a security lattice via a ground instantiation. Again we rely on the modularity of nodes — that no recursive calls are permitted, and nodes do not have free variables.

Definition 1 (Node Security; Definition III.1 in [15]). *Let g be a node in the program graph G with security signature $\overset{Node}{\vdash}$ Node f $(\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}$. Let s be a ground instantiation that maps the security type variables in the set $\{(\alpha_1, \ldots \alpha_n)\} \cup \{(\beta_1, \ldots \beta_m)\} \cup \{\gamma\}$ to the security class lattice SC.*
Node g is secure with respect to s if (i) ρ is satisfied by s; (ii) For each node g' on which g is directly dependent, g' is secure with respect to the appropriate ground instantiations for each call to g' in g as given by Lemma 4.

This definition captures the intuition of node security in that all the constraints generated for the equations within the node must be satisfied, and that each internal node call should also be secure.

The notion of non-interference requires limiting observation to streams whose security level is at most a given security level t.

Definition 2 ($(\sqsubseteq t)$-projected Stream; Definition IV.1 in [15]). *Suppose $t \in SC$ is a security class. Let X be a set of program variables, Γ be security type assumptions for variables in X, and s be a ground instantiation, i.e., $\Gamma \circ s$ maps variables in X to security classes in SC. Let us define $X_{\sqsubseteq t} = \{x \in X \mid (\Gamma \circ s)(x) \sqsubseteq t\}$. Let H_* be a Stream history such that $X \subseteq dom(H_*)$. Define $H_*|_{X_{\sqsubseteq t}}$ as the projection of H_* to $X_{\sqsubseteq t}$, i.e., restricted to those variables that are at security level t or lower:*

$$H_*|_{X_{\sqsubseteq t}}(x) = H_*(x) \quad \text{for } x \in X_{\sqsubseteq t}.$$

Theorem 4 (Non-interference for NLustre; Theorem 5 in [15]). *Let $g \in G$ be a node with security signature*

$$\overset{Node}{\vdash} \texttt{Node}\ f\ \vec{\alpha}^{\gamma} \xrightarrow{\rho} \vec{\beta}$$

which is secure with respect to ground instantiation s of the type variables.
Let eqs be the set of equations in g. Let $X = fv(eqs) - dv(eqs)$, i.e., the input variables in eqs.
Let $V = fv(eqs) \cup dv(eqs)$, i.e., the input, output and local variables.
Let Γ (and s) be such that $\Gamma \overset{eqn}{\vdash} eqs$:> ρ and ρ is satisfied by s. Let $t \in SC$ be any security level. Let bs be a given (base) clock stream.
Let H_* and H'_* be such that

1. *for all $eq \in eqs$: $G, H_*, bs \vdash eq$ and $G, H'_*, bs \vdash eq$, i.e., both H_* and H'_* are consistent Stream histories on each of the equations.*
2. *$H_*|_{X_{\sqsubseteq t}} = H'_*|_{X_{\sqsubseteq t}}$, i.e., H_* and H'_* agree on the input variables which are at a security level t or below.*

Then $H_*|_{V_{\sqsubseteq t}} = H'_*|_{V_{\sqsubseteq t}}$, i.e, H_* and H'_* agree on all variables of the node that are given a security level t or below.

Theorem 5 (Non-interference for Lustre). *If program G is well-security-typed in Lustre, then it exhibits non-interference with respect to Lustre's stream semantics.*

Proof sketch. Let G be well-security-typed in Lustre. This means that each node $g \in G$ is well-security-typed. By induction on the DAG structure of G, using Theorem 1, $\lfloor G \rfloor$ is well-security-typed. By Theorem 4, $\lfloor G \rfloor$ exhibits non-interference. By Theorems 2 and 3, $\lfloor G \rfloor$ and G have the same *extensional* semantics for each node. Therefore, G exhibits non-interference.

6 Conclusions

We have presented a novel security type system for Lustre using the notion of constraint-based refinement (sub)types. Using security-type preservation and earlier results, we have shown its semantic soundness, expressed in terms of non-interference, with respect to the language's stream semantics.

We are developing mechanised proofs of these results, which can be integrated into the Velús verified compiler framework [5].

While Lustre's value type system is quite jejune, this security type system is not. It is therefore satisfying to see that it satisfies a subject reduction property[1]. A difficult aspect encountered during the transcription phase [4] concerns alignment of clocks in the presence of complex clock dependencies. We clarify that our type system, being static, only considers security levels of clocks, not actual clock behaviour, and therefore is free from such complications. Further, the clocks induce no timing side-channels since the typing rules enforce, *a fortiori*, that the security type of any (clocked) expression is at least as high as that of its clock.

[1] At SYNCHRON 2020, De Simone asked Jeanmaire and Pesin whether the terminology "normalisation" used in their work [4] was related in any way to notions of normalisation seen in, *e.g.*, the λ-calculus. It is!

References

1. Auger, C. *Certified compilation of SCADE / LUSTRE.* Theses, Université Paris Sud - Paris XI, Feb. 2013.
2. Bourke, T., Brun, L., Dagand, P.-E., Leroy, X., Pouzet, M., and Rieg, L. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 586–601.
3. Bourke, T., Brun, L., and Pouzet, M. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *Proc. ACM Program. Lang. 4*, POPL (Dec. 2019).
4. Bourke, T., Jeanmaire, P., Pesin, B., and Pouzet, M. Normalisation vérifiée du langage lustre. In *32ièmes Journées Francophones des Langages Applicatifs (JFLA 2021)* (online, Apr. 2021), Y. Regis-Gianas and C. Keller, Eds.
5. Brun, L., Bourke, T., and Pouzet, M. Vélus compiler repository. https://github.com/INRIA/velus, 2020. Accessed: 2020-01-20.
6. Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. LUSTRE: A Declarative Language for Programming Synchronous Systems. In *Proc. 14th Symposium on Principles of Programming Languages (POPL'87). ACM* (1987).
7. Colaço, J., Pagano, B., and Pouzet, M. SCADE 6: A Formal Language for Embedded Critical Software Development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)* (Sep. 2017), pp. 1–11.
8. Denning, D. E. A Lattice Model of Secure Information Flow. *Commun. ACM 19*, 5 (May 1976), 236–243.
9. Goguen, J. A., and Meseguer, J. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982* (1982), IEEE Computer Society, pp. 11–20.
10. Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE 79*, 9 (Sep. 1991), 1305–1320.
11. Jahier, E. *The Lurette V2 User guide*, V2 ed. Verimag, October 2015. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lurette/doc/lurette-man.pdf.
12. Kind 2 group. *Kind 2 User Documentation*, version 1.2.0 ed. Department of Computer Science, The University of Iowa, April 2020. https://kind.cs.uiowa.edu/kind2_user_doc/doc.pdf.
13. Knuth, D. E., and Bendix, P. B. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, J. LEECH, Ed. Pergamon, 1970, pp. 263–297.
14. Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (2009), 107–115.
15. Prasad, S., Yerraguntla, R. M., and Sharma, S. Security types for synchronous data flow systems. In *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)* (2020), pp. 1–12.
16. Raymond, P. *Synchronous Program Verification with Lustre/Lesar.* Wiley, 2010, pp. 171 – 206.
17. Viry, P. Equational rules for rewriting logic. *Theor. Comput. Sci. 285*, 2 (2002), 487–517.
18. Volpano, D., Irvine, C., and Smith, G. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur. 4*, 2–3 (Jan. 1996), 167–187.

A Free Variable Definitions

The definitions of free variables (fv) in expressions and equations, and defined variables (dv) in equations are given in Figure 13 and Figure 14.

$$
\begin{aligned}
fv(c) &= \{\} \\
fv(x) &= \{x\} \\
fv(\diamond\ e) &= fv(e) \\
fv(e_1 \oplus e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\vec{e}\ \texttt{when}\ x = k) &= fv(\vec{e}) \cup \{x\} \\
fv(\texttt{merge}\ x\ \vec{e_1}\ \vec{e_2}) &= \{x\} \cup fv(\vec{e_1}) \cup fv(\vec{e_2}) \\
fv(\vec{e_1}\ \texttt{fby}\ \vec{e_2}) &= fv(\vec{e_1}) \cup fv(\vec{e_2}) \\
fv(\texttt{if}\ e_1\ \texttt{then}\ \vec{e_2}\ \texttt{else}\ \vec{e_3}) &= fv(e_1) \cup fv(\vec{e_2}) \cup fv(\vec{e_3}) \\
fv(f(\vec{e})) &= \bigcup_i fv(\vec{e_i})
\end{aligned}
$$

$$
\begin{aligned}
fv(\texttt{base}) &= \{\texttt{base}\} \\
fv(ck\ \texttt{on}\ x = k) &= fv(ck) \cup \{x\}
\end{aligned}
$$

$$
fv(\vec{e}) = \bigcup_i fv(e_i)
$$

$$
fv(e :: ck) = fv(e) \cup fv(ck)
$$

Fig. 13. Free variables for expressions

$$
fv(\vec{x} = \vec{e}) = fv(\vec{e}) \setminus \{\vec{x}\}
$$

$$
dv(\vec{x} = \vec{e}) = \{\vec{x}\}
$$

$$
\begin{aligned}
fv(x =_{ck} ce) &= fv(ck) \cup fv(ce) \setminus \{x\} \\
fv(x =_{ck} c\ \texttt{fby}\ e) &= fv(ck) \cup fv(e) \setminus \{x\} \\
fv(\vec{x} =_{ck} f(\vec{e})) &= fv(ck) \cup fv(\vec{e}) \setminus \{\vec{x}\}
\end{aligned}
$$

$$
\begin{aligned}
dv(x =_{ck} ce) &= \{x\} \\
dv(x =_{ck} c\ \texttt{fby}\ e) &= \{x\} \\
dv(\vec{x} =_{ck} f(\vec{e})) &= \{\vec{x}\}
\end{aligned}
$$

Fig. 14. Free and defined variables for equations

B Stream Semantics

We present here a specification of core Lustre's co-inductive stream semantics, with some commentary and intuition. This consolidates various earlier pre-

sentations of rules [2,3,15,4], and can be seen as an abstract Coq-independent specification of the semantics encoded in the Vélus development.

The semantics of LUSTRE and NLUSTRE programs are *synchronous*: Each variable and expression defines a data stream which pulses with respect to a *clock*. A clock is a stream of booleans (CompCert/Coq's true and false in Velus). A flow takes its n^{th} value on the n^{th} clock tick, *i.e.*, some value, written $\langle v \rangle$, is present at instants when the clock value is true, and none (written $\langle\rangle$) when it is false. The *temporal operators* `when`, `merge` and `fby` are used to express the complex clock-changing and clock-dependent behaviours of sampling, interpolation and delay respectively.

Formally the stream semantics is defined using predicates over the program graph G, a (co-inductive) stream *history* ($H_* : \mathit{Ident} \to \mathit{value}\ \mathit{Stream}$) that associates value streams to variables, and a clock bs [3,15,4]. Semantic operations on (lists of) streams are written in blue sans serif typeface. Streams are written in red, with lists of streams usually written in **bold face**. All these stream operators, defined co-inductively, enforce the clocking regime, ensuring the presence of a value when the clock is true, and absence when false.

The predicate $G, H_*, bs \vdash e \Downarrow_{\mathbf{e}} \mathbf{es}$ relates an *expression* e to a *list* of streams, written **es**. A list consisting of only a single stream es is explicitly denoted as $[es]$. The semantics of *equations* are expressed using the predicate $G, H_*, bs \vdash \overrightarrow{eq_i}$, which requires *consistency* between the assumed and defined stream histories in H_* for the program variables, as induced by the equations. Finally, the semantics of *nodes* is given as a stream history transformer predicate $G \vDash \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}$.

Figure 15 presents the stream semantics for LUSTRE. While rules for *some* constructs have been variously presented [2,3,15,4], our presentation can be considered as a definitive consolidated specification of the operational semantics of LUSTRE, consistent with the Vélus compiler encoding [5].

Rule (LScnst) states that a constant c denotes a constant stream of the value $\langle c \rangle$ pulsed according to given clock bs. This is effected by the semantic operator const. Rule (LSvar) associates a variable x to the stream given by $H_*(x)$. In rule (LSunop), $\hat{\diamond}$ denotes the operation $\diamond$ *lifted* to apply instant-wise to the stream denoted by expression e. Likewise in rule (LSbinop), the binary operation $\oplus$ is applied paired point-wise to the streams denoted by the two sub-expressions (which should both pulse according to the same clock). In all these rules, an expression is associated with a *single* stream.

The rule (LSwhn) describes *sampling* whenever a variable x takes the boolean value k, from the flows arising from a list of expressions $\vec{e_i}$, returning a list of streams of such sampled values. The predicate $\widehat{\mathsf{when}}$ *maps* the predicate when to act on the corresponding components of *lists* of streams, *i.e.*,

$$\widehat{\mathsf{when}}\ k\ xs\ [es_1, \ldots, es_k] = [os_1, \ldots, os_k] \text{ abbreviates } \bigwedge_{i \in [1,k]} \mathsf{when}\ k\ xs\ es_i = os_i.$$

(Similarly for the predicates $\widehat{\mathsf{merge}}$, $\widehat{\mathsf{ite}}$, and $\widehat{\mathsf{fby}_L}$. The operation $\flat(\_)$ flattens a list of lists (of possibly different lengths) into a single list. Flattening is required since expression e_i may in general denote a *list* of streams $\mathbf{es}_i$.

$$\frac{\mathsf{const}\ c\ bs = cs}{G, H_*, bs \vdash c \Downarrow_{\mathbf{e}} [cs]}\ (\text{LScnst}) \qquad \frac{H_*(x) = xs}{G, H_*, bs \vdash x \Downarrow_{\mathbf{e}} [xs]}\ (\text{LSvar})$$

$$\frac{G, H_*, bs \vdash e \Downarrow_{\mathbf{e}} [es] \quad \widehat{\diamond}\ es = os}{G, H_*, bs \vdash \diamond\ e \Downarrow_{\mathbf{e}} [os]}\ (\text{LSunop})$$

$$\frac{G, H_*, bs \vdash e_1 \Downarrow_{\mathbf{e}} [es_1] \quad G, H_*, bs \vdash e_2 \Downarrow_{\mathbf{e}} [es_2] \quad es_1 \widehat{\oplus} es_2 = os}{G, H_*, bs \vdash e_1 \oplus e_2 \Downarrow_{\mathbf{e}} [os]}\ (\text{LSbinop})$$

$$\frac{\forall i\ G, H_*, bs \vdash e_i \Downarrow_{\mathbf{e}} \mathbf{es}_i \quad H_*(x) = xs \quad \forall i : \widehat{\mathsf{when}}\ k\ xs\ \mathbf{es}_i = \mathbf{os}_i}{G, H_*, bs \vdash \vec{e_i}\ \mathtt{when}\ x = k \Downarrow_{\mathbf{e}} \flat(\vec{\mathbf{os}_i})}\ (\text{LSwhn})$$

$$\frac{\begin{array}{c} G, H_*, bs \vdash e \Downarrow_{\mathbf{e}} [es] \quad \forall i : \quad G, H_*, bs \vdash et_i \Downarrow_{\mathbf{e}} \mathbf{ets}_i \\ \forall j : \ G, H_*, bs \vdash ef_j \Downarrow_{\mathbf{e}} \mathbf{efs}_j \quad \widehat{\mathsf{ite}}\ es\ (\flat(\vec{\mathbf{ets}_i}))\ (\flat(\vec{\mathbf{efs}_j})) = \mathbf{os} \end{array}}{G, H_*, bs \vdash \mathtt{if}\ e\ \mathtt{then}\ \vec{et_i}\ \mathtt{else}\ \vec{ef_j} \Downarrow_{\mathbf{e}} \mathbf{os}}\ (\text{LSite})$$

$$\frac{\begin{array}{c} H_*(x) = xs \quad \forall i : \ G, H_*, bs \vdash et_i \Downarrow_{\mathbf{e}} \mathbf{ets}_i \\ \forall j : \ G, H_*, bs \vdash ef_j \Downarrow_{\mathbf{e}} \mathbf{efs}_j \quad \widehat{\mathsf{merge}}\ xs\ (\flat(\vec{\mathbf{ets}_i}))\ (\flat(\vec{\mathbf{efs}_j})) = \mathbf{os} \end{array}}{G, H_*, bs \vdash \mathtt{merge}\ x\ \vec{et_i}\ \vec{ef_j} \Downarrow_{\mathbf{e}} \mathbf{os}}\ (\text{LSmrg})$$

$$\frac{\begin{array}{c} \forall i : \quad G, H_*, bs \vdash e0_i \Downarrow_{\mathbf{e}} \mathbf{e0s}_i \quad \forall j : \ G, H_*, bs \vdash e_j \Downarrow_{\mathbf{e}} \mathbf{es}_j \\ \widehat{\mathsf{fby}_L}\ (\flat(\vec{\mathbf{e0s}_i}))\ (\flat(\vec{\mathbf{es}_j})) = \mathbf{os} \end{array}}{G, H_*, bs \vdash \vec{e0_i}\ \mathtt{fby}\ \vec{e_j} \Downarrow_{\mathbf{e}} \mathbf{os}}\ (\text{LSfby})$$

$$\frac{\forall i \in [1,k]\ G, H_*, bs \vdash e_i \Downarrow_{\mathbf{e}} \mathbf{es}_i \quad [H_*(x_1), \ldots, H_*(x_n)] = \flat(\vec{\mathbf{es}_i})}{G, H_*, bs \vdash \vec{x_j} = \vec{e_i}}\ (\text{LSeq})$$

$$\frac{\left\{\begin{array}{rl} \mathsf{name} = & \mathtt{f};\ \mathsf{in} = \ \vec{x};\ \mathsf{var} = \ \vec{z}; \\ \mathsf{out} = & \vec{y};\ \mathsf{eqs} = \ \vec{eqn} \end{array}\right\} \in G \quad H_*(n.\mathbf{in}) = \mathbf{xs} \quad \mathsf{base\text{-}of}\ \mathbf{xs} = bs \qquad H_*(n.\mathbf{out}) = \mathbf{ys} \quad \forall eq \in \vec{eq} : \ G, H_*, bs \vdash eq}{G \models \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}}\ (\text{LSndef})$$

$$\frac{G, H_*, bs \vdash \vec{e_i} \Downarrow_{\mathbf{e}} \mathbf{xs} \quad G \models \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}}{G, H_*, bs \vdash f(\vec{e_i}) \Downarrow_{\mathbf{e}} \mathbf{ys}}\ (\text{LSncall})$$

Fig. 15. Stream semantics of Lustre

The expression `merge` $x\ \overrightarrow{et_i}\ \overrightarrow{ef_j}$ achieves (lists of) streams on a faster clock. The semantics in rule (LSmrg) assume that for each pair of corresponding component streams from $\flat(\mathbf{ets}_i)$ and $\flat(\mathbf{efs}_j)$, a value is present in the first stream and absent in the second at those instances where x has a true value $\langle T \rangle$, and complementarily, a value is present in the second stream and absent in the first when x has a false value $\langle F \rangle$. Both values must be absent when x's value is absent. These conditions are enforced by the auxiliary semantic operation merge. In contrast, the conditional expression `if` e `then` $\overrightarrow{et}$ `else` $\overrightarrow{ef}$ requires that all three argument streams es, and the corresponding components from $\flat(\overrightarrow{\mathbf{ets}_i})$ and $\flat(\overrightarrow{\mathbf{efs}_j})$ pulse to the same clock. Again, values are selected from the first or second component streams depending on whether the stream es has the value $\langle T \rangle$ or $\langle F \rangle$ at a particular instant. These conditions are enforced by the auxiliary semantic operation ite. A delay operation is implemented by $e0$ `fby` e. The rule (LSfby) is to be read as follows. Let each expression $e0_i$ denote a list of streams $\mathbf{e0s_i}$, and each expression e_j denote a list of streams $\mathbf{es_j}$. The output list of streams consists of streams whose first elements are taken from the each stream in $\flat(\overrightarrow{\mathbf{e0s}_i})$ with the rest taken from the corresponding component of $\flat(\overrightarrow{\mathbf{es}_j})$. These are achieved using the semantic operation fby_L.

The rule (LSeq) for equations checks the consistency between the assumed meanings for the defined variables x_j according to the history H_* with the corresponding components of the list of streams $\flat(\overrightarrow{\mathbf{es}_i})$ to which a tuple of right-hand side expressions evaluates.

The rule (LSndef) presents the meaning given to the definition named f of a node as a stream list transformer $\widehat{f}$. If history H_* assigns lists of streams to the input and output variables for a node in a manner such that the semantics of the equations in the node are satisfied, then the semantic funnction $\widehat{f}$ transforms input stream list **xs** to output stream list **ys**. The operation base-of finds an appropriate base clock with respect to which a given list of value streams pulse. The rule (LSncall) applies the stream transformer semantic function $\widehat{f}$ to the stream list **xs** corresponding to the tuple of arguments $\overrightarrow{e_i}$, and returns the stream list **ys**.

Clocks and clock-annotated expressions. We next present rules for clocks. Further, we assume that all (NLustre) expressions in equations can be clock-annotated, and present the corresponding rules.

The predicate $H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} bs'$ in Figure 16 defines the meaning of a NLustre clock expression ck with respect to a given history H_* and a clock bs to be the resultant clock bs'. The `on` construct lets us define coarser clocks derived from a given clock — whenever a variable x has the desired value k and the given clock is true. The rules (LSonT), (LSonA1), and (LSonA2) present the three cases: respectively when variable x has the desired value k and clock is true; the clock is false; and the program variable x has the complementary value and the clock is true. The auxiliary operations tl and htl, give the tail of a stream, and the tails of streams for each variable according to a given history H_*. Rules (NSaeA)-(NSae) describe the semantics of clock-annotated expressions, where the output stream carries a value exactly when the clock is true.

$$\frac{}{H_*, bs \vdash \texttt{base} \Downarrow_{\mathsf{ck}} bs} \text{ (LSbase)}$$

$$\frac{H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} (\mathsf{true} \cdot bk) \quad H_*(x) = (\langle k \rangle \cdot xs) \quad (\mathsf{htl}\ H_*), (\mathsf{tl}\ bs) \vdash ck\ \texttt{on}\ x = k \Downarrow_{\mathsf{ck}} bs'}{H_*, bs \vdash ck\ \texttt{on}\ x = k \Downarrow_{\mathsf{ck}} (\mathsf{true} \cdot bs')} \text{ (LSonT)}$$

$$\frac{H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} (\mathsf{false} \cdot bk) \quad H_*(x) = (\langle \rangle \cdot xs) \quad (\mathsf{htl}\ H_*), (\mathsf{tl}\ bs) \vdash ck\ \texttt{on}\ x = k \Downarrow_{\mathsf{ck}} bs'}{H_*, bs \vdash ck\ \texttt{on}\ x = k \Downarrow_{\mathsf{ck}} (\mathsf{false} \cdot bs')} \text{ (LSonA1)}$$

$$\frac{H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} (\mathsf{true} \cdot bk) \quad H_*(x) = (\langle k \rangle \cdot xs) \quad (\mathsf{htl}\ H_*), (\mathsf{tl}\ bs) \vdash ck\ \texttt{on}\ x = \neg k \Downarrow_{\mathsf{ck}} bs'}{H_*, bs \vdash ck\ \texttt{on}\ x = \neg k \Downarrow_{\mathsf{ck}} (\mathsf{false} \cdot bs')} \text{ (LSonA2)}$$

$$\frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} [\langle \rangle \cdot es] \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{false} \cdot cs}{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} [\langle \rangle \cdot es]} \text{ (NSaeA)} \qquad \frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} [\langle v \rangle \cdot es] \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{true} \cdot cs}{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} [\langle v \rangle \cdot es]} \text{ (NSae)}$$

Fig. 16. Stream semantics of clocks and annotated expressions

Stream semantics for NLustre. The semantic relations for NLustre are either identical to (as in constants, variables, unary and binary operations) or else the (simple) singleton cases of the rules given for Lustre (as in `merge`, `ite`, `when`).

The significant differences are in treatment of `fby`, and the occurrence of `fby` and node call only in the context of equations.

The (NSndef') rule only differs from (LSndef) in that post-transcription clock alignment, we have an additional requirement of H_* being in accordance with the base clock bs, enforced by respects-clock. The (NSeq) rule for simple equations mentions the clock that annotates the defining expression, checking that it is consistent with the assumed history for the defined variable x. The (NSfby') rule for `fby` in an equational context uses the semantic operation fby, which differs from fby_L in that it requires its first argument to be a constant rather than a stream. Finally, the rule rule (NSncall') for node call, now in an equational context, is similar to (LSncall) except that it constrains the clock modulating the equation to be the base clock of the input flows.

C Auxiliary Predicate Definitions

The definitions of the auxiliary semantic stream predicates when, const, merge, ite are given in Figure 18. All predicates except fby_L and fby_{NL} (defined in Figure 20) are reused to define semantics of both Lustre and NLustre.

All auxiliary stream operators are defined to behave according to the clocking regime. For example, the rule (DcnstF) ensures the absence of a value when the clock is false. Likewise the unary and binary operators lifted to stream operations

$$\frac{\left\{\begin{array}{l}\mathsf{name} = \mathtt{f};\ \mathsf{in} = \vec{x};\ \mathsf{var} = \vec{z};\\ \mathsf{out} = \vec{y};\ \mathsf{eqs} = \overrightarrow{eqn}\end{array}\right\} \in G \quad H_*(n.\mathbf{in}) = \mathbf{xs} \quad \mathsf{base\text{-}of}\ \mathbf{xs} = bs \qquad \mathsf{respects\text{-}clock}\ H_*\ bs \quad H_*(n.\mathbf{out}) = \mathbf{ys} \quad \forall eq \in \vec{eq} :\ G, H_*, bs \vdash eq}{G \vdash \widehat{f}(\mathbf{xs}) \searrow \mathbf{ys}}\ (\text{NSndef'})$$

$$\frac{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} H_*(x)}{G, H_*, bs \vdash x =_{ck} e}\ (\text{NSeq})$$

$$\frac{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} [vs] \quad \mathsf{fby}_{NL}\ c\ vs = H_*(x)}{G, H_*, bs \vdash x =_{ck} c\ \mathtt{fby}\ e}\ (\text{NSfby'})$$

$$\frac{H_*, bs \vdash \vec{e} \Downarrow_{\mathsf{e}} \mathbf{vs} \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{base\text{-}of}\ \mathbf{vs} \quad G \vdash \widehat{f}(\mathbf{vs}) \searrow \overrightarrow{H_*(x_i)}}{G, H_*, bs \vdash \vec{x} =_{ck} f(\vec{e})}\ (\text{NSncall'})$$

Fig. 17. Stream semantics of NLustre nodes and equations

$\hat{\diamond}$ and $\hat{\oplus}$ operate only when the argument streams have values present, as in (Dunop) and (Dbinop), and mark absence when the argument streams' values are absent, as shown in (DunopA) and (DbinopA). The rules (Dtl) and (Dhtl) are obvious.

Note that in the rules (DmrgT) and (DmrgF) for merge, a value is present on one of the two streams being merged and absent on the other. When a value is absent on the stream corresponding to the boolean variable, values are absent on all streams (DmrgA). The rules for ite require all streams to have values present, *i.e.*, (DiteT) and (DiteF), or all absent, *i.e.*, (DiteA). We have already discussed the when operation in some detail earlier.

The fby_{NL} operation is a bit subtle, and rule (Dfby) may look non-intuitive. However, its formulation corresponds exactly to the Vélus formalisation, ensuring that a value from the first argument stream is prepended exactly when a leading value would have been present on the second argument stream. The operation base-of converts a value stream to a clock, *i.e.*, a boolean stream. The operation respects-clock is formulated corresponding to the Vélus definition.

The main difference between fby_L and fby_{NL} is that the former takes a **stream** while the latter takes a constant value. The fby_{NL} predicate assigns the currently saved constant as the stream value and delays the current operand stream by storing its current value for the next clock cycle (effectively functioning as an initialized D-flip flop). fby_L on the other hand extracts the 0^{th} tick value of the first operand stream and uses the predicate fby_{dl} for delaying.

D Example: A More Detailed Analysis

We adapt the examples given in [4] of translation from Lustre to NLustre, and show how our typing rules and security analysis works. We also illustrate

$$\frac{\mathsf{const}\ bs'\ c = cs'}{\mathsf{const}\ (\mathsf{true}\cdot bs')\ c = \langle c\rangle\cdot cs'}\ (\text{DcnstT}) \qquad \frac{\mathsf{const}\ bs'\ c = cs'}{\mathsf{const}\ (\mathsf{false}\cdot bs')\ c = \langle\rangle\cdot cs'}\ (\text{DcnstF})$$

$$\frac{\widehat{\diamond}\ es' = os' \quad v' = \diamond\ v}{\widehat{\diamond}\ (\langle v\rangle\cdot es') = \langle v'\rangle\cdot os'}\ (\text{Dunop}) \qquad \frac{\widehat{\diamond}\ es' = os'}{\widehat{\diamond}\ (\langle\rangle\cdot es') = \langle\rangle\cdot os'}\ (\text{DunopA})$$

$$\frac{es = v\cdot es'}{(\mathsf{tl}\ es) = es'}\ (\text{Dtl}) \qquad \frac{es'_1\widehat{\oplus}es'_2 = os' \quad v_1\oplus v_2 = v}{(\langle v_1\rangle\cdot es'_1)\widehat{\oplus}(\langle v_2\rangle\cdot es'_2) = \langle v\rangle\cdot os'}\ (\text{Dbinop})$$

$$\frac{x\in dom(H_*)}{(\mathsf{htl}\ H_*)(x) = (\mathsf{tl}\ H_*(x))}\ (\text{Dhtl}) \qquad \frac{es'_1\widehat{\oplus}es'_2 = os'}{(\langle\rangle\cdot es'_1)\widehat{\oplus}(\langle\rangle\cdot es'_2) = \langle\rangle\cdot os'}\ (\text{DbinopA})$$

$$\frac{\mathsf{respects\text{-}clock}\ H_*\ bs\ vs}{\mathsf{respects\text{-}clock}\ H_*\ (\mathsf{false}\cdot bs)\ (\langle\rangle\cdot vs)}\ (\text{DresA}) \qquad \frac{\mathsf{base\text{-}of}\ vs = \ bs}{\mathsf{base\text{-}of}\ (v\cdot vs) = \ \mathsf{true}\cdot bs}\ (\text{Dbase1})$$

$$\frac{\mathsf{respects\text{-}clock}\ H_*\ bs\ vs}{\mathsf{respects\text{-}clock}\ H_*\ (\mathsf{true}\cdot bs)\ (\langle v\rangle\cdot vs)}\ (\text{Dres}) \qquad \frac{\mathsf{base\text{-}of}\ vs = \ bs}{\mathsf{base\text{-}of}\ (\langle\rangle\cdot vs) = \ \mathsf{false}\cdot bs}\ (\text{Dbase2})$$

Fig. 18. Definitions of auxiliary predicates-1

the preservation of the security types during the translation. We annotate the programs with security types (as superscripts) and constraints on them for each equation (as comments), according to the typing rules.

The node `cnt_dn` implements a count down timer `cpt` which is initialized with the value of `n` on 0^{th} tick and whenever there is a `T` on reset `res`. Changing the value of `n` when the reset is `F` doesn't affect the count.

We assign security types α_1 to input `res`, and α_2 to input `n`. The output `cpt` is assigned security type β, and the clock *ck* the type γ. There are no local variables. Based on the rules (LTvar), (LTbinop), (LTfby) and (LTite), we get constraint ρ_L. After simplification, the resultant security signature of `cnt_dn` is given by:

$$\overset{Node}{\vdash}\ \texttt{Node cnt\_dn}\ (\alpha_1,\alpha_2)^{\gamma}\xrightarrow{\{\gamma\sqcup\alpha_1\sqcup\alpha_2\ \sqsubseteq\ \beta\}}\beta$$

The normalisation pass de-nests the `fby` expression and explicitly initializes it into 3 different local streams (`v14`,`v24`,`v25`). These have security types $\delta_1, \delta_2, \delta_3$. The local variables generate constraints ρ_1, ρ_2, ρ_3 which are eliminated by simplify.

The resultant signature of `cnt_dn` in the translated program is also given by:

$$\overset{Node}{\vdash}\ \texttt{Node cnt\_dn}\ (\alpha_1,\alpha_2)^{\gamma}\xrightarrow{\{\gamma\sqcup\alpha_1\sqcup\alpha_2\ \sqsubseteq\ \beta\}}\beta$$

The `re_trig` node in Figure 22 uses the `cnt_dn` node (Figure 21) to implement a count-down timer that is explicitly triggered whenever there is a rising edge (represented by `edge`) on `i`. If the count `v` expires to 0 before a `T` on `i`, the

$$\frac{\mathsf{merge}\ xs'\ ts'\ fs' = os'}{\mathsf{merge}\ (\langle T\rangle \cdot xs')\ (\langle v_t\rangle \cdot ts')\ (\langle\rangle \cdot fs') = \langle v_t\rangle \cdot os'}\ (\text{DmrgT})$$

$$\frac{\mathsf{merge}\ xs'\ ts'\ fs' = os'}{\mathsf{merge}\ (\langle F\rangle \cdot xs')\ (\langle\rangle \cdot ts')\ (\langle v_f\rangle \cdot fs') = \langle v_f\rangle \cdot os'}\ (\text{DmrgF})$$

$$\frac{\mathsf{merge}\ xs'\ ts'\ fs' = os'}{\mathsf{merge}\ (\langle\rangle \cdot xs')\ (\langle\rangle \cdot ts')\ (\langle\rangle \cdot fs') = \langle\rangle \cdot os'}\ (\text{DmrgA})$$

$$\frac{\mathsf{ite}\ es'\ ts'\ fs' = os'}{\mathsf{ite}\ (\langle T\rangle \cdot es')\ (\langle v_t\rangle \cdot ts')\ (\langle v_f\rangle \cdot fs') = \langle v_t\rangle \cdot os'}\ (\text{DiteT})$$

$$\frac{\mathsf{ite}\ es'\ ts'\ fs' = os'}{\mathsf{ite}\ (\langle F\rangle \cdot es')\ (\langle v_t\rangle \cdot ts')\ (\langle v_f\rangle \cdot fs') = \langle v_f\rangle \cdot os'}\ (\text{DiteF})$$

$$\frac{\mathsf{ite}\ es'\ ts'\ fs' = os'}{\mathsf{ite}\ (\langle\rangle \cdot es')\ (\langle\rangle \cdot ts')\ (\langle\rangle \cdot fs') = \langle\rangle \cdot os'}\ (\text{DiteA})$$

$$\frac{\mathsf{when}\ k\ xs'\ es' = os'}{\mathsf{when}\ k\ (\langle\neg k\rangle \cdot xs')\ (\langle v\rangle \cdot es') = \langle\rangle \cdot os'}\ (\text{DwhA1})$$

$$\frac{\mathsf{fby}_{NL}\ v\ xs = ys}{\mathsf{fby}_{NL}\ c\ (\langle v\rangle \cdot xs) = \langle c\rangle \cdot ys}\ (\text{Dfby}) \qquad \frac{\mathsf{when}\ k\ xs'\ es' = os'}{\mathsf{when}\ k\ (\langle k\rangle \cdot xs')\ (\langle v\rangle \cdot es') = \langle v\rangle \cdot os'}\ (\text{Dwhk})$$

$$\frac{\mathsf{fby}_{NL}\ c\ xs = ys}{\mathsf{fby}_{NL}\ c\ (\langle\rangle \cdot xs) = \langle\rangle \cdot ys}\ (\text{DfbyA}) \qquad \frac{\mathsf{when}\ k\ xs'\ es' = os'}{\mathsf{when}\ k\ (\langle\rangle \cdot xs')\ (\langle\rangle \cdot es') = \langle\rangle \cdot os'}\ (\text{DwhA2})$$

Fig. 19. Definitions of auxiliary predicates-2

$$\frac{\mathsf{fby}_{L}\ xs\ ys = os}{\mathsf{fby}_{L}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) = \langle\rangle \cdot os} \qquad \frac{\mathsf{fby}_{dl}\ y\ xs\ ys = os}{\mathsf{fby}_{L}\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) = \langle x\rangle \cdot os}$$

$$\frac{\mathsf{fby}_{dl}\ v\ xs\ ys = os}{\mathsf{fby}_{dl}\ v\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) = \langle\rangle \cdot os} \qquad \frac{\mathsf{fby}_{dl}\ y\ xs\ ys = os}{\mathsf{fby}_{dl}\ v\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) = \langle v\rangle \cdot os}$$

Fig. 20. Lustre's fby semantic predicates

```
node cnt_dn
  (res^α1: bool; n^α2: int)
  returns (cpt^β: int);
let
  (cpt^ck)^βγ = if res^α1 then n^α2
    else (n^α2 fby (cpt^β -1));
  -- ρ_L = {γ ⊔ α1 ⊔ α2 ⊔ α2 ⊔ β ⊔ ⊥⊑β}
tel
```

$\mathsf{simplify}_L\ (\beta, \rho_L)\ \{\} = (\beta, \{\gamma \sqcup \alpha_1 \sqcup \alpha_2 \sqsubseteq \beta\})$

$\mathsf{simplify}_{NL}\ (\beta, (\rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4))\ \vec{\delta}$
$= (\beta, \{\gamma \sqcup \alpha_1 \sqcup \alpha_2 \sqsubseteq \beta\})$
where $\vec{\delta} = \{\{\delta_1, \delta_2, \delta_3\}\}$

```
node cnt_dn
  (res^α1: bool; n^α2: int)
  returns (cpt^β: int);
  var v14^δ1, v24^δ2, v25^δ3:int;
let
  v24^δ2 =^γ true fby false;
  -- ρ1 = {γ ⊔ ⊥ ⊔ ⊥ ⊑ δ2}
  v25^δ3 =^γ 0 fby (cpt^β -1);
  -- ρ2 = {γ ⊔ β ⊔ ⊥ ⊑ δ3}
  v14^δ1 =^γ if v24^δ2 then n
    else v25^δ3;
  -- ρ3 = {γ ⊔ δ2 ⊔ δ3 ⊑ δ1}
  cpt^β =^γ if res^α1 then n^α2
    else v14^δ1;
  -- ρ4 = {γ ⊔ α1 ⊔ α2 ⊔ δ1⊑β}
tel
```

Fig. 21. Example of normalisation with security analysis

counter isn't allowed restart the count. Output `o` represents an active count in progress.

Eliminating the security types $\delta'_1, \delta'_2, \delta'_3$, and δ'_6, of the local variables `edge`, `c`, `v` and nested call to `cnt_dn` respectively, we get the constraint $\{\gamma' \sqcup \alpha'_1 \sqcup \alpha'_2 \sqsubseteq \beta'\}$.

Normalisation introduces local variables (`v21,v22,v24`) with security types $\delta'_4, \delta'_5, \delta'_6$. (Identical names have been used to show the correspondence.) The δ'_i are eliminated by simplify, and the refinement type $\delta'_6\{\!|\rho'|\!\}$ for the node call in the LUSTRE version becomes an explicit constraint ρ_5 in NLUSTRE. We see that the security signature of `re_trig` remains the same.

```
node re_trig(i^α'1 :bool; n^α'2 :int)
  returns (o^β' : bool)
  var edge^δ'1, c^δ'2 :bool,
   v^δ'3 :int;
let
  (edge^ck)^δ'1γ' = i^α'1 and
    (false^⊥ fby (not i^α'1));
-- ρ1L = {⊥ ⊔ α'1 ⊔ ⊥ ⊔ α'1 ⊑ δ'1}
  (c^ck)^δ'2γ' = edge^δ'1 or
    (false^⊥ fby o^β');
-- ρ2L = {γ' ⊔ δ'1 ⊔ ⊥ ⊔ β' ⊑ δ'2}
  (v^c)^δ'3δ'2 = merge c^δ'2
   (cnt_dn((edge^δ'1, n^α'2)
    when c^δ'2))^δ'6{|ρ'|}δ'2
   (0 when not c^δ'2);
-- ρ' = {δ'2 ⊔ (δ'1 ⊔ δ'2) ⊔ (α'2 ⊔ δ'2) ⊑ δ'6}
-- ρ3L = {δ'2 ⊔ δ'2 ⊔ δ'6 ⊔ ⊥ ⊔ δ'2 ⊑ δ'3} ∪ ρ'
   (o^c)^β'δ'2 = v^δ'3 > 0^⊥;
-- ρ4L = {δ'2 ⊔ δ'3 ⊔ ⊥ ⊑ β'}
tel
```

```
node re_trig(i^α'1 :bool; n^α'2 :int)
 returns (o^β' : bool)
 var edge^δ'1, ck^δ'2 :bool, v^δ'3 :int,
  v22^δ'4 :bool, v21^δ'5 :bool,
  v24^δ'6 :int when ck;
let
  v22^δ4 =δ'2 false^⊥ fby
    (not i^α'1);
-- ρ1 = {δ'2 ⊔ ⊥ ⊔ α'1 ⊑ δ'4}
  edge^δ'1 =⊥ i^α'1 and v22^δ'4;
-- ρ2 = {⊥ ⊔ α'1 ⊔ δ'4 ⊑ δ'1}
  v21^δ'5 =⊥ false^⊥ fby o^β';
-- ρ3 = {⊥ ⊔ ⊥ ⊔ β' ⊑ δ'5}
  ck^δ'2 =γ' edge^δ'1 or v21^δ'5;
-- ρ4 = {⊥ ⊔ δ'1 ⊔ δ'5 ⊑ δ'2}
  v24^δ'6 =δ'2 cnt_dn(
    edge^δ'1 when ck^δ'2,
   n^α'2 when ck^δ'2);
-- ρ5 = {δ'2 ⊔ (δ'1 ⊔ δ'2) ⊔ (α'2 ⊔ δ'2) ⊑ δ'6}
  v^δ'3 =δ'2 merge ck^δ'2 v24^δ'6
    (0^⊥ when not ck^δ'2);
-- ρ6 = {δ'2 ⊔ δ'2 ⊔ δ'6 ⊔ ⊥ ⊔ δ'2 ⊑ δ'3}
  o^β' =δ'2 v^δ'3>0^⊥;
-- ρ7 = {δ'2 ⊔ δ'3 ⊔ ⊥ ⊑ β'}
tel
```

$$\mathsf{simplify}_L\ (\beta', \{\rho_{1L} \cup \rho_{2L} \cup \rho_{3L} \cup \rho_{4L}\})\ \{\delta'_1, \delta'_2, \delta'_3, \delta'_6\} = (\beta', \{\gamma' \sqcup \alpha'_1 \sqcup \alpha'_2 \sqsubseteq \beta'\})$$

$$\mathsf{simplify}_{NL}\ (\beta', \{\rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4 \cup \rho_5 \cup \rho_6 \cup \rho_7\})\ \{\delta'_1, \delta'_2, \delta'_3, \delta'_4, \delta'_5, \delta'_6\} = (\beta', \{\gamma' \sqcup \alpha'_1 \sqcup \alpha'_2 \sqsubseteq \beta'\})$$

Fig. 22. Example: Security analysis and normalisation