Liquid Clocks - Refinement Types for Time-Dependent Stream Functions

Jean-Pierre Talpin, Pierre Jouvelot, Sandeep Kumar Shukla

Liquid Clocks Refinement Types for Time-Dependent Stream Functions

Jean-Pierre Talpin, INRIA project-team TEA
Pierre Jouvelot , MINES ParisTech, PSL Research University
Sandeep Kumar Shukla, IIT Kanpur

RESEARCH REPORT

N° 8747

01/06/2015

Project-Team TEA

ISSN 0249-6399

Liquid Clocks
Large Refinement Types for Time-Dependent Stream Functions

Jean-Pierre Talpin[1] , Pierre Jouvelot [2], Sandeep Shukla[3]

Research Report N° 8747 — 01/06/2015 — 40 pages.

Abstract: The concept of liquid clocks introduced in this paper is a significant step towards a more precise compile-time framework for the analysis of synchronous and polychromous languages. Compiling languages such as Lustre or SIGNAL indeed involves a number of static analyses of programs before they can be synthesized into executable code, e.g., synchronicity class characterization, clock assignment, static scheduling or causality analysis. These analyses are often equivalent to undecidable problems, necessitating abstracting such programs to provide sound yet incomplete analyses. Such abstractions unfortunately often lead to the rejection of programs that could very well be synthesized into deterministic code, provided abstraction refinement steps could be applied for more accurate analysis. To reduce the false negatives occurring during the compilation process, we leverage recent advances in type theory -- with the definition of decidable classes of value-dependent type systems -- and formal verification, linked to the development of efficient SAT/SMT solvers, to provide a type-theoretic approach that considers all the above analyses as type inference problems.

In order to simplify the exposition of our new approach in this paper, we define a refinement type system for a minimalistic, synchronous, stream-processing language to concisely represent, analyse, and verify logical and quantitative properties of programs expressed as stream-processing data-flow networks. Our type system provides a new framework to represent logical time (clocks) and scheduling properties, and to describe their relations with stream values and, possibly, other quantas. We show how to analyze synchronous stream processing programs à la Lustre, Signal) to enable previously described analyzes involved in compiling such programs. We also prove the soundness of our type system and elaborate on the adaptability of this core framework by outlining its extensibility to specific models of computations and other quantas.

Key-words: type theory, program verification, data-flow networks, cyber-physical systems

[1] Project-team TEA, INRIA Rennes, France – talpin@irisa.fr

[2] MINES ParisTech, PSL Research University, France – pierre.jouvelot@mines-paristech.fr

[3] IIT Kanpur, India – sandeeps@cse.iitk.ac.in

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Les horloges liquides

Résumé : Le concept d'horloges liquides introduits dans le présent document est une étape importante vers un cadre de compilation de plus de temps précise pour l'analyse des langages synchrones et polychromes. Compilation langues tels que Lustre ou SIGNAL implique en effet un certain nombre d'analyses statiques de programmes avant qu'ils peuvent être synthétisés en code exécutable, par exemple, la synchronicité classe caractérisation, missions d'horloge, ordonnancement statique ou l'analyse de la causalité. Ces analyses sont souvent équivalent à des problèmes indécidables, nécessitant abstraction de tels programmes pour offrir un son encore analyses incomplètes. Ces abstractions conduisent malheureusement souvent le rejet des programmes qui pourraient très bien être synthétisé dans le code déterministe, à condition abstraction étapes de raffinement pourraient être appliquées pour l'analyse plus précise. Pour réduire les faux négatifs qui se produisent au cours du processus de compilation, nous misons sur les progrès récents dans la théorie des types - avec la définition des classes décidables de systèmes de type dépendant de la valeur - et la vérification formelle, liée au développement de l'efficacité des solveurs SAT / SMT, à fournir une approche de type théorie qui considère toutes les analyses ci-dessus que les problèmes d'inférence.

Afin de simplifier l'exposé de notre nouvelle approche dans le présent document, nous définissons un système de type de raffinement pour un, synchrone, la langue minimaliste flux de traitement pour représenter de façon concise, analyser et vérifier les propriétés logiques et quantitatifs de programmes exprimées en flux de données de traitement -Débit réseaux. Notre système de type fournit un nouveau cadre pour représenter le temps logique (horloges) et des propriétés de planification, et de décrire leurs relations avec les valeurs des flux et, éventuellement, d'autres quantas. Nous montrons comment analyser les programmes synchrones de traitement de flux (à la Lustre, Signal) pour permettre décrit précédemment analyses impliqué dans l'établissement de ces programmes. Nous prouvons aussi la solidité de notre système de type et élaborons sur l'adaptabilité de ce cadre de base en décrivant son extensibilité à des modèles spécifiques de calculs et d'autres quantas.

Mots clés : théorie des types, verification de programmes, réseaux flot-de-données, systèmes cyber-physiques

Liquid Clocks

Refinement Types for Time-Dependent Stream Functions

Jean-Pierre Talpin
INRIA Rennes, France
talpin@irisa.fr

Pierre Jouvelot
MINES ParisTech, PSL Research University, France
pierre.jouvelot@mines-paristech.fr

Sandeep Kumar Shukla
IIT Kanpur, India
sandeeps@cse.iitk.ac.in

Abstract

The concept of liquid clocks introduced in this paper is a significant step towards a more precise compile-time framework for the analysis of synchronous and polychromous languages. Compiling languages such as Lustre or SIGNAL indeed involves a number of static analyses of programs before they can be synthesized into executable code, e.g., synchronicity class characterization, clock assignment, static scheduling or causality analysis. These analyses are often equivalent to undecidable problems, necessitating abstracting such programs to provide sound yet incomplete analyses. Such abstractions unfortunately often lead to the rejection of programs that could very well be synthesized into deterministic code, provided abstraction refinement steps could be applied for more accurate analysis. To reduce the false negatives occurring during the compilation process, we leverage recent advances in type theory – with the definition of decidable classes of value-dependent type systems – and formal verification, linked to the development of efficient SAT/SMT solvers, to provide a type-theoretic approach that considers all the above analyses as type inference problems.

In order to simplify the exposition of our new approach in this paper, we define a refinement type system for a minimalistic, synchronous, stream-processing language to concisely represent, analyse, and verify logical and quantitative properties of programs expressed as stream-processing data-flow networks. Our type system provides a new framework to represent logical time (clocks) and scheduling properties, and to describe their relations with stream values and, possibly, other quantas. We show how to analyze synchronous stream processing programs (à la Lustre, Signal) to enable previously described analyzes involved in compiling such programs. We also prove the soundness of our type system and elaborate on the adaptability of this core framework by outlining its extensibility to specific models of computations and other quantas.

Contents

I. Introduction

A cyber-physical system is an entity of heterogeneous constituents: software, embedded in hardware, interfaced with the physical world. Time takes different forms when observed from each of these viewpoints: continuous in physics; discrete and event-based in software; time-triggered in hardware. Moreover, modelling and programming paradigms used to represent time in software (synchronous), hardware (RTL, TLM) or physics (ODEs) significantly alter the perception of time. On top of that, heterogeneous timing constraints need not only be mitigated across system components, but so do all relations in time of its other quantas and metrics: speed, frequency, size, throughput, volume, pressure, capacity, heat, angle, ...

Designing robust control for cyber-physical systems is a challenging problem for control system engineers. Designing resilient control systems for today's power plants, automotive dynamics, or avionics, are problems that are solved mathematically, simulated in MATlab/Simulink, or similar tools, for validation, and then implemented in software by engineers. The process of going from the mathematical equations to software implementation is often error prone, hence the considerable research over the last two decades on automated code synthesis from formal specifications of control algorithms.

Given that control algorithms are equational, to capture the computation involved, data-flow-oriented formal languages such as Kahn networks, Lustre, Signal, SDF (Synchronous Data-Flow) have been proposed in the late 80s [3], [12–14]. Synchronous data-flow-oriented formal languages differ from other data-flow languages in that they are based on the "synchronous hypothesis", which requires that computation be performed as a sequence of reactions to inputs taken from multiple streams of inputs. These input streams are usually either sampled values of physical quantities measured from the physical system under control, or events generated from other parts of the control system such as interrupts, completion signals, acknowledgements, timer signals.

One major difference between languages like SDF, Lustre and Signal lies in the underlying model of time. Even though both adhere to the synchronous hypothesis, they differ in how reactions are ordered, and hence in how reaction time boundaries are formed. In SDF and Lustre, reactions are totally ordered, and form the necessary artefact to create the reaction boundaries, namely 'clocks' have to satisfy certain constraints which are checked at compile time. The notion of 'clock' in a data-flow language captures how data/events streams participate in the reactions.

For example, if the temperature `temp` of a boiler is sampled (from the wire/stream of its sensor `temperature_sensor`) during every reaction (when `check`) to compute an actuation, whereas the `pressure` is only sampled `when` the temperature falls below a certain `threshold`, then the 'clock' of the stream of temperature values is faster than (or equal to) the `check` condition of the stream of pressure values. Moreover, the two clocks are conditionally related by the condition "temp < threshold". In Signal, the timing model being polychronous, the different streams may have non-synchronized clocks, thus giving rise to a partial order relation among them.

```
  temp     = temperature_sensor when check
| sample   = true when temp < threshold
| pressure = pressure_sensor when sample
```

The compilation problem of data-flow syn/poly-chronous languages thus involves a 'clock calculus' which, in the case of Lustre, amounts to checking that all clocks are related to a main one from which 'activation' of every computation is derived [12].

In the case of Signal, this becomes the more general issue of finding, first, if such a main clock exists [2], and, second, how others relate to it. If this is the case, then the program is said 'endochronous' [15], and sequential code can be synthesised from the analysed program, provided that, within each reaction, no cyclic data dependency exists. This is the case above, as `temp` is a sample of `check` (it is defined when stream `check` is true), `temp` is synchronous to `sample` (it is defined at the same times) and `pressure` samples `sample`.

```
      check   <=> clk (temp)
| clk (temp)  <=> clk (sample)
| sample      <=> clk (pressure)
```

If no main clock exists, then each equation or sub-program may still execute separately, as a concurrent thread. In that case, reaction loops may still be synthesised with deterministic synchronisations between them. Such a program is said patient, confluent, or 'weakly endochronous' [24], meaning that the synchronous or asynchronous composition of its equations have the same meaning.

While Lustre is, from the above, better sensed as the synchronous abstraction of sequential programs (that are generated from it), Signal's model of time is better understood as an abstraction, or specification, of a globally asynchronous network of locally synchronous programs. All these problems of establishing endochrony and cycle-freedom are all undecidable: they depend on relations between static clocks, e.g. `pressure`, and dynamic values, e.g. `sample` (hence the integer `temp`). However, they can be solved in most practical cases, by abstract reasoning on Boolean relations (i.e., by abstracting the relation `sample <=> temp < threshold` as above). The set of programs for which these problems are decidable can be extended if we have at our disposal decision algorithms for more elaborate theories, and thus both dependent types, to specify or infer invariants, and Satisfiability Modulo Theories (SMT) solvers, to verify or validate invariants, come in handy.

However, so far, such SMT solver-based extensions of program synthesis have been kludged into the implementation of synthesis engines by adding these decision algorithms into the 'clock calculus' component [9], [23]. This entire process of static analysis, capturing program properties in a logic with suitable decision procedures, is the crux of extending synthesis techniques to quantitative specifications. In this paper, instead of integrating time and quantitative reasoning at the implementation level of the synthesis engine or defining specific type systems to represent each of its logical, quantitative or causal aspect [4], [8], [18], we propose a generic type theory based on refinement types to integrate this ability into the language's static semantic itself and improve synthesisability of programs from its data-flow specifications [9], [23]. The advantage of doing that is manifold, but, most importantly, the type inference algorithms themselves will also decide synthesisability.

We adopt the liquid type theory introduced by Jhala et al. [27], [32], [35], [36] to apply and extend it to the context of timed data-flow languages by the introduction of properties on time and causality: the theory of liquid clocks.

Capturing quantitative properties of timed data-flow specifications, in a decidable manner, using liquid types, opens to a variety of applications, from the integration of contract systems, the traceability of program properties from specification to generated code, to translation validation and certified code generation. Moreover, liquid types allow to revisit many of the ad-hoc and problem-specific algebras and/or type theories that have been proposed to capture many variations of the strictly synchronous hypothesis of Lustre using periodic, multi-rate, affine, regular, integer, cyclo-static, continuous time models, all into one single, unified, verification framework. Liquid types also open to considering a large variety of models of computation and communication, not only synchronous and polychronous data-flow in the spirit of SDF, Simulink, Lustre and Signal, but also multi-rate, cyclo-static, data-parallel models of computation and communication (MoCC) (Appendix VIII-A).

Outline Section II starts with the presentation of a generic data-flow language to describe the network structuring cooperating stream functions. We use a simple, first-order, functional syntax to describe this language. Section III defines liquid types and clocks and Section IV presents our refinement type inference system. A constructive dynamic semantics of the language is given in Section V for the purpose of stating a subject reduction property, Section VI. Section VIII extends our data-flow language with primitives that implement the MoCC of synchronous data-flow languages like Lustre and Signal. In this context, we formulate safety-critical properties of determinism and deadlock-freedom by means of an interpretation of liquid types. Section IX addresses the related work before the conclusion. The appendix contain materials from the complete version of this paper, available as a technical report [31] and for readers perusal. Appendix VIII-A hints on extensions of our liquid clock system to affine data-flow graphs, cyclo-static data-flow networks, high-performance data-parallel networks, real-time calculi. Appendix A gives the proof of subject reduction of the type system.

II. A simple data-flow language

This section presents a minimalistic data-flow language, yet of sufficient expressive capability to manipulate discrete streams of timed events. It takes the form of a lambda-calculus over stream functions. An equation $x = y \star z$ defines the stream x by the repeated application of the operator $\star$ on values incoming from y and z. It usually requires the availability of a value along y and z before performing the operation that defines x. Such an occurrence is called an event of x and its repetition a clock, $\hat{x}$.

Example We consider a simple numerical simulation program similar to the bathtub example of [9]. In the `tank` simulation model below, a faucet is actioned when stream `up` is active, thus filling the tank up, while a `pump` dumps water when stream `down` is active. The `diff` of these two actions modifies the previous tank level to yield a new one, which is `output`. Here, for instance, the stream equation `output = level + diff` awaits a value from the input streams `level` and `diff` to define one on the `output` stream. All three streams are said synchronous: they are defined by the same events which logically relate them in time. All five equations that define the `tank` are meant as simultaneous. The execution of `tank` amounts to choosing a value of its defined output streams from the values available from its input streams in a way that satisfy the equations.

Special stream functions like `pre`, `when` and `merge` are used to delay, sample or merge streams: they primarily manipulate concurrency and time (relations between streams in time). Hence, their meanings greatly differ from one dialect to another, as they are specific to the model of time or concurrency (synchrony, polychrony, asynchrony) that underlies the dynamic semantic of the language.

```
let tank (up, down) =
  let  output = level + diff
     | diff   = faucet - pump
     | level  = output pre 1
     | faucet = (faucet pre 0) + (1 when up)
     | pump   = (pump pre 0) + (1 when down)
  in output
```

They all, however, have similar intended use. For instance, above, `pre` defines a one-time delay: `faucet pre 0` initially returns `0` (the first time `faucet` is defined). Then it returns the previous value of `faucet`. The `alarm`, below, is raised when either `scarce` or `overflow` is signaled. It is the merge of two input streams. A sensor calculation `error` is signaled when both are. It is a sample of a stream (a constant stream, `true`) by one or several conditions.

```
let alert (level) =
  let  overflow = true when level >= 9
     | scarce   = true when 0 >= level
     | alarm    = scarce merge overflow
     | error    = true when scarce when overflow
  in (alarm, error)
```

Expressions From now on, an expression e defines a network of stream functions build from definitions d. An expression can reference a stream x, apply it to a function expression $e\,x$ or add a local definition d in the scope of e with $\mathsf{let}\,d\,\mathsf{in}\,e$. A definition d is either a (non-recursive) function $f(x)=e$ that parameterises the expression e over x, or the simultaneous composition of (possibly recursive) equations $x=e$ to locally define the output of e by x.

$$\begin{array}{rlll@{\qquad}rlll} e & ::= & x & \textit{stream} & d & ::= & f(x)=e & \textit{function} \\ & | & e\,x & \textit{application} & & | & x=e & \textit{equation} \\ & | & \mathsf{let}\,d\,\mathsf{in}\,e & \textit{definition} & & | & d \,\|\, x=e & \textit{composition} \end{array}$$

Meta-variables used in the grammars and rules follow some naming conventions. Streams are noted x, y, z and sometimes c, n if they hold a constant value (a Boolean, an integer). Operators, seen as stream functions, or processes are written f. The constant identifier $()$ stands for void.

Pairing Function `alert` uses pairs: `(alarm, error)`. Pairs do not define new streams, they just bundle streams. We write $\mathsf{fst}\,(x,y)$ and $\mathsf{snd}\,(x,y)$ (or $(x,y)_1$ and $(x,y)_2$) to respectively designate the streams x and y of a pair (x,y). This allows us to use simple pattern matching by identifying $\lambda(x,y).e$ to $\lambda x.\lambda y.e$ and $\mathsf{let}\,(x,y) = e_1\,\mathsf{in}\,e_2$ to $\mathsf{let}\,z = e_1\,\mathsf{in}\,\mathsf{let}\,x = z_1 \,\|\, y = z_2\,\mathsf{in}\,e_2$ for a z not free in $e_{1,2}$.

$$\begin{array}{rcll} e & ::= & \ldots \mid (x,y) \mid \mathsf{fst}\,x \mid \mathsf{snd}\,x & \textit{pair expression} \\ d & ::= & \ldots \mid (x,y) = e & \textit{pair definition} \end{array}$$

Data-types As in related works, we assume expressions e decorated with explicit type polymorphic annotations obtained from classical Hindley-Milner type inference. As usual, we write $\Lambda\alpha.e$ to bind the scope of a data-type variable α to an expression e and $e[b]$ to instantiate it. Similarly, a function definition $f(x\!:\!b) = e$ is decorated with the data-type of its parameter.

$$\begin{array}{rcll} e & ::= & \ldots \mid \Lambda\alpha.e \mid e[b] & \textit{typed expression} \\ d & ::= & \ldots \mid f(x\!:\!b) = e & \textit{typed definition} \end{array}$$

III. Liquid Types

The type system for stream functions defines three classes of types for data, noted b, streams, noted s, and program objects, noted t. A data-type b can be void, a Boolean, an integer, or a type variable α that must be defined in the lexical scope of its occurrence. A stream value variable is noted v and denotes the output of primitive operators on streams. A stream type s is a data-dependent type structure consisting of a value identifier v, a base type b and a property p of v. Streams are first-order objects (they do not carry functions).

A *liquid type* t is either a stream s, the bundle of a type t and another stream s, or a function $x\!:\!s \to t$. The notation $\forall\alpha.t$ allows to define a universally quantified type variable α in the scope of t. The liquid type $x\!:\!s \to t$ of a function associates a property to the type of its parameter s and its result (a program object t). For instance, the type $x\!:\!\mathsf{int} \to \langle y : \mathsf{int} \mid y \geqslant x\rangle \to \langle v\!:\!\mathsf{int} \mid v \geqslant y\rangle$ denotes a function that maps an integer x to a function that, first, expects its argument y to be greater or equal to x and, second, yields a result greater than y.

Finally, a type environment E is a set registering declared type variables and relating stream, bundle and function names to types. We note $E, (x\!:\!t)$, resp. E, α, for the extension of E with a name x. It is important to remind that $E, (x\!:\!t)$ is defined iff $x \notin dom(E)$, resp. $\alpha \notin E$.

$$\begin{array}{rcllrcll} b & ::= & () \mid \mathsf{bool} \mid \mathsf{int} \mid \alpha & \textit{data-type} & t & ::= & s & \textit{stream} \\ s & ::= & \langle v\!:\!b \mid p\rangle & \textit{stream} & & \mid & t \times s & \textit{bundle} \\ & & & & & \mid & \forall\alpha.t & \textit{variable} \\ E & ::= & [] \mid E, (x : t) \mid E, \alpha & \textit{context} & & \mid & x\!:\!s \to t & \textit{function} \end{array}$$

Logical Qualifiers Properties p and atomic logical qualifiers q are defined on the quantifier-free logic of uninterpreted functions and linear arithmetic (QF-EUFLIA or EUFA for short) amenable to automatic verification using, e.g., satisfaction-modulo-theory or theorem proving [20]. Properties are limited to the conjunction of and implication (or equivalence) between qualifiers q, in $\boldsymbol{Q}$. A liquid type always holds a property p (of type boolean) in conjunctive form: p is a conjunctive logical qualifier. We chose to limit properties p to conjunctions and equivalences in order to greatly reduce the size of annotations in programs and facilitate efficient subtyping (to find the greatest implication of a set of clauses that abstract a local variable). Our preliminary experiments show that one can handle many interesting use cases within this limited framework.

$$\begin{array}{rcll} p & ::= & & \textit{liquid refinement} \\ & \mid & q & \textit{qualifier in } \boldsymbol{Q}^\star \\ & \mid & p \wedge p & \textit{conjunction} \\ & \mid & p \Rightarrow p & \textit{implication} \end{array}$$

In the same manner as [36], qualifiers q are typed boolean formulas constructed by a grammar starting from constants b (Booleans true and false), streams x, and unary and binary measures $\star$. Measures $\star$ are uninterpreted boolean operators and integer operators. The clock operator $\hat{x}$ will be of particular interest in our context. It stands for the symbolic or arithmetic period of data on a stream x. It can be a boolean here or an integer (see Section VIII-A), depending on the model of time under consideration. Another peculiarity is the distinction between causal (non-reflexive) input-output stream equality, noted $x = q$, and its implied reflexive (a-causal) logical equality $x == q$. Both are only distinguished from, explained and used in Section VIII.

$$\begin{array}{llll}
q & ::= & \nu \,|\, \star q \,|\, q \star q & \textit{qualifier} \\
\star & ::= & & \textit{measure} \\
 & | & \hat{} & \textit{clock} \\
 & | & \wedge \,|\, \neg \,|\, \Rightarrow \,| & \textit{boolean} \\
 & | & \times \,|\, + \,|\, - \,|\, < \,|\, \leqslant \,|\, = \,|\, == & \textit{integer}
\end{array}$$

IV. Type inference

The proposed type inference system is defined in the spirit of Razou et al. [36] and extended the specification, inference and verification of timed, or clocked, properties. We do not need to use polymorphism at all since all process types are data parameter-dependent; defining the appropriate model and type inference algorithm becomes this way a lot easier, while providing little limitations for our application domain.

Intrinsic stream functions The typing judgment $E \vdash \star : t$ for intrinsic functions $\star$ is very much in the spirit of related work in synchronous programming as to its logical, timed and arithmetic properties [5], [9], [27]. In the remainder, we use the following abbreviations to designate liquid types in which a value variable ν isn't used or when its scope is not ambiguous.

$$b \triangleq \nu : b \triangleq \langle \nu : b \,|\, \mathsf{true} \rangle \qquad b\langle p \rangle \triangleq \langle \nu : b \,|\, p \rangle$$

Intrinsic functions $\star$ are all synchronous. An intrinsic expression/definition $x = y \star z$ requires its input and output streams x, y, z to be present at the same time: it repeatedly pulls values from input streams, computes a result and pushes it on the output stream. This yields two timing invariants. First, the clocks of x, y, z must be true at the same time: $\hat{x} \Leftrightarrow \hat{y} \iff \hat{z}$. Second, the value of the output stream is defined only if its clock is present: $\hat{x} \Rightarrow x = y \star z$.

$$\begin{array}{l}
E \vdash \mathsf{true} \!:\! \langle \nu : \mathsf{bool} \,|\, \hat{\nu} \Rightarrow \nu \rangle \quad E \vdash \mathsf{false} : \langle \nu : \mathsf{bool} \,|\, \hat{\nu} \Rightarrow \neg \nu \rangle \\
E \vdash \mathsf{not} : x : \mathsf{bool} \rightarrow \langle \nu : \mathsf{bool} \,|\, (\hat{x} \Leftrightarrow \hat{\nu}) \wedge \hat{\nu} \Rightarrow (\nu = \neg x) \rangle \\
E \vdash \mathsf{and} \!:\! x : \mathsf{bool} \rightarrow y : \mathsf{bool} \rightarrow \\
\qquad\qquad \langle \nu : \mathsf{bool} \,|\, (\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{\nu}) \wedge \hat{\nu} \Rightarrow (\nu = x \wedge y) \rangle \\
E \vdash \mathsf{or} \;\; : x : \mathsf{bool} \rightarrow y : \mathsf{bool} \rightarrow \\
\qquad\qquad \langle \nu : \mathsf{bool} \,|\, (\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{\nu}) \wedge \hat{\nu} \Rightarrow (\nu = x \vee y) \rangle \\
E \vdash \mathsf{iff} \;\; : x : \mathsf{bool} \rightarrow y : \mathsf{bool} \rightarrow \\
\qquad\qquad \langle \nu : \mathsf{bool} \,|\, (\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{\nu}) \wedge \hat{\nu} \Rightarrow (\nu = x \Leftrightarrow y) \rangle
\end{array}$$

All Boolean and arithmetic intrinsic functions are typed according to that schema to define timed extensions of liquid types for all intrinsic operators. Finally, Boolean and integer constants are lifted to constant streams carrying the specified value. Clocks can be made explicit in the language by the stream function clk.

One subtlety regards the signature of the logical "or" operation. We arguably chose to qualify the value of the disjunction operator by the predicate $\nu = (x \vee y)$, since $x \vee y$ is, in this scope, a qualifier and hence can be considered an uninterpreted function. A similar reasoning is applied to integer stream functions.

$$\begin{array}{l}
E \vdash \mathsf{clk} \;\; : x : b \rightarrow \langle \nu : \mathsf{bool} \,|\, \nu = \hat{x} \rangle \\
E \vdash n \quad\; : \langle \nu : \mathsf{int} \,|\, \hat{\nu} \Rightarrow (\nu = n) \rangle \\
E \vdash \mathsf{plus} \!:\! x : \mathsf{int} \rightarrow y : \mathsf{int} \rightarrow \langle \nu : \mathsf{int} \,|\, (\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{\nu}) \wedge \hat{\nu} \Rightarrow (\nu = x + y) \rangle \\
E \vdash \mathsf{mod} \!:\! x : \mathsf{int} \rightarrow \langle y : \mathsf{int} \,|\, y > 0 \rangle \rightarrow \\
\qquad\qquad \langle \nu : \mathsf{int} \,|\, (\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{\nu}) \wedge \hat{\nu} \Rightarrow (\nu = x \,\mathsf{mod}\, y) \rangle \\
E \vdash \mathsf{pos} \; : x : \mathsf{int} \rightarrow \langle \nu : \mathsf{bool} \,|\, (\hat{\nu} \Leftrightarrow \hat{x}) \wedge \hat{\nu} \Rightarrow (\nu = (x \geqslant 0)) \rangle
\end{array}$$

Example The axioms defined so far give an intuition on what the type of `tank` should be. `output` and `level` could, *hypothetically*, be associated with a type that would literally quote their definition. After all, these properties are supposed to be uninterpreted functions. But this immediately raises three issues, that will be addressed in the next few sub-section thanks to refinement types.

$$\begin{array}{l} E \vdash \mathsf{output}\!:\!\langle v\!:\!\mathsf{int} \,|\, \hat{v} \Leftrightarrow \hat{}\mathsf{level} \Leftrightarrow \hat{}\mathsf{diff} \wedge \hat{v} \Rightarrow v = \mathsf{level} + \mathsf{diff}\rangle \\ E \vdash \mathsf{level}\ \ :\!\langle w\!:\!\mathsf{int} \,|\, \hat{w} \Leftrightarrow \hat{}\mathsf{output} \wedge \hat{v} \Rightarrow v = \mathsf{output}\ \text{“pre”}\ 1\rangle \end{array}$$

First, notice that the property of `output` is defined via the local variable `level`. We could existentially quantify the type of `tank` over it, but this would make type inference undecidable. Refinement types provide a solution to that, by commanding to approximate the properties of `tank` by that of a prime implicant that only mentions the output: it is a reduction or abstraction method. In the case of `tank`, without knowledge on `up` and `down`, we need to conclude `true`, the largest property.

$$E \vdash \mathsf{tank} : \mathsf{up}\!:\!\mathsf{bool} \to \mathsf{down}\!:\!\mathsf{bool} \to \langle v\!:\!\mathsf{int} \,|\, \mathsf{true}\rangle$$

Second, notice that the property of `output` is recursively defined with that of `level`. The resolution of such a constraint requires monotonic fixed-point resolution techniques using abstract interpretation on polyhedra, which are defined by the composition of numerical inequalities [5], [9], [21].

```
output = level + diff | level  = output pre 1
```

Third and last, notice that `level` is "initially" defined by 1 and, "otherwise", by the "previous" value of `output`. Again, "initially", "otherwise" and "previous" refer to individual events, and we cannot quantify our formula over these. Instead, we will need, again, to find a prime implicant that approximates them.

Pairing functions By contrast with other intrinsics, pairing does not create a new stream: it bundles two streams together to form a larger wire or plug. It is more a network-building function than a stream function. It accepts two streams which it bundles into a type t. Conversely, the first and second streams of a pair reference its constituants. As in a dependent type system, name substitution does the rest to track each individual stream properties.

$$\mathsf{pair}\!:\!x\!:\!s \to y\!:\!t \to s \times t \quad \mathsf{fst}\!:\!x\!:\!s \times t \to s \quad \mathsf{snd}\!:\!x\!:\!s \times t \to t$$

Well-formed types The notions of well-formedness and sub-typing play key roles to tackle the issues raised in the examples of the previous section. In a refinement type system, types are subject to the well-formed relation $E \vdash t\,\checkmark$. Its role is crucial. First, it ensures type-consistency of properties generated during type inference. For instance, the rule for stream types $\langle v\!:\!b \,|\, p\rangle$ applies the type inference rules to check that a property has a Boolean type. Second, it ensures proper lexical scoping of terms during type inference.

By using it, the abstraction of qualifiers is strictly enforced once a term escapes the scope of its definition, just as our examples required previously. Abstraction, or reduction, is formalised using the sub-typing relation in the next section. The property of well-formedness extends to type environments E.

$$E \vdash ()\,\checkmark \quad E \vdash \mathsf{bool}\,\checkmark \quad E \vdash \mathsf{int}\,\checkmark \quad \frac{\alpha \in E}{E \vdash \alpha\,\checkmark}$$

$$\frac{E, \alpha \vdash t\,\checkmark}{E \vdash \forall\alpha.t\,\checkmark} \quad \frac{E \vdash s\,\checkmark \quad E, x\!:\!s \vdash t\,\checkmark}{E \vdash (x\!:\!s) \to t\,\checkmark} \quad \frac{E, v\!:\!b \vdash p\!:\!\mathsf{bool}}{E \vdash \langle v\!:\!b \,|\, p\rangle\,\checkmark}$$

$$\frac{E \vdash s\,\checkmark \quad E, x\!:\!s \vdash F\,\checkmark}{E \vdash x\!:\!s, F\,\checkmark} \quad \frac{E \vdash t\,\checkmark \quad E, f\!:\!t \vdash F\,\checkmark}{E \vdash f\!:\!t, F\,\checkmark} \quad \frac{E, \alpha \vdash F\,\checkmark}{E \vdash \alpha, F}$$

In the base rule for stream types $\langle v\!:\!b \,|\, p\rangle\,\checkmark$, we assume that p is well-typed with $E, v\!:\!b \vdash p\!:\!\mathsf{bool}$. This means that we interpret property p as a Boolean expression e check that it does not contain any variable free in the scope of E and v, and that the operation it is composed are of correct arity and type.

Sub-typing The abstraction, or reduction, of inferred properties is performed by using sub-typing. We use the sub-typing relation for liquid types defined in [36] to restrict ourselves to a type system decidable in the QF-EUFLIA theory. The sub-typing rule for streams, Rule (S-SIG), makes use of the logical interpretation

$\llparenthesis p \rrparenthesis$ of a property p (see below) to convey the expected definition that $\llparenthesis E \vdash s \preceq t \rrparenthesis \Rightarrow \llparenthesis E \rrparenthesis \Rightarrow \llparenthesis s \rrparenthesis \Rightarrow \llparenthesis t \rrparenthesis$ is a tautology. The logical meaning of a property p is noted $\llparenthesis p \rrparenthesis$. As in [36], it is interpreted as a formula in the QF-EUFLIA theory, in order to facilitate SAT/SMT checking ($\models q$ indicates that q is a tautology).

$$E \vdash b \preceq b \quad \text{(S-BAS)} \qquad \frac{\models \llparenthesis E, v\!:\!b \rrparenthesis \Rightarrow \llparenthesis p \rrparenthesis \Rightarrow \llparenthesis p' \rrparenthesis}{E \vdash \langle v\!:\!b \,|\, p \rangle \preceq \langle v\!:\!b \,|\, p' \rangle} \quad \text{(S-SIG)}$$

Logical interpretation $\llparenthesis t \rrparenthesis$ is defined by induction on the structure of types and environments as the interpreted conjunction of properties implied by the meaning of a type t. We write $p[x/v]$ for the substitution of v by x in p, and $t\!t$ denotes the true value.

$$\begin{array}{rlrl}
\llparenthesis \alpha \rrparenthesis & \Rightarrow t\!t & \llparenthesis x\!:\!s \times t \rrparenthesis & \Rightarrow \llparenthesis x_1\!:\!s, x_2\!:\!t \rrparenthesis \\
\llparenthesis \langle v : b \,|\, p \rangle \rrparenthesis & \Rightarrow p & \llparenthesis x\!:\!s \rightarrow t \rrparenthesis & \Rightarrow t\!t \\
\llparenthesis x\!:\!\langle v : b \,|\, p \rangle \rrparenthesis & \Rightarrow p[x/v] & \llparenthesis E, x\!:\!t \rrparenthesis & \Rightarrow \llparenthesis E \rrparenthesis \wedge \llparenthesis t \rrparenthesis
\end{array}$$

Example Applied to the case of `tank`, using sub-typing algorithmically amounts to reducing our problem to one of constraint satisfaction. We can easily express the type of `output` as the solution to the set of constraints implied by its definition. Properties "variables" should be calculated from `level` plus `faucet` minus `pump`, and `level` itself by `output` or 1. But we can't say "or" in a conjunctive logic. Instead we need to approximate the output. For convenience in the example, let us name v' the value type of the "previous" v and v_x for the value type of x and p_x for its property. We should for instance infer constraints such as

$$\begin{array}{l}
E \vdash \mathsf{faucet}: \langle v_f\!:\!\mathsf{int} \,|\, p_f \rangle \;\; s.t. \;\; p_f \Rightarrow \mathsf{up} \Rightarrow v_f = v'_f + 1 \\
E \vdash \mathsf{faucet}': \langle v'_f\!:\!\mathsf{int} \,|\, p'_f \rangle \;\; s.t. \;\; v'_f = 0 \Rightarrow p'_f \wedge p_f \Rightarrow p'_f
\end{array}$$

Still, without knowledge on `up` and `down`, the approximation yields a very conservative result: $E \vdash \mathsf{faucet}\!:\!\langle v_f\!:\!\mathsf{int} \,|\, v_f \geqslant 0 \rangle$, $E \vdash \mathsf{pump}\!:\!\langle v_p\!:\!\mathsf{int} \,|\, v_p \geqslant 0 \rangle$ and $E \vdash \mathsf{output}\!:\!\langle v_p\!:\!\mathsf{int} \,|\, \mathsf{true} \rangle$. A better solution is, in the next example, to combine the definition of the talk with that of the up and down buttons.

Inference system We can now put well-formedness and sub-typing to work for type inference. The type inference system inductively defines the relation $E \vdash e\!:\!t$ on expressions. Its co-inductive relation $E \vdash d\!:\!F$ associates a definition d with an environment F under the assumptions E. Structural rules are defined first. Rule (T-FUN) simply references the type t of a named function f. Rule (T-SIG) defines a value stream v from a reference to the named stream x. It hence binds the clock and value of v and x together.

Rules (T-GEN) and (T-INST) respectively bind and instantiate an explicit type variable α as a placeholder for a data-type b. We write $t[s/\alpha]$ for the substitution of α by s in t. Sub-typing is allowed at any time during type inference, by using Rule (T-SUB), but requires the production of a well-formed type t' with respect to the parent environment E. Beta-reduction expressions for application and scoping operate differently. Rule (T-APP) defines the type of the application $e\,y$ of a stream y to a function e by returning its result type t, and by substituting the name of its formal parameter x by that of the actual, y, noted $t[y/x]$. Streams x and y must have the same type s.

$$E, f\!:\!t \vdash f\!:\!t \qquad \text{(T-FUN)}$$

$$E, x\!:\!\langle v\!:\!b \,|\, p \rangle \vdash x\!:\!\langle v\!:\!b \,|\, \hat{x} \Leftrightarrow \hat{v} \Rightarrow v = x \rangle \qquad \text{(T-SIG)}$$

$$\frac{E, \alpha \vdash e : t \quad \alpha \notin E}{E \vdash \Lambda\alpha.e : \forall\alpha.t} \qquad \text{(T-GEN)}$$

$$\frac{E \vdash e : \forall\alpha.t \quad E \vdash s \checkmark}{E \vdash e[s] : t[s/\alpha]} \qquad \text{(T-INST)}$$

$$\frac{E \vdash e\!:\!t \quad E \vdash t \preceq t' \quad E \vdash t' \checkmark}{E \vdash e\!:\!t'} \qquad \text{(T-SUB)}$$

$$\frac{E \vdash e : (x\!:\!s) \rightarrow t \quad E \vdash y : s}{E \vdash e\,y : t[y/x]} \qquad \text{(T-APP)}$$

Rule (T-LET) handles the local definition of d in e. First, the type of d must be inferred. It is an environment F listing the type of all names defined by d. This environment is then used with E to determine the type t of e and return it. However, since t must escape the scope of d, we have to ensure that t does not reference a name introduced in F, hence check that it is well-formed with respect to E: $E \vdash t\checkmark$.

The rule for definition uses the type inference relation $E \vdash e\!:\!t$ to associate names x defined in equations $x\!=\!e$ to the corresponding type environment $(x\!:\!s)$, in Rule (T-DEF). Rule (T-COM) composes them. Rule (T-ABS) is that of lambda-abstraction. It chooses a type $b\langle p\rangle$ for the formal parameter x and deduces the type t of its body e. Because the property p is "chosen", the resulting function type must be well-formed with respect to E.

$$\frac{E,F \vdash d\!:\!F \quad E \vdash F\checkmark \quad E,F \vdash e\!:\!t \quad E \vdash t\checkmark}{E \vdash \mathsf{let}\ d\,\mathsf{in}\,e\!:\!t} \qquad \text{(T-LET)}$$

$$\frac{E \vdash e : s}{E \vdash x\!=\!e : (x\!:\!s)} \qquad \text{(T-DEF)}$$

$$\frac{E \vdash d : F \quad E \vdash x\!=\!e : F'}{E \vdash d \| x\!=\!e\!:\!F, F'} \qquad \text{(T-COM)}$$

$$\frac{E \vdash x\!:\!b\langle p\rangle \checkmark \quad E, x\!:\!b\langle p\rangle \vdash e\!:\!t}{E \vdash f(x\!:\!b)\!=\!e\!:\!(f\!:\!(x\!:\!b\langle p\rangle) \rightarrow t)} \qquad \text{(T-ABS)}$$

Example Let's merge functions `tank` and `control` by the equation `output = tank (control (output pre 1))` we get

```
let   output = level + diff
    | diff   = faucet - pump
    | level  = output pre 1
    | faucet = (faucet pre 0) + (1 when up)
    | pump   = (pump pre 0) + (1 when down)
    | down   = level >= 7
    | up     = level <= 4
in  output
```

Implementing the iterative fixed-point reasoning on local streams mentioned in our previous example yields the observations that:

- `faucet` and `pump` are positive integers;
- but `up` and `down` alternate;
- increments of `faucet` and `pump` alternate;
- the difference between faucet and pump is at most one.

Based on that, one can deduce the type of `output` to be $\langle v\!:\!\mathsf{int} \,|\, 0 \leqslant v \leqslant 8\rangle$ and, using our type inference system, collect its relations with local streams from a proof tree to finally check its validity using an SMT solver [20]. A possible improvement to this dummy example could additionally be to bind `faucet` and `pump` by a maximum throughput, e.g., 10.

V. Constructive semantics

We consider the constructive, small-step, reduction semantics of [30] to describe the behaviour of data-flow networks. Its key feature is to embed a set of stream statuses, e.g. unknown, absent, present, inconsistent, into a lattice of data values $\mathbb{V} = \mathbb{B} \cup \mathbb{Z}$: the domain $\mathbb{D} = \mathbb{V} \cup \{?, \bot, \top, \lightning\}$ of statuses. Statuses have the following meaning: ? stands for unknown, $\bot$ for absent or inhibited, $\top$ for present or activated, and $\lightning$ for inconsistent. We write $\mathbb{V}^{\bot} = \mathbb{V} \cup \{\bot\}$ for the set of final stream statuses. Starting from $\mathbb{D}$, we define a partial order $\sqsubseteq$ on $\mathbb{D} \times \mathbb{D}$: the *greater* a status is, the more information we have about it. In [30], status $\lightning$ is the greatest element. We will not make use of it here: transition from a status to inconsistent will instead block. For all $v \in \mathbb{V}$, one has the relations

$$? \sqsubseteq \bot \sqsubseteq \lightning \text{ and } ? \sqsubseteq \top \sqsubseteq v \sqsubseteq \lightning$$

The reduction of an expression e or definition d is defined by a monotonic progress rule $r, d \rightharpoonup r', d'$ that defines all legal transitions that gain knowledge from r, d by evolving into r', d'. Its parameter r, the registry, associates stream variables x to statuses $\delta \in \mathbb{D}$. A registry is defined by a function $r \in \mathcal{X} \nrightarrow \mathbb{D}$ from a finite set of stream names x in $\mathcal{X}$ to statuses in $\mathbb{D}$. The step relation $\rightharpoonup$ iteratively gains knowledge about the status of the stream variables defined in its domain $dom(r)$.

Example The execution of `tank` is started from a registry knowing the status of input streams, e.g. $(u, \mathit{tt}), (d, \boldsymbol{f})$, but none of the statuses of its local and output streams (noted $(i, ?), (l, ?), (f, ?), (p, ?), (o, ?)$). It is triggered by the environment of the function, which delivers values to its input streams `up` and `down` (noted, $(u, \mathit{tt}), (d, \boldsymbol{f})$). One first determines which synchronous streams must be present and computed: all are; and which must remain absent: none must. Then, the level can be fetched (initially 1). Same for the pump, initially 0, and the faucet, $0 + 1$ because up is true. Hence the output, 2. The output, faucet and pump values $(2, 0, 1)$ are finally stored in the registry in place of the initial values $(1, 0, 0)$ for reuse at the next run.

```
let output = level + diff
  | diff   = faucet - pump
  | level  = output pre 1
  | faucet = (faucet pre 0) + (1 when up)
  | pump   = (pump pre 0) + (1 when down)
in output
```

$$\begin{array}{lll}
 & (u, \mathit{tt}), (d, \boldsymbol{f}), (\delta, ?), (l, ?), (f, ?), (p, ?), (o, \top) & \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, \top), (l, \top), (f, ?), (p, ?), (o, \top) & \text{From 2} \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, \top), (l, \top), (f, \top), (p, \top), (o, \top) & \text{From 3} \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, \top), (l, 1), (f, \top), (p, \top), (o, \top) & \text{From 4} \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, \top), (l, 1), (f, 1), (p, \top), (o, \top) & \text{From 5} \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, \top), (l, 1), (f, 1), (p, 0), (o, \top) & \text{From 6} \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, 1), (l, 1), (f, 1), (p, 0), (o, 2) & \text{From 3} \\
\rightharpoonup & (u, \mathit{tt}), (d, \boldsymbol{f}), (i, 1), (l, 1), (f, 1), (p, 0), (o, 2) & \text{From 2}
\end{array}$$

Reduction to equations Without loss of generality, we reduce our language of stream functions to the scoped composition of synchronous (simultaneous) data-flow equations, seen as base definitions d. In the remainder, base definitions are defined by the composition $d \| d$ of equations $x = y \star z$ built from intrinsic and data-flow operators $\star \in \{\mathsf{and}, \mathsf{or}, \mathsf{not}, \ldots\}$. The scope of a stream x is lexically bound to a definition d by d/x, as in process calculi.

$$d ::= x = y \star z \mid d \| d \mid d/x \qquad \textit{base definitions}$$

The reduction of an expression e to its output x is written $x = e \rightsquigarrow d$. It is recursively defined on the structure of terms by connecting the result of an expression e to a virtual stream x that materialises its continuation [rules (R-COM), (R-LET)]. We note $dom(d)$, $im(d)$ and $fv(d)$ the input, output and unbound streams in d ($dom(d) = fv(d) \backslash im(d)$). Obviously a function cannot be recursive hence (R-FUN) requires $f \notin fv(e')$.

$$\frac{y = e' \rightsquigarrow d' \quad x = \mathsf{let}\, d \,\mathsf{in}\, e \rightsquigarrow d''}{x = \mathsf{let}\, y = e' \| d \,\mathsf{in}\, e \rightsquigarrow (d'' \| d')/y} \quad \text{(R-COM)}, y \notin im(d)$$

$$\frac{y = e' \rightsquigarrow d' \quad x = e \rightsquigarrow d}{x = \mathsf{let}\, y = e' \,\mathsf{in}\, e \rightsquigarrow (d \| d')/y} \quad \text{(R-LET)}$$

$$\frac{x = (e[\lambda y.e'/f]) \rightsquigarrow d}{x = \mathsf{let}\, f(y) = e' \mathsf{in}\, e \rightsquigarrow d} \quad \text{(R-FUN)}, f \notin fv(e')$$

$$\frac{x = (e'[z/y]) \rightsquigarrow d}{x = (\lambda y.e')\, z \rightsquigarrow d} \quad \frac{x = (e'[z/y])\, e \rightsquigarrow d}{x = (\lambda y.e')\, z\, e \rightsquigarrow d} \quad \text{(R-APP)}$$

A local function definition f is substituted by its anonymised definitions $\lambda x.e$, Rule (R-FUN), and its application is expanded (see Rules (R-APP)). This way, each sub-expression connects to the parent stream of its output to form a data-flow network of equations. Again, we write $e[y/x]$ for the substitution of a name x by a term y in a term e. Reduction preserves typing and produces a definition d in static single assignment (SSA) form: all local and output streams have exactly one definition.

Lemma 1 (Reduction preserves typing): If $E \vdash e : s$, $x \notin dom(E)$ and $x = e \rightsquigarrow d$ then $E \vdash d : (x : s)$ (and d is SSA).

Scheduling equations Executing the composition of equations $d \| d'$ consists of choosing a schedule of transitions among its sub-terms d and d' (see Rule (PAR)). A restriction d/x lexically binds the scope of a local stream x to the definition d. Its determines a status $v \in \mathbb{V}^{\bot}$ of x starting from the initial unknown status. This amounts to choosing a fixpoint $\rightharpoonup^*$ of transitions in d.

$$\frac{r, d \rightharpoonup r', d'}{r, d \| d'' \rightharpoonup r', d' \| d''} \quad \frac{r, d \rightharpoonup r', d'}{r, d'' \| d \rightharpoonup r', d'' \| d'} \text{ (PAR)}$$

$$\frac{(r, (x, ?)), d \rightharpoonup^* (r', (x, v)), d'}{r,\ d/x \rightharpoonup r',\ d'/x} \qquad \text{(LET)}, x \notin dom(r)$$

Intrinsic operators The transition rule (op) of a ternary equation $x = y \star z$ relies on the relation $\rightharpoonup_{\star}$ to check progress from the current status $r(y, z, x)$ of its inputs and outputs to an hypothetical triple of statuses (v_y, v_z, v_x). If such a relation exists, a transition occurs and the status of (x, y, z) is updated. We write $r \uplus (x, v)$ to accumulate knowledge of x in r by $(r \uplus (x, v))(x) = r(x) \sqcup v$, where $\sqcup$ is the corresponding lattice lub.

$$\frac{r(y, z, x) \rightharpoonup_{\star} (v_y, v_z, v_x)}{r,\ x = y \star z \rightharpoonup r \uplus (x, v_x) \uplus (y, v_y) \uplus (z, v_z),\ x = y \star z} \quad \text{(op)}$$

Let us first consider the case of a synchronous ternary equation $x = y \, \mathsf{and} \, z$, below. From the initial status $r(y, z, x)$, there are three ways to progress: if one of the parameters is known to be absent, written $\bot$, then the other parameters are deemed absent as well. A second case is when one parameter is known to be present. In that case, the other must be present as well. The last case is when the values $v_y, v_z \in \mathsf{bool}$ of the inputs are known. The only choice is to set the output to $v_y \wedge v_z$.

$r(y)$	$r(z)$	$r(x)$	$\rightharpoonup_{\mathsf{and}}$	$r(y)$	$r(z)$	$r(x)$	$\rightharpoonup_{\mathsf{and}}$
$\bot$	${}^{?}\!/_{\bot}$	${}^{?}\!/_{\bot}$	$\bot\ \bot\ \bot$	$\top$	${}^{?}\!/_{\top}$	${}^{?}\!/_{\top}$	$\top\ \top\ \top$
${}^{?}\!/_{\bot}$	$\bot$	${}^{?}\!/_{\bot}$	$\bot\ \bot\ \bot$	${}^{?}\!/_{\top}$	$\top$	${}^{?}\!/_{\top}$	$\top\ \top\ \top$
${}^{?}\!/_{\bot}$	${}^{?}\!/_{\bot}$	$\bot$	$\bot\ \bot\ \bot$	${}^{?}\!/_{\top}$	${}^{?}\!/_{\top}$	$\top$	$\top\ \top\ \top$
				v_y	v_z	v_x	$v_z\ v_y\ (v_y \wedge v_z)$

As a result of this transition table, one observes that information on absence or presence can possibly flow from the output of an equation back to its inputs, and possibly inhibit or trigger other streams in its context. We note ${}^{?}\!/_{\bot}$ (resp. ${}^{?}\!/_{\top}$) for either unknown ? or absent $\bot$ (resp. either unknown ? or present $\top$) and v_x for any status $v_x \sqsubseteq v_y \wedge v_z$.

Example From the above rules, we can define an operational semantics for clock synchronization. The process $x \, \mathsf{sync} \, y$ only accepts streams x and y that have the same status, present with a value or absent, or yield an error otherwise:

$$x \, \mathsf{sync} \, y \triangleq (a = (x = x) \| b = (y = y) \| c = (a = b)) \, /abc$$

Suppose that both x and y are present. Then, the equations $a = (x = x)$ and $b = (y = y)$ evaluate a and b to true. Therefore, $c = (a = b)$ must be true as well. Now, suppose that both x and y are absent. Then, the equations $a = (x = x)$ and $b = (y = y)$ must evaluate a and b to absent. Therefore, $c = (a = b)$ must be absent as well.

Streams operators The case of the delay equation $x = y \, \mathsf{pre} \, v$ requires a specific rule (delay) to account for the fact that its third argument is a constant $v \in \mathbb{D}$. Apart from this peculiarity, delay behaves like a synchronous operator:

$$\frac{r(y), v, r(x) \rightharpoonup_{\mathsf{pre}} (v_y, w, v_x)}{r,\ x = y\,\mathsf{pre}\,v \rightharpoonup r \uplus (x, v_x) \uplus (y, v_y),\ x = y\,\mathsf{pre}\,w} \quad \text{(delay)}$$

$r(y)$	v	$r(x)$	$\rightharpoonup_{\mathsf{pre}}$			$r(y)$	v	$r(x)$	$\rightharpoonup_{\mathsf{pre}}$		
$^?/_\bot$	v	$\bot$	$\bot$	v	$\bot$	$^?/_\top$	v	$\top$	$\top$	v	v
$\bot$	v	$^?/_\bot$	$\bot$	v	$\bot$	$\top$	v	$^?/_\top$	$\top$	v	v
						w	v	v	w	w	v

The standard evaluation principle applies to downsampling equations $x = y\,\mathsf{when}\,z$. The rules for propagating absence are the same. There is only one possible way to propagate presence, namely if the output value is known. If the input z is false, then the output x is deemed absent regardless of y. Finally, there is only one case where the output can have a value $a \in \mathsf{bool}$, namely when z is true and y equals $a \in \mathbb{D}$.

$r(y)$	$r(z)$	$r(x)$	$\rightharpoonup_{\mathsf{when}}$			$r(y)$	$r(z)$	$r(x)$	$\rightharpoonup_{\mathsf{when}}$		
$\bot$	$^?/_\bot$	$^?/_\bot$	$\bot$	$\bot$	$\bot$	v_y	ff	$^?/_\bot$	v_y	ff	$\bot$
$^?/_\bot$	$\bot$	$^?/_\bot$	$\bot$	$\bot$	$\bot$	$^?/_\top$	$^?/_\top$	$\top$	$\top$	tt	$\top$
$^?/_\bot$	$^?/_\bot$	$\bot$	$\bot$	$\bot$	$\bot$	v_y	tt	$^?/_\top$	v_y	tt	v_y

Prioritised merge $x = y\,\mathsf{default}\,z$ does the opposite to sampling: its result x can only be ruled absent when both inputs are absent. Contrarily to sampling, there are many ways to propagate presence (it is sufficient that one argument is present): if y is present, then its value is forwarded to x; otherwise, if z is present, then its value is forwarded to x. If both are absent, so is x. We note $v_y, v_z \in \mathbb{D}$ for a value (of y, z), and tt, ff for true and false.

$r(y)$	$r(z)$	$r(x)$	$\rightharpoonup_{\mathsf{default}}$			$r(y)$	$r(z)$	$r(x)$	$\rightharpoonup_{\mathsf{default}}$		
$^?/_\bot$	$^?/_\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	v_z	$^?/_\top$	$\top$	v_z	$\top$
$\bot$	$\bot$	$^?/_\bot$	$\bot$	$\bot$	$\bot$	v_y	v_z	$^?/_\top$	v_y	v_z	v_y
v_y	$\top$	$^?/_\top$	v_y	$\top$	$\top$	$\bot$	v_z	$^?/_\top$	$\bot$	v_z	v_z

Example Consider a `countdown` definition in equational form, with input stream n and output stream o. Every time its execution is triggered, its purpose is to provide the value of a count along with its output stream o. If that count reaches 0, the counter synchronizes with the input stream n to reset the count. The local stream c is the current count, x its decrement, y the reset condition.

$$\left(\begin{array}{lll} \ \ c & = & o\ \mathsf{pre}\ 0 \\ \mid o & = & n\ \mathsf{default}\ x \\ \mid x & = & c - 1 \\ \mid y & = & \mathit{true}\ \mathsf{when}\ (c = 0) \\ \mid () & = & n\ \mathsf{sync}\ y \end{array}\right)/c/x/y$$

The execution of the counter is depicted by a series of steps. Changes are marked with a bullet •. The internal state x of the counter is underscored count_x. Let us assume for now that the environment has triggered execution of the program by furnishing the input stream n with the value 1 to start counting. This allows us to determine the output o of the counter by the merge rule. Consequently, and from the delay rule, the initial count 0 can be loaded into c and the new 1 is stored in place of it. Once c is known, x can be decremented by the subtraction rule, and y be deduced by the sampling rule. We are left with the synchronization constraint $n\,\mathsf{sync}\,y$ which, luckily, is true, as both n and y are present.

$$\begin{array}{lllllll} & (c,?) & (n,1_\bullet) & (o,?) & (x,?) & (y,?) & \mathit{count}_0 & \\ \rightharpoonup & (c,?) & (n,1) & (o,1_\bullet) & (x,?) & (y,?) & \mathit{count}_0 & \mathit{from} \rightharpoonup_{\mathsf{default}} \\ \rightharpoonup & (c,0_\bullet) & (n,1) & (o,1) & (x,?) & (y,?) & \mathit{count}_{1_\bullet} & \mathit{from} \rightharpoonup_{\mathsf{pre}} \\ \rightharpoonup & (c,0) & (n,1) & (o,1) & (x,-1_\bullet) & (y,?) & \mathit{count}_1 & \mathit{from} \rightharpoonup_{\mathsf{sub}} \\ \rightharpoonup & (c,0) & (n,1) & (o,1) & (x,-1) & (y,\mathit{tt}_\bullet) & \mathit{count}_1 & \mathit{from} \rightharpoonup_{\mathsf{when}} \\ \rightharpoonup & (c,0) & (n,1) & (o,1) & (x,-1) & (y,\mathit{tt}) & \mathit{count}_1 & \mathit{from} \rightharpoonup_{\mathsf{sync}} \end{array}$$

Let us continue with a second round of execution. Only now, the environment does not deliver a new count and the status of all the streams is unknown. We therefore have to first use the trigger rule to start execution. Here, the aim of the game is to do so in a way that allows all equations to execute. To find a winning strategy, we can use the synchronization relations implied by the counter to determine which stream to trigger.

From the delay, subtraction and merge equations, we can first deduce that c, o, and x are synchronous. We assign them to an equivalence class $V_1 = \{c, o, x\}$ which means that, as soon as one of them is known to be present or absent, all the others have the same status. Next, there is an explicit synchronization constraint between n and y, hence let $V_2 = \{n, y\}$. Now, since n is defined by a sampling of c, we can deduce n from c, hence:

$$\text{status of } V_1 = \{c, o, x\} \text{ known} \Rightarrow \text{status of } V_2 = \{n, y\} \text{ known}$$

Therefore, setting either of c, o or x present should suffice to determine the status of all other streams. Since the value of o depends on x which depends on c, we set c present. From the rule of delay, this has the effect of loading the current state 1 of the count into c and setting the output o present.

We can now execute both the subtraction, setting x to 0, and the sampling, setting y to absent. Once we know the status of y, we can set the status of n to absent as well by the synchronization rule. The output o can now be determined to be x by the merge rule and, concurrently, its value can be stored as the new state of the counter due to the delay equation.

$$\begin{array}{llllllll}
\multicolumn{8}{l}{(c,\top)(n,?)(o,?)(x,?)(y,?),\ \mathit{count}_1} \\
\rightharpoonup(c,1_\bullet) & (n,?) & (o,\top_\bullet) & (x,?) & (y,?) & \mathit{count}_1 & \mathit{from} & \rightharpoonup_{\textsf{pre}} \\
\rightharpoonup(c,1) & (n,?) & (o,\top) & (x,0_\bullet) & (y,?) & \mathit{count}_1 & \mathit{from} & \rightharpoonup_{\textsf{sub}} \\
\rightharpoonup(c,1) & (n,?) & (o,\top) & (x,?) & (y,\perp_\bullet) & \mathit{count}_1 & \mathit{from} & \rightharpoonup_{\textsf{when}} \\
\rightharpoonup(c,1) & (n,\perp_\bullet) & (o,\top) & (x,0) & (y,\perp) & \mathit{count}_1 & \mathit{from} & \rightharpoonup_{\textsf{sync}} \\
\rightharpoonup(c,1) & (n,\perp) & (o,0_\bullet) & (x,0) & (y,\perp) & \mathit{count}_1 & \mathit{from} & \rightharpoonup_{\textsf{default}} \\
\rightharpoonup(c,1) & (n,\perp) & (o,0) & (x,0) & (y,\perp) & \mathit{count}_{0\bullet} & \mathit{from} & \rightharpoonup_{\textsf{pre}}
\end{array}$$

VI. Soundness of liquid clocks

The type environment E of an expression e defines the type of its registry r: it holds the type of all stream variables defined in r. The status of a given stream x at all times is obtained by its clock $\hat{x}$. When looking at boolean models for clocks, we posit, by construction, that $\hat{x}$ is true at a given point in time (a time tag) iff x holds a value at the same relative stream point, or x is activated and holds $x = \top$.

Conversely, $\hat{x}$ is false iff x does not carry a value (with respect to another stream's time tag), denoted $x = \perp$; hence $\hat{x} \Leftrightarrow x \neq \perp$. In the remainder, all properties pertaining to values are guarded or conditioned by properties on clocks (e.g., $\hat{x} \Leftrightarrow v = x$) to enforce stratified reasoning on boolean expressions.

Definition 1 (Type of registry): We say that E is a type environment for r, i.e. $\vdash r : E$, iff $\vdash E\checkmark$, $\mathit{dom}(E) = \mathit{dom}(r)$ and, for all $x \in \mathit{dom}(r)$, $E(x) = \langle v{:}b \,|\, p\rangle$ and $\vdash r(x) : b$. In particular, $\vdash ? : b$, $\vdash \perp : b$, $\vdash \top : b$ for all b.

A registry r is also a, possibly partial, model of properties p specified in E. We say that it is a complete model iff its statuses are values or absence, i.e. $\mathit{im}(r) \subset \mathbb{V}^\perp$. A partial or incomplete model has at least one unknown or undetermined (present) status, i.e. $\mathit{im}(r) \not\subset \mathbb{V}^\perp$. Additionally, we say that a model r' progresses from r, written $r \sqsubseteq r'$, iff $r(x) \sqsubseteq r'(x)$ for all $x \in \mathit{dom}(r) = \mathit{dom}(r')$. We write $r \models (\!|p|\!)$ to mean that the model r satisfies the logical meaning of Property p.

Definition 2 (Model of registry): A complete registry r is a model of E, i.e. $r \models E$ iff $\vdash r{:}E$ and, for all $x \in \mathit{dom}(r)$, $E(x) = \langle v{:}b \,|\, p\rangle$ and $r \models (\!|p[x/v]|\!)$. A partial registry r is a model of E iff there exists a complete model $r' \sqsupseteq r$ s.t. $r' \models E$.

Subject reduction considers an expression $E \vdash e{:}t$ of type t under well-formed hypothesis E and reduces it to the definitions $x{=}e \rightsquigarrow d$ using a fresh $x \notin \mathit{fv}(e)$. If, given a model $r \models E, (x{:}t)$, evaluation progresses by $r, d \rightharpoonup r', d'$, then $r' \models E, (x, t)$ and $E, (x, t) \vdash d' : (x, t)$.

Theorem 1 (Type preservation): Let $E \vdash e : t$, $x \notin dom(E)$, $F = E, x : t$ and $x = e \rightsquigarrow d$. If $r \models F$ and $r, d \rightharpoonup r', d'$ then $r' \models F$ and $F \vdash d' : (x : t)$.

The proof of Theorem 1 is given in Appendix A. It reduces to showing the invariance of E as a model of the updated registry in the calculus of base definitions, by factoring the type inference rules (T-*) with the rewriting rules of $\rightsquigarrow$.

$$\frac{E \vdash d : F \quad E \vdash d' : F'}{E \vdash d \| d' : F, F'} \quad \text{(T-COM)}$$

$$\frac{E, (x : s) \vdash d : F, (x : s) \quad E \vdash s \checkmark \quad E \vdash F \checkmark}{E \vdash d / x : F} \quad \text{(T-LOC)}, x \notin dom(E)$$

Just as for liquid type inference, subject reduction is a type preservation property, not a progress property. It states that the type and property of a program are sound approximations of its streams' statuses. Logical satisfiability does not suffice, however, to guarantee a progress property, as this depends on the calculability of the $\preceq$-maximal type of an expression (it could simply be true and any model would satisfy it).

Conjecture 1 (Progress): Let $E \vdash e : t$ and t be $\preceq$-maximal. If $r \models E, (x : t)$ and, consequently, assigns statuses in $\mathbb{V}^\perp$ to input streams satisfying the clock properties specified in E, then e can execute and return a well-typed result.

Progress depends on the clocks relations implied by the logical properties of a program (the synthesis of a function that defines the clock of all streams) as well as the schedulability of causal relations implied by its value properties (the synthesis of a static schedule for all equations). These are addressed next (Section VIII), where progress is defined by the property of, so-called, patient stream functions.

VII. Typing clock operators

Critical safety properties, such as deadlock-freedom and input-output determinism, are these guaranteeing the correct executability of programs synthesised from data-flow specifications. Synchronous data-flow languages like SDF, Lustre and Signal support similar definitions of these properties, as all support similar primitives to delay, merge and sample streams.

Delay In a synchronous data-flow language, a delay equation $x = y\,\mathsf{pre}\,z$ (in Lustre, `x = z -> pre y`; in Signal `x := y$1 init z`) sends the value v of z (a constant c) along the output stream x and stores the value w of y in place of z. As a result, it delays the delivery of events from stream y by one evaluation tick and prefixes this output by z. To determine the type of pre using axiom (T-PRE) in lieu of (T-FUN) with $f = \mathsf{pre}$, some temporal reasoning is in order. pre initially (at clock i_0) accepts an input constant z and then (at clock $\neg i_0$) outputs the previous value of y (at some clock i_{n-1}) to the output stream v (at clock i_n), $n > 0$. Had we had time tags into the static semantics, we could have written something such as:

$$E, i_0 \vdash v_0 : t_z \preceq t_x \text{ and } \forall n > 0, E, i_n \vdash v_n : t_y^{n-1} \preceq t_x \text{ with } E \vdash s \checkmark$$

Therefore, here, as in [27], the liquid type t_z of z (at the first instant i_0 of v, e.g. $c = 0$) and the liquid type t_y of y (at instants $\neg i_0$ of y, e.g., $1 < y < 9$) need to have a common implicant (e.g. $0 \leqslant v < 9$) that does mention none of the branches or fictive clocks $i_0 \ldots i_n$. Let t_x be that common upper-bound; it is the invariant of x, i.e., the property guaranteed by x at all times. Hence the resulting type for delay (we renamed the arguments):

$$E \vdash \mathsf{pre} : x{:}b\langle p \rangle \to y{:}b\langle p \rangle \to b\langle v{:}b \mid p \wedge \hat{v} \Leftrightarrow \hat{x} \Leftrightarrow \hat{y} \rangle \quad \text{(T-PRE)}$$

Merge Lustre and Signal only differ in the way merge and sampling operations are processed in time. This, in turn, makes an important difference as to how and when clocks can be computed. In Signal, an equation `x := y default z` defines the output x by y if it is present and by z otherwise. In Lustre, merge is implemented by the conditional `x = if c then y else z` over four synchronous streams x, c, y, z. At all times, x takes the value of y if c is true and takes z otherwise.

Path sensitivity reasoning is used to define the axioms default and if by considering the clocks of input streams. If we apply the same reasoning to Lustre's conditional, we get value relations guarded by clocks.

$$\begin{array}{l} E \vdash \mathsf{if}\!: x\!:\!\mathsf{bool} \to y\!:\!b \to z\!:\!b \to \\ \quad \langle v\!:\!b \mid \hat{v} \Leftrightarrow \hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{z} \wedge x \Rightarrow v = y \wedge \neg x \Rightarrow v = z \rangle \end{array}$$

For Signal's default axiom, clocks are these of both input streams (data-flow paths). The output v is defined by the stream y when its clock is present and by the stream z otherwise (at the clock $\hat{z}$ "minus" $\hat{y}$). This analysis yields the conjunction of properties that defines the type of default.

$$\begin{array}{l} E \vdash \mathsf{default}\!: x\!:\!b \to y\!:\!b \to \\ \quad \langle v\!:\!b \mid \hat{v} \Leftrightarrow \hat{x} \vee \hat{y} \wedge \hat{x} \Rightarrow v = x \wedge \hat{y} - \hat{x} \Rightarrow v = y \rangle \end{array}$$

Finally, Lustre's conditional is compatible with Signal, as it can also be written as

```
   x := y when c default z when not c
 | x synchro c synchro y synchro z
```

Sample   The same reasoning applies to the rule for sampling: in Lustre, `x = y when z` and, in Signal, `x := y when z`. In Lustre, y, z are synchronous and x is defined by y when z is true. In Signal, the output stream is defined by y if z is true.

$$\begin{array}{l} E \vdash \mathsf{when}_{lu}\!: x\!:\!b \to y\!:\!\mathsf{bool} \to \langle v\!:\!b \mid \hat{v} \Leftrightarrow \hat{x} \Leftrightarrow \hat{y} \wedge y \Rightarrow v = x \rangle \\ E \vdash \mathsf{when}_{sig}\!: x\!:\!b \to y\!:\!\mathsf{bool} \to \langle v\!:\!b \mid \hat{v} \Leftrightarrow (\hat{x} \wedge \hat{y} \wedge y) \wedge \hat{v} \Rightarrow v = x \rangle \end{array}$$

Example   As we observed in the earlier example of stream function `tank`, sub-typing in the presence of path-sensitivity and name masking implements a property of widening found in abstract interpretation by the automatic (and algorithmically bounded) determination of an implicant to the approximated clauses. In the case of synchronous data-flow, let us consider another classical program, `stopwatch`, from a more algorithmic standpoint.

```
let stopwatch (n:int) =
  let x = y pre 0
    | y = 0 when x >= n default x + 1
  in y
```

Let $E = (n\!:\!\mathsf{int}\langle p_n \rangle, x\!:\!\mathsf{int}\langle p_x \rangle, y\!:\!\langle p_y \rangle)$ with the properties $p_{n,x,y}$ unknown. From Rule (T-PRE),

$$E \vdash x = y \ \mathsf{pre}\ 0\!: p_x, \ \text{s.t.}\ (x = 0) \Rightarrow p_x \text{ and } p_y \Rightarrow p_x$$

From the rule for `when` and the first path of the merge (we informally allow properties into environments to mean they are "and-ed" to them), $E, (x \geqslant n) \vdash y = 0 : p_y$, s.t. $(y = 0 \Leftrightarrow (x \geqslant n)) \Rightarrow p_y$ and $\hat{x} \Leftrightarrow \hat{n}$. From the rule for `default` and the second path of the merge, $E, (x < n) \vdash y = x + 1 : p_y$, s.t. $(y = x + 1 \Leftrightarrow (x < n)) \Rightarrow p_y$. Last, constraint resolution from

$$\begin{array}{l} \hat{x} \Leftrightarrow \hat{y} \\ p_y \Rightarrow p_x \\ (x = 0) \Rightarrow p_x \\ (y = 0 \Leftrightarrow (x \geqslant n)) \Rightarrow p_y \\ (y = x + 1 \Leftrightarrow (x < n)) \Rightarrow p_y \end{array}$$

yields the deduction of $p_y \triangleq (0 \leqslant y \leqslant n)$ and the type of the stopwatch.

$$\mathsf{stopwatch} : n\!:\!\mathsf{int} \to y\!:\!\mathsf{int}\langle (\hat{y} \Leftrightarrow \hat{n}) \wedge (0 \leqslant y \leqslant n) \rangle$$

VIII. Safety Properties

Critical safety properties are these guaranteeing the correct executability of programs synthesised from data-flow specifications. Those pertaining to the dead-lock freedom and schedulability of synchronous data-flow programs can be deduced directly from liquid types and represented using specific algebraic structures.

Schedulability The determination of a static schedule of operations is an essential part of the compilation of synchronous languages. Imperative synchronous languages such as Esterel and SyncCharts define graphs representing write-to-read relations between program instructions [1], [16]. Data-flow synchronous languages similarly define graphs to materialise causal dependencies from definitions to uses of stream values in programs [17], [25].

Liquid types allows us to formulate schedule synthesis from the type of a data-flow network. This is done by constructing a guarded scheduling graph (in the form of [23]) from the interpretation of uninterpreted guarded equations present in liquid types. We implement this by the introduction of qualifiers q, of the form $\hat{y} \Rightarrow x \rightarrow z$, that abstract (or, equivalently, are implied by) inferred properties of the form $\hat{y} \Rightarrow z = q[x]$, where a context $q[]$ is defined by a term with a hole. The relation of dominance $x \gg y$ will be explained in the next section.

$$\begin{array}{lll} a & ::= v \mid \hat{v} & \textit{clock and value qualifiers} \\ q & ::= \ldots \mid a \rightarrow a \mid a \gg a & \textit{abstract properties} \\ q[] & ::= [] \mid \star q[] \mid q[] \star q \mid q \star q[] & \textit{context} \end{array}$$

The type of a synchronous data-flow network features properties of two forms: relations between clocks and/or boolean conditions as well as (directed) equalities guarded by clocks $\hat{x} \Rightarrow y = q[z]$. One can interpret the latter as causal relations to build a graph of scheduling relations of the form $z \rightarrow^{\hat{x}} y$ to mean that the computation of z precedes that of y at Clock $\hat{x}$.

However, a stream equality $x = y$ is causal. It hence needs to be disambiguated from logical equality, now written $x == y$, as promised earlier and to avoid confusion. As a result, Axiom (c-eq) below decomposes an equation $x = y$ into a logical equality $x == y$ and the causality relations $x \leftarrow y$ and $\hat{x} \rightarrow x$, for all $q, q'[], x, y$.

Additionally, in Rule (c-clk), a clock $\hat{v}$ defined by a boolean condition $q[x]$ depends on the value of stream x, i.e. $\hat{v} \Leftrightarrow q[x]$. Therefore, it implies a causality relation $x \rightarrow^{\hat{x}} \hat{v}$, as the clock of v cannot be determined before x is computed. Last, and thanks to the sub-typing rule (T-SUB), the introduction (and elimination) of these relations can be performed in any place suited, e.g. to abstract a local stream x in a definition d from its type, using the transitive rule (c-trs).

$$\begin{array}{ll} \models (\!| q \Rightarrow x = q'[y] |\!) \Rightarrow (\!| q \Rightarrow x == q'[y] \wedge q \Rightarrow y \rightarrow x \leftarrow \hat{x} |\!) & \text{(c-eq)} \\ \models (\!| \hat{x} \Leftrightarrow q[y] |\!) \Rightarrow (\!| y \rightarrow \hat{x} |\!) & \text{(c-clk)} \\ \models (\!| q \Rightarrow a \rightarrow a' \wedge q' \Rightarrow a' \rightarrow a'' |\!) \Rightarrow (\!| (q \wedge q') \Rightarrow a \rightarrow a'' |\!) & \text{(c-trs)} \end{array}$$

We note $\vec{E}$ the scheduling graph implied by E and note $\vec{E}^*$ for its transitive closure. $\vec{E}$ can also be seen as an interpretation of E pertaining to its scheduling logic. Scheduling graphs are naturally subject to a set-theoretic containment relation, noted $\subseteq$. Now, since our liquid type system permits property elimination using the sub-typing rule, not all types of a given data-flow network are good candidates for causal analysis. In fact, the only equivalence class of interest is that maximal with respect to $\subseteq$.

We say that E is $\subseteq$-*maximal* (w.r.t. d) iff $E \vdash d : E$ and, for any other $F \vdash d : F$, $\vec{F}^* \subseteq \vec{E}^*$. Definition d is deadlock-free iff all cycles $q \Rightarrow v \rightarrow v$ of its maximal scheduling graph invalidate q.

Definition 3 (Schedulability): Let $E \vdash d : E$ and E be $\subseteq$-maximal for d. d is schedulable iff, for all $q \Rightarrow x \rightarrow x \subseteq \vec{E}^*$, $\models (\!| E |\!) \Rightarrow \neg q$. We say that d/x is schedulable iff d is.

Example Consider the `countdown` function of the previous section. From its definition and/or type specification, one syntactically obtains the following causal stream relations.

```
let countdown (n) =
    let c = o pre 0
    |   o = n default x
    |   x = c - 1
    |   y = true when (c=0)
in synchro n y; o
```

$$\begin{array}{rcl} \hat{n} & \Rightarrow & n \to o \\ \hat{x} - \hat{n} & \Rightarrow & x \to o \\ \hat{c} & \Rightarrow & c \to x \\ \hat{c} & \Rightarrow & c \to \hat{y} \end{array}$$

Patience In complement to schedulability analysis, the correctness of a synchronous data-flow network relies on a definition of timed determinism: a correct synchronous program is said *endochronous* iff it is able to autonomously decide when its streams need to be read or written, i.e. when their status is present or absent. In Lustre, this is quite simple since all input streams are, by construction, synchronous: a Lustre (or Simulink or SDF) program is an endochronous stream function, by construction.

In Signal, endochrony or constructivity is defined with respect to the asynchronous stream interface of synchronous programs [30]: the capability of a synchronous program to deterministically peek values from input streams based on the internal logic of its specification. This internal, or implicit, computation of the program is synthesised by the compiler from a reasoning on clocks, called hierarchisation. Hierarchisation attempts to build a dominance relation between clocks to order to sort those that can be computed from the others: a dominance $x \gg y$ means that the clock of y can be computed or deduced from the one of x. It is, by construction, a pre-order or graph. Its aim is, here, to abstract qualifiers in E.

$$\begin{array}{ll} \models \llparenthesis \hat{x} \Leftrightarrow \hat{y} \rrparenthesis \Rightarrow \llparenthesis x \simeq y \rrparenthesis & \text{(c-sim)} \\ \models \llparenthesis \hat{x} \Leftrightarrow q[y] \rrparenthesis \Rightarrow \llparenthesis y \gg x \rrparenthesis & \text{(c-sub)} \\ \models \llparenthesis \hat{x} \Leftrightarrow q[y, z] \wedge (v \gg y) \wedge (v \gg z) \rrparenthesis \Rightarrow \llparenthesis v \gg x \rrparenthesis & \text{(c-op)} \end{array}$$

Dominance can be defined over clock equivalence classes to facilitate analysis. We say that $\hat{x}$ and $\hat{y}$ are synchronous, or $\simeq$-equivalent in E, written $\hat{x} \simeq \hat{y}$ iff E implies $\hat{x} \Leftrightarrow \hat{y}$.

Moreover, notice that the dominance graph of a schedulable program is acyclic. Indeed, $y \gg x \gg y$ would require $\hat{x} \Leftrightarrow q[y]$ and $\hat{y} \Leftrightarrow q'[x]$, hence $x \to \hat{y} \to y \to \hat{x}$ and thus $x \to x$. We call $\tilde{E}$ the dominance graph of E. The dominance relation is subject to set-theoretic inclusion and we shall consider the $\subseteq$-maximal dominance relation for a given d.

Definition 4 (Patience): Let $E \vdash d : E$ and $\tilde{E}$ be $\subseteq$-maximal. d is patient iff $\tilde{E}$ has one $\gg$-maximal $\simeq$-equivalence class.

Example Consider the countdown function again. From its definition and/or type specification, one syntactically obtains the following clock relations.

```
let countdown (n) =
    let c = o pre 0
    |   o = n default x
    |   x = c - 1
    |   y = true when (c=0)
in synchro n y; o
```

$$\begin{array}{rcl} c & \simeq & o \\ c & \simeq & x \\ c & \gg & y \\ y & \simeq & n \end{array}$$

This provides a strategy to evaluate the status of streams in countdown. Execution is triggered by a stream of the $\gg$-highest class $\{o, c, x\}$, here obviously o, by setting its status to present. The status of all other streams can be deduced from that of o.

Progress with patience The above yields a formulation of the notion of progress in the framework of liquid types: if d is schedulable and patient, then it progresses in the sense of Conjecture 1. In Lustre, and similarly Simulink, Ptolemy's SDF, all the input streams of a block are synchronous, by construction. This implies that the $\gg$-maximal class of a program contains all its input streams. Hence the proposition that patient and schedulable programs progress.

Proposition 1 (Synchronous progress): Let $E \vdash d : E$, E be $\subseteq$-$\gg$-maximal and d be a schedulable (and patient) Lustre program. Let $\vdash r : E$ such that $r(x) \in \mathbb{V}^{\perp}$ for all $x \in dom(d)$ and $r(x) = ?$ otherwise. Then, $r, d \rightharpoonup^{*} r', d'$ and $im(r') \subset \mathbb{V}^{\perp}$.

In addition to that, the Signal language allows for a program to internally control the delivery of values along input streams. To represent this feature, [30] defines the interface semantics between a synchronous process and an asynchronous network of FIFO buffers, reminiscent of Kahn networks [13], and shows that a patient process is constructive with respect to that interface (i.e. asynchronously constructive).

To keep the presentation within the synchronous semantics of Section V, we instead assume a set of available inputs values I for a given definition d, and show how execution progresses by iteratively choosing and peeking these values to produce outputs. Let $\max_{\gg}(d)$ be the $\gg$-maximal $\simeq$-equivalence class of streams in d. Call $succ_{\gg}(X) = \cup_{X \gg Y} Y$ the union of X's immediate successors in $\gg$. Again, patient and schedulable programs progress, Proposition 2.

Proposition 2 (Controlled progress): Let $E \vdash d : E$, E be $\subseteq$-$\gg$-maximal. Let

- $I = dom(d)$, $O = im(d)$ be the inputs and outputs of d,
- $V : I \to \mathbb{V}$ be a set of well-typed input values $\vdash V : E_V$
- $r : I \cup O \to \mathbb{D}$ be a registry and set its image to unknown except for $\max_{\gg}(d)$ (called its trigger):

$$\forall x \in dom(r), \text{if } x \in \max_{\gg}(d) \text{ then } r(x) = V(x) \text{ else } r(x) = ?$$

If $d = d_0$ schedulable and patient, then progress is defined by a finite series of steps $0 \leqslant n < N \leqslant |I|$ such that

1) Computation strictly progresses:

$$r_n, d_n \rightharpoonup^* r'_n, d'_n \wedge \exists x \in fv(d'_n),\ r_n(x) \sqsubset r'_n(x)$$

2) If $r(I) \not\subset \mathbb{V}^{\perp}$, define $d_{n+1} = d'_n$ and r_{n+1} by

$$\forall x \in I, \text{if } r'_n = \top \text{ then } r_{n+1}(x) = V(x) \text{ else } r_{n+1}(x) = r'_n(x)$$

and repeat step 1.

3) Otherwise, all inputs have been fetched and outputs are now defined: $r'_n(O) \subset \mathbb{V}^{\perp}$, $N = n + 1$.

Example An example of a patient program is `countdown`, in Section V, whose execution can be triggered by setting its output stream to present, to then read the input if needed in its state. It first iterates until the status of the input is known. Then, the value of n, here 42, can be loaded (input), and the iteration resumed until the output is computed. 42 is the answer.

$$\begin{array}{lllllll}
(c,?) & (n,?) & (o,\top_\bullet) & (x,?) & (y,?) & count_0 & \\
\rightharpoonup(c,\top_\bullet) & (n,?) & (o,\top) & (x,?) & (y,?) & count_0 & from \rightharpoonup_{\mathsf{pre}} \\
\rightharpoonup(c,\top) & (n,?) & (o,\top) & (x,\top_\bullet) & (y,?) & count_0 & from \rightharpoonup_{\mathsf{sub}} \\
\rightharpoonup(c,0_\bullet) & (n,?) & (o,\top) & (x,\top) & (y,?) & count_0 & from \rightharpoonup_{\mathsf{pre}} \\
\rightharpoonup(c,0) & (n,?) & (o,\top) & (x,-1_\bullet) & (y,?) & count_0 & from \rightharpoonup_{\mathsf{sub}} \\
\rightharpoonup(c,0) & (n,?) & (o,\top) & (x,-1) & (y,\top_\bullet) & count_0 & from \rightharpoonup_{\mathsf{when}} \\
\rightharpoonup(c,0) & (n,?) & (o,\top) & (x,-1) & (y,\mathit{tt}_\bullet) & count_0 & from \rightharpoonup_{\mathsf{when}} \\
\rightharpoonup(c,0) & (n,\top_\bullet) & (o,\top) & (x,-1) & (y,\mathit{tt}) & count_0 & from \rightharpoonup_{\mathsf{sync}} \\
\rightharpoonup(c,0) & (n,42_\bullet) & (o,\top) & (x,-1) & (y,\mathit{tt}) & count_0 & from\ input \\
\rightharpoonup(c,0) & (n,42) & (o,42_\bullet) & (x,-1) & (y,\mathit{tt}) & count_0 & from \rightharpoonup_{\mathsf{default}} \\
\rightharpoonup(c,0) & (n,42) & (o,42) & (x,-1) & (y,\mathit{tt}) & count_{42\bullet} & from \rightharpoonup_{\mathsf{pre}}
\end{array}$$

A. Linear clocks and array streams

Besides the rich interface properties refinement types capture, one most exciting aspect is their applicability to a large variety of models of computation and communication found in data-flow processing. It is, furthermore, at an easy reach, with the availability of Liquid Haskell [32] and, more importantly, of stream monads [19] as a medium to modularly specify these MoCCs.

Array processing One straightforward extension of our liquid clock system is linked to the one proposed in [27] for arrays and applies it to data-intensive/data-parallel functions found in signal processing applications. A data-parallel/data-intensive array processing MoCC can be characterised by a few array functions:

- map $f\,x$, to apply a function f to every tuple of elements at the same index in Array stream x;
- reduce $f\,x\,v$, to perform a reduction on the array stream x using the function f and initial value v, etc...
- scan $f\,x\,v$, to perform a prefix-scan on array stream x using function f and initial value v
- permute $i\,x$, to permute an array stream x using the bijective index function i
- gather $i\,x$, to gather an array stream x using the injective index function i
- filter $f\,x$, to sample array stream x with the filter function f

Liquid types for array-processing functions are proposed in [27]. We write $|x|$ for the length of an array stream x. Function map accepts a function f of type $x\!:\!s \to b$ and an array stream y of type vec s. Its output v is an array of bs of same clock and length as y. Function reduce accepts a function f of type $x\!:\!s \to b$ and an array stream y of type vec s. Its output v is of type b. Its logical clock is that of y.

$$\begin{aligned}
\mathsf{map}:\ & f\!:\!(x\!:\!s \to b) \to y\!:\!\mathsf{vec}\,s \to \langle v\!:\,\mathsf{vec}\,b \mid \hat{v} = \hat{y} \wedge |y| = |v|\rangle \\
\mathsf{reduce}:\ & f\!:\!(x\!:\!s \to b) \to y\!:\!\mathsf{vec}\,s \to \langle v\!:\!b \mid \hat{v} = \hat{y}\rangle \\
& \mathsf{etc}\ldots
\end{aligned}$$

Array scheduling The introduction of index/length array-based reasoning with the above type definitions opens to considering a variety of quantitative reasoning issues in data-flow processing, commencing with approximated (linear) real-time scheduling. For instance, the computation time of reduce $f\,y$ could be said to be that of Function f multiplied by the size of Array y.

In this aim, we have the possibility to extend our boolean clock model to interpret clocks as linear approximations of (possibly cyclo-static) data-flow processes [6]. In this aim, Function every accepts an input stream x and defines the clock of its output by the rate m and phase n of the input x. Instead of referring to clocks as symbolic, logical relations, it denotes by $\hat{v} = m * \hat{x} + n$ the period m and phase n relations between its input x and output v.

To accommodate the tradeoff between regular data-processing and linear communication rates, simple solutions are to define adapters between clock domains [11], to install FIFO buffers when possible [28], the most straightforward technique being to pack/unpack data at clock domain crossings. Function pack$[n\!:\!\mathsf{int}]$ takes an integer constant (stream) n and packs data from its input stream x into arrays of size n at the rate $\hat{v}$ such that $\hat{x} = n \times \hat{v}$. Function unpack does the opposite.

$$\begin{aligned}
\mathsf{every}:\ & m\!:\!\mathsf{int} \to \langle n\!:\!\mathsf{int} \mid 0 \leqslant n < m\rangle \to x\!:\!b \\
& \to \langle v\!:\!b \mid (\hat{v} = m * \hat{x} + n) \wedge (\hat{v} \Leftrightarrow v = x)\rangle \\
\mathsf{pack}[n\!:\!\mathsf{int}]:\ & x\!:\!s \to \langle v\!:\!\mathsf{vec}\,s \mid |v| = n \wedge \hat{x} = n \times \hat{v}\rangle \\
\mathsf{unpack}:\ & x\!:\!\mathsf{vec}\,s \to \langle v\!:\!s \mid \hat{v} = |x| \times \hat{x}\rangle
\end{aligned}$$

Time inference In the same manner as the above, it becomes also possible to address a variety of issues found in real-time calculi, such as for instance estimating the throughput of a given data-flow network, its processing time, its end-to-end latency... This can start from the collection of fine-grained clock relations that estimate one output stream's delivery time from the real-time clock of its computation per input clocks.

The time at which the output value of an addition plus $x\,y$ becomes available can be specified to be equal to the maximum between the delivery time, noted $\tilde{x}$, of its inputs x (and y) plus the δ_{plus} time of executing the addition.

$$\begin{aligned}
\mathsf{plus}:\ & x\!:\!\mathsf{int} \to y\!:\!\mathsf{int} \to \langle v\!:\!\mathsf{int} \mid \tilde{v} = \max(\tilde{x}, \tilde{y}) + \delta_{plus}\rangle \\
\mathsf{map}:\ & f\!:\!(x\!:\!s \to v\!:\!t) \to y\!:\!\mathsf{vec}\,s \\
& \to \Big\langle w\!:\!\mathsf{vec}\,t \mid \tilde{w} = \tilde{v} \times \lceil |y| \times \delta_f/\delta_{\mathsf{map}}\rceil \wedge |y| = |w|\Big\rangle \\
\mathsf{reduce}:\ & f\!:\!(x\!:\!s \to v\!:\!t) \to y\!:\!\mathsf{vec}\,s \\
& \to \langle w\!:\!t \mid \tilde{w} = \tilde{v} \times \lceil \delta_f \times |y|)/(log(\delta_{\mathsf{red}})) + 1)\rceil\rangle
\end{aligned}$$

This computation time aspect can be reflected in the type of array-processing functions as well, e.g. map $f\,x$, by dividing the amount of computation needed by the function f to process all the array elements of x

divided by the amount of parallelism δ_{map} provided by the implementation of the mapping protocol, with the same for reduce using pipelined execution.

$$\begin{aligned}\mathsf{map} :\ & f\!:\!(x\!:\!s \rightarrow v\!:\!t) \rightarrow y\!:\!\mathsf{vec}\, s \\ & \rightarrow\!\left\langle w\!:\!\mathsf{vec}\, t \,\middle|\, \tilde{w} = \tilde{v} \times \lceil |y| \times \delta_f / \delta_{\mathsf{map}} \rceil \wedge |y| = |w| \right\rangle \\ \mathsf{reduce} :\ & f\!:\!(x\!:\!s \rightarrow v\!:\!t) \rightarrow y\!:\!\mathsf{vec}\, s \\ & \rightarrow\!\left\langle w\!:\!t \,\middle|\, \tilde{w} = \tilde{v} \times \lceil \delta_f \times |y|)/(log(\delta_{\mathsf{red}})) + 1)\rceil \right\rangle\end{aligned}$$

Music synthesis Along the lines of the above, an exciting venue of future work considers the signal processing and synthesis language Faust [10] which allows to produce music from signal data-flow networks.

Faust is both a functional and block-diagrammatic specification language in which audio is produced from composing elementary sound processing blocks: signals of constant amplitude, e.g. `n`; `_`, the identity signal function; +, to sum up amplitudes; *, to scale them; `:` to compose functions; and ~ to create feedback. For instance, the tiny code snippet `*(1-n) : + ~ *(n)` defines a low-pass filter.

In similar manners as the liquid clock system presented in this paper, refinement types provide an algebraically rich framework to reason about the many quanta at play in synthesised audio: amplitude, rate, volume, throughput, timing and latency.

IX. Related Work

The framework of liquid clocks introduced in this paper relates to the theory of refinement types developed by the Liquid Haskell project of Jahla et al. [32]. The type system expresses quantitative properties on Boolean and integer values and indexes in the spirit of [27] and the inference system of [36]. Timed properties are represented by guarded proposition much like Pnueli's synchronous transition systems (STS) [26] which makes them amenable to SAT/SMT-verification related works [9], [23], here, by expressing them in QF-EUFLIA logic. Merging both approaches offers a powerful logical framework for both analysis, verification and code generation.

It allows us to revisit and improve several lines of earlier works concerned with clock calculi [2], [15], [24], scheduling analysis [6], [17], [25], [28], verification by abstract interpretation [5], [9] or just type-based analysis [4], [8], [18], [29], and yet extend and apply these to a context now far more general than synchronous data-flow, with the correct theoretical concepts to guarantee type soundness.

Most related data-flow synchronous languages [3], [8], [11], [12], [14], [18] now have well-developed and understood analysis and verification frameworks to help validate specification correctness and automatically generate code. While many different approaches have been considered to cast these analysis and verification techniques into the framework of type theory, none have been prominently successful in practice. An obvious choice seems to have leaned toward type polymorphism in early attempts [8], [29], albeit parametric polymorphism appears in practice inadequate to represent value dependencies: it makes type inference prone to variable capturewhich can only be circumvented by over-approximations and results in inefficiency. Another possible choice is to represent clocks using regular expressions [11], [18], which works well with sub-typing, but unfortunately builds types of exponential space complexity,just as polymorphic type inference, if applied to scheduling [4], [29]. One hence seeks drastic approximations (envelopes, counters, harmonics, adapters) or sophisticated constraint simplifications to reduce complexity.

Refinement types offer a valuable alternative to the above from several standpoints, both in terms of soundness guarantees, space complexity and abstraction capabilities. Sub-typing in refinement type systems defines a sound and effective means of abstraction, very much comparable, and actually implemented with, widening techniques as in abstract interpretation, and reduction techniques as in model checking. As Section 8 shows, if solely applied to static scheduling, liquid clocks allow to maintain a scheduling graph representation that provably remains in the size of its input-output interface, and definitely not in its number of internal operations.

Example For instance, abstracting the clock relations and scheduling graph of the stream function `countdown`, in Section VIII, is enforced by the sub-typing and well-formedness relations to be $\hat{n} \Rightarrow$

$n \to o \wedge \hat{o} \to o \wedge \hat{n} \leqslant \hat{o} \wedge 0 \leqslant o \leqslant n$. It is provably, and by design, limited to its interface streams.

```
let countdown (n) =
    let c = o pre 0
    |   o = n default x
    |   x = c - 1
    |   y = true when (c=0)
in synchro n y; o
```

$$\begin{array}{rcl} \hat{n} & \Rightarrow & n \to o \\ \hat{x} - \hat{n} & \Rightarrow & x \to o \\ \hat{c} & \Rightarrow & c \to x \\ \hat{c} & \Rightarrow & c \to \hat{y} \end{array}$$

X. Conclusions

Our preliminary theoretical study on so-called liquid clocks offers exciting evidence on the capability to capture quantitative properties of data-flow specifications, in a decidable, concise and sound manner, using refinement types. These promising results open to a variety of applications, from the integration of contract systems and modular specifications, to qualification by traceability of program properties from specification to generated code, to certified code generation using translation validation and type certificates, to correct-by-construction orchestrated music synthesis.

Liquid types allow to revisit many of the ad-hoc, problem-specific, algebraic frameworks and/or type theories that have been proposed to capture many variations of the Kahn principle using synchronous, periodic, multi-rate, affine, regular, integer, cyclo-static, continuous time models, all into one single, straightforward, verification framework. Liquid types open to considering a large variety of models of computation and communication, not only synchronous, polychronous, or asynchronous data-flow in the spirit of SDF, Simulink, Lustre, Signal, Kahn process networks, and multi-rate, cyclo-static, data-parallel, DDF MoCCs.

We are currently furthering our experiments with an evolving prototype that uses Liquid Haskell as language front-end and Z3 as SAT/SMT verifier, opening up to unprecedented expression capabilities.

Acknowledgments This work is partly funded by the ANR, project FEEVER, and by the USAF Office for Scientific Research, grant FA8655-13-1-3049.

The authors wish to thank Imré Frotier de la Messelière for constructive comments on a previous version of the paper.

References

[1] J. Aguado, M. Mendler and R. von Hanxleden and I. Fuhrmann. "Grounding Synchronous Deterministic Concurrency in Sequential Programming". European Symposium on Programming, LNCS v. 8410. Springer, 2014.

[2] P. Amagbegnon, L. Besnard, and P. Le Guernic. "Implementation of the data-flow synchronous language SIGNAL". Conference on Programming Language Design and Implementation. ACM, 1995.

[3] A. Benveniste, P. Le Guernic, C. Jacquemot. "Synchronous programming with events and relations: the SIGNAL language and its semantics". Science of Computer Programming. Elsevier, 1991.

[4] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet. "A Type-based Analysis of Causality Loops in Hybrid Systems Modelers". In International Conference on Hybrid Systems. ACM, 2014.

[5] F. Besson, T. Jensen, and J.-P. Talpin. "Polyhedral analysis for synchronous languages". Symposium on Static Analysis. Springer, 1999.

[6] A. Bouakaz and J.-P. Talpin. "Buffer minimization in earliest-deadline first scheduling of dataflow graphs". Conference on Languages, Compilers and Tools for Embedded Systems. ACM, 2013.

[7] A. Cohen, et al. "N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems." SIGPLAN Notices v.41. ACM, 2006.

[8] JL. Colaco, A. Girault, G. Hamon, and M. Pouzet. "Towards a Higher-order Synchronous Data-flow Language". International Conference on Embedded Software. ACM, 2004.

[9] P. Feautrier, A. Gamatie and L. Gonnord. "Enhancing the compilation of synchronous data-flow programs with a combined numerical-boolean abstraction'. Journal of Computing. Computer Society of India, 2012.

[10] Grame. "Faust, quick path from ideas to efficient DSP". http://faust.grame.fr.

[11] J. Forget, F. Boniol, D. Lesens and C. Pagetti. A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems. Symposium on Applied Computing. ACM, 2010.

[12] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. "The synchronous data flow programming language LUSTRE," Proc. of the IEEE, v.79, 1991.

[13] G. Kahn. "The semantics of a simple language for parallel programming". Proceedings of the IFIP Congress on Information Processing. IFIP, 1974.

[14] E.A. Lee, D.G. Messerschmitt. "Synchronous data flow." Proceedings of the IEEE v.75, 1987.

[15] Le Guernic, P., Talpin, J.-P., Le Lann, J.-C. "Polychrony for system design". Journal for Circuits, Systems and Computers. World Scientific, 2003.

[16] R. von Hanxleden, et al. "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications". Conference on Programming Language Design and Implementation. ACM, 2014.

[17] O. Maffeis, P. Le Guernic. "Distributed Implementation of Signal: Scheduling and Graph Clustering". Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS v. 863. Springer, 1994.

[18] L. Mandel, F. Plateau, and M. Pouzet. "Static Scheduling of Latency Insensitive Designs with Lucy-n". In International Conference on Formal Methods in Computer-Aided Design, 2011.

[19] Marlow, S., Newton, R., Peyton Jones, S. "A monad for deterministic parallelism". In ACM SIGPLAN Notices v. 46, n. 12. ACM, 2011.

[20] L. de Moura and N. Bjorner. "Z3: An efficient SMT solver". International conference on Tools and algorithms for the construction and analysis of systems. Springer, 2008. https://z3.codeplex.com

[21] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla. "Synthesising embedded software with safety wrappers through polyhedral analysis in a polychronous framework". Electronic System Level Synthesis Conference, 2012.

[22] G. Nelson. "Techniques for program verification". Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[23] C. Ngo, J.-P. Talpin, T. Gautier. "Efficient deadlock detection for polychronous data-flow specifications". Electronic System Level Synthesis Conference. IEEE, 2014.

[24] D. Potop-Butucaru, Y. Sorel, R. de Simone, JP. Talpin. "From concurrent multi-clock programs to deterministic asynchronous implementations". Fundamenta Informaticae. IOS Press, 2011.

[25] M. Pouzet, P. Raymond. "Modular Static Scheduling of Synchronous Data-flow Networks – An efficient symbolic representation". International Conference on Embedded Software. ACM, 2009.

[26] A. Pnueli, M. Siegel, E. Singerman. "Translation validation". International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 1998.

[27] P. M. Rondon, M. Kawaguchi, R. Jhala. "Liquid Types". Conference on Programming language Design and Implementation. ACM, 2008.

[28] I. Smarandache, T. Gautier and P. Le Guernic. "Validation of mixed Signal-Alpha real-time systems through an affine calculus on clock synchronisation constraints". World Congress on Formal Methods in the Development of Computing Systems. Springer, 1999.

[29] Talpin, J.-P., Nowak, D. "A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems". Foundations of Software Technology and Theoretical Computer Science. Springer, 1998.

[30] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. "Constructive Polychronous Systems". In Science of Computer Programming. Elsevier, 2014.

[31] J.-P. Talpin, P. Jouvelot, S. Shukla. "Liquid clocks: refinement types for time-dependent stream functions". Technical Report n. 0000. INRIA, June 2015. Available from http://hal.inria.fr and Arxiv http://arxiv.org/abs/0000.00000.

[32] The Liquid Haskell project. "LiquidHaskell, Refinement Types via SMT and Predicate Abstraction". http://goto.ucsd.edu/~rjhala/liquid.

[33] The SMT-Lib project. "SMT-Lib, the satisfaction-modulo theories library". http://smt-lib.org.

[34] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E.A. Lee. "Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs". Transactions on Embedded Computing Systems. ACM, 2015.

[35] N. Vazou, P. M. Rondon, R. Jhala. "Abstract refinement types". European conference on Programming Languages and Systems. Springer, 2013.

[36] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, S. Peyton-Jones. "Refinement Types For Haskell". SIGPLAN International Conference on Functional Programming. ACM, 2014.

Appendix

A. Proof of Theorem 1

The type soundeness property of Theorem 1 reduces to showing the invariance of E as a model of the updated registry in the calculus of base definitions, by factoring the type inference rules (T-*) with the rewriting rules of $\rightsquigarrow$.

$$\frac{E \vdash d\!:\!F \quad E \vdash d'\!:\!F'}{E \vdash d \,\|\, d'\!:\!F, F'} \qquad \text{(T-COM)}$$

$$\frac{E, (x\!:\!s) \vdash d\!:\!F, (x\!:\!s) \quad E \vdash s\checkmark \quad E \vdash F\checkmark}{E \vdash d/x\!:\!F} \qquad \text{(T-LOC)}, x \notin dom(E)$$

The type preservation theorem (Theorem 1) states that,
if $E \vdash e\!:\!s$, $F = E, (x\!:\!s)$, $x\!=\!e \rightsquigarrow d$, $r \models F$ and $r, d \rightharpoonup r', d'$
for $x \notin dom(E)$, then $r' \models F$ and $F \vdash d'\!:\!(x\!:\!s)$.

By application of Lemma 1, reduction preserves typing.
If $E \vdash e\!:\!s$, $F = E, x\!:\!s$ and $x\!=\!e \rightsquigarrow d$,
for $x \notin dom(E)$, then $F \vdash d\!:\!(x\!:\!s)$ and d is SSA.

Proving Theorem 1 amounts to an inductive proof by case analysis on the grammar of base definitions d in SSA form, as stated in Proposition 3, below. We say that E includes F, written $E \supseteq F$ iff $E(x) = F(x)$ for all $x \in dom(F)$.

Proposition 3:
If $E \vdash d\!:\!F$, $E \supseteq F$, $r \models E$ and $r, d \rightharpoonup r', d'$
then $r' \models E$ and $E \vdash d'\!:\!F$.

The proof of Proposition 3 is done by:

- structural induction on the grammar of base definitions d

$$d ::= x = y \star z \mid d \| d \mid d/x$$

- induction on the (monotonic) transition relation $\rightharpoonup$
- case analysis of all transitions $\rightharpoonup_\star$ of ground equations

Finally, since d is in static-single assignment form, we both guarantee (in structural cases) and use (in ground cases) the fact that each stream in F is defined by exactly one equation in d. These are labelled by a $(*)$.

Case d/x

By hypothesis, $E \supseteq F$ and

$$(1)\ E \vdash d/x\!:\!F \quad (2)\ r \models E \quad (3)\ r, d/x \rightharpoonup r', d'/x$$

Since (1), Rule (T-LOC) implies $x \notin dom(E)$ and

$$(4)\ E, (x\!:\!s) \vdash d\!:\!F, (x\!:\!s) \quad (5)\ E \vdash s\checkmark \quad (6)\ E \vdash F\checkmark$$

Since (3), Rule (LET) implies $x \notin dom(r)$ and

$$(7)\ r, (x, ?), d \rightharpoonup^* r', (x, v), d'$$

Moreover, since $x \notin dom(r)$, x is only defined in the scope of d

$$(*)\ x \notin dom(E) \wedge x \notin dom(r)$$

By Definition 2, for all $w \sqsupset ?$ such that $r, (x, w) \models E, (x\!:\!s)$

$$(8)\ r, (x, ?) \models E, (x\!:\!s)$$

From (4,7,8), by induction hypothesis on both the structure of d and the transition relation

$$(9)\ r', (x,v) \models E, (x\!:s) \quad (10)\ E, x\!:s \vdash d'\!:F, x\!:s$$

From (9), $r' \models E$.
From Rule (T-LOC) with (5,6,10), the conclusion: $E \vdash d'/x\!:F$.

CASE $d \,\|\, d'$
By hypothesis, $E \supseteq F, F'$ and

$$(1)\ E \vdash d \,\|\, d'\!:F, F' \quad (2)\ r \models E \quad (3)\ r, d \,\|\, d' \rightharpoonup r', d \,\|\, d''$$

From (1) with Rule (T-COM)

$$(4)\ E \vdash d\!:F \quad (5)\ E \vdash d'\!:F'$$

From (4,5), by definition of extension for F, F', $dom(F) \cap dom(F') = \varnothing$. Hence, every stream x in F, F' has exactly one definition in d or d'.

$$(*)\ im(d) \cap im(d') = \varnothing$$

Now, from (3), with Rule (PAR), there are two choices: $r, d' \rightharpoonup r', d''$ or $r, d \rightharpoonup r', d''$. We choose the former (the proof of the latter is identical). We get

$$(6)\ r, d' \rightharpoonup r', d''$$

Now, from (2,5,6) and by induction hypothesis

$$(7)\ r' \models E \quad (8)\ E \vdash d''\!:F$$

From Rule (T-COM) with (4,8), the conclusion: $E \vdash d \,\|\, d''\!:F, F'$.

CASE $x = y\,\mathsf{pre}\,v$
By hypothesis, $E \supseteq (x\!:s)$ and,

$$\begin{array}{l} (1)\ E \vdash x = y\,\mathsf{pre}\,v\!:(x\!:s) \\ (2)\ r \models E \\ (3)\ r, x = y\,\mathsf{pre}\,v \rightharpoonup r', x = y\,\mathsf{pre}\,w \end{array}$$

From (1) and by the guarantees $(*)$ from both previous structural cases, $y\,\mathsf{pre}\,v$ is the unique definition of x.
From (1), and Rule (T-DEF)

$$(4)\ E \vdash y\,\mathsf{pre}\,v\!:s$$

From Axiom (T-PRE), naming u_v the value identifier of v

$$(5)\ E \vdash \mathsf{pre}\!:(y\!:b\langle p\rangle) \rightarrow \langle u_v\!:b \mid p\rangle \rightarrow x\!:b\langle p \wedge q\rangle$$

where $s = \langle u\!:b \mid p \wedge q\rangle$ and $q = \hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{u}_v$.
By Rule (T-APP) with (4,5),

$$(6)\ E \vdash y\!:b\langle p\rangle \quad (7)\ E \vdash v\!:\langle u_v\!:b \mid p\rangle$$

From (7), by the Definition of constants and Rule (T-SUB)

$$(7')\ E \vdash \langle u_v\!:b \mid \hat{u}_v \Rightarrow u_v = v\rangle \leq \langle u_v\!:b \mid p\rangle$$

Now, from (3) and rule (pre),

$$(8)\ r(y), v, r(x) \rightharpoonup_{\mathsf{pre}} r'(y), w, r'(x)$$

From Table ($\rightharpoonup_{\mathsf{pre}}$), there are six cases to consider

1) $r(x,y) = (?, \bot)$, $r'(x,y) = (\bot, \bot)$, $d' = d$.

r is a partial model, since $r(x) = ?$. From (2) and by Definition 2, there exists a complete model r'' such that

$$(9)\ r \sqsubset r'' \quad im(r'') \subset \mathbb{V}^{\perp} \quad r'' \models E$$

From the hypothesis, $E \supseteq (x\!:\!s)$ and by definition of s,

$$(10)\ \models \llparenthesis E \rrparenthesis \Rightarrow \llparenthesis s \rrparenthesis \Rightarrow \llparenthesis \hat{x} \Leftrightarrow \hat{y} \rrparenthesis$$

Since $r(y) = \perp$, the only $r''(x)$ satisfying $\llparenthesis \hat{x} \Leftrightarrow \hat{y} \rrparenthesis$ is $\perp$. Hence,

$$(11)\ r'(x,y) = r''(x,y) = \perp$$

From (11), $r' \sqsubset r''$ and, by Definition 2, $r' \models E$.

2) $r(x,y) = (\perp, ?)$, $r'(x,y) = (\perp, \perp)$, $d' = d$. Same as case 1.
3) $r(x,y) = (?, \top)$, $r'(x,y) = (v, \top)$, $d' = d$. Again, the proof is similar as case 1: from (2) and Definition 2, there exists a complete model $r'' \sqsupset r$ s.t. $r'' \models E$. Since $r(y) = \top$ and $\llparenthesis E \rrparenthesis$ implies $\llparenthesis \hat{x} \Leftrightarrow \hat{y} \rrparenthesis$, the only choice is $r''(x) \sqsupset \top$. However, from $(*)$, no other equation defines x. Hence, the only choice is $r''(x) = r'(x) = v$. Hence, $r'' \sqsupseteq r'$ and, by Definition 2, $r' \models E$.
4) $r(x,y) = (\top, ?)$, $r'(x,y) = (v, \top)$, $d' = d$. Same as case 3.
5) $r(x,y) = (\top, \top)$, $r'(x,y) = (v, \top)$, $d' = d$. Same as case 3.
6) $r(x,y) = (v, w)$, $r'(x,y) = (v, w)$ and $d' \triangleq (x = y\, \mathsf{pre}\, w)$.
Obviously, $r' \models E$ from (2). Now, it remains to show that equation d' yields type $(x\!:\!s)$. This reduces to showing that $E \vdash w\!:\!b\langle p\rangle$ as follows to (15).
By the definition of w as a constant stream:

$$(12)\ E \vdash w\!:\!\langle u_w\!:\!b \mid \hat{u}_w \Rightarrow u_w = v\rangle$$

Now, from (6), since $r'(y) = w$ and E is a model of r', by Definition 2,

$$(13)\ r \models \llparenthesis p[y/w] \rrparenthesis$$

From (13) and Rule (S-BAS),

$$(14)\ \frac{\llparenthesis E, u_w\!:\!b \rrparenthesis \models \llparenthesis \hat{u}_w \Rightarrow u_w = v \rrparenthesis \Rightarrow \llparenthesis p[y/u_w] \rrparenthesis}{E \vdash \langle u_w\!:\!b \mid \hat{u}_w \Rightarrow u_w = v\rangle \preceq \langle u_w\!:\!b \mid p[y/u_w]\rangle}$$

Hence, from (7') and by Rule (T-SUB),

$$(15)\ E \vdash w\!:\!b\langle p\rangle$$

From (5,6,15) and by application of Rule (T-APP)

$$E \vdash y\, \mathsf{pre}\, w : s$$

Hence, by rule (T-DEF), the conclusion: $E \vdash d' : (x\!:\!s)$

Case $x = y \star z$

The base cases of all combinatorial equations $d = x = y \star z$ with $\star \neq \mathsf{pre}$ are done in a similar manner as for pre by inspection of the transition rules $\rightharpoonup_{\star}$ except that there is no state transition, and hence $d = d'$.

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399