PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Sémantique Mécanisée et Compilation Vérifiée pour un Langage Synchrone à Flots de Données avec Réinitialisation

Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset

Soutenue par
Lélio BRUN
Le 6 juillet 2020

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Gerwin KLEIN DATA61 / CSIRO / UNSW	*Rapporteur*
Xavier THIRIOUX ISAE-SUPAERO	*Rapporteur*
Jean-Louis COLAÇO ANSYS	*Examinateur*
Emmanuel LEDINOT THALES Research & Technology	*Examinateur*
Xavier LEROY Collège de France	*Président*
Pascal RAYMOND UGA / CNRS / VERIMAG	*Examinateur*
Timothy BOURKE Inria / ENS	*Co-Directeur*
Marc POUZET Sorbonne Université / ENS / Inria	*Co-Directeur*

ENS | PSL

To my friend Kevin Kenmogne
November 20, 1990 – September 5, 2014

Special thanks (excuse my French)

I am deeply grateful to my advisors Tim and Marc, for basically everything during these four years. I also want to thank all the members of my committee: Gerwin Klein, Xavier Thirioux, Jean-Louis Colaço, Emmanuel Ledinot, Xavier Leroy and Pascal Raymond, for having accepted to take the (considerable) time to read and evaluate this dissertation.

En Français maintenant : merci à Tim et Marc pour leur confiance, pour m'avoir intégré à PARKAS dès mon stage de recherche, pour s'être battus pour m'obtenir une bourse de doctorat, pour avoir toujours été présents, pour m'avoir encouragé quand la motivation me faisait (régulièrement) défaut. Merci aussi de m'avoir trouvé les financements supplémentaires qui m'ont permis de prolonger mon doctorat sans avoir à trop stresser (et merci au centre Inria de Paris au passage pour ces financements). J'ai l'intuition de leur avoir fait un peu peur de temps en temps, ma façon de travailler pouvant sembler quelque peu irrégulière par moments... J'espère que j'ai pu montrer suffisamment d'*engagement* quand il le fallait pour les rassurer, et pas uniquement sur des skis ou sur un terrain de rugby. J'ai une conscience aiguë de la chance de les avoir eus pour directeurs, et je ne mesure certainement pas encore l'étendue de ce qu'ils m'ont apporté. En plus de leur expertise scientifique évidente, je citerai deux traits que j'ai pu développer à leur contact : l'esprit critique vis-à-vis du métier de chercheur et du monde de la recherche en général, et la rigueur scientifique.

Merci à tous les membres de l'équipe PARKAS que j'ai pu côtoyer durant ces quatre années, pour la super ambiance qui y règne et pour les discussions, scientifiques ou moins : les permanents Albert et Francesco, les thésards Guillaume B., Ulysse, Ismail et Basile, l'ancien Adrien, et tous les autres, Guillaume I., Andy, Paul, Baptiste, Marc, et j'en oublie. Merci aussi aux anciens thésards d'autres labos, comme Pierre, à ceux du LIP6, en particulier à Rémy, avec qui j'ai développé une belle amitié depuis mon premier SYNCHRON en 2016.

Merci aux nombreux chercheurs qui m'ont donné envie de faire de la recherche. En particulier, merci à Yann Régis-Gianas, dont le cours de compilation en M1 restera dans mes souvenirs comme l'un des meilleurs cours que j'ai eu la chance de pouvoir suivre. Merci aussi à lui de m'avoir fait poser mon premier pas dans le monde de la recherche, et de m'avoir soutenu et aiguillé dans mes objectifs pédagogiques. Un grand merci également à Jean-Christophe Filliâtre, pour m'avoir fait confiance en me laissant assurer les TP de son génial cours de compilation trois années durant. Merci pour son soutien et ses encouragements tout au long de ma thèse. C'est à l'enthousiasme de Yann et Jean-Christophe pour la pédagogie et la transmission que je dois mon envie de poursuivre dans l'enseignement. Merci aussi à Xavier Leroy, pour son exploit avec CompCert qui est une inspiration pour ce travail bien sûr, mais aussi pour ses conseils et surtout pour

m'avoir orienté vers le stage de recherche que proposaient Tim et Marc ! Merci à Gérard Berry pour la leçon de rigueur et d'humilité qu'il m'a donnée, à SYNCHRON en 2017.

Merci à mes amis, mes gars sûrs, les frères Maxime, Stéphane, Fabien et Quentin. Pour leur détachement qui m'a souvent fait garder les pieds sur terre et me rappeler que la vie ce n'est pas que la coinduction et l'axiome du choix. Maxime, j'ai de la chance d'avoir quelqu'un comme toi dans ma vie. Merci à Tom, la belette, le chacal, pour son amitié infaillible depuis bien 25 ans. Merci à Yann, sans le coup de fil duquel rien ne serait arrivé. Merci à Kévin, tu es parti mais je me souviens. Merci à Gaëlle, qui m'a soutenu inconditionnellement aux premiers instants de ma réorientation. Merci Marie, pour ton amour et ton soutien, pour m'avoir ouvert des voies qui demeuraient obstruées, pour ton regard et ton intellect, pour être ce que tu es, pour tout.

Enfin, merci à ma famille pour leur soutien sans faille. Merci à mes parents Frédéric et Christine de m'avoir permis de mener cette réorientation jusqu'au bout de ce qui semblait probablement être un coup de tête ou de poker. Merci à mon frère Christophe pour ses encouragements et sa confiance débordante. Enfin merci à ma sœur Solène, pour beaucoup de choses. Son amour, et son soutien, même quand je pense ne pas les mériter, la complicité que l'on a fini par réussir à développer au fil des ans (c'était pas gagné), les discussions passionnantes qui m'ont fait de nombreuses fois réviser mon jugement sur le monde. Et bien sûr pour l'inspiration : voir sa petite sœur soutenir sa thèse avant soi, ça motive (et ça met la pression grave) !

Résumé

Les spécifications basées sur les schémas-blocs et machines à états sont utilisées pour la conception de systèmes de contrôle-commande, particulièrement dans le développement d'applications critiques. Des outils tels que Scade et Simulink/Stateflow sont équipés de compilateurs qui traduisent de telles spécifications en code exécutable. Ils proposent des langages de programmation permettant de composer des fonctions sur des flots, tel que l'illustre le langage synchrone à flots de données Lustre.

Cette thèse présente Vélus, un compilateur Lustre vérifié dans l'assistant de preuves interactif Coq. Nous développons des modèles sémantiques pour les langages de la chaîne de compilation, et utilisons le compilateur C vérifié CompCert pour générer du code exécutable et donner une preuve de correction de bout en bout. Le défi principal est de montrer la préservation de la sémantique entre le paradigme flots de données et le paradigme impératif, et de raisonner sur la représentation bas niveau de l'état d'un programme.

En particulier, nous traitons le reset modulaire, une primitive pour réinitialiser des sous-systèmes. Ceci implique la mise en place de modèles sémantiques adéquats, d'algorithmes de compilation et des preuves de correction correspondantes. Nous présentons un nouveau langage intermédiaire dans le schéma habituel de compilation modulaire dirigé par les horloges de Lustre. Ceci débouche sur l'implémentation de passes de compilation permettant de générer un meilleur code séquentiel, et facilite le raisonnement sur la correction des transformations successives du reset modulaire.

Mots clés : langages synchrones à flots de données, Lustre, Scade, compilation vérifiée, sémantique mécanisée, Vélus, assistants de preuve interactifs, Coq, reset modulaire

Abstract

Specifications based on block diagrams and state machines are used to design control software, especially in the certified development of safety-critical applications. Tools like SCADE and Simulink/Stateflow are equipped with compilers that translate such specifications into executable code. They provide programming languages for composing functions over streams as typified by dataflow synchronous languages like Lustre.

In this thesis we present Vélus, a Lustre compiler verified in the interactive theorem prover Coq. We develop semantic models for the various languages in the compilation chain, and build on the verified CompCert C compiler to generate executable code and give an end-to-end correctness proof. The main challenge is to show semantic preservation between the dataflow paradigm and the imperative paradigm, and to reason about byte-level representations of program states.

We treat, in particular, the modular reset construct, a primitive for resetting subsystems. This necessitates the design of suitable semantic models, compilation algorithms and corresponding correctness proofs. We introduce a novel intermediate language into the usual clock-directed modular compilation scheme of Lustre. This permits the implementation of compilation passes that generate better sequential code, and facilitates reasoning about the correctness of the successive transformations of the modular reset construct.

Keywords: synchronous dataflow languages, Lustre, Scade, verified compilation, mechanized semantics, Vélus, interactive theorem provers, Coq, modular reset

Contents

Figures, Tables and Listings

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Vélus is a verified prototype compiler, for a synchronous dataflow language with constructs from Lustre [Caspi, Pilaud, et al. (1987)] and Scade [Colaço, Pagano, and Pouzet (2017)]. The main goal in the development of Vélus is to provide an end-to-end proof of correctness in an Interactive Theorem Prover (ITP) from the synchronous dataflow paradigm to the imperative one. This involves specifying the syntax, type systems and clock systems of the source and several intermediate languages, mechanizing their semantic models, implementing the compilation passes that successively transform programs from one language to another, and finding invariants and proofs to establish the corresponding correctness results. We use the Coq ITP [The Coq Development Team (2019)] and extend the proof of correctness to the machine level by building on top of the verified CompCert C compiler [Leroy (2009b)].

This thesis presents, in particular, novel extensions to the original Vélus compiler to incorporate the *modular reset* construct [Hamon and Pouzet (2000)]. We build on earlier work to formalize the semantics of this construct in Coq. We introduce a novel intermediate language to compile it effectively and show how to adapt existing invariants and proofs to re-establish the end-to-end correctness result.

1.1 Synchronous dataflow languages

1.1.1 Background

Synchronous programming is a paradigm that was developed in the 1980s concomitantly with the growing importance of real-time embedded software. Such systems run in interaction with the environment, are subject to strict non functional constraints like bounded time and memory resources and are often safety-critical. They belong to the class of *reactive systems*, as named by Harel and Pnueli (1985), that defines systems that "continuously react to their environment at a speed determined by this environment" [Halbwachs (1993)]. In contrast with the standard approach based on asynchronous execution and interleaving, synchronous languages build on two key concepts: *synchrony* and *deterministic concurrency*.

The synchrony hypothesis assumes that synchronous programs "produce their outputs *synchronously* with their inputs, their reaction taking no observable time" [Benveniste and Berry (1991)]. This simplifying premise decouples the specification of programs logic from the need to ensure that implementations meet real-time constraints. It permits

```
node euler(x0, u: float64) returns (x: float64);
let
  x = x0 fby (x + 0.1 * u);
tel

node ins(gps, xv: float64) returns (x: float64; alarm: bool);
  var k: int;
let
  x = merge alarm ((0. fby x) when alarm)
                  (euler((gps, xv) when not alarm));
  alarm = (k >= 50);
  k = 0 fby (k + 1);
tel
```

Lustre

Listing 1.1: A simple inertial navigation system

reasoning in a discrete model of time where events are totally ordered relative to one another [Benveniste, Caspi, et al. (2003)]. In this model, concurrency, that is, parallel composition, is simply the conjunction of synchronous sub-systems.

The implementation of these fundamental concepts gave independently birth to three early languages: Esterel [Boussinot and de Simone (1991); Berry and Gonthier (1992); Berry (2000a,b, 2002)], imperative, event-driven and control-oriented, Signal [Benveniste and Le Guernic (1990); Benveniste, Le Guernic, and Jacquemot (1991); Le Guernic et al. (1991)], declarative and geared towards the specification of whole systems, and Lustre [Caspi, Pilaud, et al. (1987); Halbwachs, Caspi, et al. (1991); Halbwachs (2005); Jahier, Raymond, and Halbwachs (2019)], declarative, time-driven and dataflow-oriented. Besides adopting the synchronous model, all share two major traits: (1) they have a mathematical model, and (2) their implementation must satisfy strong memory and time constraints. The combination of these two aspects results from the twofold origin of the synchronous languages, inspired both by control theory and computer science, and from their targeted application domain. New ideas were introduced in later languages such as Scade [Colaço, Pagano, and Pouzet (2017)], Argos [Maraninchi (1991); Maraninchi and Rémond (2001)], Reactive C [Boussinot (1991)], Lucid Synchrone [Pouzet (2006)], ReactiveML [Mandel and Pouzet (2005); Mandel, Pasteur, and Pouzet (2015)], Zelus [Bourke and Pouzet (2013); Bourke, Colaço, et al. (2015)] or SCCharts [von Hanxleden et al. (2014)].

1.1.2 Lustre

Lustre was inspired by the dataflow programming language Lucid [Wadge and Ashcroft (1985)] and by Kahn process networks [Kahn (1974)]. A Lustre program operates on streams, that is, infinite sequences of values. The basic unit of a Lustre program is called a *node*. A node defines a function of streams via a set of *equations*. Equations equate variables to expressions, and are to be understood as "temporal invariants" [Halbwachs

(2005)], in the sense that the equation $x = e$ is interpreted as the formula $\forall n,\ x_n = e_n$. Throughout this dissertation, we will refer to the example Lustre program in listing 1.1. The `euler` node receives two input streams `x0` and `u`, an initial quantity and a derivative, respectively, and outputs stream `x`, representing an approximate quantity defined by a single equation that implements the forward Euler scheme $x_{n+1} = x_n + 0.1u_n$. Arithmetic operators operate point-wise on streams: `x + y` defines the stream $x_0 + y_0 \cdot x_1 + y_1 \cdots$. The **`fby`** operator is an *initialized delay* operator: `x fby y` defines the stream $x_0 \cdot y_0 \cdot y_1 \cdots$. A **`fby`** whose first argument is a constant represents a *unit delay* in control theory, or a *register* in synchronous circuits.[1]

Once a node is declared, it can be instantiated in other nodes. In the example, the `ins` node composes an instance of `euler`, initialized from a `gps` input and fed a stream of displacement values `xv`, with a local counter `k` used to signal an `alarm` when the resulting odometric approximation is judged outdated. The order of the equations is inconsequential. The **`when`** operator samples a stream: `x when y` defines the sub-stream of `x` determined by whether `y` is true or not. In the example, the arguments of the `euler` instance are sampled when `alarm` is false, consequently the instance is not activated when the `alarm` condition occurs. The `x` output is defined using a **`merge`** operator that combines complementary streams into a *faster* one.[2] The effect here is to freeze the value of `x` when `alarm` occurs. With sampling comes the notion of *clock*. A clock is an abstract representation of a timescale, that is, a stream of booleans indicating when the values of another stream are *present* or *absent*.

The semantics of a node can be represented as a *chronogram*, that is, a grid associating each variable with a row representing a stream of values. Below is an example of such a chronogram for the `ins` node, where holes represent absence of value.

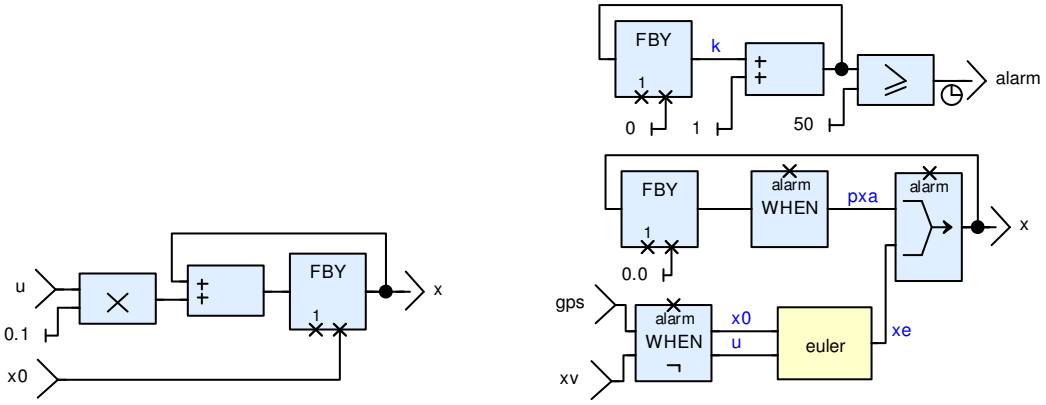
`gps`	5.	5.	5.	···	5.	5.	5.	5.	···
`xv`	20.	30.	10.	···	60.	60.	50.	70.	···
`k`	0	1	2	···	49	50	51	52	···
`alarm`	F	F	F	···	F	T	T	T	···
`0. fby x`	0.	7.	10.	···	36.	42.	42.	42.	···
`(0. fby x) when alarm`				···		42.	42.	42.	···
`euler(gps, xv)`	7.	10.	11.	···	42.	48.	53.	60.	···
`euler(gps, xv) when not alarm`	7.	10.	11.	···	42.				···
`x`	7.	10.	11.	···	42.	42.	42.	42.	···

The Lustre language finds an industrial application in the SCADE[3] tool-set used for developing safety-critical embedded software. SCADE evolved from the graphical syntax that it initially provided over a Lustre kernel towards a high-level language, a

[1] In this dissertation, we do not treat the *uninitialized delay* operation **`pre`** `x`, that defines the stream $nil \cdot x_0 \cdot x_1 \cdots$, where *nil* denotes an undefined value, nor the *initialization* operation `x -> y`, that defines the stream $x_0 \cdot y_1 \cdot y_2 \cdots$. Indeed, `x fby y` defines the same stream as `x -> pre y`, with the advantage of avoiding the complications related to possibly undefined values.

[2] We do not treat **`current`** `x`, that *oversamples* `x`, because it can introduce uninitialized values.

[3] www.ansys.com/products/embedded-software/ansys-scade-suite



(a) Forward Euler node **(b)** Instantiation of the Euler node

Figure 1.1: SCADE graphical representation

programming environment and a qualified code generator known as KCG. Released in 2008, the major version SCADE 6 enrich its kernel with more complex features, inspired by concepts from Esterel and Lucid Synchrone, among others. SCADE provides two syntaxes: a Lustre-like syntax, the Scade language [Colaço, Pagano, and Pouzet (2017)] and a graphical syntax [Dormoy (2008)]. Graphical representation is pervasive in synchronous languages and in model-based development in general. In particular, the SCADE graphical syntax can be compared to that of Simulink[4] which is the de facto standard tool for non-critical model-based development. The SCADE graphical counterpart of the `euler` node in listing 1.1 is shown in figure 1.1a: each block represents an operator, and each wire represents a stream. There is a multiplication block, an addition block and a FBY block. On the graphical view of the `ins` node in figure 1.1b, there is again a block for each operator, plus a block instantiating the `euler` node.

1.1.3 A brief history of the compilation of Lustre

The generation of sequential code from Lustre code has been thoroughly studied from the beginnings of the language. The general idea is to generate a transition function that is repeatedly executed to compute the successive synchronous steps of the system. There are two main approaches to generate such a function.

1. **Single-loop compilation** is the simplest technique. The idea is to sequentialize the calculations of the variables of the program, in an infinite loop that alternates reads of inputs, calculation of a step of the system, and writes to outputs. Control is realized by tests.

2. **Finite automaton generation** is based on the idea of reducing the number of conditional structures in the generated code. It works by identifying a subset of state

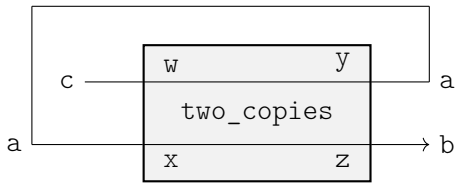
[4] www.mathworks.com/products/simulink.html

variables from which a finite state automaton can be constructed, whose states and transitions encode control, avoiding unnecessary tests.

An important orthogonal question is whether and when to inline node instantiations. This is not just a question of executable performance. It determines, in fact, which source programs can be compiled. Consider the following example, due to Gonthier (1988).

```
node two_copies(w, x: int)
  returns (y, z: int)
let
  y = w;
  z = x;
tel
```

Lustre



The `two_copies` node simply transmits its two inputs and can be sequentialized either as the sequence of assignments `y := w; z := x` or as `z := x; y := w`. This node cannot be modularly compiled into a single function because in an instantiation like `(a, b) = two_copies(c, a)`, shown graphically above on the right, the first output must be calculated before it can be provided as the second input, forbidding the second choice of sequentialization. It means that a static scheduling cannot be found for a node independently of the context of its instantiation.

In his thesis, Plaice (1988) presents the compiler LUSTRE-V2, which generates a finite automaton from a Lustre program. The compiler builds on the approach of [Caspi, Pilaud, et al. (1987)] which, in turn, was inspired by the compilation of Esterel. Plaice remarks that generating a finite automaton produces more efficient code than generating a sequential transition function per node, in particular because the latter generates unnecessary tests at each cycle to distinguish the initial state. He describes a *data-driven* algorithm that generates the automaton *forward* by firstly inlining all node instantiations and enumerating a set of boolean variables representing the state vector of the program. To avoid an explosion in the size of the automaton, minimisation techniques are applied afterward, but in some cases the minimal automaton may be excessively large compared to the source program. Moreover, it may be infeasible to generate the initial automaton which can be very large.

The problem of combinatorial explosion in the size of the generated automaton is studied by Raymond (1991) in his thesis. He proposes a new version of the compiler, LUSTRE-V3, which directly produces a minimal automaton, with new heuristics to choose the set of enumerated control variables based on binary decision diagrams. The approach defines a *demand-driven* algorithm to generate a minimal automaton that works *backward* from the outputs of the program. A comparison of execution time and code size between single-loop compilation, data-driven and demand-driven approaches appears in [Halbwachs, Raymond, and Ratel (1991)]. Raymond notes that inlining all node instantiations is the first source of combinatorial explosion and advocates for separate compilation. Indeed, in prior work [Raymond (1988)], he presents a method to rewrite a Lustre program into a network of finite automata, following a static analysis that avoids the problem exemplified by Gonthier's `two_copies` program.

Right from the beginnings of SCADE in the mid-nineties, it appeared that modular compilation was mandatory, despite its relative inefficiency. As it is an industrial tool that targets critical embedded systems, it cannot rely on the generation of finite automata because of the loss of traceability between source programs and generated code, and because of the complexity of the generation process. The choice in the SCADE code generator was thus to generate a single main loop, and a *step* function per node, even if that meant proscribing causal loops like the one in Gonthier's example. In Scade, each feedback loop must cross an explicit delay, but inlining can be requested explicitly by the programmer. Starting from 1999, the prototype compiler ReLuC of Jean-Louis Colaço developed as a reference compiler for future versions of Scade implements a simpler and more efficient compilation method than earlier versions of Scade. This method is formalized in [Biernacki et al. (2008)] and relies on source-to-source transformations and compilation toward a minimal kernel on which several static analysis are defined (initialization, causality, clocking and typing). The code generator for the latest version of SCADE, SCADE 6, follows these principles.

Rather than generate a single *step* function per node, an alternative modular approach is to decompose a node into sets of *step* functions with associated scheduling constraints. This avoids the restrictions that Scade imposes on feedback loops. Lublinerman, Szegedy, and Tripakis (2009) formalize this problem as optimal graph clustering and show that it is NP-complete using a reduction of the *clique cover* problem. They propose a SAT-solver-based iterative algorithm to found the optimal number of functions and the scheduling constraints. Several trade-offs between modularity (number of interface functions generated), reusability (usability in different contexts), and code size are studied. Pouzet and Raymond (2009) give an equivalent formalization of the problem as an optimal static scheduling problem. They propose a solution that simplifies the problem over an input / output dependency relation inspired from [Raymond (1988)]. They define a polynomial algorithm that is able to find optimal solutions in most cases and otherwise give a lower start bound to use iterative SAT-solver-based solutions like the one of [Lublinerman, Szegedy, and Tripakis (2009)].

In this work, we focus on the single-loop modular compilation scheme described in [Biernacki et al. (2008)]. This is the simplest scheme, and the basis of the current Scade code generator. Technically, it is also well suited to induction-based proof techniques.

1.2 Verified compilation

Compilers are complex programs and, as such, particularly prone to bugs. Compiler correctness is an old topic that dates back to the first published pen-and-paper proof of McCarthy and Painter (1967). Dave (2003) surveys about one hundred articles about compiler verification. Since then, some promising achievements have been made: a byte-code compiler from a subset of Java to a subset of the Java Virtual Machine [Klein and Nipkow (2006)] verified in the Isabelle ITP, the Verisoft project [Leinenbach, Paul, and Petrova (2005)] and its compiler for a subset of C verified in Isabelle, the C compiler CompCert [Blazy, Dargaye, and Leroy (2006); Leroy (2006, 2009b)], verified in Coq,

Pilsner [Neis et al. (2015)], a compositional compiler for an ML-like language verified in Coq, the CakeML project [Kumar et al. (2014); Tan et al. (2016)] and its compiler for a subset of Standard ML verified in HOL4. The different techniques used in these various endeavours can be summarized in the three following categories, that may be combined.

1. **Verified compilers** are proved correct directly: if compilation succeeds, the correctness proof provides a formal guarantee that the semantics is preserved.

2. **Verified validators,** following the *translation validation* approach [Pnueli, Siegel, and Singerman (1998)], check after the compilation that the semantics of the source and generated programs coincide.

3. **Certifying compilers,** following the *proof-carrying code* approach [Necula (1997)], generate both compiled code and a proof, or *certificate*, checked by an independent verified checker, that the code satisfies a given specification.

Our main challenge is to pass from a dataflow semantics to an imperative one. This distinguishes our work from other verified compilers that either treat imperative languages like C (CompCert) or that normally require a managed runtime like ML (CakeML). In the following, we present some existing works on the formalization and verified compilation for synchronous languages, then describe our approach.

1.2.1 Existing approaches for synchronous languages

Several synchronous languages have been formalized in ITPs, including synchronous circuits in Coq [Paulin-Mohring (1996); Coupet-Grimal and Jakubiec (1999)], a variant of Esterel in HOL [Schneider (2001)], a shallow embedding of Lucid Synchrone in Coq [Boulmé and Hamon (2001)] and Kahn networks in Coq [Paulin-Mohring (2009)]. There are many partial formalizations of Simulink and its Statecharts-like graphical formalism Stateflow. Hamon and Rushby (2004) and Hamon (2005) formalize the semantics of Stateflow and show that it is essentially an imperative language with a graphical syntax. Several other works propose formalizations, but that are oriented towards formal verification of Simulink/Stateflow continuous-time models. None of these works treats code generation.

There are several proposals around verified compilation for synchronous languages. An unpublished report about the development of a Scade 3 compiler verified in Coq focuses on semantics and clocking [Giménez and Ledinot (2000)]. The GeneAuto project aimed at developing a qualified code generator from a subset of Simulink and showed the correctness of the scheduling of dataflow equations in Coq [Toom, Näks, et al. (2008); Toom, Izerrouken, et al. (2010); Izerrouken (2011)]. In his PhD thesis, Auger (2013) describes the development of a Lustre compiler with reset partially verified in Coq, based on a semantic model for finite lists rather than for streams. Recent work introduces a compilation pass from Signal to an intermediate representation verified in Coq [Yang et al. (2016)]. There is ongoing unpublished work by Gérard Berry and Lionel Rieg to verify the translation of Esterel to digital circuits in Coq. None of these works gives an end-to-end proof of correctness for the generation of executable code. Several passes of

the Lustre compiler of Shi, Gan, et al. (2017) and Shi, Zhang, et al. (2019) have been verified in Coq. Their dataflow semantics is defined sequentially in an imperative fashion, which makes reasoning on dataflow transformations like scheduling difficult. They do not support the reset operator. Finally, their proof states the correctness of the execution only for a finite number of steps.

Translation validation is another approach that was applied to synchronous languages, firstly by Pnueli, Strichman, and Siegel (1998). Starting from the initial proposal of [Pnueli, Strichman, and Siegel (1999)], recent work integrate this technique into Signal compilers. It has also been applied to a subset of Simulink [Ryabtsev and Strichman (2009)]. Translation validation can provide formal guarantees as strong as verified compilation if the validators are formally verified, but this is not the case for recent work on Signal.

1.2.2 Our approach

For safety-critical software, the most dreaded compiler bugs are those that are silently introduced by the compiler into code generated from a correct source. Studies [Eide and Regehr (2008); Yang et al. (2011); Le, Afshari, and Su (2014)] have shown various crashes or miscompilations in C compilers. Yet, the motivations for formalizing C and verifying a C compiler do not transfer automatically to languages like Lustre and Scade which benefit from a mathematical model that C lacks. Moreover, the code generator that SCADE embeds, KCG, is *qualified* under the avionic norms DO-178B (1992) and its replacement DO-178C (2012), among others. Qualification is a rigorous process that gives strong correctness guarantees in practice. Qualification and formal correctness are distinct topics, as Halbwachs (2005) writes:

> Let's say at once that such a qualification has nothing to do with formal proof of the compiler, but is rather a matter of design process, test coverage, quality of the documentation, requirements traceability, etc.

Thus our goal is not to propose a replacement for Scade but rather to experiment with mechanized verification. We consider this approach as a complement to qualification, that raises interesting scientific questions but whose efficiency and industrial applicability remains uncertain.

What is a correct compiler? The standard notion of compiler correctness asserts that a compiler is correct if any property that holds on the source program also holds on the compiled program. This property, called *refinement* [Abadi and Lamport (1988)], is usually stated in terms of semantics preservation: observable behaviors of the compiled program must also be observable behaviors of the source program. Leroy (2009b) gives a well summarized overview of the notions of semantic preservation and associated proof techniques.

The observable behavior of a Lustre program S is the list of streams $\boldsymbol{ys}$ obtained by feeding the main node of S with a given list of input streams $\boldsymbol{xs}$. The generated program runs in an infinite loop alternating the consumption of inputs and the production of outputs. Such operations are observable through an infinite sequence of *events*, that is, a

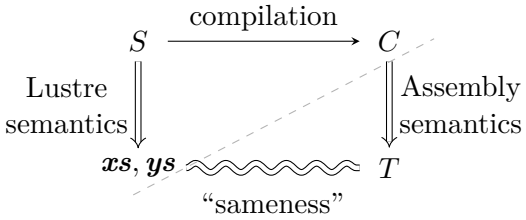


Figure 1.2: The overall schema of the correctness result

trace T recording the values read or written. We state the correctness of our compiler as a *simulation* [Lynch and Vaandrager (1994)] from the source semantics to the target semantics[5]. The correctness proof consists in establishing (1) that such a trace T exists and (2) that it represents the *same* behavior as the pair of lists of streams $(\boldsymbol{xs}, \boldsymbol{ys})$. This is described schematically in figure 1.2 in the form of a simulation diagram. The hypotheses are on the left of the dashed line, while the proof obligations are on the right.

Remark that this simulation result is the reverse of *refinement*: it states that observable behaviors of the source program are also observable behaviors of the compiled program. Nonetheless, the two directions are equivalent provided that the target semantics is deterministic, which is the case here, and this direction is easier to prove than the converse.

1.3 The modular reset

The ability to reinitialize the *state* of a node is important for modular programming. Consider the example in listing 1.2a that implements a general counter. Listing 1.2b, inspired from [Caspi, Pilaud, et al. (1987)], is a rewriting of the `counter` node with an additional input stream `reset` that indicates when the counter must restart. This node can then be instantiated, for example, by the equation `n = counter(0, 1, r)`. Unfortunately, this non-modular rewriting approach generates poor imperative code. A modular approach was expressed by Caspi (1994, §4.1) with block diagrams. Hamon and Pouzet (2000) introduced a *modular reset* construct in Lucid Synchrone, that was later adapted for SCADE. In Scade, one writes `n = (restart counter every r)(0, 1)` in a modular way, using the `counter` version of listing 1.2a. The syntax expresses that the `counter` instance is reset every time `r` is true. A similar feature, called *Resettable Subsystem*,[6] exists in Simulink.

In control system design, sophisticated applications are often best expressed using

[5] There is an unfortunate diversity in the terminology used for qualifying such simulation results. For example, in the verified micro-kernel sel4 [Klein, Elphinstone, et al. (2009)], the term *forward simulation* is used to mean a simulation from a concrete model to an abstract model, while in CompCert [Leroy (2009a)], it is used to mean a simulation from source (abstract) semantics to target (concrete) semantics. Despite this difference in terminology, both works refer to the same notion of *refinement*. In this dissertation we will only use the unqualified term of simulation, specifying the direction to avoid the confusion.

[6] mathworks.com/help/simulink/ug/reset-block-states-in-a-subsystem.html

```
node counter(init, incr: int)
  returns (n: int);
let
  n = init fby (n + incr);
tel
```

Lustre

(a) Without reset

```
node counter(init, incr: int; reset: bool)
  returns (n: int);
let
  n = if reset then init else init fby (n + incr);
tel
```

Lustre

(b) With reset

Listing 1.2: The `counter` node

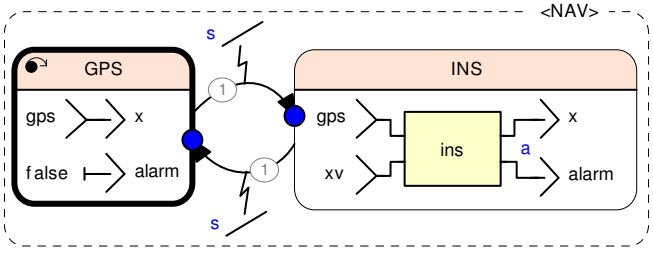


Figure 1.3: A SCADE graphical representation of a state machine

```
node nav(gps, xv: float64; s: bool) returns (x: float64; alarm: bool);
  var c: bool; r: bool;
let
  (x, alarm) = merge c (gps when c, false)
                       ((restart ins every r) ((gps, xv) when not c);
  c = true fby (merge c (not s when c) (s when not c));
  r = false fby (s and c);
tel
```

Lustre

Listing 1.3: Simplified compilation of a state machine

state machines, or automata. Hierarchical state machines have been introduced by Harel (1987) with the StateCharts formalism. Maraninchi (1991, 1992) designed the Argos language that gives a synchronous interpretation of the composition of automata. Later, ideas from Argos were adapted to propose an extension of a kernel of Lustre with state machines, within the *mode-automata* formalism [Maraninchi and Rémond (1998, 2003)]. Finally, building on these ideas, Colaço, Pagano, and Pouzet (2005) and Colaço, Hamon, and Pouzet (2006) propose a conservative modular extension of Lustre with state machines, with associated compilation techniques. This extension exploits the modular reset construct to reset states on entry and has been adapted to Scade.

Continuing our previous example, figure 1.3 presents a graphical representation of a state machine representing a simple embedded navigation system. An `s` input toggles between two modes: *GPS*, where the `x` output position is defined directly by the `gps` signal; and *INS*, where the `ins` node is instantiated to approximate the position. The signal `s` represents the loss of the GPS signal, that could occur, for example, inside a tunnel. When entering a mode, the **`fby`**s and node instances within must be *reinitialized*. While **`fby`**s are reinitialized by adding conditionals, treating node instances modularly requires a modular reset construct. A simplified compilation of the example state machine is shown in listing 1.3. Sampling and merging operators are used to encode the different states, or *modes*, and the transitions between them. We do not describe in detail the equations that define the clocks `c` and `r`: `c` is true when the *GPS* mode is active and false when the *INS* mode is active, and `r` is true only at the instant that follows an entry into *INS* mode. This last stream is used by the **`restart`** operator to reinitialize the `ins` node instance in a modular way.

We propose an original formalization of this operator and show that it is suitable for verifying compiler correctness. Our approach builds on the idea that the modular reset can be expressed recursively. Caspi and Pouzet (1997, §A.1) gave a formalization in a co-iterative Kahn semantics, that was later adapted in [Hamon and Pouzet (2000)] where a node instantiation "acts as a totally new function" whenever it is reset; the authors remark that this can be given a recursive interpretation. This idea is exploited by [Cohen, Gérard, and Pouzet (2012)] for the generation of parallel code from Lustre. Auger (2013) adapts unpublished work [Auger et al. (2012)], that builds on similar ideas, to give a mechanized formalization of the modular reset in Coq.

1.4 Vélus: an overview

The Vélus compiler is written in Coq with the exception of the lexer, the main application, the intermediate printers and the validated scheduling pass which are written in OCaml. It follows the modular approach of [Biernacki et al. (2008)]. We focus on dataflow synchronous languages and their compilation, and therefore take advantage of the abstract low-level machine model provided by CompCert and its verified algorithms for producing assembly code. The reason for choosing Coq over other ITPs is that CompCert is written in Coq. All passes are verified in the sense that their correctness is stated and proved within Coq. Figure 1.4 gives an overview of the compiler architecture, and we briefly describe its elements below.

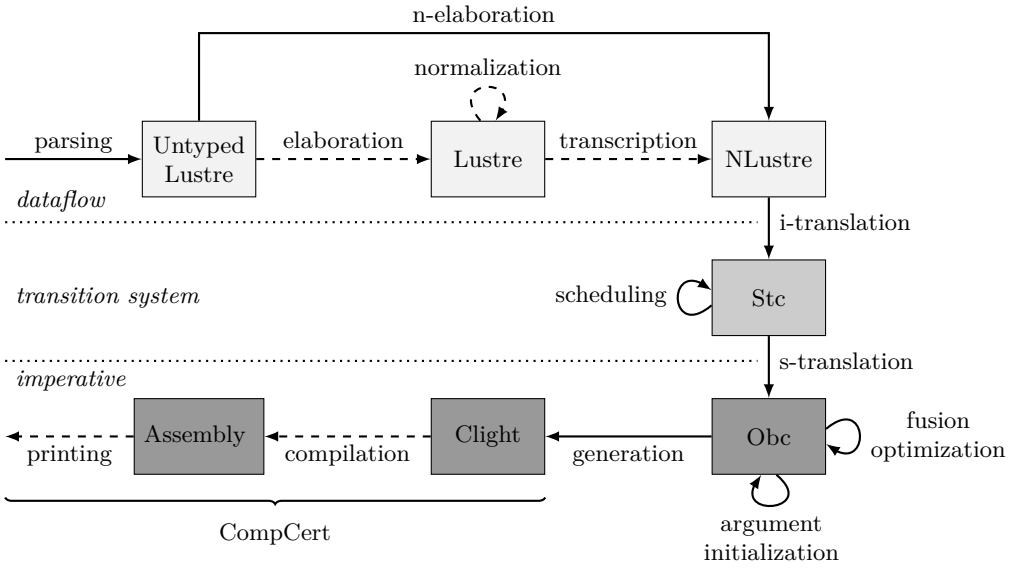


Figure 1.4: The architecture of Vélus

parsing reads a source file as an unannotated abstract syntax tree. It is handled by the Menhir [Pottier and Régis-Gianas (2018)] tool with the `--coq` option. With this flag, the generated parser comes with a Coq proof of correctness [Jourdan, Pottier, and Leroy (2012)].

elaboration implements type checking and clock checking. It turns raw syntax trees into Lustre programs annotated with types, clocks and related predicates. It is not yet complete, and is thus dashed in figure 1.4.

normalization is a source-to-source transformation that rewrites a Lustre program into a normalized form to prepare for further compilation to imperative code. This pass has been studied in previous work by Auger (2013) and is not yet implemented in Vélus.

transcription transforms normalized Lustre code into NLustre whose abstract syntax encodes the normalized form. This pass is not yet complete.

n-elaboration bypasses elaboration, normalization and transcription to directly produce annotated NLustre code from manually normalized Lustre programs.

i-translation represents the first change of paradigm: the code is translated from NLustre to Stc which is a transition system language introduced in this dissertation, and where synchronous reactions are modeled as transitions between states.

scheduling of dataflow equations in order to determine the order of future generated imperative instructions is performed on Stc: this allows, in particular, to schedule modular reset equations independently.

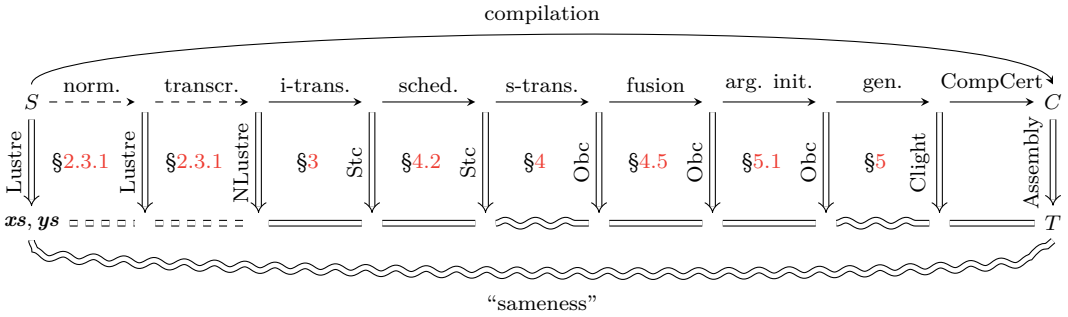


Figure 1.5: Simulation diagrams composition

s-translation is the second and final change of paradigm: we generate imperative Obc code, Obc being a lightweight object-like imperative language.

fusion optimization aims at reducing the amount of conditional statements introduced by the translation of clocked equations.

argument initialization is a stage where the compiler adds initialization values and validity assertions to ensure that function calls arguments are always initialized (as required by Clight but not otherwise guaranteed by the compilation scheme).

generation of Clight code, a C-like language from the frontend of CompCert, is the last pass of Vélus which relies on CompCert for the rest of the compilation chain.

To verify that the generated assembly code correctly implements the source language semantics, we establish relations for each transformation, like the one presented in figure 1.2, and compose them, as shown in figure 1.5.

We organized the development to mirror as much as possible the architecture pictured in figure 1.4. In appendix A, we outline concrete details of this organization.

Figure 1.6 on the next two pages shows hows the Vélus compiler transforms a simple Lustre program. The example uses the `counter` node of listing 1.2a with shorter names to save space. The `counter_rst` node uses the modular reset construct to restart the counter every time `r` is true. The first transformation is normalization, which we perform here manually. Two equations are introduced to define the **`fby`** with a constant value and an initialization clock `i` that is tested at each cycle to detect the initial instant. The next transformation, i-translation, produces Stc code where nodes have become systems. State is made explicit: state variables are distinguished with the **`init`** keyword and sub-systems are declared with the keyword **`sub`**. Stc is a language that defines transition systems. A system has one *default transition* that is defined by declarative equations that we call *transition constraints*, and an implicit *reset transition* that reinitializes state variables to their initial values. At the level of transition constraints, reset and default transitions are made distinct. The scheduling transformation orders the transition constraints in anticipation of the next transformation, s-translation, where each system is translated into a class, with a *step* method corresponding to the default transition and a *reset*

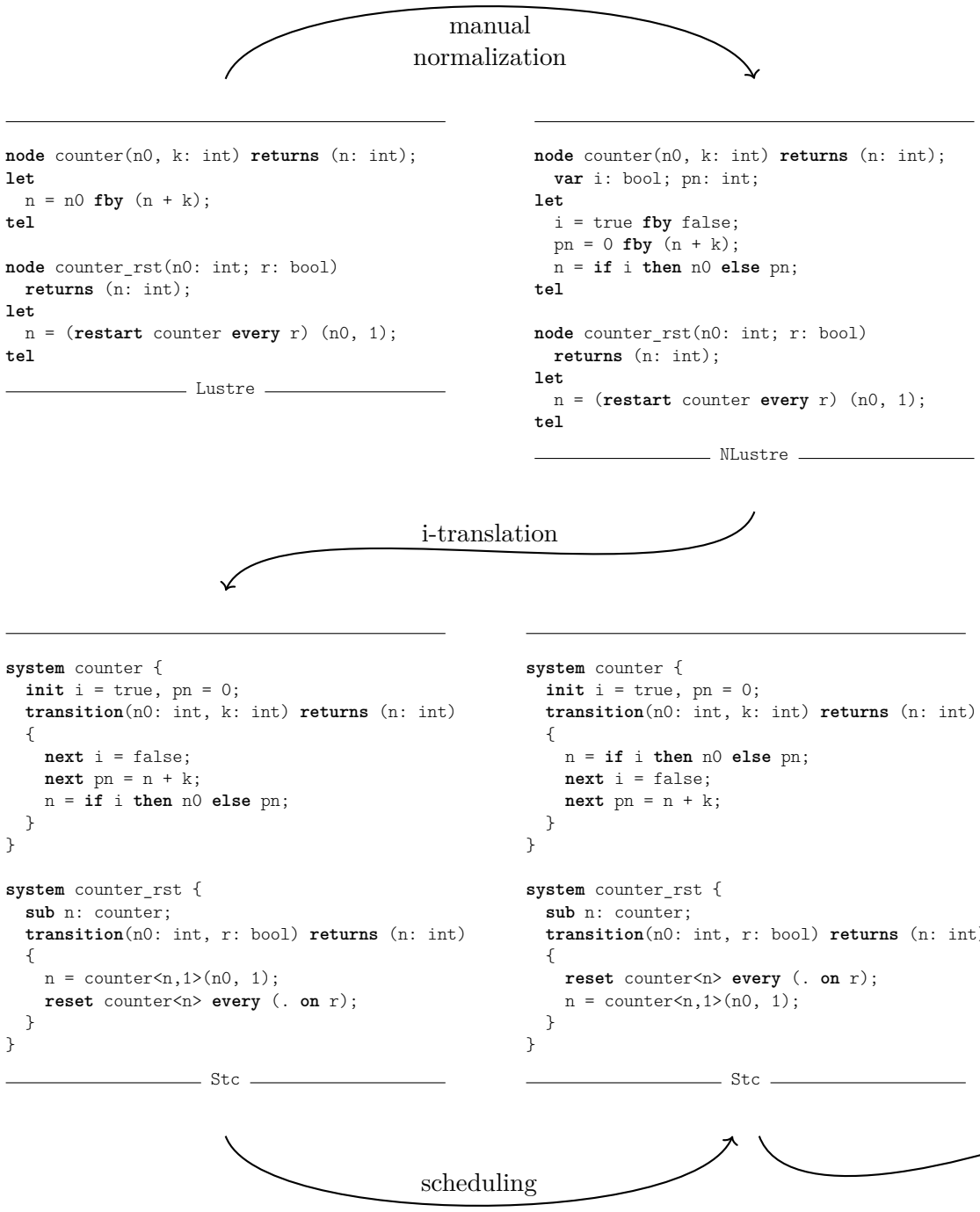
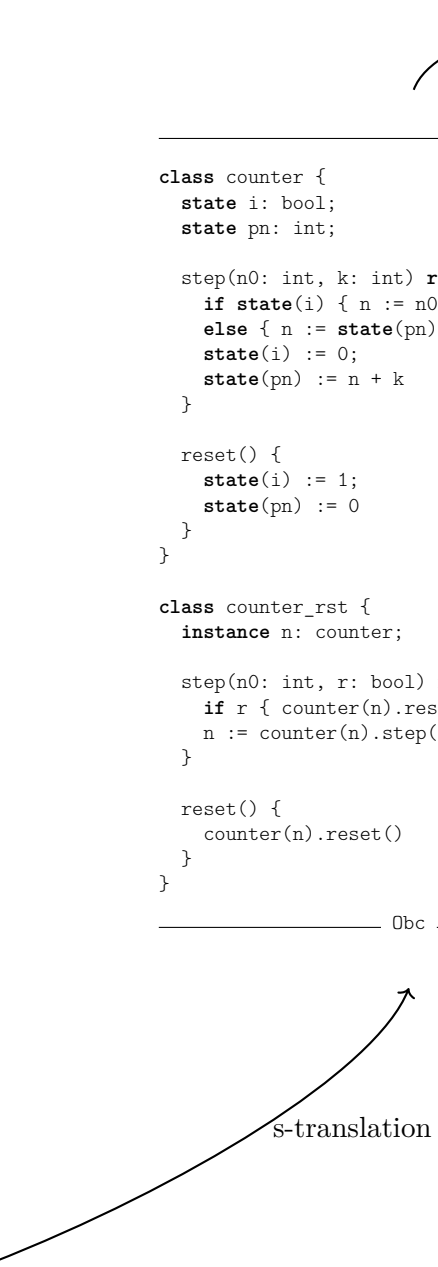


Figure 1.6 (I): Successive transformations of an example Lustre program




```
class counter {
  state i: bool;
  state pn: int;

  step(n0: int, k: int) returns (n: int) {
    if state(i) { n := n0 }
    else { n := state(pn) };
    state(i) := 0;
    state(pn) := n + k
  }

  reset() {
    state(i) := 1;
    state(pn) := 0
  }
}

class counter_rst {
  instance n: counter;

  step(n0: int, r: bool) returns (n: int) {
    if r { counter(n).reset() } else { };
    n := counter(n).step([n0], 1)
  }

  reset() {
    counter(n).reset()
  }
}
```

Obc


```
struct counter { _Bool i; int pn; };
struct counter_rst { struct counter n; };

int fun$counter$step
  (struct counter *self, int n0, int k)
{
  register int n;
  if ((*self).i) { n = n0; }
  else { n = (*self).pn; }
  (*self).i = 0;
  (*self).pn = n + k;
  return n;
}

void fun$counter$reset(struct counter *self)
{
  (*self).i = 1;
  (*self).pn = 0;
  return;
}

int fun$counter_rst$step
  (struct counter_rst *self, int n0, _Bool r)
{
  register int n;
  register int step$n;
  if (r) { fun$counter$reset(&(*self).n); }
  step$n = fun$counter$step(&(*self).n, n0, 1);
  n = step$n;
  return n;
}

void fun$counter_rst$reset
  (struct counter_rst *self)
{
  fun$counter$reset(&(*self).n);
  return;
}

struct counter_rst self$;
int volatile n$;
int volatile n0$;
_Bool volatile r$;

int main(void)
{
  register int n;
  register int n0;
  register _Bool r;
  fun$counter_rst$reset(&self$);
  while (1) {
    n0 = builtin volatile load int32(&n0$);
    r = builtin volatile load int8u(&r$);
    n = fun$counter_rst$step(&self$, n0, r);
    builtin volatile store int32(&n$, n);
  }
}
```

Clight

Figure 1.6 (II): Successive transformations of an example Lustre program

method corresponding to the implicit reset transition. The transition constraints are translated into imperative statements, that is, sequences of assignments to variables or state variables, conditionals and method calls. The final transformation (fusion optimization and argument initialization do not change the Obc code in this example) produces Clight code. The state is held by structures passed by reference, and the entry point of the program is defined by an infinite single loop that alternates reads of inputs, calculation of outputs by the main *step* transition function, and writes to outputs. From this point, CompCert compilation function is applied to produce assembly code, that we do not show here.

1.5 Organization

Chapter 2 presents the formalization of the Lustre dialect treated by Vélus. In particular, we give a semantics to the modular reset construct. We motivate the need for two equivalent semantics for the normalized NLustre language, based on whether streams are represented coinductively or as functions from $\mathbb{N}$ to values.

Chapter 3 diverges from the usual modular compilation scheme: before being translated to imperative code, the NLustre program is transformed into the Stc intermediate transition system language. We show the interest of this approach, in terms of semantic reasoning and efficiency of the generated code.

Chapter 4 describes the first step into the imperative paradigm. Stc code is translated into Obc code where each node is represented as a class that encapsulates its state and two methods that act on this state: a *step* method that implements one step of the execution and a *reset* method that reinitializes the state. Before being translated, the dataflow equations are scheduled to fix the imperative sequential order of execution. We describe how the generated Obc code is optimized.

Chapter 5 presents how we extend correctness to the machine level by using CompCert. The low-level memory model of CompCert is intricate and we use Separation Logic to facilitate the proof of correctness of the translation from Obc to Clight.

Figure 1.5 gives references to the various chapters and sections where the corresponding proofs are described.

1.6 Remarks

About proofs Every lemma, corollary or theorem presented in this dissertation is proved in Coq. Thus I do not provide detailed descriptions of the proofs in this dissertation. A version with outlines of the proofs can be found at:

www.leliobrun.net/files/thesis-with-proofs.pdf

Source code Besides lemmas, every definition that appears in the text is associated with a reference to the actual development, indicating the corresponding file and line numbers. In the electronic PDF version, these references are links to files in the following public repository:

github.com/INRIA/velus/tree/lelio-thesis

Common notations Throughout this dissertation, we use some common notations. To indicate a list, we use the bold face $\boldsymbol{x}$. The empty list is written ε, and $h \cdot \boldsymbol{t}$ is the *consing* operation of the *head* h to the *tail* $\boldsymbol{t}$. We also write $[h]$ for the singleton $h \cdot \varepsilon$. When relevant, we use the ellipsis notation $x_1 \cdots x_n$ where x_i is the standard projection notation, and the range notation $(x_i)_{i \in [1,n]}$. Since we work a lot with composite lists, we may write, for example, $\boldsymbol{x}^{\boldsymbol{y}}$ for the list of tuples $x_1^{y_1} \cdots x_n^{y_n}$. Finally, we write $\boldsymbol{x} + \boldsymbol{y}$ for the concatenation of the two lists $\boldsymbol{x}$ and $\boldsymbol{y}$.

Let E be an environment that maps identifiers to elements of a given type A, we write $E(x)$ to describe the *lookup* operation. In particular we write $E(x) = a$ to mean that x is bound to a in E. We write $x \notin E$ to mean that x does not appear in E. We use the symbol $\doteq$ as a general convention to designate the result of such partial operations but environments have a special treatment for convenience. We will also write $E\{x \mapsto a\}$ for the standard *update* operation, and $\varnothing$ for the empty environment. These notations extend to lists: $E(\boldsymbol{x})$ is the list $E(x_1) \cdots E(x_n)$, and $E\{\boldsymbol{x} \mapsto \boldsymbol{a}\}$ is the environment $E\{x_1 \mapsto a_1\} \cdots \{x_n \mapsto a_n\}$.

Related publications

[Bourke, Brun, Dagand, et al. (2017)] This work presents the first version of Vélus on which I started my work, without modular reset and coinductive-based semantics. Chapter 5 describes in detail the generation of Clight code from Obc code that the article briefly presents.

[Bourke, Brun, and Pouzet (2018)] This work presents the initial version of the formalization of the semantics of the modular reset further described in chapter 2.

[Bourke, Brun, and Pouzet (2020)] This work presents the second version of Vélus, in which the compilation chain is adapted to handle the modular reset in a way that is described in chapters 3 and 4 with the addition of Stc.

Chapter 2

Mechanized formalization of Lustre

Contents

In this chapter we present the formalization of the Lustre dialect used in Vélus. In section 2.1 we will explain how the formalization is made abstract over the operators of the target language, and how absence and streams are modeled. Section 2.2 presents the syntax and the formal coinductive-based semantics of Lustre. In particular, we show how to give a semantics to the modular reset construct.

After the elaboration stage that produces annotated Lustre code, the first code transformation pass is normalization, needed for the sequential code generation modular approach that we follow. In section 2.3 we focus on NLustre (for *normalized* Lustre) which is a syntactic variant of Lustre. NLustre is given two provably equivalent semantics: the first models streams coinductively while the second models streams as functions from natural numbers to values. The former is used to prove correct the *transcription* pass that transforms a normalized Lustre program into an NLustre program, and the latter is used for further compilation correctness proofs.

Table 2.1: Abstract notations

Coq name	notation
`type`	τ
`const`	c
`true_val`	T
`false_val`	F
`sem_const c`	$[\![c]\!]$
`unop`	$\diamond$
`binop`	$\oplus$
`sem_unop op v t = Some v'`	$[\![op]\!]_\tau\ v \doteq v'$
`sem_binop op v1 t1 v2 t2 = Some v'`	$[\![op]\!]_{\tau_1 \times \tau_2}\ v_1\ v_2 \doteq v'$
`type_unop op t = Some t'`	$\vdash op : \tau \to \tau'$
`type_binop op t1 t2 = Some t'`	$\vdash op : \tau_1 \times \tau_2 \to \tau'$

2.1 Preliminary definitions

Before presenting the semantic model of Lustre, we describe several fundamental modeling choices, namely on the integration of values, types and operators from the underlying target language, the representation of presence and absence, and the modeling of streams.

2.1.1 Abstraction layer

In Lustre, the definition of constants, types and operators depend on the target language [Caspi, Pilaud, et al. (1987)]. We follow the same approach in Vélus, where the target language is made abstract throughout most of the development. The front-end, that is, the lexer and parser, are not abstracted, but directly adapted from those used by CompCert (see appendices B and C). In principle, it should be possible to instantiate our development for a different target language, though we are not aware of any suitable formalization. To achieve this abstraction, we use functors over a signature, the *abstraction layer*, that abstracts the constants, types and operators of the target language. Each development *unit*—be it a function implementing a pass of the compiler, a correctness proof, or an internal library—is encapsulated within a functor over this signature, instantiated in only one *place* for the Clight code generation pass.

The key parameters of this signature, named `OPERATORS`, are shown in listing 2.1. The parameters `val`, `type` and `const` represent, respectively, the semantic values, the types, and the constants of the target language. Note that `val` and `type` contain booleans. The parameters `unop` and `binop` represent, respectively, unary operators and binary operators. More generally, generic operators with parametric arity could have been used instead of hard-coded unary and binary arity, but this approach, taken initially, was abandoned for the sake of simplicity. The function `type_const` gives the type of an arbitrary constant, `init_type` gives a constant inhabiting a type, which is needed to initialize variables, and `sem_const` is a total function defining the semantics of a constant. The `sem_unop` and

```
(* Types *)

Parameter val : Type.
Parameter type : Type.
Parameter const : Type.

(* Booleans *)

Parameter true_val  : val.
Parameter false_val : val.
Axiom true_not_false_val : true_val <> false_val.

Parameter bool_type : type.

(* Constants *)

Parameter type_const : const -> type.
Parameter sem_const  : const -> val.
Parameter init_type  : type -> const.

(* Operations *)

Parameter unop  : Type.
Parameter binop : Type.

Parameter sem_unop  : unop -> val -> type -> option val.
Parameter sem_binop : binop -> val -> type -> val -> type -> option val.

(* Typing *)

Parameter type_unop  : unop -> type -> option type.
Parameter type_binop : binop -> type -> type -> option type.
```

Coq (src/Operators.v:18-49)

Listing 2.1: Key parameters of the OPERATORS signature

`sem_binop` functions give the semantics of the application of an operator. Note that both functions are partial: they return an optional value, and have to be given not only their evaluated arguments but also the types of these arguments. Indeed, citing CompCert's documentation [Leroy (2019)]:

> Most C operators are overloaded (they apply to arguments of various types) and their semantics depend on the types of their arguments. [...] Since operators are overloaded, the result [of an operator application] depends both on the static types of the arguments and on their run-time values.

The semantics of operators is partial because some operations can fail or yield an *undefined behavior* in C. For example, the integer division-by-zero is undefined behavior, as well as the less known division of the smallest signed integer by minus one.[1] While partiality is something that we want to be able to model, to allow the possibility to target complex non-total operators, we could have chosen to forbid overloading and to only allow type-specific operators. Finally, `type_unop` and `type_binop` are responsible for resolving (partially again) the result type of an operator application, given the argument types. Table 2.1 lists useful corresponding notations that will be used throughout this dissertation.

Besides the aforementioned parameters, the signature also contains various *axioms* about well-typing and decidability. An axiom in a signature is a fact for modules that use the signature and a proof obligation for modules that implement it. To explain how the development is affected by the way the abstraction layer is set up, we take a toy example in appendix D. This example, that also serves as an introduction to useful Coq basics, is very simple but the whole Vélus development relies on exactly the same model: each unit is enclosed in a functor with an argument for each unit it depends on. The caveat is that it is rather cumbersome to duplicate the dependence graph into the call graph when the number of dependencies and separate libraries grows. Indeed some functors having more than 15 parameters, it becomes quite tedious to maintain and modify, but otherwise works well in practice.

2.1.2 Absent and present values

As we have seen, in Lustre, a synchronous value is either present or absent at each instant. This principle is the base of the clock system and at the core of the semantics of the language. We will write $\langle v \rangle$ to designate the present value v, and $\langle\ \rangle$ to represent the absence of value. Note that, as explained, v is an abstract value to be implemented by the target language semantics: the type `val` is provided by the abstraction layer.

In the Coq implementation we define `value` as an inductive type with two constructors: `absent` for $\langle\ \rangle$ and `present v` for $\langle v \rangle$.

[1] The operation indeed overflows [Seacord (2018)], which is also undefined behavior.

```
Inductive value :=
| absent
| present (v: val).
```
Coq (src/Operators.v:175-177)

2.1.3 Modeling infinite sequences

Lustre is a language of streams, that is, infinite sequences, of present or absent values. Mathematically, a sequence can be defined as a function. For infinite sequences, the domain of such a function is the set of natural numbers $\mathbb{N}$. In Coq, when faced with the task of formalizing a system based on infinite sequences, there are two main choices:

1. using Coq's functions over `nat`, which is the set of Peano natural numbers, or
2. using *coinductive* definitions, which allow to describe and reason about infinite objects.

In Vélus we did both. Indeed, in the early life of the project, the choice was made to use functions, for three reasons. First by elimination: proofs about coinductive objects can involve technical difficulties, namely the *guardedness* condition on proof terms and a lack of dedicated tactics. Secondly, functions are well-known objects that are rather easy to use and reason about in Coq. Lastly, this choice allows to describe the dataflow synchronous semantics in a practical way inspired by the initial formalization [Caspi, Pilaud, et al. (1987)]: as a generalization of an instantaneous behavior. The problem is that the definition of this semantics is rather cumbersome and somewhat different from modern presentations [Colaço and Pouzet (2003); Auger et al. (2012)].

The decision is thus to somehow hide this particular semantics by restricting it to the internal passes—that is, those involving NLustre, presented in section 2.3—while giving the Lustre frontend a semantics based on coinductive streams, that is easier to read and closer to the literature. Still the notoriously hard proof-handling of coinduction remains, as we will see.

2.1.3.1 Indexed streams in Coq

A stream of elements of type A can be represented as a function $\mathbb{N} \to A$, that we term an *indexed stream*. In Coq:

```
Definition stream A := nat -> A.
```
Coq (src/IndexedStreams.v:42)

We use the usual function application notation $s\ n$ to designate the nth element of the stream s.

2.1.3.2 Coinductive streams in Coq

As Bertot (2005) notes, the addition of coinductive types has been studied during the first half of the nineties. For example, [Leclerc and Paulin-Mohring (1994); Paulin-Mohring (1996)] use an encoding in Coq of coinductive types based on greatest fixpoints to model infinite streams. Native coinduction support was finally added to Coq by Giménez (1996), based on his extension of the Calculus of Inductive Constructions (CIC) [Giménez (1995)].

Thus in our implementation, we base our definitions on the standard library *Streams*[2] which defines the coinductive type `Stream` as follows.

```
CoInductive Stream : Type :=
  Cons : A -> Stream -> Stream.
```
Coq

That is, an infinite stream of elements of type `A` is made by infinitely consing values of type `A`. The library also provides the projections `hd` to access the *head*, that is, the first element of a stream, and `tl` to access the *tail*, that is, the stream after the head. We will write $x \cdot s$ to designate the stream whose head is x and tail is s. Also, `Str_nth n s`, where `n` is a natural number, that we write s_n in this dissertation and `s # n` in our Coq development, accesses the `n`th element of the stream `s`.

2.1.3.3 Equality

What does it mean for two infinite objects to be equal? This question has to be tackled whatever model we choose, be it functions or coinductive streams. Indeed, even if functions are not formally considered infinite objects, we can still ask: what does it mean for two functions to be equal? The intuitive answer, for both models, is a kind of pointwise equality, or more precisely *observational* equality. We say that two functions are observationally equal if they have the same graph, that is, if their graphs are *equal*. Two streams are observationally equal if their elements are pairwise *equal*.

Now one may think that observational equality implies equality, but not in type theory. For example, the following assertion of "Leibniz equality"—that is, syntactic equality modulo reduction—cannot be proved in Coq:

$$\lambda\, x.\, 2 \times x = \lambda\, x.\, x + x$$

However, those two functions are observationally equal. In Coq, observational equality does not imply Leibniz equality, at least not without including the axiom of *functional extensionality*. The *FunctionalExtensionality*[3] library provides this axiom, stated for the more general case of dependent functions:

[2] coq.inria.fr/stdlib/Coq.Lists.Streams.html
[3] coq.inria.fr/stdlib/Coq.Logic.FunctionalExtensionality.html

```
Axiom functional_extensionality_dep : forall {A} {B : A -> Type},
    forall (f g : forall x : A, B x),
      (forall x, f x = g x) -> f = g.
```
Coq

We prefer not to use this axiom in Vélus—as it could lead to inconsistency—and choose to use the observational equality as the default equality for infinite objects. That is, our definitions and proofs are based on observational equality rather than Leibniz equality.

For indexed streams, observational equality, written $s_1 \approx s_2$, means pointwise Leibniz equality.

Definition 2.1.1 (eq_str, src/IndexedStreams.v:49)

$$s_1 \approx s_2 \leftrightarrow \forall n,\ s_1\ n = s_2\ n$$

For coinductive streams, observational equality, also named *bisimilarity*, is defined in the *Streams* library.

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
  eqst :
    hd s1 = hd s2 -> EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```
Coq

It is also essentially Leibniz equality applied pointwise, although the predicate itself is defined coinductively. We will write $s_1 \equiv s_2$ to indicate that the coinductive streams s_1 and s_2 are bisimilar.

2.2 Lustre

In this section we present the Lustre dialect used in Vélus. We describe the syntax and the semantics of the language, and particularly focus on the semantics of the modular reset.

2.2.1 Abstract syntax

In figure 2.1, we present the—lightly sugared—syntax of Lustre as it is defined the Vélus development. This syntax is parameterized over the abstraction layer described before. Hence, c, $\diamond$, $\oplus$ and τ represent the abstracted constants, unary operators, binary operators and types respectively, as summarized in table 2.1 on page 22. The category x represents identifiers, which we define as positive integers. We use standard notations: e^+ is a non-empty sequence $e \cdots e$; $e^+_{,}$ is a non-empty sequence separated by "," $e, \cdots, e$; $[e]$ is either e or nothing.

Note that we focus here on a dialect that is closer to Scade 6 [Colaço, Pagano, and Pouzet (2017)] than to early Lustre [Caspi, Pilaud, et al. (1987)], see figure 2.2. In particular we

e	::=		**expression**
	|	c	(constant)
	|	x^{a}	(variable)
	|	$(\diamond\ e)^{a}$	(unary operator)
	|	$(e \oplus e)^{a}$	(binary operator)
	|	$(e^{+}_{,}$ `fby` $e^{+}_{,})^{a^{+}_{;}}$	(unit delay)
	|	$(e^{+}_{,}$ `when (`x `=` b`))`^{la}	(sampling)
	|	(`merge` $x\ e^{+}_{,}\ e^{+}_{,})^{la}$	(merging)
	|	(`if` e `then` $e^{+}_{,}$ `else` $e^{+}_{,})^{la}$	(conditional)
	|	$(x($`(`$e^{+}_{,}$`))`$)^{a^{+}_{;}}$	(application)
	|	(`(restart` x `every` e`)(`$e^{+}_{,}$`))`$^{a^{+}_{;}}$	(modular reset)
a	::=	τ, nck	**single flow annotation**
la	::=	τ^{+}, nck	**synchronized flows annotation**
nck	::=		**named clocks**
	|	ck	(regular clock)
	|	(`(`x `:` ck`)`)	(stream clock)
eq	::=	$x^{+}_{,}$ `=` $e^{+}_{,}$	**equation**
d	::=	$x^{\tau,ck}$	**variable declaration**

n	::=	(`node`|`function`) x`(`$d^{+}_{,}$`)` `returns` `(`$d^{+}_{,}$`)`
		[`var` d^{+}];
		`let`
		$eq^{+}_{;}$
		`tel`
			node

g	::=	n^{+}	**program**

Figure 2.1: The Lustre abstract annotated syntax

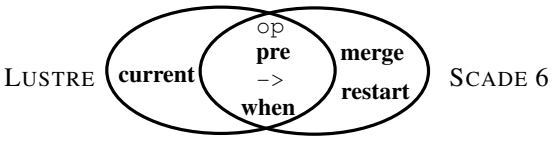


Figure 2.2: Operators origins [Colaço, Pagano, and Pouzet (2017)]

drop the operators **pre** and **current** as they cause problems with initialization. More precisely, **pre** is not yet treated since extra formalization and proofs are required to ensure that it is used correctly [Colaço and Pouzet (2004)]. As a replacement, we focus on the **fby** and **merge** operators. For convenience, we use a special notation for the sampling operation: the concrete code `e` **when** `x` is written e `when` (*x* = true) and `e` **whenot** `x`, or `e` **when not** `x` are written e `when` (*x* = false). The same convention is used for clocks.

Named clocks are introduced to track dependencies that occur with nested node applications. This is beyond the scope of this work, so we omit the details. More generally, we postpone the description of the clocks ck, as a simpler clock system will be explained later, for the normalized form NLustre.

Remark also that there exist two types of annotations in our syntax: the single flow annotation (a) and the synchronized multiple flows annotation (la). This distinction is used to annotate lists of sub-expressions. The single flow annotation is simply a pair of a type and a named clock and is used to annotate sub-expressions that do not need to be synchronized on the same clock. The multiple synchronized flows annotation pairs a list of types with a single clock: it is used to annotate a list of sub-expressions that must be on the same clock.

We will use the notations n.name, n.**in**, n.**out**, n.**vars**, n.**eqs** to indicate, respectively, the name, the input variables, output variables, local variables and equations of the node n.

In the following, we present the implementation of selected parts of the abstract syntax of Lustre in the Coq development. Recall that the syntax definitions and—unless specified—every piece of Coq development of Vélus is parameterized over the `OPERATORS` signature described in section 2.1.1. Consequently, every occurrence of `const`, `unop`, `binop` and `type` refers to this abstract signature.

Expressions Listing 2.2a presents the implementation of expressions: the inductive type `exp` corresponds to e in figure 2.1. The `Ewhen` constructor possesses a boolean parameter which is used to distinguish between **when** if the parameter is true and **when not** otherwise. The modular reset, as it shares most of its syntax with the regular application, is not given its own constructor. Rather, we add an optional expression parameter to the `Eapp` constructor signaling the presence of a reset. We work with possibly empty lists because they are easier to manipulate, and use independent predicates to ensure that they are not empty.

Equations The implementation of equations shown in listing 2.2b is straightforward: an equation pairs a list of variables and a list of expressions.

Nodes The abstract syntax for a node is defined using a dependent record, as shown in listing 2.2c. Such records have three main advantages:

1. We can refer to their elements using named fields.
2. Fields may be *propositions* that refer to other fields.

```
Definition ann : Type := (type * nclock)%type.
Definition lann : Type := (list type * nclock)%type.

Inductive exp : Type :=
| Econst : const -> exp                                        (* constant *)
| Evar   : ident -> ann -> exp                                 (* variable *)
| Eunop  : unop -> exp -> ann -> exp                           (* unary operator *)
| Ebinop : binop -> exp -> exp -> ann -> exp                   (* binary operator *)

| Efby   : list exp -> list exp -> list ann -> exp             (* unit delay *)
| Ewhen  : list exp -> ident -> bool -> lann -> exp            (* sampling *)
| Emerge : ident -> list exp -> list exp -> lann -> exp        (* merging *)
| Eite   : exp -> list exp -> list exp -> lann -> exp          (* conditional *)

| Eapp   : ident -> list exp -> option exp -> list ann -> exp.  (* application / reset *)
```

Coq (src/Lustre/LSyntax.v:44-58)

(a) Expressions

```
Definition equation : Type := (list ident * list exp)%type.
```

Coq (src/Lustre/LSyntax.v:64)

(b) Equations

```
Record node : Type :=
  mk_node {
      n_name     : ident;                          (* name *)
      n_hasstate : bool;                           (* statefulness *)
      n_in       : list (ident * (type * clock));  (* inputs *)
      n_out      : list (ident * (type * clock));  (* outputs *)
      n_vars     : list (ident * (type * clock));  (* local variables *)
      n_eqs      : list equation;                  (* equations *)

      n_ingt0    : 0 < length n_in;
      n_outgt0   : 0 < length n_out;
      n_defd     : Permutation (vars_defined n_eqs)
                               (map fst (n_vars ++ n_out));
      n_nodup    : NoDupMembers (n_in ++ n_vars ++ n_out);
      n_good     :  Forall ValidId (n_in ++ n_vars ++ n_out)
                    /\ valid n_name
    }.
```

Coq (src/Lustre/LSyntax.v:157-173)

(c) Nodes

Listing 2.2: Implementation of Lustre

3. When building an object of record type, Coq allows omitting some of the fields, if they can be inferred or left as a proof obligation using the Coq *Program* standard library [Sozeau (2007, 2008)].

The boolean field `n_hasstate` encodes the *statefulness* of a node: it is false for nodes that are purely combinatorial and true for nodes that may be stateful, that is, that may contain **`fby`**'s or recursively stateful node instances. For now, this distinction is not used in Vélus.

Using dependent records means that we can encode some basic well formedness predicates directly with the node definition. Namely that a node has at least one input, `n_ingt0`; at least one output, `n_outgt0`; that equations define all outputs and local variables, `n_defd` (the `vars_defined` function gathers variables from the left-hand sides); that input, local and output variable declarations do not overlap and are unique, `n_nodup` (the `NoDupMembers` predicate ensures that an association list has no duplicates relative to the keys, that is, the list represents a map); and that the name of a node and its variable declarations are *valid* identifiers, `n_good` (an identifier is valid if it is not a reserved keyword and if it does not use reserved special characters).

2.2.2 Formal semantics

2.2.2.1 Synchronous streams operators

To formalize the semantics of Lustre, it is usual [Caspi and Pouzet (1998); Colaço and Pouzet (2003); Auger et al. (2012)] to define synchronous streams operators specifying the behavior of the primitive constructs of the language. The first operator defines the stream associated to a constant.

Definition 2.2.1 (const, src/CoindStreams.v:290)

$$\mathsf{const}\,(\mathit{true} \cdot bs)\ c \triangleq \langle [\![c]\!] \rangle \cdot \mathsf{const}\ bs\ c$$
$$\mathsf{const}\,(\mathit{false} \cdot bs)\ c \triangleq \langle\ \rangle \cdot \mathsf{const}\ bs\ c$$

To indicate when the stream is present or absent, the operator takes a *clock stream* argument, that is, a stream of booleans giving the tempo. The constant c is evaluated using the semantics for constants provided by the abstraction layer (the function `sem_const` in listing 2.1 on page 23) and written $[\![c]\!]$.

Figure 2.3 displays the semantics of the other operators. Note that as they are all partially defined—they impose stream synchronization—we decide to define them as relations rather than as functions, hence the inference rules. This is, I think, easier to define because it avoids optional result or error monad, easier to reason about because unwanted cases just do not exist, and more elegant. To make the predicates easier to read, we add the $\doteq$ symbol in front of their last arguments. This symbol has no formal value. Moreover, all of these are defined in Coq as coinductive predicates, a fact which we represent using doubled horizontal lines in the style of [Leroy and Grall (2009)], to distinguish from the presentation of inductive predicates with single-lined inference rules. Thus the definitions all follow the same pattern: there is one inference rule by definition

case, each one stating that the relation holds for consed streams, given that it holds coinductively for their tails.

Figure 2.3a shows how unary and binary operators semantics are lifted up to pointwise application on streams of values. The notation $\diamond_\tau^\uparrow$ is to be read "the lifting of the unary operator $\diamond$ for type τ". Taking the binary operation case, the operation is defined on only two cases: either (1) all heads are absent, and the relation must hold for the tails, either (2) all heads are present, the relation must hold for the tails and the result value must be the the result of the semantics of the operator on the two input values. Taking the usual addition operator on 32-bit integers, for example, the rule formalizes the following example, where white spaces represent absence of value.

x	1	2	3	4	5	$\cdots$
y	2	4	6	8	10	$\cdots$
$x + y$	3	6	9	12	15	$\cdots$

Figure 2.3b describes the synchronous streams operator responsible for specifying the conditional behavior. In Lustre, the conditional branching construct can be seen as a multiplexer applied point-wise. Hence, it behaves as a ternary operator, as shown on the chronogram below.

c	T	T	F	T	F	$\cdots$
x	1	2	3	4	5	$\cdots$
y	2	4	6	8	10	$\cdots$
`if` c `then` x `else` y	1	2	6	4	10	$\cdots$

Figures 2.3c and 2.3d show the behavior of the synchronous operators we will use to define the semantics of the temporal operators **`when`** and **`merge`**. The subtlety is the boolean parameter of the when operator: it represents the distinction between the complementary behaviors of **`when`**, when it is true, and **`when not`** otherwise. Consider the $b = \mathsf{true}$ case, the *result* stream is the sampled version of the first stream operand, according to the boolean values of the second. Thus the result stream has present values only when the variable stream outputs true boolean values, and absent values everywhere else. The merge operator takes two complementary streams as second and third operands and combines them according to its first stream operand to output a stream that combines the complementary values. The behavior of the synchronous operators for **`when`** and **`merge`** is illustrated below.

c	T	T	F	T	F	$\cdots$
x	1	2	3	4	5	$\cdots$
y	2	4	6	8	10	$\cdots$
$u = x$ `when` c	1	2		4		$\cdots$
$v = y$ `when not` c			6		10	$\cdots$
`merge` $c\ u\ v$	1	2	6	4	10	$\cdots$

The behavior of the operator describing the semantics of the **`fby`** unit delay, as presented in [Colaço and Pouzet (2003)], is more involved. As shown in figure 2.3e, it can be seen

$$\frac{\Diamond^{\uparrow}_{\tau}\, xs \doteq vs}{\Diamond^{\uparrow}_{\tau}\,(\langle\,\rangle \cdot xs) \doteq \langle\,\rangle \cdot vs} \qquad \frac{\Diamond^{\uparrow}_{\tau}\, xs \doteq vs \quad [\![\Diamond]\!]_{\tau}\; x \doteq v}{\Diamond^{\uparrow}_{\tau}\,(\langle x\rangle \cdot xs) \doteq \langle v\rangle \cdot vs}$$

$$\frac{\oplus^{\uparrow}_{\tau_1\times\tau_2} xs\; ys \doteq vs}{\oplus^{\uparrow}_{\tau_1\times\tau_2}\,(\langle\,\rangle \cdot xs)\;\;(\langle\,\rangle \cdot ys) \doteq \langle\,\rangle \cdot vs} \qquad \frac{\oplus^{\uparrow}_{\tau_1\times\tau_2} xs\; ys \doteq vs \quad [\![\oplus]\!]_{\tau_1\times\tau_2}\; x\; y \doteq v}{\oplus^{\uparrow}_{\tau_1\times\tau_2}\,(\langle x\rangle \cdot xs)\;\;(\langle y\rangle \cdot ys) \doteq \langle v\rangle \cdot vs}$$

(a) Unary and binary operators lifting
(lift1, src/CoindStreams.v:293 and lift2, src/CoindStreams.v:305)

$$\frac{\mathsf{ite}\; cs\; ts\; fs \doteq vs}{\mathsf{ite}\,(\langle\,\rangle \cdot cs)\;\;(\langle\,\rangle \cdot ts)\;\;(\langle\,\rangle \cdot fs) \doteq \langle\,\rangle \cdot vs}$$

$$\frac{\mathsf{ite}\; cs\; ts\; fs \doteq vs}{\mathsf{ite}\,(\langle \mathsf{T}\rangle \cdot cs)\;\;(\langle v\rangle \cdot ts)\;\;(\langle f\rangle \cdot fs) \doteq \langle v\rangle \cdot vs}$$

$$\frac{\mathsf{ite}\; cs\; ts\; fs \doteq vs}{\mathsf{ite}\,(\langle \mathsf{F}\rangle \cdot cs)\;\;(\langle t\rangle \cdot ts)\;\;(\langle v\rangle \cdot fs) \doteq \langle v\rangle \cdot vs}$$

(b) Conditional (ite, src/CoindStreams.v:351)

$$\frac{\mathsf{when}^{b}\; xs\; cs \doteq vs}{\mathsf{when}^{b}\,(\langle\,\rangle \cdot xs)\;\;(\langle\,\rangle \cdot cs) \doteq \langle\,\rangle \cdot vs}$$

$$\frac{\mathsf{when}^{b}\; xs\; cs \doteq vs \quad \mathsf{val\text{-}to\text{-}bool}\; c \doteq \neg b}{\mathsf{when}^{b}\,(\langle x\rangle \cdot xs)\;\;(\langle c\rangle \cdot cs) \doteq \langle\,\rangle \cdot vs}$$

$$\frac{\mathsf{when}^{b}\; xs\; cs \doteq vs \quad \mathsf{val\text{-}to\text{-}bool}\; c \doteq b}{\mathsf{when}^{b}\,(\langle v\rangle \cdot xs)\;\;(\langle c\rangle \cdot cs) \doteq \langle v\rangle \cdot vs}$$

(c) Sampling operator (when, src/CoindStreams.v:317)

Figure 2.3 (I): The semantic synchronous streams operators

$$\frac{\mathsf{merge}\ xs\ ts\ fs \doteq vs}{\mathsf{merge}\ (\langle\ \rangle \cdot xs)\ (\langle\ \rangle \cdot ts)\ (\langle\ \rangle \cdot fs) \doteq \langle\ \rangle \cdot vs}$$

$$\frac{\mathsf{merge}\ xs\ ts\ fs \doteq vs}{\mathsf{merge}\ (\langle \mathsf{T} \rangle \cdot xs)\ (\langle v \rangle \cdot ts)\ (\langle\ \rangle \cdot fs) \doteq \langle v \rangle \cdot vs}$$

$$\frac{\mathsf{merge}\ xs\ ts\ fs \doteq vs}{\mathsf{merge}\ (\langle \mathsf{F} \rangle \cdot xs)\ (\langle\ \rangle \cdot ts)\ (\langle v \rangle \cdot fs) \doteq \langle v \rangle \cdot vs}$$

(d) Merging operator (merge, src/CoindStreams.v:334)

$$\frac{\mathsf{fby1}\ v\ xs\ ys \doteq vs}{\mathsf{fby1}\ v\ (\langle\ \rangle \cdot xs)\ (\langle\ \rangle \cdot ys) \doteq \langle\ \rangle \cdot vs} \qquad \frac{\mathsf{fby1}\ y\ xs\ ys \doteq vs}{\mathsf{fby1}\ v\ (\langle x \rangle \cdot xs)\ (\langle y \rangle \cdot ys) \doteq \langle v \rangle \cdot vs}$$

$$\frac{\mathsf{fby}\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle\ \rangle \cdot xs)\ (\langle\ \rangle \cdot ys) \doteq \langle\ \rangle \cdot vs} \qquad \frac{\mathsf{fby1}\ y\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle x \rangle \cdot xs)\ (\langle y \rangle \cdot ys) \doteq \langle x \rangle \cdot vs}$$

(e) Unit-delay operator (fby1, src/Lustre/LSemantics.v:45 and fby, src/Lustre/LSemantics.v:56)

Figure 2.3 (II): The semantic synchronous streams operators

```
CoInductive when (b: bool)
  : Stream value -> Stream value -> Stream value -> Prop :=
| WhenA:
    forall xs cs rs,
      when b xs cs rs ->
      when b (absent · xs) (absent · cs) (absent · rs)
| WhenPA:
    forall x c xs cs rs,
      when b xs cs rs ->
      val_to_bool c = Some (negb b) ->
      when b (present x · xs) (present c · cs) (absent · rs)
| WhenPP:
    forall x c xs cs rs,
      when b xs cs rs ->
      val_to_bool c = Some b ->
      when b (present x · xs) (present c · cs) (present x · rs).
```

Coq (src/CoindStreams.v:317-332)

Listing 2.3: The synchronous stream operator for the **when** construct

as a state machine with two states: the initial state is described by the fby synchronous operator, and the other one by fby1. At the first occurrence of present values on the operands of the fby, the state machine outputs the *initial value* found on the first operand and switches once and for all to the state described by fby1. This operator implements the delay implied by the **fby** using its first operand as a memory. Note that besides providing an initial value, the second operand is constrained to remain synchronous with the third one. Note that the only role of the second parameter, the initial stream, is then to guarantee the synchronization. The overall behavior is displayed in the example below.

x	1	2	3	4	5	$\cdots$
y	2	4	6	8	10	$\cdots$
x fby y	1	2	4	6	8	$\cdots$

As an example of the implementation, listing 2.3 presents the Coq definition of the when operator. Compare to figure 2.3c: the operator is defined as a coinductive relation that has one constructor per valid case.

2.2.2.2 Semantics of Lustre

Now that we have defined the synchronous stream operators that define the behavior of the core constructs of Lustre, we can give a semantics to the language. The first challenge is to formalize the chronograms seen until now. We choose to represent such grids with a stream environment, that we call a *history*. A history is a partial map from identifiers to streams of present and absent values. Thus for a node to have a semantics, there must exist a history that associates every variable with a stream (row) in a way

that satisfies all the equations of the node. The semantics of equations and expressions is parameterized by the *base clock* of the enclosing node, that is, the fastest clock at which the node is activated. This choice is to express the semantics of constants without having to rely on the semantics of their clock annotations. Indeed this way, a constant is always evaluated on the base clock. In other semantics presented in the literature [Caspi and Pouzet (1998); Colaço and Pouzet (2003); Auger et al. (2012)], constants are annotated with their clock and the semantics is based on these annotations. In our case, elaboration automatically adds **`when`**'s to constants if required. Our approach has the advantage of formally separating clocking, typing and semantics. Therefore we omit the annotations in the semantic rules.

Figure 2.4a shows the semantics for expressions. We write $G, H, bs \vdash e \Downarrow \boldsymbol{s}$ to mean that in the program G, under the history H and with base clock bs the expression e is associated to the list of streams $\boldsymbol{s}$. Note that the semantics is not presented in a constructive fashion: it rather imposes constraints on the history rather than describing how to build it. This is why the semantics rules are inductive, not coinductive: they follow the structure of a term and use the coinductive operators already introduced. This design is mainly to formalize the semantics independently of the order of the equations of a node. Moreover it allows for easy isolation and case analysis (inversion) of predicates.

We now describe each rule. A constant c is associated with a single stream calculated from the base clock using the function const defined before.

A variable x is associated with any stream bisimilar to the one specified for x in H. In this particular context, the notation $H(x)$ thus designates a lookup modulo bisimilarity. This helps writing more readable and concise rules.

An operator application is evaluated using the lifting operators presented in the previous section, provided that the expression operands are recursively associated with single streams. The function types retrieves the list of types that annotates an expression.

Given the synchronous streams operators fby, when, merge and ite, the corresponding semantics rules are straightforward. We lift semantic predicates and synchronous stream operators to lists, writing:

- $G, H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{s}$ as a shorthand for $\forall i,\ G, H, bs \vdash e_i \Downarrow u_i \wedge \boldsymbol{s} = \mathsf{concat}\, \boldsymbol{u}$ (recall that the semantics associates an expression with a list of streams, so each u_i is a list, hence the concat)
- $\mathsf{fby}\, \boldsymbol{s_0}\ \boldsymbol{s} \doteq \boldsymbol{vs}$ for $\forall i,\ \mathsf{fby}\, s_{0i}\ s_i \doteq vs_i$
- $\mathsf{when}^b\, \boldsymbol{s}\ u \doteq \boldsymbol{vs}$ for $\forall i,\ \mathsf{when}^b\, s_i\ u \doteq vs_i$
- $\mathsf{merge}\, s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}$ for $\forall i,\ \mathsf{merge}\, s\ ts_i\ fs_i \doteq vs_i$
- $\mathsf{ite}\, s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}$ for $\forall i,\ \mathsf{ite}\, s\ ts_i\ fs_i \doteq vs_i$

The semantics of a node application uses a dedicated predicate $G \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$ presented in figure 2.4c. The predicate relates input streams $\boldsymbol{xs}$ to output streams $\boldsymbol{ys}$. We write $\mathsf{node}(G, f) \doteq n$ to mean that the node n appears with the name f in the program G: it is a lookup of f in G. The predicate is defined internally in terms of a

$$\frac{s \equiv \mathsf{const}\ bs\ c}{G, H, bs \vdash c \Downarrow [s]} \qquad \frac{}{G, H, bs \vdash x \Downarrow [H(x)]}$$

$$\frac{G, H, bs \vdash e \Downarrow [s] \quad \mathsf{types}\, e = [\tau] \quad \diamond^{\uparrow}_{\tau}\, s \doteq vs}{G, H, bs \vdash \diamond e \Downarrow [vs]}$$

$$\frac{\begin{array}{c} G, H, bs \vdash e_1 \Downarrow [s_1] \quad G, H, bs \vdash e_2 \Downarrow [s_2] \\ \mathsf{types}\, e_1 = [\tau_1] \quad \mathsf{types}\, e_2 = [\tau_2] \quad \oplus^{\uparrow}_{\tau_1 \times \tau_2} s_1\ s_2 \doteq vs \end{array}}{G, H, bs \vdash e_1 \oplus e_2 \Downarrow [vs]}$$

$$\frac{G, H, bs \vdash \boldsymbol{e_0} \Downarrow \boldsymbol{s_0} \quad G, H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{s} \quad \mathsf{fby}\ \boldsymbol{s_0}\ \boldsymbol{s} \doteq \boldsymbol{vs}}{G, H, bs \vdash \boldsymbol{e_0}\ \texttt{fby}\ \boldsymbol{e} \Downarrow \boldsymbol{vs}}$$

$$\frac{G, H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{s} \quad \mathsf{when}^{b}\ \boldsymbol{s}\ (H(x)) \doteq \boldsymbol{vs}}{G, H, bs \vdash \boldsymbol{e}\ \texttt{when}\ (x = b) \Downarrow \boldsymbol{vs}}$$

$$\frac{G, H, bs \vdash \boldsymbol{e_t} \Downarrow \boldsymbol{ts} \quad G, H, bs \vdash \boldsymbol{e_f} \Downarrow \boldsymbol{fs} \quad \mathsf{merge}\,(H(x))\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{G, H, bs \vdash \texttt{merge}\ x\ \boldsymbol{e_t}\ \boldsymbol{e_f} \Downarrow \boldsymbol{vs}}$$

$$\frac{G, H, bs \vdash e \Downarrow [s] \quad G, H, bs \vdash \boldsymbol{e_t} \Downarrow \boldsymbol{ts} \quad G, H, bs \vdash \boldsymbol{e_f} \Downarrow \boldsymbol{fs} \quad \mathsf{ite}\, s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{G, H, bs \vdash \texttt{if}\ e\ \texttt{then}\ \boldsymbol{e_t}\ \texttt{else}\ \boldsymbol{e_f} \Downarrow \boldsymbol{vs}}$$

$$\frac{G, H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{xs} \quad G \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}{G, H, bs \vdash f\,(\boldsymbol{e}) \Downarrow \boldsymbol{ys}}$$

(a) Expressions without modular reset (sem_exp, src/Lustre/LSemantics.v:80)

Figure 2.4 (I): The semantics of Lustre

$$\frac{G, H, bs \vdash \boldsymbol{e} \Downarrow H(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} = \boldsymbol{e}}$$

(b) Equations (sem_equation, src/Lustre/LSemantics.v:152)

$$\frac{\mathsf{node}(G, f) \doteq n \quad H(\boldsymbol{x}) = \boldsymbol{xs} \quad H(\boldsymbol{y}) = \boldsymbol{ys} \quad \forall eq \in n.\mathbf{eqs},\ G, H, \mathsf{base\text{-}of}\ \boldsymbol{xs} \vdash eq}{G \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\tau_x, ck_x} \\ n.\mathbf{out} = \boldsymbol{y}^{\tau_y, ck_y} \end{array}$$

(c) Nodes (sem_node, src/Lustre/LSemantics.v:159)

Figure 2.4 (II): The semantics of Lustre

history H that must match the input and output streams and satisfy the constraints given by the semantics of the equations. Importantly, the universal quantification over $n.\mathbf{eqs}$ is invariant under permutation of $n.\mathbf{eqs}$. In other words, the semantics of a node does not depend on the order of its equations. The base clock used to defined the semantics of the equations is obtained from the input streams with the function base-of. The boolean stream base-of $\boldsymbol{xs}$ is the base clock for the node activation: it is true only when at least one input is present, that is, not absent.

Definition 2.2.2 (clocks_of, src/CoindStreams.v:368)

$$\mathsf{base\text{-}of}\ \boldsymbol{xs} \triangleq \mathsf{exists}_{\mathbb{B}}\ (\langle\ \rangle \neq_{\mathbb{B}})\ (\mathsf{map\ hd}\ \boldsymbol{xs}) \cdot \mathsf{base\text{-}of}\ (\mathsf{map\ tl}\ \boldsymbol{xs})$$

The semantics of an equation is given by a third dedicated sequent presented in figure 2.4b: it simply constrains the history to bind the left-hand side variables to streams resulting from the semantics of the right-hand side expressions. Note that the three presented sequents for expressions, equations and nodes are mutually recursive.

2.2.2.3 The modular reset

The rules presented in the previous subsection are an original formalization of Lustre in a proof assistant (we make several different choices to [Auger (2013)]), but they more or less directly implement the standard rules (see, for example, [Colaço and Pouzet (2003)]). In this section, we present a new way to formalize the modular reset in a "predicate style"—whether in an ITP or not.

As already explained in the introduction, the intuitive behavior of the reset is highly imperative: it effectively *restarts* a dataflow node by recursively resetting all **`fby`**s to their initial values. A node instantiation of `f(x)` reset by a boolean expression `r` is written `(`**`restart`**` f `**`every`**` r)(x)`, following the Scade syntax. The challenge is to give a formal dataflow semantics within an ITP.

Our starting point is the *recursive* intuition expressed by the program in listing 2.4 [Hamon and Pouzet (2000)]—not valid in actual Lustre because of forbidden recursion.

```
node true_until(r: bool) returns (c: bool)
let
  c = true -> if r then false else (pre c);
tel

node reset_f(x: int; r: bool) returns (y: int)
  var c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c)) (reset_f((x, r) when not c));
tel
```

Lustre

Listing 2.4: A (forbidden) recursive specialized reset node [Hamon and Pouzet (2000)]

The application `reset_f(x, r)` behaves as `f(x)` until `r` is true for the first time, then it recursively behaves as `reset_f(x', r')` where `x'` and `r'` are `x` and `r` taken starting from this very instant. This behavior is illustrated on the example below.

x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	$\cdots$
r	F	F	T	F	F	T	F	$\cdots$
c	T	T	F	F	F	F	F	$\cdots$
x when c	x_0	x_1						$\cdots$
$f(x$ when $c)$	y_0	y_1						$\cdots$
x'			x_2	x_3	x_4	x_5	x_6	$\cdots$
r'			T	F	F	T	F	$\cdots$
c'			T	T	T	F	F	$\cdots$
x' when c'			x_2	x_3	x_4			$\cdots$
$f(x'$ when $c')$			y_2	y_3	y_4			$\cdots$
x''						x_5	x_6	$\cdots$
r''						T	F	$\cdots$
c''						T	T	$\cdots$
x'' when c''						x_5	x_6	$\cdots$
$f(x''$ when $c'')$						y_5	y_6	$\cdots$
$\vdots$								
$reset\_f(x, r)$	y_0	y_1	y_2	y_3	y_4	y_5	y_6	$\cdots$

With this idea, a new *instance* of the node f comes into being on an accordingly sampled input each time r is true. Each of theses instances is activated only on the temporal window from when the instant r is true until *just before* the next instant where r is true, and forever if r is never true again. As a consequence of the semantics of the **`fby`**, and because all node signals are absent when all the inputs are, each such instance begins in its initial state, that is, all its **`fby`**s have their initial values. This is precisely what

models the imperative reset: instead of resetting the current instance *itself*, we simply start a new instance.

Rather than try to express this recursion directly in Coq, we take another approach: we formalize the semantics of the reset by *infinitely unrolling* the recursion. We define a special mask operator that filters its input by defining a window where its input is transmitted directly, and outside of which its value is absent.

Definition 2.2.3 (mask, src/CoindStreams.v:378)

$$\begin{aligned}
\mathsf{mask}^0_{\mathit{true}\cdot rs}\,(x \cdot xs) &\triangleq \mathsf{always\text{-}absent}\\
\mathsf{mask}^0_{\mathit{false}\cdot rs}\,(x \cdot xs) &\triangleq x \cdot \mathsf{mask}^0_{rs}\,xs\\
\mathsf{mask}^1_{\mathit{true}\cdot rs}\,(x \cdot xs) &\triangleq x \cdot \mathsf{mask}^0_{rs}\,xs\\
\mathsf{mask}^{k+1}_{\mathit{true}\cdot rs}\,(x \cdot xs) &\triangleq \langle\,\rangle \cdot \mathsf{mask}^k_{rs}\,xs\\
\mathsf{mask}^{k+1}_{\mathit{false}\cdot rs}\,(x \cdot xs) &\triangleq \langle\,\rangle \cdot \mathsf{mask}^{k+1}_{rs}\,xs
\end{aligned}$$

Where $\mathsf{always\text{-}absent} \triangleq \langle\,\rangle \cdot \mathsf{always\text{-}absent}$

The mask operator has three parameters: an instance number k, a reset stream rs and a stream to filter, xs. Intuitively, $\mathsf{mask}^k_{rs}\,xs$ counts at each instant n the number of reset ticks on the boolean stream rs seen so far and compares the result to k: if equal then it outputs the (absent or present) value of the stream xs at n, otherwise it outputs $\langle\,\rangle$. Below is shown the behavior of mask on the previous example, $\mathsf{count}\,r$ being the number of true ticks seen so far on the boolean stream r.

x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	$\cdots$
r	F	F	T	F	F	T	F	$\cdots$
$\mathsf{count}\,r$	0	0	1	1	1	2	2	$\cdots$
$\mathsf{mask}^0_r\,x$	x_0	x_1						$\cdots$
$f(\mathsf{mask}^0_r\,x)$	y_0	y_1						$\cdots$
$\mathsf{mask}^1_r\,x$			x_2	x_3	x_4			$\cdots$
$f(\mathsf{mask}^1_r\,x)$			y_2	y_3	y_4			$\cdots$
$\mathsf{mask}^2_r\,x$						x_5	x_6	$\cdots$
$f(\mathsf{mask}^2_r\,x)$						y_5	y_6	$\cdots$
$\vdots$								
`(reset` f `every` r`)(`x`)`	y_0	y_1	y_2	y_3	y_4	y_5	y_6	$\cdots$

We use the mask operator to formalize the semantics of the modular reset. First, recall the rule for the node application equation:

$$\frac{G, H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{xs} \quad G \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}{G, H, bs \vdash f(\boldsymbol{e}) \Downarrow \boldsymbol{ys}}$$

The rule for application with reset is very similar, we keep the predicate on the inputs of f. Additionally we have to give a semantics to the reset condition and to replace the premise on node application:

$$\frac{G, H, bs \vdash e_r \Downarrow [s] \quad \mathsf{bools\text{-}of}\ s \doteq r \qquad G, H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{xs} \quad \forall k,\ G \vdash f\left(\mathsf{mask}_r^k\ \boldsymbol{xs}\right) \Downarrow \mathsf{mask}_r^k\ \boldsymbol{ys}}{G, H, bs \vdash (\texttt{restart}\ f\ \texttt{every}\ e_r)(\boldsymbol{e}) \Downarrow \boldsymbol{ys}}$$

We simply use a universal quantifier to denote the unrolling of instances. Each instance's output is fully specified using the mask operator, where we write $\mathsf{mask}_r^k\ \boldsymbol{xs}$ as a shorthand for $\mathsf{mask}_r^k\ xs_1 \cdots \mathsf{mask}_r^k\ xs_i$. The boolean reset stream is obtained using the bools-of relation, which specifies the transformation of a stream of values into a stream of Coq booleans.

Definition 2.2.4 (bools_of, src/CoindStreams.v:371)

$$\frac{\mathsf{bools\text{-}of}\ vs \doteq bs \quad \mathsf{val\text{-}to\text{-}bool}\ v \doteq b}{\mathsf{bools\text{-}of}\ (v \cdot vs) \doteq b \cdot bs}$$

The overall result is consequently constrained as shown on the chronogram above: when the kth instance is applied to the k-masked inputs, its outputs are the k-masked part of the overall outputs. This approach works very well in Coq and is modular: the use of a universal quantifier allows for reasoning inductively about instances. Moreover, the semantics of the rest of the language is independent of our formalization of the reset operator.

This approach differs from the semantics proposed by Hamon and Pouzet (2000), where clocks are modeled as pairs of boolean streams indicating the presence and the resetting. The semantics of the `fby` is adapted to take this "semantic wire" into account, where our model treats the reset orthogonally.

In the formalization given in [Caspi and Pouzet (1997)], the Kahn semantics are defined co-iteratively using a transition function and an explicit state. For the reset, the transition function explicitly reinitializes the state whenever a reset occurs. Our solution does not expose state.

Our model resembles the semantics of Auger (2013, Figure 5.7, EVERY rule), but as his models treats lists rather than streams, the recursion is unrolled only for a finite number of instances, using an explicit sequencing operator to build the input and output lists of values.

2.3 NLustre: normalized Lustre

We present NLustre, an intermediate Lustre language that encodes the normalized form necessary for further compilation. We describe two semantic models based on the two models of streams: a coinductive semantics where streams are defined coinductively as in our formalization of Lustre, and an indexed semantics where streams are modeled as functions. The former is used to prove the correctness of the transcription pass that transforms a normalized Lustre program into an NLustre program, while the latter is used

to prove the correctness of the i-translation pass that transforms an NLustre program into an Stc program.

2.3.1 Normalization and transcription

Normalization is a standard stage in the modular approach for compiling Lustre. The aim is to rewrite the code into a form that can be directly translated into imperative code. This objective is mainly realized by identifying the *state* of a node by introducing specific equations for stateful computations, that is, for unit delays and consequently, by recursion, for node instantiations. Extracting the state facilitates both the scheduling of dataflow equations and their compilation to imperative code.

As stated by Auger et al. (2012, Remark 1), normalization can be implemented in two ways: by a source-to-source transformation along with a dedicated invariant, or by a translation towards a dedicated intermediate language. In Vélus, we choose to mix both approaches. The normalization process is a source-to-source transformation in Lustre, which has yet to be implemented and proved correct. Then, the normalized code is translated directly into NLustre, which encodes in its syntax the invariants obtained by normalization. Hence, we keep the advantages of both approaches: the correctness proof itself is easier since the semantics remains the same, but we use a simpler language with simpler semantics for the subsequent compilation pass.

As already mentioned, currently, normalization is not implemented. Consequently, Lustre code that is not in normalized form is simply rejected by the compiler, and only the transcription pass, that generates NLustre code from Lustre code is implemented. A version of the transcription without modular reset was proved correct by Jeanmaire (2019) during his Master's internship. We present nonetheless briefly the goals of the normalization pass, so as to provide the context necessary to understand the overall compilation scheme. As explained, the main objective is to unveil the basic equations that define the *state* of the node and to untangle conditional statements. Ultimately, the clock-directed compilation scheme recursively associates persistent variables to represent the delays of a node. Normalization achieves five goals, whose motivation details will become clearer in the next chapters. We describe them referring to the normalization in listing 2.5b of our Lustre example repeated in listing 2.5a.

1. Normalization extracts the equations whose compilation requires the introduction of persistent state. Consequently, the delays have to be named: the **`fby`** operator must appear only at top-level in the equations. Consider, for example, the expression `(0. fby x)` at line 9 in listing 2.5a, normalization introduces the local variable `px` that is defined by the equation at line 16 in listing 2.5b. The same holds recursively for node instantiations, in order to identify sub-states: the expression `euler((gps, xv) when not alarm)` at line 10 is given its own equation defining the fresh variable `xe` at line 14.

2. The compilation scheme uses clocks as guards: each equation is turned into a statement that is executed only when its associated clock is true. Operations that introduce branching, that is, the **`merge`** and conditional **`if/then/else`** operations,

```
node euler(x0, u: float64) returns (x: float64);
let
  x = x0 fby (x + 0.1 * u);
tel

node ins(gps, xv: float64) returns (x: float64; alarm: bool);
  var k: int;
let
  x = merge alarm ((0. fby x) when alarm)
                  (euler((gps, xv) when not alarm));
  alarm = (k >= 50);
  k = 0 fby (k + 1);
tel

node nav(gps, xv: float64; s: bool) returns (x: float64; alarm: bool);
  var c: bool; r: bool;
let
  (x, alarm) = merge c (gps when c, false)
                       ((restart ins every r) ((gps, xv) when not c);
  c = true fby (merge c (not s when c) (s when not c));
  r = false fby (s and c);
tel
```

Lustre

(a) Before

```
node euler(x0, u: float64) returns (x: float64);
var i: bool; px: float64;
let
  i = true fby false;
  x = if i then x0 else px;
  px = 0.0 fby (x + 0.1 * u);
tel

node ins (gps: float64, xv: float64) returns (x: float64, alarm: bool)
  var k: int32, px: float64, xe: float64 when not alarm;
let
  k = 0 fby k + 1;
  alarm = (k >= 50);
  xe = euler(gps when not alarm, xv when not alarm);
  x = merge alarm (px when alarm) xe;
  px = 0. fby x;
tel

node nav (gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
  var r: bool, c: bool, cm: bool, insr: float64 when not c, alr: bool when not c;
let
  (insr, alr) = (restart ins every r)(gps when not c, xv when not c);
  x = merge c (gps when c) insr;
  alarm = merge c (false when c) alr;
  cm = merge c (not s when c) (s when not c);
  c = true fby cm;
  r = false fby (s and c);
tel
```

Lustre

(b) After

Listing 2.5: Normalization of the example

must also only occur at the top of expressions. For example, the **merge** expression at line 20 is extracted to a dedicated equation at line 25.

3. To simplify the initialization of the state of a node, expressions at left of a **fby** operator are required to be constants rather than arbitrary expressions. The idea of the transformation is that the expression `e0 fby e` can be rewritten into `if (true fby false) then e0 else (c fby e)` where `c` is any constant of the right type. This expression is then normalized following the first point. The equation at line 3 is hence normalized into the three equations at lines 4–6.

4. An output variable must not be defined directly with a **fby**, that is it cannot be transformed into a persistent variable.

5. Finally, the reset condition expression of a node application with modular reset must be a variable, to facilitate compilation.

2.3.2 Abstract syntax

Figure 2.5 shows how the syntax of NLustre enforces the normal form:

- Two classes of expressions are defined: *basic expressions* which may not contain **merge** or **if/then/else** constructs, and *control expressions* which may.
- Stateful operations are available only at toplevel. Equations are divided into three main categories: basic equations, node instantiations with or without reset, and **fby** equations.
- Unit delay definitions are normalized: only constants can appear on the left of a **fby**.
- Except for node instantiation equations, the left-hand sides of equations are not lists anymore.
- The condition of a node application with reset is a variable rather than an expression.

Furthermore, the language remains annotated with (1) types, because every intermediate language is parameterized over the abstraction layer (see section 2.1.1), and (2) clocks, at equation level, following [Biernacki et al. (2008)] and to give a deterministic semantics to **fby** equations, as we will see. Additionally, the reset variable condition of a node application with modular reset is annotated with its clock: this is to avoid an unnecessary lookup in further code generation.

The Coq implementation is similar of that of Lustre described in section 2.2.1. Listings 2.6a and 2.6b show the implementation of expressions and control expressions as inductive types. The three kinds of equations are also represented as an inductive type, as shown by listing 2.6d. Finally, nodes are modeled as before using the dependent record presented in listing 2.6e. Compared to the predicates for a Lustre node, note the extra condition that forbids defining an output variable directly with a **fby**.

e	::=		**expression**
	|	c	(constant)
	|	x^t	(variable)
	|	$(\diamond\ e)^\tau$	(unary operator)
	|	$(e \oplus e)^\tau$	(binary operator)
	|	e `when (`x `=` b`)`	(sampling)
ce	::=		**control expression**
	|	`merge` x ce ce	(merging)
	|	`if` e `then` ce `else` ce	(conditional)
	|	e	(simple expression)
ck	::=		**clock**
	|	$\bullet$	(base clock)
	|	ck `on (`x `=` b`)`	(sub-clock)
eq	::=		**equation**
	|	$x =_{ck} ce$	(basic equation)
	|	$x =_{ck} c$ `fby` e	(**`fby`** equation)
	|	$x^+_, =_{ck} x(e^+_,)$	(node instantiation)
	|	$x^+_, =_{ck}$ `(restart` x `every` x^{ck}`)(`$e^+_,$`)`	(modular reset)
d	::=	$x^{\tau,ck}$	**variable declaration**

n	::=	`node` x`(`$d^+_,$`) returns (`$d^+_,$`)`
		`[var` $d^+_,$`]`
		`let`
		$eq^+_;$
		`tel`
			node

g	::=	n^+	**program**

Figure 2.5: The NLustre abstract syntax

```
Inductive exp : Type :=
| Econst : const -> exp
| Evar   : ident -> type -> exp
| Ewhen  : exp -> ident -> bool -> exp
| Eunop  : unop -> exp -> type -> exp
| Ebinop : binop -> exp -> exp -> type -> exp.
```

Coq (src/CoreExpr/CESyntax.v:23-28)

(a) Expressions

```
Inductive cexp : Type :=
| Emerge : ident -> cexp -> cexp -> cexp
| Eite   : exp -> cexp -> cexp -> cexp
| Eexp   : exp -> cexp.
```

Coq (src/CoreExpr/CESyntax.v:32-35)

(b) Control expressions

```
Inductive clock : Type :=
| Cbase : clock                          (* base clock *)
| Con   : clock -> ident -> bool -> clock. (* subclock *)
```

Coq (src/ClockDefs.v:22-24)

(c) Clocks

```
Inductive equation : Type :=
| EqDef : ident -> clock -> cexp -> equation
| EqApp : list ident -> clock -> ident -> list exp -> option (ident * clock) -> equation
| EqFby : ident -> clock -> const -> exp -> equation.
```

Coq (src/NLustre/NLSyntax.v:35-38)

(d) Equations

```
Record node : Type :=
  mk_node {
      n_name   : ident;                          (* name *)
      n_in     : list (ident * (type * clock));  (* inputs *)
      n_out    : list (ident * (type * clock));  (* outputs *)
      n_vars   : list (ident * (type * clock));  (* local variables *)
      n_eqs    : list equation;                  (* equations *)

      n_ingt0  : 0 < length n_in;
      n_outgt0 : 0 < length n_out;
      n_defd   : Permutation (vars_defined n_eqs)
                             (map fst (n_vars ++ n_out));
      n_vout   : forall out, In out (map fst n_out) ->
                        ~ In out (vars_defined (filter is_fby n_eqs));
      n_nodup  : NoDupMembers (n_in ++ n_vars ++ n_out);
      n_good   : Forall ValidId (n_in ++ n_vars ++ n_out) /\ valid n_name
    }.
```

Coq (src/NLustre/NLSyntax.v:74-90)

(e) Nodes

Listing 2.6: Implementation of NLustre syntax

2.3.3 NLustre code elaboration

In the Vélus compiler, the lexing and parsing stages (see appendices B and C) transform a Lustre source file into a raw abstract syntax tree (AST), without type and clock annotations (see src/Lustre/Parser/LustreAst.v). As Lustre elaboration, normalization and transcription are not complete yet, we directly elaborate NLustre code from manually normalized Lustre source code. The n-elaboration stage implements type-checking and clock-checking that enrich the AST with type and clock annotations. Since the syntax of nodes in Coq embeds some well-formedness invariants, elaboration must check them too. Type checking and clock checking are done simultaneously, because it is not only more efficient but also simplifies the proofs. The whole process, implemented by a function named n-elab (elab_declarations, src/NLustre/NLElaboration.v:2672), is not described in this dissertation.

2.3.4 Clock system

The *clock system* is a static calculus ensuring that a program can be executed synchronously [Caspi, Pilaud, et al. (1987); Caspi and Pouzet (1996)]. The idea is to statically reject programs that would not have a semantic model due to incorrect combinations of present and absent values, for example, those where a **merge** would have two stream arguments that are present at the same time. Each construct is associated with a clock, that gives its execution *tempo*, that is, it specifies the instant when the construct is absent or present. Consider the category *ck* in figure 2.5: a clock is either the base clock denoted •, that is, the fastest pace at which the node is activated, or a sub-clock that indicates sampling on a condition variable. The corresponding implementation appears in listing 2.6c.

The clock system is shown in figure 2.6. Figure 2.6a presents the clocking of expressions. The rules are parameterized over a clocking environment Ω that maps identifiers to clocks. We write $\Omega \vdash e :: ck$ to mean that under the clocking environment Ω, the expression e has clock ck. A constant always has the base clock. The clock of a variable is looked up in the environment. Unary and binary operations have the same clock as their operands. A sub-clock is introduced by the sampling **when** construct: the sub-expression and the condition variable must have the same clock.

The clocking rules of control expressions are shown in figure 2.6b. A **merge** expression on variable x has clock ck if the two sub-expressions have complementary sub-clocks deriving from ck with condition x, that must have clock ck. A conditional construct has the same clock as its branches. The clock of a basic expression does not change when it becomes a control expression. For example, consider line 15 in listing 2.5b, the replacement **merge** `alarm px xe` would not be well-clocked, because `px` is on the base clock •, while the operator expects an expression on clock • `on (alarm = true)`.

Figure 2.6c presents the rules for equations. Unlike expressions, equations are annotated with their activation clock. One goal of the calculus is thus to check that the clocks of the left-hand side and of the right-hand side of an equation match the annotation. This is what the rules for basic and **fby** equations implement. For a node instantiation, the

$$\frac{}{\Omega \vdash c :: \bullet} \qquad \frac{}{\Omega \vdash x :: \Omega(x)} \qquad \frac{\Omega \vdash e :: ck}{\Omega \vdash \diamond e :: ck} \qquad \frac{\Omega \vdash e_1 :: ck \quad \Omega \vdash e_2 :: ck}{\Omega \vdash e_1 \oplus e_2 :: ck}$$

$$\frac{\Omega \vdash e :: ck \quad \Omega(x) = ck}{\Omega \vdash e \texttt{ when } (x = b) :: ck \texttt{ on } (x = b)}$$

(a) Expressions (wc_exp, src/CoreExpr/CEClocking.v:41)

$$\frac{\Omega(x) = ck \quad \Omega \vdash_c e_t :: ck \texttt{ on } (x = \mathsf{true}) \quad \Omega \vdash_c e_f :: ck \texttt{ on } (x = \mathsf{false})}{\Omega \vdash_c \texttt{merge } x\ e_t\ e_f :: ck}$$

$$\frac{\Omega \vdash_c e :: ck \quad \Omega \vdash_c e_t :: ck \quad \Omega \vdash_c e_f :: ck}{\Omega \vdash_c \texttt{if } e \texttt{ then } e_t \texttt{ else } e_f :: ck} \qquad \frac{\Omega \vdash e :: ck}{\Omega \vdash_c e :: ck}$$

(b) Control expressions (wc_cexp, src/CoreExpr/CEClocking.v:64)

$$\frac{\Omega(x) = ck \quad \Omega \vdash_c e :: ck}{G, \Omega \vdash x =_{ck} e} \qquad \frac{\Omega(x) = ck \quad \Omega \vdash e :: ck}{G, \Omega \vdash x =_{ck} c \texttt{ fby } e}$$

$$\frac{\mathsf{node}(G, f) \doteq n \quad \sigma(\boldsymbol{x}) \overset{\text{var}}{=} \boldsymbol{e} \quad \sigma(\boldsymbol{y}) = \boldsymbol{z} \quad \Omega \vdash \boldsymbol{e} :: \sigma^{ck}[\boldsymbol{ck_x}] \quad \Omega(\boldsymbol{z}) = \sigma^{ck}[\boldsymbol{ck_y}]}{G, \Omega \vdash \boldsymbol{z} =_{ck} f(\boldsymbol{e})} \text{ where } \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

$$\frac{\mathsf{node}(G, f) \doteq n \quad \sigma(\boldsymbol{x}) \overset{\text{var}}{=} \boldsymbol{e} \quad \sigma(\boldsymbol{y}) = \boldsymbol{z} \quad \Omega \vdash \boldsymbol{e} :: \sigma^{ck}[\boldsymbol{ck_x}] \quad \Omega(\boldsymbol{z}) = \sigma^{ck}[\boldsymbol{ck_y}] \quad \Omega(r) = ck_r}{G, \Omega \vdash \boldsymbol{z} =_{ck} (\texttt{restart } f \texttt{ every } r^{ck_r})(\boldsymbol{e})} \text{ where } \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

(c) Equations (wc_equation, src/NLustre/NLClocking.v:52)

Figure 2.6 (I): The clock system of NLustre

rule is more involved. The idea is to exhibit a substitution σ, a map from identifiers to identifiers, that is used to instantiate the formal declared clocks of the node interface with the actual inferred clocks of the arguments and defined variables, following [Bourke and Pouzet (2019)]. This encodes within Coq the generalization/substitution mechanism of the clock calculus described in [Colaço and Pouzet (2003)]. The rule uses the following two definitions. The first definition extends the substitution lookup to expressions: if the substitution maps a declaration x in the node interface to a variable y in the context of the instantiation, then the corresponding argument expression must be the variable y; otherwise any argument expression is acceptable.

Definition 2.3.1 (SameVar, src/CoreExpr/CEClocking.v:31)

$$\frac{\sigma(x) = y}{\sigma(x) \overset{\text{var}}{=} y^{\tau}} \qquad\qquad \frac{x \notin \sigma}{\sigma(x) \overset{\text{var}}{=} e}$$

The second definition is a function that applies the substitution to a formal clock. We write $\sigma^{ck}\left[ck'\right]$ to designate the clock derived from ck' by substituting variables with σ and replacing the base clock by ck.

Definition 2.3.2 (instck, src/ClockDefs.v:27)

$$\sigma^{ck}\left[\bullet\right] \triangleq ck$$
$$\sigma^{ck}\left[ck' \text{ on } (x = b)\right] \triangleq \sigma^{ck}\left[ck'\right] \text{ on } (\sigma(x) = b)$$

Both definitions are lifted to lists: $\sigma(\boldsymbol{x}) \overset{\text{var}}{=} \boldsymbol{e}$ stands for $\forall i,\ \sigma(x_i) \overset{\text{var}}{=} e_i$; and $\sigma^{ck}\left[\boldsymbol{ck'}\right]$ for $\forall i,\ \sigma^{ck}\left[ck'_i\right]$. The rule for the node application with modular reset only adds the requirement that the clock annotation of the reset variable match its declared clock in the environment. As an example, consider the node interface:

```
node f(a : bool; b : int when a) returns (x : int on y; y : bool)
```

and the equation:

```
u, v = f(w, (10 when i) when w);
```

Assume that the variables `v` and `w` are declared with clock $ck = \bullet$ `on (`i `= true)`, and `u` with clock ck `on (`v `= true)`. That is, we consider a clocking environment $\Omega = \{i :: \bullet, v :: ck, w :: ck, u :: ck \text{ on } (v = \text{true})\}$. Hence the clock annotation of this equation is ck. The second two antecedents of the node instantiation rule constrain σ to contain the substitution $\{a \mapsto w, x \mapsto u, y \mapsto v\}$. The fourth antecedent ensures that the arguments are well-clocked, relative to the substitution:

1. The expression `w` must have clock $\sigma^{ck}\left[\bullet\right]$, that is, ck. We do have $\Omega(w) = ck$.
2. The expression `(10 when i) when w` must have clock $\sigma^{ck}\left[\bullet \text{ on } (a = \text{true})\right]$, that is, ck `on (`w `= true)`, since $\sigma(a) = w$. Using the clocking rule for constants and **when** expressions, this is also verified.

The last antecedent ensures that the clocks of right-hand side variables are correctly instantiated:

$$\frac{}{\Omega \vdash \bullet} \qquad \frac{\Omega(x) = ck \quad \Omega \vdash ck}{\Omega \vdash ck \text{ on } (x = b)}$$

(d) Clocks (wc_clock, src/Clocks.v:90)

$$\vdash \Omega \triangleq \forall x,\ \Omega(x) = ck \rightarrow \Omega \vdash ck$$

(e) Environments (wc_env, src/Clocks.v:101)

$$\frac{G, \Omega \vdash n.\mathbf{eqs} \quad \vdash \Omega \qquad \vdash \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\} \quad \vdash \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\}\{\boldsymbol{y} \mapsto \boldsymbol{\tau_y}\}}{G \vdash_{\text{WC}} n} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \\ n.\mathbf{vars} = \boldsymbol{z}^{\boldsymbol{\tau_z}, \boldsymbol{ck_z}} \\ \Omega = \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\}\{\boldsymbol{y} \mapsto \boldsymbol{ck_y}\}\{\boldsymbol{z} \mapsto \boldsymbol{ck_z}\} \end{array}$$

(f) Nodes (wc_node, src/NLustre/NLClocking.v:79)

Figure 2.6 (II): The clock system of NLustre

1. We do have $\Omega(u) = ck$ on $(v = \mathsf{true})$, and $\sigma^{ck}[\bullet \text{ on } (y = \mathsf{true})] = ck$ on $(v = \mathsf{true})$, since $\sigma(y) = v$.
2. We do have $\Omega(v) = ck$, and $\sigma^{ck}[\bullet] = ck$.

To extend the clock calculus to nodes, we introduce the notion of *clock consistency*, as shown in figure 2.6d. The idea is to verify that the declared clocks of the interface of a node are well defined. The base clock is consistent. A sub-clock is consistent if its sampling variable is associated to the parent clock in the environment and if this parent clock is consistent. In figure 2.6e we define a clocking environment to be consistent if all clocks that appear in it are consistent. Finally, as shown in figure 2.6f, a node is well-clocked if all its equations are well-clocked under the consistent clocking environment obtained from the input, output and local variable declarations of the node. Moreover we also require that the environments obtained from input declarations and from input and output declarations are consistent. A program is well-clocked if all of its nodes are well-clocked.

NLustre is also given a type system. In Vélus, well typing is only used to prove the correctness of the Clight code generation pass. Type systems are not central to this dissertation, but the reader may refer to appendix E.2 for the details.

2.3.5 The coinductive semantics

The coinductive semantics of NLustre is similar to the one of Lustre described in section 2.2.2. The corresponding inference rules are shown in figure 2.7. Most synchronous

$$\frac{s \equiv \mathsf{const}\ bs\ c}{H, bs \vdash c \Downarrow s} \qquad \frac{}{H, bs \vdash x \Downarrow H(x)} \qquad \frac{H, bs \vdash e \Downarrow s \quad \diamond^{\uparrow}_{\mathsf{type}\, e}\, s \doteq s'}{H, bs \vdash \diamond e \Downarrow s'}$$

$$\frac{H, bs \vdash e_1 \Downarrow s_1 \quad H, bs \vdash e_2 \Downarrow s_2 \quad \oplus^{\uparrow}_{\mathsf{type}\, e_1 \times \mathsf{type}\, e_2} s_1\ s_2 \doteq s'}{H, bs \vdash e_1 \oplus e_2 \Downarrow s'} \qquad \frac{H, bs \vdash e \Downarrow s \quad \mathsf{when}^b\ s\ (H(x)) \doteq s'}{H, bs \vdash e\ \mathtt{when}\ (x = b) \Downarrow s'}$$

(a) Expressions (sem_exp, src/NLustre/NLCoindSemantics.v:112)

$$\frac{H, bs \vdash_c e_t \Downarrow s_t \quad H, bs \vdash_c e_f \Downarrow s_f \quad \mathsf{merge}\,(H(x))\ s_t\ s_f \doteq s}{H, bs \vdash_c \mathtt{merge}\ x\ e_t\ e_f \Downarrow s}$$

$$\frac{H, bs \vdash e \Downarrow s \quad H, bs \vdash_c e_t \Downarrow s_t \quad H, bs \vdash_c e_f \Downarrow s_f \quad \mathsf{ite}\ s\ s_t\ s_f \doteq s}{H, bs \vdash_c \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \Downarrow s} \qquad \frac{H, bs \vdash e \Downarrow s}{H, bs \vdash_c e \Downarrow s}$$

(b) Control expressions (sem_cexp, src/NLustre/NLCoindSemantics.v:139)

Figure 2.7 (I): The coinductive semantics of NLustre

operators are reused, namely, const, the lifting of unary and binary operations, when, merge and ite. The exception is fby: here, the initial value of a unit delay is given by a constant. This difference greatly simplifies the semantics of the **`fby`** construct in NLustre. Indeed this operator can now be defined directly as a total function:

Definition 2.3.3 (fby, src/NLustre/NLCoindSemantics.v:177)

$$\mathsf{fby}\ v_0\ (\langle\ \rangle \cdot xs) \triangleq \langle\ \rangle \cdot \mathsf{fby}\ v_0\ xs$$
$$\mathsf{fby}\ v_0\ (\langle x \rangle \cdot xs) \triangleq \langle v_0 \rangle \cdot \mathsf{fby}\ x\ xs$$

2.3.5.1 Expressions

Figure 2.7a shows the semantics judgment for simple expressions: we write $H, bs \vdash e \Downarrow s$ to state that under the history H and with base clock bs the expression e is associated with the stream s. The function type retrieves the type annotation of an expression. Figure 2.7b presents the semantics of control expressions. We write $H, bs \vdash_c e \Downarrow s$ to distinguish control expressions from simple expressions. Compared to the rules presented in figure 2.4a on page 37, there is no G parameter since in normalized form, node instantiations no longer appear at expression level. The rules are also simpler in the sense that normalized expressions cannot be lists and consequently are associated with single streams instead of lists of streams. Nonetheless, the behavior formalized by the rules is essentially the same as for Lustre.

$$\frac{bs' \equiv bs}{H, bs \vdash \bullet \Downarrow bs'} \qquad \frac{H, bs \vdash ck \Downarrow \mathsf{true} \cdot bck \quad H(x) \equiv \langle v \rangle \cdot xs \quad \mathsf{val\text{-}to\text{-}bool}\, v \doteq b \quad \mathsf{tl}\, H, \mathsf{tl}\, bs \vdash ck \;\mathtt{on}\; (x = b) \Downarrow bs'}{H, bs \vdash ck \;\mathtt{on}\; (x = b) \Downarrow \mathsf{true} \cdot bs'}$$

$$\frac{H, bs \vdash ck \Downarrow \mathsf{true} \cdot bck \quad H(x) \equiv \langle v \rangle \cdot xs \quad \mathsf{val\text{-}to\text{-}bool}\, v \doteq \neg b \quad \mathsf{tl}\, H, \mathsf{tl}\, bs \vdash ck \;\mathtt{on}\; (x = b) \Downarrow bs'}{H, bs \vdash ck \;\mathtt{on}\; (x = b) \Downarrow \mathsf{false} \cdot bs'}$$

$$\frac{H, bs \vdash ck \Downarrow \mathsf{false} \cdot bck \quad H(x) \equiv \langle\ \rangle \cdot xs \quad \mathsf{tl}\, H, \mathsf{tl}\, bs \vdash ck \;\mathtt{on}\; (x = b) \Downarrow bs'}{H, bs \vdash ck \;\mathtt{on}\; (x = b) \Downarrow \mathsf{false} \cdot bs'}$$

(c) Clocks (sem_clock, src/NLustre/NLCoindSemantics.v:62)

$$\frac{H, bs \vdash_{(c)} e \Downarrow \langle v \rangle \cdot s \quad H, bs \vdash ck \Downarrow \mathsf{true} \cdot bs' \quad \mathsf{tl}\, H, \mathsf{tl}\, bs \vdash_{(c)} e :: ck \Downarrow s}{H, bs \vdash_{(c)} e :: ck \Downarrow \langle v \rangle \cdot s}$$

$$\frac{H, bs \vdash_{(c)} e \Downarrow \langle\ \rangle \cdot s \quad H, bs \vdash ck \Downarrow \mathsf{false} \cdot bs' \quad \mathsf{tl}\, H, \mathsf{tl}\, bs \vdash_{(c)} e :: ck \Downarrow s}{H, bs \vdash_{(c)} e :: ck \Downarrow \langle\ \rangle \cdot s}$$

(d) Clocked (control) expressions (sem_annot, src/NLustre/NLCoindSemantics.v:159)

Figure 2.7 (II): The coinductive semantics of NLustre

2.3.5.2 Clocking constraints

Before presenting the semantics of equations, we must first introduce some details about clock annotations. As noted earlier, the left-hand side of a **`fby`** equation in NLustre is a constant. This has a non trivial consequence on the associated semantic rule. To see why, consider the following equation:

```
x = true fby (not x);
```

In Lustre, the `true` expression denotes a stream whose clock is obtained from the expression. Here it is the base clock since there is no **`when`**. Consequently, the associated stream is always present with the boolean value T. Now, the same equation is also valid in NLustre. But in NLustre, the constant `true` does not yield a stream, therefore one can no longer give a unique clock to the stream associated to `x`. The solution to the resulting introduction of non-determinism is explained in [Caspi and Pouzet (1997), §4.3]: the semantics must rely on clocks. Here, the variable `x` is a declared variable, so its clock is statically known and available locally since **`fby`**s in NLustre only appear at equation level and equations are annotated with their clock. Therefore, in NLustre the semantics relies on clock annotations, unlike in Lustre.

Figure 2.7c presents the semantics of clocks. The base clock $\bullet$ is associated with the

$$\frac{H, bs \vdash_c e :: ck \Downarrow H(x)}{G, H, bs \vdash x =_{ck} e} \qquad \frac{H, bs \vdash e :: ck \Downarrow s \quad H(x) \equiv \mathsf{fby}\,[\![c]\!]s}{G, H, bs \vdash x =_{ck} c\ \texttt{fby}\ e}$$

$$\frac{H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{xs} \quad H, bs \vdash ck \Downarrow \mathsf{base\text{-}of}\ \boldsymbol{xs} \quad G \vdash f\,(\boldsymbol{xs}) \Downarrow H(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})}$$

$$\frac{\begin{array}{c} H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{xs} \quad H, bs \vdash ck \Downarrow \mathsf{base\text{-}of}\ \boldsymbol{xs} \\ \mathsf{bools\text{-}of}\,(H(y)) \doteq rs \quad \forall k,\ G \vdash f\left(\mathsf{mask}^k_{rs}\,\boldsymbol{xs}\right) \Downarrow \mathsf{mask}^k_{rs}\,H(\boldsymbol{x}) \end{array}}{G, H, bs \vdash \boldsymbol{x} =_{ck} (\texttt{restart}\ f\ \texttt{every}\ y)\,(\boldsymbol{e})}$$

(e) Equations (sem_equation, src/NLustre/NLCoindSemantics.v:187)

$$\frac{\begin{array}{c} \mathsf{node}(G, f) \doteq n \quad H(\boldsymbol{x}) \equiv \boldsymbol{xs} \quad H(\boldsymbol{y}) \equiv \boldsymbol{ys} \quad bs = \mathsf{base\text{-}of}\ \boldsymbol{xs} \\ \mathsf{respects\text{-}clock}\ H\ bs\ \Gamma \quad \forall eq \in n.\mathbf{eqs},\ G, H, bs \vdash eq \end{array}}{G \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\tau_{\boldsymbol{x}}, ck_{\boldsymbol{x}}} \\ n.\mathbf{out} = \boldsymbol{y}^{\tau_y, ck_y} \\ \Gamma = \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\} \end{array}$$

(f) Nodes (sem_node, src/NLustre/NLCoindSemantics.v:216)

Figure 2.7 (III): The coinductive semantics of NLustre

base stream parameter of the semantics (up to bisimilarity). The semantics for sampled clocks ensures that the sampling variable x yields a present value if and only if the sampled clock ck is true. Indeed, it means that x must have clock ck. This is guaranteed by the clocking rule for **when** (see figure 2.6a on page 48). The overall sub-clock is associated with a boolean stream which is false when ck is false, and true only if ck is true and the sampling condition is met. Note that the rules for subclocks are *corecursive*: they are instantiated on the tail of the base stream and on the tail of the history to constrain the tail of the overall result stream. We overload the tl notation for streams to histories—which are environments of streams—in the following way: we write $\mathsf{tl}\,H$ for the environment resulting from the composition $\mathsf{tl} \circ H$.

Figure 2.7d shows how the semantics for clocks and for (control) expressions are combined to enforce clocking constraints on the semantics of (control) expressions. The rules ensure that an expression evaluated stream yields present values if and only if its annotated clock yields true values. In Lustre this relation is a property of the semantics. Here, it is directly encoded in our semantics rules but "discharged" by the proof of correctness for the transcription pass. The theorem was shown by Jeanmaire (2019), again without reset.

2.3.5.3 Equations and nodes

Figure 2.7e shows the dedicated semantics rules for each kind of equation. A basic equation $x =_{ck} e$ has a semantics if its left-hand side and its right-hand side are evaluated to the same stream (up to bisimilarity). The left-hand side x is looked-up in the history H, while the right-hand side expression e is constrained together with the clock annotation ck, as described in the previous section. This may seem odd: we just explained that we need clock annotations to give the rhythm of a **`fby`** equation, but do we need it for a basic equation? Ideally the clocking constraints, the consistency of the synchrony, should be a consequence of the semantics and of a well clocking static predicate. But, by using clock annotations for one kind of equation, we already lose the independence of the semantics: we embedded this particular synchrony result directly into the semantics. As we need this result anyway, we might as well embed it further, that is for other kind of equations. This is essentially a matter of design choice, as NLustre is an intermediate language whose semantics is essentially used for internal proofs and the Lustre semantics is already shown to satisfy the property. Similarly, a **`fby`** equation has a semantics if the left-hand side variable is associated in the history with a stream that is bisimilar to the stream obtained using the fby function on the stream associated with the delayed expression. For a node application with or without reset, the input expressions are associated with a list of streams that must be related by the semantics of the node with streams associated with the left-hand side variables in the history (up to bisimilarity). Note that the clock annotation must correspond to the actual clock of activation of the node. The only particularity for the node application with reset is the use of the mask operator, exactly as described in section 2.2.2.3 for Lustre.

As for Lustre, the node instantiation semantics uses a dedicated sequent presented in figure 2.7f. In comparison to the judgment in figure 2.4c on page 38, we just add the respects-clock predicate, which embeds the clocking constraints, as we have seen for expressions, at the node input level. It expresses the alignment between the streams resulting from the evaluation of the input variables and of their clock annotations. We give its formal definition below.

Definition 2.3.4 (sem_clocked_var, src/NLustre/NLCoindSemantics.v:98)

$$\begin{aligned}\mathsf{respects\text{-}clock}\ H\ bs\ x^{ck} \triangleq (\forall xs,\ H(x) \equiv xs \rightarrow \exists bs,\ H, bs \vdash ck \Downarrow bs \\ \wedge\ \mathsf{clock\text{-}aligned}\ xs\ bs) \\ \wedge\ \forall bs,\ H, bs \vdash ck \Downarrow bs \rightarrow \exists xs,\ H(x) \equiv xs\end{aligned}$$

where

$$\frac{\mathsf{clock\text{-}aligned}\ vs\ bs}{\mathsf{clock\text{-}aligned}\ (\langle v \rangle \cdot vs)\ (\mathit{true} \cdot bs)} \qquad \frac{\mathsf{clock\text{-}aligned}\ vs\ bs}{\mathsf{clock\text{-}aligned}\ (\langle\ \rangle \cdot vs)\ (\mathit{false} \cdot bs)}$$

Having $\mathsf{respects\text{-}clock}\ H\ bs\ x^{ck}$ means that x has a semantics xs if and only if ck has a semantics bs such that xs and bs are *clock aligned*. That is, as we previously saw for the expressions, xs yields present values if and only if bs yields true values.

2.3.6 The indexed semantics

Having presented the coinductive semantics for NLustre, used to show the correctness of the transcription pass (Lustre to NLustre), we now present the indexed semantics used for the proof of correctness described in the next chapter. In the indexed model, streams are represented as functions from natural numbers to values. The main idea of the model is to describe the *instantaneous* behavior of the expressions, and to generalize it over multiple instants at the equation level. Therefore, with chronograms in mind, we give a specification to each column in the semantics of expressions, and the semantics for equations "glues" the columns into a grid. In changing from the coinductive semantics to the indexed semantics, we reflect a first step of compilation from a model where each variable is associated with a stream—a row of a chronogram—towards a sequential model where calculation is performed on a state cycle-by-cycle—the columns of a chronogram. In this way, the coinductive semantics facilitates the proof of transcription while the indexed semantics facilitates the proof of the generation of imperative code.

2.3.6.1 Instantaneous semantics

To describe the semantics at one particular instant, we have to consider the current value of the base clock stream b, and a *slice* R of the history. This slice is an environment mapping identifiers to absent or present values. As outlined above, using indexed streams and an instantaneous environment is fundamental in this approach: the idea is to reason instant-by-instant as easily as possible. The inference rules are presented in figure 2.8. Although there are no surprises compared to the behavior modeled by the coinductive semantics presented in section 2.3.5, we give a short explanation of the rules. The main difference is the absence of extra synchronous stream operators: here their behavior is directly *inlined* in the rules.

Figure 2.8a shows the inference rules for the instantaneous semantics of expressions. We write $R, b \vdash e \downarrow v$ to mean that in the environment R, with current base clock value b, the expression e evaluates to v. A constant is present and has the value assigned by the target language semantics when b is true, and is absent otherwise. A variable x is evaluated to the value it is bound to in R, provided that x actually belongs to the domain of R. Unary and binary operations simply use the abstracted semantics for operators when their operands are present. If the operands are absent then the overall result is also absent. A sampled expression e `when` $(x = k)$ is absent if e and x are also absent. If e and x are both present, then the sampled expression depends on the present value of x, that must be a boolean, and on k. If x evaluates to k, then the overall expression is evaluated to the present value of e, otherwise it is absent.

Figure 2.8b shows the instantaneous semantics of the control expressions. Again, we write $R, b \vdash_c e \downarrow v$ to distinguish control expressions from regular expressions. The merging operation is defined if its expression operands are complementary: if one is present, the other one has to be absent. The choice is made using the boolean value of the condition variable. The semantics of the conditional is similar except that the operands must both be present or absent at the same time.

$$\frac{}{R, \mathsf{true} \vdash c \downarrow \langle [\![c]\!] \rangle} \qquad \frac{}{R, \mathsf{false} \vdash c \downarrow \langle\ \rangle} \qquad \frac{}{R, b \vdash x \downarrow R(x)}$$

$$\frac{R, b \vdash e \downarrow \langle v_e \rangle \quad [\![\diamond]\!]_{\mathsf{type}\, e}\ v_e \doteq v}{R, b \vdash \diamond e \downarrow \langle v \rangle} \qquad \frac{R, b \vdash e \downarrow \langle\ \rangle}{R, b \vdash \diamond e \downarrow \langle\ \rangle}$$

$$\frac{H, bs \vdash e_1 \Downarrow \langle v_1 \rangle \quad H, bs \vdash e_2 \Downarrow \langle v_2 \rangle \quad [\![\oplus]\!]_{\mathsf{type}\, e_1 \times \mathsf{type}\, e_2}\ v_1\ v_2 \doteq v}{R, b \vdash e_1 \oplus e_2 \downarrow \langle v \rangle} \qquad \frac{H, bs \vdash e_1 \Downarrow \langle\ \rangle \quad H, bs \vdash e_2 \Downarrow \langle\ \rangle}{R, b \vdash e_1 \oplus e_2 \downarrow \langle\ \rangle}$$

$$\frac{R, b \vdash e \downarrow \langle v \rangle \quad R(x) = \langle v_x \rangle \quad \mathsf{val\text{-}to\text{-}bool}\, v_x \doteq k}{R, b \vdash e\ \mathtt{when}\ (x = k) \downarrow \langle v \rangle}$$

$$\frac{R, b \vdash e \downarrow \langle v \rangle \quad R(x) = \langle v_x \rangle \quad \mathsf{val\text{-}to\text{-}bool}\, v_x \doteq \neg k}{R, b \vdash e\ \mathtt{when}\ (x = k) \downarrow \langle\ \rangle} \qquad \frac{R, b \vdash e \downarrow \langle\ \rangle \quad R(x) = \langle\ \rangle}{R, b \vdash e\ \mathtt{when}\ (x = k) \downarrow \langle\ \rangle}$$

(a) Expressions (sem_exp_instant, src/CoreExpr/CESemantics.v:93)

$$\frac{R(x) = \langle \mathsf{T} \rangle \quad R, b \vdash_c e_t \downarrow \langle v \rangle \quad R, b \vdash_c e_f \downarrow \langle\ \rangle}{R, b \vdash_c \mathtt{merge}\ x\ e_t\ e_f \downarrow \langle v \rangle}$$

$$\frac{R(x) = \langle \mathsf{F} \rangle \quad R, b \vdash_c e_t \downarrow \langle\ \rangle \quad R, b \vdash_c e_f \downarrow \langle v \rangle}{R, b \vdash_c \mathtt{merge}\ x\ e_t\ e_f \downarrow \langle v \rangle}$$

$$\frac{R(x) = \langle\ \rangle \quad R, b \vdash_c e_t \downarrow \langle\ \rangle \quad R, b \vdash_c e_f \downarrow \langle\ \rangle}{R, b \vdash_c \mathtt{merge}\ x\ e_t\ e_f \downarrow \langle\ \rangle}$$

$$\frac{R, b \vdash e \downarrow \langle \mathsf{T} \rangle \quad R, b \vdash_c e_t \downarrow \langle v_t \rangle \quad R, b \vdash_c e_f \downarrow \langle v_f \rangle}{R, b \vdash_c \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \downarrow \langle v_t \rangle}$$

$$\frac{R, b \vdash e \downarrow \langle \mathsf{F} \rangle \quad R, b \vdash_c e_t \downarrow \langle v_t \rangle \quad R, b \vdash_c e_f \downarrow \langle v_f \rangle}{R, b \vdash_c \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \downarrow \langle v_f \rangle}$$

$$\frac{R, b \vdash e \downarrow \langle\ \rangle \quad R, b \vdash_c e_t \downarrow \langle\ \rangle \quad R, b \vdash_c e_f \downarrow \langle\ \rangle}{R, b \vdash_c \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \downarrow \langle\ \rangle} \qquad \frac{R, b \vdash e \downarrow s}{R, b \vdash_c e \downarrow s}$$

(b) Control expressions (sem_cexp_instant, src/CoreExpr/CESemantics.v:143)

Figure 2.8 (I): Instantaneous semantics of NLustre

$$\frac{}{R, b \vdash \bullet \downarrow b} \qquad \frac{R, b \vdash ck \downarrow \mathsf{true} \quad R(x) = \langle v \rangle \quad \mathsf{val\text{-}to\text{-}bool}\, v \doteq k}{R, b \vdash ck \text{ on } (x = k) \downarrow \mathsf{true}}$$

$$\frac{R, b \vdash ck \downarrow \mathsf{true} \quad R(x) = \langle v \rangle \quad \mathsf{val\text{-}to\text{-}bool}\, c \doteq \neg k}{R, b \vdash ck \text{ on } (x = k) \downarrow \mathsf{false}} \qquad \frac{R, b \vdash ck \downarrow \mathsf{false} \quad R(x) = \langle\ \rangle}{R, b \vdash ck \text{ on } (x = k) \downarrow \mathsf{false}}$$

(c) Clocks (sem_clock_instant, src/CoreExpr/CESemantics.v:65)

$$\frac{R, b \vdash_{(c)} e \downarrow \langle v \rangle \quad R, b \vdash ck \downarrow \mathsf{true}}{R, b \vdash_{(c)} e :: ck \downarrow \langle v \rangle} \qquad \frac{R, b \vdash_{(c)} e \downarrow \langle\ \rangle \quad R, b \vdash ck \downarrow \mathsf{false}}{R, b \vdash_{(c)} e :: ck \downarrow \langle\ \rangle}$$

(d) Clocked (control) expressions (sem_annotated_instant, src/CoreExpr/CESemantics.v:189)

Figure 2.8 (II): Instantaneous semantics of NLustre

As explained before, the semantics of NLustre relies on clock annotations at equation level to ensure clocking constraints. Figure 2.8c shows the instantaneous semantics of clocks. The base clock $\bullet$ always evaluates to the current value b of the base stream clock. The semantics for sampled clocks ensures that the condition variable x is present if and only if the sampled clock ck is recursively evaluated to true. The sampled clock is then evaluated to false when ck is false and to true only if ck is true and the condition on x is met.

The semantics for expressions and clocks are combined in figure 2.8d to ensure instantaneous clock alignment between a (control) expression and its clock annotation.

2.3.6.2 Stream semantics

Once the instantaneous semantics is defined, we can derive the stream semantics by a simple generalization. Unlike the coinductive semantics, the history H we use here is not an environment of streams but a stream of environments. Again, this choice follows from the *slicing* idea described before: having a stream of instantaneous environments facilitates instant-by-instant reasoning on a set of equations.

We lift the previous definitions as follow:

$$H, bs \vdash_{(c)} e \Downarrow s \triangleq \forall n,\ H\ n, bs\ n \vdash_{(c)} e \downarrow s\ n$$
$$H_*(x) \triangleq \lambda\, n.\ (H\ n)(x)$$

Thus, we write $H, bs \vdash_{(c)} e \Downarrow s$ to mean that under the stream of environments H and base clock stream bs the (control) expression e is associated with the stream s. The notation $H_*(x)$ represents the stream of successive values associated to x in the stream of environments H. As for other lookup notations, we use it for clarity of the presentation while ignoring partiality.

$$\frac{H, bs \vdash_c e :: ck \downarrow\downarrow H_*(x)}{G, H, bs \vdash x =_{ck} e} \qquad \frac{H, bs \vdash e :: ck \downarrow\downarrow s \quad H_*(x) \approx \mathsf{fby}\, [\![c]\!] s}{G, H, bs \vdash x =_{ck} c \,\mathtt{fby}\, e}$$

$$\frac{H, bs \vdash \boldsymbol{e} \downarrow\downarrow xs \quad H, bs \vdash ck \downarrow\downarrow \mathsf{base\text{-}of}\, xs \quad G \vdash f\,(xs) \downarrow\downarrow H_*(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})}$$

$$\frac{\begin{array}{c} H, bs \vdash \boldsymbol{e} \downarrow\downarrow xs \quad H, bs \vdash ck \downarrow\downarrow \mathsf{base\text{-}of}\, xs \\ \mathsf{bools\text{-}of}\,(H_*(y)) \doteq rs \quad \forall k,\ G \vdash f\left(\mathsf{mask}^k_{rs}\, xs\right) \downarrow\downarrow \mathsf{mask}^k_{rs}\, H_*(\boldsymbol{x}) \end{array}}{G, H, bs \vdash \boldsymbol{x} =_{ck} (\mathtt{restart}\ f\ \mathtt{every}\ y)\,(\boldsymbol{e})}$$

(a) Equations (sem_equation, src/NLustre/NLIndexedSemantics.v:70)

$$\frac{\begin{array}{c} \mathsf{node}(G, f) \doteq n \quad H_*(\boldsymbol{x}) \approx xs \quad H_*(\boldsymbol{y}) \approx ys \\ \mathsf{respects\text{-}clock}\, H\ bs\ \Gamma \quad \forall eq \in n.\mathbf{eqs},\ G, H, bs \vdash eq \end{array}}{G \vdash f\,(xs) \downarrow\downarrow ys} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \\ \Gamma = \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\} \\ bs = \mathsf{base\text{-}of}\, xs \end{array}$$

(b) Nodes (sem_node, src/NLustre/NLIndexedSemantics.v:99)

Figure 2.9: Indexed semantics of NLustre

Figure 2.9a gives the semantics for equations. The structure of the rules is all but identical to that of the coinductive semantics rules presented in figure 2.7e on page 53. The main difference is that the semantics of a node does not use lists of streams of values but instead streams of lists of values. Again this choice reflects the instant-by-instant behavior that we model: slicing a stream of lists is simpler than slicing every stream of a list of streams. Therefore the overloadings of notations for lists do not have the same meanings as before. While $H, bs \vdash \boldsymbol{e} \Downarrow \boldsymbol{xs}$ was a shorthand for $\forall i,\ H, bs \vdash e_i \Downarrow xs_i$, now $H, bs \vdash \boldsymbol{e} \downarrow\downarrow xs$ is a shorthand for $\forall n\, i,\ H\ n, bs\ n \vdash e_i \downarrow (xs\ n)_i$. Similarly, while we wrote $H(\boldsymbol{x})$ as a shorthand for the list of streams $H(x_1) \cdots H(x_i)$, we now write $H_*(\boldsymbol{x})$ as a shorthand for the stream of lists $\lambda\, n.\ (H\ n)(x_1) \cdots (H\ n)(x_i)$. Working in a proof assistant like Coq helps to track such tedious but inevitable details.

The operators base-of, bools-of and fby are re-defined. The first two are equivalent to their coinductive counterparts: base-of computes the compound clock stream which is true whenever at least one element of the given list is present and bools-of projects a stream of values into a stream of Coq booleans.

Definition 2.3.5 (clock_of, src/CoreExpr/CESemantics.v:281)

$$\begin{aligned}\textsf{base-of-now}\ \boldsymbol{v} &\triangleq \textsf{exists}_{\mathbb{B}}\ (\langle\ \rangle \neq_{\mathbb{B}})\ \boldsymbol{v}\\ \textsf{base-of}\ vs &\triangleq \lambda\, n.\ \textsf{base-of-now}\ (vs\ n)\end{aligned}$$

Definition 2.3.6 (bools_of, src/CoreExpr/CESemantics.v:284)

$$\textsf{bools-of}\ xs \doteq rs \leftrightarrow \forall n,\ \mathit{if}\ rs\ n\ \mathit{then}\ (xs\ n = \langle \mathsf{T} \rangle)\ \mathit{else}\ (xs\ n = \langle \mathsf{F} \rangle \vee xs\ n = \langle\ \rangle)$$

Compare with definitions 2.2.2 and 2.2.4 on page 38 and on page 41.

The new fby definition is more intricate than the straightforward coinductive definition 2.3.3 on page 51 and uses an auxiliary function hold.

Definition 2.3.7 (fby, src/NLustre/NLIndexedSemantics.v:59)

$$\begin{aligned}\textsf{hold}\ v_0\ vs\ 0 &\triangleq v_0\\ \textsf{hold}\ v_0\ vs\ (n+1) &\triangleq \mathit{if}\ (vs\ n = \langle v \rangle)\ \mathit{then}\ v\ \mathit{else}\ \textsf{hold}\ v_0\ vs\ n\\ \textsf{fby}\ v_0\ vs &\triangleq \lambda\, n.\ \mathit{if}\ (vs\ n = \langle\ \rangle)\ \mathit{then}\ \langle\ \rangle\ \mathit{else}\ \langle \textsf{hold}\ v_0\ vs\ n \rangle\end{aligned}$$

Basically the `fby` definition just describes the reaction to absence and presence, the actual recursive behavior is implemented by the hold function. In fact, this hold function describes the content of a *memory register*: indeed it could be seen as yielding the stream of successive values—without presence nor absence—that a register corresponding to a **`fby`**-defined variable will take. Its behavior is to output its first argument at the first instant, then at each instant the previous present value of its second argument. If the previous value is absent, then the register just *holds* its value, without updating it. The following chronogram gives a visual intuition.

y			2	4	6			8	10	$\cdots$
hold $[\![1]\!]$ y	1	1	1	2	4	6	6	6	8	$\cdots$
fby $[\![1]\!]$ $y \doteq$			1	2	4			6	8	$\cdots$
1 `fby` y			1	2	4			6	8	$\cdots$

The auxiliary definitions introduced for the modular reset, explained in section 2.2.2.3, are adapted in the indexed model. This means rewriting the mask operator, this time in a more intuitive way than for its coinductive definition 2.2.3 on page 40.

Definition 2.3.8 (mask, src/IndexedStreams.v:217)

$$\begin{aligned}\mathsf{count}\ rs\ n &\triangleq \mathit{if}\ rs\ n\ \mathit{then}\ c+1\ \mathit{else}\ c\\ &\qquad \mathit{where}\ c = \mathit{if}\ (n=0)\ \mathit{then}\ 0\ \mathit{else}\ \mathsf{count}\ rs\ (n-1)\\ \mathsf{mask}^k_{rs}\ xs &\triangleq \lambda\, n.\ \mathit{if}\ (\mathsf{count}\ rs\ n = k)\ \mathit{then}\ xs\ n\ \mathit{else}\ \mathsf{absent\text{-}list}\ \|xs\ 0\|\end{aligned}$$

Where $\mathsf{absent\text{-}list}\ i \triangleq \underbrace{\langle\ \rangle \cdots \langle\ \rangle}_{i}$

The definition uses the auxiliary count function which calculates the cumulative sum of the values on the rs clock stream. This auxiliary function encodes the intuition that was given to explain the coinductive version of mask (see the associated chronogram page 40). The subtlety compared with the coinductive version of mask is the output in the case where $k \neq \mathsf{count}\ rs\ n$: here we deal with streams of lists instead of lists of streams, as explained before. The *opaque* output must thus be a list of absent values. Since this operator is only used in the context of the semantics rules, we know that the stream of lists xs has the same length at each instant. That is, we have:

$$\forall i\ j,\ \|xs\ i\| = \|xs\ j\|$$

Thus we arbitrarily choose the length of the list at the first instant. We could have taken the length at the current instant n but that is slightly less convenient in the proofs.

Finally, figure 2.9b gives the semantics of nodes: again, there are no structural differences with the coinductive version. The coinductive definition 2.3.4 on page 54 of respects-clock is re-defined as follow:

Definition 2.3.9 (sem_clocked_var, src/CoreExpr/CESemantics.v:233)

$$\begin{aligned}\mathsf{respects\text{-}clock\text{-}now}\ R\ b\ x^{ck} &\triangleq R, b \vdash ck \downarrow \mathit{true} \leftrightarrow \exists v,\ R(x) = \langle v \rangle\\ &\qquad \wedge\ R, b \vdash ck \downarrow \mathit{false} \leftrightarrow R(x) = \langle\ \rangle\\ \mathsf{respects\text{-}clock}\ H\ bs &\triangleq \forall n,\ \mathsf{respects\text{-}clock\text{-}now}\ (H\ n)\ (bs\ n)\end{aligned}$$

2.4 Relating the coinductive and indexed semantics

Now that NLustre is given two different semantics, a coinductive one for the correctness of the transcription pass and an indexed one for further compilation, we have to show

that the former is included in the latter. First we have to set up a way of comparing coinductive streams with indexed streams. We do so by defining a straightforward notion of equivalence between the two kinds of streams.

Definition 2.4.1 (Observational equivalence)
A coinductive stream u is observationally equivalent *to an indexed stream v, written $u \sim v$, if and only if*

$$\forall n,\ u_n = v\ n$$

We also need a version for comparing lists of coinductive streams with indexed streams of lists.

Definition 2.4.2 (Generalized observational equivalence)
A list of coinductive streams $\boldsymbol{u}$ is observationally equivalent to an indexed stream of lists v, written $\boldsymbol{u} \sim v$, if and only if

$$\forall n\, k,\ (\boldsymbol{u}_k)_n = (v\ n)_k$$

Then we want to prove that the indexed semantics of NLustre is included in the coinductive semantics.

Lemma 2.4.1
Given a program G, a name f, two lists of coinductive streams of values $\boldsymbol{xs}$ and $\boldsymbol{ys}$ such that $G \vdash f\ (\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$, then there exist two indexed streams of lists of values xs' and ys' such that $G \vdash f\ (xs') \downdownarrows ys'$, $\boldsymbol{xs} \sim xs'$, and $\boldsymbol{ys} \sim ys'$.

Even if the result is not needed in our compilation correctness proof, we also show the converse: the coinductive semantics of NLustre is included in the indexed semantics.

Lemma 2.4.2
Given a program G, a name f, two streams of lists of values xs and ys such that $G \vdash f\ (xs) \downdownarrows ys$, then there exist two lists of coinductive streams of values $\boldsymbol{xs'}$ and $\boldsymbol{ys'}$ such that $G \vdash f\ (\boldsymbol{xs'}) \Downarrow \boldsymbol{ys'}$, $\boldsymbol{xs'} \sim xs$, and $\boldsymbol{ys'} \sim ys$.

We describe the proofs over the following two sections.

2.4.1 From the coinductive semantics to the indexed semantics

The inclusion result is presented in lemma 2.4.1 in an existential manner. For the proof, we define a function that translates coinductive streams into indexed streams and use it to provide witnesses for the existential variables. The function ensures the observational equivalence, and we show that it is a morphism with respect to the two semantics.

First we define the function that translates a single coinductive stream into an indexed stream.

Definition 2.4.3 (`tr_Stream`, src/NLustre/NLCoindToIndexed.v:65)

$$\mathsf{to\text{-}idx}\ s \triangleq \lambda\, n.\ s_n$$

We lift this definition to the translation of lists of coinductive streams into streams of lists.

Definition 2.4.4 (tr_Streams, src/NLustre/NLCoindToIndexed.v:69)

$$\textsf{to-idx}\ \boldsymbol{s} \triangleq \lambda\, n.\ (\textsf{to-idx}\, x\ n)_{x \in \boldsymbol{s}}$$

Finally, we propose a function hist-to-idx for translating a history. This function transforms an environment of coinductive streams of values into an indexed stream of environments of values. Building such a stream is straightforward: at each instant, we extract a snapshot of the history by indexing all streams appearing in it. The definition can be written as follows, using function composition, if we represent the environments as functions.[4]

Definition 2.4.5 (tr_History, src/NLustre/NLCoindToIndexed.v:75)

$$\textsf{hist-to-idx}\, H \triangleq \textsf{to-idx} \circ H$$

We must now show that the semantics is preserved by our family of functions. In other words, we must show that these functions are morphisms with respect to the two semantics, for each syntactic class of NLustre: expressions, control expressions, equations and nodes. Eventually, the main result states that **to-idx** is a morphism with respect to the two semantics for a node.

Theorem 2.4.3 (implies, src/NLustre/NLCoindToIndexed.v:660)
Given a program G, a name f, two lists of coinductive streams of values $\boldsymbol{xs}$ and $\boldsymbol{ys}$ such that $G \vdash f\ (\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$, then

$$G \vdash f\ (\textbf{to-idx}\ \boldsymbol{xs}) \downdownarrows \textbf{to-idx}\ \boldsymbol{ys}$$

The proof follows by mutual induction over nodes and equations, using intermediate results about expressions and control expressions. In the following, we detail a complication that seems unavoidable and some results of interest.

The technical complication comes from the different *nature* of the hypotheses and of the goals in the proofs of some intermediate results. Consider the following intermediate result stating the semantics preservation for expressions.

Lemma 2.4.4 (sem_exp_impl, src/NLustre/NLCoindToIndexed.v:277)
Given a coinductive history H, a base clock stream bs, an expression e and a coinductive stream s such that $H, bs \vdash e \Downarrow s$, then $\textsf{hist-to-idx}\, H, \textsf{to-idx}\, bs \vdash e \downdownarrows \textsf{to-idx}\, s$.

In the coinductive semantics, e is evaluated using coinductive operators like the synchronous streams operator when, for example. As we saw in section 2.2.2.1, such operators constrain their input and output streams by constraining their heads and then passing coinductively into their tails. However, we saw that the indexed semantics is a lifting of an

[4] Internally, as mentioned already, environments are modeled as trees, but in this dissertation we prefer the representation as functions for convenience.

```
Lemma when_spec:
  forall k xs cs rs,
    when k xs cs rs <->
    (forall n,
        (xs # n = absent
         /\ cs # n = absent
         /\ rs # n = absent)
        \/
        (exists x c,
            xs # n = present x
            /\ cs # n = present c
            /\ val_to_bool c = Some (negb k)
            /\ rs # n = absent)
        \/
        (exists x c,
            xs # n = present x
            /\ cs # n = present c
            /\ val_to_bool c = Some k
            /\ rs # n = present x)).
```

Coq (src/CoindStreams.v:479-497)

Listing 2.7: Inversion result for the coinductive when operator

instantaneous behavior. Eventually, we are left with a coinductive hypothesis specifying *coiteratively* the semantics of e while we have to show that the instantaneous behavior of e holds for all instants n. In Coq, we cannot directly do a proof by coinduction here, since the result we want to prove is not coinductive, only the hypothesis is. I solved this problem by stating intermediate results giving a generalized instantaneous equivalent specification of each coinductive synchronous stream operator and judgment. These results can be seen as independent inversion lemmas. For example, consider the Coq lemma presented in listing 2.7: it simply expresses the coinductive behavior of the `when` operator in terms of instantaneous behavior. It states that the relation `when k xs cs rs` is equivalent to the given instant-by-instant specification: at each instant `n`, either the three streams are absent, or the value of the output stream `rs` is given by sampling `xs` according to `cs`. It is proved by induction on `n` without any difficulty and can be used as an inversion result in the proof of lemma 2.4.4, by induction over the kinds of expressions.

Interestingly, as the fby operator is a total function in both semantics, we can directly state the following result.

Lemma 2.4.5 (fby_impl, src/NLustre/NLCoindToIndexed.v:374)
Given a coinductive stream of values s and a constant v_0,

$$\mathsf{to\text{-}idx}\,(C.\mathsf{fby}\,v_0\;s) \approx I.\mathsf{fby}\,v_0\;\;(\mathsf{to\text{-}idx}\,s)$$

where $C.$fby is the Coinductive operator, and $I.$fby the Indexed one.

Another point of interest is the conversion of the mask operator, involved in the proof of preservation for the semantics of the modular reset. Recall the intricacy of the coinductive

definition of mask, given in definition 2.2.3 on page 40. This kind of coinductive definition is rather unpractical in the proofs, so the idea is to give the operator a specification following the intuitive behavior, as does its definition in the indexed model. The point is to define a coinductive version of count and use it to state the specification.

Definition 2.4.6 (count, src/CoindStreams.v:392)

$$\begin{aligned}\mathsf{count\text{-}from}\ c\ (r \cdot rs) &\triangleq c' \cdot \mathsf{count\text{-}from}\ c'\ rs \\ &\qquad \textit{where } c' = \textit{if } r \textit{ then } c+1 \textit{ else } c \\ \mathsf{count}\ rs &\triangleq \mathsf{count\text{-}from}\ 0\ rs\end{aligned}$$

We define count-from to count coinductively the number of true ticks, starting from an initial number. Then count is a simple specialization that we can use to express the specification of the coinductive mask operator.

Lemma 2.4.6 (mask_nth, src/CoindStreams.v:419)
Given a natural number k, a boolean coinductive stream rs and a coinductive stream of values s, then for all instants n,

$$\left(\mathsf{mask}^k_{rs}\ s\right)_n = \textit{if } \left((\mathsf{count}\ rs)_n = k\right) \textit{ then } xs_n \textit{ else } \langle\,\rangle$$

This specification result is very similar to the indexed definition 2.3.8 on page 60. Now we use this similarity to show the following correspondence.

Lemma 2.4.7 (mask_impl, src/NLustre/NLCoindToIndexed.v:561)
Given a natural integer k, a boolean coinductive stream rs and list of coinductive streams of values $\boldsymbol{xs}$,

$$\mathbf{to\text{-}idx}\left(C.\mathsf{mask}^k_{rs}\ s\right)_{s\in\boldsymbol{xs}} \approx I.\mathsf{mask}^k_{\mathsf{to\text{-}idx}\ rs}\ (\mathbf{to\text{-}idx}\ \boldsymbol{xs})$$

2.4.2 From the indexed semantics to the coinductive semantics

The proof of semantics inclusion in the other direction essentially follows the same structure as the preceding proof: we define morphisms to pass from the indexed model to the coinductive model. We first define a function that translates an indexed stream into a coinductive stream.[5]

Definition 2.4.7 (tr_stream, src/NLustre/NLIndexedToCoind.v:74)

$$\begin{aligned}\mathsf{to\text{-}coind\text{-}from}\ n\ s &\triangleq s\ n \cdot \mathsf{to\text{-}coind\text{-}from}\ (n+1)\ s \\ \mathsf{to\text{-}coind}\ s &\triangleq \mathsf{to\text{-}coind\text{-}from}\ 0\ s\end{aligned}$$

Building a list of coinductive streams from a stream of lists is more intricate than in the original direction. This is expected: while passing from lists of coinductive streams to indexed streams of lists is simply *slicing*, we need a well-formedness property on the length of the lists to go the other way. The idea follows three steps:

[5] Implemented in Coq with a `CoFixpoint`.

1. find a way to extract each kth indexed stream from a stream of lists,
2. translate each of those obtained streams into coinductive streams, and,
3. gather them into the resulting list.

The first step is to define the following function to extract the kth stream from a stream of lists s.

Definition 2.4.8 (streams_nth, src/NLustre/NLIndexedToCoind.v:82)

$$\mathsf{streams\text{-}nth}\ k\ s \triangleq \lambda\, n.\ (s\ n)_k$$

The subtlety in Coq is that the indexing operation on a list will fail if the index is out of bounds, so we have to provide a default value for the operation to be total. This detail is ignored in this dissertation but in the development we give up polymorphism for simplicity and specify the absent value as the default.

Secondly, we can combine to-coind-from and streams-nth to define a function which translates the kth stream of a stream of lists s into a coinductive stream.

Definition 2.4.9 (nth_tr_streams_from, src/NLustre/NLIndexedToCoind.v:87)

$$\mathsf{nth\text{-}to\text{-}coind\text{-}from}\ n\ s\ k \triangleq \mathsf{to\text{-}coind\text{-}from}\ n\ \ (\mathsf{streams\text{-}nth}\ k\ s)$$

Finally, we need a way to build a list of each kth translated stream. We begin by defining a function that builds a list of streams by iterating a function over a range of integers $[0, m)$.[6]

Definition 2.4.10 (seq_streams, src/NLustre/NLIndexedToCoind.v:77)

$$\mathsf{seq\text{-}streams}\ f\ m \triangleq (f\ k)_{k\in[0,m)}$$

Then we can use seq-streams with nth-to-coind-from as an iterator function to build the function. We need to provide an upper bound m for the integer range, so we use the length of the list in the stream at a given instant n. As mentioned, we will need the well-formedness property on the lengths of the lists in the stream s:

$$\forall i\ j,\ \|s\ i\| = \|s\ j\|$$

Definition 2.4.11 (tr_streams, src/NLustre/NLIndexedToCoind.v:97)

$$\mathbf{to\text{-}coind\text{-}from}\ n\ s \triangleq \mathsf{seq\text{-}streams}\ (\mathsf{nth\text{-}to\text{-}coind\text{-}from}\ n\ s)\ \ \|s\ n\|$$
$$\mathbf{to\text{-}coind}\ s \triangleq \mathbf{to\text{-}coind\text{-}from}\ 0\ s$$

[6] In Coq, we use the standard library function seq.

The function for translating a history, that is, a stream of environments, into an environment of streams is also not trivial to define in Coq: we must reconstruct (1) coinductive streams from values found in *sliced* instantaneous environments, and (2) a domain of definition from *sliced* domains. The idea to solve the second point is to rely on the semantics constraints to ensure that the domains remain consistent at each instant. This way it suffices to use the domain of a given environment at a given instant, for example, the very first instant $n = 0$. The first problem is solved by performing lookups at each instant into the corresponding environment: here again thanks to the semantics constraints, we can deduce that the lookup cannot fail. In this dissertation, these technical details are hidden behind the convenient function composition notation.

Definition 2.4.12 (tr_history_from, src/NLustre/NLIndexedToCoind.v:105)

$$\mathsf{hist\text{-}to\text{-}coind\text{-}from}\ n\ H \triangleq (\mathsf{to\text{-}coind\text{-}from}\ n) \circ H_*$$
$$\mathsf{hist\text{-}to\text{-}coind}\ H \triangleq \mathsf{hist\text{-}to\text{-}coind\text{-}from}\ 0\ H$$

The structure of the proof is essentially the converse of that of section 2.4.1. We show that the previous functions are morphisms with respect to the two semantics, for each syntactic class and arrive at the following result: **to-coind** is a morphism with respect to the two semantics of a node.

Theorem 2.4.8 (implies, src/NLustre/NLIndexedToCoind.v:1116)
Given a program G, a name f, two streams of lists of values xs and ys such that $G \vdash f\,(xs) \downdownarrows ys$, then

$$G \vdash f\,(\textbf{to-coind}\ xs) \Downarrow \textbf{to-coind}\ ys$$

As before, the proof follows by mutual induction on equations and nodes, using intermediate results on expressions and control expressions, sometimes requiring proofs by coinduction. In the following we outline some difficulties and interesting results.

To prove some intermediate results we must solve a difficulty, arguably more involved than for the converse proof. Consider again the intermediate inclusion result for expressions.

Lemma 2.4.9 (sem_exp_impl, src/NLustre/NLIndexedToCoind.v:630)
Given an environment stream H, a base clock stream bs, an expression e and an indexed stream of values s such that $H, bs \vdash e \downdownarrows s$, then $\mathsf{hist\text{-}to\text{-}coind}\ H, \mathsf{to\text{-}coind}\ bs \vdash e \Downarrow \mathsf{to\text{-}coind}\ s$.

This time when doing an induction on the expression e, we are left with some coinductive sub-goals, while the hypotheses are not coinductive. Precisely, recall that the hypothesis $H, bs \vdash e \downdownarrows s$ actually stands for $\forall n,\ H\ n, bs\ n \vdash e \downarrow s\ n$. Thus when dealing with constructs like the **`when`**, for example, one cannot perform inversion (elimination) on the hypothesis directly. In other words, from the indexed semantics of an expression `e` **`when`** `x`, one cannot exhibit the indexed streams associated with the semantics of the sub-expressions `e` and `x`. Yet, to pass the induction, we actually need these streams, so that we can use the induction hypothesis on the sub-expressions. Intuitively, it seems obvious that exhibiting sub-streams from the semantics of an expression is possible, but

```
Lemma when_inv:
  forall H b e x k es,
    CESem.sem_exp b H (Ewhen e x k) es ->
    exists ys xs,
      CESem.sem_exp b H e ys
      /\ CESem.sem_var H x xs
      /\
      (forall n,
          (exists sc xc,
              val_to_bool xc = Some k
              /\ ys n = present sc
              /\ xs n = present xc
              /\ es n = present sc)
          \/
          (exists sc xc,
              val_to_bool xc = Some (negb k)
              /\ ys n = present sc
              /\ xs n = present xc
              /\ es n = absent)
          \/
          (ys n = absent
           /\ xs n = absent
           /\ es n = absent)).
```

Coq (src/NLustre/NLIndexedToCoind.v:417-439)

Listing 2.8: Inversion result for the semantics of the **when** construct

the practical details are not trivial. Consider, for example, the particular intermediate result we want to prove for the semantics of the **when** construct, shown in listing 2.8. The lemma simply states in an inversion-like fashion the specification of the behavior of the constructs as described by the instantaneous semantics. That is, if we know that the indexed semantics for the expression `Ewhen e x k` associates a stream `es` to it, then we can exhibit two indexed sub-streams `ys` and `xs` that are associated by the semantics to the sub-expressions `e` and `x` respectively. Moreover, these streams are constrained according to the semantics for the **when** operator. To prove this intermediate lemma, we would like to have the following general result:

$$\begin{aligned}&\forall H\ bs\ e\ s,\\&\quad (\forall n,\ \exists v,\ H\ n, bs\ n \vdash e \downarrow v) \rightarrow \exists s,\ \forall n,\ H\ n, bs\ n \vdash e \downarrow s\ n\end{aligned}$$

That is, if the instantaneous semantics associates a value to an expression e at each instant, then there exists a stream s that is associated to e by the stream semantics. This is clearly an instance of the axiom of functional choice:

$$\forall R,\ (\forall x,\ \exists y,\ R\ x\ y) \rightarrow \exists f,\ \forall x,\ R\ x\ (f\ x)$$

There are two obvious solutions to this goal. The first is to apply the axiom of functional choice, but, in Coq, this means importing a non-trivial axiom, as it is not a property of the core theory. We preferred not to do this. The second solution is to explicitly describe how to calculate the instantaneous values needed to build the witness stream. In other words, the solution is to *construct* the choice function, which is what we did. It is possible to design an interpreter for NLustre, at least for the expressions. This interpreter precisely follows the instantaneous semantics rules described in section 2.3.6.1 to produce instantaneous values—the absent value being used as default value to make it total. The interpreter is proved complete, in the sense that it produces the value expected by the semantics. For example, consider the instantaneous interpreter $\mathsf{interp\text{-}now}_e$ that produces values for expressions, we show that the instantaneous interpreter for expressions is correct.

Lemma 2.4.10 (interp_exp_instant_complete, src/CoreExpr/CEInterpreter.v:173)
Given a value environment R, a base clock b, an expression e and a value v such that $R, b \vdash e \downarrow v$, then $v = \mathsf{interp\text{-}now}_e\ R\ b\ e$.

We lift the instantaneous interpreter to define an interpreter giving streams of values. For example, we can define the following stream interpreter for expressions:

Definition 2.4.13 (interp_exp, src/CoreExpr/CEInterpreter.v:273)

$$\mathsf{interp}_e\ H\ bs\ e \triangleq \lambda n.\ \mathsf{interp\text{-}now}_e\ (H\ n)\ (bs\ n)\ e$$

Now we can use this interpreter as a choice function to prove the result expressed by listing 2.8 without relying on an external axiom. The same technique works to obtain similar results for the other operators and to complete the proof of lemma 2.4.9.

Interestingly, for the fby operator, the correspondence result which is the converse of the one presented in the previous section, relies on the following intermediate result where the indexed hold operator appears.

Lemma 2.4.11 (fby_impl_from, src/NLustre/NLIndexedToCoind.v:819)
Given an indexed stream of values s and a constant v_0, then, for all instants n,

$$\mathsf{to\text{-}coind\text{-}from}\ n\ (I.\mathsf{fby}\ v_0\ s) \equiv C.\mathsf{fby}\ (\mathsf{hold}\ v_0\ s\ n)\ (\mathsf{to\text{-}coind\text{-}from}\ n\ s)$$

This result emphasizes the fact that the hold function is fundamental to both semantics: it explicitly encodes the *register*-like nature of the `fby` operator.

Chapter 3

From dataflow nodes to transition systems

Contents

Once a Lustre program is in normalized form, that is, translated into NLustre, the compilation to imperative code follows relatively straightforward techniques. The challenge in a verified compiler is to reason formally about the change from a dataflow model constraining streams to an imperative model that describes a sequence of calculations and memory updates.

We introduce a novelty in the usual modular compilation scheme of Lustre. Before being translated to imperative code, the NLustre program is transformed into an intermediate transition system language, called Stc, for Synchronous Transition Code. This language is designed to allow better code optimization when the modular reset is used.

3.1 Motivation

3.1.1 Syntactic granularity

The next step in the modular compilation approach is normally to schedule the dataflow equations and then to generate imperative code directly from NLustre. To explain why we introduce a new intermediate language, we anticipate a little the next chapter. The idea of the compilation scheme is to associate each node of a program with a persistent state and *methods* that act on its state: (1) a *reset* method that is responsible for setting the

```
node driver(gps, xv, yv: float64; r: bool)
  returns (x, y: float64);
  var alarmx, alarmy: bool;
let
  x, alarmx = (restart ins every r)(gps, xv);
  y, alarmy = (restart ins every r)(gps, yv);
tel
```

NLustre

(a) NLustre example with two resets

```
step(gps, xv, yv: float64, r: bool)
  returns (x, y: float64)
  var alarmx, alarmy: bool
{
  if r { ins(x).reset(); } else { };
  x, alarmx := ins(x).step(gps, xv);
  if r { ins(y).reset(); } else { };
  y, alarmy := ins(y).step(gps, yv);
}
```

Obc

(b) Direct translation into Obc

```
if r { ins(x).reset() } else { };
if r { ins(y).reset() } else { };
y, alarmy := ins(y).step(gps, yv);
x, alarmx := ins(x).step(gps, xv)
```

Obc

(c) Better desired code

```
if r {
  ins(x).reset();
  ins(y).reset();
} else { };
x, alarmx := ins(x).step(gps, xv);
y, alarmy := ins(y).step(gps, yv);
```

Obc

(d) Better optimized code

Figure 3.1: Scheduling and compiling the modular reset

initial state of the node, and (2) a *step* method that implements a step of the synchronous execution of the node, updating the state of the node for the next instant. Scheduling is necessary to sequentialize correctly the generated imperative calculations.

Consider the node `driver` in figure 3.1a, that uses the `ins` node from our running example. The `ins` node is instantiated twice and both instances are reset by the variable `r`. If we generate Obc imperative code (presented in the next chapter) directly from this NLustre node, which is already well scheduled (there are no data dependencies between the equations), we obtain the Obc *step* method in figure 3.1b. Two instance variables that designate sub-states are introduced: `ins(x)` and `ins(y)`. Each node instantiation equation with reset is translated into a sequence of two instructions: a guarded call to the *reset* method, then a call to the *step* method of the corresponding instance. The problem is that this code is not optimal since the conditional calls to the *reset* methods cannot be *fused* together. Indeed we would want the instructions to be sequentialized as in figure 3.1c, allowing further fusion optimization (described in the next chapter) to produce the code shown in figure 3.1d.

Our solution to this problem is the introduction of an intermediate language where the modular resets appear independently from instantiations, while keeping a declarative presentation. The idea is to have a semantics that still does not depend on the order of equations while allowing a finer scheduling that takes resets into account.

3.1.2 Semantic granularity

Besides the performance issue raised by the compilation of the modular reset, we failed anyway to directly adapt the proofs between NLustre and Obc [Bourke, Brun, Dagand, et al. (2017)] to handle this construct. Intuitively, the reason is that the NLustre semantic models are not fine enough to allow establishing a correspondence to track the changes to the persistent state when a reset occurs. Indeed, while it is straightforward to describe in an imperative paradigm ("reset then step"), it is not trivial to model in a synchronous dataflow paradigm where reactions are *atomic* ("reset-step").

Thus the second fundamental goal of the new intermediate language is to describe what we call *transient states*, that is, states that a node may reach *during* a single time-step, but that are not observable from the outside.

3.2 From NLustre to Stc

The goal of the Stc language is to increase the granularity in both the syntax and semantics. In the syntax, it allows considering node instantiations and resets as separate constructs. In the semantics, it exposes transient states. The solution proposed here can be considered a proof-of-concept developed to solve our particular issues. But we would expect the model to generalize to support reasoning about modes and scheduling policies in the spirit of [Caspi, Colaço, et al. (2009)].

The dataflow semantics of NLustre is based on composing stream functions, and consuming and producing streams. In contrast, Stc is based on composing state transitions, and consuming and producing values. Given input values at a given instant, a transition imposes constraints between two states and determines output values. Individual transitions are either composed in sequence, encoding a reset followed by a step, or in parallel, to encode simultaneous constraints. Composite transitions are built-up recursively from transitions on local state memories. Each composite transition is synchronous: seen from outside, it is a single atomic constraint between two states.

Before presenting the syntax, we show in listing 3.1b the result of translating the NLustre running example recalled in listing 3.1a. The translation function will be detailed later. Note the appearance of the distinction between regular variables and state variables with the keywords **`init`** and **`next`**, the appearance of sub-instance variables with the keyword **`sub`**, and the separation of the node instantiation with reset of `ins` in the node `nav` into two distinct transition constraints. To simplify the translation and the proof, we avoid generating fresh identifiers, and instead reuse variable names from NLustre equations to name instances. In particular, remark how the single NLustre condition variable `r` is combined with the (implicit) clock of the source equation.

```
node euler(x0, u: float64) returns (x: float64);
var i: bool; px: float64;
let
  i = true fby false;
  x = if i then x0 else px;
  px = 0.0 fby (x + 0.1 * u);
tel

node ins (gps: float64, xv: float64) returns (x: float64, alarm: bool)
  var k: int32, px: float64, xe: float64 when not alarm;
let
  k = 0 fby k + 1;
  alarm = (k >= 50);
  xe = euler(gps when not alarm, xv when not alarm);
  x = merge alarm (px when alarm) xe;
  px = 0. fby x;
tel

node nav (gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
  var r: bool, c: bool, cm: bool, insr: float64 when not c, alr: bool when not c;
let
  (insr, alr) = (restart ins every r)(gps when not c, xv when not c);
  x = merge c (gps when c) insr;
  alarm = merge c (false when c) alr;
  cm = merge c (not s when c) (s when not c);
  c = true fby cm;
  r = false fby (s and c);
tel
```

NLustre

(a) Before

Listing 3.1 (I): Translation of the example

```
system euler {
  init i = true, px = 0.;
  transition(x0: float64, u: float64) returns (x: float64)
  {
    next i = false;
    x = if i then x0 else px;
    next px = x + 0.1 * u;
  }
}

system ins {
  init k = 0, px = 0.;
  sub xe: euler;
  transition(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64 when not alarm;
  {
    next k = k + 1;
    alarm = (k >= 50);
    xe = euler<xe,0>(gps when not alarm, xv when not alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}

system nav {
  init c = true, r = false;
  sub insr: ins;
  transition(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64 when not c, alr: bool when not c;
  {
    (insr, alr) = ins<insr,1>(gps when not c, xv when not c);
    reset ins<insr> every (. on r);
    x = merge c (gps when c) insr;
    alarm = merge c (false when c) alr;
    cm = merge c (not s when c) (s when not c);
    next c = cm;
    next r = s and c;
  }
}
```

Stc

(b) After

Listing 3.1 (II): Translation of the example

tc ::= **transition constraint**

$x =_{ck} ce$	(basic constraint)
`next` $x =_{ck} e$	(next constraint)
$x^*_{;} =_{ck} x\langle x,k\rangle(e^+_{;})$	(default transition)
`reset` $x\langle x\rangle$ `every` ck	(reset transition)

d ::= $x^{\tau,ck}$ **variable declaration**

s ::= **system**

```
system x {
  [sub (x^x)+;]
  [init (x^{c,ck})+;]
  transition(d+;) returns (d*;) [var d+;]
  {
    tc+;
  }
}
```

p ::= s^+ **program**

Figure 3.2: The Stc abstract syntax

3.2.1 Syntax of Stc

Figure 3.2 presents the syntax of Stc. Expressions and control expressions are shared with NLustre: this allows to simplify both the translation function and its proof of correctness. As explained, a system only possesses two *transitions*: an explicit default transition, which is defined by a list of *transition constraints*, and an implicit reset transition. In this language, state variables and instances are made explicit in the syntax through the **`init`** and **`sub`** declarations, respectively. Transition constraints, as for dataflow equations in NLustre, are annotated with their activation clock. Standard variables are constrained by basic constraints and state variables by **`next`** constraints. A default transition $\boldsymbol{x} =_{ck} f\langle i, k\rangle(\boldsymbol{e})$ constrains the state of the instance i declared with system f. The integer parameter k equals 1 if the list of transition constraints also contains a reset transition on the same instance i and 0 otherwise. Its precise role will be explained later. A state variable is declared at system level with its initial value as a constant, and a sub-system instance is declared with its system name.

Listing 3.2 shows the Coq implementation of systems. We do not show the implementation of transition constraints, as it is very similar to that of listing 2.6d on page 46 for NLustre equations. Again, we use a dependent record to model the syntactic elements of a system together with some syntactic invariants. As for NLustre, an Stc default transition must have at least one input (`s_ingt0`); the declarations of a system must be unique (`s_nodup` and `s_nodup_init_subs`); the sub-instance variables, state variables and non-input variables appearing in the transition constraints must be declared (`s_subs_calls_of`, `s_inits_in_tcs`, and `s_vars_out_in_tcs`); the reset transitions are

```
Record system :=
  System {
      s_name : ident;                            (* name *)
      s_in   : list (ident * (type * clock));    (* inputs *)
      s_vars : list (ident * (type * clock));    (* local variables *)
      s_inits: list (ident * (const * clock));   (* state variables *)
      s_subs : list (ident * ident);             (* sub-instances *)
      s_out  : list (ident * (type * clock));    (* outputs *)
      s_tcs  : list trconstr;                    (* transition constraints *)

      s_ingt0            : 0 < length s_in;

      s_nodup            : NoDup (map fst s_in ++ map fst s_vars ++
                                  map fst s_out ++ map fst s_inits);
      s_nodup_inits_subs : NoDup (map fst s_inits ++ map fst s_subs);

      s_subs_calls_of    : Permutation s_subs (calls_of s_tcs);

      s_inits_in_tcs     : Permutation (map fst s_inits) (inits_of s_tcs);
      s_vars_out_in_tcs  : Permutation (map fst s_vars ++ map fst s_out) (variables s_tcs);

      s_reset_incl       : incl (resets_of s_tcs) (calls_of s_tcs);
      s_reset_consistency: reset_consistency s_tcs;

      s_good             : Forall ValidId (s_in ++ s_vars ++ s_out)
                           /\ Forall ValidId s_inits
                           /\ Forall ValidId s_subs
                           /\ valid s_name
    }.
```

Coq (src/Stc/StcSyntax.v:95-123)

Listing 3.2: Implementation of Stc systems

$$
\begin{aligned}
\textsf{i-tr}_{eq}\,(x =_{ck} e) &\triangleq [x =_{ck} e] \\
\textsf{i-tr}_{eq}\,(x =_{ck} c\ \texttt{fby}\ e) &\triangleq [\texttt{next}\ x =_{ck} e] \\
\textsf{i-tr}_{eq}\,(\varepsilon =_{ck} f\,(\boldsymbol{e})) &\triangleq \varepsilon \\
\textsf{i-tr}_{eq}\,(i \cdot \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})) &\triangleq [i \cdot \boldsymbol{x} =_{ck} f\texttt{<}i, 0\texttt{>}(\boldsymbol{e})] \\
\textsf{i-tr}_{eq}\left(i \cdot \boldsymbol{x} =_{ck} (\texttt{restart}\ f\ \texttt{every}\ r^{ck_r})\,(\boldsymbol{e})\right) &\triangleq [\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ (ck_r\ \texttt{on}\ (r = \textsf{true}))\,; \\
&\qquad\qquad i \cdot \boldsymbol{x} =_{ck} f\texttt{<}i, 1\texttt{>}(\boldsymbol{e})]
\end{aligned}
$$

Figure 3.3: Translation of NLustre equations
(translate_eqn, src/NLustreToStc/Translation.v:57)

always associated with corresponding default transitions (`s_reset_incl`); the transition constraints are *reset consistent* (`s_reset_consistency`, see below); and the identifiers are valid (`s_good`).

We introduce the notion of *reset-consistency* that expresses the expected meaning of the integer parameter k of default transitions.

Definition 3.2.1 (reset_consistency, src/Stc/StcSyntax.v:69)
A list of transition constraints ***tc*** *is reset-consistent if for each default transition* (`- = -<`i, k`>(-)`) *in* ***tc***, *a reset transition* (`reset -<i> every -`) *appears in* ***tc*** *if and only if* $k = 1$.

3.2.2 The translation function

The translation of an NLustre program into Stc is relatively direct. In particular because expressions and control expressions are unchanged. Two major modifications are made: state variables and instances are distinguished in declarations (but not in expressions), and node instantiations with reset are split into distinct reset and default transitions.

Programs are translated node-by-node by a function named i-tr. A node n is translated into a system s with the same name by a function called $\textsf{i-tr}_{\text{node}}$. A declaration is added to s.**inits** for each `fby` equation and to s.**subs** for each node instantiation, with or without reset, using the first defined variable as an instance identifier. For instance names, as mentioned, it avoids handling fresh identifiers. The n.**in** and n.**out** declarations are transferred, respectively, to s.**in** and s.**out** without modification, and n.**vars** is filtered into s.**vars** by removing variables defined by `fby` equations. The translation to the transition constraints s.**tcs** is defined by concat-map $\textsf{i-tr}_{\text{eq}}\,n$.**eqs**. In the Coq development, the function $\textsf{i-tr}_{\text{node}}$ translates a dependent record representing an NLustre node into a dependent record representing an Stc system. As already explained, we use the *Program* library that allows us to omit the dependent predicates while writing the function, and to prove them interactively to complete the definition. Some of these obligations are almost trivial but others require intricate reasoning and intermediate results. Unfortunately, a

$$\frac{\Omega(x) = ck \quad \Omega \vdash_c e :: ck}{P, \Omega \vdash x =_{ck} e} \qquad \frac{\Omega(x) = ck \quad \Omega \vdash e :: ck}{P, \Omega \vdash \texttt{next}\ x =_{ck} e}$$

$$\frac{\begin{array}{c}\mathsf{system}(P, f) \doteq (s, P') \quad \sigma(\boldsymbol{x}) \overset{\text{var}}{=} \boldsymbol{e} \quad \sigma(\boldsymbol{y}) = \boldsymbol{z} \\ \Omega \vdash \boldsymbol{e} :: \sigma^{ck}[\boldsymbol{ck_x}] \quad \Omega(\boldsymbol{z}) = \sigma^{ck}[\boldsymbol{ck_y}]\end{array}}{P, \Omega \vdash \boldsymbol{z} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})} \quad \text{where} \quad \begin{array}{l} s.\mathbf{in} = \boldsymbol{x}^{\tau_x, ck_x} \\ s.\mathbf{out} = \boldsymbol{y}^{\tau_y, ck_y}\end{array}$$

$$\frac{\Omega \vdash ck}{P, \Omega \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

Figure 3.4: The clock system of Stc: transition constraints

classical pitfall with ITPs, these difficult-to-prove results are also not very interesting, so I do not go further into details.

The translation function for equations $\mathsf{i\text{-}tr}_{\text{eq}}$ is shown in figure 3.3. It maps an equation into a list of transition constraints. Basic equations are translated directly, as are `fby` equations but with the elision of the initial constant, which is shifted to the corresponding declaration of x in $s.\mathbf{inits}$. Node instantiations without reset are translated into default transitions; the existence of at least one defined variable is guaranteed by a syntactic predicate, presented earlier (see listing 2.6e on page 46). Each node instantiation with reset is translated into two transitions on the same instance: a default transition and a reset transition.

3.2.3 Clock system of Stc

We present the clock system of Stc, directly adapted from the clock system of NLustre presented in section 2.3.4. Figure 3.4 presents the clocking rules for the transition constraints: expressions and control expressions keep the same rules as for NLustre. As they are very similar to the clocking rules for NLustre (see figure 2.6c on page 48), we do not go into details. Notice that the rule for the reset transition only requires that the clock condition is consistent. For other transition constraints, it can be shown that the consistency of the clock annotations is a property of the clock calculus. The clocking rules for systems and programs are essentially the same as for NLustre.

The type system of Stc is presented in appendix E.3.

3.2.4 Semantics of Stc

The semantics of Stc defines a notion of *state*. We model states by memory trees that are recursively defined by the following generic record type.

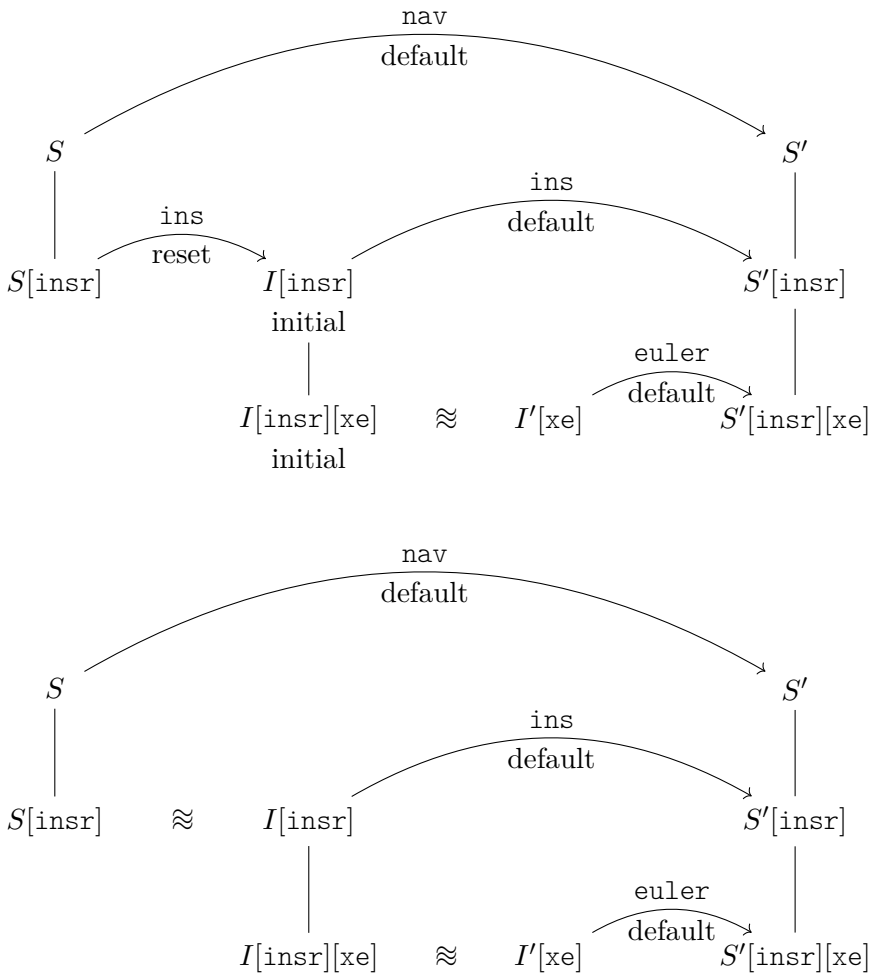


Figure 3.5: Example transition diagrams

Definition 3.2.2 (memory, src/VelusMemory.v:39)

$$\mathsf{memory}\ \alpha \triangleq \{\!|\ \mathsf{values} : \mathsf{env}\ \alpha\ ;\ \mathsf{instances} : \mathsf{env}\ (\mathsf{memory}\ \alpha)\ |\!\}$$

We will write $M(x)$ to denote the lookup of the identifier x in the values environment field of the memory tree M, and $M[x]$ to denote the lookup of x in the instances field. Similarly $M(x \mapsto v)$ represents the updating operation on the values field of M, and $M[x \mapsto M_x]$ the updating operation on its instances field. For both update operations, if the identifier x is not already defined, the binding is added.

The main feature of Stc is the introduction of *transient* states. Consider the translation of the running example in listing 3.1b. The corresponding states and transitions are depicted in figure 3.5. The activation of the default transition of the system nav constrains a start state S and an end state S'. In the top diagram, the reset condition r is true, so the reset transition of ins is taken. It constrains the sub-state S[insr] and a transient

sub-state I[insr]. This sub-state is recursively constrained by the reset transition to contain the initial values of the memory tree for ins. In the bottom diagram, r is false, so the reset transition of ins is not taken, and the transient state I[insr] must simply be unchanged from the start state S[insr]. In both diagrams, the activated default transition of ins constrains the transient state I[insr] with the global sub-state S'[insr]. Similarly, the scheme for this default transition introduces another transient state I'[xe]. As the euler system does not have a reset transition constraint, this transient state is always observationally equal to the sub-state I[insr][xe], which is recursively initial.

There are three important things to understand about this model:

1. The scheme presented in the example diagrams in figure 3.5 can be repeated an arbitrary number of times within any default transition, following the structure of the sub-systems. Consequently, in this model, a single time-step can be subdivided an arbitrary number of times, modeling the simultaneity of an ordered sequence of events.

2. The transient states are only exposed in the internals of a system. From the outside, the activation of the default transition of a system is *atomic*.

3. The *positions* of states that a transition constrains are fixed: a reset transition always constrains a start state with a transient state, while a default transition always constrain a transient state with an end state.[1]

Before presenting the formal rules, we stress a fundamental property that we want the semantics to satisfy. So far, the presented dataflow semantic models are constraints systems: the semantics are not operational in the sense that the environment is not built sequentially, transition constraint by transition constraint—or equation by equation for Lustre or NLustre. Rather, the semantics are relational and constrain the environment to satisfy the semantics of the transition constraints independently of their order. The fact that the semantics does not depend on the order of the equations is a strength of these models: it facilitates reasoning and makes the proof of semantics preservation by scheduling trivial. Another consequence is that these systems of constraints are convenient when working in Coq: the manipulated data structures are only loosely specified, which makes the reasoning about them very flexible. The drawback is that having only a loose specification on the domain of these structures makes it difficult to state correspondences between states. Eventually it is easier to state these correspondences as an observational one-to-one equivalence, defined below.

[1] This loss of generality is assumed: we wanted a working model to fit our needs and to simplify the formalization and its presentation. The language could potentially be extended to allow multiple, ordered transitions within a system and the semantic model would then involve a corresponding number of successive transient states.

Definition 3.2.3 (equal_memory, src/VelusMemory.v:105)
We define the observational equivalence between two memory trees M and M' by the following relation.[2]

$$\frac{\forall x,\ M(x) = M'(x) \quad \forall i,\ M[i] \approxeq M'[i]}{M \approxeq M'}$$

Figure 3.6 shows the semantic rules for Stc. As mentioned, unlike NLustre, Stc is not a language of streams but one of values. Since Stc and NLustre share the same expressions, we can directly reuse the instantaneous semantics defined in section 2.3.6.1. This choice permits a lightweight formalization of Stc in Coq: we reuse many existing definitions and lemmas.

Figure 3.6a presents the semantics of transition constraints. It is parameterized by a program P, an environment of values R, a base clock b, and three memory trees: a start state S, a transient state I and an end state S'.

A basic constraint does not impose anything on the states, it only ensures that the value for x in the environment coincides with the value of the expression e.

A **`next`** constraint specifies the value that the state variable x will take in the end state: if the constraint is activated, the state variable is updated with the value of the expression e, otherwise the value in the start state is maintained. If the transition constraint is activated, the current value of the state variable in the environment is taken from the start state, otherwise, it is absent.

A default transition constrains the transient sub-state $I[i]$ and an end sub-state $S'[i]$, using the dedicated predicate shown in figure 3.6b. The input expressions are evaluated to a list of values and the environment is constrained to hold the output values for the defined variables. The role of the parameter k becomes clearer: if it equals 0, meaning that no reset transition is present then we need a way to ensure that the transient sub-state is always (observationally) equal to the start sub-state.[3]

For a reset transition, we have to consider two cases. If the reset transition is activated then we constrain the transient sub-state to be *initial*, that is, it must contain the declared initial values of its state variables (recursively). The initial-state predicate will be presented shortly. If it is not activated, then we constrain the transient sub-state to be *copied* from the start sub-state. Our model can be compared to the semantics of Auger (2013, Figure 8.8, rules EqAppAbs and EqAppPres). This semantics for a version of NLustre, defined at a single instant, constrains two memories for tracking the pre- and post-states of **`fby`**s and node instantiations, and introduces for the reset an intermediate memory that must equal either an initial memory or the existing instance memory according to the reset stream.

The semantics of a system is shown in figure 3.6b. The system lookup in the program is different from NLustre: we write $\mathsf{system}(P, f) \doteq (s, P')$ to mean that the system s

[2] The way the relation is depicted could be misleading for identifiers that are not in the domains. Recall that, for the sake of simplicity, we omit the details on failing lookups, so assume that the relation implies that the domains are *exactly* the same.

[3] This is an *ad-hoc* way of simulating the presence of a reset transition that never fires.

$$\frac{R, b \vdash_c e :: ck \downarrow R(x)}{P, R, b, S, I, S' \vdash x =_{ck} e}$$

$$\frac{R, b \vdash e :: ck \downarrow \langle v \rangle \quad R(x) = \langle S(x) \rangle \quad S'(x) = v}{P, R, b, S, I, S' \vdash \texttt{next}\ x =_{ck} e}$$

$$\frac{R, b \vdash e :: ck \downarrow \langle\,\rangle \quad R(x) = \langle\,\rangle \quad S'(x) = S(x)}{P, R, b, S, I, S' \vdash \texttt{next}\ x =_{ck} e}$$

$$\frac{\begin{array}{c} R, b \vdash \boldsymbol{e} \downarrow \boldsymbol{v} \quad R, b \vdash ck \downarrow \mathsf{base\text{-}of\text{-}now}\ \boldsymbol{v} \\ k = 0 \rightarrow I[i] \approx S[i] \quad P, I[i], S'[i] \vdash f\,(\boldsymbol{v}) \Downarrow R(\boldsymbol{x}) \end{array}}{P, R, b, S, I, S' \vdash \boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})}$$

$$\frac{R, b \vdash ck \downarrow \mathsf{true} \quad \mathsf{initial\text{-}state}\ P\ f\ I[i]}{P, R, b, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck} \qquad \frac{R, b \vdash ck \downarrow \mathsf{false} \quad I[i] \approx S[i]}{P, R, b, S, I, S' \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

(a) Transition constraints (sem_trconstr, src/Stc/StcSemantics.v:89)

$$\frac{\begin{array}{c} \mathsf{system}(P, f) \doteq (s, P') \quad R(\boldsymbol{x}) = \boldsymbol{xs} \quad R(\boldsymbol{y}) = \boldsymbol{ys} \\ \mathsf{respects\text{-}clock\text{-}now}\ R\ b\ \Gamma \\ \mathsf{s\text{-}closed}\ P\ f\ S \quad \mathsf{s\text{-}closed}\ P\ f\ I \quad \mathsf{s\text{-}closed}\ P\ f\ S' \\ \forall tc \in s.\mathbf{tcs},\ P, R, b, S, I, S' \vdash tc \end{array}}{P, S, S' \vdash f\,(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}} \quad \text{where} \quad \begin{array}{l} s.\mathbf{in} = \boldsymbol{x}^{\tau_{\boldsymbol{x}}, ck_{\boldsymbol{x}}} \\ s.\mathbf{out} = \boldsymbol{y}^{\tau_{\boldsymbol{y}}, ck_{\boldsymbol{y}}} \\ \Gamma = \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\} \\ b = \mathsf{base\text{-}of\text{-}now}\ \boldsymbol{xs} \end{array}$$

(b) Systems (sem_system, src/Stc/StcSemantics.v:121)

$$\frac{P, S, S' \vdash f\,(xs_n) \Downarrow ys_n \quad P, S' \vdash f\,(xs) \overset{n+1}{\bigcirc_Q} ys}{P, S \vdash f\,(xs) \overset{n}{\bigcirc_Q} ys}$$

(c) Loop (loop, src/Stc/StcSemantics.v:210)

Figure 3.6: Semantics of Stc

exists with name f in the program P and that P' is the list of system declarations that come after it. This is to ensure the well-foundedness of recursive predicates. As for NLustre, the local environment R is constrained to hold the values for the inputs and outputs of the node, and the inputs are constrained to be clock-aligned with their declared clocks. Along with R, the local start state S, transient state I and end state S' are constrained by the semantics of each transition constraint of the system. The order of these constraints is immaterial. The s-closed predicate restrains the domains of the memory trees S, I and S'.

Definition 3.2.4 (state_closed, src/Stc/StcSemantics.v:50)

$$\frac{\mathsf{system}(P, f) \doteq (s, P') \qquad \forall x\, v,\ S(x) = v \rightarrow x^{\dot{},\dot{}} \in s.\textbf{\textit{inits}} \quad \forall i\, Si,\ S[i] = Si \rightarrow \exists g,\ i^g \in s.\textbf{\textit{subs}} \wedge \mathsf{s\text{-}closed}\, P'\ g\ S_i}{\mathsf{s\text{-}closed}\, P\ f\ S}$$

The statement s-closed P f S expresses that each variable in the values field of the tree S appears as a state variable declaration of the system f, and that each sub-instance of the tree S corresponds to a sub-system declaration in f and is also recursively closed. This predicate facilitates stating and reasoning about memory equivalence. Without it, an equivalence between two memories would be relative to a domain—the state variables and instance variables that are constrained—which changes as one descends into the tree. Including it in the intermediate model captures a property of the system that is exploited and transmitted by later proofs.

Now we can explain the initial-state predicate.

Definition 3.2.5 (initial_state, src/Stc/StcSemantics.v:73)

$$\frac{\mathsf{system}(P, f) \doteq (s, P') \quad \forall x^{c,ck} \in s.\textbf{\textit{inits}},\ S(x) = [\![c]\!] \qquad \forall i^g \in s.\textbf{\textit{subs}},\ \mathsf{initial\text{-}state}\, P'\ g\ S[i]}{\mathsf{initial\text{-}state}\, P\ f\ S}$$

Whereas s-closed P f S expresses the inclusion of the domain of S in the domain of the declarations of f, initial-state P f S expresses the converse. A tree S is initial if and only if all state variables of f appear in S with their initial values and all sub-systems of f correspond to initial sub-trees in S.

Finally, figure 3.6c presents the coinductive rule implementing the endless iteration of a system. At a given instant n, the system f is activated between an entry state S and a local end state S', taking its input and output values from the streams xs and ys. The loop is repeated at the next instant $n + 1$ with S' as the start state.

3.3 Correctness

Even before the introduction of the modular reset in Vélus, a direct proof between NLustre and Obc could not be done, because the statement of the desired result is too weak to be proved directly by induction. To solve this issue, an alternative *memory*

semantics for NLustre was proposed in [Bourke, Brun, Dagand, et al. (2017)]. This model exposes the *state* of a node in the semantics, that is, its *state variables* (variables defined by a `fby` equation) and its *sub-instances* (the sub-parts of the state for instantiated child nodes). Hence we were able to do the proof in two steps:

1. A proof of correctness between the indexed semantics of NLustre and its memory semantics. This step is relatively easy: for a node it boils down to showing the existence of a memory satisfying the memory semantics, with the same input and output streams as the standard semantics.

2. A much more involved proof of correctness between the memory semantics and the generated imperative Obc code.

We adapt this approach for the proof between NLustre and Stc. We begin by presenting the memory semantics of NLustre and its extension to handle the modular reset.

3.3.1 The memory semantics for NLustre

3.3.1.1 The memory model and the modular reset

The main idea of the memory semantic is to expose the state of the nodes. In a sense, it reflects the syntactic exposition of state performed by the normalization pass. In this model, the semantics is additionally parameterized by a stream of memory trees. The semantic model constrains the leaves of successive memory trees to reflect the successive values to be memorized (without absence) when implementing the `fby`s. The sub-trees of the memories are recursively constrained by the node instantiations.

As explained, the introduction of this alternative semantics allowed Bourke, Brun, Dagand, et al. (2017) to complete the correctness proof between NLustre and Obc in two steps. The challenge concerning is to adapt this model to take the modular reset into account.

A flawed approach: explicitly resetting the memory

To explain the ideas behind the memory model, let us consider the simple example below.[4]

```
node nat(i: int) returns (x: int)
let
  x = 0 fby (x + 1);
tel
```

Lustre

The node `nat` simply outputs the sequence of successive integers. We show its behavior on the chronogram below, where b denotes the activation clock of the node and M represents its associated memory.

[4] Note that in Vélus, nodes must have at least one input to give the base clock. Also in NLustre an output cannot be defined by a `fby` equation, but this is not relevant for this simple example.

b	T	T	T	F	T	T	T	$\cdots$
$M(x)$	0	1	2	3	**3**	4	5	$\cdots$
x	0	1	2		3	4	5	$\cdots$
$x+1$	1	2	3		4	5	6	$\cdots$
$\texttt{nat}(i)$	0	1	2		3	4	5	$\cdots$

The value of x at each instant is obtained from the memory content $M(x)$, according to the clock. Observe that the memory is *causal*: for each instant $n > 0$, the content of M depends only on values of $x + 1$ at instants strictly before n. This property is a consequence of the semantics of `fby`. Indeed, we can verify that for each instant, the value for M is obtained from preceding instants. In particular, at the fifth instant, it is the absence of x at the fourth instant which constrains the value to be maintained in the memory.

This causality property is essential to the proof of correctness of imperative code generation: it allows to formulate a state correspondence just *before* the current reaction. That is, at the end of the previous reaction, when the update is performed in the imperative code, the memory is guaranteed to contain the values available for the current reaction.

Directly extending this model with the modular reset breaks the causality property, and the proof of correctness. The modular reset we want to formalize is *strong*, meaning that the state of the reset node is actually reinitialized at the very same instant the reset condition is true.[5] Consequently, if we define the memory behavior of the reset naively as depicted below, the memory is not *causal* anymore: its content can be modified without delay by a reset.

b	T	T	T	F	T	T	T	$\cdots$
r	F	F	F		T	F	F	$\cdots$
$M(x)$	0	1	2	3	**0**	1	2	$\cdots$
x	0	1	2		0	1	2	$\cdots$
$x+1$	1	2	3		1	2	3	$\cdots$
$(\texttt{restart nat every } r)(i)$	0	1	2		0	1	2	$\cdots$

Here the reset fires just after the absence, to show the consequence on the crucial memory invariant that the memory is left unchanged when the node is not activated. This is not true anymore, because at the fifth instant after the absence of the fourth instant, the reset reinitializes the memory. Thus this model complies with the fact that the value of x coincides with the value in $M(x)$, but not with the fact that the values in the memory are the values before the current reaction. In particular, the value of $M(x)$ at the fifth instant is 0, which matches the expected value of x, but the value at the end of the previous reaction should be 3. Practically, this is a matter of constraint conflict: one constraint says what the next value should be, and another (later) one says that it should be reset.

[5] We borrow the terminology from state machines, where a *strong* transition is taken immediately, while a *weak* transition is delayed.

A causal model with strong reset

The solution to the causality problem raised by the modular reset is to re-establish the delayed behavior of the memory, in order to keep the properties of the memory. With this approach, the effect of the reset on the memory is delayed by one instant, as shown below.

b	T	T	T	F	T	T	T	$\cdots$
r	F	F	F		T	F	F	$\cdots$
$M(x)$	0	1	2	3	3	**1**	2	$\cdots$
x	0	1	2		0	1	2	$\cdots$
$x+1$	1	2	3		1	2	3	$\cdots$
(`restart nat every` r)(i)	0	1	2		0	1	2	$\cdots$

Here we can check the fact that each value of $M(x)$ actually depends on previous values of $x+1$, and that the causality property holds. In particular, the essential property that the memory is left unchanged when the node is not activated is now ensured. There is still an inevitable discontinuity in the model: the value of x at the fifth instant does not match the value of $M(x)$ anymore. This is not a problem though, given the way we define the formal semantics, as described in the next section.

3.3.1.2 Formal rules

The essence of the memory model is to expose the state of the nodes, thus only the semantics of equations differs from the standard indexed semantics presented in section 2.3.6.2. We only deal with indexed streams and not with coinductive streams from now on. Thus, since there is no longer any risk of ambiguity between the indexing notations for coinductive streams and that for indexed streams, we will use the former notation xs_n in place of the latter $xs\ n$ since it is more readable. In particular, we will keep the lambda notation for building streams, so we may mix notations as in $\lambda\, n.\, xs_n$ to designate $\lambda\, n.\, (xs\ n)$. Figure 3.7a shows this new semantics of the equations. The judgments now have an additional parameter M which is a stream of memory trees.

A basic equation has exactly the same semantics as in the standard model (see figure 2.9a on page 58), since it does not involve the state.

The semantics for an equation `x = c fby e` is the core of the model. It is not given by a synchronous stream operator fby anymore but by a set of constraints on the memory stream M. First, the expression e is associated with a stream s. The second antecedent of the rule constrains the initial content of the memory stream to be equal to the semantics of the constant c. Then, the third antecedent constrains both the current *slice* of the stream of environments H and the next slice of the stream of memories M. At each instant n, there are two cases: (1) if s is absent, then x is absent and the memory *stutters*: its current value is held for the next instant $(n+1)$, or (2) if s is present with value v, then x is given the current value from the memory, and the memory content is updated to v for the next instant.

For a node instantiation, the only change is to expose the corresponding instance in the memory tree. A convenient way to do this is to name it after the first variable of the

$$\frac{H, bs \vdash_c e :: ck \downarrow\downarrow H_*(x)}{G, H, bs, M \vdash x =_{ck} e}$$

$$\frac{M_0(x) = [\![c]\!] \quad \forall n,\ \begin{cases} H_n(x) = \langle\ \rangle & \wedge M_{n+1}(x) = M_n(x) \quad \text{if } s_n = \langle\ \rangle \\ H_n(x) = \langle M_n(x)\ \rangle & \wedge M_{n+1}(x) = v \quad \text{if } s_n = \langle v \rangle \end{cases} \quad H, bs \vdash e :: ck \downarrow\downarrow s}{G, H, bs, M \vdash x =_{ck} c \texttt{ fby } e}$$

$$\frac{H, bs \vdash \boldsymbol{e} \downarrow\downarrow xs \quad H, bs \vdash ck \downarrow\downarrow \mathsf{base\text{-}of}\ xs \quad G, M_*[i] \vdash f\ (xs) \downarrow\downarrow H_*(\boldsymbol{x})}{G, H, bs, M \vdash i \cdot \boldsymbol{x} =_{ck} f\,(\boldsymbol{e})}$$

$$\frac{H, bs \vdash \boldsymbol{e} \downarrow\downarrow xs \quad H, bs \vdash ck \downarrow\downarrow \mathsf{base\text{-}of}\ xs \quad \mathsf{bools\text{-}of}\ (H_*(y)) \doteq rs \quad \forall k,\ \exists M^k,\ G, M^k \vdash f\left(\mathsf{mask}^k_{rs}\ xs\right) \downarrow\downarrow \mathsf{mask}^k_{rs}\ (H_*(\boldsymbol{x})) \wedge \mathsf{m\text{-}masked}^k_{rs}\ (M_*[i])\ \ M^k}{G, H, bs, M \vdash i \cdot \boldsymbol{x} =_{ck} (\texttt{restart } f \texttt{ every } y)\,(\boldsymbol{e})}$$

(a) Equations (msem_equation, src/NLustre/NLMemSemantics.v:141)

$$\frac{\begin{array}{c}\mathsf{node}(G, f) \doteq n \quad H_*(\boldsymbol{x}) \approx xs \quad H_*(\boldsymbol{y}) \approx ys \\ \mathsf{respects\text{-}clock}\ H\ \ bs\ \Gamma \quad \mathsf{m\text{-}closed}\ M\ n.\mathbf{eqs} \\ \forall eq \in n.\mathbf{eqs},\ \ G, H, bs, M \vdash eq\end{array}}{G, M \vdash f\ (xs) \downarrow\downarrow ys} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \\ \Gamma = \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{ck_x}\} \\ bs = \mathsf{base\text{-}of}\ xs \end{array}$$

(b) Nodes (msem_node, src/NLustre/NLMemSemantics.v:176)

Figure 3.7: Memory semantics of NLustre

left-hand side of the equation. We know that this variable is uniquely defined from the syntactic requirements on nodes, and we know that it exists since in Vélus, nodes must have at least one output. Indeed, in our setup, nodes without output make no sense practically.[6] We write $M_*[i]$ as a shorthand for $\lambda\, n.\, M_n[i]$, for the stream of sub-memories associated with the instance named i.

The corresponding mutually inductive rule for the semantics of a node is shown in figure 3.7b. It is more or less a direct adaptation of the standard rule (see figure 2.9b on page 58). We keep the crucial property that the semantics for a node is independent from the order of its equations. There is an additional antecedent, though, requiring that for any identifier defined in the values field there is a corresponding `fby` equation, and that any identifier defined in the instances field is the first variable in some node instantiation with or without reset.

Definition 3.3.1 (memory_closed, src/NLustre/NLMemSemantics.v:134)

$$\begin{aligned}\mathsf{m\text{-}closed}\ M\ \boldsymbol{eq} \triangleq \forall n,\ &(\forall x,\ (\exists v,\ M_n(x) = v) \rightarrow x = \texttt{-}\ \texttt{fby}\ \texttt{-} \in \boldsymbol{eq})\\ &\wedge\ \forall i,\ (\exists Mi,\ M_n[i] = Mi) \rightarrow i \cdot \texttt{-} = \texttt{-}(\texttt{-}) \in \boldsymbol{eq}\end{aligned}$$

Compared to the definition 3.2.4 on page 84 of s-closed, there are two differences:

1. In Stc, state variables and instances are retrieved in the s-closed predicate via their explicit declarations within the enclosing system, but here we use the list of equations of the node.

2. Unlike the s-closed predicate, m-closed is not recursive. The fact that a whole memory M is closed is thus ensured here by the semantics itself, by recursively imposing m-closed at each level of M. This is an arbitrary choice.

The rule for the modular reset also uses the first defined variable as an instance name. As for the coinductive and indexed standard semantics models, we use a universal quantification together with the mask operator to model the idea of shifting to a fresh instance whenever a reset occurs. Here, each instance k must have its own memory M^k, that satisfies a recursive invocation of the rule for nodes. These local memories are, in turn, combined to constrain the overall memory of the instance, that is, to constrain $M_*[i]$. To do this, we adapt the *masking* idea used for input and output streams, to guarantee the causality property discussed in 3.3.1.1 and avoid conflicting constraints at instants of reset. We introduce the following predicate:

Definition 3.3.2 (memory_masked, src/NLustre/NLMemSemantics.v:107)

$$\mathsf{m\text{-}masked}^{k}_{rs}\ M\ N \triangleq \forall n,\ \mathsf{count}\ rs\ n = (\mathit{if}\ rs_n\ \mathit{then}\ k+1\ \mathit{else}\ k) \rightarrow M_n = N_n$$

This predicate differs from mask in a rather subtle way. First, it is a *predicate*, rather than a *function*. Indeed, while an instance k can satisfy the semantics rule for nodes by producing absent outputs before and after its interval of activation, the treatment of

[6] They would if we handled side-effects, but this is not the case.

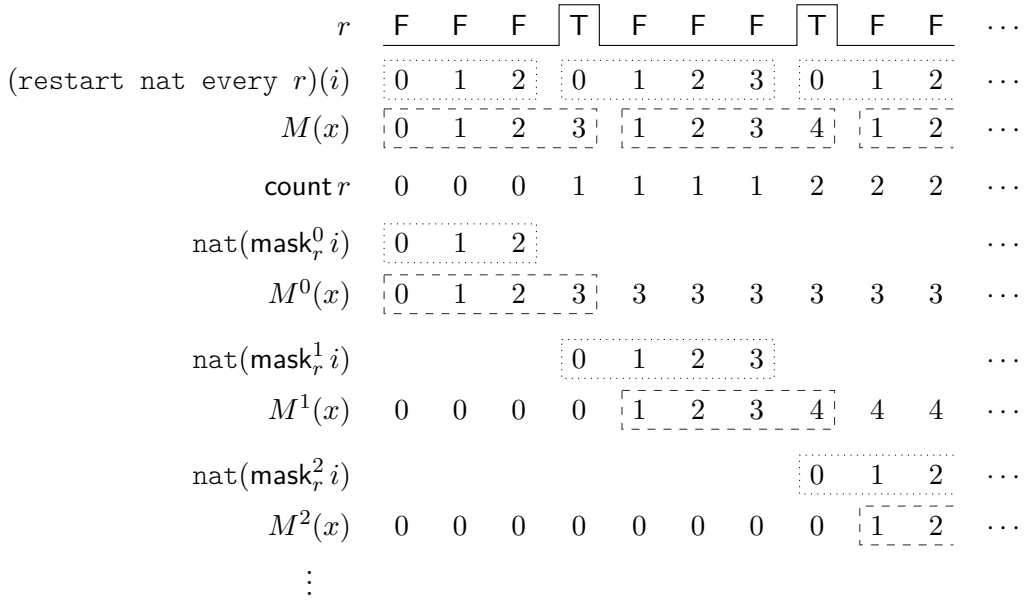


Figure 3.8: Masking with memories explained on an example

memories is more intricate. Even when not activated, constant memories must satisfy the stuttering property, that is, their contents must be held. This is ensured by the memory semantics for `fby`s. Second, the masking for streams lets the underlying stream values appear until just before the next reset. On the contrary, the masking for memories lets the underlying memory content appear until the very instant of reset. This is to ensure the causality property on the memory stream.

This will become clearer by studying the trace of the previous `nat` example depicted in figure 3.8. The count r row gives the active instance over time, below it are presented both the $M^k(x)$ values and the result of the application of `nat` on a masked input[7] for $k = 0$, 1, and 2. The dotted lines correspond to stream masking. The dashed lines show the intervals *selected* by the m-masked predicate. Note the stuttering of the instances' memory streams and the shift between the intervals selected by stream masking and by memory masking.

3.3.1.3 Properties of the memory model

From the standard indexed semantics to the memory semantics

The fundamental property that the memory model satisfies is that it is implied by the standard indexed model. The crucial step is to prove the result at the level of equations.

[7] Recall that in this example, the `i` input only serves to give the activation tempo.

Lemma 3.3.1 (sem_msem_eq, src/NLustre/NLMemSemantics.v:639)
Given a program G, a base clock stream bs, an environment stream H, an equation eq such that $G, H, bs \vdash eq$, and the induction hypothesis for nodes:

$$\forall f\ xs\ ys,\ G \vdash f\ (xs) \downarrow\downarrow ys \rightarrow \exists M,\ G, M \vdash f\ (xs) \downarrow\downarrow ys$$

*Furthermore, assume the following induction hypothesis for a list of equations **eqs** and a stream of memories M:*

1. *the variables defined by the equations **eqs** and the equation eq are distinct,*
2. $\mathsf{m\text{-}closed}\ M\ \boldsymbol{eqs}$, *and*
3. $\forall eq' \in \boldsymbol{eqs},\ G, H, bs, M \vdash eq'$.

Then there exists a new memory stream M' such that:

1. $\forall eq' \in eq \cdot \boldsymbol{eqs},\ G, H, bs, M' \vdash eq'$, *and*
2. $\mathsf{m\text{-}closed}\ M'\ (eq \cdot \boldsymbol{eqs})$.

The proof is by cases on the four possibilities for eq. In each case, one has to *construct* a witness for the new memory stream M'. The reset case $i \cdot \boldsymbol{x} =_{ck} (\texttt{restart}\ f\ \texttt{every}\ y)(\boldsymbol{e})$ is arguably one of the most problematic proof obligations in Vélus, in terms of logical principles. As for the node instantiation case, we would like to exhibit a well-defined Mi in order to give the witness $M' = \lambda n.\, M_n[i \mapsto Mi_n]$. The problem is to build that Mi, such that, according to the reset rule for in figure 3.7a,

$$\forall k,\ \exists M^k,\ G, M^k \vdash f\left(\mathsf{mask}^k_{rs}\ xs\right) \downarrow\downarrow \mathsf{mask}^k_{rs}\ ys \text{ and } \mathsf{m\text{-}masked}^k_{rs}\ Mi\ M^k$$

Now, consider the two involved hypotheses.

1. The node induction hypothesis:
   $\forall f\ xs\ ys,\ G \vdash f\ (xs) \downarrow\downarrow ys \rightarrow \exists M,\ G, M \vdash f\ (xs) \downarrow\downarrow ys$
2. The result of the inversion of $G, H, bs \vdash i \cdot \boldsymbol{x} =_{ck} (\texttt{restart}\ f\ \texttt{every}\ y)(\boldsymbol{e})$:
   $\forall k,\ G \vdash f\left(\mathsf{mask}^k_{rs}\ xs\right) \downarrow\downarrow \mathsf{mask}^k_{rs}\ ys$

By combining these two hypotheses, we deduce the following:

$$\forall k,\ \exists M^k,\ G, M^k \vdash f\left(\mathsf{mask}^k_{rs}\ xs\right) \downarrow\downarrow \mathsf{mask}^k_{rs}\ ys$$

We must construct the function Mi from the potentially infinite sequence of instance memory streams M^k, whose existence of each is given by the combined hypothesis above. One way to solve the problem is to postulate a function $F = \lambda k.\, M^k$ allowing us to *choose* the kth instance's memory stream M^k. To account for the masking predicate $\mathsf{m\text{-}masked}$, a good choice for Mi would be the following:

$$Mi = \lambda n.\ (F\ (\text{if } rs_n \text{ then } \mathsf{count}\ rs\ n - 1 \text{ else } \mathsf{count}\ rs\ n))_n$$

That is, Mi is composed of the M^k memory streams, chosen by F according to the cumulative number of reset ticks at an instant n. So clearly the proof boils down to constructing F. Unfortunately, our proof obligation here is another instance of the axiom of functional choice:

$$\forall R,\ (\forall x,\ \exists y,\ R\ x\ y) \rightarrow \exists f,\ \forall x,\ R\ x\ (f\ x)$$

Using this axiom is the only way I found to solve the problem. Still, the trick used in section 2.4.1 could be repeated here: indeed, it should be possible to define an interpreter at equations/nodes level for the memory model that would build both the environment stream and the memory stream. We could then use it to explicitly construct the choice function F. The definition of this interpreter is current work in the project and beyond the scope of this thesis.

To prove the result at node level, we proceed by induction on the program. This induction relies on the following ordering predicate.

Definition 3.3.3 (Ordered_nodes, src/NLustre/NLOrdered.v:50)

$$\frac{}{\mathsf{NL\text{-}Ordered}\ \varepsilon}$$

$$\frac{\mathsf{NL\text{-}Ordered}\ G \quad \forall n' \in G,\ n.\mathit{name} \neq n'.\mathit{name} \qquad \forall f,\ \texttt{-} = [\texttt{restart}]\ f\ [\texttt{every}\ \texttt{-}]\ (\texttt{-}) \in n.\boldsymbol{\mathit{eqs}} \rightarrow\ f \neq n.\mathit{name} \wedge \exists n',\ \mathsf{node}(G, f) \doteq n'}{\mathsf{NL\text{-}Ordered}\ (n \cdot G)}$$

The statement $\mathsf{NL\text{-}Ordered}\ G$ expresses that (1) all nodes in G are uniquely defined, (2) recursion is forbidden, and (3) each node instantiation in an enclosing node corresponds to a later declaration in the program. This unusual order is more convenient in proofs simply implies that the parsed program has to be reversed before further compilation.

Theorem 3.3.2 (sem_msem_node, src/NLustre/NLMemSemantics.v:772)
Given a well-ordered program G, a name f, and two streams of lists of values xs and ys such that $G \vdash f\ (xs) \downarrow\downarrow ys$, then

$$\exists M,\ G, M \vdash f\ (xs) \downarrow\downarrow ys$$

Determinism of the initial memory

Now we can state a property ensuring a kind of determinism of the memory semantics relative to the initial memory. That is, we show that the initial content of the memory only depends on the node itself. We follow the same scheme described in the previous section, combining an induction on the program G and an intermediate result on equations.

Lemma 3.3.3 (msem_eqs_same_initial_memory, src/NLustre/NLMemSemantics.v:813)
*Given a program G, a list of equations **eqs**, two base clock streams bs and bs', two environment streams H and H', and two memory streams M and M' such that:*

1. *the variables defined by the equations **eqs** are distinct,*

2. $\forall eq \in \boldsymbol{eqs},\ G, H, bs, M \vdash eq$ *and* $G, H', bs', M' \vdash eq$,

3. m-closed M $\boldsymbol{eqs}$ *and* m-closed M' $\boldsymbol{eqs}$.

and assuming the induction hypothesis:

$$\forall f\ xs\ ys\ xs'\ ys'\ M\ M',\ G, M \vdash f\ (xs) \downarrow\downarrow ys \rightarrow G, M' \vdash f\ (xs') \downarrow\downarrow ys' \rightarrow M_0 \approx\approx M'_0$$

Then $M_0 \approx\approx M'_0$.

Theorem 3.3.4 (same_initial_memory, src/NLustre/NLMemSemantics.v:914)
Given a well-ordered program G, a name f, four streams of lists of values xs, ys, xs' and ys', and two memory streams M and M' such that $G, M \vdash f\ (xs) \downarrow\downarrow ys$ and $G, M' \vdash f\ (xs') \downarrow\downarrow ys'$, then

$$M_0 \approx\approx M'_0$$

Initial memory until the first presence
Another important property that we show about the memory model is the persistence of the initial memory until the first activation of the considered node.

Theorem 3.3.5 (msem_node_absent_until, src/NLustre/NLMemSemantics.v:1076)
Given a well-ordered program G, a name f, a natural integer n_0, two streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f\ (xs) \downarrow\downarrow ys$ and where for all instants strictly before n_0, all values in xs are absent, then

$$\forall n \leq n_0,\ M_n \approx\approx M_0$$

Equivalence of the semantic models
It is rather easy to show the reverse inclusion of the semantic models.

Lemma 3.3.6 (msem_sem_node_equation, src/NLustre/NLMemSemantics.v:1130)
For a program G,

- *Given a name f, two streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f\ (xs) \downarrow\downarrow ys$, then*

$$G \vdash f\ (xs) \downarrow\downarrow ys$$

- *Given a base clock stream bs, an environment stream H, a memory stream M, and an equation eq such that $G, H, bs, M \vdash eq$, then*

$$G, H, bs \vdash eq$$

The equivalence of the two models follows directly from theorem 3.3.2 and lemma 3.3.6.

Theorem 3.3.7 (equivalence, src/NLustre/NLMemSemantics.v:1173)
Given a well-ordered program G, a name f, two streams of lists of values xs and ys, then

$$G \vdash f\ (xs) \downarrow\downarrow ys \leftrightarrow \exists M,\ G, M \vdash f\ (xs) \downarrow\downarrow ys$$

3.3.2 The proof of correctness

The proof of correctness of the i-tr translation function between NLustre and Stc consists in showing that the semantics of the source program is preserved. The memory semantics for NLustre allows us to directly state a correspondence between the memory stream M of NLustre and the memory states S and S' of Stc.

The first step is to show that the NLustre memory stream is indeed *initial*, in the terms of Stc, at the very first instant.

Lemma 3.3.8 (msem_node_initial_state, src/NLustreToStc/Correctness.v:135)
Given a well-ordered program G, a name f, two streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f\,(xs) \downarrow\downarrow ys$, then initial-state (i-tr G) $f\ M_0$.

The following two lemmas transfer key properties from NLustre to Stc. First, the well-ordering is preserved by the i-translation, this is a basis to prove more involved properties about translated programs. The well-ordering predicate of is extended straightforwardly for Stc.

Definition 3.3.4 (Ordered_systems, src/Stc/StcOrdered.v:32)

$$\frac{}{\text{Stc-Ordered}\ \varepsilon} \qquad \frac{\text{Stc-Ordered}\ P \quad \forall s' \in P,\ s.\mathit{name} \neq s'.\mathit{name} \quad \forall i^f \in s.\mathbf{subs},\ f \neq s.\mathit{name} \wedge \exists s'\,P',\ \text{system}(P, f)\,s'P'}{\text{Stc-Ordered}\ (s \cdot P)}$$

This predicate expresses exactly the same kind of properties than those of the well-ordering predicate for NLustre of definition 3.3.3 on page 92. The only difference is that here again we directly use the declarations of the sub-systems of a considered system rather than a scan of the equations of the considered node.

Lemma 3.3.9 (Ordered_nodes_systems, src/NLustreToStc/Correctness.v:65)
Given a well-ordered program G, the translated program i-tr G *is well-ordered.*

Second, the memory semantics of a node, by ensuring the m-closed predicate at each depth level in the structure of the program guarantees the s-closed property on the translated node.

Lemma 3.3.10 (msem_node_state_closed, src/NLustreToStc/Correctness.v:359)
Given a well-ordered program G, a name f, two streams of lists of values xs and ys and a memory stream M such that $G, M \vdash f\,(xs) \downarrow\downarrow ys$, then $\forall n$, s-closed (i-tr G) $f\ M_n$.

Then, we follow the classical proof scheme to show the correctness theorem. We start by proving an intermediate result for a single equation. The goal is to exhibit a transient state I that satisfies the Stc semantics of the translated equation and also those of already translated equations. This lemma and its proof are very similar in spirit to lemma 3.3.1 and its proof.

Lemma 3.3.11 (equation_correctness, src/NLustreToStc/Correctness.v:389)
Given a well-ordered program G, a base clock stream bs, an environment stream H, a memory stream M, a clocking environment Ω such that respects-clock $bs\ H\ \Omega$, *a well-clocked equation eq such that $G, H, bs, M \vdash eq$, and the node induction hypothesis:*

$$\forall f\ xs\ ys\ M,\ G, M \vdash f\ (xs) \Downarrow ys \rightarrow \forall n,\ \text{i-tr}\ G, M_n, M_{n+1} \vdash f\ (xs_n) \Downarrow ys_n$$

*Let **tcs** be a list of transition constraints, **subs** a list of sub-systems declarations and I a memory stream such that:*

1. *the variables defined by the transition constraints **tcs** and the equation eq are distinct,*
2. *at each instant the* values *field of I is empty and its sub-instances are closed relative to the declarations in **subs**, and*
3. $\forall n,\ \forall tc \in \mathbf{tcs},\ \text{i-tr}\ G, H_n, bs_n, M_n, I_n, M_{n+1} \vdash tc.$

Then there exists a new memory stream I' such that:

1. $\forall n,\ \forall tc \in (\text{i-tr}_{eq}\ eq + \mathbf{tcs}),\ \text{i-tr}\ G, H_n, bs_n, M_n, I'_n, M_{n+1} \vdash tc$, *and*
2. *at each instant the* values *field of I' is empty and its sub-instances are closed relative to the declarations in **subs**.*

We directly generalize the result to a list of equations.

Corollary 3.3.11.1 (equations_correctness, src/NLustreToStc/Correctness.v:579)
Given a well-ordered program G, a base clock stream bs, an environment stream H, a memory stream M, a clocking environment Ω such that respects-clock $bs\ H\ \Omega$, *a list of well-clocked equations **eqs** such that the variables defined by the equations **eqs** are distinct and $\forall eq \in \mathbf{eqs},\ G, H, bs, M \vdash eq$, and the node induction hypothesis:*

$$\forall f\ xs\ ys\ M,\ G, M \vdash f\ (xs) \Downarrow ys \rightarrow \forall n,\ \text{i-tr}\ G, M_n, M_{n+1} \vdash f\ (xs_n) \Downarrow ys_n$$

Then there exists a memory stream I such that:

1. $\forall n,\ \forall tc \in (\text{concat-map i-tr}_{eq}\ \mathbf{eq}),\ \text{i-tr}\ G, H_n, bs_n, M_n, I_n, M_{n+1} \vdash tc$, *and*
2. *at each instant the* values *field of I is empty and its sub-instances are closed relative to the declarations in **subs**.*

We can finally state the main correctness result.

Theorem 3.3.12 (correctness, src/NLustreToStc/Correctness.v:628)
Given a well-ordered and well-clocked program G, a name f, two streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f\ (xs) \Downarrow ys$, then

$$\forall n,\ \text{i-tr}\ G, M_n, M_{n+1} \vdash f\ (xs_n) \Downarrow ys_n$$

This result states that, at each instant, the default transition of the translated node constrains the current memory of the node and its next memory, consuming and producing, respectively, the current values of the input and output streams. We use it to show semantics preservation of the repeated activation of the default transition of the translated node, starting from the initial state.

Corollary 3.3.12.1 (correctness_loop, src/NLustreToStc/Correctness.v:678)
Given a well-ordered and well-clocked program G, a name f, two streams of lists of values xs and ys, and a memory stream M such that $G, M \vdash f\,(xs) \downdownarrows ys$, then

$$\mathsf{initial\text{-}state}\,(\mathsf{i\text{-}tr}\,G)\ \ f\ M_0 \quad \textit{and} \quad \mathsf{i\text{-}tr}\,G, M_0 \vdash f\,(xs) \overset{0}{\circlearrowright} ys$$

Chapter 4

Generation of imperative code

Contents

After having introduced explicit manipulation of the state in the transformation to Stc, we turn to the problem of fixing the execution order of the operations, and assignments, necessary to calculate the behaviour described by the original program. The target language Obc represents the first step into the imperative paradigm. Obc is a version of the imperative object-oriented language used in the clock-directed modular compilation scheme [Biernacki et al. (2008)]. In this approach, each node is modularly translated into a class with fields for state variables and sub-instances, and two methods: a *step* method that calculates a single cycle of the original node, and a *reset* method that is used whenever the node state has to be (re)initialized. Within a node, equations are scheduled to fix the evaluation order and then translated into a sequence of conditional statements whose guards reflect the activation clocks of the dataflow equations.

The translation from Stc to Obc is all but identical to the standard scheme, whose verification is described in [Bourke, Brun, Dagand, et al. (2017)]. Only minor adaptations

are required to treat the distinction between reset and default transitions. To understand the need for scheduling, we begin by describing the Obc language. Then we explain how to schedule the Stc transition constraints in order to generate a correct imperative program. The translation function from Stc to Obc and its proof of correctness are described in the two subsequent sections. Last, we present the fusion optimization performed on generated Obc programs.

4.1 Obc: Object Code language

4.1.1 Syntax of Obc

Figure 4.1 presents the syntax of Obc. Working from the bottom up, an Obc program is a list of classes. Each class has its own state variables declared with their names and types, sub-instances i^c, each declared with its name i and class c, and a list of methods. Each method has a name, a list of input parameters, a list of output parameters and a list of local variables. The body of a method is a single statement. A statement is an assignment to a (local or output) variable of an expression; an assignment to a state variable; a conditional branching statement; a method call $\boldsymbol{x} := i^c.f(\boldsymbol{e})$ specifying the list of variables $\boldsymbol{x}$ to receive the results, if any, a class name c, an instance variable i, a method name f and a list of expressions $\boldsymbol{e}$ for the inputs; the sequence of two statements or a no-operation statement. An expression is a constant, a standard variable, a state variable, a unary or binary operation or a validity assertion. A validity assertion is a syntactic decoration that is used in method call arguments to indicate that the enclosed expression is initialized. This will be explained later.

Listing 4.1 shows the Coq implementation of Obc syntax for methods and classes. Both are implemented using dependent records, allowing the direct embedding of syntactic properties into the syntax itself. We ensure that input, local and output variables of methods are uniquely declared (`m_nodupvars`), as well as state variable and sub-instance (`c_nodup`) and methods (`c_nodupm`). As usual, identifiers must be valid (`m_good` and `c_good`).

The type system of Obc is described in appendix E.4.

4.1.2 Semantics of Obc

The semantics of Obc is defined, as for the other languages, as a relation, but this time the semantics is not by constraints. It is a classical big-step operational semantics—or *natural* semantics, as introduced by Kahn (1987).

The notion of variable validity is introduced in Obc to mediate between presence and absence in the dataflow model and the *undefinedness* of variables in the imperative model. Bourke, Brun, Dagand, et al. (2017) propose a semantics for expressions that relates them directly to values. This approach is not sufficient to handle nodes inputs with different clocks. Indeed, the method calls generated for such nodes may involve expressions containing variables that have not been written to because the corresponding streams are absent at particular instants in the source dataflow program. Thus the

e	$::=$	**expression**
	$\mid\ c$	(constant)
	$\mid\ x^t$	(variable)
	$\mid\ \texttt{state}(x)^t$	(state variable)
	$\mid\ (\diamond\ e)^t$	(unary operation)
	$\mid\ (e \oplus e)^t$	(binary operation)
	$\mid\ [x]^t$	(validity assertion)
s	$::=$	**statement**
	$\mid\ x\ \texttt{:=}\ e$	(assignment)
	$\mid\ \texttt{state}(x)\ \texttt{:=}\ e$	(state assignment)
	$\mid\ \texttt{if}\ e\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \}$	(conditional)
	$\mid\ [x^+_{,}\ \texttt{:=}]\ x^x.x(e^*_{,})$	(method call)
	$\mid\ \texttt{s ; s}$	(sequence)
	$\mid\ \texttt{skip}$	(no-operation)
d	$::=\ x^\tau$	**variable declaration**
m	$::=\ x(d^+_{,})\ \texttt{returns}\ (d^+_{,})$	**method**
	$[\texttt{var}\ d^+_{,}]$	
	$\{$	
	$\quad s$	
	$\}$	
cls	$::=\ \texttt{class}\ x\ \{$	**class**
	$\quad [\texttt{state}\ d^+_{,}]$	
	$\quad [\texttt{instance}\ (x^x)^+_{,}]$	
	$\quad m^*$	
	$\}$	
p	$::=\ cls^*$	**program**

Figure 4.1: The Obc abstract syntax

```
Record method : Type :=
  mk_method {
      m_name : ident;
      m_in   : list (ident * type);
      m_vars : list (ident * type);
      m_out  : list (ident * type);
      m_body : stmt;

      m_nodupvars : NoDupMembers (m_in ++ m_vars ++ m_out);
      m_good      : Forall ValidId (m_in ++ m_vars ++ m_out)
    }.
```

Coq (src/Obc/ObcSyntax.v:60-70)

(a) Methods

```
Record class : Type :=
  mk_class {
      c_name    : ident;
      c_mems    : list (ident * type);
      c_objs    : list (ident * ident);   (* (instance, class) *)
      c_methods : list method;

      c_nodup   : NoDup (map fst c_mems ++ map fst c_objs);
      c_nodupm  : NoDup (map m_name c_methods);
      c_good    : Forall (fun xt => valid (fst xt)) c_objs /\ valid c_name
    }.
```

Coq (src/Obc/ObcSyntax.v:160-170)

(b) Classes

Listing 4.1: Implementation of Obc syntax

semantic model of Obc is extended by Bourke and Pouzet (2019) to account for undefined values in the semantics and to indicate explicitly in the sub-expressions when a variable is *valid*, that is, guaranteed to be defined. The importance of this feature to the generation of Clight will be explained with more details later.

Before presenting the semantic rules, it is necessary to introduce the following notations.

1. We write $\lfloor v \rfloor$ to indicate that the value v is defined, and $\lfloor\ \rfloor$ to indicate the undefined value. In the implementation, we use the standard `option` type.

2. We keep the notations $me(x \mapsto v)$ and $me[x \mapsto me_x]$ to denote the update operation in the values and instances fields, respectively, of a memory tree me. Recall that we use the notation $ve\{x \mapsto v\}$ to denote the update operation in a standard environment ve, that is, a map from identifiers to values.

3. We use the notation $ve\{x \mapsto\!\!\!\!\!\!\sim\ v\}$ to denote the update operation for when v is either defined or undefined:

$$ve\{x \mapsto\!\!\!\!\!\!\sim\ \lfloor v \rfloor\} \triangleq ve\{x \mapsto v\}$$
$$ve\{x \mapsto\!\!\!\!\!\!\sim\ \lfloor\ \rfloor\} \triangleq ve$$

4. Recall that, for the sake of simplicity, we have chosen not to emphasize the partiality of maps. For example, when we write $ve(x) = v$ to state that v is bound to x in ve, we assume that x is in the domain of ve. That is, this functional notation itself has a sense only on the actual domain of definition. Now that we need to treat partiality more precisely, we will write $ve((x)) = v$, with double parentheses, in order to explicitly represent failure:

$$ve((x)) \triangleq \begin{cases} \lfloor\ \rfloor & \text{if } x \notin ve \\ \lfloor ve(x) \rfloor & \text{otherwise} \end{cases}$$

The semantics of expressions is shown in Figure 4.2a. We write $me, ve \vdash e \Downarrow v$ to state that in the memory tree me and value environment ve, the expression e evaluates to a defined or undefined value v. An Obc constant always evaluates to a defined value, calculated through the abstracted semantics for constants. A variable is looked up in the environment: if the lookup succeeds, then the variable is evaluated to the corresponding defined value, otherwise it evaluates to the undefined value. Consequently the semantics of a variable is *always* defined, even if the value of the variable is not. A state variable should always be defined in the memory tree. Its value is simply looked up in the values field of me. Its semantics is not defined if the variable is not in the domain of me, unlike for standard variables. The semantics of unary and binary operations is defined recursively. The semantics of these operations on undefined values is not defined. Indeed, the undefined value is not propagated through expressions: only a variable can evaluate to an undefined value. In contrast to the rule for a "naked" variable, the semantics of a variable wrapped in a validity assertion is only defined when the variable has a value in the environment ve.

$$\frac{}{me, ve \vdash c \Downarrow \lfloor [\![c]\!] \rfloor} \qquad \frac{}{me, ve \vdash x \Downarrow ve((x))} \qquad \frac{}{me, ve \vdash \texttt{state}(x) \Downarrow \lfloor me(x) \rfloor}$$

$$\frac{me, ve \vdash e \Downarrow \lfloor v_e \rfloor \quad [\![\diamond]\!]_{\mathsf{type}\, e}\ v_e \doteq v}{me, ve \vdash \diamond e \Downarrow \lfloor v \rfloor}$$

$$\frac{me, ve \vdash e_1 \Downarrow \lfloor v_1 \rfloor \quad me, ve \vdash e_2 \Downarrow \lfloor v_2 \rfloor \quad [\![\oplus]\!]_{\mathsf{type}\, e_1 \times \mathsf{type}\, e_2}\ v_1\ v_2 \doteq v}{me, ve \vdash e_1 \oplus e_2 \Downarrow \lfloor v \rfloor}$$

$$\frac{}{me, ve \vdash [x] \Downarrow \lfloor ve(x) \rfloor}$$

(a) Expressions (exp_eval, src/Obc/ObcSemantics.v:88)

Figure 4.2 (I): Semantics of Obc

Statements update the state of the program. Their semantics is presented in figure 4.2b. We write $p, me, ve \vdash s \Downarrow (me', ve')$ to state that, in the program p, the statement s updates the memory me and the environment ve to me' and ve', respectively. A variable assignment updates the content of the environment with the value of the expression, provided that it is defined. Similarly, an assignment to a state variable updates the corresponding field of the memory. A conditional statement is evaluated to the result of the recursive evaluation of one of its branches, according to the value of the first expression. The sequence and no-operation statements semantics is self-explanatory. After the evaluation of its arguments to defined *or* undefined values $\boldsymbol{v}$, a call to a method named f on an instance i of class name c is defined using a mutually inductive semantic predicate. This predicate, described in details later, evaluates the call itself, updating the sub-memory of the instance if it exists or the empty memory if it does not exist (yet) to a new memory me'_i resulting from the call, and producing a list of return values $\boldsymbol{w}$. We define the following function, responsible for extracting a sub-memory.

Definition 4.1.1 (instance_match, src/Obc/ObcSemantics.v:48)

$$\mathsf{sub}\, i\ me \triangleq \begin{cases} \{\varnothing\} & \text{if } i \notin me.\mathsf{instances} \\ me[i] & \text{otherwise} \end{cases}$$

The memory before the call is updated with the produced sub-memory me'_i, while the environment is updated with left-hand side variables associated to the returned values $\boldsymbol{w}$. The returned values, as well as the input values, may be undefined.

Figure 4.2c presents the predicate responsible for evaluating a method call. First the program p is searched for a class *cls* together with the class declarations that come before p'.[1] Then we check that the method name f corresponds to an actual method m in the list of methods *cls*.**methods** of the class *cls*. The statement $m.\mathsf{body}$ constituting

[1] As usual in the implementation, the list is in reverse order of declaration.

$$\frac{me, ve \vdash e \Downarrow \lfloor v \rfloor}{p, me, ve \vdash x := e \Downarrow (me, ve\{x \mapsto v\})} \qquad \frac{me, ve \vdash e \Downarrow \lfloor v \rfloor}{p, me, ve \vdash \texttt{state}(x) := e \Downarrow (me(x \mapsto v), ve)}$$

$$\frac{me, ve \vdash e \Downarrow \lfloor \mathsf{T} \rfloor \quad p, me, ve \vdash s_t \Downarrow (me', ve')}{p, me, ve \vdash \texttt{if } e \texttt{ \{ } s_t \texttt{ \} else \{ } s_f \texttt{ \}} \Downarrow (me', ve')}$$

$$\frac{me, ve \vdash e \Downarrow \lfloor \mathsf{F} \rfloor \quad p, me, ve \vdash s_f \Downarrow (me', ve')}{p, me, ve \vdash \texttt{if } e \texttt{ \{ } s_t \texttt{ \} else \{ } s_f \texttt{ \}} \Downarrow (me', ve')}$$

$$\frac{p, me, ve \vdash s_1 \Downarrow (me_1, ve_1) \quad p, me_1, ve_1 \vdash s_2 \Downarrow (me_2, ve_2)}{p, me, ve \vdash s_1 \texttt{ ; } s_2 \Downarrow (me_2, ve_2)}$$

$$\frac{}{p, me, ve \vdash \texttt{skip} \Downarrow (me, ve)}$$

$$\frac{me, ve \vdash \boldsymbol{e} \Downarrow \boldsymbol{v} \quad p, \mathsf{sub}\, i\ me \vdash c.f\,(\boldsymbol{v}) \overset{\boldsymbol{w}}{\Downarrow} me'_i}{p, me, ve \vdash \boldsymbol{x} := i^c.f\,(\boldsymbol{e}) \Downarrow \left(me\left[i \mapsto me'_i\right], ve\{\boldsymbol{x} \mapsto \boldsymbol{w}\}\right)}$$

(b) Statements (stmt_eval, src/Obc/ObcSemantics.v:115)

$$\frac{\begin{array}{c}\mathsf{class}(p, c) \doteq (cls, p') \quad \mathsf{method}(cls, f) \doteq m \\ p', me, \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{v}\} \vdash m.\mathsf{body} \Downarrow (me', ve') \quad ve'((\boldsymbol{y})) = \boldsymbol{w}\end{array}}{p, me \vdash c.f\,(\boldsymbol{v}) \overset{\boldsymbol{w}}{\Downarrow} me'} \quad \text{where} \quad \begin{array}{l} m.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}} \\ m.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}}\end{array}$$

(c) Method calls (stmt_call_eval, src/Obc/ObcSemantics.v:145)

$$\frac{p, me \vdash c.f\,(xs_n) \overset{ys_n}{\Downarrow} me' \quad p, me' \vdash c.f\,(xs) \overset{n+1}{\circledcirc} ys}{p, me \vdash c.f\,(xs) \overset{n}{\circledcirc} ys}$$

(d) Loop (loop_call, src/Obc/ObcSemantics.v:161)

Figure 4.2 (II): Semantics of Obc

the body of the method is evaluated starting from the provided memory and a fresh environment created by binding the formal parameters $m.\mathbf{in}$ of m to the list of defined or undefined values $\boldsymbol{v}$. The evaluation of the body produces an updated sub-memory me' and a new environment ve' from which the output parameters $m.\mathbf{out}$ are looked up to retrieve the list of defined or undefined return values $\boldsymbol{w}$.

Finally, figure 4.2d presents a coinductive predicate that co-iterates a method call over streams of input and output (defined or undefined) values.

4.2 Scheduling the transition constraints of Stc

While the order of transition constraints in an Stc system is unimportant from a semantic point-of-view, the translation presented in the next section works syntactically to produce Obc statements where the order of evaluation is significant, as we have seen. Thus, translation is preceded by a scheduling pass that orders the transition constraints to ensure the correctness of the generated Obc code and also the efficacy of the subsequent fusion optimization. The fact that our semantic models for Lustre, NLustre and Stc do not depend on the order of the dataflow equations / transition constraints is fundamental: it makes verifying the correctness of the scheduling trivial.

In the usual modular compilation scheme, Lustre equations are scheduled so that the variables defined by basic equations and node instantiations are written before being read, and the variables defined by `fby` equations are read before being written. Additionally, the scheduler tries to group together equations with similar clock annotations and control expressions to maximize the later fusion of adjacent conditional statements. Here, scheduling is adapted readily to Stc. Listing 4.2b shows the result of scheduling on the running example recalled in listing 4.2a.

We adapt the approach of Auger et al. (2012) and Bourke, Brun, Dagand, et al. (2017), which applies a heuristic implemented in OCaml to find a suitable ordering, a sorting function in Coq whose output is guaranteed to be a permutation of its input, and a verified translation validator to establish the required well-scheduling predicate or signal an error. The only non-trivial change for Stc is to ensure that reset transitions are executed before corresponding default transitions, reflecting the hard-coded scheduling policy in the semantics of Stc (see section 3.2).

4.2.1 The well-scheduling predicate

Before presenting the well-scheduling predicate, we first need to define some useful sets.[2] Note that for all the sets we define, we overload the notation for lists in the natural way, that is, we write $\mathsf{S}(\boldsymbol{a})$ for the set $\bigcup_{a \in \boldsymbol{a}} \mathsf{S}(a)$. We begin by defining the set of *free* variables of a transition constraint.[3]

[2] In this presentation, we use the usual set notation for readability but the actual Coq implementation uses inductive relations for convenient reasoning.

[3] We do not give the definitions of the set of free variables of clocks, expressions and control expressions; they are unsurprising.

```
system euler {
  init i = true, px = 0.;
  transition(x0: float64, u: float64) returns (x: float64)
  {
    next i = false;
    x = if i then x0 else px;
    next px = x + 0.1 * u;
  }
}

system ins {
  init k = 0, px = 0.;
  sub xe: euler;
  transition(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64 when not alarm;
  {
    next k = k + 1;
    alarm = (k >= 50);
    xe = euler<xe,0>(gps when not alarm, xv when not alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}

system nav {
  init c = true, r = false;
  sub insr: ins;
  transition(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64 when not c, alr: bool when not c;
  {
    (insr, alr) = ins<insr,1>(gps when not c, xv when not c);
    reset ins<insr> every (. on r);
    x = merge c (gps when c) insr;
    alarm = merge c (false when c) alr;
    cm = merge c (not s when c) (s when not c);
    next c = cm;
    next r = s and c;
  }
}
```

Stc

(a) Before

Listing 4.2 (I): Scheduling of the example

```
system euler {
  init i = true, px = 0.;
  transition(x0: float64, u: float64) returns (x: float64)
  {
    x = if i then x0 else px;
    next i = false;
    next px = x + 0.1 * u;
  }
}

system ins {
  init k = 0, px = 0.;
  sub xe: euler;
  transition(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64 when not alarm;
  {
    alarm = (k >= 50);
    next k = k + 1;
    xe = euler<xe,0>(gps when not alarm, xv when not alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}

system nav {
  init c = true, r = false;
  sub insr: ins;
  transition(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64 when not c, alr: bool when not c;
  {
    reset ins<insr> every (. on r);
    next r = s and c;
    (insr, alr) = ins<insr,1>(gps when not c, xv when not c);
    cm = merge c (not s when c) (s when not c);
    x = merge c (gps when c) insr;
    alarm = merge c (0 when c) alr;
    next c = cm;
  }
}
```

Stc

(b) After

Listing 4.2 (II): Scheduling of the example

Definition 4.2.1 (Is_free_in_tc, src/Stc/StcIsFree.v:33)

$$\begin{aligned}
\mathsf{Free}\,(x =_{ck} e) &\triangleq \mathsf{Free}\,(ck) \cup \mathsf{Free}\,(e)\\
\mathsf{Free}\,(\texttt{next}\ x =_{ck} e) &\triangleq \mathsf{Free}\,(ck) \cup \mathsf{Free}\,(e)\\
\mathsf{Free}\,(\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck) &\triangleq \mathsf{Free}\,(ck)\\
\mathsf{Free}\,(\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})) &\triangleq \mathsf{Free}\,(ck) \cup \mathsf{Free}\,(\boldsymbol{e})
\end{aligned}$$

We also define the set of *defined* variables, that is, the set of standard variables or state variables that appear on the left-hand sides of transition constraints.

Definition 4.2.2 (Is_defined_in_tc, src/Stc/StcIsDefined.v:35)

$$\begin{aligned}
\mathsf{Def}\,(x =_{ck} e) &\triangleq \{x\}\\
\mathsf{Def}\,(\texttt{next}\ x =_{ck} e) &\triangleq \{x\}\\
\mathsf{Def}\,(\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck) &\triangleq \emptyset\\
\mathsf{Def}\,(\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})) &\triangleq \{\boldsymbol{x}\}
\end{aligned}$$

Then, we define the set of standard variables that appear on the left-hand side of—non **next**—transition constraints.

Definition 4.2.3 (Is_variable_in_tc, src/Stc/StcIsVariable.v:30)

$$\begin{aligned}
\mathsf{Var}\,(x =_{ck} e) &\triangleq \{x\}\\
\mathsf{Var}\,(\texttt{next}\ x =_{ck} e) &\triangleq \emptyset\\
\mathsf{Var}\,(\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck) &\triangleq \emptyset\\
\mathsf{Var}\,(\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})) &\triangleq \{\boldsymbol{x}\}
\end{aligned}$$

Finally, we define the set of sub-instances that appear with a parameter indicating their *level* in a transition constraint. The level indicates the position in the scheduling policy: here our *ad-hoc* model fixes the reset transition to have level 0 and the default transition to have level 1, but we wanted to prepare scheduling for a more general model.

Definition 4.2.4 (Is_sub_in_tc, src/Stc/StcSyntax.v:46)

$$\begin{aligned}
\mathsf{Sub}\,(x =_{ck} e) &\triangleq \emptyset\\
\mathsf{Sub}\,(\texttt{next}\ x =_{ck} e) &\triangleq \emptyset\\
\mathsf{Sub}\,(\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck) &\triangleq (i, 0)\\
\mathsf{Sub}\,(\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})) &\triangleq (i, 1)
\end{aligned}$$

The well-scheduling predicate itself is shown in figure 4.3. It is relative to a list of inputs ***ins*** and a set of state variables *regs*, that are both readily constructed for a system s from, respectively, s.**in** and s.**inits**. Note that the transition constraints must be sorted in the reverse order of their realization in the generated code, as it simplifies the

$$\frac{}{\mathsf{WellSch}^{\boldsymbol{ins}}_{regs}\ \varepsilon} \qquad \frac{\mathsf{WellSch}^{\boldsymbol{ins}}_{regs}\ \boldsymbol{tcs} \quad \forall x \in \mathsf{Free}\,(tc)\,,\ \begin{cases} x \notin \mathsf{Def}\,(\boldsymbol{tcs}) & \text{if } x \in regs \\ x \in \mathsf{Var}\,(\boldsymbol{tcs}) \cup \boldsymbol{ins} & \text{otherwise} \end{cases} \quad \forall\,(i,k) \in \mathsf{Sub}\,(tc)\ k',\ (i,k') \in \mathsf{Sub}\,(\boldsymbol{tcs}) \rightarrow k' < k}{\mathsf{WellSch}^{\boldsymbol{ins}}_{regs}\ (tc \cdot \boldsymbol{tcs})}$$

Figure 4.3: Well-scheduled Stc transition constraints (`Is_well_sch`, src/Stc/StcWellDefined.v:49)

definition of the predicate. The empty list is well scheduled. For $tc \cdot \boldsymbol{tcs}$, $\boldsymbol{tcs}$ must be well scheduled, and for every free variable x in tc, if x is a state variable, then it must not be defined in $\boldsymbol{tcs}$—variables defined by `next`s must be read before being written,—otherwise, x must be defined by a basic or default transition constraint, or as an input—other variables must be written before being read. Finally if tc is a transition on i then $\boldsymbol{tcs}$ cannot contain another transition on i with a greater or equal level. In our model with only reset and default transitions, this boils down to forbidding the presence of a default transition on i in $\boldsymbol{tcs}$ if tc is a reset transition on i.

4.2.2 The verified scheduling validator

The approach that we take for scheduling the transition constraints is that of *translation validation*. To prove a function in an ITP, there are two basic approaches:

1. Write the function within the ITP and prove that the function respects a specification.

2. Implement the function in another language and write a verified validator in the ITP, that returns true on the result of the function if and only if that result satisfies the specification.

These two approaches give the same confidence for the result, so the choice depends, among other criteria, on what is easier to implement and prove.

Usually in Vélus, we adopt the first technique: the translation functions of most passes are directly proved in Coq. This is because the compilation scheme is simple enough so that it is particularly suitable to be implemented in Gallina, the purely functional language of Coq, and because we do not know how the translation validation approach could be adapted to these passes. Now, the scheduling function is precisely a counter-example: the scheduling problem is NP-complete and we want to implement a heuristic which is easier to do in OCaml, whereas a validator for the well-scheduling predicate is rather straightforward to write in Coq and not difficult to prove.

```
1  Definition check_var (defined: PS.t) (variables: PS.t) (x: ident) : bool :=
2    if PS.mem x mems then negb (PS.mem x defined) else PS.mem x variables.
3
4  Definition sub_tc (tc: trconstr) : option (ident * nat) :=
5    match tc with
6    | TcReset i _ _       => Some (i, 0)
7    | TcCall i _ _ _ _ _ => Some (i, 1)
8    | _ => None
9    end.
10
11 Definition check_sub (i: ident) (k: nat) (ik': ident * nat) : bool :=
12   negb (ident_eqb (fst ik') i) || Nat.ltb (snd ik') k.
13
14 Definition check_tc (tc: trconstr) (acc: bool * PS.t * PS.t * PNS.t)
15   : bool * PS.t * PS.t * PNS.t :=
16   match acc with
17   | (true, defs, vars, subs) =>
18     let b := PS.for_all (check_var defs vars) (free_in_tc tc PS.empty) in
19     let defs := ps_adds (defined_tc tc) defs in
20     let vars := ps_adds (variables_tc tc) vars in
21     match sub_tc tc with
22     | Some (i, k) =>
23       (PNS.for_all (check_sub i k) subs && b, defs, vars, PNS.add (i, k) subs)
24     | None => (b, defs, vars, subs)
25     end
26   | acc => acc
27   end.
28
29 Definition is_well_sch_tcs (args: list ident) (tcs: list trconstr) : bool :=
30   fst (fst (fst (fold_right check_tc
31                             (true, PS.empty, ps_from_list args, PNS.empty)
32                             tcs))).
```

Coq (src/Stc/StcSchedulingValidator.v:64-95)

Listing 4.3: The scheduling validator (`mems` is a global section variable)

4.2.2.1 Implementation

Basically the validator is a direct implementation of the well-scheduling predicate as it is presented in figure 4.3. Listing 4.3 presents the Coq implementation: it fits in only about 35 lines of code. The function `well_sch` at line 29 folds the `check_tc` function backwards over the list of transition constraints. The `check_tc` function (line 14) takes a single transition constraint `tc` and checks that it is well-scheduled with regard to the accumulator `acc`. This accumulator is a 4-tuple composed of a boolean holding the result of the validation so far and the sets of defined variables, standard variables and pairs of instances variables and levels of already processed transition constraints.[4] If the boolean

[4]These sets are implemented using the *MSetPositive* (coq.inria.fr/library/Coq.MSets.MSetPositive.html) library for the first two and *MSetList* (coq.inria.fr/library/Coq.MSets.MSetList.html) for the last one. Thus `PS.t` designates a set of `positive` integers—whose type `ident` is an alias to—while

is true, we begin (line 18) to check that all the free variables of the transition constraints respect the scheduling rules, using the function `check_var`. This function (line 1) checks that an identifier is not in the set of defined variables if it is a state variable and that it is in the set of standard variables otherwise. Compared to the corresponding antecedent in figure 4.3, note that the inputs are not present: this is because they are directly included in the set of standard variables. Then (lines 19 and 20), we update the defined and standard variables sets with the corresponding variables of the current transition constraint, for the next iteration of the fold. The last step is to check that no reset transition can occur after a default transition. The `sub_tc` function (line 4) determines the instance and level of the transition, if it is a default or reset transition. In that case, we additionally check that no transition occurs in the sub-instances set with the same instance name and a greater level, using the function `check_sub`[5] (line 11) and update the sub-instances set.

4.2.2.2 Proof of correctness

The validator must be verified, that is, we have to prove that it respects the specification given by the well-scheduling predicate. We establish the following theorem, that states that the validator is a decision procedure for the well-scheduling predicate.

Theorem 4.2.1 (well_sch_spec, src/Stc/StcSchedulingValidator.v:384)
Given a list of inputs $\boldsymbol{ins}$*, a set of state variables regs and a list of transition constraints* $\boldsymbol{tc}$*, then*

$$\text{is-well-sch-tcs}\ regs\ \boldsymbol{ins}\ \boldsymbol{tc} = true \leftrightarrow \mathsf{WellSch}^{regs}_{\boldsymbol{ins}}\ \boldsymbol{tc}$$

4.2.3 The external scheduler

4.2.3.1 The Coq interface

The idea is to provide a generic way to use an external scheduler, written in OCaml, that interfaces with the Coq development. Thus the Coq functors responsible for the scheduling stage are parameterized over a module containing a single function `schedule` whose declaration is given in listing 4.4a. This function takes the name of the Stc system (only used for error reporting) and the list of transition constraints to be scheduled, and returns a list of positive integers that give the absolute position of each transition constraint in the ordering.

This external scheduler function is used within the `schedule_tcs` Coq function shown in listing 4.4b. The function `ocombine` combines the list of positives resulting from the call to the external scheduler `Sch.schedule` with the initial list of transition constraints giving a list of pairs, or failing with `None` if the lists are of different lengths. This

`PNS.t` denotes a set of `positive * nat` pairs.

[5] The Coq functions `negb`, `ident_eqb` and `Nat.ltb` are respectively the boolean negation, the boolean equality between two positive and the boolean less-than comparison.

```
Parameter schedule : ident -> list trconstr -> list positive.
```

Coq (src/Stc/StcSchedule.v:66)

(a) The parameterized scheduling function

```
Definition schedule_tcs (f: ident) (tcs: list trconstr) : list trconstr :=
  let sch := Sch.schedule f tcs in
  match ocombine sch tcs with
  | None        => tcs
  | Some schtcs => map snd (SchSort.sort schtcs)
  end.
```

Coq (src/Stc/StcSchedule.v:153-158)

(b) The scheduling wrapper

Listing 4.4: The Coq interface

composite list is then reverse-sorted[6] with the *Mergesort*[7] from the standard library, using comparison on the first positive projection of the pairs. Once the composite list is sorted, the *now sorted* list of transition constraints is obtained by stripping away the positive indexes.

We show the following property.

Lemma 4.2.2 (schedule_tcs_permutation, src/Stc/StcSchedule.v:160)
The list of scheduled transition constraints is a permutation of the initial list of transition constraints.

We directly extend the scheduling function to systems and programs and show that scheduling preserves the semantics. We write $\mathsf{sch}\, P$ to designate the scheduled program obtained from P by scheduling the transition constraints of all its system declarations.

Theorem 4.2.3 (scheduler_sem_system, src/Stc/StcSchedule.v:416)
Given a program P, a name f, two lists of values $\boldsymbol{xs}$ and $\boldsymbol{ys}$ and two states S and S' such that $P, S, S' \vdash f\,(\boldsymbol{xs}) \Downarrow\!\Downarrow \boldsymbol{ys}$, then

$$\mathsf{sch}\, P, S, S' \vdash f\,(\boldsymbol{xs}) \Downarrow\!\Downarrow \boldsymbol{ys}$$

We directly deduce the preservation of the looping semantics.

Corollary 4.2.3.1 (scheduler_loop, src/Stc/StcSchedule.v:433)
Given a program P, a name f, two streams of lists of values xs and ys and a states S such that for all instants n, $P, S \vdash f\,(xs) \overset{n}{\bigcirc} ys$, then

$$\mathsf{sch}\, P, S \vdash f\,(xs) \overset{n}{\bigcirc} ys$$

[6] We want the transition constraints in descending order to match the specification of the well-scheduling predicate.

[7] coq.inria.fr/library/Coq.Sorting.Mergesort.html

4.2.3.2 The OCaml implementation

The algorithm used to sort the transition constraints of Stc is a direct adaptation of the topological sort variant used in [Bourke, Brun, Dagand, et al. (2017)]. The OCaml implementation is about 400 lines of code (LoC) (src/Stc/stclib.ml), that is, about ten times larger than the Coq checker, confirming our choice of the translation validation approach. The algorithm is a variant of the well-known topological sort algorithm [Kahn (1962)] that uses a particular queue structure based on activation clock sub-dependencies. A heuristic tries to group together transition constraints that are on the same clock, in order to maximize the fusion optimization that we describe later. Essentially this optimization tries to fuse adjacent conditionals statements on the same guard, in order to reduce the number of branching constructs in the generated code.

To cope with reset and default transitions scheduling, the only modification is in the construction of the dependency graph where we add a dependency edge from a default transition to a reset transition on the same instance. The core of the algorithm does not change.

4.3 Translating Stc to Obc

The translation function, named s-tr, generates an Obc class for each Stc system. The class has the same name as the system, a register field for each system state variable, an instance field for each sub-system declaration and two methods *reset* and *step*. The *step* method is obtained by translating the system transition constraints into guarded assignments, with NLustre/Stc control expressions translated into conditional statements and expressions into Obc expressions.

We will now explain in detail the formal definitions of the translation functions. To facilitate the understanding, we rely on the running example recalled in listing 4.5a and its translation in listing 4.5b.

In figure 4.4a, we can see how the bodies of the *reset* and *step* methods are generated. The body of the *reset* method is the composition of two statements: (1) a sequence of assignments to the declared state variables with their initial constant values, and (2) a sequence of calls to the *reset* methods of the declared sub-systems. Consider the generated *reset* method for the class `ins`, at line 29 in listing 4.5b: the state variables `k` and `px` are assigned with their initial values declared at lines 12 in the original Stc program in listing 4.5a, then the *reset* method of the sub-instance `xe`, corresponding to the Stc sub-system declared at line 13, is called.

The body of the *step* method is obtained by sequencing—in reverse order for consistency with the scheduling pass—the list of translated transition constraints $s.\mathbf{tcs}$ of a system s. The folded function translates a single transition constraint with s-tr-tc and composes the result before the accumulated result of previous translations. The two parameters *regs* and Ω are instantiated with, respectively, the names of the state variable declarations of s and the clocking environment obtained from the input, output and local variable declarations. They are used in the translation of expressions.

Figure 4.4b presents the translation of a single transition constraint. The function has a

```
system euler {
  init i = true, px = 0.;
  transition(x0: float64, u: float64) returns (x: float64)
  {
    x = if i then x0 else px;
    next i = false;
    next px = x + 0.1 * u;
  }
}

system ins {
  init k = 0, px = 0.;
  sub xe: euler;
  transition(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64 when not alarm;
  {
    alarm = (k >= 50);
    next k = k + 1;
    xe = euler<xe,0>(gps when not alarm, xv when not alarm);
    x = merge alarm (px when alarm) xe;
    next px = x;
  }
}

system nav {
  init c = true, r = false;
  sub insr: ins;
  transition(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64 when not c, alr: bool when not c;
  {
    reset ins<insr> every (. on r);
    next r = s and c;
    (insr, alr) = ins<insr,1>(gps when not c, xv when not c);
    cm = merge c (not s when c) (s when not c);
    x = merge c (gps when c) insr;
    alarm = merge c (0 when c) alr;
    next c = cm;
  }
}
```

Stc

(a) Before

Listing 4.5 (I): Translation of the example

```
class euler {
  state i: bool;
  state px: float64;

  step(x0: float64, u: float64) returns (x: float64) {
    if state(i) { x := x0 } else { x := state(px) };
    state(i) := false;
    state(px) := x + 0.1 * u
  }

  reset() { state(i) := true; state(px) := 0. }
}

class ins {
  instance xe: euler;
  state k: int32;
  state px: float64;

  step(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64
  {
    alarm := state(k) >= 50;
    state(k) := state(k) + 1;
    if alarm { } else { xe := euler(xe).step([gps], [xv]) };
    if alarm { x := state(px) } else { x := xe };
    state(px) := x
  }

  reset() { state(k) := 0; state(px) := 0.; euler(xe).reset() }
}

class nav {
  instance insr: ins;
  state c: bool;
  state r: bool;

  step(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64, alr: bool
  {
    if state(r) { ins(insr).reset() } else { };
    state(r) := s and state(c);
    if state(c) { } else { insr, alr := ins(insr).step([gps], [xv]) };
    if state(c) { cm := not s } else { cm := s };
    if state(c) { x := gps } else { x := insr };
    if state(c) { alarm := false } else { alarm := alr };
    state(c) := cm
  }

  reset() { state(c) := true; state(r) := false; ins(insr).reset() }
}
```

Obc

(b) After (**skip** statements are not printed)

Listing 4.5 (II): Translation of the example

$$\begin{aligned}
&\textsf{reset-body}\ s \triangleq \textsf{reset-inits}\ s.\textbf{inits}\ ;\ \textsf{reset-subs}\ s.\textbf{subs}\\
&\qquad\text{where}\quad \textsf{reset-inits}\ \boldsymbol{inits} \triangleq \textsf{fold-l}\left(\lambda\ st\ x^{c,ck}.\ st\ ;\ \texttt{state}(x)\ \texttt{:=}\ c\right)\ \boldsymbol{inits}\ \texttt{skip}\\
&\qquad\qquad\quad\ \textsf{reset-subs}\ \boldsymbol{subs} \triangleq \textsf{fold-l}\left(\lambda\ st\ i^{f}.\ st\ ;\ i^{f}\texttt{.reset()}\right)\ \boldsymbol{subs}\ \texttt{skip}\\
&\textsf{step-body}\ s \triangleq \textsf{fold-l}\ (\lambda\ st\ tc.\ \textsf{s-tr-tc}^{regs}_{\Omega}\ tc\ ;\ st)\ \ s.\textbf{tcs}\ \texttt{skip}\\
&\qquad\qquad s.\textbf{inits} = \boldsymbol{y^{c_y, ck_y}}\\
&\qquad\text{where}\quad regs = \textsf{set-of-list}\ \boldsymbol{y}\\
&\qquad\qquad \Omega = \textsf{fold-l}\left(\lambda\ \Omega\ x^{t,ck}.\ \Omega\{x \mapsto ck\}\right)\ (s.\textbf{in} + s.\textbf{vars} + s.\textbf{out})\ \varnothing
\end{aligned}$$

(a) Generation of the special *reset* and *step* methods bodies (reset_method, src/StcToObc/Translation.v:154 and step_method, src/StcToObc/Translation.v:121)

Figure 4.4 (I): Translation of Stc to Obc

case for each of the four forms of transition constraint. Each case applies the ctrl function to transform the clock annotation *ck* into a nesting of conditional statements that will control the activation of an assignment or method call. A basic transition constraint is translated into an assignment of the translation of its expression to a standard variable. For example, the basic transition constraint at line 17 is translated into the assignment at line 22. A **`next`** transition constraint is translated into an assignment of the translation of its expression to a state variable. A default transition is translated into a call to the *step* method of the corresponding instance whose class is taken from the name of the Stc system. In the example, the default transition at line 19 is translated into the *step* method call at line 24 wrapped into a conditional introduced by the ctrl function since the original transition constraint is activated only when `alarm` is false. A reset transition is translated into a call to the *reset* method. For example, the reset transition at line 31 is translated into the conditional *reset* method call at line 40. In the translation of the arguments of a method call, we do not directly use s-tr-exp but rather s-tr-arg generalized to lists. This function adds validity assertions around standard variables with the same clock as the transition constraint. We know that such variables are defined when the method is called. This detail has no impact on the correctness proof between Stc and Obc: it is in anticipation of the correctness proof between Obc and Clight. Note, though, that this function is responsible for the addition of the clocking environment parameter Ω.

Control expressions are translated by the s-tr-cexp function shown in figure 4.4c, which takes a variable to be assigned as first parameter. This variable is taken from the left-hand side of a basic transition constraint, the only kind of transition constraint to allow control expressions on the right-hand side. The **`merge`** and **`if/then/else`** constructs are translated into conditional statements, propagating recursively the assignment variable through both branches until a simple expression is translated and assigned to the variable. For example, the basic transition constraint at line 35 is translated into the conditional

$$\textsf{s-tr-tc}^{regs}_{\Omega}\ (x =_{ck} e) \triangleq \textsf{ctrl}^{regs}\ ck\ (\textsf{s-tr-cexp}^{regs}(x, e))$$
$$\textsf{s-tr-tc}^{regs}_{\Omega}\ (\texttt{next}\ x =_{ck} e) \triangleq \textsf{ctrl}^{regs}\ ck\ (\texttt{state(}x\texttt{)}\ \texttt{:=}\ \textsf{s-tr-exp}^{regs}\ e)$$
$$\textsf{s-tr-tc}^{regs}_{\Omega}\ (\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>(}\boldsymbol{e}\texttt{)}) \triangleq \textsf{ctrl}^{regs}\ ck\ \left(\boldsymbol{x}\ \texttt{:=}\ i^{f}\ \texttt{.}\ \textsf{step}\texttt{(}\ \textsf{s-tr-arg}^{regs}_{\Omega}(ck, \boldsymbol{e})\ \texttt{)}\right)$$
$$\textsf{s-tr-tc}^{regs}_{\Omega}\ (\texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck) \triangleq \textsf{ctrl}^{regs}\ ck\ \left(i^{f}\ \texttt{.}\ \textsf{reset}\texttt{()}\right)$$

$$\textsf{ctrl}^{regs} \bullet s \triangleq s$$
$$\textsf{ctrl}^{regs}\ (ck\ \texttt{on}\ (x = \textsf{true}))\ s \triangleq \textsf{ctrl}^{regs}\ ck\ (\texttt{if}\ (\textsf{s-tr-var}^{regs}\ x^{\texttt{bool}})\ \texttt{\{}\ s\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ \texttt{skip}\ \texttt{\}})$$
$$\textsf{ctrl}^{regs}\ (ck\ \texttt{on}\ (x = \textsf{false}))\ s \triangleq \textsf{ctrl}^{regs}\ ck\ (\texttt{if}\ (\textsf{s-tr-var}^{regs}\ x^{\texttt{bool}})\ \texttt{\{}\ \texttt{skip}\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ s\ \texttt{\}})$$
$$\textsf{s-tr-arg}^{regs}_{\Omega}(ck, e) \triangleq \begin{cases} [x]^{\tau} & \text{if } e = x^{\tau} \wedge x \notin regs \wedge \Omega(x) = ck \\ \textsf{s-tr-exp}^{regs}\ e & \text{otherwise} \end{cases}$$

(b) Transition constraints (translate_tc, src/StcToObc/Translation.v:93)

$$\textsf{s-tr-cexp}^{regs}(x, \texttt{merge}\ y\ e_t\ e_f) \triangleq \texttt{if}\ \ (\textsf{s-tr-var}^{regs}\ y^{\texttt{bool}})\ \ \texttt{\{}\ \ \textsf{s-tr-cexp}^{regs}(x, e_t)\ \ \texttt{\}}$$
$$\texttt{else}\ \texttt{\{}\ \ \textsf{s-tr-cexp}^{regs}(x, e_f)\ \ \texttt{\}}$$
$$\textsf{s-tr-cexp}^{regs}(x, \texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f) \triangleq \texttt{if}\ \ (\textsf{s-tr-exp}^{regs}\ e)\ \ \texttt{\{}\ \ \textsf{s-tr-cexp}^{regs}(x, e_t)\ \ \texttt{\}}$$
$$\texttt{else}\ \texttt{\{}\ \ \textsf{s-tr-cexp}^{regs}(x, e_f)\ \ \texttt{\}}$$
$$\textsf{s-tr-cexp}^{regs}(x, e) \triangleq x\ \texttt{:=}\ \textsf{s-tr-exp}^{regs}\ e$$

(c) Control expressions (translate_cexp, src/StcToObc/Translation.v:67)

$$\textsf{s-tr-var}^{regs}\ x^{t} \triangleq \text{if } x \in regs \text{ then } \texttt{state(}x\texttt{)}^{\tau} \text{ else } x^{\tau}$$
$$\textsf{s-tr-exp}^{regs}\ c \triangleq c$$
$$\textsf{s-tr-exp}^{regs}\ x^{\tau} \triangleq \textsf{s-tr-var}^{regs}\ x^{t}$$
$$\textsf{s-tr-exp}^{regs}\ (\diamond\, e)^{\tau} \triangleq (\diamond\, \textsf{s-tr-exp}^{regs}\ e)^{\tau}$$
$$\textsf{s-tr-exp}^{regs}\ (e_1 \oplus e_2)^{\tau} \triangleq (\textsf{s-tr-exp}^{regs}\ e_1 \oplus \textsf{s-tr-exp}^{regs}\ e_2)^{\tau}$$
$$\textsf{s-tr-exp}^{regs}\ (e\ \texttt{when}\ (x = b)) \triangleq \textsf{s-tr-exp}^{regs}\ e$$

(d) Expressions (translate_exp, src/StcToObc/Translation.v:58)

Figure 4.4 (II): Translation of Stc to Obc

at line 44. In particular, notice how the body of the *step* method of the `nav` class reveals the need for the fusion optimization further described in section 4.5: the translation generates a lot of successive conditionals that have the same guard, because of the heavy use of sampling and merging operations in the original Lustre source node (encoding a state machine).

Figure 4.4d presents the translation of expressions. The only technicality is the explicit distinction between standard variables and state variables in Obc. The s-tr-var function makes the choice according to membership in the set of state variables *regs*. Note that **`when`** conditions are outright dropped since conditional activation is now addressed at the statement level.

4.4 Translation correctness

In this section we describe the proof of translation correctness, that is, the proof of semantics preservation between an Stc system and its translation in Obc.

First, we present a fundamental property of the semantic model of Stc that is crucial to further proofs. Then we show the definition of correspondence predicates that relate an Stc state with an Obc state. The proof of correctness is split in two sub-sections: first we state the correctness result for a call to the generated *reset* method, then for the *step* method.

4.4.1 Fundamental property of Stc

The Obc conditional statements introduced by the ctrl function ensure that the expressions and assignment statements generated for a transition constraint are only executed when the associated Stc clock is true. In particular, for default transitions, this means that the corresponding instance field in the generated code is not changed when the clock is false. We must thus show that this is the behaviour specified by the Stc semantics. We do this by following the same proof scheme as for theorems 3.3.4 and 3.3.5 on page 93 for the NLustre memory semantics.

We first show an intermediate result on transition constraints.

Lemma 4.4.1 (sem_trconstrs_absent_states, src/Stc/StcSemantics.v:636)
Given a program P, a list of transition constraints $\boldsymbol{tcs}$, an environment R, and Stc states S, I and S' such that:

1. *S and S' are closed relative to the state variables and instances appearing in $\boldsymbol{tcs}$,*
2. *if a default transition on any instance i appears in $\boldsymbol{tcs}$ with parameter k, then $k = 1$ if and only if a reset transition on i also appears,*
3. *$\forall tc \in \boldsymbol{tcs},\ P, R, \mathsf{false}, S, I, S' \vdash tc$, and*

assuming the system induction hypothesis:

$$\forall f\ \boldsymbol{xs}\ \boldsymbol{ys}\ S\ S',\ (P, S, S' \vdash f\,(xs) \downdownarrows ys \wedge (\forall v \in \boldsymbol{xs},\ v = \langle\ \rangle)) \rightarrow S' \approx\!\!\approx S$$

then $S' \approx\!\!\approx S$.

We extend the intermediate result from lists of transition constraints to systems.

Theorem 4.4.2 (sem_system_absent, src/Stc/StcSemantics.v:768)
Given a well-ordered program P, a name f, lists of values $\boldsymbol{xs}$ and $\boldsymbol{ys}$ such that all values in $\boldsymbol{xs}$ are absent, and memory trees S and S' such that $P, S, S' \vdash f\,(\boldsymbol{xs}) \Downarrow\!\Downarrow \boldsymbol{ys}$, then $S' \approx\!\!\approx S$, and all values in $\boldsymbol{ys}$ are absent.

4.4.2 State correspondence relations

As usual when stating a proof of correctness for a compilation function between two languages, a state correspondence relation must be defined. In the proof for Stc code generation, this relation was direct since the same objects are used to describe states in both languages. Here, although the objects are still essentially the same (environments and memory trees), the respective semantics of the two languages do not constrain them in the same way. Hence, the relation is more involved, in particular to handle transient states.

We must relate present and absent values in Stc to defined or undefined values in Obc. We use two distinct relations. First we define a predicate that equates a present value with a defined value but allows absent to be related to any value. That is, when a value is absent in Stc, there is no constraint on the corresponding value in Obc.

Definition 4.4.1 (eq_if_present, src/StcToObc/Correctness.v:42)

$$\frac{}{\langle v \rangle \mathrel{\langle = \rangle} \lfloor v \rfloor} \qquad\qquad \frac{}{\langle\ \rangle \mathrel{\langle = \rangle} v}$$

The second relation, stronger, is a direct function.

Definition 4.4.2 (value_to_option, src/StcToObc/Correctness.v:51)

$$\langle v \rangle^{\lfloor\rfloor} \triangleq \lfloor v \rfloor$$
$$\langle\ \rangle^{\lfloor\rfloor} \triangleq \lfloor\ \rfloor$$

The state correspondence is organized around two key predicates. The first one compares an Stc environment R with an Obc memory tree me and environment ve. It uses the relation $\langle = \rangle$ only to require the correspondence for present values in R.

Definition 4.4.3 (equiv_env, src/StcToObc/Correctness.v:57)

$$R \overset{\mathcal{D}}{\underset{regs}{\Longleftrightarrow}} (me, ve) \triangleq \forall x \in \mathcal{D},\ R(x) \mathrel{\langle = \rangle} \begin{cases} me(\!(x)\!) & \text{if } x \in regs \\ ve(\!(x)\!) & \text{otherwise} \end{cases}$$

The predicate has a parameter *regs* which is a set of state variables to determine where the variable is to be constrained: in the values field of the memory tree me or in the environment ve. The $\mathcal{D}$ parameter specifies the variables that are constrained, we exploit it in inductive proofs to gradually expand the domain of the correspondence relation.

The next predicate relates the Stc states S, I and S' to an Obc memory tree me.

Definition 4.4.4 (Memory_Corres, src/StcToObc/StcMemoryCorres.v:45)

$$(S, I, S') \overset{\boldsymbol{tcs}}{\Longleftrightarrow} me \triangleq \forall x,\ me((x)) = \begin{cases} S'((x)) & \text{if } (\texttt{next } x \texttt{ = -}) \in \boldsymbol{tcs} \\ S((x)) & \text{otherwise} \end{cases}$$

$$\wedge\ \forall i,\ me[[i]]\ \lfloor\approx\rfloor \begin{cases} S[[i]] & \text{if } (\texttt{reset -<}i\texttt{> every -}) \notin \boldsymbol{tcs} \\ & \quad \wedge\ (\texttt{- = -<}i\texttt{, ->(-)}) \notin \boldsymbol{tcs} \\ I[[i]] & \text{if } (\texttt{reset -<}i\texttt{> every -}) \in \boldsymbol{tcs} \\ & \quad \wedge\ (\texttt{- = -<}i\texttt{, ->(-)}) \notin \boldsymbol{tcs} \\ S'[[i]] & \text{if } (\texttt{- = -<}i\texttt{, ->(-)}) \in \boldsymbol{tcs} \end{cases}$$

The predicate is parameterized by a list of transition constraints ***tcs***. The idea is to relate *me* to the Stc specification under the assumption that it results from executing the Obc code generated for the constraints in ***tcs***. A variable x is defined in *me* if and only if it is defined with the same value in either S', if ***tcs*** contains the corresponding **next** constraint, or S otherwise. We use the double parentheses to explicitly take failure into account: it expresses the "if and only if" part. Similarly, an instance is defined in *me* if and only if it is defined with an equivalent value either in S, if ***tcs*** contains neither the corresponding reset or default transition, in I, if ***tcs*** contains the reset transition but not the default transition, or in S', if ***tcs*** contains the default transition. Here we extend the double parentheses notation to brackets to model failure in the same way. Consequently, we use the notation $\lfloor\approx\rfloor$ to lift the memory equivalence in the following way.

Definition 4.4.5 (orel, src/Common/Common.v:1019)

$$\frac{}{\lfloor\ \rfloor\ \lfloor\approx\rfloor\ \lfloor\ \rfloor} \qquad\qquad \frac{M \approx M'}{\lfloor M \rfloor\ \lfloor\approx\rfloor\ \lfloor M' \rfloor}$$

4.4.3 The *reset* method call

A *reset* method comprises two phases. The first, generated by reset-inits, initializes state variables to their initial values. The second, generated by reset-subs, resets sub-instances by calling their generated *reset* methods. To reason about a call to a *reset* method and state properties about it, we focus on this composite nature.

We introduce the following specification function to describe the expected state of the memory after having initialized the state variables declarations given by an ***inits*** parameter.

Definition 4.4.6 (add_mems, src/StcToObc/Correctness.v:315)

$$\text{add-inits } \boldsymbol{inits}\ me \triangleq \text{fold-l}\left(\lambda\, me\ x^{t,c}.\ me(x \mapsto [\![c]\!])\right)\ \boldsymbol{inits}\ me$$

We can now state the correctness of the Obc statement generated by reset-inits.

Lemma 4.4.3 (reset_mems_spec, src/StcToObc/Correctness.v:385)
Given a program p, a memory tree me, an environment ve and a list of state variables declarations ***inits****, then* $p, me, ve \vdash \text{reset-inits } \boldsymbol{inits} \Downarrow (\text{add-inits } \boldsymbol{inits}\ me, ve)$

Then we can extend the specification to the statement generated by the reset-body function which combines the results of reset-inits and reset-subs.

Lemma 4.4.4 (translate_reset_comp, src/StcToObc/Correctness.v:397)
Given a program p, a system s, memory trees me and me' and environments ve and ve', then $p, me, ve \vdash$ reset-body $s \Downarrow (me', ve')$ if and only if:

- $p, me, ve \vdash$ reset-inits $s.\mathit{inits} \Downarrow$ (add-inits $s.\mathit{inits}$ me, ve)
- $p,$ add-inits $s.\mathit{inits}$ $me, ve \vdash$ reset-subs $s.\mathit{subs} \Downarrow (me', ve')$.

We state the correctness result for a call to a *reset* method.

Theorem 4.4.5 (reset_spec, src/StcToObc/Correctness.v:700)
Given a well-ordered program P containing a system s with name f, then for any memory tree me there exists a memory tree me' such that:

1. s-tr $P, me \vdash f.\mathit{reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me'$
2. initial-state $P\ f\ me'$
3. *if* s-closed $P\ f\ me$ *then* s-closed $P\ f\ me'$

4.4.4 The *step* method call

The proof of correctness for the *step* method is more involved than for the *reset* method. We describe it over the next few pages. We start with the correctness for expressions, extend it for the special case of transition arguments, present the invariants and reasoning for transition constraints and then extend the results to systems.

4.4.4.1 Expressions

Since Obc code is only executed when the source corresponding clock is true, we restrict ourselves to the case where the base clock is indeed true. The idea is to assume that the correspondence relation holds for all the free variables in an expression and then to show that the same result is obtained by both Stc and Obc.

Lemma 4.4.6 (exp_correct, src/StcToObc/Correctness.v:201)
Given an expression e, a value v, a set of state variables regs, an Stc environment R, a memory tree me and an environment ve such that $R \overset{\mathsf{Free}(e)}{\underset{regs}{\Longleftrightarrow}} (me, ve)$ and $R,$ true $\vdash e \downarrow \langle v \rangle$, then $me, ve \vdash$ s-tr-exp$^{regs}\ e \Downarrow \lfloor v \rfloor$.

For control expressions, the correctness result involves the semantics of statements since control expressions are translated into conditional statements.

Lemma 4.4.7 (cexp_correct, src/StcToObc/Correctness.v:233)
Given a control expression e, a value v, a set of state variables regs, an Stc environment R, a memory tree me and an environment ve such that $R \overset{\mathsf{Free}(e)}{\underset{regs}{\Longleftrightarrow}} (me, ve)$ and $R,$ true $\vdash_c e \downarrow \langle v \rangle$, then for any program p and variable x,
$p, me, ve \vdash$ s-tr-cexp$^{regs}(x, e) \Downarrow (me, ve\{x \mapsto v\})$.

$$\frac{}{\mathsf{noops} \bullet e} \qquad \frac{}{\mathsf{noops}\ ck\ c} \qquad \frac{}{\mathsf{noops}\ ck\ x^{\tau}} \qquad \frac{\mathsf{noops}\ ck\ e}{\mathsf{noops}\ (ck\ \texttt{on}\ (x = b))\ (e\ \texttt{when}\ (y = b))}$$

Figure 4.5: Normalization condition between an argument and its clock (noops_exp, src/CoreExpr/CESyntax.v:47)

4.4.4.2 Default transitions arguments

The correctness lemma 4.4.6 about expressions is rather restricted: it only holds when the base clock is true and the expression has a present value in Stc. This suffices for basic, next and reset transition constraints. For the first two since their translations into Obc give guarded (state) assignments and translated (control) expressions are only ever evaluated when they would have a present value in the original Stc constraint, and for the last one since it contains no expression. But for default transitions, the arguments can be on different clocks, that is, a Stc system can be instantiated on values that may be absent when the transition clock is true. This is the very reason why the semantics of Obc takes undefined values into account. But, as we saw, the possibility of being *undefined* is limited to variables, that is, when the variable `x` is not defined, the expression `x` has a semantics, but `3 / x` does not. Now consider the following Stc default transition:

```
y = f<i,0>(ck, (3 when ck) / x)
```

Assume that `y` and `ck` are on the base clock, and `x` is on clock `ck`. The translation to Obc involves two cases. First, if `x` is not a state variable then we obtain:

```
y := (i:f).step(ck, 3 / x)
```

When `ck` is false, the Stc transition constraint has a semantics since `3 / x` evaluates to the absent value, but the Obc statement does not, since `3 / x` has no meaning when `x` is not defined. Thus it is impossible to show translation correctness in this case. Second, if `x` is a state variable:

```
y := (i:f).step(ck, 3 / state(x))
```

This time when `ck` is false, the Obc expression `3 / state(x)` has a semantics in general since a state variable is always defined given that its previous value is available in the memory. But, what if this previous value is zero? Then the expression has no meaning either, and a division by zero that never occurs in the source semantics does occur in the generated code, making it not possible to show translation correctness in this case either.

Figure 4.5 presents the normalization condition,[8] introduced in [Bourke and Pouzet (2019)] that the arguments passed to a NLustre node instantiation must meet to avoid the problem described above. It relates an expression passed as an argument to the corresponding clock declaration in the instantiated node. Any expression on the base clock is accepted since the base clock is the fastest execution clock of the instantiated node. A constant is always defined, and so is a variable. Finally, a sampled expression is recursively checked: for each level of sampling present in the node declaration, there must be a corresponding level of sampling in the argument expression. This condition

[8] Here presented as an inductive relation for legibility, it is implemented as a Coq recursive function.

eliminates all the expressions that can lead to unmeaningful computations in the generated imperative code and suffices to prove the corresponding obligations in the correctness proof.

This syntactic condition is checked on the source NLustre program and readily propagated trough the translation to Stc and the scheduling of Stc transition constraints.

Lemma 4.4.8 (translate_normal_args, src/NLustreToStc/NL2StcNormalArgs.v:57)
Given an NLustre program G that respects the normalization condition on node instantiation arguments, then so does the translation to Stc $\mathsf{i\text{-}tr}\, G$.

Lemma 4.4.9 (scheduler_normal_args, src/Stc/StcSchedule.v:478)
Given an Stc program P that respects the normalization condition on default transition arguments, then so does the scheduled program P.

Then, we can state and prove a dedicated correctness result for arguments of default transitions that accounts for absent/undefined values.

Lemma 4.4.10 (TcCall_check_args_translate_arg, src/StcToObc/Correctness.v:848)
Given a well-clocked default transition $\boldsymbol{x} =_{ck} f\langle i, k\rangle(\boldsymbol{e})$ respecting the normalization condition on its arguments, a list of present or absent values $\boldsymbol{v}$, a set of state variables regs, a clocking environment Ω, an Stc environment R, a memory tree me and an environment ve such that:

1. $R \overset{\mathsf{Free}(ck)\cup\mathsf{Free}(\boldsymbol{e})}{\Longleftrightarrow}_{regs} (me, ve)$,
2. $\forall x \in regs,\ \exists v,\ me(x) = v$,
3. $R, b \vdash ck \downarrow \mathsf{true}$, *and*
4. $R, \mathsf{true} \vdash \boldsymbol{e} \downarrow \boldsymbol{v}$,

then[9] *there exists a list of defined or undefined values $\boldsymbol{w}$ such that $\boldsymbol{v} \mathrel{‹=›} \boldsymbol{w}$ and $me, ve \vdash \mathsf{s\text{-}tr\text{-}arg}^{regs}_{\Omega}(ck, \boldsymbol{e}) \Downarrow \boldsymbol{w}$.*

This result is a generalization of an intermediate result about a single expression verifying the normalization condition, rather than a list. The first hypothesis, as for the previous results on expressions and control expressions, guarantees the correspondence for variables and state variables between R and the pair (me, ve). The second one expresses that all variables in *regs* are defined in me. The third one ensures that ck is true, that is, the transition is activated. Finally, the last hypothesis is the main one: all the Stc argument expressions have a semantics that is not constrained to present values.

This lemma shows the importance of the $‹=›$ relation: while we might expect the witness list to be $\boldsymbol{v}^{\lfloor\rfloor}$, this would contradict the semantics in the case of state variables. Consider, for example, the following Stc default transition, which respects the normalization condition on the arguments of default transitions:

[9] Under additional omitted clocking constraints, notably on Ω.

```
y = f<i,0>(ck, x)
```

Assume as before that `y` and `ck` are on the base clock and that `x` is on clock `ck`. Then, if `x` is a state variable, the following Obc code is generated:

```
y := (i:f).step(ck, state(x))
```

If `ck` is false, then the Stc variable `x` is evaluated to the absent value $\langle\,\rangle$, but the state variable `state(x)` is not evaluated to the undefined value $\lfloor\,\rfloor = \langle\,\rangle^{\sqcup}$, but rather to the previous value available in the memory.

4.4.4.3 Transition constraints

Transition constraints are translated into conditional statements guarded by the source clock by the ctrl function (see figure 4.4b on page 116). We show the following two key results that relate the semantics of clocks in Stc to the execution of such statements. The first one expresses the fact that the conditional statement is not executed when the clock is false.

Lemma 4.4.11 (stmt_eval_Control_absent', src/StcToObc/Correctness.v:293)
Given a clock ck, a statement s, a set of state variables $regs$, an Stc environment R, a memory tree me and an environment ve such that $R \overset{\mathsf{Free}(ck)}{\underset{regs}{\Longleftrightarrow}} (me, ve)$ and $R, \mathit{true} \vdash ck \downarrow \mathit{false}$, then for any program p, $p, me, ve \vdash \mathsf{ctrl}^{regs}\ ck\ s \Downarrow (me, ve)$.

The second result expresses the converse: the conditional statement is executed when the clock is true.

Lemma 4.4.12 (stmt_eval_Control_present', src/StcToObc/Correctness.v:301)
Given a clock ck, a statement s, a base clock b, a set of state variables $regs$, an Stc environment R, memory trees me and me', environments ve and ve' and a program p such that $R \overset{\mathsf{Free}(ck)}{\underset{regs}{\Longleftrightarrow}} (me, ve)$; $R, \mathit{true} \vdash ck \downarrow \mathit{true}$; and $p, me, ve \vdash s \Downarrow (me', ve')$; then $p, me, ve \vdash \mathsf{ctrl}^{regs}\ ck\ s \Downarrow (me', ve')$.

The proof of the correctness result for transition constraints is rather complex. We have to show both the preservation of the semantics and the preservation of the correspondence predicates. The sequential nature of Obc makes it impossible to reason about a single transition constraint and then to generalize to a list of transition constraints. Indeed, the memory correspondence given by definition 4.4.4 on page 119 is relative to a list of transition constraints, since it reflects the successive execution of the sequence of generated statements. Hence we start by considering the head element tc in a list of transition constraints $\boldsymbol{tcs}$. We want to show that, starting from an Obc program state (me, ve) in correspondence with an Stc environment R and states S, I and S', relative to the list $\boldsymbol{tcs}$, we can expose a new pair (me', ve') that is the result of executing the translated tc statement, and that the new pair remains in correspondence with R, S, I, and S'. While the environment correspondence for standard variables remains relatively easy to prove, the memory correspondence requires more work. We start by stating intermediate results tailored to each kind of transition constraint, in case of activation and non-activation.

Memory correspondence results

For the following lemmas (4.4.13 to 4.4.19) we fix an Stc state (S, I, S') and an Obc memory tree me, and we make the hypothesis that the correspondence holds between them, relative to a given list of transition constraints $\boldsymbol{tcs}$.

Hypothesis 4.4.1

$$(S, I, S') \overset{\boldsymbol{tcs}}{\Longleftrightarrow} me$$

All the results show the preservation of the correspondence by the addition of a transition constraint on top of $\boldsymbol{tcs}$. They are proved by systematic inspection of the involved definitions and simple applications of the hypotheses. Most hypotheses are relevant pieces of the Stc semantics of the added transition constraint.

The first result states that the memory correspondence is preserved by the addition of a basic transition constraint, activated or not, as it does not modify the memory.

Lemma 4.4.13 (Memory_Corres_Def, src/StcToObc/StcMemoryCorres.v:69)
Given a basic transition constraint $x =_{ck} e$, we have $(S, I, S') \overset{(x =_{ck} e)\cdot \boldsymbol{tcs}}{\Longleftrightarrow} me$.

For a **next** transition constraint, there are two cases. Either the transition constraint is activated, that is, S' holds the next value for the defined state variable, and we show the correspondence with an updated memory tree; or it is not activated, that is, the value in S is repeated in S' and the memory tree is not changed.

Lemma 4.4.14 (Memory_Corres_Next_present, src/StcToObc/StcMemoryCorres.v:93)
*Given a **next** transition constraint $\mathtt{next}\ x =_{ck} e$ and a value v such that $S'(x) = v$, then $(S, I, S') \overset{(\mathtt{next}\ x =_{ck} e)\cdot \boldsymbol{tcs}}{\Longleftrightarrow} me(x \mapsto v)$.*

Lemma 4.4.15 (Memory_Corres_Next_absent, src/StcToObc/StcMemoryCorres.v:130)
*Given a **next** transition constraint $\mathtt{next}\ x =_{ck} e$, if $S'(\!(x)\!) = S(\!(x)\!)$, then $(S, I, S') \overset{(\mathtt{next}\ x =_{ck} e)\cdot \boldsymbol{tcs}}{\Longleftrightarrow} me$.*

In the case of default and reset transitions, additional hypotheses on the presence of transitions on the same instance in the list of transition constraints are needed in some cases to properly apply hypothesis 4.4.1. These hypotheses are later discharged by reasoning from the well-scheduling predicate and some syntactic invariants.

For a default transition on an instance i, if the transition is activated, then the correspondence must hold for the memory tree updated with a sub-tree that is observationally equivalent to the corresponding sub-state in S'. If the transition is not taken, that is, the sub-state in S' is observationally equivalent to the sub-state in I, then the correspondence holds for an unchanged memory tree. A bit more work and some extra hypotheses are needed in this case to show that $me[[i]] \lfloor\approx\rfloor S'[[i]]$, since the presence of a reset transition on i in the list of transition constraints $\boldsymbol{tcs}$ has an impact on the conclusion obtained from hypothesis 4.4.1.

Lemma 4.4.16 (Memory_Corres_Call_present, src/StcToObc/StcMemoryCorres.v:232)
Given a default transition $\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})$ *and a memory tree* me_i*, if* $me_i \approx S'[i]$*, then* $(S, I, S') \overset{(\boldsymbol{x} =_{ck} f\texttt{<}i,k\texttt{>}(\boldsymbol{e}))\cdot \boldsymbol{tcs}}{\Longleftarrow} me[i \mapsto me_i]$.

Lemma 4.4.17 (Memory_Corres_Call_absent, src/StcToObc/StcMemoryCorres.v:271)
Given a default transition $\boldsymbol{x} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})$*, if* $S'[i] \approx I[i]$*;* $I[i] \approx S[i]$ *whenever* $k = 0$*;* $(\texttt{- = -<}i, \texttt{->(-)}) \notin \boldsymbol{tcs}$*; and* $(\texttt{reset -<}i\texttt{> every -}) \in \boldsymbol{tcs}$ *if and only if* $k = 1$*, then* $(S, I, S') \overset{(\boldsymbol{x} =_{ck} f\texttt{<}i,k\texttt{>}(\boldsymbol{e}))\cdot \boldsymbol{tcs}}{\Longleftarrow} me$.

Finally, for a reset transition on an instance i, the reasoning is similar to that of default transitions. If the transition is taken, then the correspondence must hold for the memory tree updated with a sub-tree that is observationally equivalent to the corresponding sub-state in the transient state I. If the transition is not taken, that is, the sub-state in I is observationally equivalent to the sub-state in S, then the correspondence holds for an unchanged memory tree.

Lemma 4.4.18 (Memory_Corres_Reset_present, src/StcToObc/StcMemoryCorres.v:158)
Given a reset transition `reset` $f\texttt{<}i\texttt{>}$ `every` *ck and a memory tree* me_i*, if* $me_i \approx I[i]$ *and* $(\texttt{- = -<}i, \texttt{->(-)}) \notin \boldsymbol{tcs}$*, then* $(S, I, S') \overset{(\texttt{reset } f\texttt{<}i\texttt{> every } ck)\cdot \boldsymbol{tcs}}{\Longleftarrow} me[i \mapsto me_i]$.

Lemma 4.4.19 (Memory_Corres_Reset_absent, src/StcToObc/StcMemoryCorres.v:199)
Given a reset transition `reset` $f\texttt{<}i\texttt{>}$ `every` *ck, if* $I[i] \approx S[i]$ *and* $(\texttt{reset -<}i\texttt{> every -}) \notin \boldsymbol{tcs}$*, then* $(S, I, S') \overset{(\texttt{reset } f\texttt{<}i\texttt{> every } ck)\cdot \boldsymbol{tcs}}{\Longleftarrow} me$.

Correctness results for transition constraints

The overall correctness result uses the already presented proof scheme: an induction on the program and the application of intermediate results on transition constraints that themselves use the induction hypothesis for the case of default transitions. Thus, before presenting these intermediate results, we fix the correctness induction hypothesis—that has exactly the same shape as the final correctness result—that holds for a given well-ordered Stc program P.

Hypothesis 4.4.2 (Induction hypothesis)
Given a name f*, a list of values* $\boldsymbol{xs}$ *which are not all absent, a list of values* $\boldsymbol{ys}$*, two states* S *and* S' *such that* $P, S, S' \vdash f\,(\boldsymbol{xs}) \downdownarrows \boldsymbol{ys}$*, a list of defined or undefined values* $\boldsymbol{xs'}$ *such that* $\boldsymbol{xs} \langle{=}\rangle \boldsymbol{xs'}$ *and a memory tree* $me \approx S$*, then there exists a memory tree* $me' \approx S'$ *such that* $\textsf{s-tr}\, P, me \vdash f.\textit{step}\,(\boldsymbol{xs'}) \overset{\boldsymbol{ys}\sqcup}{\Downarrow} me'$.

This correctness hypothesis states that if the Stc system named f in P has a semantics relating the states S and S' as well as the lists of present or absent values $\boldsymbol{xs}$ and $\boldsymbol{ys}$, then we can expose a memory tree that is observationally equivalent to the state S', and the result of executing the corresponding translated *step* method on a memory tree that is observationally equivalent to the state S, with inputs related to $\boldsymbol{xs}$ by the $\langle{=}\rangle$ relation and producing defined or undefined values that are directly mapped from $\boldsymbol{ys}$.

Lemma 4.4.20 (trconstr_cons_correct, src/StcToObc/Correctness.v:903)
*Given a well-clocked transition constraint tc that respects the normalization condition, Stc states S, I and S', an Stc environment R, a set of state variables $regs$, a clocking environment Ω, a memory tree me, an environment ve, a list of additional transition constraints **tcs** and a list of input names **ins** such that:*

1. $P, R, \mathsf{true}, S, I, S' \vdash tc$,
2. $R \overset{\mathsf{Free}(tc)}{\underset{regs}{\Longleftrightarrow}} (me, ve)$,
3. $(S, I, S') \overset{\boldsymbol{tcs}}{\Longleftrightarrow} me$,
4. *the transition constraints in* $tc \cdot \boldsymbol{tcs}$ *are well-scheduled*,
5. *the list of transition constraints* $tc \cdot \boldsymbol{tcs}$ *is reset-consistent (see definition 3.2.1 on page 78)*,
6. *standard variables defined by* $tc \cdot \boldsymbol{tcs}$ *and* **ins** *are mutually distinct*,
7. *standard and state variables defined by* $tc \cdot \boldsymbol{tcs}$ *are all distinct*,
8. *any identifier x defined in ve is either in* **ins** *or defined by* **tcs**,
9. *for all reset transitions (`reset` f `<i> every -`) in* $tc \cdot \boldsymbol{tcs}$, *we have* $\mathsf{s\text{-}closed}\ P\ f\ S[i]$, *and*
10. *for all reset transitions (`reset` f `<i> every -`) in* $tc \cdot \boldsymbol{tcs}$, *we have* $\mathsf{s\text{-}closed}\ P\ f\ I[i]$,

then (some hypotheses being omitted, notably clocking constraints, for the sake of brevity) there exists a memory tree me' and an environment ve' such that:

1. *(semantics preservation)* $\mathsf{s\text{-}tr}\ P, me, ve \vdash \mathsf{s\text{-}tr\text{-}tc}_{\Omega}^{regs}\ tc \Downarrow (me', ve')$,
2. *(environment correspondence)* $\forall x \in \mathsf{Var}\,(tc)\,,\ ve'(\!(x)\!) = R(x)^{\sqcup}$, *and*
3. *(memory correspondence)* $(S, I, S') \overset{tc \cdot \boldsymbol{tcs}}{\Longleftrightarrow} me'$.

A clarification: why is the second conclusion not $R \overset{\mathsf{Free}(tc)}{\underset{regs}{\Longleftrightarrow}} (me', ve')$? That is, why is the correspondence predicate not preserved for the new pair (me', ve')? The answer lies in the differences in the semantics of Stc and Obc for state variable definition / assignment. Consider the transition constraint `next x = x + 1` and its translation into the Obc statement `state(x) := state(x) + 1`. The Stc semantics (see figure 3.6a on page 83) constrains R to associate x to the *previous* value held for x in S. The correspondence predicate before executing the Obc statement guarantees the consistency for the free variables, in this case, `x`: its value is in me (since it is a state variable). Now, at the end of the execution, the memory tree is updated with the next value, giving $me' = me(x \mapsto me(x) + 1)$. Obviously me' is no longer in correspondence with R

for x. Thus we propagate a correspondence relation that restricts the correspondence to standard variables that are defined by *tc*. Indeed, in Obc, state variables are not bound in the environment *ve* anyway, and what we want to ensure after the generated statement executes, is that the variable assignment is consistent with the equational definition of the source transition constraint. State variable correspondence is expressed by the third conclusion.

Now we want to generalize the result to a list of transition constraints by induction. Unfortunately one hypothesis, needed to discharge the environment equivalence hypothesis of lemma 4.4.20 when we apply it in the induction step, is too weak for the induction. This hypothesis states that the set of state variables *regs* is included in the set of state variables defined by a **`next`** transition constraint in the list of transition constraints. We apply a simple trick to solve this technicality, and state a stronger invariant which is inductive. The idea is to cut the list of transition constraints into the appending $\boldsymbol{tcs} = \boldsymbol{tcs}_1 + \boldsymbol{tcs}_2$ and to perform the induction on $\boldsymbol{tcs}_2$.

Lemma 4.4.21 (trconstrs_app_correct, src/StcToObc/Correctness.v:1204)
Given lists of transition constraints $\boldsymbol{tcs}_1$ *and* $\boldsymbol{tcs}_2$ *that respect the normalization condition, Stc states* S, I *and* S', *an Stc environment* R, *a set of state variables regs, a clocking environment* Ω, *a memory tree* $me \approx\!\!\approx S$, *an environment ve, and a list of input names* ***ins*** *such that:*

1. $\forall tc \in (\boldsymbol{tcs}_1 + \boldsymbol{tcs}_2)\,,\ P, R, \mathsf{true}, S, I, S' \vdash tc$,
2. *each identifier in regs is defined by a* **`next`** *transition constraint in* $\boldsymbol{tcs}_1 + \boldsymbol{tcs}_2$,
3. *no identifier in* ***ins*** *is defined in* $\boldsymbol{tcs}_1 + \boldsymbol{tcs}_2$, ***ins*** *is exactly the domain of ve and present values in R for identifiers in* ***ins*** *coincide with those in ve, and*
4. *straightforward extensions of hypotheses 4 to 7, 9 and 10 from lemma 4.4.20 to the list* $\boldsymbol{tcs}_1 + \boldsymbol{tcs}_2$,

then there exists a memory tree me' *and an environment* ve' *such that:*

1. $\mathsf{s\text{-}tr}\ P, me, ve \vdash \mathsf{fold\text{-}l}\ (\lambda\ st\ tc.\ \mathsf{s\text{-}tr\text{-}tc}^{regs}_{\Omega}\ tc\ ;\ st)\ \ \boldsymbol{tcs}_2\ \ skip \Downarrow (me', ve')$,
2. $\forall x \in \mathsf{Var}\,(\boldsymbol{tcs}_2)\,,\ ve'(\!(x)\!) = R(x)^{\sqcup}$, *and*
3. $(S, I, S') \overset{\boldsymbol{tcs}_2}{\Longleftrightarrow} me'$.

We deduce the correctness result for a list of transition constraints directly by instantiating lemma 4.4.21 with $\boldsymbol{tcs}_1 = \varepsilon$.

Corollary 4.4.21.1 (trconstrs_correct, src/StcToObc/Correctness.v:1295)
Given a list of well-clocked transition constraints ***tcs*** *that respects the normalization condition, Stc states* S, I *and* S', *an Stc environment* R, *a set of state variables regs, a clocking environment* Ω, *a memory tree* $me \approx\!\!\approx S$, *an environment ve, and a list of input names* ***ins*** *such that* $\forall tc \in \boldsymbol{tcs},\ P, R, \mathsf{true}, S, I, S' \vdash tc$, *then (again, omitting some common hypotheses for clarity) there exists a memory tree* me' *and an environment* ve' *such that:*

1. $\mathsf{s\text{-}tr}\, P, me, ve \vdash \mathsf{fold\text{-}l}\,(\lambda\, st\ tc.\ \mathsf{s\text{-}tr\text{-}tc}_{\Omega}^{regs}\, tc\ ;\ st)\ \ \boldsymbol{tcs}\ \mathtt{skip} \Downarrow (me', ve')$,
2. $\forall x \in \mathsf{Var}\,(\boldsymbol{tcs}),\ \ ve'((x)) = R(x)^{\lfloor\rfloor}$, *and*
3. $(S, I, S') \overset{\boldsymbol{tcs}}{\Longleftrightarrow} me'$.

4.4.4.4 Systems

We extend the semantics preservation result to systems by induction on the program P, fixed from the induction hypothesis 4.4.2 on page 125 through corollary 4.4.21.1. In order to satisfy the hypothesis in the inductive case, we need a way to pass from the memory correspondence tailored for lists of transition constraints to the plain observational equivalence between memory trees. We can show that under some structural hypotheses, we can deduce the latter from the former.

Lemma 4.4.22 (Memory_Corres_equal_memory, src/StcToObc/Correctness.v:1340)
Given a program P; a list of transition constraints $\boldsymbol{tcs}$; states S, I and S'; a memory tree me; a list of state variables $\boldsymbol{inits}$ and a list of sub-systems declarations $\boldsymbol{subs}$ corresponding, respectively, to the state variables and instances appearing in $\boldsymbol{tcs}$; assuming that $(S, I, S') \overset{\boldsymbol{tcs}}{\Longleftrightarrow} me$, S and S' are closed relative to $\boldsymbol{inits}$ and $\boldsymbol{subs}$, and no reset transition appears in $\boldsymbol{tcs}$ without an associated default transition, then $me \approx S'$.

Now we can prove the correctness of translation of an Stc system.

Theorem 4.4.23 (correctness, src/StcToObc/Correctness.v:1427)
Given a well-clocked and well-scheduled program P that respects the normalization condition, a name f, a list of values $\boldsymbol{xs}$ that are not all absent, a list of values $\boldsymbol{ys}$, states S and S' such that $P, S, S' \vdash f\,(\boldsymbol{xs}) \downdownarrows \boldsymbol{ys}$, a list of defined or undefined values $\boldsymbol{xs'}$ such that $\boldsymbol{xs} \langle = \rangle \boldsymbol{xs'}$ and a memory tree $me \approx S$, then there exists a memory tree me' such that

$$\mathsf{s\text{-}tr}\, P, me \vdash f.\mathit{step}\,(\boldsymbol{xs'}) \overset{\boldsymbol{ys}^{\lfloor\rfloor}}{\Downarrow} me' \quad \text{and} \quad me' \approx S'$$

We can then deduce the correctness of the endless execution of the translated *step* method call after a call to the *reset* method.

Corollary 4.4.23.1 (correctness_loop_call, src/StcToObc/Correctness.v:1564)
Given a well-clocked and well-scheduled program P that respects the normalization condition, a name f, a stream of lists of values xs that at each instant are not all absent, a stream of lists of values ys, a state S such that $\mathsf{initial\text{-}state}\, P\ f\ S$ and $P, S \vdash f\,(xs) \overset{0}{\circlearrowright} ys$, a stream of lists of defined or undefined values xs' such that at each instant n, $xs_n \langle = \rangle xs'_n$, then there exists a memory tree me such that:

1. $me \approx S$
2. $\mathsf{s\text{-}tr}\, P, \{\varnothing\} \vdash f.\mathit{reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me$
3. $\mathsf{s\text{-}tr}\, P, me \vdash f.\mathit{step}\,(xs') \overset{0}{\circlearrowright} \left(\lambda\, n.\ ys_n^{\lfloor\rfloor}\right)$

4.5 Obc fusion optimization

The translation of Stc to Obc generates code with needlessly many conditional statements. The fusion optimization is a pass that is used to fuse adjacent conditionals that have the same guard expression. It is effective because the scheduling pass is designed to group transition constraints with identical or related clocks and guards from which the conditionals are generated. Incorporating this optimization directly into the translation pass would complicate both compilation and proof, so it is a separate source-to-source transformation, as in [Biernacki et al. (2008)].

For example, the result of fusion on the running example, recalled in listing 4.6a, is shown in listing 4.6b. Note that the two successive conditionals in the *step* method of the `ins` class have been fused into one, and similarly for the four successive conditionals in the *step* method of the `nav` class. Remark that we do not optimize useless copies in Vélus. For instance, we could eliminate the `cm` variable from the *step* method of `nav`. This optimization is common in Lustre compilers and will be considered in future work. In this particular case, though, the register allocation from CompCert does further optimize this code.

4.5.1 The optimization function

The optimization can be described in terms of two functions.

Definition 4.5.1 (fuse, src/Obc/Fusion.v:84)

$$\begin{aligned}
\textsf{fuse-stmt}\ (s_1\ ;\ s_2) &\triangleq \textsf{zip}\ s_1\ s_2\\
\textsf{fuse-stmt}\ s &\triangleq s\\
\textsf{zip}\ (\texttt{if}\ e\ \{\ t_1\ \}\ \texttt{else}\ \{\ f_1\ \})\ (\texttt{if}\ e\ \{\ t_2\ \}\ \texttt{else}\ \{\ f_2\ \}) &\triangleq \texttt{if}\ e\ \{\ \textsf{zip}\ t_1\ t_2\ \}\\
&\qquad\qquad \texttt{else}\ \{\ \textsf{zip}\ f_1\ f_2\ \}\\
\textsf{zip}\ (s_1\ ;\ s_2)\ t &\triangleq s_1\ ;\ \textsf{zip}\ s_2\ t\\
\textsf{zip}\ s\ (t_1\ ;\ t_2) &\triangleq \textsf{zip}\ (\textsf{zip}\ s\ t_1)\ t_2\\
\textsf{zip}\ \texttt{skip}\ t &\triangleq t\\
\textsf{zip}\ s\ \texttt{skip} &\triangleq s\\
\textsf{zip}\ s\ t &\triangleq s\ ;\ t
\end{aligned}$$

The fuse-stmt function operates on a composition of two statements by calling zip to fuse them. The zip function implements the actual optimization, by trying to fuse its two arguments. When it encounters two conditionals on the same guard expression, it fuses them into a single conditional and recursively optimizes in the branches. For sequential compositions, it simply acts recursively. There are two further remarks to make.

1. The second rule of zip seems to lack a recursive call, one would expect

$$\textsf{zip}\ (s_1\ ;\ s_2)\ t \triangleq \textsf{zip}\ s_1\ (\textsf{zip}\ s_2\ t)$$

```
class euler {
  state i: bool;
  state px: float64;

  step(x0: float64, u: float64) returns (x: float64) {
    if state(i) { x := x0 } else { x := state(px) };
    state(i) := false;
    state(px) := x + 0.1 * u
  }

  reset() { state(i) := true; state(px) := 0. }
}

class ins {
  instance xe: euler;
  state k: int32;
  state px: float64;

  step(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64
  {
    alarm := state(k) >= 50;
    state(k) := state(k) + 1;
    if alarm { } else { xe := euler(xe).step([gps], [xv]) };
    if alarm { x := state(px) } else { x := xe };
    state(px) := x
  }

  reset() { state(k) := 0; state(px) := 0.; euler(xe).reset() }
}

class nav {
  instance insr: ins;
  state c: bool;
  state r: bool;

  step(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64, alr: bool
  {
    if state(r) { ins(insr).reset() } else { };
    state(r) := s and state(c);
    if state(c) { } else { insr, alr := ins(insr).step([gps], [xv]) };
    if state(c) { cm := not s } else { cm := s };
    if state(c) { x := gps } else { x := insr };
    if state(c) { alarm := false } else { alarm := alr };
    state(c) := cm
  }

  reset() { state(c) := true; state(r) := false; ins(insr).reset() }
}
```

Obc

(a) Before

Listing 4.6 (I): Fusion of the example

```
class euler {
  state i: bool;
  state px: float64;

  step(x0: float64, u: float64) returns (x: float64) {
    if state(i) { x := x0 } else { x := state(px) };
    state(i) := false;
    state(px) := x + 0.1 * u
  }

  reset() { state(i) := true; state(px) := 0. }
}

class ins {
  instance xe: euler;
  state k: int32;
  state px: float64;

  step(gps: float64, xv: float64) returns (x: float64, alarm: bool)
    var xe: float64
  {
    alarm := state(k) >= 50;
    state(k) := state(k) + 1;
    if alarm { x := state(px) } else {
      xe := euler(xe).step([gps], [xv]);
      x := xe
    };
    state(px) := x
  }

  reset() { state(k) := 0; state(px) := 0.; euler(xe).reset() }
}

class nav {
  instance insr: ins;
  state c: bool;
  state r: bool;

  step(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
    var cm: bool, insr: float64, alr: bool
  {
    if state(r) { ins(insr).reset() } else { };
    state(r) := s and state(c);
    if state(c) {
      cm := not s;
      x := gps;
      alarm := false
    } else {
      insr, alr := ins(insr).step([gps], [xv]);
      cm := s;
      x := insr;
      alarm := alr
    };
    state(c) := cm
  }

  reset() { state(c) := true; state(r) := false; ins(insr).reset() }
}
```

Obc

(b) After

Listing 4.6 (II): Fusion of the example

Indeed, with the actual definition we have:

```
(if e { a } else { b };
 if e { c } else { d });
if e { f } else { g }
```

Obc

fuse-stmt →

```
if e { a } else { b };
if e { zip c f } else { zip d g }
```

Obc

The first conditional is not fused with the subsequent ones. But, in fact, we do not generate code with this form: statement composition always occurs to the right. We take advantage of this particularity to save an extra recursive call and the associated proof effort.

2. The third rule makes two recursive calls, as expected, but the outer one is not on a syntactically smaller first argument, as are the other recursive calls. Coq rejects such definitions because they are not supported by the syntactic criterion used to check well foundedness. The classical *nested fixpoints* approach does not apply either, so we choose to implement zip as two distinct fixpoints.

We will write fuse p to designate the program obtained from p by applying fuse-stmt in all methods of all of its classes.

Parts of the example (see the NLustre original program in listing 2.5b on page 43) can be re-used to highlight the need for independent scheduling of the modular reset, mentioned in section 3.2. Consider the following `driver` node that simply instantiate twice the `ins` node with reset.

```
node driver(gps, xv, yv: float64; r: bool) returns (x, y: float64);
  var alarmx, alarmy : bool;
let
  x, alarmx = (restart ins every r)(gps, xv);
  y, alarmy = (restart ins every r)(gps, yv);
tel
```

Lustre

Without the introduction of Stc and its independent reset construct, we would generate the following Obc code, translating an NLustre node instantiation with reset directly into a sequence of a guarded *reset* method call and a *step* method call.

```
step(gps: float64, xv: float64, yv: float64, r: bool) returns (x: float64, y: float64)
  var alarmx: bool, alarmy: bool
{
  if r { ins(x).reset(); } else { };
  x, alarmx := ins(x).step([gps], [xv]);
  if r { ins(y).reset(); } else { };
  y, alarmy := ins(y).step([gps], [yv]);
}
```

Obc

Since the fusion optimization only fuses adjacent conditionals and does not reorder statements, it will not change this program. We can see, however, that reordering is possible without changing the overall effect of the method. The fact that the translation from NLustre to Stc introduces distinct transition constraints for resetting and stepping allows the subsequent scheduling pass to group the reset transition constraints together, as in the following Stc default transition.

```
transition(gps: float64, xv: float64, yv: float64, r: bool) returns (x: float64, y: float64)
  var alarmx: bool, alarmy: bool;
{
  reset(ins<x>) every (. on r)
  reset(ins<y>) every (. on r)
  (y, alarmy) = ins<y>(gps, dy)
  (x, alarmx) = ins<x>(gps, xv)
}
```

Stc

This transition is then translated into the following Obc step method.

```
step(gps: float64, xv: float64, yv: float64, r: bool) returns (x: float64, y: float64)
  var alarmx: bool, alarmy: bool
{
  if r { ins(x).reset() } else { };
  if r { ins(y).reset() } else { };
  y, alarmy := ins(y).step([gps], [yv]);
  x, alarmx := ins(x).step([gps], [xv])
}
```

Obc

This code will be optimized:

```
step(gps: float64, xv: float64, yv: float64, r: bool) returns (x: float64, y: float64)
  var alarmx: bool, alarmy: bool
{
  if r {
    ins(x).reset();
    ins(y).reset()
  } else { };
  y, alarmy := ins(y).step([gps], [yv]);
  x, alarmx := ins(x).step([gps], [xv])
}
```

Obc

4.5.2 Correctness of the optimization

In general the fusion optimization does not preserve the semantics of an arbitrary program, as shown on the example below:

```
if x { x := false } else { x := true };
if x { s1 } else { s2 }
```

Obc

fuse-stmt →

```
if x { x:= false; s1 }
else { x := true; s2 }
```

Obc

$$\frac{}{\mathsf{Fusible\text{-}stmt}\ (x := e)} \qquad \frac{}{\mathsf{Fusible\text{-}stmt}\ (\texttt{state}(x) := e)} \qquad \frac{}{\mathsf{Fusible\text{-}stmt}\ (\boldsymbol{x} := i^c.\, f\,(\boldsymbol{e}))}$$

$$\frac{}{\mathsf{Fusible\text{-}stmt}\ \texttt{skip}} \qquad \frac{\mathsf{Fusible\text{-}stmt}\ s_1 \quad \mathsf{Fusible\text{-}stmt}\ s_2}{\mathsf{Fusible\text{-}stmt}\ (s_1\ ;\ s_2)}$$

$$\frac{\begin{array}{c}\mathsf{Fusible\text{-}stmt}\ s_1 \quad \mathsf{Fusible\text{-}stmt}\ s_2 \\ \forall x \in \mathsf{Free}\ (e)\,,\ \neg\, \mathsf{MayWrite}\ s_1\ x\ \wedge\ \neg\, \mathsf{MayWrite}\ s_2\ x\end{array}}{\mathsf{Fusible\text{-}stmt}\ (\texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \})}$$

Figure 4.6: The "fusible" predicate on Obc statements (Fusible, src/Obc/Fusion.v:250)

$$\frac{}{\mathsf{MayWrite}\ (x := e)\ \ x} \qquad \frac{}{\mathsf{MayWrite}\ (\texttt{state}(x) := e)\ \ x} \qquad \frac{x \in \boldsymbol{x}}{\mathsf{MayWrite}\ (\boldsymbol{x} := i^c.\, f\,(\boldsymbol{e}))\ \ x}$$

$$\frac{\mathsf{MayWrite}\ s_1\ x\ \vee\ \mathsf{MayWrite}\ s_2\ x}{\mathsf{MayWrite}\ (s_1\ ;\ s_2)\ \ x} \qquad \frac{\mathsf{MayWrite}\ s_1\ x\ \vee\ \mathsf{MayWrite}\ s_2\ x}{\mathsf{MayWrite}\ (\texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \})\ \ x}$$

Figure 4.7: The "may write" predicate on Obc statements (Can_write_in, src/Obc/ObcInvariants.v:33)

In this section we show that it does for the code generated from Stc.

We define a Fusible-stmt predicate, presented in figure 4.6, that is, a sufficient condition for an Obc statement for its semantics to be preserved by the fusion optimization. The interesting case is that for a conditional: a conditional is eligible for fusion if its branches are and if every free variable of the guard is never written in the branches.

To express the fact that a variable may—we say *may* because of conditionals—be written in a statement, we introduce the MayWrite predicate presented in figure 4.7. The variable x may be written by an assignment to x (regular assignment, state variable assignment or method call assignment). It may be written in a sequence of two statements if it may be written in one of them. Finally, it may be written in a conditional if it may be written by one of its branches.

We write Fusible p to express the fact that the whole program p is eligible for fusion, that is, that the body of each method of each class satisfies Fusible-stmt.

We start by showing that zip preserves the Fusible-stmt predicate.

Lemma 4.5.1 (zip_free_write, src/Obc/Fusion.v:373)
Given statements s_1 and s_2 such that Fusible-stmt s_1 *and* Fusible-stmt s_2, *then* Fusible-stmt (zip s_1 s_2)

Then we can show that zip preserves the semantics relative to the sequential composition.

Lemma 4.5.2 (fuse'_Comp, src/Obc/Fusion.v:438)
Given statements s_1 and s_2 such that Fusible-stmt s_1 *and* Fusible-stmt s_2, *then for any program p, memory tree me and environment ve, $p, me, ve \vdash$ zip s_1 $s_2 \Downarrow (me, ve)$ if and only if $p, me, ve \vdash s_1 \ ; \ s_2 \Downarrow (me, ve)$.*

It follows directly that the fusion optimization on a statement preserves the semantics provided that the statement is eligible for fusion.

Corollary 4.5.2.1 (fuse_Comp, src/Obc/Fusion.v:457)
Given a statement s such that Fusible-stmt s, *then for any program p, memory tree me and environment ve, $p, me, ve \vdash$ fuse-stmt $s \Downarrow (me, ve)$ if and only if $p, me, ve \vdash s \Downarrow (me, ve)$.*

We can now show that the optimization on a program preserves the semantics of a method call.

Lemma 4.5.3 (fuse_call, src/Obc/Fusion.v:468)
Given a program p such that Fusible p, *a class name c, a method name f, memory trees me and me' and lists of defined or undefined values $\boldsymbol{v}$ and $\boldsymbol{w}$ such that $p, me \vdash c.f\ (\boldsymbol{v}) \overset{\boldsymbol{w}}{\Downarrow} me'$ then* fuse $p, me \vdash c.f\ (\boldsymbol{v}) \overset{\boldsymbol{w}}{\Downarrow} me'$.

We extend this result to the looping execution of a method call.

Corollary 4.5.3.1 (fuse_loop_call, src/Obc/Fusion.v:505)
Given a program p such that Fusible p, *a class name c, a method name f, a memory tree me, streams of lists of defined or undefined values xs and ys and an integer n such that $p, me \vdash c.f\ (xs) \overset{n}{\circledcirc} ys$ then* fuse $p, me \vdash c.f\ (xs) \overset{n}{\circledcirc} ys$.

4.5.3 Eligibility of generated Obc code for fusion optimization

To show that the fusion optimization preserves the semantics of the generated code, it suffices to prove that this code is eligible for fusion. During the generation of Obc code, statements are produced from both control expressions and transition constraints.

The translation of control expressions is parameterized by the variable to be assigned: we show that the generated statement is fusible if this variable is not free in the expression.

Lemma 4.5.4 (Fusible_translate_cexp, src/StcToObc/Stc2ObcInvariants.v:69)
Given a control expression e, a variable $x \notin \mathsf{Free}\,(e)$ and a set of state variables regs, then $\mathsf{Fusible\text{-}stmt}\,(\mathsf{s\text{-}tr\text{-}cexp}^{regs}(x, e))$.

For transition constraints, guards are recursively added by the ctrl function, following the structure of the clock annotation of the transition constraint. We show that this function preserves the Fusible-stmt predicate provided that no free variable of the clock is written in the enclosed statement.

Lemma 4.5.5 (Fusible_Control, src/StcToObc/Stc2ObcInvariants.v:92)
Given a statement s, a clock ck and a set of state variables regs such that $\mathsf{Fusible\text{-}stmt}\,s$ *and for all variables* $x \in \mathsf{Free}\,(ck)$, $\neg\,\mathsf{MayWrite}\,s\,x$, *then* $\mathsf{Fusible\text{-}stmt}\,(\mathsf{ctrl}^{regs}\,ck\,s)$.

We state that the translation of a list of transition constraints produces a fusible statement, provided that it is well-scheduled and well-clocked. We do not present this intermediate result, but note that it is proved by induction on the list of transition constraints, using lemmas 4.5.4 and 4.5.5. Eventually we prove that the translation of an Stc program generates an Obc program that is eligible for the fusion optimization.

Theorem 4.5.6 (ClassFusible_translate, src/StcToObc/Stc2ObcInvariants.v:204)
Given a well-clocked and well-scheduled program P, we have $\mathsf{Fusible}\,(\mathsf{s\text{-}tr}\,P)$.

Chapter 5

Generation of Clight code

Contents

Even though Obc is much more operational than the dataflow semantics of Lustre, it is still quite far from the level of detail needed to execute on a real machine. In particular, Obc does not treat data representation, stacking needed for function calls, or generation of assembly instructions for a given architecture. All these issues are not specific to the compilation of Lustre, and they are already addressed by CompCert, that we interface with: (1) we generate Clight abstract code and let the algorithms of CompCert do the remaining of the compilation to assembly code, and (2) we chain the proofs of Vélus and CompCert together to get an end-to-end correctness proof.

In this chapter we present the generation of Clight code from Obc code. We begin by describing the argument initialization transformation, which is necessary prior to generating Clight code. Then, we present the Clight language and the generation function. In order to present the proof of correctness, we describe a subset of the formal semantic model of Clight. We present in the fourth section, how we use separation logic (SL) in the proof of correctness, described in the last section. The main challenge is to establish a correspondence between the tree-shaped memory model of Obc and the intricate machine-level memory model of CompCert.

$$\frac{}{\mathsf{NoNakedVars}\,(x := e)} \qquad \frac{}{\mathsf{NoNakedVars}\,(\texttt{state}(x) := e)} \qquad \frac{}{\mathsf{NoNakedVars}\,\texttt{skip}}$$

$$\frac{\forall e \in \boldsymbol{e},\ e \text{ is not a variable}}{\mathsf{NoNakedVars}\,(\boldsymbol{y} := i^c.\,f\,(\boldsymbol{e}))} \qquad \frac{\mathsf{NoNakedVars}\,s_1 \quad \mathsf{NoNakedVars}\,s_2}{\mathsf{NoNakedVars}\,(s_1\ ;\ s_2)}$$

$$\frac{\mathsf{NoNakedVars}\,s_1 \quad \mathsf{NoNakedVars}\,s_2}{\mathsf{NoNakedVars}\,(\texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \})}$$

Figure 5.1: The NoNakedVars predicate (No_Naked_Vars, src/Obc/ObcInvariants.v:175)

5.1 Obc argument initialization

We have seen the introduction of validity assertions in Obc (section 4.1.2). These assertions are used to denote that the value of a variable is guaranteed to be defined at a given point in a program. In Obc, a method may be invoked even if some arguments (variables only) are undefined, but method calls will be translated into Clight function calls in which, as we will see, all arguments must be defined.

The Obc code generated from Stc already contains validity assertions for argument variables known to be defined at call time (see the s-tr-arg function, section 4.3 on page 112). Now the idea is to wrap all other variable arguments with validity assertions, which in turn have to be justified by initializing those variables before the call.

Bourke and Pouzet (2019) present the corresponding compilation pass and its proof of correctness in detail, so I will only present the key aspects here. A recursive function is introduced to add validity assertions and initialization assignments. This function implements a compromise between adding useless initialization assignments and the minimal number of assignments to perform.

We prove that the code produced by the argument initialization function satisfies the predicate NoNakedVars shown in figure 5.1. This predicate simply expresses that no variable appears as an argument of a method call. It is generalized to methods, classes and programs: we write init-args p to designate the program obtained from p by applying the argument initialization function to the bodies of the methods of all of its classes. The NoNakedVars predicate is used in the correctness proof of the Clight generation function to show that the arguments of method calls are always defined.

We must also show that the argument initialization function preserves the semantics. Unlike earlier source-to-source transformations, namely, the scheduling of Stc transition constraints and fusion optimization, the values of variables in the transformed program may differ from those in the original one. That is, at a given point in the program, the environment can be *more defined* than the environment of the source program. Thus, Bourke and Pouzet (2019) introduce a *refinement* relation between environments.

$$\frac{}{\mathsf{NoOverwrites}\,(x := e)} \qquad \frac{}{\mathsf{NoOverwrites}\,(\texttt{state}(x) := e)} \qquad \frac{}{\mathsf{NoOverwrites}\,\texttt{skip}}$$

$$\frac{}{\mathsf{NoOverwrites}\,(\boldsymbol{x} := i^c\,.\,f\,(\boldsymbol{e}))} \qquad \frac{\mathsf{NoOverwrites}\,s_1 \quad \mathsf{NoOverwrites}\,s_2}{\mathsf{NoOverwrites}\,(\texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \})}$$

$$\frac{\mathsf{NoOverwrites}\,s_1 \quad \mathsf{NoOverwrites}\,s_2 \qquad \forall x,\ \mathsf{MayWrite}\,s_1\ x\ \rightarrow\ \neg\,\mathsf{MayWrite}\,s_2\ x \quad \forall x,\ \mathsf{MayWrite}\,s_2\ x\ \rightarrow\ \neg\,\mathsf{MayWrite}\,s_1\ x}{\mathsf{NoOverwrites}\,(s_1\ ;\ s_2)}$$

Figure 5.2: The NoOverwrites predicate (No_Overwrites, src/Obc/ObcInvariants.v:138)

Definition 5.1.1 (refines, src/Environment.v:907)

$$ve_2 \sqsupseteq ve_1 \triangleq \forall x\,v,\ ve_1(x) = v\ \rightarrow\ ve_2(x) = v$$

We say that ve_2 refines ve_1 and write $ve_2 \sqsupseteq ve_1$ if any variable defined in ve_1 is also defined in ve_2 with the same value, leaving the possibility for variables not defined in ve_1 to be defined in ve_2 with any values. The notation differs slightly from that used in [Bourke and Pouzet (2019)], $\sqsubseteq$, because this way operands in $ve_2 \sqsupseteq ve_1$ are in the same order than in the sentence "ve_2 refines ve_1".

This relation is used to define a notion of semantic refinement for statements. Intuitively we want to express that the statement t resulting from the argument initialization transformation applied to a statement s produces an environment that refines the environment produced by s: additional variable assignments performed by t do not matter as long as all variables defined by s are defined by t with the same values.

Definition 5.1.2 (stmt_refines, src/Obc/Equiv.v:224)

$$\begin{aligned} s_2 \sqsupseteq_P^{p_1,p_2} s_1 \triangleq \forall &me\ me'\ ve_1\ ve_2\ ve_1',\\ &P\ ve_1\ ve_2\ \rightarrow\\ &ve_2 \sqsupseteq ve_1\ \rightarrow\\ &p_1, me, ve_1 \vdash s_1 \Downarrow (me', ve_1')\ \rightarrow\\ &\exists ve_2' \sqsupseteq ve_1',\ p_2, me, ve_2 \vdash s_2 \Downarrow (me', ve_2') \end{aligned}$$

This definition uses a precondition P, needed for the proof of semantics preservation.

This semantic refinement relation is extended to methods, classes and programs, and Bourke and Pouzet (2019) eventually show that the arguments initialization function init-args preserves the semantics relative to program refinement, under a well-chosen precondition and three additional hypotheses: (1) well-typing, (2) the methods do not assign to their inputs, and (3) the program is in a kind of static single assignment form (SSA). The third hypothesis is expressed using a NoOverwrites predicate, shown in

figure 5.2. This predicate ensures that, in a sequential composition s_1 ; s_2, if a variable may be written in s_1, it is never written in s_2, and likewise for s_2 and s_1.

It can be shown that program refinement implies method call semantics preservation in the strict sense, expressing that a method call with refined inputs (*more defined*) in a refined program produces refined outputs. We specialize this result to the case of generated Obc code where both the inputs and outputs of a method call are defined.

Theorem 5.1.1 (stmt_call_eval_add_defaults, src/Obc/ObcAddDefaults.v:1994)
Given a well-typed SSA program p whose method inputs are never assigned, a class name c, a method name f, memory trees me and me' and lists of defined values written $\lfloor \boldsymbol{v} \rfloor$ and $\lfloor \boldsymbol{w} \rfloor$ such that $p, me \vdash c.f\,(\lfloor \boldsymbol{v} \rfloor) \overset{\lfloor \boldsymbol{w} \rfloor}{\Downarrow} me'$, then $\mathsf{init\text{-}args}\, p, me \vdash c.f\,(\lfloor \boldsymbol{v} \rfloor) \overset{\lfloor \boldsymbol{w} \rfloor}{\Downarrow} me'$.

The result extends to the looping execution of a method.

Corollary 5.1.1.1 (loop_call_add_defaults, src/Obc/ObcAddDefaults.v:2016)
Given a well-typed SSA program p whose method inputs are never assigned, a class name c, a method name f, a memory tree me, streams of lists of defined values written $\lfloor xs \rfloor$ and $\lfloor ys \rfloor$ and an integer n such that $p, me \vdash c.f\,(\lfloor xs \rfloor) \overset{n}{@} \lfloor ys \rfloor$, then $\mathsf{init\text{-}args}\, p, me \vdash c.f\,(\lfloor xs \rfloor) \overset{n}{@} \lfloor ys \rfloor$.

5.2 From Obc to Clight

5.2.1 Clight overview

Clight [Blazy and Leroy (2009)] is "a simplified version of CompCert C where all expressions are pure and assignments and function calls are statements, not expressions" [Leroy (2019)]. We use it as our target language for the following reasons:

1. Its memory model and its semantics are precise, low-level and close enough to the machine to reason about executable code.

2. It is very close to C [Kernighan and Ritchie (1988)], to which compilation from Lustre is well-known.

3. It is part of the frontend of CompCert: as such, its semantics is fully specified in Coq, and we can rely on both the compilation algorithms of CompCert and their proof of correctness for the remaining of the compilation chain.

The abstract syntax of the subset of Clight used for code generation is presented in figure 5.3. The class of expressions[1] comprises four forms of constants, local variables, temporary variables, unary and binary operations, pointer dereference, adress-of operation, and field access. Temporary variables are a special class of local variables which do not reside in memory—i.e., registers. They are declared in the concrete syntax with the

[1] compcert.inria.fr/doc/html/compcert.cfrontend.Clight.html#expr

e	::=		**expression**
	|	$i \mid l \mid f \mid d$	(constants (int, long, float, double))
	|	x	(local variable)
	|	$\overline{x}$	(temporary variable)
	|	$\diamond^C\ a$	(unary operation)
	|	$a \oplus a$	(binary operation)
	|	$*a$	(pointer dereference)
	|	$\&a$	(address-of operator)
	|	$a\,.\,x$	(field access)
a	::=	e^τ	(typed expression)
	|	$\tau(a)$	(type cast)
s	::=		**statement**
	|	a = a	(assignment)
	|	$\overline{x}$ = a	(assignment to a temporary variable)
	|	`if (`a`) {`s`} else {`s`}`	(conditional)
	|	$[\overline{x}$ =$]\ a(a^*_,)$	(function call)
	|	s ; s	(sequence)
	|	`skip`	(do nothing)
	|	`return` $[a]$	(return statement)
	|	`loop {`s`} {`s`}`	(infinite loop)
	|	$\overline{x}$ = `vload(`κ`,`a`:`τ`)`	(volatile read)
	|	`vstore(`κ`,`a`:`τ`,`a`:`τ`)`	(volatile write)
d	::=	$\tau\ x$	**variable declaration**
ds	::=	`struct` x `{`$d^*_;$`}`	**structure declaration**
df	::=	$\tau\ x(d^*_,)$ `{(register` $d)^*_;$ $d^*_;$ s`}`	**function declaration**
p	::=	$([\texttt{volatile}]\ d)^*_;\ ds^*_;\ df^*$	**program**

Figure 5.3: Clight subset abstract syntax

register keyword, we distinguish them by adding an overbar. The class of annotated expressions comprises expressions annotated with their types and explicit type casts. In the following, we will write $ae\texttt{->}x$ as an abbreviation for $(*ae).x$.

For statements[2], Clight distinguishes two kinds of assignments with different semantics: (1) assignment of an r-value to an l-value, (2) assignment of an r-value to a temporary variable. In addition, the return value of a function call can only be assigned to a single temporary variable. Clight does not provide for-loops or while-loops, but only an infinite loop, `loop {s1} {s2}`, which executes `s1` then `s2` forever. In the full language, there is a **break** statement which can be used to interrupt infinite loops, but we do not need it, since the only loop we generate is the main one which is never interrupted. Finally, we use two compiler builtin operations: a load from a volatile variable and a store to a volatile variable. Both take a parameter κ, a *memory chunk*, that is, an annotation "indicating the type, size and signedness of the chunk of memory being accessed" [Leroy (2019)]. Both also have additional type annotations that indicate statically how the parameters must be cast before the call.

A Clight program is a list of declarations: global variables, possibly with **volatile** attribute (and initialization data, that we omit here), structures and functions. A program must have an entry point, usually named `main`. A function declaration consists of a return type, a name, formal parameters, local—temporary and classic—variables, and a body as a single statement.

5.2.2 Generation function

We present the translation function before presenting the semantics of the target language. This is for two reasons: first, the C language is sufficiently well-known for the syntax of Clight to be understood intuitively, and second, our presentation of the intricate and complex mechanized semantics of Clight will be restricted to the subset that we use.

Obc and Clight are both sequential imperative languages, so compilation can translate the former's control structures fairly directly. Some care is needed, however, to implement encapsulation and multiple return values of Obc. The state of a Lustre node instance is represented in Clight as a structure generated from the instantiated Obc class. The structure has the same name as the class, and fields for *register* and *instance* fields of the class. Obc methods are translated, in general, into void functions with two additional parameters:

1. A *self* pointer toward the structure representing the object on which the method is called. This is standard way of implementing objects in C.

2. An *out* pointer toward a special structure used as an output parameter to store the return values of the call. The structure has one field for each output parameter of the translated method.

As we will see in the next section, method input declarations are encoded as declarations of temporaries. The body of a method is translated into a Clight statement preceded by a

[2] compcert.inria.fr/doc/html/compcert.cfrontend.Clight.html#statement

list of temporary variables declarations corresponding to the local variables of the method, and followed by a **`return`** statement. As a specialization to generate more idiomatic code, the *out* pointer and the output structure type declaration are omitted if the method returns a single value or no value at all. In the first case, the corresponding function is not void but rather returns a value using a single additional temporary variable declaration.

Recall the Obc running example in listing 5.1a, we present its translation into Clight code in three steps. First, listing 5.1b shows the declarations of the various structures: `euler`, `ins` and `nav` are the structure types that represent the state of the corresponding Obc classes, while `fun$ins$step` and `fun$nav$step` are the structure types to hold the return values of the functions generated from the *step* methods of the classes `ins` and `nav`—such a structure is not needed for `euler` since its *step* method produces only one output. Listing 5.1c shows the translation of the methods of the classes. Note the temporary declaration at line 2 corresponding to the unique output of the generated function for the *step* method of the `euler` class. In contrast, at line 16, see how the `alarm` output of the *step* method of the `ins` class is assigned through the use of the *out* pointer. On lines 45 and 46, we can observe the special sequence of assignments that occurs after a call to a function with multiple return values.

A simplified abstract version of the generation algorithm—in particular, we omit some type annotation details—is presented in figure 5.4. The translation of expressions by the gen-exp function in figure 5.4a is rather direct. A variable is translated using gen-var. If it is a local variable or the only output variable, then it is translated into a temporary variable, otherwise, it is accessed through the output structure *out*. A state variable is translated into a field access on the *self* pointer representing the state. A type cast operation, considered as an unary operation in Vélus, is distinguished from regular Clight unary operations and translated into a Clight type cast towards the type annotation τ of the whole expression: well-typing will ensure that $\tau' = \tau$ anyway. Translation of other unary and binary operations is direct. Finally, validity assertions are dropped.

The translation of statements by gen-stmt in figure 5.4b is also rather direct as the control flow is preserved. As for expressions, the only complication comes from the handling of multiple return values. First, assign works like gen-var, but for generating assignments. Second, it complicates method call translation, by funcall. There are three cases, that we illustrate using the running example in listings 5.1a and 5.1c.

1. If the method has no output, the call is translated into a function call without return value. Nonetheless, this cannot happen in Obc code that we generate.

2. If the method has a unique output, we use an intermediate fresh temporary variable to store the result, and the call is followed by an assignment to the appropriate local variable or output field. For example, the call at line 25

   ```
   xe := euler(xe).step([gps], [xv]);
   ```

   is translated into the sequence at lines 19 and 20

   ```
   step$x = fun$euler$step(&self->xe, gps, xv);
   xe = step$x;
   ```

```
1  class euler {
2    state i: bool;
3    state px: float64;
4
5    step(x0: float64, u: float64) returns (x: float64) {
6      if state(i) { x := x0 } else { x := state(px) };
7      state(i) := false;
8      state(px) := x + 0.1 * u
9    }
10
11   reset() { state(i) := true; state(px) := 0. }
12 }
13
14 class ins {
15   instance xe: euler;
16   state k: int32;
17   state px: float64;
18
19   step(gps: float64, xv: float64) returns (x: float64, alarm: bool)
20     var xe: float64
21   {
22     alarm := state(k) >= 50;
23     state(k) := state(k) + 1;
24     if alarm { x := state(px) } else {
25       xe := euler(xe).step([gps], [xv]);
26       x := xe
27     };
28     state(px) := x
29   }
30
31   reset() { state(k) := 0; state(px) := 0.; euler(xe).reset() }
32 }
33
34 class nav {
35   instance insr: ins;
36   state c: bool;
37   state r: bool;
38
39   step(gps: float64, xv: float64, s: bool) returns (x: float64, alarm: bool)
40     var cm: bool, insr: float64, alr: bool
41   {
42     if state(r) { ins(insr).reset() } else { };
43     state(r) := s and state(c);
44     if state(c) {
45       cm := not s;
46       x := gps;
47       alarm := false
48     } else {
49       insr, alr := ins(insr).step([gps], [xv]);
50       cm := s;
51       x := insr;
52       alarm := alr
53     };
54     state(c) := cm
55   }
56
57   reset() { state(c) := true; state(r) := false; ins(insr).reset() }
58 }
```

Obc

(a) Before

Listing 5.1 (I): Translation of the example

```
struct euler {
  _Bool i;
  double px;
};

struct ins {
  int k;
  double px;
  struct euler xe;
};

struct nav {
  _Bool c;
  _Bool r;
  struct ins insr;
};

struct fun$ins$step {
  double x;
  _Bool alarm;
};

struct fun$nav$step {
  double x;
  _Bool alarm;
};
```

Clight

(b) After: structure declarations

Listing 5.1 (II): Translation of the example

```
double fun$euler$step(struct euler *self, double x0, double u) {
  register double x;
  if (self->i) { x = x0; } else { x = self->px + 0.1 * u; }
  self->i = 0;
  self->px = x;
  return x;
}
void fun$euler$reset(struct euler *self) {
  self->i = 1;
  self->px = 0;
  return;
}

void fun$ins$step(struct ins *self, struct fun$ins$step *out, double gps, double xv) {
  register double step$x; register double xe;
  out->alarm = self->k >= 50;
  self->k = self->k + 1;
  if (out->alarm) { out->x = self->px; } else {
    step$x = fun$euler$step(&self->xe, gps, xv);
    xe = step$x;
    out->x = xe;
  }
  self->px = out->x;
  return;
}
void fun$ins$reset(struct ins *self) {
  self->k = 0;
  self->px = 0;
  fun$euler$reset(&self->xe);
  return;
}

void fun$nav$step(struct nav *self, struct fun$nav$step *out, double gps, double xv, _Bool s) {
  struct fun$ins$step out$insr$step;
  register _Bool cm; register double insr; register _Bool alr;
  if (self->r) { fun$ins$reset(&self->insr); }
  self->r = s & self->c;
  if (self->c) {
    cm = !s;
    out->x = gps;
    out->alarm = 0;
  } else {
    fun$ins$step(&self->insr, &out$insr$step, gps, xv);
    insr = out$insr$step.x;
    alr = out$insr$step.alarm;
    cm = s;
    out->x = insr;
    out->alarm = alr;
  }
  self->c = cm;
  return;
}
void fun$nav$reset(struct nav *self) {
  self->c = 1;
  self->r = 0;
  fun$ins$reset(&self->insr);
  return;
}
```

Clight

(c) After: functions

Listing 5.1 (III): Translation of the example

$$\mathsf{gen\text{-}var}\ x^{\tau} \triangleq \begin{cases} \overline{x}^{\tau} & \text{if } \|\boldsymbol{outs}\| \leq 1 \vee x \notin \boldsymbol{outs} \\ (\overline{\mathsf{out}}\texttt{->}x)^{\tau} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathsf{gen\text{-}exp}\ c &\triangleq c^{\mathsf{type\text{-}const}\ c} \\
\mathsf{gen\text{-}exp}\ x^{\tau} &\triangleq \mathsf{gen\text{-}var}\ x^{\tau} \\
\mathsf{gen\text{-}exp}\ (\texttt{state}(x)^{\tau}) &\triangleq \left(\overline{\mathsf{self}}\texttt{->}x\right)^{\tau} \\
\mathsf{gen\text{-}exp}\ (\tau'(e))^{\tau} &\triangleq \tau(\ \mathsf{gen\text{-}exp}\ e) \\
\mathsf{gen\text{-}exp}\ \left(\diamond^{C}\ e\right)^{\tau} &\triangleq \left(\diamond^{C}\ \mathsf{gen\text{-}exp}\ e\right)^{\tau} \\
\mathsf{gen\text{-}exp}\ (e_1 \oplus e_2)^{\tau} &\triangleq (\mathsf{gen\text{-}exp}\ e_1 \oplus \mathsf{gen\text{-}exp}\ e_2)^{\tau} \\
\mathsf{gen\text{-}exp}\ [x]^{\tau} &\triangleq \mathsf{gen\text{-}var}\ x^{\tau}
\end{aligned}$$

(a) Translation of expressions (translate_exp, src/ObcToClight/Generation.v:88)

$$\mathsf{assign}(x, a) \triangleq \begin{cases} \overline{x} = a & \text{if } \|\boldsymbol{outs}\| \leq 1 \vee x \notin \boldsymbol{outs} \\ \overline{\mathsf{out}}\texttt{->}x = a & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathsf{funcall}(\varepsilon, c, i, f, \boldsymbol{a}) &\triangleq f_c(\&\overline{\mathsf{self}}\texttt{->}i, \boldsymbol{a}) \\
\mathsf{funcall}([x], c, i, f, \boldsymbol{a}) &\triangleq \overline{x'} = f_c(\&\overline{\mathsf{self}}\texttt{->}i, \boldsymbol{a})\ ; \\
&\qquad \mathsf{assign}\left(x, \overline{x'}\right) \\
\mathsf{funcall}(\boldsymbol{x}, c, i, f, \boldsymbol{a}) &\triangleq f_c(\&\overline{\mathsf{self}}\texttt{->}i, \&i_f, \boldsymbol{a})\ ; \qquad \mathsf{class}(p, c) \doteq (cls, p') \\
&\qquad \mathsf{assign}(\boldsymbol{x}, i_f\,.\,\boldsymbol{y}) \qquad \text{where } \mathsf{method}(cls, f) \doteq m \\
&\qquad\qquad\qquad\qquad\qquad\quad m.\mathbf{out} = \boldsymbol{y}^{\tau_{\boldsymbol{y}}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{gen\text{-}stmt}\ (x := e) &\triangleq \mathsf{assign}(x, \mathsf{gen\text{-}exp}\ e) \\
\mathsf{gen\text{-}stmt}\ (\texttt{state}(x) := e) &\triangleq \overline{\mathsf{self}}\texttt{->}x = \mathsf{gen\text{-}exp}\ e \\
\mathsf{gen\text{-}stmt}\ (\texttt{if}\ e\ \{\ s_t\ \}\ \texttt{else}\ \{\ s_f\ \}) &\triangleq \texttt{if}\ (\ \mathsf{gen\text{-}exp}\ e)\ \{\ \mathsf{gen\text{-}stmt}\ s_t\ \}\ \texttt{else}\ \{\ \mathsf{gen\text{-}stmt}\ s_f\ \} \\
\mathsf{gen\text{-}stmt}\ (\boldsymbol{x} := i^{c}\,.\,f(\boldsymbol{e})) &\triangleq \mathsf{funcall}(\boldsymbol{x}, c, i, f, \mathsf{gen\text{-}exp}\ \boldsymbol{e}) \\
\mathsf{gen\text{-}stmt}\ (s_1\ ;\ s_2) &\triangleq \mathsf{gen\text{-}stmt}\ s_1\ ;\ \mathsf{gen\text{-}stmt}\ s_2 \\
\mathsf{gen\text{-}stmt}\ \texttt{skip} &\triangleq \texttt{skip}
\end{aligned}$$

(b) Translation of statements (translate_stmt, src/ObcToClight/Generation.v:167)

Figure 5.4: Translation function from Obc to Clight (p and ***outs*** represent the Obc program and the outputs of the translated Obc method)

```
struct nav self$;
double volatile x$;
_Bool volatile alarm$;
double volatile gps$;
double volatile xv$;
_Bool volatile s$;

int main(void) {
  struct fun$nav$step out$step;
  register double gps;
  register double xv;
  register _Bool s;
  fun$nav$reset(&self$);
  while (1) {
    gps = builtin volatile load float64(&gps$);
    xv = builtin volatile load float64(&xv$);
    s = builtin volatile load int8u(&s$);
    fun$nav$step(&self$, &out$step, gps, xv, s);
    builtin volatile store float64(&x$, out$step.x);
    builtin volatile store int8u(&alarm$, out$step.alarm);
  }
}
```

Clight

Listing 5.2: Generated entry point of the example

3. If the method has several outputs, then we use an additional locally declared output structure passed to the callee as a return pointer parameter, and the call is followed by a sequence of assignments from the fields of the structure, named according to the corresponding output declarations, to the proper locations. The implicit lifting of assign works as a fold left. For example, the call at line 49

   ```
   insr, alr := ins(insr).step([gps], [xv]);
   ```

   is translated into the sequence at lines 43 to 45

   ```
   fun$ins$step(&self->insr, &out$insr$step, gps, xv);
   insr = out$insr$step.x;
   alr = out$insr$step.alarm;
   ```

After translating the Obc classes one-by-one into a lists of structure and function definitions, the generation pass, named gen in the following, produces a global entry point. The resulting "main" function implements the general reactive scheme for a specific Lustre node, whose name, designated main-node in the following, is a parameter of the generation function. This name is nav in listing 5.2 which shows the generation of the global volatile input and output variables and of the entry point from the original Obc example program. The body of the entry point is shown in figure 5.5, where s_{main} and r_{main} designate the *step* and *reset* methods, respectively, of the main class that corresponds to the main node. First, the function corresponding to r_{main}, named $\mathsf{reset}_{\mathsf{main\text{-}node}}$, is called on a globally

$$
\begin{aligned}
\textsf{main-body} &\triangleq \textsf{reset}_{\textsf{main-node}}(\&\textsf{self})\,; \\
&\quad\ \textsf{main-loop} \\
\textsf{main-loop} &\triangleq \texttt{loop}\ \{\ \textsf{main-loop-body}\ \} \\
\textsf{main-loop-body} &\triangleq \textsf{read}\,; \\
&\quad\ \textsf{step-call}\,; \\
&\quad\ \textsf{write}\,;
\end{aligned}
$$

Where, given $s_{\textsf{main}}.\textbf{in} = {x_1}^{\tau_1^x} \cdots {x_i}^{\tau_i^x}$,

$$
\begin{aligned}
\textsf{read} \triangleq\ & \overline{x_1} = \texttt{vload}(\kappa^{\tau_1^x}, \&x_1)\,; \\
& \cdots \\
& \overline{x_i} = \texttt{vload}(\kappa^{\tau_i^x}, \&x_i)
\end{aligned}
$$

$$
\textsf{step-call} \triangleq \begin{cases}
\textsf{step}_{\textsf{main-node}}(\&\textsf{self}, \overline{x_1}, \cdots, \overline{x_i}) & \text{if } s_{\textsf{main}}.\textbf{out} = \varepsilon \\
\overline{y} = \textsf{step}_{\textsf{main-node}}(\&\textsf{self}, \overline{x_1}, \cdots, \overline{x_i}) & \text{if } s_{\textsf{main}}.\textbf{out} = [y^{\tau^y}] \\
\textsf{step}_{\textsf{main-node}}(\&\textsf{self}, \&\textsf{out}_{\textsf{step}}, \overline{x_1}, \cdots, \overline{x_i}) & \text{otherwise}
\end{cases}
$$

$$
\textsf{write} \triangleq \begin{cases}
\texttt{skip} & \text{if } s_{\textsf{main}}.\textbf{out} = \varepsilon \\
\texttt{vstore}(\kappa^{\tau^y}, \&y, \overline{y}) & \text{if } s_{\textsf{main}}.\textbf{out} = [y^{\tau_y}] \\
\texttt{vstore}(\kappa^{\tau_1^y}, \&y_1, \textsf{out}_{\textsf{step}}.y_1)\,; & \\
\cdots & \text{if } s_{\textsf{main}}.\textbf{out} = {y_1}^{\tau_1^y} \cdots {y_j}^{\tau_j^y} \\
\texttt{vstore}(\kappa^{\tau_j^y}, \&y_i, \textsf{out}_{\textsf{step}}.y_i) &
\end{cases}
$$

Figure 5.5: Generation of the entry point body (main_body, src/ObcToClight/Generation.v:389)

declared *self* structure. Then, the infinite main loop alternates (1) a sequence of volatile loads from globally declared input variables into local temporaries, (2) a call to the main *step* function, named $\mathsf{step}_{\mathsf{main\text{-}node}}$, with these temporaries as inputs, and (3) a sequence of volatile stores to globally declared output variables. Note that we omit the second statement of the loop construct since we do not use it—it is actually `skip`. The call to the main *step* function is also subject to the technicalities around the handling of return values.

5.3 Clight semantics

CompCert provides two semantic variants for Clight based on whether function parameters are considered as local variables[3] (Clight_1) or as temporary variables[4] (Clight_2). Since we never need to take the address of a function parameter, we use the Clight_2 variant, as it has one less level of indirection, which facilitates reasoning about programs.

CompCert also provides two styles of semantics: a small-step continuation-based operational semantics[5] and a big-step operational semantics[6]. The latter is described in [Blazy and Leroy (2009)] and proved sound with respect to the former. In the correctness proofs for the *reset* and *step* functions, we choose to work with the big-step semantics because (1) the generated code for these functions always terminates and never diverges, and (2) it is easier to reason between two big-step semantics. In the original version of CompCert, the big-step semantics, unlike the small-step one, is only defined for the Clight_1 variant, but we adapted it to handle the Clight_2 variant as well.

In the correctness proof of the entry point and thus of the overall program, we must, however, reason with the small-step semantics because the big-step one is not fine-grained enough to describe the observable behavior of the generated program. A generated Clight program runs forever, but the big-step semantics for programs cannot distinguish between *divergence* ("at some point, the program runs forever without doing any I/O" [Leroy (2019)]) and *reactive divergence* ("the program performs infinitely many I/O operations separated by finite amounts of internal computations").

Regardless of the variant and style, a single model is used to describe the state of a program's memory. The memory model is described in [Leroy and Blazy (2008)]. A memory state M is a collection of contiguous blocks, each identified by an integer b. Within a block, byte offsets δ within a fixed range are mapped to values. Hence, a memory M is a (partial) mapping from *locations* (b, δ) to values. Table 5.1 lists the basic memory operations that CompCert defines. Note that the alloc function never fails as CompCert models an infinite memory.

5.3.1 Big-step semantic rules for code generated from Obc classes

The big-step semantic rules for Clight are parameterized by:

[3] compcert.inria.fr/doc/html/compcert.cfrontend.Clight.html#function_entry1
[4] compcert.inria.fr/doc/html/compcert.cfrontend.Clight.html#function_entry2
[5] compcert.inria.fr/doc/html/compcert.cfrontend.Clight.html
[6] compcert.inria.fr/doc/html/compcert.cfrontend.ClightBigstep.html

Table 5.1: Operations over memory states [Blazy and Leroy (2009); Leroy (2019)]

$\mathsf{alloc}(M, i, j) = (M', b)$	Allocates a fresh block of bounds $[i, j)$ and returns the updated memory and the identifier of the allocated block.
$\mathsf{free}(M, b, i, j) \doteq M'$	Deallocates the range of offsets $[i, j)$ in the block b and returns, if the addresses are freeable, the updated memory.
$\mathsf{load}(\kappa, M, l) \doteq v$	Reads a quantity of consecutive bytes (as determined by the memory chunk κ) starting from the location l and returns, if the addresses are readable, the value read.
$\mathsf{store}(\kappa, M, l, v) \doteq M'$	Writes the value v as a quantity of consecutive bytes (as determined by κ) at location l and returns, if the addresses are writable, the updated memory.

- a *global environment* G that maps global variables to locations (with zero offset), function pointers to function definitions and structure names to their definitions,
- a *local environment* E that maps local variables to memory locations and types,
- a *temporary environment* L that directly maps temporaries to values, and
- a *memory* M.

We only present the semantics rules required for the subset of Clight used by the generation function, and, to further simplify the presentation, our rules are often combinations of the actual rules described in [Blazy and Leroy (2009)],[7] from which we keep the notations. Since we use only a restricted syntax subset of Clight, we only use an adapted relevant subset of the semantics rules. We present rules for the following restrictions of the subset of Clight presented in figure 5.3:

$$e ::= c \mid \overline{x} \mid \overline{x}\texttt{->}x \mid \texttt{\&}x \mid \texttt{\&(}\overline{x}\texttt{->}x\texttt{)} \mid x\,.\,x \mid \diamond^C\ e \mid e \oplus e$$

$$s ::= \overline{x}\texttt{->}x\ \texttt{=}\ \mathrm{a} \mid \overline{x}\ \texttt{=}\ \mathrm{a} \mid \texttt{if (}a\texttt{) \{}s\texttt{\} else \{}s\texttt{\}} \mid [\overline{x}\ \texttt{=}]\ x\texttt{(}a^*_{,}\texttt{)} \mid s\ \texttt{;}\ s \mid \texttt{skip} \mid \texttt{return}\ [a]$$

A generated expression is a constant, a temporary, an indirect field access on a temporary (the *self* pointer or the *out* pointer), a reference to a local output structure, a reference to an indirect field access on a temporary (a sub-state reference), a field access on a variable (a local output structure), or a unary or binary operation. A generated statement is an assignment to an indirect field access on a temporary, an assignment to a temporary, a conditional, a function call where the function expression is a variable, a sequence, the no-operation statement, or a return statement.

Adapting the notation of [Blazy and Leroy (2009)], we write $G, E, L \vdash a, M \Rightarrow v$ to denote that in the global environment G, environment E, temporary environment L and memory M, the expression a evaluates to a value v. The rules for the evaluation of generated expressions in r-value position are shown in figure 5.6a. A constant is evaluated using the dedicated constant semantics (e.g, rules (5) and (6) in [Blazy and Leroy (2009),

[7] Note that at the time, no temporaries were used.

$$\frac{}{G, E, L \vdash c, M \Rightarrow [\![c]\!]} \qquad \frac{}{G, E, L \vdash \overline{x}, M \Rightarrow L(\overline{x})}$$

$$\frac{\mathsf{temp\text{-}ind\text{-}field\text{-}loc}(G, L, \overline{x}, y) \doteq l \quad \mathsf{load}\left(\kappa^{\mathsf{type}(\overline{x}\texttt{->}y)}, M, l\right) \doteq v}{G, E, L \vdash \overline{x}\texttt{->}y, M \Rightarrow v}$$

$$\frac{E(x) = (b, \mathsf{type}\, x)}{G, E, L \vdash \&x, M \Rightarrow \mathsf{Vptr}\,(b, 0)} \qquad \frac{\mathsf{temp\text{-}ind\text{-}field\text{-}loc}(G, L, \overline{x}, y) \doteq l}{G, E, L \vdash \&(\overline{x}\texttt{->}y), M \Rightarrow \mathsf{Vptr}\, l}$$

$$\frac{\begin{array}{c}E(x) = (b, \texttt{struct}\ s) \quad \mathsf{type}\, x = \texttt{struct}\ s \quad \mathsf{comp}(G, s) \doteq \boldsymbol{\varphi} \\ \mathsf{field\text{-}offset}(G, y, \boldsymbol{\varphi}) \doteq \delta \quad \mathsf{load}\left(\kappa^{\mathsf{type}(x\,.\,y)}, M, (b, \delta)\right) \doteq v\end{array}}{G, E, L \vdash x\,.\,y, M \Rightarrow v}$$

$$\frac{G, E, L \vdash a, M \Rightarrow v_a \quad \mathsf{eval\text{-}unop}\left(\diamond^C, v_a, \mathsf{type}\, a, M\right) \doteq v}{G, E, L \vdash \diamond^C a, M \Rightarrow v}$$

$$\frac{G, E, L \vdash a, M \Rightarrow v_a \quad \mathsf{cast}(v_a, \mathsf{type}\, a, \tau, M) \doteq v}{G, E, L \vdash \tau\,(a), M \Rightarrow v}$$

$$\frac{\begin{array}{c}G, E, L \vdash a_1, M \Rightarrow v_1 \quad G, E, L \vdash a_2, M \Rightarrow v_2 \\ \mathsf{eval\text{-}binop}(G, \oplus, v_1, \mathsf{type}\, a_1, v_2, \mathsf{type}\, a_2, M) \doteq v\end{array}}{G, E, L \vdash a_1 \oplus a_2, M \Rightarrow v}$$

(a) Expression evaluation

$$\frac{\begin{array}{c}L(x) = \mathsf{Vptr}\,(b, \delta_x) \quad \mathsf{type}\,\overline{x} = \texttt{struct}\ s* \\ \mathsf{comp}(G, s) \doteq \boldsymbol{\varphi} \quad \mathsf{field\text{-}offset}(G, y, \boldsymbol{\varphi}) \doteq \delta_y\end{array}}{\mathsf{temp\text{-}ind\text{-}field\text{-}loc}(G, L, \overline{x}, y) \doteq (b, \delta_x + \delta_y)}$$

(b) Location of an indirect field access on a temporary

Figure 5.6 (I): Big-step semantics of Clight

Fig. 6, p. 9]). A temporary is simply looked up in the temporary environment. An indirect field access on a temporary is evaluated using temp-ind-field-loc, that we describe in the next paragraph, to get back the corresponding memory location, and load to return the stored value in the memory (rule (8)). The address of a variable x is evaluated to a pointer to the block associated with x in the environment (rules (9) and (1)). A reference to an indirect field access on a temporary is evaluated to a pointer to the location given by temp-ind-field-loc (rule (9)). A field access on a variable is evaluated to a value loaded from the location calculated by combining the block identifier looked up in E and the offset obtained by field-offset (rules (8), (3) and (1)). Unary, cast and binary operations recursively evaluate their arguments and use dedicated partial semantics functions, respectively eval-unop, cast and eval-binop (rules (10), (14) and (11) respectively). All these functions take the memory M as parameter, and eval-binop takes the global environment G as additional parameter to handle pointers.

Figure 5.6b presents a relation—we use our partiality notation—that we define to obtain the location of an indirect field access on a temporary $\overline{x}$->y. This relation is a combination of formal semantic rules ((3), (8) and (2), plus a rule for temporaries evaluation) that evaluates $\overline{x}$->y to a location. The block identifier b is obtained by looking up x in the temporary environment, verifying that the obtained value is a pointer to a location (b, δ_x). The offset is the sum of δ_x and δ_y, the relative offset of the field y in structure s (field-offset is described in [Blazy and Leroy (2009), §3.2, p 10]). The comp function retrieves the field declarations $\boldsymbol{\varphi}$ from the global declaration of s.

We write $G, E, L \vdash s, M \overset{t}{\Longrightarrow} out, L', M'$ to denote that in the global environment G, environment E, temporary environment L, the statement s terminates its execution in an updated temporary environment L' and memory M', with outcome out, producing an event trace t. There are only three possible outcomes in our subset of the semantics: Normal, the statement has terminated normally, Return, a function body has terminated, and Return (v, τ), similarly but the result of the function is the value v with type τ. The semantics for generated statements is presented in figure 5.6c. An assignment of an expression to an indirect field access on a temporary updates the memory by storing the properly cast value of the evaluated expression to the appropriate location given by temp-ind-field-loc (rule (20) in Fig. 8, p. 11). An assignment to a temporary simply updates the temporary environment with the result of the evaluated expression. A conditional statement recursively executes one of its branches according to the boolean projection of the evaluated condition. The projections are realized by the is-true and is-false predicates presented in [Blazy and Leroy (2009), §3.2, p. 11], that take M as additional parameter again for handling pointers. A function call is evaluated in several steps (rules (30) in Fig. 10, p. 13, (8), and (1)). The function variable f is not defined in the environment E but associated in the global environment G with a block identifier b (function pointer), which is in turn resolved in G to a function definition F_d. The type of f is checked against the prototype of F_d that must be a function type, composed of a list of argument types $\boldsymbol{\tau}_{args}$, a return type τ_{res} and a calling convention cc. The arguments are evaluated and cast to the corresponding types in $\boldsymbol{\tau}_{args}$. Then, the call is evaluated using a dedicated mutually defined judgment presented later in figure 5.6d, producing

$$\frac{\mathsf{temp\text{-}ind\text{-}field\text{-}loc}(G, L, \overline{x}, y) \doteq l \quad G, E, L \vdash a, M \Rightarrow v_a \quad \mathsf{cast}(v_a, \mathsf{type}\, a, \mathsf{type}\,(\overline{x}\texttt{->}y), M) \doteq v \quad \mathsf{store}\left(\kappa^{\mathsf{type}(\overline{x}\texttt{->}y)}, M, l, v\right) \doteq M'}{G, E, L \vdash \overline{x}\texttt{->}y = a, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Normal}, L, M'}$$

$$\frac{G, E, L \vdash a, M \Rightarrow v}{G, E, L \vdash \overline{x} = a, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Normal}, L\{x \mapsto v\}, M}$$

$$\frac{G, E, L \vdash a, M \Rightarrow v \quad \mathsf{is\text{-}true}(v, \mathsf{type}\, a, M) \quad G, E, L \vdash s_1, M \overset{t}{\Longrightarrow} out, L', M'}{G, E, L \vdash \texttt{if (}a\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, M \overset{t}{\Longrightarrow} out, L', M'}$$

$$\frac{G, E, L \vdash a, M \Rightarrow v \quad \mathsf{is\text{-}false}(v, \mathsf{type}\, a, M) \quad G, E, L \vdash s_2, M \overset{t}{\Longrightarrow} out, L', M'}{G, E, L \vdash \texttt{if (}a\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, M \overset{t}{\Longrightarrow} out, L', M'}$$

$$\frac{\begin{array}{c} f \notin E \quad \mathsf{symbol}(G, f) \doteq b \quad \mathsf{funct}(G, b) \doteq F_d \\ \mathsf{type}\, f = \mathsf{type\text{-}of\text{-}fundef}\, F_d = \boldsymbol{\tau}_{args}, \tau_{res}, cc \\ G, E, L \vdash \boldsymbol{\tau}_{args}(\boldsymbol{a}), M \Rightarrow \boldsymbol{v}_{args} \quad G \vdash F_d\,(\boldsymbol{v}_{args}), M \overset{t}{\Longrightarrow} v, M' \end{array}}{G, E, L \vdash [\overline{x} =] f\texttt{(}\boldsymbol{a}\texttt{)}, M \overset{t}{\Longrightarrow} out, \begin{cases} L\{x \mapsto v\} & \text{if } \overline{x} \text{ is given} \\ L & \text{otherwise} \end{cases}, M'}$$

$$\frac{G, E, L \vdash s_1, M \overset{t1}{\Longrightarrow} \mathsf{Normal}, L_1, M_1 \quad G, E, L_1 \vdash s_2, M_1 \overset{t_2}{\Longrightarrow} out, L_2, M_2}{G, E, L \vdash s_1\ \texttt{;}\ s_2, M \overset{t1+t_2}{\Longrightarrow} out, L_2, M_2}$$

$$\frac{G, E, L \vdash s_1, M \overset{t}{\Longrightarrow} out, L', M' \quad out \neq \mathsf{Normal}}{G, E, L \vdash s_1\ \texttt{;}\ s_2, M \overset{t}{\Longrightarrow} out, L', M'}$$

$$\frac{}{G, E, L \vdash \texttt{skip}, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Normal}, L, M}$$

$$\frac{}{G, E, L \vdash \texttt{return}, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Return}, L, M}$$

$$\frac{G, E, L \vdash a, M \Rightarrow v}{G, E, L \vdash \texttt{return}\ a, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Return}\,(v, \mathsf{type}\, a), L, M}$$

(c) Statement evaluation

Figure 5.6 (II): Big-step semantics of Clight

$$\frac{\begin{array}{c}\mathsf{alloc\text{-}vars}(G, M, F_d.\mathbf{vars}) \doteq (E, M_1) \\ L = \varnothing\{F_d.\mathbf{temps} \mapsto \mathsf{Vundef}\}\{F_d.\mathbf{params} \mapsto \boldsymbol{v}_{args}\} \\ \text{all names in } F_d.\mathbf{params} \text{ are distinct} \quad \text{all names in } F_d.\mathbf{vars} \text{ are distinct} \\ \text{names in } F_d.\mathbf{vars} \text{ and } F_d.\mathbf{temps} \text{ do not overlap}\end{array}}{\mathsf{function\text{-}entry}(F_d, \boldsymbol{v}_{args}, M) \doteq (E, L, M_1)}$$

$$\frac{\begin{array}{c}\mathsf{function\text{-}entry}(F_d, \boldsymbol{v}_{!args}, M) \doteq (E, L, M_1) \\ G, E, L \vdash F_d.\mathsf{body}, M_1 \overset{t}{\Longrightarrow} out, L', M_2 \quad M_2, out, F_d.\mathsf{return}\#v \quad \mathsf{free\text{-}env}(M_2, E) \doteq M'\end{array}}{G \vdash F_d\,(\boldsymbol{v}_{args}), M \overset{t}{\Longrightarrow} v, M'}$$

(d) Function call evaluation

Figure 5.6 (III): Big-step semantics of Clight

the overall trace t, the overall updated memory M' and a return value v. If a return temporary is supplied, the temporary environment is updated with the return value. A sequence of statements is executed recursively: either the first part terminates normally producing an intermediate state from which the second is executed, or it does not and the execution is interrupted. The `skip` statement does nothing, while the `return` statement yields a Return outcome with an optional pair of an evaluated value and a type.

Figure 5.6d gives two dedicated rules used for function call evaluation (rule (32) in Fig. 10, p. 13). The first presents a function-entry predicates that defines how the memory and environments are allocated at the entry of a function. The memory M is updated by the function alloc-vars that allocates blocks (using alloc) for each local variable declaration in the function definition F_d. It returns the allocated memory M_1 and the corresponding environment E. The temporary environment is created in two steps: first the register declarations are all bound to the undefined value Vundef, then the formal parameters are bound to the values $\boldsymbol{v}_{args}$ since we use the variant where parameters are temporaries. Finally, function-entry also ensures that no duplicates are found in the variable names. The second rule defines the evaluation of a function: the body of the function is evaluated in the state given by function-entry, the return value is computed from the return type of the function and the outcome of its body evaluation (we use the notation $M, out, \tau\#v$ of [Blazy and Leroy (2009), Fig.10, p. 13]), and the blocks in the environment E that were allocated for the local variables are freed by free-env (using free).

5.3.1.1 Semantic rules for the generated program

The generated *main* function requires three additional statements:

$s ::= \ldots \mid \overline{x}$ = `vload(`κ`,&`x`)` | `vstore(`κ`,&`x`,`a`)` | `loop {`s`}`

The volatile load and store operations always take the address of a global volatile variable, and we elide the type annotations, since we ensure that the types are the same for the formal and actual parameters of these operations.

$$\frac{y \notin E \quad \mathsf{symbol}(G, y) \doteq b \quad \mathsf{volatile}(G, b) = \mathsf{true} \quad w \overset{\kappa}{\underset{G}{\simeq}} v}{G, E, L \vdash \overline{x} = \texttt{vload}(\kappa, \&y), M \xrightarrow{[\mathcal{E}_{\mathsf{vload}}(\kappa, y, 0, w)]} \mathsf{Normal}, L\{x \mapsto v^{\kappa}\}, M}$$

$$\frac{y \notin E \quad \mathsf{symbol}(G, y) \doteq b \quad \mathsf{volatile}(G, b) = \mathsf{true} \quad G, E, L \vdash a, M \Rightarrow v \quad w \overset{\kappa}{\underset{G}{\simeq}} v^{\kappa}}{G, E, L \vdash \texttt{vstore}(\kappa, \&y, a), M \xrightarrow{[\mathcal{E}_{\mathsf{vstore}}(\kappa, y, 0, w)]} \mathsf{Normal}, L, M}$$

Figure 5.7: Volatile load and store operations evaluation

The statements we consider in the previous section do not produce events. Indeed, the only observable events in the generated program are produced by the volatile operations whose semantics is shown in figure 5.7. In both cases, the variable that is being read or written must be globally declared with the **`volatile`** attribute. The operation produces a single event $\mathcal{E}_{\mathsf{vload}}$ or $\mathcal{E}_{\mathsf{vstore}}$ parameterized by the chunk, the address (global name and offset) and the value being read or stored. The carried value is an *event value* w, that is, to simplify, the subset of standard values minus the undefined value and pointers that do not address global variables. We use the notation $w \overset{\kappa}{\underset{G}{\simeq}} v$ to denote the correspondence between the standard value v and the event value w (see `eventval_match`, CompCert/common/Events.v:269). This relation ensures a kind of well-typing property by using the type information of the chunk κ. We also write v^{κ} to model the normalization that may occur when reading a stored value, depending on the chunk and on the type of the value (see `load_result`, CompCert/common/Values.v:910).

As previously explained, to treat the infinite evaluation of the main loop of the generated program, we use the small-step semantics. The small-step semantics of Clight in CompCert is formalized as a state-transitions system that consists of a step relation, an initial state, a final state and a global environment. The rules that we present here are adapted from those of Cminor, that is, the next intermediate language after Clight in CompCert, and presented in [Appel and Blazy (2007); Leroy (2009a)], from which we keep the notations. We give the rules for the evaluation of the statements that we use in figure 5.8a. A Clight *regular state*, written $\mathcal{S}(F_d, s, k, E, L, M)$, comprises the current function F_d, the current statement s, a continuation k, an environment E, temporary environment L and memory M. We do not present the continuation system in detail, but simply note that s ; k reads "continue with s, then do as k" [Leroy (2009a)], which is enough to understand the first three rules that we use for sequences and loops. The last rule asserts a transition between a *call state*, written $\mathcal{C}(F_d, \boldsymbol{v}_{args}, k, M)$—that comprises the function F_d to be called, the list of argument values $\boldsymbol{v}_{args}$, a continuation k and a memory M—and the regular state built from the body of the function and the allocated environments and memory.

Figure 5.8b presents the rules defining the reflexive transitive closure and the rule for infinite reactive execution. In [Appel and Blazy (2007); Leroy (2009a)], T is a finite

$$\frac{}{G \vdash \mathcal{S}(F_d, (s_1 \ ; \ s_2), k, E, L, M) \xrightarrow{\varepsilon} \mathcal{S}(F_d, s_1, (s_2 \ ; \ k), E, L, M)}$$

$$\frac{}{G \vdash \mathcal{S}(F_d, \texttt{skip}, (s \ ; \ k), E, L, M) \xrightarrow{\varepsilon} \mathcal{S}(F_d, s, k, E, L, M)}$$

$$\frac{}{G \vdash \mathcal{S}(F_d, \texttt{loop } \{ \ s \ \}, k, E, L, M) \xrightarrow{\varepsilon} \mathcal{S}(F_d, s, (\texttt{loop } \{ \ s \ \} \ ; \ k), E, L, M)}$$

$$\frac{\mathsf{function\text{-}entry}(F_d, \boldsymbol{v}_{args}, M) \doteq (E, L, M_1)}{G \vdash \mathcal{C}(F_d, \boldsymbol{v}_{args}, k, M) \xrightarrow{\varepsilon} \mathcal{S}(F_d, F_d.\mathsf{body}, k, E, L, M_1)}$$

(a) Selected rules for statement evaluation

$$\frac{}{G \vdash S \xrightarrow{\varepsilon}{}^* S} \qquad \frac{G \vdash S \xrightarrow{t_1} S_1 \quad G \vdash S_1 \xrightarrow{t_2}{}^* S_2}{G \vdash S \xrightarrow{t_1 + t_2}{}^* S_2}$$

$$\frac{G \vdash S \xrightarrow{t}{}^* S' \quad t \neq \varepsilon \quad G \vdash S' \xrightarrow{T} \infty}{G \vdash S \xrightarrow{t +\!\!+ T} \infty}$$

(b) Reflexive transitive closure and infinite reactive transition

$$\frac{G \vdash S \xrightarrow{t_1}{}^* S_1 \quad G \vdash S_1 \xrightarrow{t_2} S_2}{G \vdash S \xrightarrow{t_1 + t_2}{}^* S_2} \qquad \frac{G \vdash S \xrightarrow{t}{}^* S' \quad G \vdash S' \xrightarrow{T} \infty}{G \vdash S \xrightarrow{t +\!\!+ T} \infty}$$

(c) Additional rules

$$\frac{\mathsf{initial\text{-}state}(P, S) \quad \mathsf{globalenv}(P) \vdash S \xrightarrow{T} \infty}{P \Downarrow \mathsf{Reacts}(T)}$$

$$\frac{\begin{array}{c} G = \mathsf{globalenv}(P) \quad \mathsf{initmem}(P) \doteq M \\ \mathsf{symbol}(G, P.\mathsf{main}) \doteq b \quad \mathsf{funct}(G, b) \doteq F_d \quad \mathsf{type\text{-}of\text{-}fundef}\ F_d = \varepsilon, \texttt{int}, cc_{\mathsf{default}} \end{array}}{\mathsf{initial\text{-}state}(P, \mathcal{C}(F_d, \varepsilon, \texttt{stop}, M))}$$

(d) Programs

Figure 5.8: Small-step semantics of Clight

or infinite trace, while here it may only be infinite. Indeed, T is akin to an infinite stream, in the sense that it must be *productive*. This leads to a different definition of the relation $G \vdash S \xrightarrow{T} \infty$: in the current development of CompCert, the finite prefix trace t is checked to be non empty to ensure that the resulting concatenated infinite trace is productive. This permits the distinction between divergence, where an infinite trace is not produced, and reactive divergence, where an infinite trace is produced.

Figure 5.8c presents additional rules, that are derived in CompCert as lemmas. The first one asserts that the reflexive transitive closure can also be derived with a step on the right rather than on the left. The second one is like the definition rule of the infinite reactive execution, without the constraint on the productivity.

The presented small-step rules are not sufficient for the whole generated program. We instead use the big-step rules presented in the last section for all of the proof except the execution of the *main* function. We rely on the fact that the big-step semantics is sound with respect to the small-step semantics.

Lemma 5.3.1 (exec_stmt_steps, CompCert/cfrontend/ClightBigstep.v:494)
Given a global environment G, a statement s, an environment E, two temporary environments L and L', two memories M and M', a trace t and an outcome out such that $G, E, L \vdash s, M \overset{t}{\Longrightarrow} out, L', M'$, then, for any function F_d and continuation k, there exists a state S such that $G \vdash \mathcal{S}(F_d, s, k, E, L, M) \xrightarrow{t}{}^{*} S$ and S is a regular state on F_d that is compatible with out, E, L' and M'.

Finally, figure 5.8d gives the rule for the infinite reactive execution of a program. The globalenv and initmem functions are described in [Leroy and Blazy (2008), Fig.5, p. 8]: globalenv builds a global environment from a given program, and initmem constructs an initial memory for executing the given program. The initial state is a call state on the main entry point of the program with no input values, the initial continuation `stop` and the initial memory.

5.3.2 Interfacing Vélus with CompCert

Recall that Vélus is entirely parameterized over an abstraction layer consisting of values, types and operators (see section 2.1.1). Before translating Obc code to Clight code, this abstraction layer must be instantiated with definitions from Clight.

Listing 5.3 shows the instantiations for values and types. Values are directly instantiated with CompCert values. CompCert values (see `val`, CompCert/common/Values.v:36) comprise the undefined value, machine integers, floating-point numbers and pointers to memory locations. We have already seen some of them in the presented semantic rules. Types are not directly instantiated with Clight types, as they are too rich: we do not need the void, pointer, array, structure or function types in the models of NLustre, Stc, and Obc. Hence we define our own set of types and a conversion function `cltype` to translate them to Clight types. Our types comprise only integer and floating-point types. This function was ignored when presenting the Clight generation function, and will often be in the following to facilitate the presentation.

```
Definition val: Type := Values.val.

Inductive type : Type :=
| Tint:   Ctypes.intsize -> Ctypes.signedness -> type
| Tlong:  Ctypes.signedness -> type
| Tfloat: Ctypes.floatsize -> type.

Definition cltype (ty: type) : Ctypes.type :=
  match ty with
  | Tint sz sg => Ctypes.Tint sz sg Ctypes.noattr
  | Tlong sg   => Ctypes.Tlong sg (Ctypes.mk_attr false (Some (Npos 3)))
  | Tfloat sz  => Ctypes.Tfloat sz Ctypes.noattr
  end.
```

Coq (src/ObcToClight/Interface.v:39-53)

Listing 5.3: Instantiation of values and types

We saw in the generation function (see figure 5.4a) that Vélus unary operators are instantiated with Clight unary operators and the type cast operator, while Vélus binary operators are directly instantiated with Clight binary operators. The corresponding semantic functions are instantiated in the following way, safely providing the empty memory and empty global environment as additional parameters of the CompCert functions:

$$[\![\diamond^C]\!]_\tau\ v = \mathsf{eval\text{-}unop}\Big(\diamond^C, v, \mathsf{cltype}\,\tau, \varnothing\Big)$$
$$[\![\oplus]\!]_{\tau_1\times\tau_2}\ v_1\ v_2 = \mathsf{eval\text{-}binop}(\varnothing, \oplus, v_1, \mathsf{cltype}\,\tau_1, v_2, \mathsf{cltype}\,\tau_2, \varnothing)$$

For a type cast, the function is instantiated by the corresponding semantic rule in figure 5.6a. We follow the same approach for the typing functions of the operators.

5.4 Separation logic and key invariants

The central challenge in reasoning about the correctness of Clight generation is the change of memory model. In Obc, the memory is modeled as a tree of environments where the separation of individual values is manifest. In contrast, Clight has a memory model that maps addresses to the bytes that comprise values, and the details of type sizes, alignment, and aliasing through pointers cannot be ignored. We apply SL [Ishtiaq and O'Hearn (2001); Reynolds (2002)] to cope with these complications which otherwise quickly overwhelm the proof effort. We use an SL library that is already designed within CompCert for the correctness of one of its backend passes.

SL is an extension of Floyd-Hoare logic [Floyd (1967); Hoare (1969)] that facilitates reasoning about programs that manipulate pointers and complex data-structures. The main innovation is the *separating conjunction* operator $*$, that enables to write program

assertions like $\{x \mapsto v * y \mapsto w\}$ that describes a memory state where the variable x holds the address of a memory location where the value v is stored, *and separately* where the variable y holds the address of a memory location where the value w is stored. The "and separately" differs from a classical "and" in that it asserts that both components describe distinct areas of the memory: the bytes comprising the values of x and y are distinct. A second operator, the *separating implication* $-\!\!*$ [O'Hearn and Pym (1999)], or "magic wand", is sometimes useful. The assertion $P -\!\!* Q$ means that given a separate memory area satisfying P, the combined memory area satisfies Q.

A dedicated rule, called *frame rule*, is added to the logic to extend local reasoning on a statement to a larger separate context by stating that unmentioned area in the memory remains unchanged:

$$\frac{\{P\}\, s\, \{Q\}}{\{P * R\}\, s\, \{Q * R\}} \quad \text{where no free variable in } R \text{ is modified by the statement } s$$

O'Hearn (2019) gives an introduction and overview of SL.

5.4.1 Separation Logic in CompCert

Citing its documentation [Leroy (2019)], the SL that CompCert provides "is not a full-fledged separation logic because there is no program logic (Hoare triples) to speak of. Also, there is no general frame rule; instead, a weak form of the frame rule is provided by the lemmas that help us reason about the logical assertions." A memory assertion is composed of four components, gathered in a Coq dependent record:

1. A *predicate* over the memory, that is, the logical content of the assertion.
2. The memory *footprint* of the assertion, that is, the set of locations it concerns.
3. A proof that the logical content is invariant under changes to locations outside the footprint, thus capturing the essence of the frame rule.
4. A proof that the blocks of the footprint are allocated.

Following CompCert notation, we write $M \vDash P$ to designate the predicate of the memory assertion P, applied to the memory M, and we add the notation $(\!|P|\!)$ to designate its footprint. In typical SL presentations, one writes $s, h \vDash P$ where s is a *store* mapping identifiers to values (containing addresses), and h is a *heap* mapping addresses to values. In the CompCert SL library however, only the heap is used, therefore assertions do not mention identifiers but only memory locations, that is, addresses. The CompCert library, that we slightly modify, builds on this definition to define useful SL operators and special assertions:

implication The usual implication is lifted over memory assertions and adjusted to account for footprints. We write $P \twoheadrightarrow Q$ if and only if, for any memory M, $M \vDash P \rightarrow M \vDash Q$ and $(\!|Q|\!) \subseteq (\!|P|\!)$. The associated equivalence is written $P \twoheadleftarrow\!\!\twoheadrightarrow Q$. Implication is shown to be a reflexive and transitive relation, while the equivalence is shown to be an equivalence relation.

Table 5.2: Memory permissions in CompCert [Leroy (2019)]

	free	store	load	pointer comparison
Freeable	✓	✓	✓	✓
Writable		✓	✓	✓
Readable			✓	✓
Nonempty				✓
Empty				

separating conjunction We write $P * Q$ to designate the memory assertion such that:

1. $\forall M,\ M \vDash P * Q \leftrightarrow (M \vDash P \wedge M \vDash Q \wedge (\!|P|\!) \cap (\!|Q|\!) = \emptyset)$
2. $(\!|P * Q|\!) = (\!|P|\!) \cup (\!|Q|\!)$

We see here that the CompCert model for memory assertion allows to model the separating conjunction without explicitly exposing *heaplets*, that is, separated parts of the memory, like other mechanizations of SL do [Klein, Kolanski, and Boyton (2012); Appel, Dockins, et al. (2014)].

pure assertion A *pure* assertion is a memory assertion that ignores the memory. We write $\mathsf{pure}(P)$ where P is a predicate to designate the memory assertion such that:

1. $\forall M,\ M \vDash \mathsf{pure}(P) \leftrightarrow P$
2. $(\!|\, \mathsf{pure}(P) \,|\!) = \emptyset$

range CompCert allows to assert that a range of consecutive bytes in the memory is allocated with a given permission[8] and unspecified content. We write $b : [i, j)^p$, where b is a block identifier, p a permission and i, j are integers, for the memory assertion such that:

1. $\forall M,\ M \vDash b : [i, j)^p \leftrightarrow 0 \leq i \wedge j \leq 2^N \wedge \forall \delta \in [i, j),$ the location (b, δ) has permission p
   where N is the target architecture word size.
2. $(\!|b : [i, j)^p|\!) = \{(b', \delta) \mid b = b' \wedge \delta \in [i, j)\}$

Table 5.2 lists the permissions, or access rights, that CompCert defines, with the memory operations that they allow. In the following, we will write $b : [i, j)$ for a range of bytes with freeable permission.

contains CompCert provides a way to assert that a memory area with a given permission[9] contains a value with a given specification. We write $l : P^p_\kappa$ where $l = (b, \delta)$ is a memory location, κ is a memory chunk, P is a predicate over values and p a permission, for the memory assertion such that:

[8] For *any* permission in the original version of CompCert.
[9] Only *Freeable* in the original version.

1. $\forall M,\ M \vDash (b,\delta) : P^p_\kappa \leftrightarrow 0 \le \delta \wedge \delta + |\kappa| \le 2^N$
   $\wedge\ \forall \delta' \in [\delta, \delta + |\kappa|),\ (b,\delta')$ has permission p
   $\wedge\ \delta$ is properly aligned relative to κ
   $\wedge\ \exists v,\ \mathsf{load}(\kappa, M, (b,\delta)) \doteq v \wedge P\ v$
   where $|\kappa|$ is the size information of the chunk κ.
2. $(\!| (b,\delta) : P^p_\kappa |\!) = \{(b',\delta') \mid b = b' \wedge \delta' \in [\delta, \delta + |\kappa|)\}$

CompCert provides several results giving a SL specification to the memory operations of table 5.1 on page 151.

Lemma 5.4.1 (alloc_rule, CompCert/common/Separation.v:411)
Given two memories M and M', two integer bounds $0 \le i$ and $j \le 2^N$, a block b and an assertion R, such that $\mathsf{alloc}(M,i,j) = (M',b)$ and $M \vDash R$, then $M' \vDash b : [i,j) * R$.

Lemma 5.4.2 (free_rule, CompCert/common/Separation.v:427)
Given a memory M, two integer bounds i and j, a block b and an assertion R, such that $M \vDash b : [i,j) * R$, then there exists a memory M' such that $\mathsf{free}(M,b,i,j) \doteq M'$ and $M' \vDash R$.

Lemma 5.4.3 (load_rule, CompCert/common/Separation.v:584)
Given a memory M, a location l, a chunk κ, a permission p that is at least readable and a value specification P such that $M \vDash l : P^p_\kappa$, then there exists a value v such that $\mathsf{load}(\kappa, M, l) \doteq v$ and $P\ v$ holds.

Lemma 5.4.4 (store_rule, CompCert/common/Separation.v:604)
Given a memory M, a location l, a chunk κ, a permission p that is at least writable, a value v, value specifications P and P' and an assertion R such that $M \vDash l : P^p_\kappa * R$ and $P'\ v^\kappa$, then there exists a memory M' such that $\mathsf{store}(\kappa, M, l, v) \doteq M'$ and $M' \vDash l : {P'}^p_\kappa * R$.

All these results except the load-related one include an additional assertion R: this implements the "weak form of the frame rule". With a separate frame rule, those lemmas could be written in the usual SL triplet-style as axioms, assuming in-place modification of the memory and abusing the notation:

$$\frac{}{\{\mathsf{emp}\}\ \mathsf{alloc}(i,j)\ \{\lambda b.\, b : [i,j)\}} \qquad \frac{}{\{b : [i,j)\}\ \mathsf{free}(b,i,j)\ \{\lambda\,().\,\mathsf{emp}\}}$$

$$\frac{}{\{l : P^p_\kappa\}\ \mathsf{load}(\kappa,l)\ \{\lambda v.\ \mathsf{pure}(P\ v) * l : P^p_\kappa\}} \quad \text{if } p \text{ is at least readable}$$

$$\frac{}{\{\mathsf{pure}(P'\ v^\kappa) * l : P^p_\kappa\}\ \mathsf{store}(\kappa,l,v)\ \{\lambda\,().\, l : {P'}^p_\kappa\}} \quad \text{if } p \text{ is at least writable}$$

5.4.2 Vélus extensions

To fit our needs, we extend the SL library of CompCert with additional operators and abbreviations:

empty heap (sepemp, src/ObcToClight/MoreSeparation.v:644)
We define the shorthand $\mathsf{emp} \triangleq \mathsf{pure}(\top)$. We also show that emp is a neutral element for the separating conjunction $*$.

false assertion (sepfalse, src/ObcToClight/MoreSeparation.v:739)
The bold shorthand $\bot \triangleq \mathsf{pure}(\bot)$ simply asserts contradictory cases in memory assertions.

separating implication (sepwand, src/ObcToClight/MoreSeparation.v:98)
We write $P \mathrel{-\!\!*} Q$ for the assertion such that:

1. $\forall M,\ M \vDash P \mathrel{-\!\!*} Q \leftrightarrow \forall M'$ invariant on $(\!|P \mathrel{-\!\!*} Q|\!)$ relative to M,
$$M' \vDash P \ \to\ M' \vDash Q$$
$$\wedge\ (\!|P \mathrel{-\!\!*} Q|\!) \text{ is allocated}$$
2. $(\!|P \mathrel{-\!\!*} Q|\!) = (\!|Q|\!) \setminus (\!|P|\!)$

generalized separating conjunction (sepall, src/ObcToClight/MoreSeparation.v:803)
Given a list $\boldsymbol{a} = a_1 \cdots a_n$ of elements of type A and a function P from A to memory assertions, we define $\divideontimes_{\boldsymbol{a}} P \triangleq a_1 * \cdots * a_n$.

We show the fundamental property about separating implication.

Lemma 5.4.5 (sep_unwand, src/ObcToClight/MoreSeparation.v:182)
Given two assertions P and Q such that the membership of $(\!|P|\!)$ is decidable, then $P * (P \mathrel{-\!\!*} Q) \mathrel{-\!\!\triangleright} Q$.

5.4.3 Separation invariants for the proof of correctness

The proof of correctness at method-execution level is based on a state correspondence predicate that relates an Obc state and a Clight state. The correspondence uses SL to relate the tree-like Obc model with explicit separation of data to the contiguous blocks memory model of Clight.

Given a complete Obc state, that is, a program p, a class *cls* with name c, a method m, a memory tree *me* and an environment *ve*, and a complete Clight state, that is, a global environment G, a location $l_{\mathsf{self}} = (b_{\mathsf{self}}, \delta_{\mathsf{self}})$ that contains the current state values (the pointer *self*), an optional pair bco_{out} of a block identifier and the fields of the structure that holds the output values (the *out* pointer if it exists), an environment E and a temporary environment L, there are four elements that we want to relate:

1. the state of the class *cls* and the data stored at l_{self},
2. the output values of m and the corresponding output representation in Clight (nothing if there are no output, a temporary if there is only one and the structure otherwise),
3. the output values of sub-instance method calls appearing in the body of m with local output structures or temporary declarations,

4. local variables of m in the Obc environment and corresponding variables in the Clight temporary environment.

5.4.3.1 Definitions

Most of the correspondence predicates rely on a relation between a defined or undefined value in Obc and a value in Clight. The idea is to use this relation as a specification for variables found in the memory. When a variable is undefined in Obc, its value in Clight is unconstrained, otherwise it has the same value in Clight. We denote this predicate with the notation $\lceil v \rceil$ where v is a defined or undefined value.

Definition 5.4.1 (match_value, src/ObcToClight/SepInvariant.v:51)

$$\frac{}{\lceil \lfloor \_ \rfloor \rceil\, v} \qquad\qquad \frac{v' = v}{\lceil \lfloor v' \rfloor \rceil\, v}$$

Class state correspondence

The following predicate relates the memory tree me for a class with name c in Obc and the block holding the corresponding state in Clight at location l_{self}. It iterates through the program p until it finds the class named c and constructs a separating conjunction over state variables and recursively over sub-instances. That is, it follows the tree structure and asserts at the leaves that the Clight memory contains corresponding values.

Definition 5.4.2 (staterep, src/ObcToClight/SepInvariant.v:73)
We define the memory assertion $\mathsf{s\text{-}rep}^{p}_{G}\, c\ me\ l_{self}$ *by cases on* p *as follows:*

$$\mathsf{s\text{-}rep}^{cls\cdot p'}_{G}\ c\ me\ l_{self} \triangleq \begin{cases} \ast_{cls.\mathbf{regs}}\ \mathsf{r\text{-}rep}_{G}\ cls\ me\ l_{self} * \ast_{cls.\mathbf{insts}}\ \mathsf{i\text{-}rep}^{p'}_{G}\ cls\ me\ l_{self} & \text{if } cls.\mathbf{name} = c \\ \mathsf{s\text{-}rep}^{p'}_{G}\ c\ me\ l_{self} & \text{otherwise} \end{cases}$$

$$\mathsf{s\text{-}rep}^{\varepsilon}_{G}\ c\ me\ l_{self} \triangleq \bot$$

where

$$\mathsf{r\text{-}rep}_{G}\ cls\ me\ (b_{self}, \delta_{self}) \triangleq \lambda\, x^{\tau}.\ \begin{cases} (b_{self}, \delta_{self} + \delta_x) : \lceil me((x)) \rceil^{writable}_{\kappa^{\tau}} & \text{if } \mathsf{field\text{-}offset}(G, x, \mathsf{fields}\ cls) \doteq \delta_x \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{i\text{-}rep}^{p}_{G}\ cls\ me\ (b_{self}, \delta_{self}) \triangleq \lambda\, i^{c'}.\ \begin{cases} \mathsf{s\text{-}rep}^{p}_{G}\ c'\ (\mathsf{sub}\ i\ me)\ (b_{self}, \delta_{self} + \delta_i) & \text{if } \mathsf{field\text{-}offset}(G, i, \mathsf{fields}\ cls) \doteq \delta_i \\ \bot & \text{otherwise} \end{cases}$$

The r-rep function asserts the correspondence for state variables within the memory tree. For each state variable of the current class, it ensures that a value is

found at the proper address, using the offset in the structure, given by field-offset applied to a list of fields calculated from the class definition by fields (make_members, src/ObcToClight/Generation.v:279). This value must be equal to the one found in the Obc memory tree if it is defined and unconstrained otherwise. Since the overall state structure is stored in a static variable, the expected permission is *writable* rather than *freeable*.

The i-rep function asserts the correspondence for sub-trees. For each sub-instance of the current class, it recursively asserts s-rep at the proper offset in the structure.

Figure 5.9 shows the s-rep instantiation on the running example. In figure 5.9a, the *me* tree on the left represents the memory tree of an instance of the class nav. Thin edges represent the state variables, while thick edges represent sub-instances. On the right is a representation of the memory layout of the generated Clight structure at location $(b_{\mathsf{self}}, \delta_{\mathsf{self}})$. Each field is accessed through a relative offset: given a class *cls* named c, we write δ_x^c for the offset calculated by $\mathsf{field\text{-}offset}(G, x, \mathsf{fields}\ cls)$. The Obc tree and the Clight memory are related by the s-rep predicate, shown by gray dashed arrows. The displayed equation recursively unveils this correspondence, descending the tree on the left and adding offsets on the right—we omit the p and G parameters for clarity. The final result is a set of primitive separated conjuncts asserting correspondence with the values in the tree. This set of conjuncts together with the memory layout they specify is shown in figure 5.9b, where we write $b = b_{\mathsf{self}}$, $\delta_{\mathsf{self}} = 0$, since a global variable holds the main structure, and assume a 64 bit architecture. The gray area represents the memory locations that are allocated and specified by the separating assertions, and the white area is padding, that ensures correct alignment.

We combine s-rep with an assertion that the pointer *self* is declared as a temporary parameter. We also ensure that the size of the structure corresponding to the class *cls* does not exceed the maximum machine integer, but we omit this constraint in this presentation.

Definition 5.4.3 (selfrep, src/ObcToClight/SepInvariant.v:1276)

$$\mathsf{self\text{-}rep}_G^p\ cls\ me\ L\ l_{self} \triangleq \mathsf{pure}(L(self) = \mathsf{Vptr}\ l_{self}) * \mathsf{s\text{-}rep}_G^p\ cls.name\ me\ l_{self}$$

Output correspondence

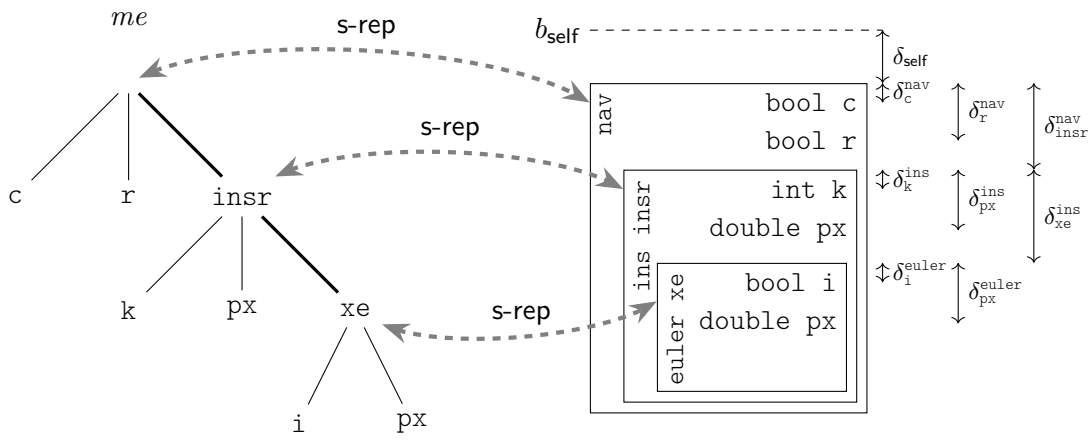
To define the predicate relating the output values of method m with the values held in a local structure, we start by defining a generic memory assertion to put local variables in correspondence with the fields of a structure.

Definition 5.4.4 (fieldsrep, src/ObcToClight/SepInvariant.v:592)
Given an Obc environment ve, a list of structure fields φ and a block identifier b, we define the following memory assertion:

$$\mathsf{fs\text{-}rep}_G\ ve\ \varphi\ b \triangleq \mathop{\Large *}_{\varphi} \mathsf{f\text{-}rep}_G\ ve\ \varphi\ b$$

where

$$\mathsf{f\text{-}rep}_G\ ve\ \varphi\ b \triangleq \lambda\, x^{\tau}.\ \begin{cases} (b, \delta_x) : \lceil ve((x)) \rceil_{\kappa^{\tau}} & \text{if } \mathsf{field\text{-}offset}(G, x, \varphi) \doteq \delta_x \\ \bot & \text{otherwise} \end{cases}$$

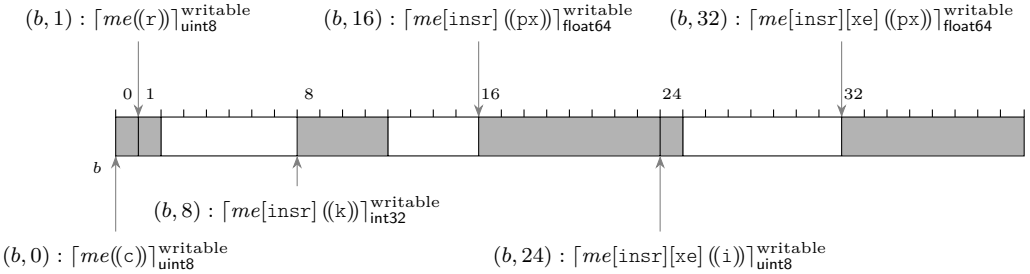


$$
\begin{aligned}
&\textsf{s-rep}\,\texttt{nav}\ me\ (b_{\textsf{self}}, \delta_{\textsf{self}}) = \\
&\quad (b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{c}}) : \lceil me((\texttt{c})) \rceil^{\text{writable}}_{\textsf{uint8}} \\
&* (b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{r}}) : \lceil me((\texttt{r})) \rceil^{\text{writable}}_{\textsf{uint8}} \\
&* \underbrace{\textsf{s-rep}\,\texttt{ins}\ me[\texttt{insr}]\ (b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{insr}})}
\end{aligned}
$$

$$
\begin{aligned}
&\quad (b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{insr}} + \delta^{\texttt{ins}}_{\texttt{k}}) : \lceil me[\texttt{insr}]\,((\texttt{k})) \rceil^{\text{writable}}_{\textsf{int32}} \\
&* \left(b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{insr}} + \delta^{\texttt{ins}}_{\texttt{px}}\right) : \lceil me[\texttt{insr}]\,((\texttt{px})) \rceil^{\text{writable}}_{\textsf{float64}} \\
&* \underbrace{\textsf{s-rep}\,\texttt{euler}\ me[\texttt{insr}][\texttt{xe}]\ \left(b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{insr}} + \delta^{\texttt{ins}}_{\texttt{xe}}\right)}
\end{aligned}
$$

$$
\begin{aligned}
&\quad \left(b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{insr}} + \delta^{\texttt{ins}}_{\texttt{xe}} + \delta^{\texttt{euler}}_{\texttt{i}}\right) : \lceil me[\texttt{insr}][\texttt{xe}]\,((\texttt{i})) \rceil^{\text{writable}}_{\textsf{uint8}} \\
&* \left(b_{\textsf{self}}, \delta_{\textsf{self}} + \delta^{\texttt{nav}}_{\texttt{insr}} + \delta^{\texttt{ins}}_{\texttt{xe}} + \delta^{\texttt{euler}}_{\texttt{px}}\right) : \lceil me[\texttt{insr}][\texttt{xe}]\,((\texttt{px})) \rceil^{\text{writable}}_{\textsf{float64}}
\end{aligned}
$$

(a) The recursive behavior



(b) Memory layout on a 64 bit architecture

Figure 5.9: The s-rep predicate on the example

The f-rep function is similar to r-rep, but there is no initial offset in the location (it is zero), and it uses the environment ve to give a specification on the memory variable. Moreover, the accessed memory must be freeable rather than only writable.

If m has only one output however, the value is stored in a temporary. In this case, we relate local variables of m with the temporaries of the corresponding Clight function.

Now we can define the memory assertion asserting the correspondence for the outputs of the considered method m.

Definition 5.4.5 (outputrep, src/ObcToClight/SepInvariant.v:1184)

$$\mathsf{out\text{-}rep}_G\ cls\ m\ ve\ L\ bco_{out} \triangleq$$

$$\begin{cases} emp & \text{if } m.\mathbf{out} = \varepsilon \\ \mathsf{pure}(\lceil ve(\!(x)\!)\rceil\ L(x)) & \text{if } m.\mathbf{out} = [x^\tau] \\ \mathsf{pure}(L(out) = \mathsf{Vptr}\ (b_{out}, 0)) \\ \quad * \mathsf{pure}(\mathsf{comp}(G, f_c) \doteq \varphi_{out}) & \text{if } 1 < \|m.\mathbf{out}\| \text{ and } bco_{out} = \mathsf{Some}\ (b_{out}, \varphi_{out}) \\ \quad * \mathsf{fs\text{-}rep}_G\ ve\ \varphi_{out}\ b_{out} \\ \bot & \text{otherwise} \end{cases}$$

This function produces different assertions according to the number of outputs of m:

- If m has no output, there is no correspondence; the memory assertion is simply the empty heap.
- If m has a unique output x^τ then we assert the correspondence between the local Obc variable x and the Clight temporary $\overline{x}$.
- If m has strictly more than one output, then we ensure that (1) the output pointer parameter *out* is declared as a temporary, (2) the corresponding structure is globally declared and (3) the structure pointed to by the pointer *out* is in correspondence with the according output values in ve.

In the third case, the output structure is named f_c: for simplicity, it has the same name as the generated function corresponding to m. In the actual development, we use the special symbol '$' to create fresh unique names. In particular, each method named f of a class named c is translated into a function named $fun\$c\$f$, and the same name is used for a corresponding output structure declaration.

One limitation of the CompCert SL library is that we cannot define *existential* memory assertions. Indeed, in the third case above, rather than provide an optional parameter bco_{out} from the outside, we would have preferred that b_{out} and φ_{out} were existentially quantified within the definition.

Sub-instances outputs correspondence

Our handling of multiple outputs in Clight necessitates that we maintain a correspondence between multiple output values returned by method calls appearing in the body of m and Clight output structures declared in the environment E (see item 3 on page 148 about the translation of function calls).

Definition 5.4.6 (subrep, src/ObcToClight/SepInvariant.v:835)

$$\mathsf{os\text{-}rep}_G\, m\ E \triangleq \bigstar_{\mathsf{mult\text{-}outs}\, m} \mathsf{o\text{-}rep}_G\, E$$

where

$$\mathsf{o\text{-}rep}_G\, E \triangleq \lambda\, x^\tau.\ \begin{cases} \mathsf{fs\text{-}rep}_G\, \varnothing\ \varphi\ b & \text{if } E(x) = (b, \tau) \wedge \tau = \mathtt{struct}\ s \wedge \mathsf{comp}(G, s) \doteq \varphi \\ \bot & \text{otherwise} \end{cases}$$

The idea is to assert that for each sub-call in m with strictly more than one output, there exists a corresponding output structure s and a local Clight variable that points to an instance of it in the memory with unspecified content—the content is only specified after a method call. We use the function mult-outs to retrieve the names and types of these (fresh) local variables. The function satisfies the following property.

$$\begin{aligned} x^\tau \in \mathsf{mult\text{-}outs}\, m \leftrightarrow \exists i\, c'\, f',\ & \boldsymbol{y} := i^{c'}.\, f'(\text{-}) \text{ appears in } m \text{ with } 1 < \|\boldsymbol{y}\| \\ & \wedge\ x = i_{f'} \\ & \wedge\ \tau = \mathtt{struct}\ f'_{c'} \end{aligned}$$

The os-rep predicate asserts that memory is allocated with enough room to hold the return values of sub-calls. The fs-rep function does not account for fields alignment, consequently it only asserts that a relevant data area is allocated, but it does not assert anything about necessary padding area. This is a problem, not because the padding is ever read or written, but rather because we must prove that the whole structure can be freed when the enclosing function returns. Technically, we must justify the free-env predicate in the rule shown in figure 5.6d on page 155, which requires invoking the "free rule" (lemma 5.4.2 on page 162), which, in turn, requires "ownership" of that area of memory as expressed in a memory assertion. This is also the reason we extend the SL library of CompCert with a separating implication operator. We define a memory assertion that asserts the allocation of enough room to hold a whole structure.

Definition 5.4.7 (subrep_range, src/ObcToClight/SepInvariant.v:870)

$$\mathsf{os\text{-}range}_G\, E \triangleq \bigstar_{\mathsf{bindings}\, E} \lambda\ (x, (b, \tau)).\, b : [0, \mathsf{sizeof}(G, \tau))$$

The function bindings turns an environment into the corresponding association list.

Now we can define a memory assertion that uses the separating implication to express ownership over the locations used to store field values ($\mathsf{os\text{-}rep}_G\, f\ E$) and keep the possibility of relinquishing this ownership to assert ownership over the whole range ($\mathsf{os\text{-}range}_G\, E$), without ever having to detail exactly where the padding is.

Definition 5.4.8 (outputsrep, src/ObcToClight/SepInvariant.v:1293)

$$\mathsf{outs\text{-}rep}_G\ m\ E \triangleq \mathsf{os\text{-}rep}_G\ f\ E * (\mathsf{os\text{-}rep}_G\ f\ E \mathrel{-\!\!*} \mathsf{os\text{-}range}_G\ E)$$

In the actual development we add an additional pure assertion ensuring that variables of the domain of E are all of the form *out*$-$-, to make sure that generated function names, of the form *fun*$-$- cannot be found in E (see the rule for function calls in figure 5.6c).

Local variables correspondence

The last correspondence is between local Obc variables that are not outputs, and Clight temporaries. The corresponding memory assertion is entirely pure since temporaries are directly bound to values.

Definition 5.4.9 (varsrep, src/ObcToClight/SepInvariant.v:1057)

$$\mathsf{vars\text{-}rep}\ m\ ve\ L \triangleq \mathsf{pure}(\forall x^\tau \in m.\textbf{\textit{in}} + m.\textbf{\textit{vars}},\ \lceil ve(\!(x)\!)\rceil\ L(x))$$

The correspondence predicate

The whole state correspondence predicate can then be defined as a memory assertion by *separately* combining all the independent assertions.

Definition 5.4.10 (match_states, src/ObcToClight/SepInvariant.v:1298)

$$\begin{aligned}\mathsf{match\text{-}state}^p_G\ cls\ m\ (me, ve)\ (E, L)\ l_{self}\ bco_{out} \triangleq\ & \mathsf{self\text{-}rep}^p_G\ cls\ me\ L\ l_{self}\\ & * \mathsf{out\text{-}rep}_G\ cls\ m\ ve\ L\ bco_{out}\\ & * \mathsf{outs\text{-}rep}_G\ m\ E\\ & * \mathsf{vars\text{-}rep}\ m\ ve\ L\end{aligned}$$

This memory assertion is the key invariant of the correctness proof: it provides all the information needed while ensuring the separation in the Clight side.

5.4.3.2 Properties

Loading operations

We use s-rep and fs-rep to prove dedicated extensions of the "load rule" (lemma 5.4.3 on page 162) for looking up values of expressions like $\overline{\mathsf{self}}\text{->}x$, $\overline{\mathsf{out}}\text{->}x$ or $i.x$ in the memory.

Lemma 5.4.6 (staterep_deref_mem, src/ObcToClight/SepInvariant.v:455)
Given a program p containing a class cls named c, a memory tree me and a memory M such that $M \vDash \mathsf{s\text{-}rep}^p_G\ c\ me\ (b_{self}, \delta_{self})$, then, for any state variable declaration $x^\tau \in cls.\textbf{\textit{regs}}$, $\mathsf{load}(\kappa^\tau, M, (b_{self}, \delta_{self} + \mathsf{field\text{-}offset}(G, x, \mathsf{fields}\ cls))) \doteq me(x)$.

Lemma 5.4.7 (fieldsrep_deref_mem, src/ObcToClight/SepInvariant.v:595)
Given a memory M, a list of fields φ, and a block identifier b such that $M \vDash \mathsf{fs\text{-}rep}_G\ ve\ \varphi\ b$, then, for any field $x^\tau \in \varphi$, $\mathsf{load}(\kappa^\tau, M, (b, \mathsf{field\text{-}offset}(G, x, \varphi))) \doteq ve(x)$.

Storing operations

Similarly, we specialize the "store rule" (lemma 5.4.4 on page 162) for the s-rep invariant. As storing operation updates the memory, so we must show the preservation of the invariant used as an hypothesis.

Lemma 5.4.8 (match_states_assign_state_mem, src/ObcToClight/SepInvariant.v:1465)
Given a program p, a name c, a memory tree me, a memory M, a global environment G, a temporary environment L, a location $(b_{self}, \delta_{self})$ and an assertion R such that $M \vDash \mathsf{s\text{-}rep}^{p}_{G}\, c\ me\ (b_{self}, \delta_{self}) * R$, then, for any state variable declaration $x^{\tau} \in cls.\mathbf{regs}$ and value v of type τ , there exists a memory M' such that:

1. $\mathsf{store}(\kappa^{\tau}, M, (b_{self}, \delta_{self} + \mathsf{field\text{-}offset}(G, x, \mathsf{fields}\ cls))\,, v) \doteq M'$, *and*
2. $M \vDash \mathsf{s\text{-}rep}^{p}_{G}\, c\ me(x \mapsto v)\ \ (b_{self}, \delta_{self}) * R$,

In practice, we lift this result over the match-state predicate, as it prevents tedious reasoning with the frame rule, and as assignments to state variables always occur in a context where match-state holds anyway.

The second type of assignment is to an output variable of m.

Lemma 5.4.9 (outputrep_assign_gt1_mem, src/ObcToClight/SepInvariant.v:1378)
Given a class cls, a method m, an environment ve, a memory M, a global environment G, a temporary environment L, an optional pair of a structure location and fields bco_{out} and an assertion R such that $M \vDash \mathsf{out\text{-}rep}_{G}\ cls\ m\ ve\ L\ bco_{out} * R$, if m has strictly more than one output, then there exist a block identifier b_{out} and fields φ_{out} such that $bco_{out} = \mathsf{Some}(b_{out}, \varphi_{out})$, and, for any output declaration $x^{\tau} \in m.\mathbf{out}$ and value v of type τ, there exist a memory M' such that:

1. $\mathsf{store}(\kappa^{\tau}, M, (b_{out}, \mathsf{field\text{-}offset}(G, x, \mathsf{fields}\ cls))\,, v) \doteq M'$, *and*
2. $M' \vDash \mathsf{out\text{-}rep}_{G}\ cls\ m\ ve\{x \mapsto v\}\ \ L\ bco_{out} * R$

Allocating operations

We also extend the "allocation rule" (lemma 5.4.1 on page 162) to show that the memory allocated at function entry corresponds with the output structure declarations used for sub-calls with strictly more than one output. The following lemma relies on several technical results that are not presented.

Lemma 5.4.10 (alloc_result, src/ObcToClight/SepInvariant.v:1687)
Given a memory M and a memory assertion R such that $M \vDash R$, then there exist an environment E and a memory M' such that, for any method m,
$\mathsf{alloc\text{-}vars}(G, M, \mathsf{mult\text{-}outs}\, m) \doteq (E, M')$ *and* $M' \vDash \mathsf{outs\text{-}rep}_{G}\, m\ E * R$.

Now we generalize this extended allocation rule to function entry specifications. The idea is to show that starting from an accordingly specified pre-call memory state, a function is entered with a memory state that satisfies match-state. The following result has two different cases, depending on the way outputs are handled. If the source Obc

method has zero or one output, then we only need a pre-call memory that satisfies only s-rep (the *self* pointer), otherwise we also need an assertion fs-rep (the *out* pointer). For this particular result, we must require that p is well-typed and successfully translated into a generated Clight program P, with associated global environment $G_P = \mathsf{globalenv}(P)$.

Lemma 5.4.11 (function_entry_match_states, src/ObcToClight/SepInvariant.v:1761)
Given a class cls named c, declared in p, having a method m named f; a list of values $\boldsymbol{v}$ *whose types are input types of m; a memory tree me and a function definition* F_d *generated from m, then:*

- *If m has zero or one output, then, for any memory M, memory assertion R and location l such that* $M \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ c\ me\ l * R$, *there exist an environment* E_f, *a temporary environment* L_f *and a memory* M_f *such that:*
  1. $\mathsf{function\text{-}entry}(F_d, (\mathsf{Vptr}\, l) \cdot \boldsymbol{v}, M) \doteq (E_f, L_f, M_f)$
  2. $M_f \vDash \mathsf{match\text{-}state}^{p}_{G_P}\ cls\ m\ \left(me, \varnothing\left\{m.\textbf{\textit{in}}^1 \mapsto \boldsymbol{v}\right\}\right)\ (E_f, L_f)\ l\ \mathit{None} * R$
- *If m has strictly more than one output, then, for any memory M, memory assertion R, location l, block identifier b and structure fields* $\boldsymbol{\varphi}$ *such that* $\mathsf{comp}(G_P, f_c) \doteq \boldsymbol{\varphi}$ *and* $M \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ c\ me\ l * \mathsf{fs\text{-}rep}_{G_P}\ \varnothing\ \boldsymbol{\varphi}\ b * R$, *there exist an environment* E_f, *a temporary environment* L_f *and a memory* M_f *such that:*
  1. $\mathsf{function\text{-}entry}(F_d, (\mathsf{Vptr}\, l) \cdot (\mathsf{Vptr}\, (b, 0)) \cdot \boldsymbol{v}, M) \doteq (E_f, L_f, M_f)$
  2. $M_f \vDash \mathsf{match\text{-}state}^{p}_{G_P}\ cls\ m\ \left(me, \varnothing\left\{m.\textbf{\textit{in}}^1 \mapsto \boldsymbol{v}\right\}\right)\ (E_f, L_f)\ l\ (\mathsf{Some}(b, \boldsymbol{\varphi})) * R$

Freeing operation
Finally, we extend the "free rule" (lemma 5.4.2 on page 162).

Lemma 5.4.12 (free_exists, src/ObcToClight/SepInvariant.v:1036)
Given a memory M, a global environment G, an assertion R and an environment E such that $M \vDash \mathsf{os\text{-}range}_G\ E * R$, *then there exists a memory* M' *such that* $\mathsf{free\text{-}env}(M, E) \doteq M'$ *and* $M' \vDash R$.

5.5 Correctness of the generation function

The proof of correctness of the generation of Clight code is the longest of the whole Vélus development. It is divided in four steps:

1. Static structural properties about the generated program upon success of the generation function, involving results about generated functions, structures or global variables.
2. State correspondence predicates defined as memory assertions.
3. Correctness at the level of execution of a method in a given class.
4. Correctness of the behavior of the whole generated program.

As for previous passes, the first step will not be discussed since the structural properties follow more or less directly from the generation function definition. The second step was presented in section 5.4.3. The last two steps are the subjects of the two following sections, where we assume that the translation of a well-typed Obc program p succeeds, generating a Clight program P with associated global environment $G_P = \mathsf{globalenv}(P)$.

5.5.1 Local correctness

The two following sections describe the intermediate results concerning generated expressions, assignments and function call evaluation. We assume in these two sections that p contains a class cls with a method m. Moreover, we fix me, ve, M, E, L, $l_{\mathsf{self}} = (b_{\mathsf{self}}, \delta_{\mathsf{self}})$, bco_{out} and R such that the following hypothesis holds.

Hypothesis 5.5.1

$$M \vDash \mathsf{match\text{-}state}^{p}_{G_P}\ \ cls\ m\ \ (me, ve)\ \ (E, L)\ \ l_{self}\ bco_{out} * R$$

The idea is to work in the context of Obc and Clight states that are in correspondence. We first show that the value of an Obc expression is correctly calculated by the Clight expression generated from it. We then extend this result for statements.

5.5.1.1 Expressions

Locations of indirect field accesses

The following two lemmas ensure the evaluation of fields accesses of *out* and *self* pointers to memory locations.

The first one below guarantees that whenever out-rep holds, then for any output variable x of the Obc method, the Clight expression $\overline{\mathsf{out}}\text{->}x$ evaluates to a valid memory location. The states bound by hypothesis 5.5.1 are not used here because of assignments to output fields that can occur after a function call. Indeed, in this particular case, as we will see later, the generic predicate match-state does not hold.

Lemma 5.5.1 (evall_out_field, src/ObcToClight/Correctness.v:230)
Given an environment ve_1, a temporary environment L_1, a memory M_1 and a memory assertion R_1 such that $M_1 \vDash \mathsf{out\text{-}rep}_{G_P}\ cls\ m\ ve_1\ L_1\ bco_{out} * R_1$, assume that m has strictly more than one output, then there exist a block identifier b_{out} and structure fields φ_{out} such that $bco_{out} = \mathsf{Some}\,(b_{out}, \varphi_{out})$, and, for any output variable $x^{\tau} \in m.\mathbf{out}$, $\mathsf{temp\text{-}ind\text{-}field\text{-}loc}(G_P, L_1, \overline{out}, x) \doteq (b_{out}, \mathsf{field\text{-}offset}(G_P, x, \varphi_{out}))$.

The second lemma below guarantees that for any Obc state variable x in the current method, the Clight expression $\overline{\mathsf{self}}\text{->}x$ evaluates to a valid memory location under hypothesis 5.5.1.

Lemma 5.5.2 (evall_self_field, src/ObcToClight/Correctness.v:336)
For any state variable $x^{\tau} \in cls.\mathbf{regs}$,
$\mathsf{temp\text{-}ind\text{-}field\text{-}loc}\Big(G_P, L, \overline{self}, x\Big) \doteq (b_{self}, \delta_{self} + \mathsf{field\text{-}offset}(G_P, x, \mathsf{fields}\ cls))$.

Local variables

The following results relate variable values in Obc to the evaluation of these variables in Clight.

Lemma 5.5.3 (eval_out_field, src/ObcToClight/Correctness.v:252)
Assume that m has strictly more than one output, then, for any output variable $x^\tau \in m.\textbf{\textit{out}}$, $G_P, E, L \vdash \overline{out}\text{->}x, M \Rightarrow ve(x)$.

Corollary 5.5.3.1 (eval_var, src/ObcToClight/Correctness.v:297)
For any declaration x^τ that belongs to the input, output or local variable declarations of m, $G_P, E, L \vdash \mathsf{gen\text{-}var}\ x^\tau, M \Rightarrow ve(x)$.

State variables

The following lemma guarantees that for any Obc state variables x with value in me, the Clight expression $\overline{\mathsf{self}}\text{->}x$ evaluates to the same value.

Lemma 5.5.4 (eval_self_field, src/ObcToClight/Correctness.v:358)
For any state variable declaration $x^\tau \in cls.\textbf{\textit{regs}}$, $G_P, E, L \vdash \overline{self}\text{->}x, M \Rightarrow me(x)$.

General expressions

Now that translated variables are shown to evaluate correctly, we can proceed to state the general theorem about the correct evaluation of the translation of an Obc expression.

Theorem 5.5.5 (expr_correct, src/ObcToClight/Correctness.v:406)
Given a well-typed Obc expression e and a value v such that $me, ve \vdash e \Downarrow \lfloor v \rfloor$, then

$$G_P, E, L \vdash \mathsf{gen\text{-}exp}\ e, M \Rightarrow v$$

We generalize this result to the evaluation of a list of translated expressions as function parameters with type casts.

Corollary 5.5.5.1 (exprs_correct, src/ObcToClight/Correctness.v:446)
Given a list of well-typed expressions $\boldsymbol{e}$ and a list of values $\boldsymbol{v}$ such that $me, ve \vdash \boldsymbol{e} \Downarrow \lfloor \boldsymbol{v} \rfloor$, then

$$G_P, E, L \vdash (\mathsf{type}\ \boldsymbol{e})\ (\mathsf{gen\text{-}exp}\ \boldsymbol{e}), M \Rightarrow \boldsymbol{v}$$

5.5.1.2 Statements

At the end of this section, we present the correspondence theorem for statements generated from Obc statements. We work towards this result by first presenting the three main base cases needed to prove it by induction: (1) assignments to local variables, (2) transfers of outputs after a function call, and (3) function calls themselves.

Local variable assignments

We show a key invariant preservation result for assignments generated by the assign function (see figure 5.4b on page 147). We do not use the whole match-state assertion, but rather a stripped-down version mentioning only out-rep for assignments to output variables, and vars-rep for assignments to unique output or other local variables. Consequently, we state the following result in a different context than that of hypothesis 5.5.1. If the defining expression evaluates to a value v in Clight, the lemma below states that the generated assignment to x evaluates to give a new state that corresponds to a suitably updated Obc environment.

Lemma 5.5.6 (exec_assign, src/ObcToClight/Correctness.v:480)
Given an environment ve_1, a memory M_1, a temporary environment L_1, a memory assertion R_1, a variable declaration x^τ of method m, a Clight expression a of type τ and a value v of type τ such that:

1. $M_1 \vDash \mathsf{out\text{-}rep}_{G_P}\ cls\ m\ ve_1\ L_1\ bco_{out} * \mathsf{vars\text{-}rep}\ cls\ ve_1\ L_1 * R_1$, *and*
2. $G_P, E, L_1 \vdash a, M_1 \Rightarrow v$,

then there exist a memory M' and a temporary environment L' such that:

1. $G_P, E, L_1 \vdash \mathsf{assign}(x, a), M_1 \overset{\varepsilon}{\Longrightarrow} \mathit{Normal}, L', M'$
2. $M' \vDash \mathsf{out\text{-}rep}_{G_P}\ cls\ m\ ve_1\{x \mapsto v\}\ L'\ bco_{out} * \mathsf{vars\text{-}rep}\ cls\ ve_1\{x \mapsto v\}\ L' * R_1$

For this lemma, the vars-rep component of the invariant is not used to prove the semantic evaluation of the statement; we only use it to show that it is preserved by evaluation of the statement.

Transfers of output variables after a function call

When a method named f' of a sub-instance i has strictly more than one output, the generation function adds a sequence of special assignments after the function call to copy the values of the output structure to the appropriate locations. As those statements do not appear in the source Obc program, we need to adapt our invariants. Looking at the definition 5.4.6 on page 168 of os-rep that appears in the outs-rep part of match-state, recall that the idea is to assert that for each pair (i, f') there exists a locally declared structure whose address is held by a local variable $i_{f'}$ and whose content is not specified. We express this assertion using fs-rep with an empty environment. After the function call, this environment should associate the output parameters of the Obc method to the values returned by that method, the fs-rep assertion then guarantees that the fields of the Clight output structure contain the same values. After a function call with multiple outputs, match-state does not hold since the output variables in Obc are updated directly, but the corresponding variables in Clight are not. We re-establish match-state by reasoning that the sequence of Clight assignments after the function call correctly transfers the values defined by fs-rep before we weaken the fs-rep assertion by *forgetting* its content.

For the following two lemmas, assume that there exist:

1. a callee method m' with name f' and strictly more than one output, of a class with name c' that is declared in p,

2. an Obc environment $ve_{f'}$ representing the return environment of the method m' after the call,

3. a Clight variable $i_{f'}$ and a block identifier b such that $E(i_{f'}) = (b, \texttt{struct}\ f'_{c'})$, that is, $i_{f'}$ holds the address of the corresponding output structure, and

4. structure fields $\boldsymbol{\varphi}$ of the output structure such that $\mathsf{comp}(G_P, f'_{c'}) \doteq \boldsymbol{\varphi}$.

We begin by proving an intermediate result that states the correctness of the evaluation of a field access to the output structure when the content of the structure is known.

Lemma 5.5.7 (eval_inst_field, src/ObcToClight/Correctness.v:592)
Given a memory M_1, a temporary environment L_1 such that $M_1 \vDash \mathsf{fs\text{-}rep}_{G_P}\ ve_{f'}\ \boldsymbol{\varphi}\ b$, then, for any output declaration $x^\tau \in m'.\boldsymbol{out}$, $G_P, E, L_1 \vdash i_{f'}\,.\,x, M_1 \Rightarrow ve_{f'}(x)$.

Now we can show that the sequence of assignments after a function call preserves the specialized invariant.

Lemma 5.5.8 (exec_funcall_assign, src/ObcToClight/Correctness.v:669)
Given a memory M_1, a memory assertion R_1, a list of values $\boldsymbol{v}$ that are well-typed relative to the output types of m' such that:

1. $ve_{f'}(\boldsymbol{y}) = \boldsymbol{v}$, *where* $m'.\boldsymbol{out} = \boldsymbol{y^\tau}$ *and*

2. $$\begin{aligned} M_1 \vDash\ & \mathsf{fs\text{-}rep}_{G_P}\ ve_{f'}\ \boldsymbol{\varphi}\ b \\ & * \mathsf{out\text{-}rep}_{G_P}\ cls\ m\ ve\ L\ bco_{out} \\ & * \mathsf{vars\text{-}rep}\ m\ ve\ L \\ & * R_1 \end{aligned}$$

then, for any list $\boldsymbol{x}$ of local variables of m, there exist a memory M' and a temporary environment L' such that:

1. $G_P, E, L \vdash \mathsf{assign}\left(\boldsymbol{x}, i_{f'}\,.\,\boldsymbol{y}\right), M_1 \overset{\varepsilon}{\Longrightarrow} \mathit{Normal}, L', M'$

2. $$\begin{aligned} M' \vDash\ & \mathsf{fs\text{-}rep}_{G_P}\ ve_{f'}\ \boldsymbol{\varphi}\ b \\ & * \mathsf{out\text{-}rep}_{G_P}\ cls\ m\ ve\{\boldsymbol{x} \mapsto \boldsymbol{v}\}\ L'\ bco_{out} \\ & * \mathsf{vars\text{-}rep}\ m\ ve\{\boldsymbol{x} \mapsto \boldsymbol{v}\}\ L' \\ & * R_1 \end{aligned}$$

Function calls

The main challenge in reasoning about the correctness of function calls is to formulate a lemma that can be proved by mutual induction over the Obc semantics. In particular, as we have to handle three distinct cases relative to the number of outputs of the method being called, the induction hypothesis is a bit involved. The following definition is not the induction hypothesis *per se*, but rather a specification on the execution of a call. Indeed, it does not mention the semantics of Obc function calls but directly states the expected execution of a call and the expected state after the call, given a state on function entry (as given by lemma 5.4.11 on page 171).

Definition 5.5.1 (call_spec, src/ObcToClight/Correctness.v:706)
Given a class cls, a method m, lists of values $\boldsymbol{v}$ and $\boldsymbol{w}$ and memory trees me and me', the predicate $\mathsf{call\text{-}spec}(cls, m, \boldsymbol{v}, \boldsymbol{w}, me, me')$ *is defined as follows.*
For any location l, memory M and memory assertion R, there exists a function definition F_d generated from m, and:

- *If m has no output and $M \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ cls.\mathit{name}\ me\ l * R$, then there exists a memory M' such that:*
  1. $G_P \vdash F_d\left((\mathsf{Vptr}\, l) \cdot \boldsymbol{v}\right), M \overset{\varepsilon}{\Longrightarrow} \mathit{Vundef}, M'$
  2. $M' \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ cls.\mathit{name}\ me'\ l * R$
- *If m has only one output and $M \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ cls.\mathit{name}\ me\ l * R$, then there exist a memory M' and a value r such that:*
  1. $G_P \vdash F_d\left((\mathsf{Vptr}\, l) \cdot \boldsymbol{v}\right), M \overset{\varepsilon}{\Longrightarrow} r, M'$
  2. $\boldsymbol{w} = [r]$
  3. $M' \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ cls.\mathit{name}\ me'\ l * R$
- *If m has strictly more than one output, and given a block identifier b and structure fields $\boldsymbol{\varphi}$ such that $\mathsf{comp}(G_P, f_c) \doteq \boldsymbol{\varphi}$ where f and c are the names of m and cls, respectively, and $M \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ cls.\mathit{name}\ me\ l * \mathsf{fs\text{-}rep}_{G_P}\ \varnothing\ \boldsymbol{\varphi}\ b * R$, then there exist a memory M' and an environment ve_f such that:*
  1. $G_P \vdash F_d\left((\mathsf{Vptr}\, l) \cdot (\mathsf{Vptr}\, (b, 0)) \cdot \boldsymbol{v}\right), M \overset{\varepsilon}{\Longrightarrow} \mathit{Vundef}, M'$
  2. $ve_f(\boldsymbol{y}) = \boldsymbol{w}$ *where* $m.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau}}$
  3. $M' \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ cls.\mathit{name}\ me'\ l * \mathsf{fs\text{-}rep}_{G_P}\ ve_f\ \boldsymbol{\varphi}\ b * R$

Let us explain each case. The first two are similar: when the method has zero or one output, we do not use an output structure, therefore we only need s-rep. The idea is to recursively select the sub-structure in the memory which is in correspondence with the sub-tree me, and to assert that the invariant is preserved by the call, with updated M' and me'. In both cases, the first value passed to the call is a pointer towards the sub-structure. In the second case, the function does return a value that must equal the single Obc return value. In the third case, when m has strictly more than one output, in addition to s-rep, we must ensure that memory for the output structure is allocated—even

if its contents are unspecified. Then, after the call, the state representation is updated, and the output structure is constrained to contain the Obc return values $\boldsymbol{w}$.

Now, recall the match-state context given by hypothesis 5.5.1 and assume additionally that there exists a callee method m' named f' of a class cls' named c' that is declared in p, and that $i^{c'}$ is the corresponding sub-instance declaration of cls. We express the correctness result for the whole translation of a method call (see the funcall function in figure 5.4b on page 147), using call-spec as the induction hypothesis.

Lemma 5.5.9 (exec_binded_funcall, src/ObcToClight/Correctness.v:748)
Given a list $\boldsymbol{x}$ of local variables of m with the same types as the outputs of m', a list of values $\boldsymbol{v}$, a list of values $\boldsymbol{w}$ well-typed relative to the outputs of m', a memory tree me'_i and a list of Clight expressions $\boldsymbol{a}$ with the same types as the inputs of m', such that $G_P, E, L \vdash (\mathsf{type}\ \boldsymbol{a})\ (\boldsymbol{a}), M \Rightarrow \boldsymbol{v}$ and $\mathsf{call\text{-}spec}(cls', m', \boldsymbol{v}, \boldsymbol{w}, (\mathsf{sub}\ i\ me), me'_i)$ holds, then, under some extra well-typing and well-formedness constraints, there exist a memory M' and a temporary environment L' such that:

1. $G_P, E, L \vdash \mathsf{funcall}(\boldsymbol{x}, c', i, f', \boldsymbol{a}), M \overset{\varepsilon}{\Longrightarrow} \mathit{Normal}, L', M'$
2. $M' \vDash \mathsf{match\text{-}state}^{p}_{G_P}\ cls\ m\ (me[i \mapsto me'_i], ve\{\boldsymbol{x} \mapsto \boldsymbol{w}\})\ (E, L')\ l_{\mathit{self}}\ bco_{\mathit{out}} * R$

5.5.1.3 General statements

Now that all key intermediate results have been presented (expressions, assignments and function calls evaluation), we can show the main correctness result. In other correctness proofs, for instance from NLustre to Stc and from Stc to Obc, we proceed in two steps: (1) we show the correctness of the evaluation of statements under the hypothesis that its calls are correct, then (2) we show the correctness of calls by induction on the program using the correctness result on statements. While sometimes a bit cumbersome, this scheme usually works well. But not for the generation of Clight, because the translation function is not recursive and is monadic. While in Lustre, NLustre, Stc and Obc, a program is a bare list of nodes, systems and classes, respectively, the Clight generation function produces a list of function and structure declarations plus a main entry point, so it is not possible to reason directly by induction on the program being translated. Still, we adapt the same idea: using the well-formedness predicate that specifies the order of declarations in the translated Obc program p, we apply a mutual induction scheme on both the semantics of statements and of method calls on *suffixes* of p. In the development, the Coq mutual induction scheme that we use requires that both parts are stated as a single conjunction. We fix the following hypothesis (both statements are satisfied by any Obc program processed by the argument initialization pass described in section 5.1).

Hypothesis 5.5.2
Assume that p satisfies the NoNakedVars *predicate and that any Obc method call on defined values only returns defined values.*

Theorem 5.5.10 (mutual_correctness, src/ObcToClight/Correctness.v:1011)

- *Given a program p' that is a suffix of p, a class cls declared in p that has a method m, a statement s that occurs in the body of m, environments ve and ve', memory trees me and me', a memory M, a memory assertion R, an environment E, a temporary environment L, a location l_{self} and an optional pair bco_{out} of a block identifier and structure fields such that:*

  1. $p', me, ve \vdash s \Downarrow (me', ve')$
  2. $M \vDash \mathsf{match\text{-}state}^{p}_{G_P}\ cls\ m\ (me, ve)\ (E, L)\ l_{self}\ bco_{out} * R$

  Then there exist a memory M' and a temporary environment L' such that:

  1. $G_P, E, L \vdash \mathsf{gen\text{-}stmt}^{p,m.\mathbf{out}}\ s, M \overset{\varepsilon}{\Longrightarrow} \mathit{Normal}, L', M'$
  2. $M' \vDash \mathsf{match\text{-}state}^{p}_{G_P}\ cls\ m\ (me', ve')\ (E, L')\ l_{self}\ bco_{out} * R$

- *Given a program p' that is a suffix of p, a class cls of name c declared in p that has a method m of name f, a list of values $\boldsymbol{v}$ that are well-typed relative to the input types of m, a list of values $\boldsymbol{w}$, and memory trees me and me' such that $p', me \vdash c.f\ (\lfloor \boldsymbol{v} \rfloor) \overset{\lfloor \boldsymbol{w} \rfloor}{\Downarrow} me'$, then $\mathsf{call\text{-}spec}(cls, m, \boldsymbol{v}, \boldsymbol{w}, me, me')$ holds.*

We then directly deduce the main correctness result.

Corollary 5.5.10.1 (stmt_call_correctness, src/ObcToClight/Correctness.v:1267)
Given a class cls named c, declared in p, and that has a method m named f, a list of values $\boldsymbol{v}$ that are well-typed relative to the input of m, a list of values $\boldsymbol{w}$, and memory trees me and me' such that

$$p, me \vdash c.f\ (\lfloor \boldsymbol{v} \rfloor) \overset{\lfloor \boldsymbol{w} \rfloor}{\Downarrow} me'$$

then

$$\mathsf{call\text{-}spec}(cls, m, \boldsymbol{v}, \boldsymbol{w}, me, me')$$

5.5.2 Program correctness

In this section, we present the specifications and proofs for the generated *main* function that resets the global program state and then repeatedly reads input values, passes them to the *step* function, and writes the resulting output values. By virtue of the generation function, we know that there exists a class c_{main} named main-node in p, and that this class has two methods s_{main} and r_{main} named step and reset respectively.

5.5.2.1 Volatile operations

The volatile reads of inputs and writes of outputs are the only operations in the generated program that produce events. We start by defining the result traces that correspond to the execution of the read and write statements described in figure 5.5 on page 149.

Definition 5.5.2 (load_events, src/Traces.v:70)
Given a list $\boldsymbol{ins} = {x_1}^{\tau_1^x} \cdots {x_i}^{\tau_i^x}$, *and list of values* $\boldsymbol{v} = v_1 \cdots v_i$,

$$\mathsf{read\text{-}trace}\,\boldsymbol{v}\ \boldsymbol{ins} \triangleq \mathcal{E}_{\mathit{vload}}\left(\kappa^{\tau_1^x}, x_1, 0, \mathsf{event\text{-}val}\,v_1\right) \cdots \mathcal{E}_{\mathit{vload}}\left(\kappa^{\tau_i^x}, x_i, 0, \mathsf{event\text{-}val}\,v_i\right)$$

Definition 5.5.3 (store_events, src/Traces.v:71)
Given a list $\boldsymbol{outs} = {y_1}^{\tau_1^y} \cdots {y_j}^{\tau_j^y}$, *and a list of values* $\boldsymbol{w} = w_1 \cdots w_j$,

$$\mathsf{write\text{-}trace}\,\boldsymbol{w}\ \boldsymbol{outs} \triangleq \mathcal{E}_{\mathit{vstore}}\left(\kappa^{\tau_1^y}, y_1, 0, \mathsf{event\text{-}val}\,w_1\right) \cdots \mathcal{E}_{\mathit{vstore}}\left(\kappa^{\tau_j^y}, y_j, 0, \mathsf{event\text{-}val}\,w_j\right)$$

Each function maps values and variable declarations to a list of events. The event-val function translates a standard value into an event value, such that $\mathsf{event\text{-}val}\,v \overset{\kappa}{\underset{G}{\simeq}} v$ holds for any G—we do not need the global environment since we do not deal with pointer values here—provided that v is well-typed relative to the type information of κ. Recall (see section 5.3.1.1) that we write $w \overset{\kappa}{\underset{G}{\simeq}} v$ to denote the correspondence between the standard value v and the event value w.

We show that the execution of the read and write statements yields the corresponding traces. We begin with the volatile loads, which produce the expected trace whatever the memory state.

Lemma 5.5.11 (exec_read, src/ObcToClight/Correctness.v:1391)
Given a list of values $\boldsymbol{v}$ *that are well-typed relative to the inputs of* s_{main}, *an environment* E, *a temporary environment* L *and a memory* M, *then*
$G_P, E, L \vdash \mathsf{read}, M \xrightarrow{\mathsf{read\text{-}trace}\,\boldsymbol{v}\ s_{\mathit{main}}.\boldsymbol{in}} \mathit{Normal}, L\{\boldsymbol{x} \mapsto \boldsymbol{v}\}, M$, *where* $s_{\mathit{main}}.\boldsymbol{in} = \boldsymbol{x}^{\boldsymbol{\tau}}$.

The volatile writes occur after the call to the main *step* function and their correctness relies on the program state at this point. We have to distinguish the cases where s_{main} has zero, one, or strictly more than one output. In the last case, recall that a structure type named $\mathsf{step}_{\mathsf{main\text{-}node}}$ is globally declared and that the *main* function declares a local variable $\mathsf{out}_{\mathsf{step}}$ with this type.

Lemma 5.5.12 (exec_write, src/ObcToClight/Correctness.v:1513)
Given a list of values $\boldsymbol{w}$ *that are well-typed relative to the outputs of* s_{main}, *an environment* E, *a temporary environment* L *and a memory* M *such that:*

- *If* s_{main} *has one output* y^{τ}, *then there exists a single value* w *such that* $\boldsymbol{w} = [w]$ *and* $L(y) = w$.
- *If* s_{main} *has strictly more than one output then there exist an environment* ve, *a block identifier* b *and structure fields* $\boldsymbol{\varphi}$ *such that* $\mathsf{comp}(G_P, \mathit{step}_{\mathit{main\text{-}node}}) \doteq \boldsymbol{\varphi}$, $E(\mathit{out}_{\mathit{step}}) = (b, \mathit{struct}\ \mathit{step}_{\mathit{main\text{-}node}})$, $M \vDash \mathsf{fs\text{-}rep}_{G_P}\ ve\ \boldsymbol{\varphi}\ b$ *and* $ve(\boldsymbol{y}) = \boldsymbol{w}$ *where* $s_{\mathit{main}}.\boldsymbol{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}}$.

Then $G_P, E, L \vdash \mathsf{write}, M \xrightarrow{\mathsf{write\text{-}trace}\,\boldsymbol{w}\ s_{\mathit{main}}.\boldsymbol{out}} \mathit{Normal}, L, M$.

$$\frac{E \vdash F_{\mathsf{step}}\,(xs)\,, M' \overset{n+1}{\circlearrowleft} ys \qquad G_P \vdash F_{\mathsf{step}}\,(\mathsf{Vptr}\,(b_{\mathsf{self}}, 0) \cdot xs_n)\,, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Vundef}, M'}{E \vdash F_{\mathsf{step}}\,(xs)\,, M \overset{n}{\circlearrowleft} ys} \quad \text{if } \|s_{\mathsf{main}}.\mathbf{out}\| = 0$$

$$\frac{E \vdash F_{\mathsf{step}}\,(xs)\,, M' \overset{n+1}{\circlearrowleft} ys \qquad G_P \vdash F_{\mathsf{step}}\,(\mathsf{Vptr}\,(b_{\mathsf{self}}, 0) \cdot xs_n)\,, M \overset{\varepsilon}{\Longrightarrow} w, M' \quad ys_n = [w]}{E \vdash F_{\mathsf{step}}\,(xs)\,, M \overset{n}{\circlearrowleft} ys} \quad \text{if } \|s_{\mathsf{main}}.\mathbf{out}\| = 1$$

$$\frac{\begin{array}{c} E \vdash F_{\mathsf{step}}\,(xs)\,, M' \overset{n+1}{\circlearrowleft} ys \quad \mathsf{symbol}(G_P, \mathsf{step}_{\mathsf{main\text{-}node}}) \doteq \boldsymbol{\varphi} \\ E(\mathsf{out}_{\mathsf{step}}) = (b, \mathtt{struct}\ \mathsf{step}_{\mathsf{main\text{-}node}}) \\ G_P \vdash F_{\mathsf{step}}\,(\mathsf{Vptr}\,(b_{\mathsf{self}}, 0) \cdot \mathsf{Vptr}\,(b, 0) \cdot xs_n)\,, M \overset{\varepsilon}{\Longrightarrow} \mathsf{Vundef}, M' \\ M' \vDash \mathsf{fs\text{-}rep}_{G_P}\ ve\ \boldsymbol{\varphi}\ b \quad ve(\boldsymbol{y}) = \boldsymbol{w} \end{array}}{E \vdash F_{\mathsf{step}}\,(xs)\,, M \overset{n}{\circlearrowleft} ys} \quad \begin{array}{l} \text{if } \|s_{\mathsf{main}}.\mathbf{out}\| > 1, \\ \text{where } s_{\mathsf{main}}.\mathbf{out} = \boldsymbol{y}^{\tau} \end{array}$$

Figure 5.10: Big-step looping predicate

5.5.2.2 Evaluation of the main loop

For the correctness of the body of the *main* function, we introduce a coinductive predicate that specifies the looping execution of the main *step* function. We use the same style of looping semantic judgments in Stc (figure 3.6c on page 83) and Obc (figure 4.2d on page 103). We use this predicate for the intermediate correctness results as it can be directly related to the looping execution semantics of Obc.

Assume, as a property of the generation function that there exists a Clight function F_{step}, named $\mathsf{step}_{\mathsf{main\text{-}node}}$, that corresponds to the translation of the method s_{main}. Similarly, assume that the variable self is globally declared, that is, there exists a block identifier b_{self} such that $\mathsf{symbol}(G_P, \mathsf{self}) \doteq b_{\mathsf{self}}$. The predicate is presented under these assumptions in figure 5.10. We write $E \vdash F_{\mathsf{step}}\,(xs)\,, M \overset{n}{\circlearrowleft} ys$ to mean that, in the context of the environment E, the function F_{step} is executed repeatedly, taking successive inputs from the stream of lists of values xs from instant n onwards and constraining the stream ys of lists of outputs values from instant n onwards. Depending on the number of outputs of s_{main}, the predicate specifies, the memory state after each call.

We show that this predicate simulates one step of the main loop body, that we designate main-loop-body, described in figure 5.5 on page 149, with the according trace obtained from the concatenation of the trace of the loads with the trace of the stores. In the following, we fix xs and ys as two streams of lists of values that are well-typed (at each instant) relative to the input and output types of s_{main} respectively.

Lemma 5.5.13 (exec_body, src/ObcToClight/Correctness.v:1700)
Given an integer n, an environment E, a memory M^n and a temporary environment L^n such that $E \vdash F_{\mathit{step}}\,(xs)\,, M^n \overset{n}{\circlearrowleft} ys$, then there exist a temporary environment L^{n+1} and a memory M^{n+1} such that:

1. $G_P, E, L^n \vdash \mathsf{main\text{-}loop\text{-}body}, M^n \overset{t^n}{\Longrightarrow} \mathit{Normal}, L^{n+1}, M^{n+1}$
   where $t^n = \mathsf{read\text{-}trace}\; xs_n\; s_{\mathit{main}}.\boldsymbol{\mathit{in}} + \mathsf{write\text{-}trace}\; ys_n\; s_{\mathit{main}}.\boldsymbol{\mathit{out}}$
2. $E \vdash F_{\mathit{step}}\,(xs)\,, M^{n+1} \overset{n+1}{\circlearrowleft} ys$

We must use the small-step semantics of Clight (presented in section 5.3.1.1), to show that generated programs diverge reactively. We coinductively define a function that builds an infinite trace from input and output streams of lists of values and input and output variable declarations.

Definition 5.5.4 (mk_trace, src/Traces.v:120)
Given a natural integer n, two streams of lists of values xs and ys and two lists of variable-type declarations $\boldsymbol{\mathit{ins}}$ and $\boldsymbol{\mathit{outs}}$,

$$\begin{aligned}\mathsf{trace\text{-}step}^n\; xs\; ys\; \boldsymbol{\mathit{ins}}\; \boldsymbol{\mathit{outs}} \triangleq\; & \mathsf{read\text{-}trace}\; xs_n\; \boldsymbol{\mathit{ins}} + \mathsf{write\text{-}trace}\; ys_n\; \boldsymbol{\mathit{outs}} \\ & +\!\!+ \mathsf{trace\text{-}step}^{n+1}\; xs\; ys\; \boldsymbol{\mathit{ins}}\; \boldsymbol{\mathit{outs}}\end{aligned}$$

provided that $\mathsf{read\text{-}trace}\; xs_n\; \boldsymbol{\mathit{ins}} + \mathsf{write\text{-}trace}\; ys_n\; \boldsymbol{\mathit{outs}} \neq \varepsilon$.

The condition on read-trace and write-trace ensures that the infinite trace is *productive*. To fulfill this requirement, we need to prove that $\forall n,\ \|xs_n\| = \|\boldsymbol{\mathit{ins}}\|$, $\forall n,\ \|ys_n\| = \|\boldsymbol{\mathit{outs}}\|$, and that $\boldsymbol{\mathit{ins}}$ and $\boldsymbol{\mathit{outs}}$ are not simultaneously empty.

We show that the looping predicate represents the infinite execution of the main loop, using the small-step semantics. This is important for the final proof.

Lemma 5.5.14 (reactive_loop, src/ObcToClight/Correctness.v:2006)
Given a natural number n, an environment E, and a memory M^n such that $E \vdash F_{\mathit{step}}\,(xs)\,, M^n \overset{n}{\circlearrowleft} ys$, then for any function F_d, temporary environment L^n and continuation k,

$$G_P \vdash \mathcal{S}(F_d, \mathsf{main\text{-}loop}, k, E, L^n, M^n) \xrightarrow{T^n} \infty$$

where $T^n = \mathsf{trace\text{-}step}^n\; xs\; ys\; (s_{\mathit{main}}.\boldsymbol{\mathit{in}})\; (s_{\mathit{main}}.\boldsymbol{\mathit{out}})$.

5.5.2.3 Correctness of the *main* entry point body

In addition to the previous assumptions on F_{step}, xs, ys, self and b_{step}, we will assume that F_{reset} is the Clight generated function named $\mathsf{reset}_{\mathsf{main\text{-}node}}$ that corresponds to the r_{main} method. We also fix a context where the main *step* method is executed repeatedly on the streams xs and ys and on a memory tree me_0 that is initialized by a unique call to the main *reset* method.

Hypothesis 5.5.3

1. $p, \varnothing \vdash \mathit{main\text{-}node.reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me_0$
2. $p, me_0 \vdash \mathit{main\text{-}node.step}\,(\lfloor xs \rfloor) \overset{0}{\circlearrowleft} \lfloor ys \rfloor$

Finally, we also assume a Clight environment E_{main}, a temporary environment L_{main} and a memory M_{main} that represent the state at the entry of the *main* entry point of the Clight program. We specify this state according to the number of outputs of the s_{main} method. If it has zero or one output then we only assert s-rep for the global self variable, and if it has strictly more than one output we add a fs-rep assertion that corresponds to the locally declared structure named $\mathsf{out}_{\mathsf{step}}$.

Hypothesis 5.5.4

- *If s_{main} has zero or one output, then* $M_{main} \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ \mathit{main\text{-}node}\ \varnothing\ (b_{self}, 0)$.
- *Otherwise, there exist a block identifier b and structure fields φ such that:*
  1. $\mathsf{comp}(G_P, \mathit{step}_{\mathit{main\text{-}node}}) \doteq \varphi$
  2. $E(\mathit{out}_{\mathit{step}}) = (b, \texttt{struct}\ \mathit{step}_{\mathit{main\text{-}node}})$
  3. $M_{main} \vDash \mathsf{s\text{-}rep}^{p}_{G_P}\ \mathit{main\text{-}node}\ \varnothing\ (b_{self}, 0) * \mathsf{fs\text{-}rep}_{G_P}\ \varnothing\ \varphi\ b$

We begin by showing that the main *reset* call in the body of the *main* entry point function, as shown in figure 5.5 on page 149, evaluates correctly and initiates the looping execution of the main *step* function.

Lemma 5.5.15 (exec_reset, src/ObcToClight/Correctness.v:1948)
There exists a memory M^0 such that:

1. $G_P, E_{main}, L_{main} \vdash \mathit{reset}_{\mathit{main\text{-}node}}\,(\&\mathit{self}), M_{main} \overset{\varepsilon}{\Longrightarrow} \mathit{Normal}, L_{main}, M^0$
2. $E \vdash F_{\mathit{step}}\,(xs)\,, M^0 \overset{0}{\circlearrowleft} ys$

5.5.2.4 Correctness of the generated program

We know, as a property of the generation function, that the *main* entry point of the generated program P exists, that is, there is a block identifier b_{main} and a function definition F_{main} such that $\mathsf{symbol}(G, P.\mathsf{main}) \doteq b_{\mathsf{main}}$ and $\mathsf{funct}(G, b) \doteq F_{\mathsf{main}}$. Using the semantics of Clight programs (figure 5.8d on page 157), to show that P diverges reactively producing an infinite trace T, we must show that there exists a memory M_{init} that satisfies the two following antecedents, where $\mathcal{C}_{\mathsf{main}} = \mathcal{C}(F_{\mathsf{main}}, \varepsilon, \texttt{stop}, M_{init})$:

1. $\mathsf{initial\text{-}state}(P, \mathcal{C}_{\mathsf{main}})$
2. $G_P \vdash \mathcal{C}_{\mathsf{main}} \xrightarrow{T} \infty$

To show the existence of M_{init}, we show the following intermediate result that additionally specifies the memory.

Lemma 5.5.16 (init_mem, src/ObcToClight/SepInvariant.v:1977)
There exists a memory M_{init} such that $\mathsf{initmem}(P) \doteq M_{init}$ *and* $M_{init} \vDash \mathsf{s\text{-}rep}_G^{G_P}\ p\ \mathit{main\text{-}node}\ \varnothing(b_{\mathit{self}}, 0)$.

As a direct consequence, we know that the witness M_{init} of lemma 5.5.16 satisfies the memory initialization predicate. For the sake of simplicity, we exhibit the existential witnesses progressively, whereas we cannot in the Coq proof.

Corollary 5.5.16.1 (initial_state_main, src/ObcToClight/Correctness.v:2051)
$\mathsf{initial\text{-}state}(P, \mathcal{C}_{\mathit{main}})$ *holds.*

We must still prove the reactive divergence of the call to the *main* entry point function, that is, $G_P \vdash \mathcal{C}_{\mathsf{main}} \xrightarrow{T} \infty$. To show the correctness of the trace, we want $T = T^0 = \mathsf{trace\text{-}step}^0\ xs\ ys\ s_{\mathsf{main}}.\mathbf{in}\ s_{\mathsf{main}}.\mathbf{out}$. The first step is the entry step into the actual *main* function.

Lemma 5.5.17 (main_entry, src/ObcToClight/Correctness.v:1340)
There exist an environment E_{main}, a temporary environment L_{main}, and a memory M_{main} such that $\mathsf{function\text{-}entry}(F_{\mathit{main}}, \varepsilon, M_{init}) \doteq (E_{\mathit{main}}, L_{\mathit{main}}, M_{\mathit{main}})$ *and they satisfy hypothesis 5.5.4.*

Finally, we show the correctness result for the whole generated program, in a context that satisfies hypotheses 5.5.2 and 5.5.3 on page 177 and on the preceding page.

Theorem 5.5.18 (correctness, src/ObcToClight/Correctness.v:2105)
Given xs and ys, two streams of lists of values that are well-typed (at each instant) relative to the input and output types of s_{main}, respectively, and a memory tree me^0 such that:

1. $p, \varnothing \vdash \mathit{main\text{-}node.reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me_0$
2. $p, me_0 \vdash \mathit{main\text{-}node.step}\,(\lfloor xs \rfloor) \overset{0}{\circledcirc} \lfloor ys \rfloor$

Then we have

$$P \Downarrow \mathsf{Reacts}\Big(\mathsf{trace\text{-}step}^0\ xs\ ys\ s_{\mathit{main}}.\mathbf{\mathit{in}}\ s_{\mathit{main}}.\mathbf{\mathit{out}}\Big)$$

Chapter 6

Conclusion

The previous chapters have presented a sequence of languages, semantic models, compilation passes, and correctness results. We now describe how they are combined to give an end-to-end correctness proof (section 6.1), before summarizing the presented work (section 6.2) and outlining future possibilities (section 6.3).

6.1 End-to-end correctness

6.1.1 The compilation function

The overall compilation function combines the individual transformations described in previous chapters and integrates a CompCert function that compiles Clight into assembly code. The elaboration pass, the scheduling validator, and the CompCert function may all fail in one way or another. This possibility is treated formally by an error monad (res, CompCert/common/Errors.v:46). A result, of type $\mathsf{res}\,A$, is either $\mathsf{OK}\,r$ where r is of type A, or an error Error—actually the Error constructor carries an error message that we omit for simplicity. We use the following notations, inspired from the programming language Haskell, for the usual monadic binding, mapping and sequencing operators. Their roles is essentially to propagate Error. The binding operator $\gg\!= : \mathsf{res}\,A \to (A \to \mathsf{res}\,B) \to \mathsf{res}\,B$ takes a result and applies a partial function on it. The mapping operator $<\$> : \mathsf{res}\,A \to (A \to B)$ applies a total function on a result. The sequencing operator $\gg : \mathsf{res}\,A \to \mathsf{res}\,B \to \mathsf{res}\,B$ sequentializes partial operations. It is defined with the $\gg\!=$ operator.

$$\mathsf{OK}\,r \gg\!= f \triangleq f\ r \qquad \mathsf{OK}\,r <\$> f \triangleq \mathsf{OK}\,(f\ r) \qquad r_1 \gg r_2 \triangleq r_1 \gg\!= \lambda\text{-}.\, r_2$$

$$\mathsf{Error} \gg\!= f \triangleq \mathsf{Error} \qquad \mathsf{Error} <\$> f \triangleq \mathsf{Error}$$

We combine the source transformations described in the previous chapters and recalled in figure 6.1 into a monadic compilation function. The compilation function takes a list of parsed raw declarations and the name of the main node, and returns the elaborated NLustre program and the generated assembly program, in the error monad.

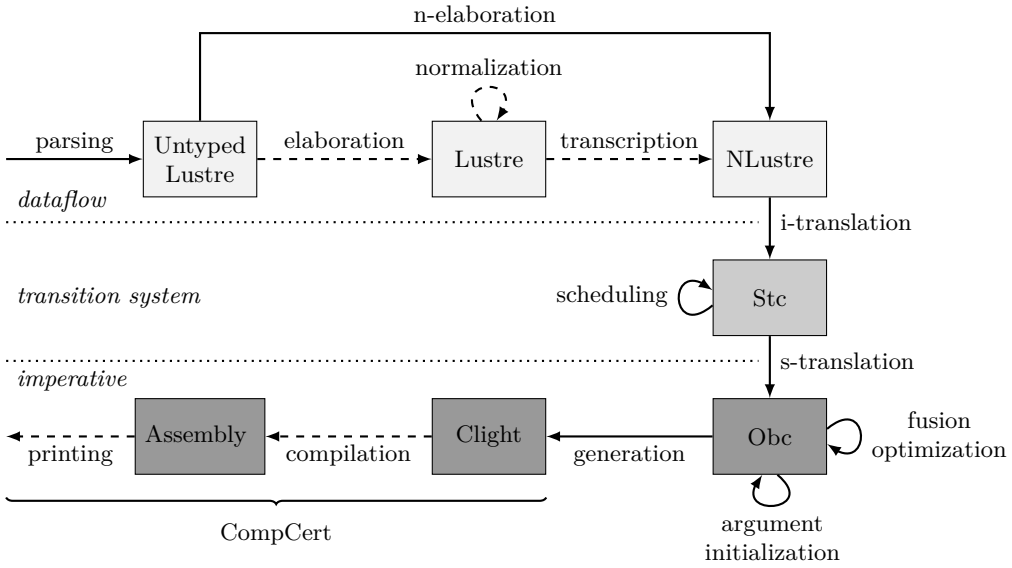


Figure 6.1: The architecture of Vélus

Definition 6.1.1 (compile, src/Velus.v:80)

$$
\begin{aligned}
\mathsf{compile}\ D\ f \triangleq\ & \mathsf{n\text{-}elab}\ D \gg\!= \lambda\ \{G \mid H\}\ .\ \mathsf{OK}\ G && \textit{(n-elaboration, §2.3.3)}\\
& <\$> \mathsf{i\text{-}tr} && \textit{(i-translation, §3.2.2)}\\
& <\$> \mathsf{sch} \gg\!= \mathsf{is\text{-}well\text{-}sch} && \textit{(scheduling, §4.2.3)}\\
& <\$> \mathsf{s\text{-}tr} && \textit{(s-translation, §4.3)}\\
& <\$> \mathsf{fuse} && \textit{(fusion optimization, §4.5.1)}\\
& <\$> \mathsf{init\text{-}args} && \textit{(argument initialization, §5.1)}\\
& \gg\!= \mathsf{gen}\ f && \textit{(generation, §5.2.2)}\\
& \gg\!= \mathsf{cl\text{-}to\text{-}asm} && \textit{(compilation)}\\
& <\$> \lambda\ P.\ (G, P)
\end{aligned}
$$

The n-elab function produces an NLustre program G from the list of declarations D, with a proof H that G is well-typed, well-clocked, and respects the normalization condition on the arguments of node instantiations. The i-tr function transforms the elaborated NLustre program into an Stc program, that is scheduled by the external scheduler sch whose result is validated by the is-well-sch function presented below. The s-tr function transforms the scheduled Stc program into an imperative Obc program, that is optimized by the fuse function that fuses adjacent conditionals. The init-args function ensures that no method is called on an undefined variable, then the Obc program is translated into Clight by the gen function. The rest of the compilation is carried out by the cl-to-asm function (transf_clight2_program, src/ClightToAsm.v:31) which is an extension of CompCert. The compile function returns both the elaborated program G and the compiled assembly program P.

The is-well-sch function lifts the test is-well-sch-tcs (listing 4.3 on page 109) into the error monad, generalizing over the whole program.

Definition 6.1.2 (is_well_sch, src/Velus.v:54)

$$\mathsf{is\text{-}well\text{-}sch\text{-}system}\ r\ s \triangleq r \gg\!= \mathit{if}\ \mathsf{is\text{-}well\text{-}sch\text{-}tcs}\,(\mathsf{set\text{-}of\text{-}list}\ \boldsymbol{y})\ \ \boldsymbol{x}\ s.\boldsymbol{tcs}\ \mathit{then}\ \mathsf{OK}()\ \mathit{else}\ \mathsf{Error}$$
$$\mathit{where}\ s.\boldsymbol{in} = \boldsymbol{x}^{\tau_x, ck_x}\ \mathit{and}\ s.\boldsymbol{inits} = \boldsymbol{y}^{c_y, ck_y}$$
$$\mathsf{is\text{-}well\text{-}sch}\ P \triangleq \mathsf{fold\text{-}l}\ \mathsf{is\text{-}well\text{-}sch\text{-}system}\ \ P\ (\mathsf{OK}()) \gg\!= \mathsf{OK}\ P$$

6.1.2 The proof of correctness

For the compile function to be correct, the generated assembly program must run forever and produce an infinite trace that alternates volatile load and store events that correspond to the values in the input and output streams of the main node in the source NLustre program.

To prove this, we first define a function that builds an infinite trace from a node and its input and output coinductive streams.

Definition 6.1.3 (trace_node, src/Correctness.v:131)
Given a natural number i, a node n and lists of coinductive streams of values $\boldsymbol{xs}$ and $\boldsymbol{ys}$,

$$\mathsf{trace\text{-}node}^i\ \boldsymbol{xs}\ \boldsymbol{ys}\ n \triangleq \mathsf{trace\text{-}step}^i\ (\mathbf{to\text{-}idx}\ \boldsymbol{xs})\ \ (\mathbf{to\text{-}idx}\ \boldsymbol{ys})\ \ \boldsymbol{x}^{\tau_x}\ \boldsymbol{y}^{\tau_y}$$

where $n.\boldsymbol{in} = \boldsymbol{x}^{\tau_x, ck_x}$ and $n.\boldsymbol{out} = \boldsymbol{y}^{\tau_y, ck_y}$

This function uses trace-step (definition 5.5.4 on page 181), that builds an infinite trace from indexed streams of lists of values and variable declarations, and **to-idx** (definition 2.4.4 on page 62) that turns a list of coinductive streams into an indexed stream of lists. The definition, as for trace-step, needs additional hypotheses, namely $\|\boldsymbol{xs}\| = \|n.\mathbf{in}\|$, $\|\boldsymbol{ys}\| = \|n.\mathbf{out}\|$ and $n.\mathbf{in}$, and $n.\mathbf{out}$ are not both empty, which is guaranteed by syntactic invariants.

We define a predicate that relates a node, its input and output streams, and a given infinite trace.

Definition 6.1.4 (bisim_IO, src/Correctness.v:176)

$$\frac{\mathsf{node}(G, f) \doteq n \quad \|\boldsymbol{xs}\| = \|n.\boldsymbol{in}\| \quad \|\boldsymbol{ys}\| = \|n.\boldsymbol{out}\| \quad \mathsf{trace\text{-}node}^0\ \boldsymbol{xs}\ \boldsymbol{ys}\ n \equiv T}{\mathsf{bisim\text{-}IO}^G\ f\ \boldsymbol{xs}\ \boldsymbol{ys}\ T}$$

Now we can state the ultimate correctness theorem.

Theorem 6.1.1 (correctness, src/Correctness.v:468)
Given a list of declarations D, a node identifier f, lists of streams of values $\boldsymbol{xs}$ and $\boldsymbol{ys}$, an NLustre program G and an assembly program P such that $\mathsf{compile}\ D\ f = \mathsf{OK}\ (G, P)$*, the values in $\boldsymbol{xs}$ are well-typed relative to the input declarations of the node named f, and $G \vdash f\ (\langle \boldsymbol{xs} \rangle) \Downarrow \langle \boldsymbol{ys} \rangle$, then there exists an infinite trace of events T such that*

$$P \Downarrow_{ASM} \mathsf{Reacts}(T) \quad \textit{and} \quad \mathsf{bisim\text{-}IO}^G\ f\ \boldsymbol{xs}\ \boldsymbol{ys}\ T$$

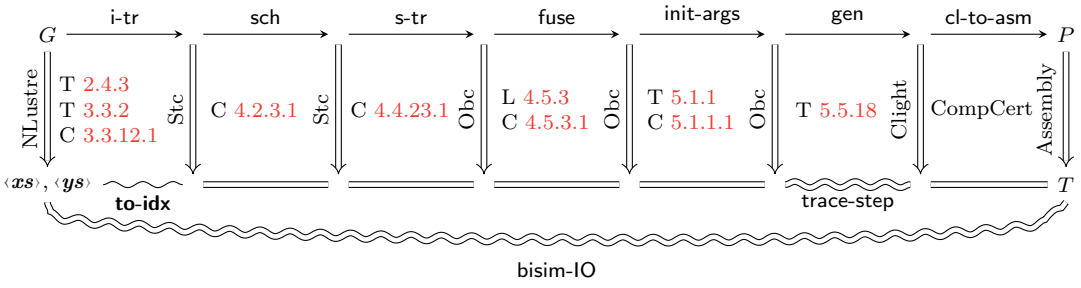


Figure 6.2: Simulation diagrams composition
with lemmas (L), theorems (T) and corollaries (C) references

Proof. Inversion on the compilation hypothesis suffices to show that each pass succeeds. Let n be the main node such that $\mathsf{node}(G, f) \doteq n$, $n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}}$ its input declarations and $n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}}$ its output declarations. We then proceed step-by-step, as represented in figure 6.2.

1. We apply theorem 2.4.3 (page 62) to deduce the indexed semantics of the node.

$$G \vdash f\,(\langle\, \mathbf{to\text{-}idx}\ \boldsymbol{xs} \,\rangle) \Downarrow \langle\, \mathbf{to\text{-}idx}\ \boldsymbol{ys} \,\rangle$$

2. Theorem 3.3.2 (page 92) gives a memory stream M such that the NLustre memory semantics holds.

$$G, M \vdash f\,(\langle\, \mathbf{to\text{-}idx}\ \boldsymbol{xs} \,\rangle) \Downarrow \langle\, \mathbf{to\text{-}idx}\ \boldsymbol{ys} \,\rangle$$

3. The first compilation step is i-translation, that is, translation to Stc. We apply corollary 3.3.12.1 (page 96) to obtain that M_0 is initial with respect to the semantics of Stc, and that the semantics of the repeated activation of the default transition of the translated node is preserved.

$$\mathsf{initial\text{-}state}\,(\mathsf{i\text{-}tr}\,G)\ \ f\ M_0$$
$$\mathsf{i\text{-}tr}\,G, M_0 \vdash f\,(\langle\, \mathbf{to\text{-}idx}\ \boldsymbol{xs} \,\rangle) \overset{0}{\bigcirc} \langle\, \mathbf{to\text{-}idx}\ \boldsymbol{ys} \,\rangle$$

4. Corollary 4.2.3.1 (page 111) ensures that scheduling preserves the semantics. An intermediate result ensures that initial-state is preserved by scheduling. Moreover, another intermediate result shows that the scheduled program sch (i-tr G) is indeed well-scheduled, since is-well-sch succeeds.

$$\mathsf{initial\text{-}state}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G))\ \ f\ M_0$$
$$\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G), M_0 \vdash f\,(\langle\, \mathbf{to\text{-}idx}\ \boldsymbol{xs} \,\rangle) \overset{0}{\bigcirc} \langle\, \mathbf{to\text{-}idx}\ \boldsymbol{ys} \,\rangle$$

5. The next pass is s-translation, that is, translation to Obc. Corollary 4.4.23.1 (page 128) shows the existence of a memory tree $me_0 \approx\!\!\approx M_0$ that is produced by an

initial call to the *reset* method of the translated class, and such that the semantics of the endless repeated execution of the *step* method is preserved.

$$\mathsf{s\text{-}tr}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G)), \{\varnothing\} \vdash f.\mathsf{reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me_0$$
$$\mathsf{s\text{-}tr}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G)), me_0 \vdash f.\mathsf{step}\,(\lfloor \mathbf{to\text{-}idx}\ \boldsymbol{xs} \rfloor) \overset{0}{\circledcirc} \lfloor \mathbf{to\text{-}idx}\ \boldsymbol{ys} \rfloor$$

6. We apply lemma 4.5.3 and corollary 4.5.3.1 (page 135) to show that the fusion optimization preserves the Obc semantics.

$$\mathsf{fuse}\,(\mathsf{s\text{-}tr}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G))), \{\varnothing\} \vdash f.\mathsf{reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me_0$$
$$\mathsf{fuse}\,(\mathsf{s\text{-}tr}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G))), me_0 \vdash f.\mathsf{step}\,(\lfloor \mathbf{to\text{-}idx}\ \boldsymbol{xs} \rfloor) \overset{0}{\circledcirc} \lfloor \mathbf{to\text{-}idx}\ \boldsymbol{ys} \rfloor$$

7. The semantics preservation for the argument initialization is given by theorem 5.1.1 and corollary 5.1.1.1 (page 140). These results need the hypotheses that the Obc program is in SSA form and that its methods inputs are never assigned. A first intermediate result ensures the former for the translation of any well-scheduled Stc program, and another one ensures the latter for the translation of any Stc program. These hypotheses are proved to be preserved by the fusion optimization, so they are satisfied.

$$\mathsf{init\text{-}args}\,(\mathsf{fuse}\,(\mathsf{s\text{-}tr}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G)))), \{\varnothing\} \vdash f.\mathsf{reset}\,(\varepsilon) \overset{\varepsilon}{\Downarrow} me_0$$
$$\mathsf{init\text{-}args}\,(\mathsf{fuse}\,(\mathsf{s\text{-}tr}\,(\mathsf{sch}\,(\mathsf{i\text{-}tr}\,G)))), me_0 \vdash f.\mathsf{step}\,(\lfloor \mathbf{to\text{-}idx}\ \boldsymbol{xs} \rfloor) \overset{0}{\circledcirc} \lfloor \mathbf{to\text{-}idx}\ \boldsymbol{ys} \rfloor$$

8. The next pass is the generation of Clight code. Let P_{CL} be the generated program. We apply theorem 5.5.18 (page 183) to obtain an infinite trace that corresponds to the execution of the generated Clight program.

$$P_{\mathrm{CL}} \Downarrow \mathsf{Reacts}\Big(\mathsf{trace\text{-}step}^0\ (\mathbf{to\text{-}idx}\ \boldsymbol{xs})\ \ (\mathbf{to\text{-}idx}\ \boldsymbol{ys})\ \ \boldsymbol{x}^{\tau_x}\ \boldsymbol{y}^{\tau_y}\Big)$$

9. The correctness lemma about cl-to-asm, that compiles the Clight program P_{CL} into the assembly program P, states that it preserves the reactive behavior of the program (reacts_trace_preservation, src/ClightToAsm.v:224).

$$P \Downarrow_{\mathrm{ASM}} \mathsf{Reacts}\Big(\mathsf{trace\text{-}step}^0\ (\mathbf{to\text{-}idx}\ \boldsymbol{xs})\ \ (\mathbf{to\text{-}idx}\ \boldsymbol{ys})\ \ \boldsymbol{x}^{\tau_x}\ \boldsymbol{y}^{\tau_y}\Big)$$

Now that we have a witness for the trace T, it remains to show that it satisfies bisim-IO. All antecedents are discharged easily but for the bisimilarity obligation. We must show that $\mathsf{trace\text{-}node}^0\ \boldsymbol{xs}\ \boldsymbol{ys}\ n \equiv \mathsf{trace\text{-}step}^0\ (\mathbf{to\text{-}idx}\ \boldsymbol{xs})\ \ (\mathbf{to\text{-}idx}\ \boldsymbol{ys})\ \ \boldsymbol{x}^{\tau_x}\ \boldsymbol{y}^{\tau_y}$. Even if it seems to be a direct consequence of the definition 6.1.3 of trace-node, we need an intermediate result shown by coinduction to get around difficulties about proof irrelevance, since trace-node and trace-step have non-represented parameters that are proofs. □

6.2 Summary

In chapter 2, we presented the formalization of the Lustre dialect treated by Vélus, and in particular, the formalization of the semantics of the modular reset. Our model of the modular reset requires only one additional semantic rule. We also presented two equivalent semantic models for NLustre, the normalized version of Lustre, based on two representations for streams: as coinductive objects, and as functions.

Chapter 3 introduced the Stc language that permits improved code generation and facilitates semantic reasoning after the addition of the modular reset. In Stc, a node is translated into a system that has a state and transitions. The proof of correctness from NLustre to Stc is based on a third semantic model for NLustre where the state of a node is made explicit.

In Chapter 4, we described the generation of imperative Obc code. Each Stc system is translated into an Obc class with fields and two methods *step* and *reset*. Before being translated, the Stc transition constraints are scheduled to fix the sequential order of execution in the generated imperative code. We showed how to optimize the generated Obc code to reduce the number of conditionals.

Chapter 5 presented the generation of Clight. The proof of correctness relies on separation logic to relate the tree-like memory model of Obc with the realistic low-level memory model that is used in CompCert.

Finally, in the previous section, we showed how we combine everything to establish an end-to-end correctness proof for a compilation function from NLustre to assembly code. This function is extracted from Coq to OCaml and integrated into a compiler from a normalized dialect of Lustre with modular reset to an executable binary.

6.3 Outlook

This thesis shows the feasibility of a verified compiler for a subset of Lustre with modular reset. Its components also have value individually. Our ideas and definitions may provide a useful starting point for formalizing other reactive languages in an ITP. The semantic rule for the reset construct is novel and interesting independently of the Coq development. The intermediate language Stc seems to be a useful compromise between NLustre and Obc for expressing and reasoning about optimizations. Like Lustre, it provides a language modeling transition systems with function abstraction and composition, but it also explicitly specifies the manipulation of states. The integration of Clight into all our languages and the techniques used to compile Obc into Clight could serve as a model for other verified compilers.

The Vélus experiment continues. In the short term, we aim to complete the elaboration and transcription passes and to implement normalization. Optimization and efficiency of generated code has always been part of the design of Lustre [Halbwachs, Raymond, and Ratel (1991); Gérard et al. (2012)] and synchronous languages in general. Beside the fusion optimization that we implement, some other optimizations could be formalized and added on dataflow and imperative programs, for instance, dead-code elimination, common

sub-expression elimination, and copy elimination. An interesting question is to determine which optimizations can be left to CompCert and whether optimizations performed upstream in a synchronous compiler interfere with those implemented downstream in CompCert. The way we handle multiple outputs in the translation from Obc to Clight makes the compilation function and its correctness proof rather intricate: we could investigate another common approach where outputs are stored in the same structure that holds the state of the node. Additionally, remark that the correctness result of theorem 6.1.1 is conditioned by the existence of a semantics for the NLustre node. Ideally, we would like a stronger statement where the existence of the semantics is deduced from the well-formedness analyses; this question is the subject of ongoing research. Moreover, our final theorem states a simulation from the source NLustre semantics to the target Assembly semantics, but to achieve *refinement*, we need the converse direction. It can be shown, provided the source program has a semantics, that if the target semantics is deterministic, then *refinement* can be deduced. In our case, the Assembly semantics of CompCert is deterministic, so it only remains to solve the problem of the existence of the semantics for an NLustre program.

A longer term goal is to mechanize the syntax, semantics, and compilation algorithms for state machines. The modular reset is a prerequisite for compiling state machines, but we also need variant types and switch expressions [Colaço, Pagano, and Pouzet (2005)]. Such additions will require generalizations of the type systems, the semantic models and the compilation algorithms. Other features provided by compilers like SCADE [Colaço, Pagano, and Pouzet (2017)] and Heptagon [Delaval et al. (2017)], which support far richer languages than our compiler, could also be investigated:

- The distinction between nodes that have a state and purely combinatorial functions is only a matter of specializing the compilation algorithms. Purely combinatorial functions can be compiled without requiring a local state and methods.

- Adding the **`pre`** construct requires implementing an initialization analysis to ensure that a value is only used when it is defined [Colaço and Pouzet (2004)]. The `->` operator requires the introduction of initial states per clock.

- Arrays [Gérard et al. (2012)], whose integration into Vélus seems difficult both in terms of semantics and of code optimization.

- Activation conditions could be given a semantics in Lustre and compiled into NLustre using the **`when`**, **`fby`** and **`merge`** operators.

- We could adapt the Lustre semantics to handle the generalized 3-arguments **`fby`** of Scade. Its efficient compilation into NLustre would require arrays and index manipulations (using chained **`fby`** equations would give the same effect but may induce excessive copying).

- Parametricity of nodes, that allows a node to take static parameters, requires specific work.

Table 6.1: Lines of Coq code in Vélus (comments and blanks excluded)

	compilation	specification	proofs	other	total
Lustre		2071			2071
CoreExpr[a]		2204			2204
NLustre	567	4360	3621		8548
Stc	65	2708	750		3523
Obc	105	2555	2352		5012
NLustre to Stc	71		1553		1624
Stc to Obc	137		2590		2727
Obc to Clight	434	2052	5097		7583
Common[b]	59	3825	388	4605	8877
Total	1438 (3%)	19775 (47%)	16351 (39%)	4605 (11%)	42169

[a] The common expression kernel language between NLustre and Stc.
[b] End-to-end compilation, common specifications and standard library extensions.

- Records could be implemented with variant types, but they require work to adapt the type systems, find a way to model in-place modifications and to compile them efficiently into Clight.

- External calls could be used to implement abstract data types. CompCert handles external calls and we could take inspiration from the way it models them.

- Side effects would require our semantic models to be adapted to take effects into account. How to do this is unclear and it may be preferable to maintain a purely functional language.

The experiment of using translation validation in combination with fully verified algorithms to implement scheduling in Stc shows the applicability of this approach. In particular it could be applied to type and clock checking, that are currently part of elaboration, making this pass very intricate. In general, we learned that simplicity is a valuable goal when working in an ITP.

Finally, we hope that our formalization could be used to do Lustre program verification directly in Coq. The idea would be to adapt our framework in order to be able to state properties on the source Lustre programs and use our verified algorithms to show the preservation of these properties on generated code.

The executable algorithms in Vélus are neither large nor complicated, yet a lot of effort was required to prove them correct. Table 6.1 gives an overview of the size of the Coq code base of Vélus: of roughly 42 000 lines of Coq code, only 3% define compilation algorithms, the rest are used for specification (e.g., syntax, semantics, type and clock systems, syntactic predicates) and correctness proofs. In CompCert, the compilation algorithms make up 14% of the code base [Leroy (2009b)]. There is thus room for

improvement in Vélus, bearing in mind, however, that one of the main challenges is to specify and relate fundamentally different semantic paradigms.

Kästner et al. (2018) report on a promising experiment on integrating CompCert into a safety-critical industrial application. The beneficial outcome of the experiment both in terms of confidence in the generated code and of performance shows that the approach of verified compilation may be worth the effort. But the rationale for a verified compiler for C does not transfer automatically to a verified compiler for Lustre. Our work does not address the question of apply ITP-based formal methods in the context of the development of certified application such as SCADE. This is an important and interesting topic in itself.

The goal of mechanizing a compiler and its end-to-end correctness proof in an ITP imposed a discipline that led us to take a fresh look at questions that are otherwise easily overlooked and to develop the key contributions of this thesis. First, to reason effectively about the modular reset, we had to design a semantic rule that is suitable for reasoning within an ITP, well suited for compilation correctness proofs, and simpler than previous models. Second, to generate reasonable code, we had to introduce a novel intermediate language and formulate a suitable semantic model. Third, to connect the proofs to CompCert, we had to to propose a solution for relating an abstract tree-like memory model with a machine-level memory model. Implementing a verified compiler in an ITP is difficult, time-consuming and fastidious, but it is also a fruitful source of challenges and ideas.

Appendix A

Vélus source files

`CompCert/` the CompCert development

`extraction/` the directory concerning the code extraction process

- `extracted/` the directory to which the OCaml code is extracted
- `Extraction.v` the Coq file driving the extraction process

`src/` the directory containing the actual Coq development of Vélus

- `Common/` some common definitions and lemmas
  - `CommonList.v` additional results over the Coq standard library for lists
  - `CommonTactics.v` common tactics
  - `Common.v` gathering both files above plus some additional definitions and lemmas
  - `CompCertLib.v` some definitions and results over CompCert internal standard library
- `Lustre/` the Lustre frontend language
  - `Parser/` lexing and parsing
    - `LustreParser.vy` the verified Menhir parser §C
    - `LustreAst.v` the raw abstract syntax
    - `LustreLexer.mll` the Ocamlyacc lexer §B
    - `Relexer.ml` conversion of tokens for the incremental version of the parser used for errors messages
  - `LSyntax.v` the abstract syntax of Lustre §2.2.1
  - `LTyping.v` typing rules §E.1
  - `LClocking.v` clocking rules
  - `LSemantics.v` coinductive semantics of Lustre §2.2.2
  - `Lustre.v` meta-library, gathering everything about Lustre
  - `lustrelib.ml` OCaml generic printer for Lustre
- `CoreExpr/` the core expression language (shared by NLustre and Stc)

§2.3.2	`CESyntax.v`	abstract syntax
	`CEIsFree.v`	free variables
§E.2	`CETyping.v`	typing rules
§2.3.4	`CEClocking.v`	clocking rules
§2.3.6	`CESemantics.v`	instantaneous and indexed semantics
§2.4.1	`CEInterpreter.v`	a correct interpreter
	`CEClockingSemantics.v`	enforcing synchronous execution
	`CEProperties.v`	additional properties
	`CoreExpr.v`	meta-library
	`coreexprlib.ml`	OCaml generic printer for CoreExpr
`NLustre/`	the normalized Lustre intermediate language	
§2.3.2	`NLSyntax.v`	abstract syntax
	`NLElaboration.v`	elaboration of NLustre code
	`NLOrdered.v`	enforcing the order of node declarations
	`Memories.v`	variables defined by a **`fby`**
	`IsDefined.v`	variables defined by a set of equations
	`IsVariable.v`	defined variables not defined by a **`fby`**
	`IsFree.v`	free variables
	`NoDup.v`	unicity of the left-hand side of the equations
§4.4.4.2	`NLNormalArgs.v`	normalization condition on nodes' arguments
§E.2	`NLTyping.v`	typing rules
§2.3.4	`NLClocking.v`	clocking rules
§2.3.6	`NLIndexedSemantics.v`	instantaneous and indexed semantics
§3.3.1	`NLMemSemantics.v`	alternative semantics bringing up node instances and state variables and the proof that it contains the standard semantics
	`NLClockingSemantics.v`	enforcing synchronous execution
§2.3.5	`NLCoindSemantics.v`	alternative coinductive streams semantics
§2.4.1	`NLCoindToIndexed.v`	the indexed semantics encompasses the coinductive one
§2.4.2	`NLIndexedToCoind.v`	*vice versa*
	`NLustre.v`	meta-library
	`nlustrelib.ml`	OCaml generic printer for NLustre

`Stc/` the intermediate transition system language

`StcSyntax.v`	abstract syntax	§3.2.1
`StcOrdered.v`	enforcing the order of declarations	
`StcIsInit.v`	state variables	
`StcIsVariable.v`	defined variables which are not state variables	
`StcIsDefined.v`	defined variables	
`StcIsFree.v`	free variables	
`StcIsSystem.v`	called blocks	
`StcTyping.v`	typing rules	§E.3
`StcClocking.v`	clocking rules	§3.2.3
`StcSemantics.v`	instantaneous transition semantics	§3.2.4
`StcClockingSemantics.v`	enforcing synchronous execution	
`StcWellDefined.v`	well scheduled, ordered and normalized programs	§§4.2.1 and 4.4.4.2
`StcSchedule.v`	scheduler specification	§4.2.3.1
`StcSchedulingValidator.v`	verified scheduling validator	§4.2.2
`Stc.v`	meta-library	
`stclib.ml`	OCaml generic printer and external scheduler for Stc	

`Obc/` the intermediate imperative language

`ObcSyntax.v`	abstract syntax	§4.1.1
`ObcInvariants.v`	syntactic invariants	§§4.5.2 and 5.1
`ObcAddDefaults.v`	argument initialization	§5.1
`Fusion.v`	fuse adjacent conditionals on the same guard	§4.5
`ObcTyping.v`	typing rules	§E.4
`ObcSemantics.v`	operational big-step style semantics	§4.1.2
`Equiv.v`	quotient set of programs by semantics equivalence	§5.1
`Obc.v`	meta-library	
`obclib.ml`	OCaml generic printer for Obc	

`NLustreToStc/` translation from NLustre to Stc

`Translation.v`	i-translation	§3.2.2
`NL2StcTyping.v`	typing preservation	
`NL2StcClocking.v`	clocking preservation	
`NL2StcNormalArgs.v`	normalization results preservation	§4.4.4.2

§3.3.2	`Correctness.v`	correctness of the translation
	`StcToObc/`	translation from Stc to Obc
§4.3	`Translation.v`	s-translation
	`Stc2ObcTyping.v`	typing preservation
§4.5.3	`Stc2ObcInvariants.v`	various syntactic invariants preservation
§4.4.2	`StcMemoryCorres.v`	correspondence between Stc state and Obc state
§4.4	`Correctness.v`	correctness of the translation
	`ObcToClight/`	interface with CompCert and Clight code generation
§5.3.2	`Interface.v`	instantiation with types, operators and values from CompCert
§5.2.2	`Generation.v`	generation of Clight code
	`GenerationProperties.v`	properties of the generation function
§5.4.2	`MoreSeparation.v`	some definitions over the internal SL library of CompCert
§5.4.3	`SepInvariant.v`	SL assertions to describe the state of a Clight program
§5.5	`Correctness.v`	correctness of the generation function
	`interfacelib.ml`	OCaml instantiated printers and printer for Clight
	`ClockDefs.v`	basic definitions around clocks
	`Clocks.v`	lemmas and properties about clocks
	`Ident.v`	identifier specification and properties
	`Environment.v`	a library extending the standard maps with positive keys of Coq
	`VelusMemory.v`	a library defining a generic tree-based memory model
	`CoindStreams.v`	streams of A as coinductive sequences of A
	`IndexedStream.v`	streams of A as functions from $\mathbb{N}$ to A (indexed streams)
	`Operators.v`	specification of values, types and operators
§5.5.2.1	`Traces.v`	specification and definitions around traces of a Clight program
	`Instantiator.v`	instantiate the various functors
	`ClightToAsm.v`	compilation of Clight to Assembly (CompCert extension)
§6.1.1	`Velus.v`	gathering all passes together
§6.1.2	`Correctness.v`	end-to-end proof of correctness
	`veluscommon.ml`	shared OCaml definitions: generic printers for operators and types, conversion between Coq and OCaml integers
	`veluslib.ml`	OCaml implementation of side-effect procedures used in the compilation chain (e.g. command-line flags handling, printers)
	`velusmain.ml`	the main driver of the compiler

Appendix B

Vélus lexical conventions

Below are the main lexical conventions used by Vélus, with the only difference with CompCert being the impossibility of using the character `$` in *identifier_nondigit*, as we use it as an internal separator in generated Clight code. References to relevant sections of the ISO C 99 standard are indicated.

Identifiers **§6.4.2.1**

digit ::= `0...9`

nondigit ::= `_` | `a...z` | `A...Z`

identifier_nondigit ::= *nondigit*

identifier ::= *identifier_nondigit* (*identifier_nondigit* | *digit*)*

Note that the standard allows *universal_character_name* (see below) to be used in *identifier_nondigit* but CompCert does not.[1] Vélus follows CompCert on this point.

White-space characters **§6.4**

whitespace ::= ␣ | `\t` | `\n` | `\012` | `\r`

The `\012` character denotes the form feed in OCaml syntax. In the standard, the vertical tab is mentioned instead of the carriage return character.

Integer constants **§6.4.4.1**

nonzero_digit ::= `1...9`

decimal_constant ::= *nonzero_digit digit**

octal_digit ::= `0...7`

octal_constant ::= `0` *octal_digit**

hex_prefix ::= `0x` | `0X`

hex_digit ::= `0...9` | `A...F` | `a...f`

hex_constant ::= *hex_prefix hex_digit*$^+$

unsigned_suffix ::= `u` | `U`

[1] The relevant line is actually commented out in CompCert sources (CompCert/cparser/Lexer.mll).

long_suffix ::= `l` | `L`

long_long_suffix ::= `ll` | `LL`

integer_suffix ::= *unsigned_suffix* [*long_suffix*]
| *unsigned_suffix long_long_suffix*
| *long_suffix* [*unsigned_suffix*]
| *long_long_suffix* [*unsigned_suffix*]

integer_constant ::= *decimal_constant* [*integer_suffix*]
| *octal_constant* [*integer_suffix*]
| *hex_constant* [*integer_suffix*]

Floating constants **§6.4.4.2**

sign ::= `-` | `+`

digit_sequence ::= *digit*$^+$

floating_suffix ::= `f` | `l` | `F` | `L`

fractional_constant ::= [*digit_sequence*] . *digit_sequence*
| *digit_sequence* .

exponent_part ::= `e` [*sign*] *digit_sequence*
| `E` [*sign*] *digit_sequence*

decimal_floating_constant ::= *fractional_constant* [*exponent_part*] [*floating_suffix*]
| *digit_sequence exponent_part* [*floating_suffix*]

hex_digit_sequence ::= *hex_digit*$^+$

hex_fractional_constant ::= [*hex_digit_sequence*] . *hex_digit_sequence*
| *hex_digit_sequence* .

binary_exponent_part ::= `p` [*sign*] *digit_sequence*
| `P` [*sign*] *digit_sequence*

hex_floating_constant ::= *hex_prefix hex_fractional_constant*
binary_exponent_part [*floating_suffix*]
| *hex_prefix hex_digit_sequence*
binary_exponent_part [*floating_suffix*]

floating_constant ::= *decimal_floating_constant*
| *hex_floating_constant*

Character constants **§6.4.4.4**

simple_escape_sequence ::= \ (`'` | `"` | `?` | \ | `a` | `b` | `e` | `f` | `n` | `r` | `t` | `v`)

octal_escape_sequence ::= \ (*octal_digit*
| *octal_digit octal_digit*
| *octal_digit octal_digit octal_digit*)

hex_escape_sequence	::=	`\x` *hex_digit*$^+$
hex_quad	::=	*hex_digit hex_digit hex_digit hex_digit*
universal_chararcter_name	::=	`\u` *hex_quad*
	|	`\U` *hex_quad hex_quad*
escape_sequence	::=	*simple_escape_sequence*
	|	*octal_escape_sequence*
	|	*hex_escape_sequence*
	|	*universal_character_name*
char	::=	*escape_sequence*
	|	any character different from `'`, `\n` and `\`
char_literal	::=	[`L`]`'`*char*$^+$`'`

Note that the escape sequence `\e`, which represents the escape character, is non-standard but is supported in e.g. GCC or Clang.

Comments §6.4.9

Vélus supports Scade-like comments, that is single line `--` comments and multi-lines `/* - */` comments.[2] In addition, to be the most general possible, to C-like comment single line `//` comments and OCaml-like nested multi-lines comments `(* - *)` are also supported.

[2] Contrary to C and Scade, multi-lines comments can be nested.

Appendix C

The Lustre parser of Vélus

Below is a textual version of the parser used for Vélus, where some rules (lists, options) have been *inlined*.[1]

Constants

bool_constant ::= `true`
| `false`

constant ::= `CONSTANT`
| *bool_constant*

Expressions

Expressions are parsed in the same way as in CompCert, reflecting the ISO C 99 standard, with some operators renamed and Vélus-specific operators and constructs added.

primary_expression ::= `ID`
| *constant*
| `(` $expression^{+}_{,}$ `)`

postfix_expression ::= *primary_expression*
| `ID (` $expression^{+}_{,}$ `)`
| `(` `restart` `ID` `every` *expression* `)` `(` *expression_list* `)`

unary_expression ::= *postfix_expression*
| *unary_operator cast_expression*
| `#` `(` $expression^{+}_{,}$ `)`

unary_operator ::= `-`
| `lnot`
| `not`

cast_expression ::= *unary_expression*
| `(` *cast_expression* `:` *type_name* `)`

multiplicative_expression ::= $cast\_expression^{+}_{\texttt{fby}}$
| *multiplicative_expression* `*` $cast\_expression^{+}_{\texttt{fby}}$
| *multiplicative_expression* `/` $cast\_expression^{+}_{\texttt{fby}}$
| *multiplicative_expression* `mod` $cast\_expression^{+}_{\texttt{fby}}$

[1] The corresponding LaTeX code used here is obtained through the Obelisk tool I developed during my PhD: github.com/Lelio-Brun/Obelisk.

additive_expression ::= *multiplicative_expression*
| *additive_expression* `+` *multiplicative_expression*
| *additive_expression* `-` *multiplicative_expression*

shift_expression ::= *additive_expression*
| *shift_expression* `lsl` *additive_expression*
| *shift_expression* `lsr` *additive_expression*

when_expression ::= *shift_expression*
| *when_expression* `when ID`
| *when_expression* `when not ID`
| *when_expression* `whenot ID`

relational_expression ::= *when_expression*
| *relational_expression* `<` *when_expression*
| *relational_expression* `>` *when_expression*
| *relational_expression* `<=` *when_expression*
| *relational_expression* `>=` *when_expression*

equality_expression ::= *relational_expression*
| *equality_expression* `=` *relational_expression*
| *equality_expression* `<>` *relational_expression*

exclusive_OR_expression ::= $\mathit{equality\_expression}^{+}_{\texttt{land}}$
| *exclusive_OR_expression* `lxor` $\mathit{equality\_expression}^{+}_{\texttt{land}}$
| *exclusive_OR_expression* `xor` $\mathit{equality\_expression}^{+}_{\texttt{land}}$

expression ::= $\mathit{logical\_AND\_expression}^{+}_{\texttt{or}}$
| `if` *expression* `then` *expression* `else` *expression*
| `merge ID` *primary_expression primary_expression*
| `merge ID ( true ->` *expression* `)`
`( false ->` *expression* `)`

`ID` represents a token carrying an *identifier* as described in the appendix B, translated into a Coq positive using a CompCert internal function.[2]

Variable declarations

Variable are declared along with a type annotation *type_name* taken among the supported type names and an optional clock annotation.

var_decl ::= $\texttt{ID}^{+}_{,}$ `:` *type_name declared_clock*

local_var_decl ::= `var` $\mathit{var\_decl}^{+}_{;}$ `;`

type_name ::= `int8` | `int16` | `int32` | `int` | `int64`
| `uint8` | `uint16` | `uint32` | `uint` | `uint64`

[2]The OCaml function `intern_string` in the file CompCert/lib/Camlcoq.ml

	`float32` | `float` | `float64` | `real`
	| `bool`
declared_clock	::= ϵ
	| `when ID`
	| `when not ID`
	| `whenot ID`
	| `::` *clock*
clock	::= `.`
	| *clock* `on ID`
	| *clock* `onot ID`
local_decl	::= *local_var_decl*

Equations, nodes and programs

A Lustre *program* is a list of nodes[3]. A node consists in a list of equations, which bind expressions to patterns. Optionally, some assertions can be provided for testing purposes.

equation	::= $\texttt{ID}^{+}_{,}$ `=` *expression* `;`
	| `(` $\texttt{ID}^{+}_{,}$ `)` `=` *expression* `;`
equations	::= ϵ
	| *equations equation*
	| *equations* `assert` *expression* `;`
node_or_function	::= `node`
	| `function`
declaration	::= *node_or_function* `ID (` $\textit{var\_decl}^{*}_{;}$ `)` [`;`]
	`returns (` $\textit{var\_decl}^{*}_{;}$ `)` [`;`]
	*local_decl**
	`let` *equations* `tel` [`;`]
program	::= $\textit{declaration}^{+}$ `EOF`
	| `EOF`

[3] Actually a list of nodes or functions, where a function is a purely combinatorial node, i.e, a node without **`fby`**. Said differently, a node is a stateful function.

Appendix D

Functorizing the development

Let *Foo* be a typed language that we want to formalize, containing only constants and binary operations, with the following syntax:

$$\begin{array}{rcll} e & ::= & c & \textbf{constant} \\ & | & e \oplus e & \textbf{operator} \end{array}$$

Assume that Foo is parametric over a simple abstraction layer, that we implement as a module signature `OPERATORS` in the file `Operators.v` (note that here, we do not require the semantics of operators to be type-dependent):

```
Module Type OPERATORS.
  Parameter const: Type.
  Parameter val: Type.
  Parameter type: Type.
  Parameter binop: Type.
  Parameter type_const: const -> type.
  Parameter sem_const: const -> val.
  Parameter sem_binop: binop -> val -> val -> option val.
  Parameter type_binop: binop -> type -> type -> option type.
End OPERATORS.
```

Coq (toy/Operators.v)

First, let us describe the abstract syntax of the language in a `FooSyntax.v` file. As we want it to be parametric, we encapsulate the definitions in a functor parameterized by the `OPERATORS` signature. Unfortunately, because of restrictions of the module system of Coq, we have to reflect the dependency graph of each abstracted functor directly in its call-graph. Consequently, we have to explicitly define both a signature and an actual functor for each unit. Fortunately in Coq a module signature can contain actual implementations, so at the end the actual functor can just include directly the signature.

```
Require Import Operators.

Module Type FOOSYNTAX (Import Op: OPERATORS).

  Inductive expr: Type :=
  | Econst: const -> expr
  | Eop: binop -> expr -> expr -> expr.
```

```
End FOOSYNTAX.

Module FooSyntaxFun (Op: OPERATORS) <: FOOSYNTAX Op.
  Include FOOSYNTAX Op.
End FooSyntaxFun.
```

Coq (toy/FooSyntax.v)

The syntax is defined using the *inductive* Coq definition `expr`, made of two constructors `Econst` for the constants and `Eop` for operator applications. The `typeof` definition is a function used to retrieve the type of an expression: it uses the parameter `type_const` to get the type of a constant or returns the type annotation in case of an operator application.

Hence, consider the following typing rules, which judge whether that an expression is well typed:

$$\frac{}{\vdash c : \iota_c}\ \textsc{WTconst} \qquad \frac{\vdash e_1 : t_1 \quad \vdash e_2 : t_2 \quad \vdash \oplus : t_1 \times t_2 \to t}{\vdash e_1 \oplus e_2 : t}\ \textsc{WTop}$$

We can formalize these rules the same way in Coq, using a functor:

```
Require Import Operators FooSyntax.

Module Type FOOTYPING
       (Import Op: OPERATORS)
       (Import Syn: FOOSYNTAX Op).

  Inductive wt: expr -> type -> Prop :=
  | WTconst: forall c, wt (Econst c) (type_const c)
  | WTop: forall op e1 e2 t1 t2 t,
      wt e1 t1 ->
      wt e2 t2 ->
      type_binop op t1 t2 = Some t ->
      wt (Eop op e1 e2) t.

End FOOTYPING.

Module FooTypingFun
       (Op: OPERATORS)
       (Syn: FOOSYNTAX Op) <: FOOTYPING Op Syn.
  Include FOOTYPING Op Syn.
End FooTypingFun.
```

Coq (toy/FooTyping.v)

The *well typedness* is encoded by the inductive predicate `wt` with one case per syntactic class of `expr`, which is made accessible by instantiating the functor `FooSyntaxFun` defined in the imported library `FooSyntax` (the above file `FooSyntax.v`) with the `Op` parameter. The predicate `wt` is a relation between expressions (`expr`) and types (`type`), that is a

pair (e, t) belongs to the relation when one of the constructors can be applied to it and its antecedents are satisfied. The constructors of `wt` can express arbitrarily complex logical content. Here `WTconst` simply states that a constant is always well typed with type `type_const c`, implementing ι_c. And the chain of implications stated by `WTop` is to be read as a list of premises before the last arrow—saying in order that e_1 is well typed, e_2 is well typed, the return type of the operator can be resolved given the types of its operands—and a conclusion after.[1]

Now consider the semantics of the language, given by the following inference rules (where the symbol $\doteq$ is used to denote the result of a partial function):

$$\textsc{Sconst}\ \frac{}{\vdash c \Downarrow [\![c]\!]} \qquad \textsc{Sop}\ \frac{\vdash e_1 \Downarrow v_1 \quad \vdash e_2 \Downarrow v_2 \quad [\![\oplus]\!]\ v_1\ v_2 \doteq v}{\vdash e_1 \oplus e_2 \Downarrow v}$$

And their implementation in Coq:

```
Require Import Operators FooSyntax.

Module Type FOOSEMANTICS
       (Import Op: OPERATORS)
       (Import Syn: FOOSYNTAX Op).

  Inductive sem: expr -> val -> Prop :=
  | Sconst: forall c, sem (Econst c) (sem_const c)
  | Sop: forall op e1 e2 v1 v2 v,
      sem e1 v1 ->
      sem e2 v2 ->
      sem_binop op v1 v2 = Some v ->
      sem (Eop op e1 e2) v.

End FOOSEMANTICS.

Module FooSemanticsFun
       (Op: OPERATORS)
       (Syn: FOOSYNTAX Op) <: FOOSEMANTICS Op Syn.
  Include FOOSEMANTICS Op Syn.
End FooSemanticsFun.
```

Coq (toy/FooSemantics.v)

The implementation is very similar to the one for typing: the semantic judgments are modeled by the `sem` inductive predicate relating expressions to values, `sem_const c` implementing the constant interpretation $[\![c]\!]$ and `sem_binop op` implementing the operator semantics $[\![\oplus]\!]$.

Now, one can try to prove a *progress-like* [Pierce (2002)] result like:

$$\forall e\, t,\ \vdash e : t \ \rightarrow\ \exists v\ \vdash e \Downarrow v$$

[1] Indeed, recall that $A \rightarrow B \rightarrow C \leftrightarrow A \wedge B \rightarrow C$

We need both semantics and typing to state the lemma, so in Coq, we write the following in a separate FooProgress.v file:

```
Require Import Operators FooSyntax FooTyping FooSemantics.

Module Type FOOPROGRESS
       (Import Op: OPERATORS)
       (Import Syn: FOOSYNTAX Op)
       (Import Typ: FOOTYPING Op Syn)
       (Import Sem: FOOSEMANTICS Op Syn).
  Section Progress.
    Hypothesis binop_progress:
      forall op e1 e2 t1 t2 t v1 v2,
        wt e1 t1 ->
        wt e2 t2 ->
        sem e1 v1 ->
        sem e2 v2 ->
        type_binop op t1 t2 = Some t ->
        exists v, sem_binop op v1 v2 = Some v.

    Lemma foo_progress:
      forall e t, wt e t -> exists v, sem e v.
    Proof.
      induction 1.
      - exists (sem_const c). constructor.
      - destruct IHwt1 as (v1 & ?), IHwt2 as (v2 & ?).
        edestruct (binop_progress op e1 e2 t1 t2) as (v & ?); eauto.
        exists v. econstructor; eauto.
    Qed.
  End Progress.
End FOOPROGRESS.

Module FooProgressFun
       (Op: OPERATORS)
       (Syn: FOOSYNTAX Op)
       (Typ: FOOTYPING Op Syn)
       (Sem: FOOSEMANTICS Op Syn) <: FOOPROGRESS Op Syn Typ Sem.
  Include FOOPROGRESS Op Syn Typ Sem.
End FooProgressFun.
```

Coq (toy/FooProgress.v)

We proceed as before, importing relevant library and instantiating according functors with the Op parameter. The lemma is placed into a *section* in which an hypothesis named binop_progress is stated. This hypothesis basically extends the progress result to the operators, allowing to prove the relevant case when doing the induction in the proof of the lemma. The section mechanism is often used in the development of Vélus because it helps to organize the content and reduces code duplication when several lemmas share the same hypotheses. The bunch of code between the **Proof** and **Qed** keywords is a *proof script*. It consists of *tactics* applications that are used to build the corresponding *proof term*.

Finally, to instantiate the abstraction layer, one needs only to provide a module giving a concrete implementation of the signature OPERATORS and instantiate the cascade of functors. For example:

```
Require Import Operators FooSyntax FooTyping FooSemantics FooProgress.
Open Scope bool_scope.

Module Op <: OPERATORS.
  Inductive const_ind: Type :=
  | Cnat: nat -> const_ind
  | Cbool: bool -> const_ind.
  Definition const := const_ind.
  Definition val := const.
  Inductive type_ind: Type :=
  | Tnat
  | Tbool.
  Definition type := type_ind.
  Inductive binop_ind: Type :=
  | Plus
  | Or.
  Definition binop := binop_ind.
  Definition type_const (c: const) : type :=
    match c with
    | Cnat _ => Tnat
    | Cbool _ => Tbool
    end.
  Definition sem_const (c: const) : val := c.
  Definition sem_binop (op: binop) (v1 v2: val) : option val :=
    match op, v1, v2 with
    | Plus, Cnat n1, Cnat n2 => Some (Cnat (n1 + n2))
    | Or, Cbool b1, Cbool b2 => Some (Cbool (b1 || b2))
    | _, _, _ => None
    end.
  Definition type_binop (op: binop) (t1 t2: type) : option type :=
    match op, t1, t2 with
    | Plus, Tnat, Tnat => Some Tnat
    | Or, Tbool, Tbool => Some Tbool
    | _, _, _ => None
    end.
End Op.

Import Op.
Module Import Syn := FooSyntaxFun Op.
Module Import Typ := FooTypingFun Op Syn.
Module Import Sem := FooSemanticsFun Op Syn.
Module Import Prog := FooProgressFun Op Syn Typ Sem.
```

Coq (toy/FooInstantiate.v)

Notice that the implementation of Op is a bit verbose: const, type and binop are just aliases to their actual inductive definitions; this is because Coq will not instantiate a parameter by an inductive definition. With such an implementation of the abstraction

layer it becomes possible to, for example, prove the progress-like lemma as we stated it the first time. Here is the actual proof script, for the interested reader:

```
Require Import Operators FooSyntax FooTyping FooSemantics FooProgress.
Open Scope bool_scope.

Module Op <: OPERATORS.
  Inductive const_ind: Type :=
  | Cnat: nat -> const_ind
  | Cbool: bool -> const_ind.
  Definition const := const_ind.
  Definition val := const.
  Inductive type_ind: Type :=
  | Tnat
  | Tbool.
  Definition type := type_ind.
  Inductive binop_ind: Type :=
  | Plus
  | Or.
  Definition binop := binop_ind.
  Definition type_const (c: const) : type :=
    match c with
    | Cnat _ => Tnat
    | Cbool _ => Tbool
    end.
  Definition sem_const (c: const) : val := c.
  Definition sem_binop (op: binop) (v1 v2: val) : option val :=
    match op, v1, v2 with
    | Plus, Cnat n1, Cnat n2 => Some (Cnat (n1 + n2))
    | Or, Cbool b1, Cbool b2 => Some (Cbool (b1 || b2))
    | _, _, _ => None
    end.
  Definition type_binop (op: binop) (t1 t2: type) : option type :=
    match op, t1, t2 with
    | Plus, Tnat, Tnat => Some Tnat
    | Or, Tbool, Tbool => Some Tbool
    | _, _, _ => None
    end.
End Op.

Import Op.
Module Import Syn := FooSyntaxFun Op.
Module Import Typ := FooTypingFun Op Syn.
Module Import Sem := FooSemanticsFun Op Syn.
Module Import Prog := FooProgressFun Op Syn Typ Sem.

Lemma type_binop_inv:
  forall op t1 t2 t,
    type_binop op t1 t2 = Some t ->
    t1 = t /\ t2 = t.
Proof.
  intros * Tbop.
```

```
  destruct op, t1, t2; simpl in *;
    inversion_clear Tbop; auto.
Qed.

Lemma sem_binop_inv:
  forall op v1 v2 v,
    sem_binop op v1 v2 = Some v ->
    match v with
    | Cbool _ => exists b1 b2, v1 = Cbool b1 /\ v2 = Cbool b2
    | Cnat _ => exists n1 n2, v1 = Cnat n1 /\ v2 = Cnat n2
    end.
Proof.
  intros * Sop.
  destruct op, v1, v2; simpl in *;
    inversion_clear Sop; eauto.
Qed.

Lemma type_nat_not_bool:
  forall e,
    wt e Tnat ->
    forall b, ~ sem e (Cbool b).
Proof.
  induction e; intros * WT ? Sem.
  - destruct c.
    + inversion Sem.
    + inversion WT.
  - inversion WT as [|???????? Tbop];
      inversion Sem as [|???????? Sop]; subst.
    apply type_binop_inv in Tbop as (?&?); subst.
    apply sem_binop_inv in Sop as (?&?&?&?); subst.
    eapply IHe1; eauto.
Qed.

Lemma type_bool_not_nat:
  forall e,
    wt e Tbool ->
    forall n, ~ sem e (Cnat n).
Proof.
  induction e; intros * WT ? Sem.
  - destruct c.
    + inversion WT.
    + inversion Sem.
  - inversion WT as [|???????? Tbop];
      inversion Sem as [|???????? Sop]; subst.
    apply type_binop_inv in Tbop as (?&?); subst.
    apply sem_binop_inv in Sop as (?&?&?&?); subst.
    eapply IHe1; eauto.
Qed.

Lemma binop_progress:
  forall op e1 e2 t1 t2 t v1 v2,
    wt e1 t1 ->
    wt e2 t2 ->
    sem e1 v1 ->
```

```
    sem e2 v2 ->
    type_binop op t1 t2 = Some t ->
    exists v, sem_binop op v1 v2 = Some v.
Proof.
  intros * WT1 WT2 S1 S2 Tbop.
  destruct op, t1, t2; inversion Tbop; subst.
  - destruct v1, v2; simpl; eauto;
      match goal with H: sem _ (Cbool _) |- _ => contradict H end;
      apply type_nat_not_bool; auto.
  - destruct v1, v2; simpl; eauto;
      match goal with H: sem _ (Cnat _) |- _ => contradict H end;
      apply type_bool_not_nat; auto.
Qed.

Theorem progress:
  forall e t, wt e t -> exists v, sem e v.
Proof.
  apply foo_progress, binop_progress.
Qed.
```

Coq (toy/FooInstantiate.v),firstline=44

Appendix E

Type systems

E.1 Lustre

The type system of Lustre formalized in Vélus is shown in figure E.1. This type system is well understood. While higher-order polymorphic variants have been formalized [Colaço, Girault, et al. (2004); Colaço, Pagano, and Pouzet (2005)] we focus on a simpler monomorphic first order setup *à la* ML. We use the standard inference rules layout, for example, the judgment $\Gamma \vdash a$ reads "under the typing environment Γ, the construct a is well typed". As our Lustre internal dialect is annotated, the type system presented here does not describe directly a type inference algorithm: it is strictly speaking a type checking calculus, that is a predicate asserting the *well-typedness* of a construct relatively to its type annotation.

Figure E.1a present the typing of clocks and named clocks. The base clock $\bullet$ is always well typed, while a sub-clock is well typed if its parent clock is well typed and if its condition variable has type `bool` (the boolean type that the abstraction layer must provide, see section 2.1.1) in the environment.

Figure E.1c introduces the type system for expressions. A constant is always well typed. A variable is well typed if it is bound to its type annotation in the environment and if its clock annotation is well typed. Here, and in all the other rules, we require that the clock annotation be well typed. Unary and binary operations are well typed if their operands are, if their type annotations match with the input signature of the operator and if the type annotation of the whole expression matches with the output signature of the operator. The operator types takes an expression and strips its annotation into a list of types. Refer to table 2.1 on page 22 for the type resolution of operators. A **`fby`** expression is well typed if both its operands are and if their types match with the type annotation of the expression. Several notations are lifted to lists in an intuitive way. $\Gamma \vdash_\alpha \boldsymbol{a}$ is a shortcut for $\forall a \in \boldsymbol{a},\ \Gamma \vdash_\alpha a$. For types, $\mathsf{types}\,\boldsymbol{e}$ indicates the flattened concatenation of all the types of all the expressions in $\boldsymbol{e}$. $\Gamma(\boldsymbol{x}) = \boldsymbol{\tau}$ expresses that each element of $\boldsymbol{x}$ is bound in Γ to the element of $\boldsymbol{\tau}$ of the same rank, assuming that both lists have the same length. A sampling **`when`** expression is well typed if the condition variable is a boolean, its sampled expression is well typed, and if its types match with the type annotation of the whole expression. The typing of the **`merge`** operation operation is similar. The hypotheses for the well typing of a conditional expression are almost the same as for a merging operation, except that the variable is replaced by a single expression which must also have a boolean type. For a node instantiation to be well typed, the arguments

$$\frac{}{\Gamma \vdash \bullet} \qquad \frac{\Gamma(x) = \texttt{bool} \quad \Gamma \vdash ck}{\Gamma \vdash ck \texttt{ on } (x = b)}$$

(a) Clocks
(wt_clock, src/Lustre/LTyping.v:44)

$$\frac{\Gamma \vdash ck}{\Gamma \vdash (x : ck)}$$

(b) Named clocks
(wt_nclock, src/Lustre/LTyping.v:52)

$$\frac{}{G, \Gamma \vdash c} \qquad \frac{\Gamma(x) = \tau \quad \Gamma \vdash nck}{G, \Gamma \vdash x^{\tau, nck}} \qquad \frac{G, \Gamma \vdash e \quad \mathsf{types}\, e = [\tau_e] \quad \vdash \diamond : \tau_e \to \tau \quad \Gamma \vdash nck}{G, \Gamma \vdash (\diamond\, e)^{\tau, nck}}$$

$$\frac{\begin{array}{c} G, \Gamma \vdash e_1 \quad G, \Gamma \vdash e_2 \\ \mathsf{types}\, e_1 = [\tau_1] \quad \mathsf{types}\, e_2 = [\tau_2] \quad \vdash \oplus : \tau_1 \times \tau_2 \to \tau \quad \Gamma \vdash nck \end{array}}{G, \Gamma \vdash (e_1 \oplus e_2)^{\tau, nck}}$$

$$\frac{G, \Gamma \vdash \boldsymbol{e_0} \quad G, \Gamma \vdash \boldsymbol{e} \quad \mathsf{types}\, \boldsymbol{e} = \mathsf{types}\, \boldsymbol{e_0} = \boldsymbol{\tau} \quad \Gamma \vdash \boldsymbol{nck}}{G, \Gamma \vdash (\boldsymbol{e_0}\ \texttt{fby}\ \boldsymbol{e})^{\boldsymbol{a}}} \quad \text{where } \boldsymbol{a} = (\boldsymbol{\tau}, \boldsymbol{nck})$$

$$\frac{\Gamma(x) = \texttt{bool} \quad G, \Gamma \vdash \boldsymbol{e} \quad \mathsf{types}\, \boldsymbol{e} = \boldsymbol{\tau} \quad \Gamma \vdash nck}{G, \Gamma \vdash (\boldsymbol{e}\ \texttt{when}\ (x = b))^{\boldsymbol{\tau}, nck}} \qquad \frac{\begin{array}{c} \Gamma(x) = \texttt{bool} \quad G, \Gamma \vdash \boldsymbol{e_t} \quad G, \Gamma \vdash \boldsymbol{e_f} \\ \mathsf{types}\, \boldsymbol{e_t} = \mathsf{types}\, \boldsymbol{e_f} = \boldsymbol{\tau} \quad \Gamma \vdash nck \end{array}}{G, \Gamma \vdash (\texttt{merge}\ x\ \boldsymbol{e_t}\ \boldsymbol{e_f})^{\boldsymbol{\tau}, nck}}$$

$$\frac{\begin{array}{c} \mathsf{types}\, e = [\texttt{bool}] \quad G, \Gamma \vdash e \quad G, \Gamma \vdash \boldsymbol{e_t} \quad G, \Gamma \vdash \boldsymbol{e_f} \\ \mathsf{types}\, \boldsymbol{e_t} = \mathsf{types}\, \boldsymbol{e_f} = \boldsymbol{\tau} \quad \Gamma \vdash nck \end{array}}{G, \Gamma \vdash (\texttt{if}\ e\ \texttt{then}\ \boldsymbol{e_t}\ \texttt{else}\ \boldsymbol{e_f})^{\boldsymbol{\tau}, nck}}$$

$$\frac{G, \Gamma \vdash \boldsymbol{e} \quad \mathsf{node}(G, f) \doteq n \quad \mathsf{types}\, \boldsymbol{e} = \boldsymbol{\tau_x} \quad \boldsymbol{\tau} = \boldsymbol{\tau_y} \quad \Gamma \vdash \boldsymbol{nck}}{G, \Gamma \vdash (f(\boldsymbol{e}))^{\boldsymbol{a}}} \quad \begin{array}{r} \boldsymbol{a} = (\boldsymbol{\tau}, \boldsymbol{nck}) \\ \text{where } n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

$$\frac{\begin{array}{c} G, \Gamma \vdash e_r \quad \mathsf{types}\, e_r = [\texttt{bool}] \\ G, \Gamma \vdash \boldsymbol{e} \quad \mathsf{node}(G, f) \doteq n \quad \mathsf{types}\, \boldsymbol{e} = \boldsymbol{\tau_x} \quad \boldsymbol{\tau} = \boldsymbol{\tau_y} \quad \Gamma \vdash \boldsymbol{nck} \end{array}}{G, \Gamma \vdash ((\texttt{restart}\ f\ \texttt{every}\ e_r)(\boldsymbol{e}))^{\boldsymbol{a}}} \quad \begin{array}{r} \boldsymbol{a} = (\boldsymbol{\tau}, \boldsymbol{nck}) \\ \text{where } n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

(c) Expressions (wt_exp, src/Lustre/LTyping.v:57)

Figure E.1 (I): The type system of Lustre

$$\frac{G, \Gamma \vdash \boldsymbol{e} \quad \Gamma(\boldsymbol{x}) = \mathsf{types}\, \boldsymbol{e}}{G, \Gamma \vdash \boldsymbol{x} = \boldsymbol{e}}$$

(d) Equations (wt_equation, src/Lustre/LTyping.v:260)

$$\frac{\varnothing\{\boldsymbol{x} \mapsto \boldsymbol{\tau_x}\} \vdash \boldsymbol{ck_x} \quad \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{\tau_x}\}\{\boldsymbol{y} \mapsto \boldsymbol{\tau_y}\} \vdash \boldsymbol{ck_y} \quad \Gamma \vdash \boldsymbol{ck_z} \quad G, \Gamma \vdash n.\mathbf{eqs}}{G \vdash_{\mathrm{WT}} n} \quad \text{where} \quad \begin{aligned} n.\mathbf{in} &= \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} &= \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \\ n.\mathbf{vars} &= \boldsymbol{z}^{\boldsymbol{\tau_z}, \boldsymbol{ck_z}} \\ \Gamma &= \varnothing\{\boldsymbol{x} \mapsto \boldsymbol{\tau_x}\}\{\boldsymbol{y} \mapsto \boldsymbol{\tau_y}\}\{\boldsymbol{z} \mapsto \boldsymbol{\tau_z}\} \end{aligned}$$

(e) Nodes (wt_node, src/Lustre/LTyping.v:271)

$$\frac{}{\vdash_{\mathrm{WT}} \varepsilon} \qquad \frac{\vdash_{\mathrm{WT}} G \quad G \vdash_{\mathrm{WT}} n \quad \forall n' \in G,\ n.\mathsf{name} \neq n'.\mathsf{name}}{\vdash_{\mathrm{WT}} n \cdot G}$$

(f) Programs (wt_global, src/Lustre/LTyping.v:278)

Figure E.1 (II): The type system of Lustre

must all be well typed, the instantiated node must exist in the program, and the call input and output type signatures must match the static declaration type signatures. The type checking of a modular reset simply adds to the node instantiation that the reset expression must be well typed and of boolean type.

In sub-figure E.1d, we see that the typing of an equation is straightforward: an equation is well typed if its right hand-side expressions are all well typed and if the variables of the left hand-side are bound to the types of the expressions. The typing of a node, presented in sub-figure E.1e, is more intricate. The input, output and local variables clocks have to be type-checked, but with different typing environments. Input clocks are type-checked over input variables, no matter the order. Then output clocks are typed-checked over input *and* output variables, and finally local clocks are type-checked over all variables, as well as the equations. A program is well typed as described by the sub-figure E.1f if all of its nodes are well typed and have distinct names.

Even if I will not reproduce all the Coq definitions implementing this type system, I still give a piece of it on the listing E.1: the typing of a **`when`** expression. This piece of Coq code shows the definition of the constructor `wt_Ewhen` of the inductive predicate `wt_exp`. This predicate is defined in a *section* in which two variables `G` of type `global` (a program G of category g in the sub-figure E.1c) and `vars` of type `list (ident * type)` (a typing environment Γ) are defined and act as implicit parameters for all subsequent definitions, including `wt_exp` which appears here as a unary predicate over expressions. Note that Γ is implemented as an association list, because it is the simplest map data structure possible and that we do not need performance here since this code is purely

```
    typeof e2 = [ty2] ->
    type_binop op ty1 ty2 = Some ty ->
    wt_nclock nck ->
    wt_exp (Ebinop op e1 e2 (ty, nck))

| wt_Efby: forall e0s es anns,
```

Coq (src/Lustre/LTyping.v:77-82)

Listing E.1: Well typing of a Lustre **when** expression

specification: it is not extracted nor compiled to executable code in the Vélus compiler. Now, let us explain part by part the definition of `wt_Ewhen`, compared to the relevant typing judgment of the sub-figure E.1c:

`Forall wt_exp es` implements $G, \Gamma \vdash_e \boldsymbol{e}$ using the `Forall` predicate from the *List*[1] Coq standard library which is satisfied if its first predicate argument is satisfied by every elements of its second list argument.

`typesof es = tys` corresponds to $\mathsf{types}\,\boldsymbol{e} = \boldsymbol{t}$. The `typesof` function uses the `flat_map` function from the standard library to lift over the function `typeof` which strips the annotation of an expression to the list of its types (both of them are defined in the file src/Lustre/LSyntax.v).

`In (x, bool_type) vars` is a simple lookup in the association list `vars`, implementing $\Gamma(x) = \texttt{bool}$. We use the `In` standard predicate which is satisfied when its first argument actually appears in its second list argument. The type `bool_type` is the one defined in the `OPERATORS` signature (see section 2.1.1).

`wt_nclock nck` encodes the antecedent $\Gamma \vdash_{nck} nck$ asserting that the clock nck is well typed. It uses the inductive predicate `wt_nclock` defined above in the file and implementing the rules of figure E.1b.

`wt_exp (Ewhen es x b (tys, nck))` corresponds to the conclusion of the corresponding rule, $G, \Gamma \vdash_e (\boldsymbol{e}\ \texttt{when}\ (x = b))^{\boldsymbol{\tau}, nck}$, stating that the **when** expression is well typed.

E.2 NLustre

The type system of NLustre is shown as inductive rules on figure E.2. As for Lustre, NLustre is an annotated language, in consequence the well-typed predicate simply checks that the type annotation is consistent. Figures E.2a and E.2b present the type system for expressions an control expressions. Compared to the typing of Lustre expressions (see figure E.1c), the rules are much simpler: (1) expressions are not annotated with

[1] coq.inria.fr/library/Coq.Lists.List.html

$$\frac{}{\Gamma \vdash c} \qquad \frac{}{\Gamma \vdash x^{\Gamma(x)}} \qquad \frac{\Gamma \vdash e \quad \vdash \diamond : \mathsf{type}\, e \to \tau}{\Gamma \vdash (\diamond e)^{\tau}}$$

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2 \quad \vdash \oplus : \mathsf{type}\, e_1 \times \mathsf{type}\, e_2 \to \tau}{\Gamma \vdash (e_1 \oplus e_2)^{\tau}} \qquad \frac{\Gamma(x) = \mathtt{bool} \quad \Gamma \vdash e}{\Gamma \vdash e \ \mathtt{when}\ (x = b)}$$

(a) Expressions (wt_exp, src/CoreExpr/CETyping.v:44)

$$\frac{\Gamma(x) = \mathtt{bool} \quad \Gamma \vdash_c e_t \quad \Gamma \vdash_c e_f \quad \mathsf{type}\, e_t = \mathsf{type}\, e_f}{\Gamma \vdash_c \mathtt{merge}\ x\ e_t\ e_f}$$

$$\frac{\mathsf{type}\, e = \mathtt{bool} \quad \Gamma \vdash e \quad \Gamma \vdash_c e_t \quad \Gamma \vdash_c e_f \quad \mathsf{type}\, e_t = \mathsf{type}\, e_f}{\Gamma \vdash_c \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f} \qquad \frac{\Gamma \vdash e}{\Gamma \vdash_c e}$$

(b) Control expressions (wt_cexp, src/CoreExpr/CETyping.v:71)

$$\frac{\Gamma(x) = \mathsf{type}\, e \quad \Gamma \vdash_c e \quad \Gamma \vdash ck}{G, \Gamma \vdash x =_{ck} e} \qquad \frac{\Gamma(x) = \mathsf{type\text{-}const}\, c = \mathsf{type}\, e \quad \Gamma \vdash e \quad \Gamma \vdash ck}{G, \Gamma \vdash x =_{ck} c\ \mathtt{fby}\ e}$$

$$\frac{\mathsf{node}(G, f) \doteq n \quad \mathsf{type}\, \boldsymbol{e} = \boldsymbol{\tau_x} \quad \Gamma(\boldsymbol{z}) = \boldsymbol{\tau_y} \quad \Gamma \vdash \boldsymbol{e} \quad \Gamma \vdash ck}{G, \Gamma \vdash \boldsymbol{z} =_{ck} f(\boldsymbol{e})} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

$$\frac{\begin{array}{c}\mathsf{node}(G, f) \doteq n \quad \mathsf{type}\, \boldsymbol{e} = \boldsymbol{\tau_x} \quad \Gamma(\boldsymbol{z}) = \boldsymbol{\tau_y} \quad \Gamma \vdash \boldsymbol{e} \\ \Gamma(r) = \mathtt{bool} \quad \Gamma \vdash ck_r \quad \Gamma \vdash ck\end{array}}{G, \Gamma \vdash \boldsymbol{z} =_{ck} (\mathtt{restart}\ f\ \mathtt{every}\ r^{ck_r})(\boldsymbol{e})} \quad \text{where} \quad \begin{array}{l} n.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ n.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

(c) Equations (wt_equation, src/NLustre/NLTyping.v:45)

Figure E.2: The type system of NLustre

$$\frac{\Gamma_{\mathrm{V}}(x) = \mathsf{type}\, e \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash_{\overline{c}} e \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash ck}{P, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash x =_{ck} e}$$

$$\frac{\Gamma_{\mathrm{SV}}(x) = \mathsf{type}\, e \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash e \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash ck}{P, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \texttt{next}\ x =_{ck} e}$$

$$\frac{\mathsf{system}(P, f) \doteq (s, P') \quad \mathsf{type}\, \boldsymbol{e} = \boldsymbol{\tau_x} \quad \Gamma_{\mathrm{V}}(\boldsymbol{z}) = \boldsymbol{\tau_y} \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash \boldsymbol{e} \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash ck}{P, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \boldsymbol{z} =_{ck} f\texttt{<}i, k\texttt{>}(\boldsymbol{e})} \quad \text{where} \quad \begin{array}{l} s.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}, \boldsymbol{ck_x}} \\ s.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}, \boldsymbol{ck_y}} \end{array}$$

$$\frac{\mathsf{system}(P, f) \doteq (s, P') \quad \Gamma_{\mathrm{V}} + \Gamma_{\mathrm{SV}} \vdash ck}{P, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \texttt{reset}\ f\texttt{<}i\texttt{>}\ \texttt{every}\ ck}$$

Figure E.3: Typing rules for transition constraints (`wt_trconstr`, src/Stc/StcTyping.v:42)

clocks anymore, (2) node instantiations do not appear at expression level anymore, and (3) expression can not be lists anymore.

Figure E.2c shows the typing of equations. As NLustre equations are annotated with their activation clocks, contrary to Lustre, the clocks are type-checked at this level, with exactly the same typing rules as for Lustre (see figure E.1a).

The well-typing predicate for a single node is similar to Lustre: it boils down to checking that all the equations of the node are well-typed, using the input, output and local variables declarations as typing environment.

The typing for a program is exactly identical as for Lustre, ensuring that each node in the program is well-typed and that all nodes in the program have distinct names.

E.3 Stc

The type system of NLustre is directly adapted to Stc, in particular because expressions are shared between the two languages. The only major difference is that two environments are used for the typing: Γ_{V} for standard variables and Γ_{SV} for state variables. The reason is to anticipate the translation to imperative Obc code so as to facilitate the well-typing preservation proof.

Figure E.3 presents the well-typing rules of Stc transition constraints. As they are highly similar with the rules for NLustre equations, we do not repeat here the explanations already given.

The well-typing predicate for a system is similar to the well-typing predicate of an NLustre node: all transition constraints must be well-typed, using the input, output and local variables declarations as typing environment for variables and the state variables declarations as typing environment for state variables.

$$\frac{}{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash c} \qquad \frac{}{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash x^{\Gamma_{\mathrm{V}}(x)}} \qquad \frac{}{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \mathtt{state}(x)^{\Gamma_{\mathrm{SV}}(x)}}$$

$$\frac{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash e \quad \vdash \diamond : \mathsf{type}\, e \to \tau}{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash (\diamond e)^{\tau}} \qquad \frac{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash e_1 \quad \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash e_2 \quad \vdash \oplus : \mathsf{type}\, e_1 \times \mathsf{type}\, e_2 \to \tau}{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash (e_1 \oplus e_2)^{\tau}} \qquad \frac{}{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash [x]^{\Gamma_{\mathrm{V}}(x)}}$$

(a) Expressions (`wt_exp`, src/Obc/ObcTyping.v:50)

$$\frac{\Gamma_{\mathrm{V}}(x) = \mathsf{type}\, e}{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash x := e} \qquad \frac{\Gamma_{\mathrm{SV}}(x) = \mathsf{type}\, e}{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \mathtt{state}(x) := e}$$

$$\frac{\Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash e \quad \mathsf{type}\, e = \mathtt{bool} \quad p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash s_1 \quad p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash s_2}{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \mathtt{if}\ e\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}}$$

$$\frac{\begin{array}{c} E_{\mathrm{I}}(i) = c \quad \mathsf{class}(p, c) \doteq (cls, p') \quad \mathsf{method}(cls, f) \doteq m \\ \text{all names in } \boldsymbol{z} \text{ are distinct} \\ \Gamma_{\mathrm{V}}(\boldsymbol{z}) = \boldsymbol{\tau_y} \quad \mathsf{type}\, \boldsymbol{e} = \boldsymbol{\tau_x} \quad \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \boldsymbol{e} \end{array}}{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \boldsymbol{z} := i^{c}.f(\boldsymbol{e})} \quad \text{where} \quad \begin{array}{l} m.\mathbf{in} = \boldsymbol{x}^{\boldsymbol{\tau_x}} \\ m.\mathbf{out} = \boldsymbol{y}^{\boldsymbol{\tau_y}} \end{array}$$

$$\frac{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash s_1 \quad p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash s_2}{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash s_1\ ;\ s_2} \qquad \frac{}{p, E_{\mathrm{I}}, \Gamma_{\mathrm{V}}, \Gamma_{\mathrm{SV}} \vdash \mathtt{skip}}$$

(b) Statements (`wt_stmt`, src/Obc/ObcTyping.v:73)

Figure E.4: Type system of Obc

E.4 Obc

The type system of Obc is rather straightforward, yet as the approach is different from what is done for Stc and NLustre, it is more informative. As for the type system of Stc, the typing of annotated expressions shown in figure E.4a uses two different typing environments Γ_{V} and Γ_{SV} for variables and state variables respectively.

Figure E.4b gives the typing rules for statements. They are all self-explanatory but the rule for method calls. Indeed, compared to the typing rule for Stc default transitions or NLustre node instantiations, we have more structural antecedents:

1. we require the defined variables to be all distinct, and
2. an additional parameter E_{I}, that holds the sub-instances declarations, is used to check that the pair (i, c) is declared.

Both properties were guaranteed syntactically in upstream languages. Apart from the fact that those languages were designed at relatively distant times, one of the reason is

that the structures of systems or nodes in Stc and NLustre, that is lists of transition constraints or equations, are easier to analyze and reason about that arbitrarily nested composed statements in Obc.

To type programs, we introduce the typing of methods and classes.

Definition E.4.1 (wt_method, src/Obc/ObcTyping.v:106)
Given a program p, a sub-instances declarations environment E_I, a typing environment for state variables Γ_{SV} and a method m, we define:

$$p, E_I, \Gamma_{SV} \vdash m \triangleq p, E_I, \Gamma_V, \Gamma_{SV} \vdash m.\textbf{\textit{body}}$$

Where $\Gamma_V = m.\textbf{\textit{in}} + m.\textbf{\textit{vars}} + m.\textbf{\textit{out}}$.

Definition E.4.2 (wt_class, src/Obc/ObcTyping.v:112)
Given a program p and a class cls, we define:

$$\begin{aligned} p \vdash cls \triangleq\ & \forall m \in cls.\textbf{\textit{methods}},\ p, cls.\textbf{\textit{insts}}, cls.\textbf{\textit{regs}} \vdash m \\ & \wedge\ \forall (i, c) \in cls.\textbf{\textit{insts}},\ \exists cls'\, p',\ \mathsf{class}(p, c) \doteq (cls', p') \end{aligned}$$

A method is well-typed under p, E_{I} and Γ_{SV} when its body is well-typed under the same parameters plus a typing environment consisting of its input, local and output variables declarations. A class is well-typed under p if (1) all its methods are well-typed using its sub-instances declarations and registers declarations as sub-instances environment and state variable typing environment, and (2) all classes of its declared sub-instances appear in p. Finally, as for other languages, a program is well-typed if all classes in the program are well-typed and have distinct names.

Bibliography

Martin Abadi and Leslie Lamport (July 1988). "The Existence of Refinement Mappings". In: *Proceedings. Third Annual Symposium on Logic in Computer Science.* Proceedings. Third Annual Symposium on Logic in Computer Science, pp. 165–175. DOI: 10.1109/LICS.1988.5115. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/abadi-existence.pdf (cit. on p. 8).

Andrew W. Appel and Sandrine Blazy (2007). "Separation Logic for Small-Step Cminor". In: *Theorem Proving in Higher Order Logics.* Ed. by Klaus Schneider and Jens Brandt. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 5–21. ISBN: 978-3-540-74591-4. URL: http://web4.ensiie.fr/~blazy/AppelBlazy07.pdf (cit. on p. 156).

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy (2014). *Program Logics for Certified Compilers.* New York, NY, USA: Cambridge University Press. ISBN: 978-1-107-04801-0 (cit. on p. 161).

Cédric Auger (Feb. 7, 2013). "Compilation certifiée de SCADE/LUSTRE". PhD thesis. Université Paris Sud - Paris XI. URL: https://tel.archives-ouvertes.fr/tel-00818169/document (cit. on pp. 7, 11, 12, 38, 41, 82).

Cédric Auger, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet (2012). "A Formalization and Proof of a Modular Lustre Code Generator". Unpublished. (cit. on pp. 11, 25, 31, 36, 42, 104).

Albert Benveniste and Gérard Berry (Sept. 1991). "The Synchronous Approach to Reactive and Real-Time Systems". In: *Proceedings of the IEEE* 79.9, pp. 1270–1282. DOI: 10.1109/5.97297. URL: https://hal.inria.fr/inria-00075115/document (cit. on p. 1).

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone (2003). "The Synchronous Languages 12 Years Later". In: *Proceedings of the Ieee*, pp. 64–83. URL: http://www-verimag.imag.fr/~halbwach/PS/iee03.pdf (cit. on p. 2).

Albert Benveniste and Paul Le Guernic (May 1990). "Hybrid Dynamical Systems Theory and the Signal Language". In: *IEEE Transactions on Automatic Control* 35.5, pp. 535–546. DOI: 10.1109/9.53519. URL: https://hal.inria.fr/inria-00075715/document (cit. on p. 2).

Albert Benveniste, Paul Le Guernic, and Christian Jacquemot (Sept. 1991). "Synchronous Programming with Events and Relations: The SIGNAL Language and Its Semantics". In: *Sci. Comput. Program.* 16.2, pp. 103–149. ISSN: 0167-6423. DOI: 10.1016/0167-6423(91)90001-E. URL: https://core.ac.uk/download/pdf/82752187.pdf (cit. on p. 2).

Gérard Berry (Dec. 16, 2002). "The Constructive Semantics of Pure Esterel". URL: http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf (cit. on p. 2).

– (June 5, 2000a). *The Esterel v5 Language Primer*. Version v5_91. École des Mines and Inria. URL: http://www-sop.inria.fr/members/Gerard.Berry/Papers/primer.zip (cit. on p. 2).

– (2000b). "The Foundations of Esterel". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press. ISBN: 978-0-262-16188-6. URL: http://www-sop.inria.fr/members/Gerard.Berry/Papers/Foundations.zip (cit. on p. 2).

Gérard Berry and Georges Gonthier (Nov. 1, 1992). "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". In: *Science of Computer Programming* 19.2, pp. 87–152. ISSN: 0167-6423. DOI: 10.1016/0167-6423(92)90005-V. URL: https://hal.inria.fr/inria-00075711/document (cit. on p. 2).

Yves Bertot (2005). "CoInduction in Coq". DEA. DEA. EU's coordination action Types Goteborg. URL: https://cel.archives-ouvertes.fr/inria-00001174/document (cit. on p. 26).

Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet (2008). "Clock-Directed Modular Code Generation for Synchronous Data-Flow Languages". In: *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '08. New York, NY, USA: ACM, pp. 121–130. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375674. URL: https://www.di.ens.fr/~pouzet/bib/lctes08a.pdf (cit. on pp. 6, 11, 44, 97, 129).

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy (Aug. 21, 2006). "Formal Verification of a C Compiler Front-End". In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Lecture Notes in Computer Science 4085. Springer Berlin Heidelberg, pp. 460–475. ISBN: 978-3-540-37215-8. DOI: 10.1007/11813040_31. URL: https://xavierleroy.org/publi/cfront.pdf (cit. on p. 6).

Sandrine Blazy and Xavier Leroy (Oct. 1, 2009). "Mechanized Semantics for the Clight Subset of the C Language". In: *Journal of Automated Reasoning* 43.3, pp. 263–288. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9148-3. URL: https://xavierleroy.org/publi/Clight.pdf (cit. on pp. 140, 150, 151, 153, 155).

Sylvain Boulmé and Grégoire Hamon (2001). “Certifying Synchrony for Free”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Robert Nieuwenhuis and Andrei Voronkov. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 495–506. ISBN: 978-3-540-45653-7. DOI: 10.1007/3-540-45653-8_34. URL: https://www.di.ens.fr/~pouzet/bib/lpar01.ps.gz (cit. on p. 7).

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg (June 2017). “A Formally Verified Compiler for Lustre”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI’17. New York, NY, USA: ACM, pp. 586–601. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062358. URL: https://www.leliobrun.net/publication/pldi17/paper.pdf (cit. on pp. 19, 73, 85, 97, 98, 104, 112).

Timothy Bourke, Lélio Brun, and Marc Pouzet (Jan. 2020). “Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset”. In: *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages*. Principles Of Programming Languages. Vol. 4. POPL’20. New Orleans, LA, USA: Association for Computing Machinery, p. 29. DOI: 10.1145/3371112. URL: https://www.leliobrun.net/publication/popl20/paper.pdf (cit. on p. 19).

– (May 2018). “Towards a Verified Lustre Compiler with Modular Reset”. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’18. ACM, pp. 14–17. ISBN: 978-1-4503-5780-7. DOI: 10.1145/3207719.3207732. URL: https://www.leliobrun.net/publication/scopes18/paper.pdf (cit. on p. 19).

Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet (Apr. 2015). “A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages”. In: *24th International Conference on Compiler Construction*. CC 2015. London, UK, pp. 69–88. URL: http://zelus.di.ens.fr/cc2015/paper.pdf (cit. on p. 2).

Timothy Bourke and Marc Pouzet (Jan. 2019). “Arguments Cadencés Dans Un Compilateur Lustre Vérifié”. In: *JFLA 2019 - Les Trentièmes Journées Francophones Des Langages Applicatifs*. Les Actes Des Trentièmes Journées Francophones Des Langages Applicatifs (JFLA 2019), p. 16. URL: https://www.di.ens.fr/~pouzet/bib/jfla19-velus.pdf (cit. on pp. 49, 101, 121, 138, 139).

– (Mar. 2013). “Zélus: A Synchronous Language with ODEs”. In: *16th International Conference on Hybrid Systems: Computation and Control*. HSCC’13. Philadelphia, USA, pp. 113–118. URL: http://www.di.ens.fr/~pouzet/bib/hscc13.pdf (cit. on p. 2).

Frédéric Boussinot (Apr. 1991). “Reactive C: An Extension of C to Program Reactive Systems”. In: *Softw. Pract. Exper.* 21.4, pp. 401–428. ISSN: 0038-0644. DOI: 10.1002/spe.4380210406 (cit. on p. 2).

Frédéric Boussinot and Robert de Simone (Sept. 1991). "The ESTEREL Language". In: *Proceedings of the IEEE* 79.9, pp. 1293–1304. DOI: 10.1109/5.97299. URL: https://hal.inria.fr/inria-00075075/document (cit. on p. 2).

Paul Caspi (Jan. 1, 1994). "Towards Recursive Block Diagrams". In: *Annual Review in Automatic Programming* 18, pp. 81–85. ISSN: 0066-4138. DOI: 10.1016/0066-4138(94)90015-9 (cit. on p. 9).

Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond (2009). "Synchronous Objects with Scheduling Policies: Introducing Safe Shared Memory in Lustre". In: *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Dublin, Ireland). LCTES '09. New York, NY, USA: ACM, pp. 11–20. ISBN: 978-1-60558-356-3. DOI: 10.1145/1542452.1542455. URL: https://www.di.ens.fr/~pouzet/bib/lctes09.pdf (cit. on p. 73).

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Alexander Plaice (1987). "LUSTRE: A Declarative Language for Programming Synchronous Systems". In: *In 14th Symposium on Principles of Programming Languages (POPL'87). ACM.* URL: https://www.cse.unsw.edu.au/~plaice/archive/JAP/P-ACM_POPL87-lustre.pdf (cit. on pp. 1, 2, 5, 9, 22, 25, 27, 47).

Paul Caspi and Marc Pouzet (Oct. 1997). *A Co-Iterative Characterization of Synchronous Stream Functions*. 07. VERIMAG. URL: https://www.di.ens.fr/~pouzet/bib/coiteration-report98.ps.gz (cit. on pp. 11, 41, 52).

– (Jan. 1, 1998). "A Co-Iterative Characterization of Synchronous Stream Functions". In: *Electronic Notes in Theoretical Computer Science*. CMCS '98, First Workshop on Coalgebraic Methods in Computer Science 11, pp. 1–21. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)00050-7. URL: https://www.di.ens.fr/~pouzet/bib/cmcs98.ps.gz (cit. on pp. 31, 36).

– (1996). "Synchronous Kahn Networks". In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP '96. New York, NY, USA: ACM, pp. 226–238. ISBN: 978-0-89791-770-4. DOI: 10.1145/232627.232651. URL: https://www.di.ens.fr/~pouzet/bib/icfp96.ps.gz (cit. on p. 47).

Albert Cohen, Léonard Gérard, and Marc Pouzet (2012). "Programming Parallelism with Futures in Lustre". In: *Proceedings of the Tenth ACM International Conference on Embedded Software* (Tampere, Finland). EMSOFT '12. New York, NY, USA: ACM, pp. 197–206. ISBN: 978-1-4503-1425-1. DOI: 10.1145/2380356.2380394. URL: https://www.di.ens.fr/~pouzet/bib/emsoft12.pdf (cit. on p. 11).

Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet (2004). "Towards a Higher-Order Synchronous Data-Flow Language". In: *Proceedings of the 4th ACM International Conference on Embedded Software* (Pisa, Italy). EMSOFT '04. New York, NY, USA: ACM, pp. 230–239.

ISBN: 978-1-58113-860-3. DOI: 10.1145/1017753.1017792.
URL: https://www.di.ens.fr/~pouzet/bib/emsoft04.pdf (cit. on p. 215).
Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet (2006).
"Mixing Signals and Modes in Synchronous Data-Flow Systems". In: *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. EMSOFT '06. New York, NY, USA: ACM, pp. 73–82. ISBN: 978-1-59593-542-7.
DOI: 10.1145/1176887.1176899.
URL: https://www.di.ens.fr/~pouzet/bib/emsoft06.pdf (cit. on p. 11).
Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet (2005).
"A Conservative Extension of Synchronous Data-Flow with State Machines".
In: *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT '05. New York, NY, USA: ACM, pp. 173–182. ISBN: 978-1-59593-091-0.
DOI: 10.1145/1086228.1086261.
URL: https://www.di.ens.fr/~pouzet/bib/emsoft05b.pdf (cit. on pp. 11, 191, 215).
– (Sept. 2017).
"SCADE 6: A Formal Language for Embedded Critical Software Development".
In: *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 1–11. DOI: 10.1109/TASE.2017.8285623.
URL: https://www.di.ens.fr/~pouzet/bib/tase17.pdf
(cit. on pp. 1, 2, 4, 27, 28, 191).
Jean-Louis Colaço and Marc Pouzet (Oct. 13, 2003).
"Clocks as First Class Abstract Types". In: *Embedded Software*.
International Workshop on Embedded Software. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 134–155. ISBN: 978-3-540-45212-6.
DOI: 10.1007/978-3-540-45212-6_10.
URL: https://www.di.ens.fr/~pouzet/bib/emsoft03.ps.gz
(cit. on pp. 25, 31, 32, 36, 38, 49).
– (Aug. 1, 2004).
"Type-Based Initialization Analysis of a Synchronous Dataflow Language".
In: *International Journal on Software Tools for Technology Transfer* 6.3, pp. 245–255.
ISSN: 1433-2787. DOI: 10.1007/s10009-004-0160-y.
URL: https://www.di.ens.fr/~pouzet/bib/sttt04.pdf (cit. on pp. 29, 191).
Solange Coupet-Grimal and Line Jakubiec (1999).
"Hardware Verification Using Co-Induction in COQ". In: *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '99.
London, UK, UK: Springer-Verlag, pp. 91–108. ISBN: 978-3-540-66463-5 (cit. on p. 7).
Maulik A. Dave (Nov. 2003). "Compiler Verification: A Bibliography".
In: *SIGSOFT Softw. Eng. Notes* 28.6, pp. 2–2. ISSN: 0163-5948.
DOI: 10.1145/966221.966235.
URL: http://www.cs.utah.edu/~skchoe/research/p2-dave.pdf (cit. on p. 6).
Gwenaël Delaval, Adrien Guatto, Hervé Marchand, Marc Pouzet, and Rutten Éric (Apr. 3, 2017). *Heptagon/BZR manual*. PARKAS (ENS) and Ctrl-A (LIG/Inria).
URL: http://heptagon.gforge.inria.fr/pub/heptagon-manual.pdf (cit. on p. 191).

François Xavier Dormoy (Jan. 2008).
"SCADE 6: A Model Based Solution For Safety Critical Software Development".
In: *Embedded Real Time Software and Systems*. ERTS. Toulouse.
URL: https://hal-insu.archives-ouvertes.fr/insu-02270108/document (cit. on p. 4).

Eric Eide and John Regehr (2008).
"Volatiles Are Miscompiled, and What to Do About It".
In: *Proceedings of the 8th ACM International Conference on Embedded Software* (Atlanta, GA, USA). EMSOFT '08. New York, NY, USA: ACM, pp. 255–264.
ISBN: 978-1-60558-468-3. DOI: 10.1145/1450058.1450093.
URL: http://www.llvm.org/pubs/2008-10-EMSOFT-Volatiles.pdf (cit. on p. 8).

Robert W. Floyd (1967). "Assigning Meanings to Programs".
In: *Proceedings of Symposium on Applied Mathematics* 19, pp. 19–32.
URL: http://www.cse.chalmers.se/edu/year/2017/course/TDA384_LP1/files/lectures/additional-material/AssigningMeanings1967.pdf (cit. on p. 159).

Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet (2012).
"A Modular Memory Optimization for Synchronous Data-Flow Languages: Application to Arrays in a Lustre Compiler".
In: *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. LCTES '12.
Beijing, China: ACM, pp. 51–60. ISBN: 978-1-4503-1212-7.
DOI: 10.1145/2248418.2248426.
URL: https://www.di.ens.fr/~pouzet/bib/lctes12.pdf (cit. on pp. 190, 191).

Eduardo Giménez (1996). "An Application of Co-Inductive Types in Coq: Verification of the Alternating Bit Protocol". In: *Types for Proofs and Programs*.
Ed. by Stefano Berardi and Mario Coppo. Lecture Notes in Computer Science.
Springer Berlin Heidelberg, pp. 135–152. ISBN: 978-3-540-70722-6 (cit. on p. 26).

– (1995). "Codifying Guarded Definitions with Recursive Schemes".
In: *Types for Proofs and Programs*.
Ed. by Peter Dybjer, Bengt Nordström, and Jan Smith.
Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 39–59.
ISBN: 978-3-540-47770-9 (cit. on p. 26).

Eduardo Giménez and Emmanuel Ledinot (Jan. 2000). *Certification de SCADE V3*.
Rapport final du projet GENIE II. Verilog SA (cit. on p. 7).

Georges Gonthier (1988). "Sémantiques et Modèles d'exécution Des Langages Réactifs Synchrones : Application à Esterel". PhD thesis. Paris 11 (cit. on p. 5).

Nicolas Halbwachs (July 2005).
"A Synchronous Language at Work: The Story of Lustre".
In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*
Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Pp. 3–11.
DOI: 10.1109/MEMCOD.2005.1487884.
URL: http://www-verimag.imag.fr/~halbwachs/PS/memocode05.pdf
(cit. on pp. 2, 8).

– (1993). *Synchronous Programming of Reactive Systems*.
The Springer International Series in Engineering and Computer Science. Springer US.
ISBN: 978-0-7923-9311-5.
URL: http://www-verimag.imag.fr/~halbwach/newbook.pdf (cit. on p. 1).
Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud (Sept. 1991).
"The Synchronous Data Flow Programming Language LUSTRE".
In: *Proceedings of the IEEE* 79.9, pp. 1305–1320. ISSN: 0018-9219.
DOI: 10.1109/5.97300.
URL: http://www-verimag.imag.fr/~raymond/publis/lustre.ieee.ps.gz
(cit. on p. 2).
Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel (1991).
"Generating Efficient Code from Data-Flow Programs".
In: *Programming Language Implementation and Logic Programming*.
Ed. by Jan Maluszyński and Martin Wirsing. PLIP'91. Berlin, Heidelberg: Springer,
pp. 207–218. ISBN: 978-3-540-38362-8. DOI: 10.1007/3-540-54444-5_100.
URL: http://www-verimag.imag.fr/PEOPLE/Nicolas.Halbwachs/PS/plilp.ps
(cit. on pp. 5, 190).
Grégoire Hamon (2005). "A Denotational Semantics for Stateflow".
In: *Proceedings of the 5th ACM International Conference on Embedded Software*
(Jersey City, NJ, USA). EMSOFT '05. New York, NY, USA: ACM, pp. 164–172.
ISBN: 978-1-59593-091-0. DOI: 10.1145/1086228.1086260.
URL: https://www.cs.york.ac.uk/rts/docs/EMSOFT-2004-2005/docs05/p164.pdf
(cit. on p. 7).
Grégoire Hamon and Marc Pouzet (2000).
"Modular Resetting of Synchronous Data-Flow Programs".
In: *Proceedings of the 2Nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '00. New York, NY, USA: ACM,
pp. 289–300. ISBN: 978-1-58113-265-6. DOI: 10.1145/351268.351300.
URL: https://www.di.ens.fr/~pouzet/bib/ppdp00.ps.gz
(cit. on pp. 1, 9, 11, 38, 39, 41).
Grégoire Hamon and John Rushby (2004). "An Operational Semantics for Stateflow".
In: *Fundamental Approaches to Software Engineering*.
Ed. by Michel Wermelinger and Tiziana Margaria-Steffen.
Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 229–243.
ISBN: 978-3-540-24721-0. DOI: 10.1007/978-3-540-24721-0_17.
URL: http://www.sdl.sri.com/~rushby/papers/sttt07.pdf (cit. on p. 7).
David Harel (June 1987). "Statecharts: A Visual Formalism for Complex Systems".
In: *Sci. Comput. Program.* 8.3, pp. 231–274. ISSN: 0167-6423.
DOI: 10.1016/0167-6423(87)90035-9. URL: http:
//www.inf.ed.ac.uk/teaching/courses/seoc/2005_2006/resources/statecharts.pdf
(cit. on p. 11).
David Harel and Amir Pnueli (1985). "On the Development of Reactive Systems".
In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt.
NATO ASI Series. Springer Berlin Heidelberg, pp. 477–498. ISBN: 978-3-642-82453-1.

URL: http://www.wisdom.weizmann.ac.il/~/dharel/SCANNED.PAPERS/ReactiveSystems.pdf (cit. on p. 1).

Charles Antony Richard Hoare (Oct. 1969). "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10, pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf (cit. on p. 159).

Samin S. Ishtiaq and Peter W. O'Hearn (Jan. 1, 2001). "BI as an Assertion Language for Mutable Data Structures". In: *ACM SIGPLAN Notices* 36.3, pp. 14–26. ISSN: 0362-1340. DOI: 10.1145/360204.375719. URL: http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/bi-assertion-lan.pdf (cit. on p. 159).

Nassima Izerrouken (July 6, 2011). "Développement prouvé de composants formels pour un générateur de code embarqué critique pré-qualifié". PhD thesis. URL: https://oatao.univ-toulouse.fr/7135/1/izerrouken.pdf (cit. on p. 7).

Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs (2019). *The Lustre V6 Reference Manual*. Version 04-09-19. Verimag. URL: http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf (cit. on p. 2).

Paul Jeanmaire (Sept. 2019). "Propriétés Dynamiques Du Système d'horloges de Lustre". MA thesis. ENS Paris-Saclay (cit. on pp. 42, 53).

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy (2012). "Validating LR(1) Parsers". In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 397–416. ISBN: 978-3-642-28869-2. URL: http://gallium.inria.fr/~fpottier/publis/jourdan-leroy-pottier-validating-parsers.pdf (cit. on p. 12).

Arthur B. Kahn (Nov. 1962). "Topological Sorting of Large Networks". In: *Commun. ACM* 5.11, pp. 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025 (cit. on p. 112).

Gilles Kahn (1987). "Natural Semantics". In: *STACS 87*. Ed. by Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 22–39. ISBN: 978-3-540-47419-7. URL: https://www-sop.inria.fr/teams/parsifal/Joelle.Despeyroux.old/papers/stacs87.ps (cit. on p. 98).

– (1974). "The Semantics of Simple Language for Parallel Programming". In: *IFIP Congress*, pp. 471–475 (cit. on p. 2).

Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy (Jan. 2018). "CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler". In: *ERTS 2018: Embedded Real Time Software and Systems*.

ERTS'18. SEE. URL: http://xavierleroy.org/publi/erts2018_compcert.pdf (cit. on p. 193).
Brian Wilson Kernighan and Dennis Ritchie (Mar. 22, 1988). *The C Programming Language*. 2nd ed. Englewood Cliffs, N.J: Prentice Hall. 288 pp. ISBN: 978-0-13-110362-7. URL: https://www.dipmat.univpm.it/~demeio/public/the_c_programming_language_2.pdf (cit. on p. 140).
Gerwin Klein, Kevin Elphinstone, et al. (Oct. 11, 2009). "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://ts.data61.csiro.au/publications/nicta_full_text/1852.pdf (cit. on p. 9).
Gerwin Klein, Rafal Kolanski, and Andrew Boyton (2012). "Mechanised Separation Algebra". In: *Interactive Theorem Proving*. Ed. by Lennart Beringer and Amy Felty. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 332–337. ISBN: 978-3-642-32347-8. DOI: 10.1007/978-3-642-32347-8_22. URL: https://ts.data61.csiro.au/publications/nicta_full_text/5676.pdf (cit. on p. 161).
Gerwin Klein and Tobias Nipkow (July 2006). "A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler". In: *ACM Trans. Program. Lang. Syst.* 28.4, pp. 619–695. ISSN: 0164-0925. DOI: 10.1145/1146809.1146811. URL: http://ts.data61.csiro.au/publications/papers/Klein_Nipkow_06.pdf (cit. on p. 6).
Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens (Jan. 2014). "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages (POPL)*. ACM Press, pp. 179–191. DOI: 10.1145/2535838.2535841. URL: https://cakeml.org/popl14.pdf (cit. on p. 7).
Vu Le, Mehrdad Afshari, and Zhendong Su (2014). "Compiler Validation via Equivalence Modulo Inputs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom). PLDI '14. New York, NY, USA: ACM, pp. 216–226. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594334. URL: http://vuminhle.com/pdf/pldi14-emi.pdf (cit. on p. 8).
Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire (Sept. 1991). "Programming Real-Time Applications with SIGNAL". In: *Proceedings of the IEEE* 79.9, pp. 1321–1336. DOI: 10.1109/5.97301. URL: https://hal.inria.fr/inria-00540460/document (cit. on p. 2).
François Leclerc and Christine Paulin-Mohring (1994). "Programming with Streams in Coq a Case Study: The Sieve of Eratosthenes". In: *Types for Proofs and Programs*. Ed. by Henk Barendregt and Tobias Nipkow.

Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 191–212. ISBN: 978-3-540-48440-0 (cit. on p. 26).

Dirk Leinenbach, Wolfgang Paul, and Elena Petrova (2005). "Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctnes". In: *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. SEFM '05. Washington, DC, USA: IEEE Computer Society, pp. 2–12. ISBN: 978-0-7695-2435-1. DOI: 10.1109/SEFM.2005.51. URL: http://www-wjp.cs.uni-saarland.de/publikationen/LeinenbachSEFM05.pdf (cit. on p. 6).

Xavier Leroy (Nov. 4, 2009a). "A Formally Verified Compiler Back-End". In: *Journal of Automated Reasoning* 43.4, p. 363. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9155-4. URL: https://xavierleroy.org/publi/compcert-backend.pdf (cit. on pp. 9, 156).

– (2006). "Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA). POPL '06. New York, NY, USA: ACM, pp. 42–54. ISBN: 978-1-59593-027-9. DOI: 10.1145/1111037.1111042. URL: https://xavierleroy.org/publi/compiler-certif.pdf (cit. on p. 6).

– (July 2009b). "Formal Verification of a Realistic Compiler". In: *Communications of the ACM* 52.7, pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://xavierleroy.org/publi/compcert-CACM.pdf (cit. on pp. 1, 6, 8, 192).

– (Feb. 27, 2019). *The CompCert verified compiler. Commented Coq development*. Version 3.5. URL: http://compcert.inria.fr/doc/ (cit. on pp. 24, 140, 142, 150, 151, 160, 161).

Xavier Leroy and Sandrine Blazy (July 1, 2008). "Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations". In: *Journal of Automated Reasoning* 41.1, pp. 1–31. ISSN: 1573-0670. DOI: 10.1007/s10817-008-9099-0. URL: http://xavierleroy.org/publi/memory-model-journal.pdf (cit. on pp. 150, 158).

Xavier Leroy and Hervé Grall (Feb. 2009). "Coinductive Big-Step Operational Semantics". In: *Inf. Comput.* 207.2, pp. 284–304. ISSN: 0890-5401. DOI: 10.1016/j.ic.2007.12.004. URL: https://xavierleroy.org/publi/coindsem-journal.pdf (cit. on p. 31).

Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis (2009). "Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: Association for Computing Machinery, pp. 78–89. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480893. URL: http://www-verimag.imag.fr/~tripakis/papers/popl09.pdf (cit. on p. 6).

Nancy Lynch and Frits Vaandrager (Aug. 1994).
Forward and Backward Simulations — Part I: Untimed Systems.
MIT Technical Memo MIT/LCS/TM-486.b. Cambridge, MA, USA: Laboratory for Computer Science, Massachusetts Institute of Technology.
URL: http://groups.csail.mit.edu/tds/papers/Lynch/IC95.pdf (cit. on p. 9).

Louis Mandel, Cédric Pasteur, and Marc Pouzet (July 2015).
"ReactiveML, Ten Years Later". In: *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*. Siena, Italy.
URL: https://www.di.ens.fr/~pouzet/bib/ppdp15.pdf (cit. on p. 2).

Louis Mandel and Marc Pouzet (July 2005). "ReactiveML, a Reactive Extension to ML".
In: *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*. Lisboa.
URL: https://www.di.ens.fr/~pouzet/bib/ppdp05.pdf (cit. on p. 2).

Florence Maraninchi (1992).
"Operational and Compositional Semantics of Synchronous Automaton Compositions".
In: *Proceedings of the Third International Conference on Concurrency Theory.*
CONCUR '92. London, UK, UK: Springer-Verlag, pp. 550–564.
ISBN: 978-3-540-55822-4 (cit. on p. 11).

– (1991). "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems". In: (cit. on pp. 2, 11).

Florence Maraninchi and Yann Rémond (Apr. 2001).
"Argos: An Automaton-Based Synchronous Language".
In: *Comput. Lang.* 27.1-3, pp. 61–92. ISSN: 0096-0551.
DOI: 10.1016/S0096-0551(01)00016-9.
URL: https://hal.archives-ouvertes.fr/hal-00273055/document (cit. on p. 2).

– (Mar. 1, 2003). "Mode-Automata: A New Domain-Specific Construct for the Development of Safe Critical Systems". In: *Science of Computer Programming.* Special Issue on Formal Methods for Industrial Critical Systems 46.3, pp. 219–254.
ISSN: 0167-6423. DOI: 10.1016/S0167-6423(02)00093-X (cit. on p. 11).

– (Mar. 28, 1998). "Mode-Automata: About Modes and States for Reactive Systems".
In: *Programming Languages and Systems.* European Symposium on Programming.
Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 185–199.
ISBN: 978-3-540-64302-9. DOI: 10.1007/BFb0053571 (cit. on p. 11).

John McCarthy and James Painter (1967).
"Correctness of a Compiler for Arithmetic Expressions".
In: *Proceedings of Symposia in Applied Mathematics.* Vol. XIX.
Amer. Math. Soc., Providence, R.I., pp. 33–41.
DOI: http://dx.doi.org/10.1090/psapm/019/0242403.
URL: http://jmc.stanford.edu/articles/mcpain/mcpain.pdf (cit. on p. 6).

George C. Necula (1997). "Proof-Carrying Code". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*
(Paris, France). POPL '97. New York, NY, USA: ACM, pp. 106–119.
ISBN: 978-0-89791-853-4. DOI: 10.1145/263699.263712.

URL: https://people.eecs.berkeley.edu/~necula/Papers/pcc_popl97.ps (cit. on p. 7).

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis (2015). "Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada). ICFP 2015. New York, NY, USA: ACM, pp. 166–178. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784764.
URL: https://people.mpi-sws.org/~viktor/papers/pilsner.pdf (cit. on p. 7).

Peter W. O'Hearn (Jan. 2019). "Separation Logic". In: *Commun. ACM* 62.2, pp. 86–95. ISSN: 0001-0782. DOI: 10.1145/3211968.
URL: https://cacm.acm.org/magazines/2019/2/234356-separation-logic/fulltext (cit. on p. 160).

Peter W. O'Hearn and David J. Pym (June 1999). "The Logic of Bunched Implications". In: *Bulletin of Symbolic Logic* 5.2, pp. 215–244. ISSN: 1079-8986, 1943-5894. DOI: 10.2307/421090. URL: http://www.lsv.fr/~demri/OHearnPym99.pdf (cit. on p. 160).

Christine Paulin-Mohring (2009). "A Constructive Denotational Semantics for Kahn Networks in Coq". In: *From Semantics to Computer Science*. Ed. by Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin. Cambridge University Press, pp. 383–413.
URL: https://hal.inria.fr/inria-00431806/document (cit. on p. 7).

– (1996). "Circuits as Streams in Coq: Verification of a Sequential Multiplier". In: *Types for Proofs and Programs*. Ed. by Stefano Berardi and Mario Coppo. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 216–230. ISBN: 978-3-540-70722-6.
URL: ftp://ftp.ens-lyon.fr/pub/LIP/COQ/Publications/mult.ps.gz (cit. on pp. 7, 26).

Benjamin Crawford Pierce (Feb. 1, 2002). *Types and Programming Languages*. 1 edition. Cambridge, Mass: The MIT Press. 645 pp. ISBN: 978-0-262-16209-8.
URL: https://www.cis.upenn.edu/~bcpierce/tapl/ (cit. on p. 209).

John Plaice (May 1988). "Sémantique et Compilation de LUSTRE, Un Langage Déclaratif Synchrone". PhD thesis. Grenoble INPG.
URL: http://www.cse.unsw.edu.au/~plaice/archive/JAP/M-88-JAP_PhD.pdf (cit. on p. 5).

Amir Pnueli, Michael Siegel, and Eli Singerman (1998). "Translation Validation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 151–166. ISBN: 978-3-540-69753-4 (cit. on p. 7).

Amir Pnueli, Ofer Strichman, and Michael Siegel (1998). "Translation Validation for Synchronous Languages". In: *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*. ICALP '98.

Berlin, Heidelberg: Springer-Verlag, pp. 235–246. ISBN: 978-3-540-64781-2.
URL: https://cs.nyu.edu/faculty/pnueli/transval-icalp98.ps.gz (cit. on p. 8).
– (1999). "Translation Validation: From SIGNAL to C".
In: *Correct System Design: Recent Insights and Advances*.
Ed. by Ernst-Rüdiger Olderog and Bernhard Steffen.
Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg,
pp. 231–255. ISBN: 978-3-540-48092-1. DOI: 10.1007/3-540-48092-7_11.
URL: https://cs.nyu.edu/faculty/pnueli/pss00.pdf (cit. on p. 8).
François Pottier and Yann Régis-Gianas (2018). *Menhir Reference Manual*.
Version 20181113. Inria. URL: http://gallium.inria.fr/~fpottier/menhir/manual.pdf
(cit. on p. 12).
Marc Pouzet (Apr. 2006). *Lucid Synchrone. Tutorial and reference manual*. Version 3.
Université Paris-Sud, LRI. URL:
https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf
(cit. on p. 2).
Marc Pouzet and Pascal Raymond (2009). "Modular Static Scheduling of Synchronous
Data-Flow Networks: An Efficient Symbolic Representation".
In: *Proceedings of the Seventh ACM International Conference on Embedded Software*.
EMSOFT '09. Grenoble, France: Association for Computing Machinery, pp. 215–224.
ISBN: 978-1-60558-627-4. DOI: 10.1145/1629335.1629365.
URL: https://www.di.ens.fr/~pouzet/bib/emsoft09.pdf (cit. on p. 6).
Pascal Raymond (Nov. 20, 1991). "Compilation Efficace d'un Langage Déclaratif
Synchrone : Le Générateur de Code Lustre-V3". PhD thesis. Grenoble INPG.
URL: https://tel.archives-ouvertes.fr/tel-00198546/document (cit. on p. 5).
– (1988). *Compilation Séparée de Programmes Lustre*. DEA report 5.
SPECTRE Project, IMAG, Université Joseph Fourier.
URL: http://www-verimag.imag.fr/~raymond/publis/spectre-L5.pdf
(cit. on pp. 5, 6).
John C. Reynolds (2002).
"Separation Logic: A Logic for Shared Mutable Data Structures".
In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*.
LICS '02. Washington, DC, USA: IEEE Computer Society, pp. 55–74.
ISBN: 978-0-7695-1483-3. URL: https://www.cs.cmu.edu/~jcr/seplogic.pdf
(cit. on p. 159).
Michael Ryabtsev and Ofer Strichman (2009).
"Translation Validation: From Simulink to C". In: *Computer Aided Verification*.
Ed. by Ahmed Bouajjani and Oded Maler. Lecture Notes in Computer Science.
Berlin, Heidelberg: Springer, pp. 696–701. ISBN: 978-3-642-02658-4.
DOI: 10.1007/978-3-642-02658-4_57.
URL: https://ie.technion.ac.il/~ofers/publications/cav09_simulink.pdf
(cit. on p. 8).
Klaus Schneider (2001).
"Embedding Imperative Synchronous Languages in Interactive Theorem Provers".
In: *Proceedings of the Second International Conference on Application of Concurrency*

to System Design. ACSD '01. Washington, DC, USA: IEEE Computer Society, pp. 143–. ISBN: 978-0-7695-1071-2.
URL: https://es.cs.uni-kl.de/publications/datarsg/Schn01a.pdf (cit. on p. 7).

Robert Seacord (2018).
INT32-C. Ensure that operations on signed integers do not result in overflow.
Ed. by Software Engineering Institute. Carnegie Mellon University.
URL: https://wiki.sei.cmu.edu/confluence/x/UtYxBQ (cit. on p. 24).

Gang Shi, Yuanke Gan, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew (2017). "A Formally Verified Sequentializer for Lustre-like Concurrent Synchronous Data-Flow Programs". In: *Proceedings of the 39th International Conference on Software Engineering Companion* (Buenos Aires, Argentina). ICSE-C '17. Piscataway, NJ, USA: IEEE Press, pp. 109–111. ISBN: 978-1-5386-1589-8.
DOI: 10.1109/ICSE-C.2017.83 (cit. on p. 8).

Gang Shi, Yucheng Zhang, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew (Jan. 2019). "A Formally Verified Transformation to Unify Multiple Nested Clocks for a Lustre-like Language".
In: *Science China Information Sciences* 62.1, p. 12801. ISSN: 1869-1919.
DOI: 10.1007/s11432-016-9270-0.
URL: http://scis.scichina.com/en/2019/012801.pdf (cit. on p. 8).

Matthieu Sozeau (2007). "Subset Coercions in Coq". In: *Types for Proofs and Programs*. Ed. by Thorsten Altenkirch and Conor McBride. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 237–252. ISBN: 978-3-540-74464-1. URL: https://www.irif.fr/~sozeau/research/publications/Subset_Coercions_in_Coq.pdf (cit. on p. 31).

– (Dec. 1, 2008). "Un environnement pour la programmation avec types dépendants". PhD thesis. Paris 11.
URL: https://www.irif.fr/~sozeau/research/publications/thesis-sozeau.pdf (cit. on p. 31).

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish (Sept. 2016). "A New Verified Compiler Backend for CakeML".
In: *International Conference on Functional Programming (ICFP)*. ACM Press, pp. 60–73. DOI: 10.1145/2951913.2951924. URL: https://cakeml.org/icfp16.pdf (cit. on p. 7).

The Coq Development Team (Jan. 18, 2019). *The Coq Proof Assistant Reference Manual*. Version 8.9.0. URL: https://coq.inria.fr/distrib/V8.9.0/refman/ (cit. on p. 1).

Andres Toom, Nassima Izerrouken, Tõnu Näks, Marc Pantel, and Olivier Ssi Yan Kai (May 2010). "Towards Reliable Code Generation with an Open Tool: Evolutions of the Gene-Auto Toolset". In: *ERTS2 2010, Embedded Real Time Software & Systems*. Toulouse, France. URL: https://hal.archives-ouvertes.fr/hal-02267640/document (cit. on p. 7).

Andres Toom, Tõnu Näks, Marc Pantel, Marcel Gandriau, and I. Wati (2008). "Gene-Auto: An Automatic Code Generator for a Safe Subset of Simulink/Stateflow and Scicos". In: *Embedded Real Time Software and Systems (ERTS2008)*.

Toulouse, France. URL: https://hal.archives-ouvertes.fr/hal-02270306/document (cit. on p. 7).

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien (2014). "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications: HW/SW-Synthesis for a Conservative Extension of Synchronous Statecharts". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom). PLDI '14. New York, NY, USA: ACM, pp. 372–383. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594310. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/pldi14.pdf (cit. on p. 2).

William W. Wadge and Edward A. Ashcroft (1985). *LUCID, the Dataflow Programming Language*. San Diego, CA, USA: Academic Press Professional, Inc. ISBN: 978-0-12-729650-0. URL: http://www.cse.unsw.edu.au/~plaice/archive/WWW/1985/B-AP85-LucidDataflow.pdf (cit. on p. 2).

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr (2011). "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA). PLDI '11. New York, NY, USA: ACM, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532. URL: https://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf (cit. on p. 8).

Zhibin Yang, Jean-Paul Bodeveix, Mamoun Filali, Kai Hu, Yongwang Zhao, and Dianfu Ma (Feb. 1, 2016). "Towards a Verified Compiler Prototype for the Synchronous Language SIGNAL". In: *Frontiers of Computer Science* 10.1, pp. 37–53. ISSN: 2095-2236. DOI: 10.1007/s11704-015-4364-y. URL: https://hal.archives-ouvertes.fr/hal-01298793/document (cit. on p. 7).

RÉSUMÉ

Les spécifications basées sur les schémas-blocs et machines à états sont utilisées pour la conception de systèmes de contrôle-commande, particulièrement dans le développement d'applications critiques. Des outils tels que Scade et Simulink/Stateflow sont équipés de compilateurs qui traduisent de telles spécifications en code exécutable. Ils proposent des langages de programmation permettant de composer des fonctions sur des flots, tel que l'illustre le langage synchrone à flots de données Lustre.
Cette thèse présente Vélus, un compilateur Lustre vérifié dans l'assistant de preuves interactif Coq. Nous développons des modèles sémantiques pour les langages de la chaîne de compilation, et utilisons le compilateur C vérifié CompCert pour générer du code exécutable et donner une preuve de correction de bout en bout. Le défi principal est de montrer la préservation de la sémantique entre le paradigme flots de données et le paradigme impératif, et de raisonner sur la représentation bas niveau de l'état d'un programme.
En particulier, nous traitons le reset modulaire, une primitive pour réinitialiser des sous-systèmes. Ceci implique la mise en place de modèles sémantiques adéquats, d'algorithmes de compilation et des preuves de correction correspondantes. Nous présentons un nouveau langage intermédiaire dans le schéma habituel de compilation modulaire dirigé par les horloges de Lustre. Ceci débouche sur l'implémentation de passes de compilation permettant de générer un meilleur code séquentiel, et facilite le raisonnement sur la correction des transformations successives du reset modulaire.

MOTS CLÉS

langages synchrones à flots de données, Lustre, Scade, compilation vérifiée, sémantique mécanisée, Vélus, assistants de preuve interactifs, Coq, reset modulaire

ABSTRACT

Specifications based on block diagrams and state machines are used to design control software, especially in the certified development of safety-critical applications. Tools like SCADE and Simulink/Stateflow are equipped with compilers that translate such specifications into executable code. They provide programming languages for composing functions over streams as typified by dataflow synchronous languages like Lustre.
In this thesis we present Vélus, a Lustre compiler verified in the interactive theorem prover Coq. We develop semantic models for the various languages in the compilation chain, and build on the verified CompCert C compiler to generate executable code and give an end-to-end correctness proof. The main challenge is to show semantic preservation between the dataflow paradigm and the imperative paradigm, and to reason about byte-level representations of program states.
We treat, in particular, the modular reset construct, a primitive for resetting subsystems. This necessitates the design of suitable semantic models, compilation algorithms and corresponding correctness proofs. We introduce a novel intermediate language into the usual clock-directed modular compilation scheme of Lustre. This permits the implementation of compilation passes that generate better sequential code, and facilitates reasoning about the correctness of the successive transformations of the modular reset construct.

KEYWORDS

synchronous dataflow languages, Lustre, Scade, verified compilation, mechanized semantics, Vélus, interactive theorem provers, Coq, modular reset