



THÈSE DE DOCTORAT

DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Compilation vérifiée d'un langage synchrone à flots de données avec machines à états

Verified Compilation of a Synchronous Dataflow Language with State Machines

Soutenue par

Basile Pesin

Le 13 octobre 2023

École doctorale n°386

**Sciences Mathématiques
de Paris Centre**

Spécialité

Informatique

Composition du jury :

Magnus Myreen Chalmers University of Technology	<i>Rapporteur</i>
Robert de Simone Inria	<i>Rapporteur</i>
Carlos Agon IRCAM	<i>Examineur</i>
Julien Forget Université de Lille	<i>Examineur</i>
Xavier Leroy Collège de France	<i>Examineur</i>
Florence Maraninchi Université Grenoble Alpes	<i>Présidente</i>
Timothy Bourke Inria / ENS	<i>Co-Directeur</i>
Marc Pouzet ENS / Inria	<i>Co-Directeur</i>

Remerciements

Avant de commencer, je souhaiterais remercier mes rapporteurs, Magnus Myreen et Robert de Simone, pour avoir pris le temps de lire ce long document, et pour leurs conseils avisés. Je remercie aussi Carlos Agon, Julien Forget, Xavier Leroy et Florence Maraninchi pour avoir accepté de participer à mon jury de thèse et pour avoir prêté attention à mon travail.

Ce travail n'aurait pas pu voir le jour sans le soutien de mes encadrants Timothy Bourke et Marc Pouzet qui m'ont accueilli comme stagiaire il y a presque quatre ans, puis m'ont renouvelé leur confiance pour trois années de thèse. Malgré deux confinements et leurs nombreuses responsabilités, ils ont toujours été disponibles pour me conseiller et m'orienter pendant ces années de travail. Il m'ont transmis une méthode de travail rigoureuse et affiné mon esprit critique. Je n'aurais pas pu espérer de meilleurs directeurs de thèse.

Ces années n'auraient pas été aussi agréables sans l'esprit de camaraderie partagé dans l'équipe PARKAS. Au delà de leurs conseils, les membres permanents, Tim, Marc, et Guillaume, ont toujours su partager leur travail et donner envie de s'y intéresser. Il a toujours été agréable de discuter de nos travaux (ou d'autres choses) avec les autres thésards de l'équipe, Léo, Paul J, Baptiste et Grégoire. En particulier, merci à Paul pour m'avoir motivé à m'entraîner au baby-foot: après tous ces efforts, j'ai enfin atteint un niveau passable ! Enfin, les stagiaires, Astyax, Antonin, Reyyan, Carolina, Vrushank, Paul R, Victor et Antoine, ont rythmé la vie de l'équipe et apporté un regard frais sur nos travaux.

J'ai aussi eu le plaisir d'enseigner à Jussieu, ma fac d'origine. Je remercie tous les responsables d'UE qui m'ont fait suffisamment confiance pour me confier des étudiants: Carlos Agon, Emmanuel Chailloux, Stéphane Doncieux, Mathieu Jaume, Pascal Manoury et Frédéric Peschanski. Enseigner a été extrêmement formateur, et toujours un plaisir.

Merci aussi à Léo Andrès, Colin Gonzalez et Loïc Sylvestre, avec qui j'ai pu organiser le meetup OCaml. Merci également à Gabriel Scherer, qui en plus de nous aider à trouver de nombreux orateurs passionnants, nous a permis de financer les pizzas sans lesquelles le meetup n'aurait pas été le même.

Enfin, je remercie ma famille pour leur soutien pendant mes (trop?) longues études. Mes parents, Laure-Anne et Philippe ont toujours supporté mes décisions. La complicité de mon frère, Jules, a ouvert mes horizons. J'ai eu la chance de rencontrer Mannielin un peu avant le début de ma thèse. Nous nous sommes installés ensemble il y a presque trois ans, et je la remercie pour sa patience et ses encouragements au quotidien. Elle a su me protéger de mes plus mauvais instincts, mon anxiété et mon workaholisme. Son soutien et son amour m'ont permis d'aller jusqu'au bout.

Résumé

Les systèmes embarqués critiques sont souvent spécifiés par des formalismes schéma-bloc. SCADE Suite est un environnement de développement pour ces systèmes utilisé depuis vingt ans dans l'industrie avionique, nucléaire, automobile, et autres domaines critiques. Son formalisme graphique se traduit en une représentation textuelle basée sur le langage synchrone à flots de données Lustre, et incorpore des fonctionnalités de langages plus récents comme Lucid Synchrone. En Lustre, un programme est défini comme un ensemble d'équations qui spécifie la relation entre entrées et sorties du programme à chaque instant. Le langage des expressions inclut des opérateurs arithmétiques et logiques, des opérateurs de délais qui permettent d'accéder à la précédente valeur d'une expression, et des opérateurs d'échantillonnage qui permettent à certaines valeurs d'être calculées moins souvent que d'autres.

Le projet Vélus est une formalisation d'un sous-ensemble du langage Scade 6 dans l'assistant de preuves Coq. Il propose une formalisation de la sémantique dynamique du langage sous forme de relations entre flots infinis d'entrées et de sorties. Il inclut aussi un compilateur qui utilise CompCert, un compilateur vérifié pour C, pour produire du code assembleur. Enfin, il fournit une preuve de bout-en-bout que ce compilateur préserve la sémantique à flots de données des programmes sources.

Cette thèse étends Vélus en y ajoutant les blocs de contrôles de Scade 6 et Lucid Synchrone, ce qui inclut une construction qui contrôle l'activation des équations selon une condition (switch), une construction permettant d'accéder à la valeur précédente d'une variable (last), une construction qui réinitialise les opérateurs de délai (reset), et, enfin, des machines à états hiérarchiques, qui permettent la spécification de comportements modaux complexes. Toutes ces constructions peuvent être arbitrairement imbriquées dans un programme. Nous étendons la sémantique de Vélus avec une nouvelle spécification pour ces constructions qui encode leur comportement par l'échantillonnage. Nous proposons un schéma d'induction générique pour les programmes bien formés qui permet de prouver certaines propriétés du modèle sémantique, comme son déterminisme ou l'adhérence des valeurs aux types déclarés. Enfin, nous décrivons la compilation de ces constructions telle qu'implémentée dans Vélus. Nous montrons que le modèle de compilation qui réécrit ces constructions dans le langage noyau peut être implémenté, spécifié et vérifié dans Coq. La compilation de last et reset nécessite des changements plus profonds dans les langages intermédiaires de Vélus.

Abstract

Safety-critical embedded systems are often specified using block-diagram formalisms. SCADE Suite is a development environment for such systems which has been used industrially in avionics, nuclear plants, automotive and other safety-critical contexts for twenty years. Its graphical formalism translates to a textual representation based on the Lustre synchronous dataflow language, with extensions from later languages like Lucid Synchrone. In Lustre, a program is defined as a set of equations that relate inputs and outputs of the program at each discrete time step. The language of expressions at right of equations includes arithmetic and logic operators, delay operators that access the previous value of an expression, and sampling operators that allow some values to be calculated less often than others.

The Vélus project aims at formalizing a subset of the Scade 6 language in the Coq Proof Assistant. It proposes a specification of the dynamic semantics of the language as a relation between infinite streams of inputs and outputs. It also includes a compiler that uses CompCert, a verified compiler for C, as its back end to produce assembly code, and an end-to-end proof that compilation preserves the semantics of dataflow programs.

In this thesis, we extend Vélus to support control blocks present in Scade 6 and Lucid Synchrone, which includes a construction that controls the activation of equations based on a condition (switch), a construction that accesses the previous value of a named variable (last), a construction that re-initializes delay operators (reset), and finally, hierarchical state machines, which allow for the definition of complex modal behaviors. All of these constructions may be arbitrarily nested in a program. We extend the existing semantics of Vélus with a novel specification for these constructs that encodes their behavior using sampling. We propose a generic induction principle for well-formed programs, which is used to prove properties of the semantic model such as determinism and type system correctness. Finally, we describe the extension of the Vélus compiler to handle these new constructs. We show that the existing compilation scheme that lowers these constructs into the core dataflow language can be implemented, specified and verified in Coq. Compiling the reset and last constructs requires deeper changes in the intermediate languages of Vélus.

Résumé étendu

La thèse étant rédigée en anglais, nous résumons ici les points scientifiques principaux décrits dans chaque chapitre, en français.

Introduction

Les systèmes embarqués critiques sont couramment programmés au moyen de langages synchrones [BB91], qui permettent d'abstraire le temps physique continu en temps logique discret. Un programme synchrone lit ses entrées, exécute des calculs internes et écrit ses sorties au sein d'un même instant logique. Cela signifie que deux programmes communiquent de manière synchrone à chaque instant logique, ce qui rend les communications atomiques et la concurrence déterministe. Dans cette thèse, on s'intéresse plus particulièrement aux langages synchrones à flots de données, dans lesquels chaque programme spécifie la relation entre ses entrées et sorties sous forme d'équations.

Vélus [PLDI17; POPL20; EMSOFT21] est un projet de mécanisation pour un langage synchrone à flots de données basé sur Lustre [Hal+91] et Scade [CPP17] dans l'assistant de preuve Coq [Coq]. Le langage est spécifié par une sémantique relationnelle à flots de données. Vélus inclut également un compilateur qui génère un programme C qui est ensuite passé au compilateur CompCert [Ler09b] qui produit du code assembleur.

Cette chaîne de compilation est prouvée correcte, au sens où, si l'on peut associer une sémantique à flots de données au programme source G , et si le compilateur produit un programme assembleur P à partir de G , alors P a une sémantique, et cette sémantique correspond à celle du source. Ce résultat est formalisé dans le théorème de la [page 11](#), qui est établi par une preuve mécanisée en Coq.

Cette thèse présente une extension de Vélus aux blocs de contrôles inspirés de Lucid Synchrone [CHP06] et Scade 6 [CPP17]. Ces constructions contrôlent l'activation des équations du programme. On les présente d'abord au moyen d'un exemple de système embarqué simple : le moteur pas à pas utilisé dans une petite imprimante thermique.

On donne d'abord, en [page 7](#), l'exemple d'un *nœud* utilisant seulement les constructions du langage noyau pour calculer la somme de ses entrées. Il utilise l'opérateur `fbv`, qui

permet d'accéder à la valeur précédente d'un flot. L'exemple présenté en [page 8](#) utilise un bloc `switch` pour implémenter la séquence de contrôle des phases du moteur pas à pas. A chaque fois que la condition `step` est vraie, les phases sont mises à jour, et le moteur tourne d'un quart de tour dans le sens anti-horaire. L'exemple utilise aussi l'opérateur `last` pour accéder à la dernière valeur d'une variable partagée. La valeur initiale de `last x` doit être systématiquement spécifiée, puisque Vélus n'utilise pas d'analyse d'initialisation. Par ailleurs, l'équation grisée dans la seconde branche du `switch` peut être omise. En effet, la sémantique et le compilateur complètent les définitions partielles des variables définies avec `last`. Cela permet d'écrire des programmes dans un style plus impératif, avec des variables d'état mises à jour seulement quand elles sont explicitement définies.

Enfin, l'exemple de la [page 9](#) présente le contrôle de haut niveau du moteur pas à pas. L'application d'un bloc `reset` au nœud `count_up` permet de calculer le temps écoulé depuis le dernier changement de phase. La longueur d'une phase est déterminée par une machine à états hiérarchique. Une notation graphique est superimposée sur le code source pour aider à la compréhension du programme. Dans l'état initial, `Starting`, cette phase est plus longue pour permettre au moteur de prendre de la vitesse. Après une phase, le moteur passe dans l'état `Moving`, où la durée des phases est constante. Le moteur peut être stoppé, auquel cas il passe dans l'état `Holding`. Si le moteur reste en pause pendant plus d'une phase, la puissance envoyée doit être modulée par le nœud `pwm`. Par ailleurs, si la pause est suffisamment courte, le retour à l'état `Feeding` est fait "avec histoire" (transition `continue`), ce qui permet d'entrer directement dans le sous-état `Moving`. Dans le cas contraire, la transition `then` réinitialise l'état de la machine imbriquée dans `Feeding`, ce qui réactive l'état initial `Starting`.

Sémantique du langage Vélus

La sémantique du langage source est donnée par un ensemble de relations entre flots de données infinis. Le jugement $G \vdash f(xss) \Downarrow yss$ indique que le nœud f du programme G associe les flots d'entrées xss aux flots de sorties yss . La règle de sémantique correspondante est donnée en [page 29](#). Elle impose l'existence d'un *historique* H , qui est un environnement associant un flot à chaque variable du nœud, et en particulier à ses entrées et sorties. La valeur d'une variable apparaissant dans une expression est également lue dans l'historique. L'historique est contraint par chaque équation du nœud. L'équation $xs = es$ force la valeur de x_i à être la i -ème valeur produite par les expressions es .

Les flots manipulés peuvent être échantillonnés, c'est-à-dire que certains sont calculés moins souvent que d'autres. Dans Vélus, on caractérise l'échantillonnage explicitement par des valeurs présentes (notées $\langle v \rangle$) et absentes (notées $\langle \rangle$). L'expression e `when` $C(x)$ échantillonne les valeurs de l'expression e , en ne les conservant qu'aux instants où la condition x vaut C . Cette opération est mécanisée par l'opérateur `when`, défini comme une fonction co-inductive et partielle. L'opérateur `merge` permet de combiner des flots échantillonnés complémentaires.

La sémantique de chaque autre opérateur du langage d'expressions est aussi exprimée par une fonction de flots. Par exemple, l'opérateur de délai initialisé `fbv` est défini par

une paire de fonctions, présentées en [page 34](#). La première, `fbv`, sélectionne la première valeur présente du flot de gauche, tandis que les valeurs du flot de droite sont passées au premier argument de `fbv1`, et sont produites à la prochaine présence.

On note que cet opérateur est insensible aux absences, c'est-à-dire qu'en insérer et supprimer dans les flots en entrée ne change pas les valeurs dans les flots en sortie. Nos définitions de la sémantique des blocs de contrôle s'appuient sur cette observation. Le cas du `switch` est le plus élémentaire. Une branche doit être insensible, dans le même sens que pour les `fbv` et les nœuds, quand l'expression de garde ne correspond pas à l'étiquette de la branche. De plus, le `switch` doit combiner les flots produits par les branches, en choisissant à chaque instant les valeurs de la branche désignée par l'expression de garde. Ces intuitions sont concrétisées par la règle présentée en [page 40](#). Elle réutilise l'opérateur d'échantillonnage `when` discuté plus tôt, mais l'applique à l'historique H pour échantillonner l'ensemble des flots lus et écrits par les sous-blocs du `switch`.

De la même manière, la sémantique d'un bloc `reset` est spécifiée par un opérateur `mask` appliqué à l'historique contraint par les sous-blocs. Finalement, les règles sémantiques pour les machines à état hiérarchiques, présentées en [page 49](#), suivent le même schéma, en utilisant l'opérateur sémantique `select`, qui est essentiellement une combinaison de `when` et `mask`. L'opérateur `select` est contrôlé par le *flots d'états*, qui indique à chaque cycle quel état doit être actif, et s'il doit être réinitialisé. Le flot est défini par les transitions de l'état actif à chaque cycle. Une particularité de notre présentation est que Vélu n'autorise pas la définition de machines à états qui mélangent transitions faibles (`until`) et fortes (`unless`). Cela simplifie la définition des règles sémantiques, du schéma de compilation, et surtout évite certaines interactions dont la sémantique n'est pas claire.

Enfin, la sémantique de l'opérateur `last` est traitée en ajoutant les flots correspondants à chaque `last x` directement dans l'historique. Ces flots sont contraints au niveau de l'équation d'initialisation du `last`.

Analyse de dépendance vérifiée

Ce modèle sémantique relationnel n'est pas suffisant pour garantir certaines propriétés du langage. Par exemple, l'équation $x = x$ n'est pas déterministe : notre modèle sémantique peut associer n'importe quel flot à x . A l'inverse, l'équation $x = x + 1$ n'admet pas de sémantique, alors qu'elle est bien typée. De plus, ces équations ne peuvent pas être compilées en un programme impératif où chaque valeur doit être calculée avant sa lecture. C'est aussi le cas pour un système d'équations contenant un cycle, comme par exemple $x = y + 1; y = x * 2$.

Un nœud contenant de telles équations est donc rejeté par Vélu au moyen d'une analyse statique de dépendance. Cette analyse construit d'abord un graphe de dépendances des variables du programme. Les sommets de ce graphe sont des étiquettes uniques associées à chaque variable défini dans le programme. La fonction $\text{UsedInst}_\Gamma(e)[k]$ collecte l'ensemble des étiquettes des variables utilisées pour définir le k -ième flot de l'expression e . Quelques cas de la définition de UsedInst sont donnés en [page 58](#). La fonction ne considère que les variables utilisées instantanément, c'est-à-dire celles qui n'apparaissent pas à droite

d'un `fbv`. En revanche, la fonction considère que les flots produits par un appel de nœud dépendent de toutes les entrées du nœud. C'est une contrainte forte, mais cohérente avec le schéma de compilation utilisé dans Vélus, qui compile tout appel de nœud atomiquement. La règle de dépendance pour les équations, présentée en [page 59](#), spécifie que la k -ième variable à gauche d'une équation dépend de toutes les variables utilisées instantanément pour calculer le k -ième flot des expressions à droite. Les cas des blocs `switch`, `reset`, et des machines à états introduisent des dépendances supplémentaires entre les conditions de gardes et transitions et les variables définies par les sous blocs contrôlés par ces conditions.

Le graphe de dépendances construit selon ces règles est ensuite analysé par un algorithme de détection de cycles basé sur la recherche en profondeur. Cet algorithme, implémenté en Coq, est présenté en [page 65](#). Le critère de récursion gardé de Coq, qui impose que tout appel récursif soit fait sur un sous terme strict du paramètre de la fonction, ne permet pas d'implémenter cet algorithme comme une fonction récursive. Pour contourner ce problème, on utilise la commande `Program Fixpoint`, qui permet de justifier de la terminaison de la fonction au moyen d'une mesure strictement décroissante sur son paramètre. On veut ensuite raisonner sur la correction de cet algorithme, c'est à dire prouver qu'un graphe accepté par l'algorithme est effectivement acyclique. Pour cela, on propose une caractérisation inductive des graphes acyclique, `AcyGraph`, présentée en [page 64](#). La preuve est réalisée a priori, en encodant l'invariant d'induction dans les types dépendants utilisés en entrée et sortie de l'algorithme.

Cette caractérisation des nœuds *causaux*, c'est à dire des nœuds pour lesquels le graphe de dépendance ne contient pas de cycle, est ensuite utilisée pour prouver deux propriétés fondamentales du modèle sémantique. La première est son déterminisme, spécifié par le théorème de la [page 73](#): l'exécution d'un nœud causal pour une entrée donnée produit toujours la même sortie. La deuxième est la correction du système d'horloge, spécifiée [page 75](#). Cette propriété indique que les absences et les présences des flots du modèle sémantique correspondent effectivement aux types d'horloges déclarés dans le programme. Cette propriété est indispensable pour prouver la correction du schéma de compilation.

Compilation source à source des blocs de contrôle

Le compilateur Vélus est structuré comme une série de passes. La partie avant du compilateur est présentée en [page 79](#).

L'analyse syntaxique des programmes sources est implémentée par un programme Coq généré par l'outil Menhir. L'arbre de syntaxe produit ne contient pas d'informations de type. Celles-ci sont ajoutés par la passe d'élaboration, qui est implémentée dans Coq. Cette passe ne peut pas être vérifiée, puisqu'il n'y a pas de modèle sémantique associé à la syntaxe non typée.

Les passes suivantes réécrivent le programme dans le même arbre de syntaxe. Chaque passe remplace l'une des constructions complexe du langage par une combinaison de constructions plus simples, ou normalise la structure du programme. La première passe complète les définitions partielles, comme celles vues dans le nœud `drive_sequence`, en insérant des équations `x = last x` quand nécessaire. La transformation est implémentée

comme une fonction Coq récursive sur les blocs d'un nœud. La preuve de correction sémantique pour cette transformation, comme pour les suivantes, est établie par induction sur la syntaxe du programme source, et analyse des cas induits par la fonction de compilation. L'invariant inductif de correction présenté en [page 94](#) est très simplifié par rapport à l'invariant utilisé en Coq, qui doit garder la trace de différents critères statiques sur le programme permettant de justifier de la correction de la compilation.

Les dépendances du programmes sont analysées après cette passe, ce qui permet de ne pas à avoir à traiter les dépendances “implicites” qui seraient introduites par les définitions partielles. Si l'analyse réussit, le résultat de correction du système d'horloge est utilisé pour construire un modèle sémantique *instrumenté*. Ce modèle contient plus d'informations sur les horloges des flots manipulés que le modèle de référence, ce qui est utile pour établir les preuves de correction sémantique pour les passes suivantes.

La passe suivante compile les machines à états en combinaison de blocs `switch` et `reset`, en suivant le schéma décrit dans [CPP05]. C'est aussi la première passe qui doit ajouter de nouvelles variables locales dans le nœud. Nous traitons le problème de la génération de noms frais dans un langage fonctionnel pure au moyen d'une approche semi-axiomatisée, semi-monadique. La fonction de génération de base, `gensym`, est axiomatisée, c'est à dire définie en OCaml plutôt qu'en Coq, ce qui lui permet de manipuler certaines structures de données, comme les tables de noms, de manière impérative. L'injectivité de la fonction est donnée comme un axiome, facilement vérifiable par inspection du code OCaml. Ensuite, on développe une monade `Fresh` dont l'état gère les noms frais générés. Dans les preuves de correction sémantiques, on raisonne à partir des propriétés fondamentales de cette monade, telle que la non-duplication des noms dans l'état.

La passe suivante élimine les blocs `switch` en les transformant en combinaison d'applications des opérateurs `when` sur les variables lues dans le `switch`, et `merge` sur celles définies par le `switch`. La preuve de correction sémantique pour cette passe est plus complexe que les deux précédentes, puisqu'elle doit connecter l'échantillonnage implicite du `switch` avec l'échantillonnage explicite des opérateurs `merge` et `when`.

Les deux passes précédentes introduisent des blocs locaux imbriqués dans le nœud. Cette passe les élimine pour se ramener à un programme ne contenant qu'un seul bloc de déclaration locales, à la racine du nœud. Certaines variables doivent être renommées pour éviter les duplications de noms. Renommer toutes les variables rendrait la traçabilité du programme généré plus difficile. On introduit donc une monade `Reuse` qui renomme les variables seulement quand nécessaire; elle réutilise certains composants de la monade `Fresh`.

La passe suivante simplifie les expressions en séparant les opérateurs induisant un état (`fby`, application de nœud) dans leur propres équations, et en distribuant les opérateurs sur leurs sous-expressions.

Il y a deux manières de traiter l'opérateur `last`: la plus simple est de l'éliminer tôt dans la compilation, en le remplaçant par une équation `fby`. Cependant, ce schéma mène à une génération de code sous-optimal, comme illustré dans la [page 93](#): les équations implicites complétées “par défaut” génèrent des instructions de mise à jour inutiles. A la place, on décide de conserver la construction `last` jusqu'à la génération de code impératif,

où elle peut être compilée plus efficacement. On introduit tout de même une passe qui normalise les équations manipulant les variables `last`, en particulier pour éliminer les initialisations non constantes.

Enfin, les équations `fbv` sont simplifiées de la même manière que les équations concernant les variables `last`.

Adaptation des passes arrières du compilateur

L'architecture des passes arrières du compilateur est présentée en [page 117](#). Elle est adaptée de travaux précédents, en particulier décrits dans la thèse de Léo Brun [\[Bru20\]](#). Le compilateur produit un programme impératif dans le langage `Obc`, un petit langage objet. Chaque nœud Lustre est traduit en une classe dont les variables d'états reflètent celles du programme source, et avec deux méthodes: `reset` qui initialise ou réinitialise l'état interne, et `step` qui réalise un pas d'exécution du nœud et met à jour l'état. Un exemple de programme `Obc` compilé depuis le nœud `drive-sequence` est présenté en [page 119](#). Chaque `switch` du programme source, qui a été compilé en une combinaison de `when` et `merge`, produit finalement un `switch` impératif. Plusieurs optimisations sont appliquées à ce programme: les deux `switchs` sur la même condition sont fusionnés pour limiter le nombre de branchements conditionnels dans le code généré. Les instructions de mise à jour inutiles, de la forme `state(x) := state(x)`, sont supprimées. Les langages intermédiaires et schémas de compilation sont conçus pour rendre ces deux optimisations les plus efficaces possible.

Le premier langage intermédiaire utilisé est `NLustre`, qui encode dans sa syntaxe la forme des programmes Lustre normalisés, avec une différence importante: le langage ne contient que des équations, alors qu'un programme Lustre normalisé peut toujours contenir des blocs `reset` imbriqués. La passe de transcription doit donc "aplatir" ces blocs en conditions de réinitialisation pour les équations `NLustre` générées. Dans les preuves, cette transformation pose quelques difficultés à cause de la quantification universelle utilisée dans la règle sémantique du `reset`. Pour compléter la preuve, nous devons utiliser l'axiome du tiers-exclus, qui n'est habituellement pas admis dans la logique constructive de `Coq`.

La simplicité du langage `NLustre` permet d'implémenter plusieurs passes d'optimisation des programmes à flots de données. En particulier, on y implémente des optimisations éliminant les inefficacités introduites pendant deux des passes avant du compilateur: la compilation des `switch` et la normalisation des `fbv`. Ces deux passes sont implémentées naïvement, ce qui simplifie leurs preuves de correction. En revanche, la première peut produire des variables inutilisées dans le nœud, et la seconde peut produire plusieurs équations `fbv` identiques. Au lieu de compliquer ces passes, on implémente, en `NLustre`, deux optimisations qui éliminent ces deux inefficacités, et dont la correction est facilement vérifiable.

Après ces optimisations, le programme `NLustre` est compilé dans le langage intermédiaire `Stc`, qui permet l'ordonnancement des équations du programme avant leur traduction vers un programme impératif. En particulier, la syntaxe de `Stc` permet l'ordonnancement

indépendant des contraintes de réinitialisation et de mise à jour des variables d'état. Pour traiter les variables `last` et les réinitialisations de variables d'état, on généralise les règles de sémantique et d'ordonnancement du langage. On ajoute également une passe permettant de couper les cycles entre mises à jour de variables d'état.

Enfin, le programme est traduit dans le langage `Obc`. La preuve de correction de cette passe de traduction dépend du bon ordonnancement des contraintes du système `Stc`. On doit également adapter l'optimisation de fusion de programmes `Obc` pour traiter les variables d'états compilées depuis des variables `last`. En revanche, la passe de génération de code `C`, et sa preuve de correction, ne demandent aucune modification.

Conclusion

Composer ces passes de compilation avec la fonction de compilation de `CompCert` produit une fonction `Coq` qui transforme un programme `Lustre` en code assembleur. Cette fonction est vérifiée de bout en bout. Elle est extraite vers un programme `OCaml`, et combinée avec la fonction d'analyse syntaxique et la fonction d'élaboration pour produire un compilateur exécutable.

Nous avons testé ce compilateur sur quelques programmes d'exemples pour calculer le Temps d'Exécution en Pire Cas (ou `Worst Case Execution Time`, `WCET`) de la fonction `step` principale pour chaque programme compilé. Nous avons comparé ces résultats avec ceux du compilateur académique `Heptagon`, couplé à `CompCert` puis `GCC`, en [page 171](#). Dans tous les cas, `Vélus` est plus efficace qu'`Heptagon` et `CompCert`. En revanche, les optimisations de `GCC`, même avec l'option `-O1`, produisent un code bien plus rapide que celui produit par `Vélus` et `CompCert`.

Nous nous sommes aussi intéressé aux aspects pratiques de l'ingénierie de preuves dans le cadre d'un compilateur réaliste comme `Vélus`. Pour cela, nous présentons, en [page 172](#), le nombre de lignes de code, le temps nécessaire à la compilation, et le temps de travail nécessaire à l'implémentation de chacune des fonctionnalités du langage. Les lignes de code sont classifiées en code exécutable, spécification, et preuve. Environ 6% de la base de code de `Vélus` est exécutable, soit trois fois moins que dans `CompCert`. Cette différence est expliquée, en partie, par les deux systèmes de types utilisés dans `Vélus`, et la nécessité de prouver que chaque passe les préserve. Cela étant dit, la taille et complexité du compilateur sont des facteurs qui pourraient ralentir et limiter de futures extensions de `Vélus`. Nous n'avons pas encore d'idée précise pour simplifier le développement, tout en gardant un langage expressif et un schéma de compilation efficace.

Notre travail de mécanisation d'un langage existant dans un assistant de preuves a pu rendre certains choix de conception et certaines difficultés d'implémentation plus explicites. La quantité de travail nécessaire pour chaque modification nous a convaincu que travailler dans un assistant de preuve n'est pas raisonnable pour prototyper un langage nouveau, et devrait être réservé aux langages déjà bien compris et spécifiés.

Nous pensons que notre travail peut être utile dans un contexte industriel. Même si il n'est pas réaliste de vérifier de bout en bout un compilateur industriel existant (nous n'avons traité qu'un sous-ensemble de `Scade`), nous pouvons imaginer plusieurs

manières d'intégrer un assistant de preuve dans le processus de développement et de qualification. Définir la sémantique formelle du langage dans l'assistant de preuve permet d'augmenter la confiance dans la conception du langage, et est utile à des fins de documentation. Un interpréteur vérifié par rapport à cette sémantique pourrait être utilisé pour tester le compilateur. Une passe de compilation particulièrement complexe pourrait être implémenté et vérifiée dans l'assistant de preuves. Chacune de ces idées exploite la formalisation mécanisée pour augmenter la confiance dans le compilateur, faciliter la qualification du logiciel, et clarifier la conception du langage et du compilateur.

Contents

1	Introduction	1
1.1	Context	1
1.1.1	Synchronous Languages and Embedded Systems	1
1.1.2	Compiler Verification	4
1.2	Programming with Vélus and State Machines	6
1.3	Overview of the Vélus Compiler	10
1.4	Prototype Implementation	13
1.5	Organization	13
2	Extending Vélus with Control Blocks	17
2.1	Syntax of the Vélus source language	17
2.1.1	Representation of the AST in the Coq Proof Assistant	19
2.2	Abstracting the CompCert Back End	22
2.3	Representing Infinite Sequences	23
2.3.1	Indexed Streams	24
2.3.2	Coinductive Streams	25
2.4	The Core Dataflow Semantics of Vélus	27
2.4.1	Histories and Equations	27
2.4.2	Sampling and Clock Typing	30
2.4.3	Stateful Operators	33
2.4.4	Node Instantiation	35
2.5	Semantics of Switch	39
2.5.1	Activation and Sampling	39
2.5.2	Clock Typing of Switch Blocks	41
2.6	Semantics of Reset	41
2.6.1	Reset as Sampling	42
2.6.2	Clock Typing of Reset	44
2.7	Semantics of Local Declarations	45
2.8	Semantics of Shared Variables	46

2.9	Semantics of State Machines	46
2.10	Partial Definitions	50
2.11	Discussion and Related Work	51
2.11.1	Mechanized Semantics for Verified Compilers	51
2.11.2	Possibly Finite Coinductive Streams	52
2.11.3	Synchronous Semantics for Dataflow Languages	53
2.11.4	Modeling the Semantics of State Machines	54
3	Verified Dependency Analysis	57
3.1	Dependency Graph of a Vélus Program	57
3.1.1	Analysis of Expressions	58
3.1.2	Dependencies induced by blocks	59
3.2	Verified Graph Analysis	63
3.3	Induction Schemes for Causal Programs	66
3.3.1	Induction on the labels of a node	67
3.3.2	Induction on the syntax of blocks and local declarations	69
3.3.3	Induction on the k th stream of an expression	71
3.4	Determinism of the Semantic Model	73
3.5	Clock Correctness	74
3.6	Discussion and Related Work	76
3.6.1	Causality Type Systems for Dataflow Languages	76
3.6.2	Verified Graph Analysis	78
4	Front-End Compilation	79
4.1	Parsing of Source Programs	79
4.2	Generating Fresh Identifiers	80
4.2.1	Gensym Axiomatization	81
4.2.2	The Fresh Monad	84
4.3	Elaboration of Lustre Programs	86
4.3.1	Clock-Type Elaboration by Monadic Unification	87
4.3.2	Translation Validation of the Elaboration	90
4.4	Structure of the source-to-source rewriting passes	91
4.4.1	Normalized subset of the language	91
4.4.2	Implementation and Notations	92
4.5	Completing Partial Definitions	94
4.5.1	Compilation Function	94
4.5.2	Correctness	94
4.6	Dependency Analysis and Clocked Semantic Model	95
4.7	Compiling State Machines	98
4.7.1	Compilation Function	98
4.7.2	Correctness	99
4.8	Compiling Switch Blocks	100
4.8.1	Compilation Function	100
4.8.2	Correctness	100

4.9	Flattening Local Scopes	102
4.9.1	Compilation Function	102
4.9.2	Fresh identifiers and the <code>Reuse</code> monad	103
4.9.3	Correctness	104
4.10	Unnesting and Distribution	105
4.10.1	Compilation Function	105
4.10.2	Correctness	106
4.11	Normalization of shared variables	106
4.11.1	Initializing lasts with constants	107
4.11.2	Removing lasts on outputs	108
4.11.3	Stateless definitions for lasts	110
4.12	Normalization of fby equations	112
4.12.1	Compilation Function	112
4.12.2	Correctness	113
4.13	Discussion and Related Works	114
4.13.1	Translation validation of synchronous dataflow programs	114
4.13.2	Generating Fresh Identifiers	115
5	Middle-End Compilation	117
5.1	The <code>Obc</code> target language	118
5.1.1	Syntax of <code>Obc</code>	118
5.1.2	A compiled example	118
5.1.3	Semantics of <code>Obc</code>	120
5.1.4	Optimizations	121
5.2	<code>NLustre</code> : a normalized dataflow language	123
5.2.1	Semantic Models	123
5.2.2	Transcription: From <code>Lustre</code> to <code>NLustre</code>	130
5.2.3	<code>NLustre</code> Optimizations	134
5.3	Generalizing the <code>Stc</code> language	143
5.3.1	Example and informal semantics	143
5.3.2	Syntax of <code>Stc</code>	145
5.3.3	Formal semantics of <code>Stc</code>	145
5.3.4	From <code>NLustre</code> to <code>Stc</code>	148
5.4	Translation to imperative <code>Obc</code> code	151
5.4.1	From <code>Stc</code> to <code>Obc</code>	151
5.4.2	Scheduling of <code>Stc</code> Constraints	153
5.4.3	<code>Stc</code> to <code>Obc</code> Correctness proof	159
5.4.4	Changes to the Fusion Optimization in <code>Obc</code>	162
5.5	Discussion and Related Work	166
5.5.1	Compilation to <code>CompCert Clight</code> and Beyond	166
5.5.2	Verified Compilation in <code>CakeML</code>	167
5.5.3	Translation Validation of Dataflow Programs	167

6 Conclusion	169
6.1 Experimental Evaluation	169
6.2 Proof Engineering and Practical Concerns	171
6.3 Open Questions	174
6.4 Concluding Remarks	175
A Type Systems and Static Predicates of Vélus	177
A.1 Node Invariants	177
A.1.1 Variables Defined	177
A.1.2 No Duplication in Declarations, No Shadowing	178
A.1.3 Shape of identifiers	179
A.2 Type System	179
A.3 Clock-Type System	182
B Full Compilation of the Introductory Example	187
C A Semantic Preservation Proof	199
Bibliography	205
Index	215

Introduction

Vélus [PLDI17; POPL20; EMSOFT21] is a mechanized formalization of a Synchronous Dataflow Programming Language based on Lustre [Hal+91] and Scade [CPP17]. It specifies the dynamic semantics and type systems of the language. It also includes a compiler that produces imperative C code. Vélus is implemented in the Coq Proof Assistant [Coq], which provides a specification language, a pure functional programming language that we use to implement compilation algorithms, and a tactic language used to build proofs. The compiler is implemented as a composition of functions that rewrite the program into successive intermediate languages, each with their own fully-specified semantic model. Each function is proven correct with regards to these models. The C program produced by Vélus is then compiled to assembly code by CompCert [Ler09b], a verified compiler also implemented in Coq. The resulting compilation chain comes with an end-to-end proof that the generated code respects the semantics of the source program.

This thesis presents the extension of Vélus with control constructs inspired from Lucid Synchronic [CHP06] and Scade 6 [CPP17]. We extend the synchronous semantics of Vélus with a novel formalization for these constructs. We show that the traditional source-to-source compilation scheme [CPP05] can be implemented and verified in an Interactive Theorem Prover (ITP), and highlight where it needs to be adapted. In particular, we had to modify the middle-end of Vélus extensively to generate efficient imperative code. We also prove some fundamental properties of this semantic model, using a novel induction principle for well-formed programs.

1.1 Context

1.1.1 Synchronous Languages and Embedded Systems

Synchronous Programming Languages Synchronous programming [BB91] was first proposed in the late 80s as a way of specifying and implementing real-time reactive systems. The core of this approach is to abstract physical time into a discrete, logical time. Programs execute cyclically by sampling their inputs, performing internal computations, and writing their outputs. Communications between two programs happen synchronously

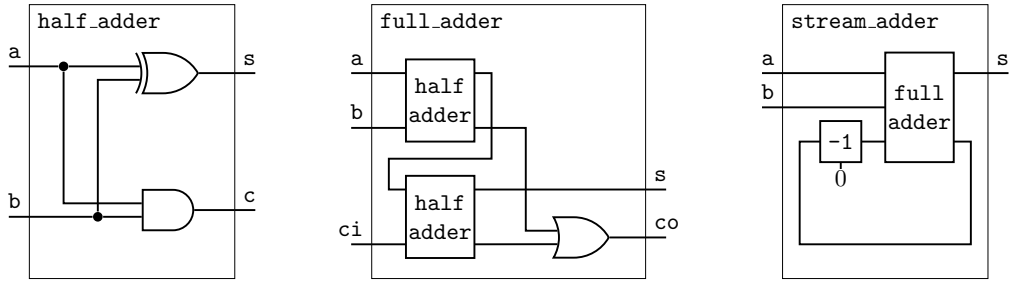


Figure 1.1: Block-Diagram representation of a bit-stream adder

at each logical time step. This makes these communications atomic, which allows for deterministic concurrency. For this abstraction to be valid, synchronous programs must execute within a logical time step : they “produce their outputs synchronously with their inputs, their reaction taking no observable time” [BB91]. This is the case, as long as the physical time necessary to execute a cycle of a program is less than the physical length of a logical time step. In particular, they must execute in bounded time.

In the late 80s, domain-specific languages that satisfy this constraint by construction were introduced. Esterel [BC84; Tec05] is a procedural synchronous programming language. Programs are specified as imperative commands, which can be composed in parallel, or in sequence, and communicate using synchronous signals. Both SIGNAL [LeG+91; GTL03], and Lustre [Cas+87; Hal+91] are synchronous dataflow languages, which were originally designed to specify control and signal-processing applications. Dataflow programs specify relations between input and output streams of values. These programs can be represented visually as block-diagrams, with blocks representing functions connected by wires representing streams. Blocks may be abstracted and composed to specify complex behaviors. These languages provide delay primitives to refer to the past values of streams. For example, the first block in figure 1.1 represents half of a one-bit binary adder, built from primitive `xor` and `and` operators. The second block composes two half-adders to form a full one-bit adder with carry. Finally, the last block defines an adder on two streams of bits starting with the least-significant bit. The carry bit is treated using an initialized delay represented by the `-1` block. These simple principles can be applied to build more complex signal-processing functions. Finally, both of these languages support sampling: some signals may be produced slower than the base rate of the system. A boolean *clock* is associated to each signal, indicating whether or not the signal is produced at each cycle. In the case of Lustre, a static clock-type system [CP03] is used to ensure sampling operators are used correctly. This clock-typing discipline allows for the generation of efficient imperative code [Bie+08].

Specifying Reactive Systems with State Machines Over the last 30 years, many research projects have aimed at extending and combining ideas from these foundations, as well as ideas from other programming languages. In this dissertation, we are particularly interested in projects that aim to extend dataflow-synchronous languages with block-based

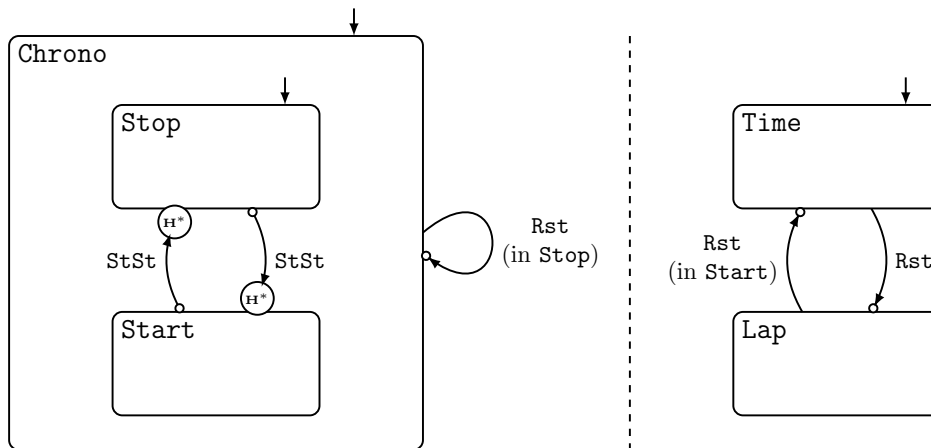


Figure 1.2: Statechart for a simple stopwatch

control constructs. We now provide a brief summary of the most relevant related works.

In [Har87], Harel proposes Statecharts, “A visual formalism for complex systems”. Figure 1.2 presents an example of a statechart, for a simple stopwatch. External events, like pressing a button, trigger transitions that change the active state. In the example, pressing button `StSt` while in state `Stop` changes the active state to `Start`. A transition may also depend on the internal state of the system: for instance, the `Rst` transition that reenters state `Chrono` checks if the state is `Stop`. The example shows a parallel composition of two state machines: the behavior of the internal chronometer (at left), is separated from the behavior of the screen (at right). It also shows the refinement of a state: the `Chrono` machine contains two sub-states, `Stop` and `Start`.

Several projects have focused on combining the features from Statecharts with the synchronous dataflow model of Lustre. SyncCharts [And95] mixes the ideas from Statecharts with the operators of Esterel. The Mode-Automata language [MR98] embeds equations written in a subset of the Lustre language (without explicit sampling) inside each state of an automaton.

Lucid Synchron is a synchronous dataflow language based on Lustre, extended with ML-like functional programming [Pou06]. In [CPP05], the authors extend Lucid Synchron to support state machines and other control blocks. These features are then compiled away into more primitive dataflow operations already present in Lustre and Lucid Synchron.

Industrial Uses Based on these academic efforts, synchronous programming has attracted industrial interest in the domain of safety-critical embedded software. Scade Suite is a development environment for embedded software based on a synchronous dataflow language, Scade 6 [CPP17]. Among other features, this language includes hierarchical state machines, similar to the ones present in Lucid Synchron. Scade Suite comes with a code generator, KCG, that produces imperative code from Scade 6 programs.

This code generator is *qualified*: its development is guided by a strict specification of the input language, and of the code transformations. The correctness of this specification, and the correspondence of the actual implementation to its specification are supported by extensive reviews and testing. This qualification means the compiler can be used to generate code for safety-critical applications.

The Vélus project aims at formalizing features of Scade 6 in the Coq Proof Assistant. Earlier works [EMSOFT21] proposed a specification for a pure dataflow Lustre-like subset of Scade 6. The main contribution of this thesis is to extend Vélus to support state machines and other control blocks inspired from Lucid Sychrone and Scade 6.

1.1.2 Compiler Verification

Safety-critical controllers for embedded systems are designed following strict methodologies. Usually, engineers first write a formal specification of the system. The actual implementation then follows this specification closely. The correspondence of the implementation with the design is checked by reviewing and testing the code, and by formal methods. Dataflow synchronous languages are designed to be syntactically close to the specification. This makes the features of the specification more easily traceable in the implementation, and reduces the risk of a disagreement between the two. However, this also means that the actual running code may differ significantly from the specification, since it is automatically generated by a compiler. It is crucial to check that the compiler does not introduce any discrepancy in the behavior of the generated code.

More generally, compilers are complex and error-prone programs. In [Yan+11], the authors use randomly-generated programs to test 11 different C compilers and check for differences in the behavior of the generated code. It was found that all but one compilers exhibited errors during their middle-end, optimizing passes. The one compiler that didn't introduce any error during optimization was CompCert [Ler09b], where these passes are verified in the Coq Proof Assistant.

But what does it mean for a compiler to be verified? In [Ler09b], the author of CompCert states it as follows. Suppose a compiler that, given a source program in language L_1 produces a program in language L_2 . The behavior, or semantics of both of these language must be formally specified. We note $P \Downarrow B$ for “program P has behavior B ”. For a low level language, like assembly, the formalized semantics should correspond to its execution. For a higher level language, it should correspond to the specification of the language. Informally, we expect that the compiler being correct means that “for any source program P , the program produced by the compiler has the same behavior as P ”. If we model a compiler by a (possibly partial) `compile` function, this could be formally expressed by: $\forall P, P \Downarrow B \iff \text{compile}(P) \Downarrow B$. Unfortunately, this equivalence is too strong in practice. For instance, the source program's behavior may not be defined because of an operation that is then compiled away. In this case, it would be impossible to prove the right-to-left implication. We can express a weaker correctness lemma: $\forall P, P \Downarrow B \implies \text{compile}(P) \Downarrow B$, which expresses that, for any program P with defined behavior B , the compiled program also has behavior B .

There are several approaches to showing that this property holds for a given `compile` function. We now discuss the most common.

Verified compilation consists in verifying a `compile` function directly, by analysis of its definition. The proof usually proceeds by induction on the structure of the input program P , or on the derivation of the semantic judgment $P \Downarrow B$. This approach can be tedious, because of the complexity of compilation functions, but proof assistants often provide automation, and at the very least check that no case was forgotten. The main advantage of this approach is that, like a pen-and-paper proof, it requires understanding and explaining in every detail the reasoning that imply that the `compile` function is actually correct. Moreover, any existing error in the `compile` function will eventually be uncovered by unsuccessful effort at proving a false statement.

Translation validation consists in treating the `compile` function as a black box, but providing a `validate` function that checks that the input and output of `compile` are equivalent [PSS98b]. This function is formally verified by a theorem of the form: $\forall P, P \Downarrow B \wedge \text{validate}(P, \text{compile}(P)) = \text{true} \implies \text{compile}(P) \Downarrow B$. In other words, if the input and output programs are equivalent in the sense checked by `validate`, then they do have the same behavior. This approach is particularly useful when `validate` is simpler to verify than `compile` would be; this is particularly the case when `compile` involves complex data structures or heuristics. If `validate` cannot verify that the input and output programs are indeed equivalent, then the whole compiler stops, never allowing incorrect code to be produced. If this happens, it either means that there is a bug in `compile`, or that the `validate` function is too strict.

Proof-carrying Code is a more general approach than translation validation, where the `compile` function is instrumented to provide a certificate C of an expected property of the generated program : $\text{compile}(P) = (P', C)$. This certificate may then be checked efficiently and independently [Nec97; App03]. In the context of verified compilation, the property of interest would be the semantic correspondence of P' and P . As with the translation validation approach, the certificate checker is verified: if $\text{check}(P, P', C) = \text{true}$, then the semantics of P' corresponds to that of P . If, for any program, `check` returns `false`, it means that either there is a bug in the compiler algorithm or certificate generator, or that the `check` function is too strict.

In a given verified compiler, several of these techniques may be combined to verify different compilation passes. For instance, most of CompCert's passes are verified directly, but the register allocation pass, which involves a complex graph coloring algorithm, uses translation validation [RL10].

Proof Assistants have been used to verify other ambitious compilers. CakeML [Kum+14] is a compiler for a subset of Standard ML implemented in Higher Order Logic (HOL). Its verified back-end [Tan+16] produces machine code and uses a mix of direct verification and translation validation. One of its front-ends [Abr+20] uses the proof-carrying code

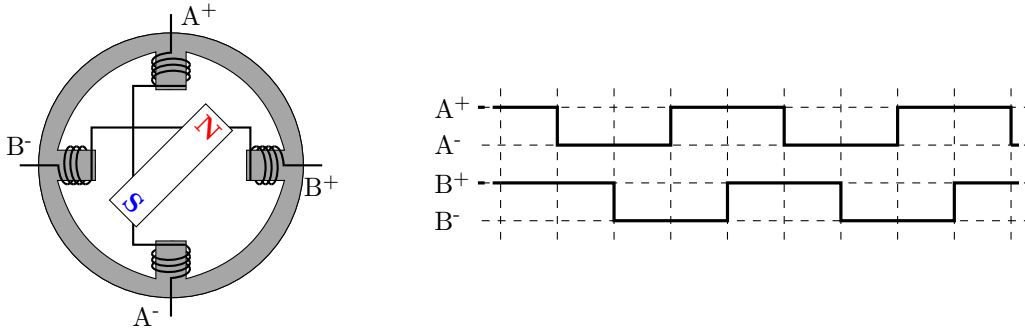


Figure 1.3: Stepper motor and its clock-wise drive sequence

approach to translate monadic HOL terms into the CakeML Abstract Syntax Tree (AST), along with a proof of semantic correspondence. CertiCoq [Ana+17] is a verified compiler for Coq’s programming language, Gallina, targeting CompCert C. Other efforts have focused on formalizing the Java programming language, including a verified compiler [Str02] implemented in Isabelle [Pau08].

The languages treated by these compilers, even if they follow different paradigms (imperative, functional, object-oriented), are all procedural. Verifying the compilation of a declarative, synchronous language poses different issues. First, the semantic model for dataflow language may be very different from that of a procedural language. Second, the compilation algorithms from declarative to imperative programs involve different passes and optimizations than that of a compiler for a procedural source language. For instance, the Esterel language admits several semantic model, which have been proven equivalent in Coq [RB22]. This work also describes the proof of correctness for the compilation of Esterel into circuits with regard to these models. Some verified compilers for such languages have turned to the translation validation approach. [PSS98a] uses it to verify the compilation of an intermediate transition-system language to imperative C code. The thesis of Auger [Aug13] implements the normalization of a Lustre-like language using this approach. The verified compiler presented in [Shi+17; Shi+19] uses direct verification for some of its passes. Its semantic model is however closer to that of an imperative language than the ones usually used for Lustre-like languages.

In the Vélus compiler, all passes are verified directly, except for scheduling, which uses translation validation. In this dissertation, we will outline the main invariants of these direct proofs of correctness, and summarize the central proof steps.

1.2 Programming with Vélus and State Machines

Dataflow-synchronous language, like Vélus, can be used to specify the behavior of embedded systems. In this section, we introduce our language, and in particular the constructions added in this thesis, through the example of a simple embedded system: a stepper motor. The inside of a stepper motor is sketched in figure 1.3. The central *rotor*

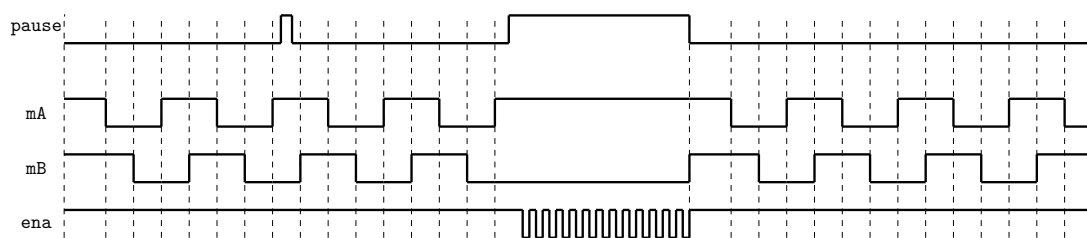


Figure 1.4: Example behavior of the stepper motor

```

node count_up(inc : int)
returns (o : int)
let
  o = (0 fby o) + inc;
tel

```

inc	50	50	50	50	50	50	50	...
o	50	100	150	200	250	300	350	...

Figure 1.5: Measuring the duration of a phase

is made from a magnet, and turns within a fixed *stator*. The stator has two windings, labelled A and B on the figure. Passing current through a winding creates a magnetic field that acts on the rotor. Energizing the windings with the sequence of phases A^+/B^+ , A^-/B^+ , A^-/B^- , A^+/B^- moves the rotor clockwise from the upper-right in 90° steps. We will call this basic behavior the *drive sequence*, illustrated in [figure 1.3](#), at right.

An example trace of the control signals is presented in [figure 1.4](#). The enable signal `ena` indicates if electrical current is sent to the motor. The `mA` (respectively `mB`) boolean signal indicates, if `ena` is `true`, whether current is sent to pole A^+ or A^- (respectively B^+ and B^-). The speed of the motor depends on the length of a phase, delimited by *step* signals, which are denoted by a dashed vertical line. Note that the first phase is longer, to allow the rotor to gain momentum. Additionally, the rotation of the motor may be *paused*, when the `pause` signal is `true`. During the pause, the rotor must be held in place. This means staying in the same phase and not issuing any step signal before the pause ends. If the pause ends before the end of the current phase, like the first one in the chronogram, this does not have any effect. If it exceeds it, the electrical current sent to the windings must be reduced to avoid physical damage. This is done by modulating the `ena` signal. Finally, if the pause lasts for too long, the next phase after the pause should, again, be longer, in order for the rotor to regain momentum.

Basic dataflow We now specify a program that generates control signals for this motor as a Vélus program. We first present a simple function that counts upward from zero by a given increment. We will later instantiate it to measure the duration of a motor phase. The definition and an example trace are shown in [figure 1.5](#). This *node*, called `count_up`, maps an input stream `inc` to an output stream `o`. Its body contains a single equation that defines `o` as zero *followed by* (`fby`) the pointwise addition of itself and `inc`.

```

node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  last mA = true; last mB = true;
  switch step
  | true do
    (mA, mB) = (not (last mB), last mA)
  | false do
    (mA, mB) = (last mA, last mB)
  end;
tel

```

step	F	F	T	F	F	T	F	...
mA	T	T	F	F	F	F	F	...
mB	T	T	T	T	T	F	F	...

Figure 1.6: Generating the drive sequence

Switch blocks and shared variables We now describe how the three digital signals `mA`, `mB` and `ena` controlling the motor are produced. Assume for now that the motor is not paused, and thus that `ena = true`. To generate the sequence of phases described above, the `drive_sequence` node in figure 1.6 alternates the values of `mA` and `mB` every time its input `step` is true. A sample execution of the node is presented on the right. In this dissertation, we will abbreviate `true` and `false` as respectively `T` and `F`. The body of this node contains a `switch` block: the value of `step` determines, at each cycle, which set of equations is active and defines the output signals.

It is tempting to put `(mA, mB) = (true, true) fby (not mB, mA)` in the true branch, but this `fby` would not allow us to refer to the previous values of these streams calculated in the false branch. Instead we use *shared variables* [CPP05, §3.2] by declaring `mA` and `mB` with *initial last values* and writing `last mA` and `last mB` to access their previous values. The overall effect is to advance to the next phase when `step` is true and to otherwise hold the current phase.

Omitting the body of the false branch, shown in grey, yields an equivalent program. Indeed, the semantics and compiler complete partial definitions for variables declared with initial last values. This allows the user to define programs in a more imperative style, with state variables that are only updated when explicitly defined.

While the `drive_sequence` node uses a boolean condition, Vélus supports general enumerated types¹. The boolean type is defined as an enumerated type with two constructors, `true` and `false`.

Reset blocks The duration of each phase and the value of `ena` are controlled by the `feed_pause` node presented in figure 1.7. We will detail the state-based logic of this node shortly. For now, we focus on its first three lines. They define a local variable `time` using the `count_up` node defined earlier. This node is instantiated with an increment of 50. The counter is reset after each phase change, that is, when `step` was true in the previous instant. Since the value of `time` is itself needed to determine that of `step`, the delay

¹Thanks to the work of Lélío Brun

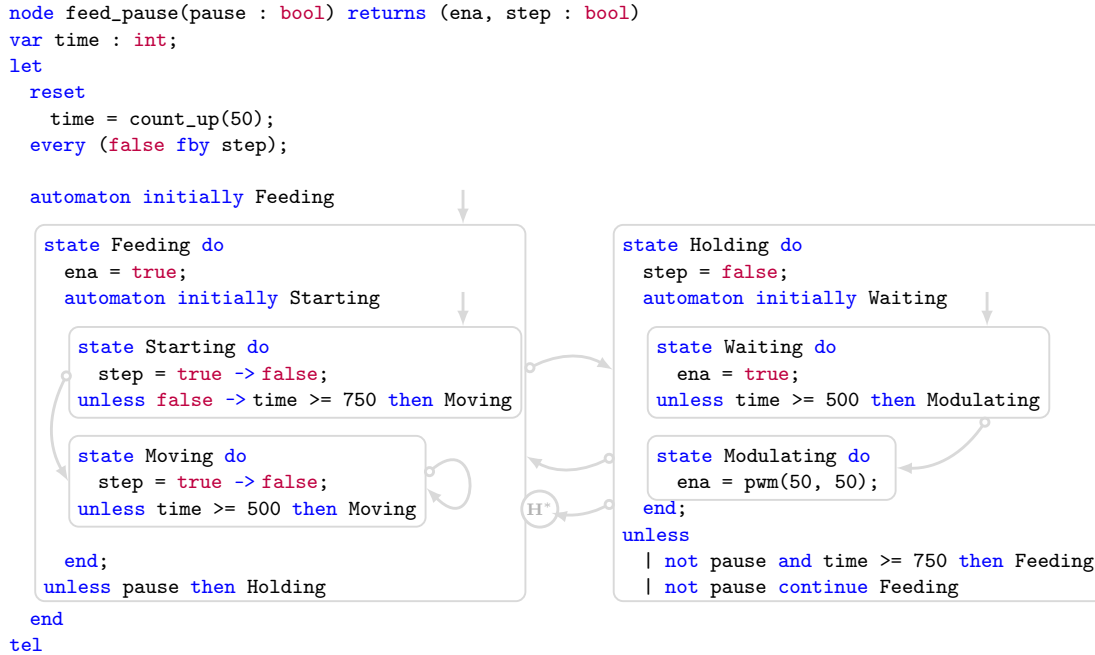


Figure 1.7: Controlling the steps

introduced by `fby` is necessary to avoid an instantaneous dependency cycle between `step` and `time`.

State Machines The behavior described above is specified by the automaton of figure 1.7. We have superimposed a graphical notation on top of the Lustre code to outline the control behavior. The automaton has two states: the initial **Feeding** state, at left, and **Holding**, at right. Transitions are listed below states after the `unless` keyword. When **Feeding** is active, `ena` is always true and a nested automaton defines the duration of each phase. As described, the first phase after a long pause should be longer as the rotor (re)gains momentum. Both inner states define the value of `step` with the initialization operator (`->`). This way, `step` is only true in the instant a state is entered. When the `pause` input is true, the automaton enters the **Holding** state. Here, `step` is always false, and the value of `ena` is defined by a nested automaton. If the current phase duration exceeds 500, the inner automaton enters the **Modulating** state, where `ena` is defined by Pulse-Width-Modulation (PWM) using the `pwm` node (not shown). The inputs of `pwm` specify the on and off periods of the generated PWM signal. The **Holding** state has two outgoing transitions. Both of them fire only if the `pause` input becomes false, and return to the **Feeding** state. If the last phase duration exceeds 750, then the rotor will have lost momentum. In this case, the first transition specifies that the **Feeding** state must be entered “with reset”. This means, in particular, that the internal state machine inside **Feeding** is reset, and that the **Starting** state will then be entered again. If the phase duration is less than 750, the `continue` keyword of the second transition specifies that

```
state Counting do
  time = count_up(50);
until step then Counting
```

Figure 1.8: Replacing `reset` with a state machine with weak transitions

the state must be entered “with history” [Har87]. The internal state machine will not be reset, and will continue executing in the state it was in before `pause` became `true`.

All transitions in figure 1.7 are marked as *strong* by the `unless` keyword [Pou06, §1.6.1]. A strong transition aborts a state immediately, specifying a state to enter in the same instant. Figure 1.8 defines `time` using the `until` keyword to specify a *weak* transition [Pou06, §1.6.2]. A weak transition exits a state after it has been active, specifying a state to enter at the next instant. This definition is equivalent to the one given with a `reset` block in figure 1.7, because the transition is done “with reset”, which happens in the next cycle after `step` equals `true`. For this state machine, it is necessary to use a weak transition rather than a strong one to avoid an instantaneous dependency cycle between `step` and `time`.

1.3 Overview of the Vélus Compiler

The work described in this thesis has been implemented by extending the Vélus compiler. We first recall the structure of the existing compiler.

Compilation Chain The architecture of Vélus is presented in figure 1.9. Each block in the chain represents a different intermediate language, with a dedicated AST. The compilation chain starts with a parser, implemented using the Menhir parser generator for Coq [PR16; JPL12]. The produced term is untyped. The elaboration pass adds type and clock-type annotations to each expression in the program. It may fail if the program is not well formed. A final analysis checks that the program does not contain dependency cycles in its definitions, which would make the program impossible to compile. The elaborated Lustre term is then successively simplified by a series of source-to-source rewritings. Each of these passes aims at replacing a more complex construct of the language with a composition of simpler ones. Once the term is sufficiently simplified, it is transcribed into the Normalized Lustre (NLustre) AST. A first series of optimizations is applied. These are the optimizations that are most practical to specify for dataflow languages. Applying them on NLustre means we can take advantage of its restricted syntax to reduce the number of cases both in the algorithms and their proofs of correctness. The program is then translated into the Synchronous Transition Code (Stc) language. This language was specifically designed to facilitate scheduling [POPL20; Bru20]. Once the program is scheduled, it is translated into an imperative Object Code Language (Obc) program. Optimizations that are only possible or easier to specify for imperative programs are

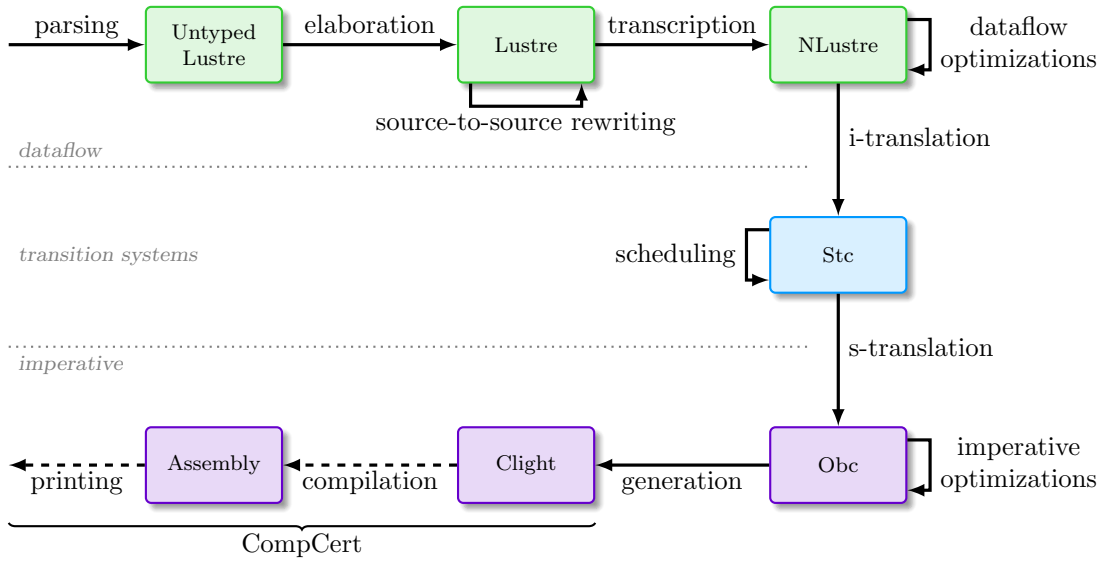


Figure 1.9: The architecture of Vélus

then applied. Finally, Vélus generates a program in Clight, the input language of the CompCert compiler, which then produces assembly code for a specified hardware target.

The intermediate languages of Vélus can be classified into three categories. Lustre and its untyped and normalized forms are dataflow languages, where the semantics are specified as relations over infinite streams. Stc is a transition system, where at each cycle, a transition function is applied to transform the internal state of the system. Finally, Obc and the languages used in CompCert have more typical imperative semantic models.

Compiler Correctness Verifying that this compilation chain is correct means proving that, for any Lustre program G compiled to an assembly program P , the behavior of P “corresponds” to the behavior of G . However, as discussed, the semantic model of assembly in CompCert is very different from that of the source Lustre language. The former describes a step-by-step execution, while the latter is defined as a set of relations over infinite streams. To relate these semantic models, we use the notion of “trace” defined in CompCert. A trace consists in a possibly infinite sequence of observable events, that is, interactions of the program with the outside world. Two of these events are the volatile load (VLoad) and store (VStore), that read from and write to a volatile variable. We use these events to model the relation between dataflow and imperative programs, as presented in [theorem 1](#). If the main node f of program G associates input streams xs to output streams ys , then we expect that the assembly program P generates an infinite trace perpetually reading the expected inputs of the node and writing the corresponding outputs.

Theorem 1 (Compiler Correctness 🍌 [VelusCorrectness.v:223](#))

$$\begin{array}{l} \text{if } G \vdash f(xs) \Downarrow ys \\ \text{and } \text{compile } G \ f = \text{OK } P \\ \text{then } \exists T, P \Downarrow T \wedge T \sim \langle \text{VLoad}(xs_{(n)}). \text{VStore}(ys_{(n)}) \rangle_{n=0}^{\infty} \end{array}$$

```

Theorem behavior_asm:
∀ D G Gp P main ins outs,
  elab_declarations D = OK (exist _ G Gp) ->
  lustre_to_asm (Some main) G = OK P ->
  wt_ins G main ins ->
  wc_ins G main ins ->
  sem_node G main ins outs ->
  ∃ T, program_behaves (Asm.semantics P) (Reacts T)
    /\ bisim_IO G main ins outs T.

```

Listing 1.1: Compiler Correctness Theorem 🍌 [VelusCorrectness.v:223](#)

The mechanization of this theorem, as used in Vélus, is presented in [listing 1.1](#). This statement has a few differences and restrictions compared with the ideal pen-and-paper theorem.

1. The `compile` function is split into two functions: `elab_declarations` which implements the elaboration pass, and `lustre_to_asm`, which implements the rest of the compilation chain. This is because there is no semantic model associated to untyped terms before the elaboration. The semantic judgment `sem_node G main ins outs` refers to the elaborated term `G`. This means that only the correctness of function `lustre_to_asm` is ensured by this theorem.
2. The parameter `main` indicates the name of the main node. It is passed as an option to the compilation function. If `None` is passed, it means the program should be compiled “as a library” without generating an input-output C function. In that case, the correctness theorem does not apply.
3. We require that the inputs streams of the node respect the declared types and clock types of the node. For instance, if a floating-point number is passed to a boolean input, the behavior of the compiled program is unspecified.
4. The `bisim_IO` relation implements the \sim operator used in the above theorem. Its specification depends on the names of inputs and outputs of the `main` node, as they correspond to the names of the volatile variables being read and written.

The lemmas and invariants presented in the rest of this dissertation will also be simplified, compared to the Coq version. We do this to avoid overwhelming the reader with the numerous details inherent to the mechanized proof of a realistic compiler. We instead focus on what we think is the meaningful core of these proofs. The other details are available in our Coq artifact and its documentation, as described below.

1.4 Prototype Implementation

The source code of the Vélus compiler is available at:

<https://github.com/INRIA/velus>

Every definition or theorem stated in this document is presented with a cross reference to this mechanization, signalled by 🐔. If this document is read as a PDF, the cross references are clickable links to the online documentation of Vélus.

An online, interactive version of the compiler is available at:

<https://velus.inria.fr/phd-pesin/try-velus/>

It is built from the Coq sources of the compiler, extracted to OCaml and compiled to JavaScript using `Js_of_ocaml` [VB14]. It includes a verified interpreter for the Obc intermediate language, which allows for programs to be simulated.

1.5 Organization

The following chapters describe the choices we have made in implementing the new constructs presented in this introduction in the context of the Vélus compiler. After having presented these choices formally, each chapter discusses them in relation with related work.

Chapter 2 presents the formalization of the Vélus front-end language. The new definitions for control blocks are presented as a conservative extension of the existing formalization of Lustre. The chapter presents the abstract syntax of the language, details its dynamic semantics, and describes the important points of its clock-type system. Along with the typeset definitions, it gives a taste of their mechanization in the Coq Proof Assistant.

Chapter 3 discusses the verified dependency analysis used in Vélus. It describes the implementation and verification of a graph analysis in Coq. It shows how to build an induction principle for programs that do not contain dependency cycles. This principle is applied to prove two central properties of the semantic models: determinism and clock correctness.

Chapter 4 focuses on the front-end of the Vélus compiler. It shows how state machines, `switch` blocks, and nested local scopes are successively translated into a smaller subset of the language. For each translation pass, it discusses the main difficulty in the associated correctness proof, and presents a simplification of the invariant used in the Coq proof.

Chapter 5 examines the middle-end of the Vélus compiler. It gives a brief presentation of each of the intermediate languages, and details the changes that were necessary to support the compilation of the new constructions. It especially focuses on the efficient compilation of nested `reset` blocks and `last` variables. Finally, it outlines the relation between the semantic models of each of the intermediate languages, and the main ideas behind the proofs of correctness of compilation passes.

Related Publications

- [JFLA21] Bourke, Jeanmaire, Pesin, and Pouzet,
“Normalisation vérifiée du langage Lustre”,
at Journées Francophones des Langages Applicatifs 2021.
This work describes the normalization passes that transforms a Lustre program into its restricted form, NLustre. Each of the three successive passes is described, along with its correctness proof.
- [EMSOFT21] Bourke, Jeanmaire, Pesin, and Pouzet,
“Verified Lustre Normalization with Node Subsampling”,
at International Conference on Embedded Software 2021.
This work details the semantics of the Vélus core dataflow language. It discusses the normalization pass, and why the clock correctness property is integral to proving its correctness.
- [JFLA23] Bourke, Pesin, and Pouzet,
“Analyse de dépendance vérifiée pour un langage synchrone à flot de données”,
at Journées Francophones des Langages Applicatifs 2023.
This work presents a first version of the semantics of `switch` and local blocks. It details the dependency analysis of Vélus of these constructions. Finally, it describes the proof of semantics determinism.
- [EMSOFT23] Bourke, Pesin, and Pouzet,
“Verified Compilation of Synchronous Dataflow with State Machines”,
at International Conference on Embedded Software 2023.
This work details the semantics of the extended Vélus language with local blocks, `switch` and `reset` blocks, and hierarchical state machines. It explains the compilation passes for each of these constructs, and gives a taste of their correctness proof.

Extending Vélus with Control Blocks

In this second chapter, we present the source language of the Vélus compiler. We first introduce the extended syntax of the Vélus language. We then present some central Coq definitions used in our mechanization, in particular regarding the representation of infinite streams. [section 2.4](#) recalls the relational synchronous semantics for the dataflow core of the language, as introduced in [Bru20; POPL20]. The following sections then demonstrate how this semantic model can be extended to support the high-level control blocks introduced in the previous chapter. The last section of the chapter discusses and compares our design choices with the related work.

2.1 Syntax of the Vélus source language

The syntax of the Vélus language is presented in [figure 2.1](#). In addition to the native constants already present in [Bru20; POPL20], the language handles enumerated constants, thanks to the work of Léo Brun. Expressions can refer to the current, or the `last` value of a variable. The unary and binary arithmetic and logic operators are the same as in CompCert C. The `fbv` operator induces a delay by taking an initial value from its left operand, followed-by the previous value of its right operand. The initialization arrow (`->`) takes the value from its left operand only at the first cycle, but does not induce a delay. A `when` is used to sample a stream when its condition is equal to a fixed enumerated constant. The opposite operator is `merge`, which combines complementary sampled streams into a faster one. A condition variable is used to control which of the branches of the `merge` should be chosen. The correct use of `when` and `merge` is ensured by a static “clock-type” system that we will detail later. The `case` operator generalizes the `if/then/else` construct for enumerated types. Contrary to the `merge`, all branches of the `case` must be active at the same time. Finally, an expression may instantiate a node, and the internal state of the instantiated node may be `reset` on a boolean signal.

Some of these operators (`fbv`, `->`, `when`, `merge` and `case`) are applied to lists of

$$\begin{aligned}
e ::= & c \quad | \quad C \quad | \quad x \quad | \quad \text{last } x \quad | \quad \diamond e \quad | \quad e \oplus e \\
& | \quad e^+ \text{ fby } e^+ \quad | \quad e^+ \rightarrow e^+ \\
& | \quad e^+ \text{ when } C (x) \quad | \quad \text{merge } x (C \Rightarrow e^+)^+ \\
& | \quad \text{case } e \text{ of } (C \Rightarrow e^+)^+ \\
& | \quad f (e^+) \quad | \quad (\text{reset } f \text{ every } e) (e^+) \\
\text{nodedecl} ::= & \text{node } f (var^+) \text{ returns } (var^+) \text{ blk} \\
\text{blk} ::= & x^+ = e^+ \\
& | \quad \text{var } var^* \text{ let } blk^+ \text{ tel} \\
& | \quad \text{switch } e (C \text{ do } blk^+)^+ \text{ end} \\
& | \quad \text{last } x = e \\
& | \quad \text{reset } blk^+ \text{ every } e \\
& | \quad \text{automaton initially } autinits (\text{state } C \text{ autscope})^+ \text{ end} \\
& | \quad \text{automaton initially } C (\text{state } C \text{ do } blk^+ \text{ unless } trans^+)^+ \text{ end} \\
\text{autinits} ::= & C \quad | \quad \text{if } e \text{ then } C \text{ else } autinits \\
\text{autscope} ::= & \text{var } var^* \text{ do } blk^+ \text{ until } trans^+ \\
\text{trans} ::= & \text{if } e \text{ continue } C \quad | \quad \text{if } e \text{ then } C \\
\text{var} ::= & x : ty \text{ on } ck \\
\text{typedecl} ::= & \text{type } tx = C^+ \\
G ::= & \text{typedecl}^* \text{ nodedecl}^+
\end{aligned}$$

Figure 2.1: Abstract Syntax of the Vélus Language

expressions. This allows, for instance, to write the expression $(0, 0) \text{ fby } f(x)$, where f has two outputs, and means that each expression produces a list of streams. While this complicates some of the definitions, algorithms and proofs, we want to provide a language with as few arbitrary restrictions as possible.

A node takes a non-empty list of inputs, and produces a non-empty list of outputs. The values of the outputs are defined in the encapsulated block. In the original language, a block blk could only contain equations $x^+ = e^+$, which define the values of the variables at left to be the ones of the expressions at right. This work enriches the block structure with the following constructions. Blocks of local declarations may be arbitrarily nested. The `switch` block controls the activation of its branches according to the enumerated value of its condition, as was shown in the `drive_sequence` example of the introduction. For each variable on which `last` is used, an initialization equation `last x = e` must be provided. The modular `reset` operator is generalized to support resetting the state of any block. The state machines, denoted by the `automaton` keyword, allow for the definition of complex modal behaviors, such as the one of the `feed_pause` node in the introduction. They support both weak (`until`) and strong (`unless`) transitions. We discuss later why

the two type of transitions cannot be mixed in our language. Transitions *trans* of either types are formed with a condition, a state to transition to, and either the keyword **then** or **continue**, indicating whether or not the entered state should be reset on entry. The order of the transitions in a state determines which one should be followed if multiple conditions are true at the same time. For state machines with weak transitions, the initial state is specified by a list of boolean conditions *autinits*. Moreover, for state machines with weak transitions, each state may declare local variables which may be used in the conditions of the transitions.

Finally, a program consists in a list of enumerated type declarations, followed by a non-empty list of node declarations.

2.1.1 Representation of the AST in the Coq Proof Assistant

The abstract syntax of the Vélus language is represented in Coq using the **Inductive** data-types presented below. We now discuss the major differences between the pen-and-paper definitions presented in the previous section and these mechanized ones.

Most expressions in the **exp** type have static type and clock-type annotations. Clock-types are represented using the inductive **clock** type, which is either the base clock **Cbase** or a sampled clock **Con**. Annotations of type **ann** are used to indicate the type and clock type of each stream generated by an expression. Annotations of type **lann** are used when all streams generated by an expression have the same clock type, but not necessarily the same type. These annotations are added during Elaboration. Functions **typeof** and **clockof** return the types and clock types associated with an expression by reading these annotations. These annotations also appear in the Coq predicates formalizing the type and clock-type systems described in [appendix A](#). Finally, they are used as witnesses for a simple and efficient decision procedure checking if a program is indeed well typed.

In both CompCert and Vélus, identifiers of type **ident** are represented by strictly positive numbers in a binary representation. The inductive type definition from the Coq Standard Library is presented in [listing 2.2](#). Constructor **xH** represents the number 1. Using constructor **xI** (respectively **x0**) adds 1 (respectively 0) as the new least significant bit of the number. For instance, **(xI (x0 (xI xH)))** represents the binary number 1101, that is the decimal number 13. This representation is relatively space efficient, compared to the traditional Peano encoding of natural numbers. Moreover, it allows for a trivial definition of tries data structures as binary trees. This definition is used to give an efficient functional implementation of sets of positives and associative maps with positives as keys.

Constructors for enumerated types are represented by the **enumtag** type, which is simply **nat**, the type of natural numbers. For a type declaration **type** $\tau = C0 \mid \dots \mid Cn$, constructor **C0** is represented by the natural 0, and **Cn** by the natural **n**.

Both the **branch** and **scope** inductive types are used in the definition of **block**. A **branch** represents one of the mutually-exclusive cases of a control block. It can represent one of the cases of a **switch** or one of the states of an **automaton**. A **scope** encapsulates local stream definitions. It can appear directly in a block or within the **branch** of a state machine. In the latter case, the weak transitions of the state machine appear within the scope, so they can refer to definitions local to the branch. Both of these inductives

```

Definition enumtag : Type := nat.
Inductive clock : Type :=
| Cbase : clock
| Con   : clock -> ident -> type * enumtag -> clock.
Definition ann : Type := (type * clock).
Definition lann : Type := (list type * clock).

Inductive exp : Type :=
| Econst : cconst -> exp
| Eenum  : enumtag -> type -> exp
| Evar   : ident -> ann -> exp
| Elast  : ident -> ann -> exp
| Eunop  : unop -> exp -> ann -> exp
| Ebinop : binop -> exp -> exp -> ann -> exp
| Efby   : list exp -> list exp -> list ann -> exp
| Earrow : list exp -> list exp -> list ann -> exp
| Ewhen  : list exp -> (ident * type) -> enumtag -> lann -> exp
| Emerge : ident * type -> list (enumtag * list exp) -> lann -> exp
| Ecase  : exp -> list (enumtag * list exp) -> lann -> exp
| Eapp   : ident -> list exp -> list exp -> list ann -> exp.

Inductive branch A := Branch : list (ident * ident) -> A -> branch A.

Definition decl : Type := ident * (type * clock * ident * option ident).
Inductive scope A := Scope : list decl -> A -> scope A.

Inductive auto_type := Weak | Strong.
Definition transition : Type := exp * (enumtag * bool).

Inductive block : Type :=
| Beq : list ident -> list exp -> block
| Blast : ident -> exp -> block
| Breset : list block -> exp -> block
| Bswitch : exp -> list (enumtag * branch (list block)) -> block
| Bauto :
  auto_type ->
  clock ->
  list (exp * enumtag) * enumtag ->
  list (enumtag *
        branch (list transition * scope (list block * list transition))) ->
  block
| Blocal : scope (list block) -> block.

Record node : Type := mk_node {
  n_name      : ident;
  n_in       : list (ident * (type * clock * ident));
  n_out      : list decl;
  n_block    : block;
  [...]
}.

```

Listing 2.1: The Vélus Abstract Syntax Tree  Lustre/LSyntax.v:46

```

Inductive positive : Set :=
| xI : positive -> positive
| x0 : positive -> positive
| xH : positive.

```

Listing 2.2: Binary representation of positive integers

take a type parameter `A` representing the type of underlying syntactic element. This parametricity allows us to define and re-use common definitions and proofs for these syntactic elements. This is particularly useful in the case of `scope`, where proofs are often complex and full of administrative details that would be tedious to repeat. The downside is that it makes the invariants for these proofs particularly long and intricate, as they must state the property for both the parameter object `A` and the syntactic constructor `scope A` or `branch A`. We tried to resolve this issue by defining a custom induction scheme, but this does not seem to be possible with such general definitions.

We provide only one constructor `Bauto` to handle both types of state machines (weak or strong transitions). The two types of state machines are distinguished by the first `auto_type` parameter. The well-typing predicate ensures that an automaton marked `Weak` (respectively `Strong`) does not contain strong (respectively weak) transitions. Each state machine is also annotated by a static clock type. As we will discuss in [section 2.9](#), this annotation is necessary to give a semantics to state machines. State machine transitions are represented by a list of triplets formed by a condition expression, a state tag, and a boolean indicating if this transitions should reset the next state on entry (`true = then C` or `false = continue C`). The order of this list gives the relative priority of transitions, from high to low. The initial state is specified by a list of pairs of conditions and `enumtag`, and by a final `enumtag` (`otherwise C`) which gives the initial state if all conditions are false. For a `Strong` state machine, the list must be empty.

Each node is represented by a `Record`. It contains the name of the node, the input and output declarations, and the toplevel block. Note that the types for inputs and outputs are different, as outputs can optionally be declared with a `last` expression. The other fields of the `Record`, not shown in the listing, contain static invariants of well-formed nodes, expressed over the previous fields. These invariants are complementary to the type and clock-type systems, and important for proving the correctness of compilation passes. They state properties like “the variables defined by equations correspond exactly to the variables declared in the node”. We give a precise inductive definition for these invariants in [appendix A.1](#). Expressing them as obligations of a dependant record is a classic technique inspired by CompCert. It allows us to always have access to them, without having to add extra assumptions to every lemma. On the other hand, this means that, when constructing a node during a compilation pass, we need to prove immediately that these obligations hold. For simplicity, we omit these invariants in the discussions of correctness proofs and invariants in the remainder of this dissertation.

```

Module Type OPERATORS.
  (* Back-end Values and Types *)
  Parameter cvalue : Type.
  Parameter ctype  : Type.
  Parameter cconst : Type.

  (* Velus Values and Types *)
  Inductive value :=
  | Vscalar : cvalue -> value
  | Venum   : enumtag -> value.

  Inductive type :=
  | Tprimitive : ctype -> type
  | Tenum      : ident -> list ident -> type.

  (* Conversions between Values, Types and Constants *)
  Parameter ctype_cconst : cconst -> ctype.
  Parameter sem_cconst  : cconst -> cvalue.
  Parameter init_ctype  : ctype -> cconst.

  (* Operators *)
  Parameter unop  : Type.
  Parameter binop : Type.

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

  Parameter type_unop  : unop -> type -> option type.
  Parameter type_binop : binop -> type -> type -> option type.
End OPERATORS.

```

Listing 2.3: Interface with the back end 🐔 `Operators.v:9`

2.2 Abstracting the CompCert Back End

In the previous section, we did not define some of the types used in expressions (`cconst`, `type`, `unop`, ...). We now discuss the definitions of these types. Vélus is built on top of the CompCert verified compiler. As such, we chose to have the types and values manipulated by Vélus be those of the host language, Clight. However, the definition of these types and values are not so simple: they depend on the architecture targetted by CompCert, and offer a lot of different options (signedness, size of ints, ...). The details of these low-level representations are irrelevant to Vélus, which, as a dataflow language, is used to specify high-level control. In order to hide these details, we provide a high level abstraction of the back-end types and values. The `OPERATORS` module signature presented in [listing 2.3](#) captures this abstraction. The `Parameter` Coq command specifies the type of a definition which must be provided in any `Module` implementing this signature.

The first three parameters are types: `cvalue` represents back-end semantic values, `ctype` represents back-end types, and `cconst` represents back-end syntactic constants. In

addition to the primitive `cvalue`, Vélus supports enumerated values. In an enumerated value, each constructor is represented by an `enumtag` represented by a natural number, using the Peano encoding. Similarly, a Vélus type is either a primitive type from the back end, or an enumerated type, represented as a name and a list of constructors. In particular, booleans are represented in Vélus by an enumerated type, with name `"bool"` and constructors `["false"; "true"]`. In the back end, these abstract enumerated types are represented as machine integers. For instance, for a type `type t = A | B | C`, value `A` is represented by 0, `B` by 1, and `C` by 2. In particular, the boolean `false` is represented by 0 and `true` by 1, as would be expected. Vélus does not yet support more general sum types, which would require a more complex representation in the back end.

The next three parameters are conversion functions between the primitive types, values and constants: `c_type_cconst` gives the type of any primitive constant; `sem_cconst` gives a semantic value for a syntactic constant; `init_c_type` provides a default constant for a given primitive type (for instance, 0 for integers, 0.0f for single-precision floating-point numbers, etc).

The next parameters specify unary and binary operators, along with partial functions giving the semantics and typing of these operators. In addition to values, the semantic functions `sem_unop` and `sem_binop` take as input the types of these values. This is because the semantics of an operator depends on the types of its input values. For instance, `1 + 2` and `1.0 + 2.0` do not have the same semantic, and are not compiled using the same arithmetic instructions. These semantic functions are partial for two reasons. First, an operator is not defined for all possible input types. For instance, the bitwise operators are not defined for floating-point numbers, and addition is not defined for two different input types. The type-system prevents this class of undefined behavior. The second class consists in runtime undefined behaviors, such as division by zero or integer overflow. These behaviors are harder to protect against statically. In Vélus, we follow the choice of CompCert of not giving a semantics to a program that would encounter an undefined behavior. In this document, we write $\diamond_{ty_1}(v_1)$ and $\oplus_{ty_1 \times ty_2}(v_1, v_2)$ for the values returned by these functions, only when they succeed. The last two functions, `type_unop` and `type_binop` give the type of the value produced by an operator for given input types. As discussed, not all operators support all input types, and these functions are also partial. We write $\vdash \diamond_{ty_1} : ty$ and $\vdash \oplus_{ty_1 \times ty_2} : ty$ to state that these functions succeed and return type `ty`. Note that all these operators manipulate Vélus values and types. In particular, logical operators may be applied to boolean operands. This means that the definition of `sem_unop` and `sem_binop` are not taken directly from the back end, but are extended to support boolean enumerated values. We do not detail these extended definitions here. They can be found at [🐔 ObcToClight/Interface.v:181](#) and [🐔 ObcToClight/Interface.v:254](#).

2.3 Representing Infinite Sequences

Programs written in reactive or synchronous languages are designed to run forever. Modeling their behavior therefore necessitates modeling this notion of infinity. In a dataflow synchronous language like Lustre, this means representing the infinite sequences –

or streams – of values produced by each syntactic component of a program. In Vélus, two different representations are used for these sequences: the indexed representation, where streams are function from natural numbers to values, and the coinductive representation, where a stream is built by infinite applications of a constructor that adds an element at its front. Below, we discuss the pros and cons of each representation with regard to mechanizing the semantics of Lustre.

2.3.1 Indexed Streams

One central operation on sequences is indexing: accessing an element of a sequence by its index. In a finite sequence, this operation is not trivial, since accessing an element may fail if the index is too large. Streams however do not have this issue and indexing is total. A stream can therefore be represented as an indexing function that associates a natural number n to the n th value in the sequence.

```
Definition stream A := nat -> A.
```

Listing 2.4: Indexed stream definition  [IndexedStreams.v:32](#)

One issue with infinite structures like streams is expressing equality. In Coq, equality is defined as a binary relation using the `Inductive` definition shown below. The only constructor requires its two arguments to be syntactically the same modulo reduction. Typical Coq proofs establish the equality of two terms by alternating steps of rewriting (that is, using the transitivity of `eq`) and reducing the terms, until they are indeed the same. The inductive definition of `eq` comes with the `eq_rect` property, which allows to replace any term with one that is proven to be equal under any context.

```
Inductive eq {A} (x: A) : A -> Prop := eq_refl : eq x x.
(* eq_rect: ∀ A (x: A) P, P x -> ∀ y, y = x -> P y *)
```

Listing 2.5: Leibniz equality [Coq]

When manipulating functions, this notion of equality becomes impractical, as it is not possible to rewrite under a function binder. For instance, it is not possible to prove, in Coq, that `(fun x => x + x) = (fun x => x * 2)`.

The following definition is for extensional equality on indexed streams. This relation is weaker than that of `eq`. The equivalence of two indexed streams can be proven by establishing that their elements are point-wise equal. The main drawback of such a definition is that we no longer have the `eq_rect` property. One must prove, for each property of interest P , that $\forall (x : \text{stream } A), P\ x \rightarrow \forall y, \text{eq_str } y\ x \rightarrow P\ y$. This happens to be true for most interesting streams properties, since they are only concerned about the elements of streams, but this is still tedious to prove systematically.

Another issue of this representation is that defining the `cons` operation, which adds an element at the start of the stream, is not trivial. A possible definition for `cons` is

```
Definition eq_str {A} (xs ys : stream A) := ∀ n, xs n = ys n.
```

Listing 2.6: Equality of indexed streams  IndexedStreams.v:39

presented below. As expected, indexing 0 in `cons hd tl` returns `hd`, and indexing with $n > 0$ is equivalent to indexing $n - 1$ in `tl`.

```
Definition cons {A} (hd: A) (tl: stream A) : stream A :=
  fun n => match n with
    | 0 => hd
    | S n => tl n
  end.
```

Listing 2.7: Defining `cons` for indexed streams

We want the mechanized semantics of dataflow operators to emulate the ones present in the literature, in particular in [CP03, Figure 2]. The `cons` operation is pervasive in these definitions, especially in the stateful operators like `fbv`. Although indexed streams are used in the semantics of intermediate languages of Vélus, we prefer to use a representation where `cons` is a more primitive operation for the front-end semantics.

2.3.2 Coinductive Streams

A second possible definition for infinite streams is based on coinductive types. While the values of inductive types can only be built from a finite number of constructor applications, coinductive values can be built from an infinite sequence of constructor applications. Conversely, the definition of a coinductive type does not generate an induction principle, as the object is not well-founded. This means one can only reason on CoInductive type by case-analysis and primitive corecursion. We will see further what that means for proofs.

Below is presented the definition of the `Stream` type from the Coq Standard Library. It has a single constructor, `Cons`, which adds an element at the front of the stream. In the following, we write $x \cdot xs$ for `Cons x xs`.

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

Listing 2.8: CoInductive stream definition [GC04]

Accessing an element of the stream by its index can be done with the `Str_nth` function defined below. Note that the decreasing argument of the function is the natural n , and not the stream s . In the following, we write $s \# n$ for `Str_nth n s`.

The `init_from` function shown below allows one to build a coinductive stream from an indexed stream. It is easy to prove that $(\text{init_from } 0 \ f) \# n = f \ n$.

```

Fixpoint Str_nth {A} (n: nat) (s: Stream A) :=
  match n with
  | 0 => hd s
  | S m => Str_nth m (tl s)
  end.

```

Listing 2.9: Accessing an element of a coinductive stream

```

CoFixpoint init_from {A} (n: nat) (f: stream A) : Stream A :=
  f n · init_from (S n) f.

```

Listing 2.10: Build a coinductive stream from an indexed stream  [CoindStreams.v:417](#)

As with indexed streams, it is not practical to use the Leibniz equality `eq` to reason about coinductive streams. Indeed, coinductive streams do not reduce to a finite value. Instead, we define the pointwise equality on streams `EqSt` below, as a coinductive property. In the following, we write $xs \equiv ys$ for `EqSt xs ys`. As with indexed streams, using this definition unfortunately means having to prove manually that every property of interest is invariant under `EqSt`. This is indeed the case for the properties stated later in this document; we will not discuss any of these boilerplate proofs.

```

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

```

Listing 2.11: Pointwise equality for coinductive streams

The `CoFixpoint` command allows for the definition of a corecursive function. [Listing 2.12](#) presents the example of `map` which applies a function pointwise to every element of a stream. Arguments to corecursive calls do not need to be subterms of the initial arguments. However, the function needs to be productive: each corecursive call should be wrapped under a constructor, here `Cons`. A function that does not respect this constraint will be rejected by the termination checker of Coq.

```

CoFixpoint map {A B} (f : A -> B) (s: Stream A) : Stream B :=
  Cons (f (hd s)) (map f (tl s)).

```

Listing 2.12: Applying a function point-wise to every element of a coinductive stream

The following proof script establishes the equivalence of two coinductive streams built using `map` with two different, but extensionally equal, functions. The `cofix` tactic creates a coinduction hypothesis of the same form as the current goal. As with `CoFixpoint` definitions, calls to the coinduction hypothesis must be wrapped under a constructor. We decompose the stream `xs` into its head and tail, and immediately apply the only

constructor `Cons`. We need to prove the equality of the two heads and the equivalence of the two tails. The first goal is solved by `lia`, a tactic for solving linear arithmetic goals. For the second goal, we apply the coinduction hypothesis. This call is indeed guarded by the constructor applied above.

```

Lemma proving_EqSt : ∀ xs,
  map (fun x => x + x) xs ≡ map (fun x => x * 2) xs.
Proof.
  cofix CoFix. (* We reason coinductively *)
  intros [x xs'].
  constructor; simpl.
  - (* head, x + x = x * 2 *) lia.
  - (* tail, corecursive case *) apply CoFix with (xs:=xs').
Qed.

```

Listing 2.13: Proving equality for two coinductive streams

Although the above proof is simple, reasoning with `CoInductive` definitions can sometimes be tricky, as the interaction of tactics may produce unexpected unguarded terms, which can be very frustrating to debug. One solution may be to use a correspondence lemma between coinductive and indexed streams, and do the actual proof in the indexed context. For instance, for `map`, we can prove that $(\text{map } f \text{ } xs) \# n = f (xs \# n)$, and use this equation to simplify proofs. However, coinductive definitions do not always have simple indexed specification; for instance, the `fb` operator which we present later is difficult to characterize in this way, because the values of its output depends on previous values of its inputs.

2.4 The Core Dataflow Semantics of Vélus

In Lustre, dataflow nodes consist of a system of equations defining the values of node outputs in terms of inputs and other outputs. It is therefore natural to express the semantics of the language as a set of constraints between input and output streams, each equation inducing an additional constraint. These constraints can be formalized, and mechanized in Coq, as inductive rules over the abstract syntax of the language. In this section, we will detail the preexisting semantic rules used in Vélus [PLDI17; POPL20]. We mostly present a pen-and-paper formalization of these rules, but we also try to give a taste of how they are mechanized in Coq.

2.4.1 Histories and Equations

As stated above, we express the semantics of Lustre as constraints between the different named streams of the node. The central object of this semantic model is therefore the *history*, an environment that associates every variable in the node to an infinite stream. The chronograms presented to illustrate the executions of examples in the previous chapter are all examples of specific histories.

In Coq, we use functional maps to represent histories. The definition of a functional environment (type `FEnv.t`) is presented below. The type variable `K` represents the type of keys, and `A` the type of values. For some of the operations on environments to make sense, we require key equality to be decidable. This is enforced using Coq’s type-classes system, as demonstrated by the `fenv_key` class below. We do not place any restriction on the type of values.

```
Class fenv_key (A : Type) := { fenv_key_eqdec : EqDec A eq }.

Section FEnv.
  Context {K : Type} '{K_key : fenv_key K}.

  Definition t A := K -> option A.
End FEnv.
```

Listing 2.14: Functional environment 🐔 [FunctionalEnvironment.v:22](#)

```
Definition history := @FEnv.t ident (Stream svalue).
```

Listing 2.15: History 🐔 [CoindStreams.v:1445](#)

Histories are then represented as functional mappings where each variable is associated with a stream of synchronous values `svalue`. For now, we assume that the keys are the identifiers used in Vélus (type `ident`). Since identifiers are represented by positive integers using an inductive binary representation, their equality is indeed decidable. We will discuss the `svalue` type later. In the following semantic statements, we write $H(x)$ to denote the value associated to x in history H . This operation is only defined if x is associated to a stream in H , which we write $x \in \text{dom}(H)$.

In Vélus one semantic judgment is associated to each syntactic type (expressions, blocks, nodes). We will now present one rule for each of these judgments, focusing on the rules that constrain the history directly.

The first judgment $G, H, bs \vdash e \Downarrow vss$ states that “under global context G , history H and base clock bs , the expression e produces the streams vss ”. Here, the global context G associates the name of each node in the program to its definition. The history H , as we discussed, associates each variable in the node to a stream. We return to base clock bs in [section 2.4.2](#).

As an expression can be formed using lists of sub-expressions, each expression may produce several streams. For example, $(1, 2, 3)$ `fbym` (`t`, `if x then (y, z) else (z, y)`) produces three streams, and the second expression at right of `fbym` produces two. To encode this, we say that each expression produces a list of streams. We write $[vs]$ for the singleton list containing only the stream vs . In semantic rules for operators where operands are lists of expressions (like `fbym`), since each expression produces a list of streams, we get a list of lists of streams, which must be flattened to give a semantics to the parent

$$\begin{array}{c}
 \frac{H(x) \equiv vs}{G, H, bs \vdash x \Downarrow [vs]} \\
 \text{(a) Svar } \color{red}{\text{🐔 Lustre/LSemantics.v:143}}
 \end{array}
 \qquad
 \frac{\forall i, H(x_i) \equiv vs_i \quad G, H, bs \vdash es \Downarrow [vs_i]^i}{G, H, bs \vdash [x_i]^i = es}
 \qquad
 \begin{array}{c}
 \text{(b) Seq } \color{red}{\text{🐔 Lustre/LSemantics.v:255}}
 \end{array}$$

$$\frac{
 \begin{array}{c}
 G(f) = \mathbf{node} \ f(x_1, \dots, x_n) \ \mathbf{returns} \ (y_1, \dots, y_m) \ \mathit{blk} \\
 \forall i \in 1\dots n, H(x_i) \equiv xs_i \\
 \forall i \in 1\dots m, H(y_i) \equiv ys_i \quad G, H, (\mathbf{base-of} \ (xs_1, \dots, xs_n)) \vdash \mathit{blk}
 \end{array}
 }{
 G \vdash f(xs_1, \dots, xs_n) \Downarrow (ys_1, \dots, ys_m)
 }
 \qquad
 \begin{array}{c}
 \text{(c) Snode } \color{red}{\text{🐔 Lustre/LSemantics.v:364}}
 \end{array}$$

Figure 2.2: Semantics of variables, equations and nodes

expression. In the pen-and-paper semantic rules, this flattening is left implicit. In our Coq mechanization, we use the `concat : list (list A) -> list A` function to flatten these lists of streams. This additional detail can sometimes make the proofs more technical, but allows for more expressivity in the language.

The first rule in [figure 2.2](#) states that a variable in an expression produces the single stream vs . For this rule to hold, the variable x needs to be associated to the corresponding stream in the history. Note that we use the stream equality defined in [listing 2.11](#) in the premise of this rule. This gives flexibility in proofs that construct a semantic derivation.

The second rule concerns equations of the form $x^+ = e^+$. In general, the judgment $G, H, bs \vdash \mathit{blk}$ states that “the semantics of block blk is consistent with global context G , history H and base clock bs ”. Note that a block does not produce anything, but instead adds constraints on the history H . This is particularly true for the equation. The expressions at right of the equation produce a flattened list of streams. The first premise constrains the history : the stream associated with each of the variables at left of the equation must be equal to the corresponding stream produced by the expressions at right.

The third rule concerns nodes. The judgment $G \vdash f(xss) \Downarrow yss$ states that “the node with name f in G associates the input streams xss to output streams yss ”. This holds if there exists an history H where the names of inputs and outputs are associated to the input and output streams, and if H satisfies the constraints defined by the body of the node. Note that the history H is not exposed by this rule: it is given as an existential in the premises of this rule. In a sense, the history is encapsulated by the node, and only input and output streams are visible when a node is instantiated. This means that having a witness of the semantics of a node does not give us a lot of information about this internal history. In particular, it is not obvious that, given a particular input, there is only one history that satisfies this rule. We will discuss these issues in [chapter 3](#).

2.4.2 Sampling and Clock Typing

Vélus streams are sampled, meaning that some may be produced at a slower rate than others. Our semantics is synchronous, meaning that presence and absence are represented explicitly. In Coq, this is encoded using the `synchronous` type presented below. In the following rules, we write $\langle v \rangle$ for a present value v and $\langle \rangle$ for an absent value. For simplicity, we will omit these notations in the chronograms showing sample executions.

```

Inductive synchronous (A : Type) :=
| absent
| present (v: A).

Definition svalue := synchronous value.

```

Listing 2.16: Synchronous values 🐦 [Operators.v:171](#)

In a synchronous setting, these presences and absences constitute a rhythm for the stream. We can say that a stream xs is faster than a stream ys if xs is always present whenever ys is present. More formally, we call this rhythm the *clock* of a stream. It is computed using the `clock-of` function below: the `true` and `false` values in the result stream correspond to presences and absences in the input stream.

Definition 1 (Clock of a stream 🐦 [CoindStreams.v:1777](#))

$$\begin{aligned} \text{clock-of } (\langle \rangle \cdot vs) &\triangleq \text{F} \cdot \text{clock-of } vs \\ \text{clock-of } (\langle v \rangle \cdot vs) &\triangleq \text{T} \cdot \text{clock-of } vs \end{aligned}$$

Constant expressions have the fastest rhythm in the node: they produce a value every time the node is activated. The clock of streams produced by constants therefore correspond to the *base clock* of the node, bs . The resulting semantic rules are given in [figure 2.3](#). The base clock depends on the inputs of the node: intuitively, a node is activated every time at least one of its inputs is present. This is encoded by the `base-of` function presented in [definition 2](#). This choice of having a single base clock restricts the set of programs our compiler can accept: other languages, like Lucid Synchronic [Pou06] allow for a node having multiple parameters with unrelated clocks.

Definition 2 (Base Clock 🐦 [CoindStreams.v:1162](#))

$$\begin{aligned} \text{base-of } ((\langle \rangle \cdot xs_1), \dots, (\langle \rangle \cdot xs_n)) &\triangleq \text{F} \cdot \text{base-of } (xs_1, \dots, xs_n) \\ \text{base-of } ((sv_1 \cdot xs_1), \dots, (sv_n \cdot xs_n)) &\triangleq \text{T} \cdot \text{base-of } (xs_1, \dots, xs_n) \end{aligned}$$

The operators presented in the previous section are lifted to streams of present and absent values. We denote this lifting with $\diamond_{ty_1}^\uparrow$ for a unary operator, and $\oplus_{ty_1 \times ty_2}^\uparrow$ for a binary operator. These lifted operators are defined coinductively in [figures 2.4a](#) and [2.4c](#). When the inputs are absent, the lifted operators produce an absence. When the inputs are present, the lifted operators produce a present value by calling the underlying abstracted

$$\begin{array}{c}
 \hline
 G, H, bs \vdash c \Downarrow [\text{const } bs \ c] \\
 \hline
 \text{(a) const } \text{🐓} \text{ CoindStreams.v:478}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{const } (T \cdot bs) \ c \triangleq \langle c \rangle \cdot \text{const } bs \ c \\
 \text{const } (F \cdot bs) \ c \triangleq \langle \rangle \cdot \text{const } bs \ c \\
 \text{(b) Sconst } \text{🐓} \text{ Lustre/LSemantics.v:133}
 \end{array}$$

Figure 2.3: Semantics of sampled constants

$$\begin{array}{c}
 \diamond_{ty}^{\uparrow} (\langle \rangle \cdot vs_1) \triangleq \langle \rangle \cdot \diamond_{ty}^{\uparrow} vs_1 \\
 \diamond_{ty}^{\uparrow} (\langle v_1 \rangle \cdot vs_1) \triangleq \langle \diamond_{ty}^{\uparrow} (v_1) \rangle \cdot \diamond_{ty}^{\uparrow} vs_1 \\
 \text{(a) lift1 } \text{🐓} \text{ CoindStreams.v:484}
 \end{array}
 \qquad
 \begin{array}{c}
 G, H, bs \vdash e_1 \Downarrow [vs_1] \\
 \text{typeof } e_1 = [ty_1] \quad \diamond_{ty_1}^{\uparrow} vs_1 \equiv vs \\
 \hline
 G, H, bs \vdash \diamond e_1 \Downarrow [vs] \\
 \text{(b) Sunop } \text{🐓} \text{ Lustre/LSemantics.v:153}
 \end{array}$$

$$\begin{array}{c}
 \oplus_{ty_1 \times ty_2}^{\uparrow} (\langle \rangle \cdot vs_1) (\langle \rangle \cdot vs_2) \triangleq \langle \rangle \cdot \oplus_{ty_1 \times ty_2}^{\uparrow} vs_1 \ vs_2 \\
 \oplus_{ty_1 \times ty_2}^{\uparrow} (\langle v_1 \rangle \cdot vs_1) (\langle v_2 \rangle \cdot vs_2) \triangleq \langle \oplus_{ty_1 \times ty_2}^{\uparrow} (v_1, v_2) \rangle \cdot \oplus_{ty_1 \times ty_2}^{\uparrow} vs_1 \ vs_2 \\
 \text{(c) lift2 } \text{🐓} \text{ CoindStreams.v:496}
 \end{array}$$

$$\begin{array}{c}
 G, H, bs \vdash e_1 \Downarrow [vs_1] \quad G, H, bs \vdash e_2 \Downarrow [vs_2] \\
 \text{typeof } e_1 = [ty_1] \quad \text{typeof } e_2 = [ty_2] \quad \oplus_{ty_1 \times ty_2}^{\uparrow} vs_1 \ vs_2 \equiv vs \\
 \hline
 G, H, bs \vdash e_1 \oplus e_2 \Downarrow [vs] \\
 \text{(d) Sbinop } \text{🐓} \text{ Lustre/LSemantics.v:160}
 \end{array}$$

Figure 2.4: Semantics of operators

operator. These lifted operators are partial, for two reasons: first, the lifted operator may itself be partial, as discussed above; second, they require their arguments to be synchronized, that is, simultaneously present or absent.

The **when** and **merge** operators respectively sample a stream on an enumerated condition, and combine complementary sampled streams. The essence of their semantics is captured by the coinductive functions **when** and **merge** presented in [figures 2.5a](#) and [2.5c](#).

The **when** function is applied to three arguments: a constructor C , a value stream xs and a control stream cs . If both the input and control heads are absent, **when** produces an absence. If both are present, and the head of the control stream matches C , then **when** produces the head value of xs . Finally, if the head of the control stream does not match C , **when** produces an absence.

The **merge** function provides the opposite operation: it takes a control stream cs and a list of value streams xss . If the control head is absent, all the value streams heads should be as well, and an absence is produced. If the head of the control stream is a constructor C_i , then only the head of the i -th should be present, and its value is produced.

These coinductive functions are used in the semantic rules of [figures 2.5b](#) and [2.5d](#). In both rules, the condition x is associated to control stream cs . The semantic judgment, applied recursively, relates the list of sub-expressions with the value streams. In the case

$$\begin{array}{l}
 \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot cs) \triangleq \langle \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot cs) \triangleq \langle v \rangle \cdot \text{when}^C xs cs \\
 \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot cs) \triangleq \langle \rangle \cdot \text{when}^C xs cs
 \end{array}
 \quad
 \frac{
 \begin{array}{l}
 G, H, bs \vdash es \Downarrow [xs_i]^i \\
 H(x) \equiv cs \quad \forall i, \text{when}^C xs_i cs \equiv vs_i
 \end{array}
 }{
 G, H, bs \vdash es \text{ when } C(x) \Downarrow [vs_i]^i
 }$$

(a) when 🐦 [CoindStreams.v:522](#) (b) Swhen 🐦 [Lustre/LSemantics.v:190](#)

$$\begin{array}{l}
 \text{merge} (\langle \rangle \cdot cs) (\langle \rangle \cdot xs_1, \dots, \langle \rangle \cdot xs_m) \triangleq \langle \rangle \cdot \text{merge } cs (xs_1, \dots, xs_m) \\
 \text{merge} (\langle C_i \rangle \cdot cs) (\langle \rangle \cdot xs_1, \dots, \langle v \rangle \cdot xs_i, \dots, \langle \rangle \cdot xs'_m) \triangleq \langle v \rangle \cdot \text{merge } cs (xs_1, \dots, xs_i, \dots, xs'_m)
 \end{array}$$

(c) merge 🐦 [CoindStreams.v:538](#)

$$\frac{
 \begin{array}{l}
 H(x) \equiv cs \quad \forall i, G, H, bs \vdash es_i \Downarrow [xs_{ij}]^j \quad \forall j, \text{merge } cs [xs_{ij}]^i \equiv vs_j
 \end{array}
 }{
 G, H, bs \vdash \text{merge } x [C_i \Rightarrow es_i]^i \Downarrow [vs_j]^j
 }$$

(d) Smerge 🐦 [Lustre/LSemantics.v:197](#)

b	T	T	F	F	T	F	T	F	F	...
x	1	2	3	4	5	6	7	8	9	...
y = x when true(b)	1	2			5		7			...
z = (x * 2) when false(b)			6	8		12		16	18	...
merge b (true -> y) (false -> z)	1	2	6	8	5	12	7	16	18	...

(e) Example trace of **when** and **merge**

Figure 2.5: Semantics of **when** and **merge**

of **when**, the operator is applied pointwise to these value streams to produce streams for the full expression. In the case of **merge**, each of the m branches of the **merge** contains a list of sub-expressions that produces n streams. The **merge** operator is applied pointwise “perpendicularly” which produces the n streams for the full expression.

Note that both **when** and **merge**, as well as the lifting of a binary operation are only defined for very specific combinations of presence and absence. In Coq, these functions are actually represented as coinductive relations between inputs and output, since Coq does not allow for partial function definitions.

These constraints are an integral part of the synchronous model [CP96]. Indeed, consider the **buffered** node in [figure 2.6](#). The **h** stream switches between **true** and **false** at every cycle. The sampled stream **x when h** is therefore only present in half of the cycles. What stream should be associated to **y**? If we consider a Kahnian semantics [Kah74] and forget explicit absences, the n th value associated to **y** should be equal to the sum of the n th value associated to **x** and the n th value associated to **x when h**. This is the behaviour presented in the chronogram at right. This has two effects: first, values for **y** are only available when values for **x when h** are available. Second, **y** does not consume values of **x** as fast as they are produced. This means that, to compile this program, values of **x** would have to be buffered until they can be consumed. Moreover, as values of **x** are produced twice as fast as they are consumed, the size of this buffer would be unbounded. This is

```

node buffered(x : int)
returns (y : int)
var h : bool;
let
  h = true fby (not h);
  y = x + (x when h);
tel

```

x	x_1	x_2	x_3	x_4	x_5	...
h	T	F	T	F	T	...
x when h	x_1		x_3		x_5	...
y	$x_1 + x_1$		$x_2 + x_3$		$x_3 + x_5$...

Figure 2.6: A non-synchronous node and its execution

not acceptable, especially in the context of embedded systems programming, and this program should thus be rejected.

This synchrony is reflected in the semantic rules we presented until now, where absences and presences have to correspond. It is possible to statically ensure that “synchrony errors” never happen in a given program by using a static *clock-typing* analysis. The clock-type system of Vélus is a direct encoding of the one proposed in [CP96].

A *clock type* is either the base clock \bullet or a sampled clock $\text{ck on } \mathcal{C}(\mathbf{x})$. Each variable in a node must be declared with its clock by the programmer. Each expression is associated with a list of clocks (one for each stream produced by the expression). The judgment $G, \Gamma \vdash_{wc} e : cks$ means “under a global context G and a local environment Γ , e is well clocked with clock type cks ”.

Figure 2.7 presents some of the core rules defining this judgment. A constant always has the base clock type; this corresponds to the boolean clock of a constant being the base clock bs . The clock type of a variable is the one with which it is declared, accessible through the environment Γ . The clock types of the two operands of a binary operator must be the same, as outlined in the `buffered` example. If the sub-expressions of a `when` have clock type ck , then the expression sampled by $\mathcal{C}(e)$ must have clock $\text{ck on } \mathcal{C}(e)$. Again, this corresponds to the semantic rule for `when`. Finally, the rule for `merge` imposes that sub-expressions have complementary clock types $\text{ck on } \mathcal{C}_i(\mathbf{x})$, where \mathbf{x} is the condition of the `merge`. The produced stream has, as expected, the parent clock type ck . As we see in the last two rules, the conditions of `when` and `merge` must be variables because they appear in clock types. The rest of the clock-type system of the Vélus language is detailed in appendix A.3.

Although these clock-typing rules intuitively correspond to the semantic rules, it is not trivial to establish that, for any program, the produced streams actually correspond to the declared clock types. We discuss this issue in section 3.5.

2.4.3 Stateful Operators

Until now, we have presented combinatorial operators, where the output value at each cycle only depends on the input value at the same cycle. The language also provides stateful operators, where output values depend on the input values from earlier cycles.

$$\begin{array}{c}
 \frac{}{G, \Gamma \vdash_{\text{wc}} c : [\bullet]} \\
 \text{(a) wc_Econst } \color{red}{\text{Lustre/LClocking.v:56}}
 \end{array}
 \qquad
 \frac{\Gamma(x) = ck}{G, \Gamma \vdash_{\text{wc}} x : [ck]}
 \qquad
 \text{(b) wc_Evar } \color{red}{\text{Lustre/LClocking.v:62}}$$

$$\frac{G, \Gamma \vdash_{\text{wc}} e_1 : [ck] \quad G, \Gamma \vdash_{\text{wc}} e_2 : [ck]}{G, \Gamma \vdash_{\text{wc}} e_1 \oplus e_2 : [ck]}$$

$$\text{(c) wc_Ebinop } \color{red}{\text{Lustre/LClocking.v:76}}$$

$$\frac{\Gamma(x) = ck \quad G, \Gamma \vdash_{\text{wc}} es : [ck]^j}{G, \Gamma \vdash_{\text{wc}} es \text{ when } C(x) : [ck \text{ on } C(x)]^j}
 \qquad
 \frac{\Gamma(x) = ck \quad \forall i, G, \Gamma \vdash_{\text{wc}} es_i : [ck \text{ on } C_i(x)]^j}{G, \Gamma \vdash_{\text{wc}} \text{merge } x [C_i \Rightarrow es_i]^i : [ck]^j}$$

$$\text{(d) wc_Ewhen } \color{red}{\text{Lustre/LClocking.v:103}}
 \qquad
 \text{(e) wc_Emerge } \color{red}{\text{Lustre/LClocking.v:110}}$$

Figure 2.7: A few clock-typing rules

$$\begin{array}{c}
 \text{fby } (\langle x \rangle \cdot xs) (\langle y \rangle \cdot ys) \triangleq \langle x \rangle \cdot \text{fby } xs \ ys \\
 \text{fby } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \triangleq \langle v_1 \rangle \cdot \text{fby1 } v_2 \ xs \ ys
 \end{array}
 \qquad
 \frac{G, H, bs \vdash es_0 \Downarrow [xs_i]^i \quad G, H, bs \vdash es_1 \Downarrow [ys_i]^i \quad \forall i, \text{fby } xs_i \ ys_i \equiv vs_i}{G, H, bs \vdash es_0 \text{ fby } es_1 \Downarrow [vs_i]^i}$$

$$\begin{array}{c}
 \text{fby1 } v_0 (\langle x \rangle \cdot xs) (\langle y \rangle \cdot ys) \triangleq \langle x \rangle \cdot \text{fby1 } v_0 \ xs \ ys \\
 \text{fby1 } v_0 (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \triangleq \langle v_0 \rangle \cdot \text{fby1 } v_2 \ xs \ ys
 \end{array}$$

$$\text{(a) fby } \color{red}{\text{Lustre/LSemantics.v:44}}
 \qquad
 \text{(b) Sfby } \color{red}{\text{Lustre/LSemantics.v:176}}$$

b	T	T	F	F	T	T	F	T	F	T	T	...
x = 0 fby (x + (1 when b))	0	1			2	3		4		5	6	...

 (c) Example trace of **fby** on a sampled stream

 Figure 2.8: Semantics of **fby**

The first of these operators is **fby** (followed-by). The stream associated to $e_0 \text{ fby } e_1$ consists of the first value of e_0 , followed by the stream associated to e_1 . The stream of $e_0 \text{ fby } e_1$ should have the same clock as the streams of e_0 and e_1 . The semantic functions and rule specifying this behavior are presented in [figure 2.8](#). The **fby** function produces the first present value of the left stream, and then applies **fby1**, holds the first present value of the right stream, and only produces it when the right stream produces another present value. Both functions produce an absence if both input streams are absent. In that case, the value held by **fby1** does not change. As usual, the semantic rule for **fby** simply applies the **fby** function pointwise to the list of streams produced by the sub-expressions.

The second core stateful operator is the initialization arrow \rightarrow . The expression $e_0 \rightarrow e_1$ replaces the first present value of e_1 with that of e_0 . In a sense, this construction allows to define a specific behavior for the first cycle. The semantic rules for this construction

$$\begin{array}{l}
 \text{arrow } (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) \triangleq \langle \rangle \cdot \text{arrow } xs \ ys \\
 \text{arrow } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \triangleq \langle v_1 \rangle \cdot \text{arrow1 } xs \ ys \\
 \\
 \text{arrow1 } (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) \triangleq \langle \rangle \cdot \text{arrow1 } xs \ ys \\
 \text{arrow1 } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) \triangleq \langle v_2 \rangle \cdot \text{arrow1 } xs \ ys \\
 \text{(a) arrow } \text{🐓 Lustre/LSemantics.v:62} \\
 \\
 \frac{G, H, bs \vdash es_0 \Downarrow (xs_1, \dots, xs_n) \quad G, H, bs \vdash es_1 \Downarrow (ys_1, \dots, ys_n) \quad \forall i \in 1 \dots n, \text{arrow } xs_i \ ys_i \equiv vs_i}{G, H, bs \vdash es_0 \rightarrow es_1 \Downarrow (vs_1, \dots, vs_n)} \\
 \text{(b) Sarrow } \text{🐓 Lustre/LSemantics.v:183}
 \end{array}$$

 Figure 2.9: Semantics of the initialization arrow \rightarrow

$$\frac{G, H, bs \vdash es \Downarrow xss \quad G \vdash f(xss) \Downarrow yss}{G, H, bs \vdash f(es) \Downarrow yss}$$

Figure 2.10: Semantics of a node instantiation

are presented in [figure 2.9](#). As with `fb`, there are two coinductive functions, `arrow` and `arrow1`. The second does not need to hold a value, as the arrow does not induce a delay. Instead, passing from `arrow` to `arrow1` simply changes which of the two streams values are produced.

The initialization arrow is particularly useful in conjunction with the `pre` operator. Like `fb`, the expression `pre e` delays the stream of `e` to the next cycle, but does not give an initial value. The value of `pre e` at the first cycle is not defined, which is not an issue if this expression only appears at right of an initialization arrow. In particular, `e0 \rightarrow pre e1` is equivalent to `e0 fb e1`. Languages with `pre` require a dedicated initialization analysis [[CP04](#)] to check that uninitialized streams are not used improperly. We have not implemented such an analysis in Vélus, and `pre` is not included in our language. It may be tempting to define `pre` with a fixed, arbitrary initial value, and see `pre e` as equivalent to `0 fb e`, but this violates the spirit of the operator and introduces an unnecessary initialization in the generated code.

2.4.4 Node Instantiation

Every node defined in a Vélus program can be instantiated within another in order to define complex behaviors modularly. To ensure that all programs run in bounded time and stack space, it is not possible to recursively instantiate a node. This is enforced by a simple static check: a node may only instantiate nodes defined earlier in the program. The semantics of a node instantiations are straightforward and given in [figure 2.10](#). The list of arguments `es` produces a list of streams `xss` that is passed to the node. The node associates these inputs to list of output streams `yss`, according to the rule presented in [figure 2.2c](#).

$$\begin{array}{c}
 \frac{}{\Gamma \vdash_{\text{wc}} \bullet} \quad \frac{\Gamma \vdash_{\text{wc}} ck \quad \Gamma(x) = ck}{\Gamma \vdash_{\text{wc}} ck \text{ on } C(x)} \quad \frac{\forall x ck, \Gamma(x) = ck \implies \Gamma \vdash_{\text{wc}} ck}{\vdash_{\text{wc}} \Gamma} \\
 \text{(a) wc_clock } \color{red}{\text{🐔 Clocks.v:155}} \qquad \qquad \qquad \text{(b) wc_env } \color{red}{\text{🐔 Clocks.v:166}} \\
 \\
 \frac{\vdash_{\text{wc}} ins \quad \vdash_{\text{wc}} (ins + outs) \quad G, (ins + outs) \vdash_{\text{wc}} blk}{G \vdash_{\text{wc}} \text{node } f(ins) \text{ returns } (outs) blk} \\
 \text{(c) wc_node } \color{red}{\text{🐔 Lustre/LClocking.v:215}}
 \end{array}$$

Figure 2.11: Clock-Typing rules for clocks, environments and nodes

Clock dependencies in node arguments The Vélus language allows for subsampling in the inputs and outputs of nodes [EMSOFT21]. This means that an input or output stream may be sampled on another input or output stream. This is modeled in the clock-type system by dependencies between the clock types associated to input and output variables. Figure 2.11 presents the corresponding clock-typing rule. First, we define what it means for a clock to be well clocked. The base clock \bullet is always well clocked. A sampled clock $ck \text{ on } C(x)$ is well clocked with respect to an environment Γ if ck is well clocked and x is associated with the same ck in Γ . This sampling rule corresponds to the rule for **when** presented in figure 2.7. We say that an environment is well clocked if all the clocks appearing within it are themselves well clocked, under the environment itself. The last rule in figure 2.11 indicates that a node is well clocked if its input environment is, the union of its input and output environments is, and if its body is.

These definitions imply several interesting properties. First, subsampling dependencies are not constrained by the order of input and output arguments. Signatures $f(b : \text{bool}; x : \text{int} \text{ when } b)$ and $g(x : \text{int} \text{ when } b; b : \text{bool})$ are both valid. Second, outputs may depend on both inputs and other outputs, but the first premise implies that an input may depend only on other inputs and never on an output.

Another interesting property is that a non-empty, well-clocked environment always contains at least one base clock type. This is stated formally by lemma 1.

Lemma 1 (Existence of a base clock 🐔 Lustre/LClocking.v:354)

$$\text{if } \vdash_{\text{wc}} \Gamma \text{ and } \Gamma \neq \emptyset \text{ then } \exists x, \Gamma(x) = \bullet$$

One implication of this lemma is that there exists at least one input stream of the node that is present at least as often as all the others. The **base-of** function described in figure 2.2 returns the clock of this particular stream.

Subsampled node instantiation We now discuss how subsampling is treated by the clock-type system. In Vélus, a node f may be instantiated by another node g with sampled inputs: the fastest input of f may be slower than other streams in g . Statically, this means that the base clock type of f is not necessarily the same as the base clock

type of the instance. To support this behavior, the clock-type system of Vélus needs to allow for a limited form of *clock-type polymorphism*. This is complicated by the possibility of dependencies between the clock types of inputs and outputs of a node. Indeed, a dependency in the parameters of a node induces a dependency in the arguments of its instance.

Consider the example of [listing 2.17](#), which contains all kinds of dependencies. Both the second input x , and the first output y of node f are sampled by its first input b . The second output of f is itself sampled by y , and implicitly by b . We do not show the body of f , which is unimportant here. Node g instantiates f , passing as a first input $b2$, which is itself sampled by $b1$. The second input of g is the constant 0 , sampled by $b1$ and $b2$. Note that these `when` do not need to be added explicitly by programmers: they can be inferred and added by the elaboration pass described in [section 4.3](#). The outputs of the node, $y1$ and $z1$ are correctly declared as sampled by $b2$ and $y1$ respectively.

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

node g(b1: bool; b2: bool when b1) returns (y1: bool when b2; z1: int when y1)
let
  (y1, z1) = f(b2, 0 when b1 when b2);
tel
```

Listing 2.17: Subsampled instantiation of a node with clock dependencies

We now show how the clock-type system relates the clock types of the formal parameters of the node, the clock types of the argument expressions, and the clock types of the instantiation. This relation is formalized by a clock type instantiation function, `instck`, presented in [figure 2.12a](#). In addition to the clock to instantiate, it takes two parameters: the base clock of the node instantiation bck and a substitution of parameter variables to argument variables. We say that the node instantiation is well clocked if we can find bck and sub such that all inputs/outputs clock types of the node can be instantiated by `instck` to the clock types of argument expressions/of the instantiation. In the example, the following pairs of parameters/arguments must be unified by `instck`.

```
node f( b :: bck                                with b2 :: • on true(b1)
      x :: bck on true(b)                       with _  :: • on true(b1) on true(b2)
returns( y :: bck on true(b)                   with y1 :: • on true(b1) on true(b2))
      z :: bck on true(b) on true(y)          with z1 :: • on true(b1) on true(b2) on true(y1)
```

This is possible by taking $bck = \bullet \text{ on true}(b1)$ and $\sigma = \{b \mapsto b2; y \mapsto y1; z \mapsto z1\}$. Note that some of these instantiated clocks depend on variables $b2$ and $y1$. For this system to be sound, these variables must be associated with the parameters b and y respectively. In order to formalize this constraint, we use the notion of *named clock* introduced in [\[CP03\]](#). We write $(x : ck)$ for the named clock ck of variable x . These annotations are used to denote streams that are associated with named variables, on which some other clocks might depend. Inversely, we write $(_ : ck)$ for the clock of an *anonymous stream*; by

$$\begin{array}{c}
 \text{instck}_{bck}^\sigma \bullet \triangleq bck \\
 \text{instck}_{bck}^\sigma ck \text{ on } C(x) \triangleq (\text{instck}_{bck}^\sigma ck) \text{ on } C(\sigma(x)) \\
 \text{(a) instck } \color{red}{\text{🐦 Clocks.v:22}}
 \end{array}
 \qquad
 \frac{\Gamma(x) = ck}{G, \Gamma \vdash_{\text{wc}} x : [(x : ck)]}$$

$$\begin{array}{c}
 \frac{x_1 \notin \sigma \quad \text{instck}_{bck}^\sigma ck_1 = ck_2}{\text{WellInstantiated}_\sigma^{bck}(x_1 : ck_1) (_ : ck_2)} \qquad
 \frac{\sigma(x_1) = x_2 \quad \text{instck}_{bck}^\sigma ck_1 = ck_2}{\text{WellInstantiated}_\sigma^{bck}(x_1 : ck_1) (x_2 : ck_2)} \\
 \text{(b) WellInstantiated } \color{red}{\text{🐦 Lustre/Locking.v:43}}
 \end{array}$$

$$\frac{G, \Gamma \vdash_{\text{wc}} es : [nck_i]^i \quad G(f) = \text{node } f([x_i \text{ on } ick_i]^i) \text{ returns } ([y_j \text{ on } ock_j]^j) \text{ blk}}{\forall i, \text{WellInstantiated}_\sigma^{bck}(x_i : ick_i) nck_i \quad \forall j, \text{WellInstantiated}_\sigma^{bck}(y_j : ock_j) (_ : ck'_j)} \\
 G, \Gamma \vdash_{\text{wc}} f(es) : [ck'_j]^j \\
 \text{(c) wc_Eapp } \color{red}{\text{🐦 Lustre/Locking.v:130}}$$

$$\frac{G, \Gamma \vdash_{\text{wc}} es : [nck_i]^i \quad G(f) = \text{node } f([x_i \text{ on } ick_i]^i) \text{ returns } ([y_j \text{ on } ock_j]^j) \text{ blk}}{\forall i, \text{WellInstantiated}_\sigma^{bck}(x_i : ick_i) nck_i \quad \forall j, \text{WellInstantiated}_\sigma^{bck}(y_j : ock_j) (y'_j : ck'_j)} \\
 G, \Gamma \vdash_{\text{wc}} [y'_j]^j = f(es) \\
 \text{(d) wc_EqApp } \color{red}{\text{🐦 Lustre/Locking.v:140}}$$

Figure 2.12: Clock-typing rules for node instantiation

abuse of notation, we may just write ck . The clock of the expression containing only a variable is named, because we know that possible dependencies can refer to this variable. For instance, in the example, the clock of expression `b2` is $(b2 : \bullet \text{ on true}(b1))$. We formalize this by modifying the clock-typing rule for the variable, as shown in [figure 2.12](#), at top right. The clock of any other expression is anonymous.

The `WellInstantiated` predicate presented in [figure 2.12b](#) uses named clocks to constrain σ and bck according to the parameters and arguments of the node. If the argument clock is anonymous, then the parameter name should not appear in the substitution: no other clock may depend on it. If it is named, then the parameter name is associated with the argument name. The argument clock must correspond to the instantiation of the parameter clock.

We now direct our attention to the full clock-typing rule for node instantiations, displayed in [figure 2.12c](#). As expected, `WellInstantiated` is applied to the list of input parameters and argument clocks, and to the list of output parameters and output clocks. Note however that all the output clocks are anonymous. This is because, in the general case, there is no variable directly associated with the output of an instantiation. If the instantiation is directly at right of an equation, we can use the variables at left of the equation, as presented in [figure 2.12d](#). This corresponds to the case treated in our example.

However, if the instantiation appears in a nested expression, there is no natural candidate for naming clocks.

An early solution was to associate unique, local names to each output of the instantiation. These names could then be used to formalize the dependencies between output clocks. However, this approach introduces too many complications in the formalization, compiler, and proofs. In particular, it was cumbersome to keep track of the uniqueness of these local names. For simplicity, we decided to drop it. For the user of the language, this means that a node with clock dependencies in its output (like the one in the example) cannot be instantiated in a nested expression, but only directly at right of an equation. For instance, in the above example, we could not write $f(f(b2, 0 \text{ when } b1 \text{ when } b2))$. While, in theory, this restricts the class of accepted programs, we have not yet encountered a real world program that would be impacted by this change, and it would not be hard to adapt it if necessary.

2.5 Semantics of Switch

As described in the introduction, the purpose of a `switch` block is to control the activation of its sub-blocks. What does it formally mean for a block to be activated during a cycle? First, only activated blocks impose their constraints on the global history. Second, the stateful constructs such as `fbv` must only be updated when the enclosing block is activated. This second constraint corresponds exactly to the behavior of stateful constructs on sampled streams, demonstrated in [figure 2.8c](#). The values in the stream produced by the `fbv` semantic operator are independent of absences in the input streams. In other words, inserting or removing absences in input streams inserts or removes corresponding absences in output streams, but does not change their actual values.

2.5.1 Activation and Sampling

The insertion and removal of absences is at the core of the semantic model we propose for control blocks such as `switch`. We describe the `switch` as essentially a block-based composition of `when` and `merge`, with activation expressed by sampling. The `switch_ex` node presented in [figure 2.13](#) gives a concrete example of this idea. When `c` is equal to `A`, the first branch is active and the equation $y = (0 \text{ fbv } y) + 1$ constrains the value of `y`. Only one branch is active and applies its constraints at each cycle. This gives three complementary views of the global stream `y`. Each view corresponds to a sampling of `y` by the condition of the `switch`.

The semantic judgment for a `switch` block is presented in [figure 2.14](#). Each branch of the `switch` is defined in a sampled context $\text{when}^{C_i}(H, bs) cs$, where `cs` is the stream produced by the `switch` guard. Sampling the history `H` ensures that the streams of variables read within a sub-block are sampled coherently. Sampling the base clock `bs` ensures that the streams of constants within a sub-block are also sampled coherently. Intuitively, all streams produced at the leaves of the AST are sampled coherently, therefore all streams produced in the block are also sampled coherently.

```

type t = A | B | C

node switch_ex(c : t)
returns (y : int)
let
  switch c
  | A do y = (0 fby y) + 1
  | B do y = (0 fby y) - 1
  | C do y = 0
end
tel
    
```

c	A	A	A	B	B	C	B	A	A	...
y when A(c)	1	2	3					4	5	...
y when B(c)				-1	-2		-3			...
y when C(c)						0				...
y	1	2	3	-1	-2	0	-3	4	5	...

 Figure 2.13: Example trace of `switch`

$$\frac{\forall i, G, H, bs \vdash e \Downarrow [cs]}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}} \quad \text{(a) Sswitch } \color{red}{\text{Lustre/LSemantics.v:293}}$$

$$\begin{aligned}
 & \text{whenb}^C (b \cdot bs) (\langle C \rangle \cdot cs) \triangleq b \cdot \text{whenb}^C bs cs \\
 & \text{whenb}^C (b \cdot bs) (sv \cdot cs) \triangleq F \cdot \text{whenb}^C bs cs \quad \text{(b) fwhenb } \color{red}{\text{CoindStreams.v:2974}}
 \end{aligned}$$

$$\begin{aligned}
 & (\text{when}^C H cs)(x) = \text{when}^C (H(x)) cs \\
 & \text{when}^C (H, bs) cs = (\text{when}^C H cs, \text{whenb}^C bs cs) \quad \text{(c) when_hist } \color{red}{\text{CoindStreams.v:2937}}
 \end{aligned}$$

 Figure 2.14: Semantics of `switch`

We now formally define the operation $\text{when}^C (H, bs) cs$ which samples histories and boolean clocks. For clocks, we define the `whenb` function, as shown in [figure 2.14b](#). When the control stream cs is present and the constructor is the expected one, the boolean value of the clock is kept. In all other cases, the output value is `false`. Indeed, absence in a stream corresponds to `false` in its boolean clock. The clock-type systems guarantees that the base clock being filtered may be faster, but not slower, than the clock of the control stream. This means that, in the first case of the definition, b will always be `true`, and in the second case, if b is `false`, sv may only be absent. We choose not to impose these restrictions in the definition of `whenb`, and instead provide this simpler, total function. Lifting `when` to histories is trickier. As we have seen, `when` applied to a stream of values is a partial function. It only applies to streams that have the same clock as the control stream. In a history H , some streams may not respect this property. We define $\text{when}^C H cs$ as a total function producing a filtered and sampled history. We have $x \in \text{dom}(\text{when}^C H cs)$ if and only if $x \in \text{dom}(H)$, and if the stream xs associated with x in H has the same clock as cs . If that is the case, $(\text{when}^C H cs)(x)$ is defined and equal to $\text{when}^C xs cs$. In our Coq mechanization, we define this function as an inclusion relation between source and sampled histories.

$$\frac{G, \Gamma \vdash_{\text{wc}} e : [ck] \quad \forall x \, ck', \Gamma'(x) = ck' \implies \Gamma(x) = ck \wedge ck' = \bullet \quad \forall i, G, \Gamma' \vdash_{\text{wc}} \text{blks}_i}{G, \Gamma \vdash_{\text{wc}} \text{switch } e [C_i \text{ do } \text{blks}_i]^i \text{ end}}$$

Figure 2.15: Clock-typing rule for `switch` 🦉 [Lustre/LClocking.v:185](#)

2.5.2 Clock Typing of Switch Blocks

As we described above, only streams on the same clock as the condition stream may be sampled by `when`. This means that only these streams may be used or defined in the sub-blocks of a `switch`. This semantic constraint is captured by a static condition: only variables with the same clock type as the condition may appear in sub-blocks of the `switch`. This is expressed in the clock-typing rule presented in [figure 2.15](#). It states that, in the clock-typing environment Γ' used for clock typing sub-blocks, x may only be associated to a clock ck' if (i) x was associated with the clock ck of the condition in the initial environment Γ , and (ii) ck' is the base clock type. Note that this restriction only applies to global variables coming from outside of the `switch`: inside each branch, a local declaration may have an arbitrary clock type.

This rule is somewhat different from the one proposed in [\[CPP05\]](#), where the authors define an operation $\Gamma \text{ on}_{ck} C(c)$ which filters and samples environment Γ to produce an environment where all remaining clocks are $ck \text{ on } C(c)$, instead of just \bullet . Our rule is slightly simpler because our clock-type system only admits a single base clock for each node, while the clock-type system described in [\[CPP05\]](#) allows the user to write nodes with multiple unrelated clock variables. Our simplification has two main consequences for the rest of the project. On the one hand, it simplifies the statement and proof of the clock correctness theorem discussed in [section 3.5](#). On the other hand, it means that the compilation of `switch` blocks presented in [section 4.8](#) must update clock-type annotations when going from a “slow” context to a “fast” context, which complicates some proof invariants. Since this complication only appears in one pass of the compiler, we think that this choice is the right one.

2.6 Semantics of Reset

In the introduction, we saw how the `reset` construct is used to reinitialize stateful constructs. The modular `reset` operator was first introduced in [\[HP00\]](#), where the authors describe its intuitive behavior as (a variant of) the program below.

```

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

node reset_f(r : bool; x : int) returns (y : int)
let
  switch true_until(r)
  | true do y = f(x)
  | false do y = reset_f(r, x)
end
tel

```

Listing 2.18: Recursive behavior of `reset`

The node `true_until(r)` outputs `true` until its input becomes `true`. Afterwards, it always outputs `false`. The second node, `reset_f` takes two inputs: a reset signal `r` and an input `x` to be passed to a hypothetical node `f`. The node is built from a `switch` with `true_until(r)` as a condition. Therefore, the first branch, which instantiates `f` directly, is activated until a first `true` is received on `r`. Afterwards, the second branch is activated, and instantiates `reset_f` recursively. This recursive instantiation has the same behavior, but since `r` is sampled by the `switch`, it ignores the first `true`, and activates the first branch a second `true` is received on `r`. The same behavior is repeated every time a `true` is received. It is exactly the expected behavior of modular resetting of `f`, which we would write `(reset f every r)(x)` in Scade 6 [CPP17] and Vélus.

Of course, writing this program is not possible in Vélus, as recursive node instantiations are forbidden. In this particular case, the program would use unbounded space to keep track of an infinite number of states for `f`. In theory, it would be possible to apply a tail-call optimization to compile this program to imperative code using bounded space, since once the recursive branch is entered, it will never be left. However, only a very clever compiler could do so: it would require detecting and proving an invariant of the `true_until` node. Instead, Vélus implements a dedicated compilation scheme for modular `resets` and `reset` blocks, which we detail in chapters 4 and 5. In this section, we show how we mechanize the semantics of this construct.

2.6.1 Reset as Sampling

As discussed above, a `reset` operation can be described using a `switch` and recursive instantiations. In the previous section, we described how the semantics of `switch` can be mechanized using sampling. Following this path, we can represent the semantics of a `reset` using sampling. This idea was already present in [POPL20], where the authors introduce the $\text{mask}_k^k rs\ xs$ operator presented in figure 2.16a, with an example in figure 2.16b. The boolean stream `rs` corresponds to the stream of the `reset` condition. An *instance* of the value stream `xs` consists of the section of the stream between two `true`s on the control stream. We number these instances: instance n is located between the n th (included) and $(n + 1)$ th (excluded) occurrences of `true` in the control stream, or before the first occurrence

$$\begin{aligned} \text{mask}_{k'}^k (\mathbf{F} \cdot rs) (sv \cdot xs) &\triangleq (\text{if } k' = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'}^k rs xs \\ \text{mask}_{k'}^k (\mathbf{T} \cdot rs) (sv \cdot xs) &\triangleq (\text{if } k' + 1 = k \text{ then } sv \text{ else } \langle \rangle) \cdot \text{mask}_{k'+1}^k rs xs \end{aligned}$$

(a) mask 🐔 CoindStreams.v:2414

xs	0	1	2	3	4	5	6	7	8	9	...
rs	F	F	T	F	F	F	T	F	T	F	...
$\text{mask}^0 rs xs$	0	1	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$...
$\text{mask}^1 rs xs$	$\langle \rangle$	$\langle \rangle$	2	3	4	5	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$...
$\text{mask}^2 rs xs$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	6	7	$\langle \rangle$	$\langle \rangle$...
$\text{mask}^3 rs xs$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	$\langle \rangle$	8	9	...

(b) Example execution of mask

$$\begin{aligned} \text{bools-of } (\langle \mathbf{T} \rangle \cdot xs) &\triangleq \mathbf{T} \cdot \text{bools-of } xs \\ \text{bools-of } (\langle \mathbf{F} \rangle \cdot xs) &\triangleq \mathbf{F} \cdot \text{bools-of } xs \\ \text{bools-of } (\langle \rangle \cdot xs) &\triangleq \mathbf{F} \cdot \text{bools-of } xs \end{aligned}$$

(c) bools_of 🐔 CoindStreams.v:1214

$$(\text{mask}^k rs H)(x) = \text{mask}^k rs (H(x))$$

$$\text{mask}^k rs (H, bs) = (\text{mask}^k rs H, \text{mask}^k rs bs)$$

(d) mask_hist 🐔 CoindStreams.v:2421

$$\frac{G, H, bs \vdash es \Downarrow xss \quad G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv rs \quad \forall k, G \vdash f(\text{mask}^k rs xss) \Downarrow (\text{mask}^k rs yss)}{G, H, bs \vdash (\text{reset } f \text{ every } e)(es) \Downarrow yss}$$

(e) Sapp 🐔 Lustre/LSemantics.v:246

$$\frac{G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv rs \quad \forall k, G, \text{mask}^k rs (H, bs) \vdash blks}{G, H, bs \vdash \text{reset } blks \text{ every } e}$$

(f) Sreset 🐔 Lustre/LSemantics.v:286

Figure 2.16: Semantics of `reset`

of `true` if $n = 0$. The index k indicates which instance must be produced. In the example, $\text{mask}^0 rs xs$ is the first instance of xs , $\text{mask}^1 rs xs$ the second, etc. In general, $\text{mask}_0^k rs xs$ produces the k th instance of xs reset by rs . Its definition uses an accumulator k' which indicates how many instances have been crossed so far. This accumulator starts at 0; in the following, we simply write $\text{mask}^k rs xs$ when the accumulator is 0. When `true` occurs on the condition stream, the accumulator is incremented. Values of xs are produced only when the correct instance is reached, that is when the accumulator is equal to k . Note that the `mask` function is total.

In [POPL20], the authors apply this masking operation to implement the modular reset on node instantiations. The corresponding rule is given in figure 2.16e. The `reset` condition produces the value stream ys that is turned into a boolean stream by `bools-of`, which extracts boolean values from a stream of synchronous values, and replaces absence with `false`. Both the inputs and outputs of the instantiation are sampled using `mask` into an infinite number of instances. This models an infinite number of instantiations to the node that do not communicate with one another, and each starts afresh.

We adapt this idea to resetting whole blocks of equations. Instead of a sampling inputs and outputs, we use `mask` to sample the history and base clock, just as we did

<pre> node expect(i : bool) returns (o : bool) let o = i or (false fby o); tel node abro(a, b, r : bool) returns (o : bool) let reset o = expect(a) and expect(b); every r; tel </pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">a</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">...</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">b</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">...</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 2px 5px;">r</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">...</td> </tr> <tr style="border-top: 1px solid black;"> <td style="border-right: 1px solid black; padding: 2px 5px;">o</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">F</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">T</td> <td style="padding: 2px 5px;">...</td> </tr> </table>	a	T	F	F	F	T	F	F	T	F	...	b	F	T	F	F	F	F	T	F	F	...	r	F	F	F	T	F	T	F	F	F	...	o	F	T	T	F	F	F	F	T	T	...
a	T	F	F	F	T	F	F	T	F	...																																			
b	F	T	F	F	F	F	T	F	F	...																																			
r	F	F	F	T	F	T	F	F	F	...																																			
o	F	T	T	F	F	F	F	T	T	...																																			

Figure 2.17: Example trace of the abro node

$$\frac{G, \Gamma \vdash_{wc} e : [ck] \quad G, \Gamma \vdash_{wc} blks}{G, \Gamma \vdash_{wc} \text{reset } blks \text{ every } e}$$

Figure 2.18: Clock-Typing rule for `reset` 🐔 [Lustre/LClocking.v:179](#)

for `switch`. As `mask` is a total function, this lifting is easier. Lifting `mask` to histories consists in applying it pointwise to every stream in the history. The `mask` function is also applied to clock streams by interpreting absences in the definition of [figure 2.16a](#) as `false`. The semantics of sub-blocks is then given under an infinite number of instances of the history and base clock.

The behavior of `reset` on blocks is illustrated by the ABRO program, initially defined in [\[Ber00, §3.1\]](#) and adapted to Lucid Synchrone in [\[Pou06, §1.3.6\]](#). Its implementation in the Vélus syntax is presented in [figure 2.17](#). The output `o` becomes `true` after a `true` has occurred on `a` and `b`, and before a `true` occurs on `r`. In this example, when `r` is true, the internal states of both `expect` nodes are reset, which corresponds to the expected behavior. An example trace for this node is presented at right.

2.6.2 Clock Typing of Reset

As we have seen above, the `mask` function is total. While `bools-of` is not total, it is defined as long as its input only contains `true` and `false` values. It simply converts absence to `false`: semantically, if the reset stream is absent, no reset should occur. This means that, together, these functions do not impose any synchrony constraint on the condition and value streams. The condition stream may be faster or slower than the value stream. The clock-typing rule for `reset` blocks reflects this leniency: the clock type of the condition is unrelated to the clock types used in the sub-blocks.

Note that this clock-typing rule is only acceptable for a synchronous semantics where

$$\frac{\forall x, x \in \text{dom}(H') \iff x \in \text{locs} \quad G, H + H', bs \vdash \text{blks}}{G, H, bs \vdash \text{var } \text{locs} \text{ let } \text{blks} \text{ tel}} \quad (a) \text{ sem_scope } \text{🐦} \text{ Lustre/LSemantics.v:108}$$

$$(H_1+H_2)(x) = \begin{cases} \text{if } x \in H_2 \text{ then } H_2(x) \\ \text{else } H_1(x) \end{cases} \quad (b) \text{ union } \text{🐦} \text{ FunctionalEnvironment.v:413}$$

Figure 2.19: Semantics of local declarations

absence is explicit. In a Kahnian semantic, where streams only contain present values, we could not convert explicit absences to `false`. If the two streams are not synchronized, then the slower one would have to be buffered, potentially with an unbounded buffer. In our semantic model, having an explicit absence meaning “do not reset” allows us to sidestep this issue, and obtain a more flexible language, which is particularly useful for intermediate compilation steps, as discussed in [chapter 4](#).

2.7 Semantics of Local Declarations

The Vélus source language allows for blocks of local declarations to be arbitrarily nested with other control blocks. This has two advantages. First, it is useful for implementing compilation algorithms. Most of these algorithms, as we will see in [chapter 4](#), need to introduce fresh variables in the node. We can use local declarations to introduce these variables locally, where they are needed, rather than globally. Second, local declarations are also useful for the programmer. For instance, one could declare a local variable only in one of the cases of a switch. The semantics of a block of local declarations is specified by the rule presented in [figure 2.19a](#). If the semantics of the declaration is given under a history H , then the semantics of the sub-blocks is given under an extended history $H + H'$. The extension operator $+$ is defined in [figure 2.19b](#). It gives priority to its right operand, but this is not actually important in the semantic rule. We now discuss why.

Shadowing Consider the node in [listing 2.19](#). Its output, y , is sampled by its input, x . However, the name of its input is reused to declare a local variable. The occurrences of x under the local declaration should therefore refer to this local x . This is a problem because y is defined under this declaration as `0 when x`. This should not be well clock-typed, as this x is not the one used in the declaration of y . This node should be rejected.

```
node shadows(x : bool) returns (y : int when x)
var x : bool;
let
  x = false;
  y = 0 when x;
tel
```

Listing 2.19: Lustre node with redeclaration of a variable

$$\begin{array}{c}
\frac{H(\mathbf{last}\ x) \equiv vs}{G, H, bs \vdash \mathbf{last}\ x \Downarrow [vs]} \\
\text{(a) Slast } \text{🐔} \text{ Lustre/LSemantics.v:148}
\end{array}
\qquad
\begin{array}{c}
\frac{G, H, bs \vdash e \Downarrow [vs_0] \quad H(x) \equiv vs_1 \quad H(\mathbf{last}\ x) \equiv \mathbf{fby}\ vs_0\ vs_1}{G, H, bs \vdash \mathbf{last}\ x = e} \\
\text{(b) Slastd } \text{🐔} \text{ Lustre/LSemantics.v:278}
\end{array}$$

Figure 2.20: Semantics of shared variables

To be able to reject it, the clock system needs to associate a unique clock variable name to each declaration, instead of using the variable name directly. Here, suppose that clock variable x_1 is associated to the input \mathbf{x} , and x_2 is associated to the local \mathbf{x} . In this case, \mathbf{y} would be declared with type $\bullet \text{ on true}(x_1)$ while the expression $0 \text{ when } \mathbf{x}$ would have type $\bullet \text{ on true}(x_2)$. These two clock types being incompatible, the equation $\mathbf{y} = 0 \text{ when } \mathbf{x}$ would be rightly rejected. This is the approach taken in Lucid Synchrone [CP03]. In Vélus however we prefer a simpler clock-type system, in particular avoiding indirections between variable names and clock variables. To reject invalid programs like the one discussed above, Vélus simply rejects programs where local declarations shadow variables already in scope. In particular, this precludes the case where $x \in H$ and $x \in H'$.

2.8 Semantics of Shared Variables

As the introductory example showed, the last value of a shared variable can be accessed with the `last` operator. The semantic rules presented in figure 2.20 specify the behavior of this operator. The stream associated to a `last` variable is read from the history, which we denote $H(\mathbf{last}\ x)$. To support this, we generalize the Coq definition of histories presented in line 2 to allow a key to be either a variable \mathbf{x} or the last value of a variable `last` \mathbf{x} . This idea works just as well in Coq as it does for typing environments [LCTES11, §3.2] and compilers [Pou10].

Each variable used with `last` must be declared with an initialization equation of the form `last` $\mathbf{x} = \mathbf{e}$. It is this equation that constrains the stream associated to `last` \mathbf{x} in the history. This stream corresponds to the stream associated with \mathbf{x} , delayed to the next present cycle, and initialized by the stream of expression \mathbf{e} . This behavior is mechanized by applying the `fby` operator presented earlier to these two streams, giving a formal definition that directly reflects the intuitive meaning of this construct.

2.9 Semantics of State Machines

We now present our mechanization of the semantics of state machines in Vélus. The semantics of state machines were originally defined by translation [CPP05] and later by a transition relation between instantaneous environments [CHP06, §4.1]. We draw on this work to propose new rules based on histories, in the same style as the rules introduced earlier for `switch` and `reset`.

The semantic rules for state machines are presented in [figure 2.22](#). In the transition-based semantics [[CHP06](#), §4], a state machine is annotated with an entry state and a boolean indicating if a reset is required. In our model, the state stream sts represents this information for every cycle. At each cycle, the state stream may be absent, which we denote $\langle \rangle$ as with synchronous values. Intuitively, this means that the whole state machine is inactive. If the state stream is present, it contains a pair $\langle C, b \rangle$ where C is the tag of the active state for this cycle, and b indicates if it must be reset.

The state stream is used as a control stream for the `select` operator that we now introduce. Just as `when` was used for `switch` and `mask` for `reset`, the `select` function samples the history and base clock for each state of the state machine. Its behavior is actually equivalent to composing `when` and `mask`, as stated in [lemma 2](#) below, where π_1 and π_2 denote the projections from state stream to tag and boolean streams respectively. We believe it is clearer to express it directly as a coinductive operator, presented in [figure 2.22a](#). Values are only kept if (i) the tag on the state stream corresponds to the expected tag, and (ii) the instance for this particular tag (counted by k') is the expected one (k). Using this definition, [lemma 2](#) is easily proven by coinduction.

Lemma 2 (Correspondence of `select`, `when` and `mask` 🐦 [CoindStreams.v:3286](#))

$$\text{select}_{k'}^{C,k} sts xs \equiv \text{mask}_{k'}^k (\text{when}^C \pi_2(sts) \pi_1(sts)) (\text{when}^C xs \pi_1(sts))$$

To illustrate how `select` is used to give a semantics to state machines, [figure 2.21](#) presents an example automaton with weak transitions, entry by history and by reset, and a possible trace of its execution. The chronogram displays the state-and-reset stream of the state machine sts , and the stream associated with output `o` in history H , as well as its first four samplings by `select`. During the first two cycles, the state machine is in state `Up`, and the value of `o` increases. Since there have not been any reset yet, the parameters of `select` are `Up` and 0. At the end of the second cycle, the second transition condition in state `Up` is fulfilled: `Down` state is entered with a first reset. In the next cycles, `select` is applied with parameters `Down` and 1. When `Up` is entered without reset, `select` is applied with arguments `Up` and 0 once again. The execution continues in the same way: for each state and reset, the history is sampled depending on the values of sts , which are given by the transitions in the active branch. We now present the corresponding formal semantic rules.

State machines with weak transitions The two rules for weak state machines are presented in [figure 2.22b](#). The first rule describes the behavior of a single state of a state machine. It mimics the rule for local declarations presented in [figure 2.19a](#), as each state may declare local variables that may be used in weak transitions. The semantic rules indicate that the transitions produce a state stream sts ; we detail the corresponding rules later. This state stream is exposed by this first rule, as it needs to be used by the second rule, which describes the behavior of a weak state machine as a whole.

In this second rule, `select` is lifted to histories and to the base clock, and used to give the semantics of each state in a sampled context. Selecting is parameterized by

```

node updown(min, max : int)
returns (o : int)
let
  automaton initially Up
  state Up do
    o = 0 fby (o + 1);
    until o = max * 2 then Up
      | o = max then Down
  state Down do
    o = 0 fby (o - 1);
    until o = min continue Up
end

```

H(min)	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
H(max)	1	1	1	1	1	1	1	1	1	1	...
$\pi_1(sts)$	U	U	D	D	U	U	U	U	D	D	...
$\pi_2(sts)$	F	F	T	F	F	F	T	F	T	F	...
(select ^{U,0} sts H)(o)	0	1	<>	<>	2	3	<>	<>	<>	<>	...
(select ^{D,1} sts H)(o)	<>	<>	0	-1	<>	<>	<>	<>	<>	<>	...
(select ^{U,1} sts H)(o)	<>	<>	<>	<>	<>	<>	0	1	<>	<>	...
(select ^{D,2} sts H)(o)	<>	<>	<>	<>	<>	<>	<>	<>	0	-1	...
H(o)	0	1	0	-1	2	3	0	1	0	-1	...

Figure 2.21: Example trace of an `automaton` with weak transitions

state stream sts , which represent the state at the current instant. It is built from two streams: the initial-state stream sts_0 , computed from the initialization conditions, and the next-state stream sts_1 , computed from the transitions in the automaton states. Note that the presence or absence of state streams is determined by the stream bs' , calculated by interpreting the clock annotation of the state machine ck . The inclusion of such an annotation in the semantics is unfortunate, but we did not find a better alternative as, unlike for the `switch` guard, there is no natural candidate to give the required rhythm.

State machines with strong transitions The rule for strong state machines differs from those for weak state machines in three ways. First, since only a single initial state is declared, the initial-state stream is simply defined with `const`. Second, state-local variables cannot be used in strong transition guards so there is no need for a special treatment of local scopes. Third, and most importantly, sts now specifies the entry-state stream. It is used to control the activation of the strong transitions of each states, which determine the current-state stream sts_1 . This current-state stream is used in turn to determine which state applies its constraints to the global history. The value of the entry-state stream corresponds to the current-state stream, delayed and initialized by the initial-state stream. This models the expected behavior of strong transitions: they are checked first to determine the active state which subsequently becomes the entry state.

$$\begin{aligned}
& \text{select}_{k'}^{C,k} (\langle \rangle \cdot sts) (\langle \rangle \cdot xs) \triangleq \langle \rangle \cdot \text{select}_{k'}^{C,k} sts xs \\
& \text{select}_{k'}^{C,k} (\langle C, F \rangle \cdot sts) (\langle v \rangle \cdot xs) \triangleq (\text{if } k' = k \text{ then } \langle v \rangle \text{ else } \langle \rangle) \cdot \text{select}_{k'}^{C,k} sts xs \\
& \text{select}_{k'}^{C,k} (\langle C, T \rangle \cdot sts) (\langle v \rangle \cdot xs) \triangleq (\text{if } k' + 1 = k \text{ then } \langle v \rangle \text{ else } \langle \rangle) \cdot \text{select}_{k'+1}^{C,k} sts xs \\
& \text{select}_{k'}^{C,k} (\langle C', b \rangle \cdot sts) (\langle v \rangle \cdot xs) \triangleq \langle \rangle \cdot \text{select}_{k'}^{C,k} sts xs
\end{aligned}$$

(a) select 🐔 [CoindStreams.v:3253](#)

$$\frac{\forall x, x \in \text{dom}(H') \iff x \in \text{locs} \quad G, H + H', bs \vdash \text{blks} \quad G, H + H', bs, C_i \vdash \text{trans} \Downarrow sts}{G, H, bs, C_i \vdash \text{var locs do blks until trans} \Downarrow sts}$$

$$\frac{H, bs \vdash ck \Downarrow bs' \quad G, H, bs' \vdash \text{autinits} \Downarrow sts_0 \quad \text{fbv } sts_0 sts_1 \equiv sts \quad \forall i, \forall k, G, (\text{select}_0^{C_i,k} sts (H, bs)), C_i \vdash \text{autscope}_i \Downarrow (\text{select}_0^{C_i,k} sts sts_1)}{G, H, bs \vdash \text{automaton initially autinits}^{ck} [\text{state } C_i \text{ autscope}_i]^i \text{ end}}$$

(b) SautoWeak 🐔 [Lustre/LSemantics.v:306](#)

$$\frac{H, bs \vdash ck \Downarrow bs' \quad \text{fbv} (\text{const } bs' (C, F)) sts_1 \equiv sts \quad \forall i, \forall k, G, (\text{select}_0^{C_i,k} sts (H, bs)), C_i \vdash \text{trans}_i \Downarrow (\text{select}_0^{C_i,k} sts sts_1) \quad \forall i, \forall k, G, (\text{select}_0^{C_i,k} sts_1 (H, bs)) \vdash \text{blks}_i}{G, H, bs \vdash \text{automaton initially } C^{ck} [\text{state } C_i \text{ do blks}_i \text{ unless trans}_i]^i \text{ end}}$$

(c) SautoStrong 🐔 [Lustre/LSemantics.v:328](#)

$ \frac{G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs' \quad G, H, bs \vdash \text{autinits} \Downarrow sts \quad sts' \equiv \text{first-of}_F^C bs' sts}{G, H, bs \vdash C \text{ if } e; \text{ autinits} \Downarrow sts'} $	$ \frac{G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs' \quad G, H, bs, C_i \vdash \text{trans} \Downarrow sts \quad sts' \equiv \text{first-of}_F^C bs' sts}{G, H, bs, C_i \vdash \text{if } e \text{ continue } C \text{ trans} \Downarrow sts'} $
$ \frac{sts \equiv \text{const } bs (C, F)}{G, H, bs \vdash \text{otherwise } C \Downarrow sts} $	$ \frac{G, H, bs \vdash e \Downarrow [ys] \quad \text{bools-of } ys \equiv bs' \quad G, H, bs, C_i \vdash \text{trans} \Downarrow sts \quad sts' \equiv \text{first-of}_T^C bs' sts}{G, H, bs, C_i \vdash \text{if } e \text{ then } C \text{ trans} \Downarrow sts'} $

(d) Initial state

$$\begin{aligned}
& \text{first-of}_r^C (T \cdot bs) (st \cdot sts) \triangleq \langle C, r \rangle \cdot \text{first-of}_r^C bs sts \\
& \text{first-of}_r^C (F \cdot bs) (st \cdot sts) \triangleq st \cdot \text{first-of}_r^C bs sts
\end{aligned}$$

(e) first_of 🐔 [CoindStreams.v:3600](#)

$$\frac{sts \equiv \text{const } bs (C_i, F)}{G, H, bs, C_i \vdash \epsilon \Downarrow sts}$$

(f) sem_transitions 🐔 [Lustre/LSemantics.v:261](#)

Figure 2.22: Semantics of state machines

Semantics of transitions State streams are generated from the transitions of the active state of each cycle, according to the rules of [figure 2.22f](#). If no transition is activated, that is if all transition conditions evaluate to `false`, then the current tag is emitted: the state machine stays in the same state, which is not reset. Each transition is formed by a boolean condition, a tag indicating which state should be entered, and either keyword `continue` or `then`, indicating whether or not the state should be reset on entry. When evaluating the condition of a transition, the corresponding boolean stream bs' is passed to the coinductive operator `first-of`. When the condition is `true`, the state value is that of the transition. Otherwise, it is that computed from the rest of the list of transitions. This `first-of` function characterizes the order of priority of transitions. The semantic rules used to calculate the initial state of a weak state machines have the same form.

Mixing weak and strong transitions The original proposition for state machines [[CPP05](#); [CHP06](#)] allows mixing weak and strong transitions in a single state machine. Weak transitions determine the next entry state and strong transitions determine the active state. It is possible to weakly enter a state only to exit it strongly the very next instant. Besides the difficulty of understanding and explaining this behavior, it is not clear whether the ephemeral state should be reset, and doing so complicates both the semantic model and the compilation scheme. Scade 6 avoids this problem by using a dynamic check that only allows at most one transition to fire per cycle [[CPP17](#), §V.C.1]. Lucid Synchronic statically rejects state machines where a weak transition enters a state with strong transitions.


Rather than further complicate the semantic rules and compilation scheme, we simply forbid mixing weak and strong transitions in the same state machine. This approach was inspired by Zélus [[Pou10](#)]. The ability to conditionally specify the initial states of weak state machines is intended to compensate for the lost expressivity.

2.10 Partial Definitions

The `drive_sequence` node presented in [section 1.2](#) contains a `switch` block, where the definitions of `mA` and `mB` are missing in the second branch. The definitions of `mA` and `mB` are *partial* [[Pou06](#), §1.4.4]. Partial definitions are allowed for variables that are defined with a `last` value. The informal semantics of a partial definition is that at each cycle, if the active branch contains a definition for the variable, then it takes the value of its definition, as expected. If the active branch does not contain a definition for the variable, then the variable keeps its previous value. In other words, it is as if the equation $x = \text{last } x$ was implicitly added in each branch where x is not defined.

Formalizing this behavior in our semantic model is not obvious. Indeed, in our model, each piece of syntax implies a corresponding constraint on the history of a node. A partial definition means that the history is under-constrained. The missing constraint has to be added elsewhere. This is accomplished by adding a premise to the `switch` semantic rule seen in [figure 2.23](#). To determine that a definition is partial, we use the function `Def`, which gives the set of variable defined by a block or list of blocks. If a variable x is defined

$$\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i}(H, bs) cs \vdash blks_i \quad \forall i, x \in \bigcup_j (\text{Def}(blks_j)) \setminus \text{Def}(blks_i), (\text{when}^{C_i} H cs)(x) \equiv (\text{when}^{C_i} H cs)(\text{last } x)}{G, H, bs \vdash \text{switch } e [C_i \text{ do } blks_i]^i \text{ end}}$$

Figure 2.23: Implicit Completion for a `switch`  [Lustre/LSemantics.v:123](#)

by at least one of the branches of a `switch`, it must be completed in any branch where it is not defined. This means constraining the streams of `x` and `last x` in the sampled environment of this particular branch to be equal. This is exactly the same constraint that would apply the equation `x = last x` was explicitly added to this branch.

A premise of the same form is also added to the semantic rules for state machines presented in [figures 2.22b](#) and [2.22c](#) to allow for partial definitions in state machines.

2.11 Discussion and Related Work

The choices we have made in defining the semantics for Vélus were not made in isolation. Verified compilers for other programming languages, and other specifications of synchronous dataflow languages have made significantly different decisions. We detail below the most relevant to our work, and discuss how their approaches may apply to Vélus, and the effect they might have.

2.11.1 Mechanized Semantics for Verified Compilers

2.11.1.1 CompCert

The semantics of the source language of CompCert, Clight, are specified as a relational big-step model [\[BL09\]](#). Each judgment relates a syntactic element, an environment, and a memory that may be updated by the evaluation of the syntactic element. For instance, $G, E \vdash s, M \xrightarrow{t} out, M'$ indicates that, under global environment G and local environment E , statement s transforms memory M into M' . out indicates the outcome of the evaluation: the statement may execute normally, or modify the control-flow (`break`, `continue`, `return`). This execution also produces a finite trace of observable events (side-effects) t . In Coq, the semantic rules for these judgments are defined as an **Inductive**, like in Vélus.

Since the language also contains loops and recursive calls that may not terminate, the semantics also needs to encode divergence. To do so, they define additional coinductive non-termination judgments [\[LG09\]](#). For instance, $G, E \vdash s, M \xrightarrow{T} \infty$ indicates that statement s diverges, and produces a possibly infinite trace of events T . In Coq, these definitions are given using **CoInductive**, which we use to define stream operators.

This semantic model is very relevant to imperative languages; we will see in [section 5.1.3](#) that the semantic model for the Obc intermediate language is similar (although simpler).

However, it does not apply directly to dataflow languages, where memory update is left implicit. One point from this model that we could take inspiration from is the notion of trace, which is used to model side effects. While Vélus does not have side effects, they could be included via external CompCert C functions called from the Vélus code. The side effects of these functions could then be modeled using a trace, following the CompCert model. Since a dataflow program does not specify the order of evaluation of equations, the order of side effects within a cycle could only be partially specified; this would introduce non-determinism in the language, and complicate the proofs of semantic correspondence. We are not sure yet how to treat these issues.

2.11.1.2 CakeML

The semantics of CakeML are based on functional big-step semantics [Owe+16; Tan+19]. They are essentially defined by an interpreter for the language, implemented by a pure HOL [SN08] function. The function may return either a value or an error. To ensure its termination, even for programs that may not terminate, the function takes an extra fuel argument, that is a natural number which decreases at each recursive call; when it reaches 0, the function returns a `timeout` error.

This style of semantic model simplifies some proofs of correctness for compilation passes, as it allows for rewriting modulo reduction in proof terms, which is not true for relational semantics. It would be interesting to see if this strategy would work in the context of a dataflow synchronous language; this may not be true, because the functional semantics for a synchronous dataflow language are less straightforward to define, as we discuss below.

2.11.2 Possibly Finite Coinductive Streams

While our semantic model is relational, one might want to implement an executable semantic model of our language. In that case, it would be necessary to explicitly model runtime errors that can occur in the program. One way of doing so is to use streams that can end when an error arises. A naive definition for such streams is presented below.

```
CoInductive ErrorStream A : Type :=
| Cons : A -> ErrorStream A -> ErrorStream A
| Error : ErrorStream A.
```

Listing 2.20: Naive Possibly Terminating Stream

A more interesting definition is proposed in [Pau09]. In this paper, Paulin-Mohring discusses CPOs, an ordered structured which allows for the definition of well-founded fixpoints. The coinductive definition of streams, reproduced below, offers both the usual `Cons`, but also an `Eps` constructor, which only extends the stream without adding any element. This means a function building a stream can always be productive by adding `Eps` in front of a stream. In particular, with this representation, it is possible to write

a function `filter`: $(A \rightarrow \text{bool}) \rightarrow \text{Str } A \rightarrow \text{Str } A$ that only keeps element respecting a boolean predicate. This would not be possible with the coinductive types defined above, as the function would not be productive when rejecting the head of the stream. Moreover, it is possible to represent a terminating stream with an infinite sequence of `Eps` constructors.

```
CoInductive Str A : Type :=
| Eps : Str A -> Str A
| Cons : A -> Str A -> Str A.
```

Listing 2.21: Streams as CPOs [Pau09]

Although this definition would be practical to define an executable semantic model, it is not necessary when manipulating a relational model. Adding a second constructor would add useless complexity in the proofs, and we chose to work with the previous definition of `Stream` presented in [listing 2.8](#).

2.11.3 Synchronous Semantics for Dataflow Languages

2.11.3.1 Reaction semantics

In our work, we see a synchronous dataflow program as a set of constraints between its inputs and outputs. It can also be seen as a transition system. At every reaction, the system receives inputs, calculates outputs based on these inputs and an internal state, and updates this internal state. This type of reaction-based semantics for synchronous dataflow languages has been extensively studied [Cas+87; CP98; CHP06]. In each of these works, a program is associated to (i) an initial state and, and (ii) a transition function that transforms a state and computes some values. This may be defined either as relations between syntax, states and values, or as executable functions [Col+23], which provides an interpreter for the language. We do not know if such a model would be practical to mechanize the proof of correctness of source-to-source rewriting passes. Indeed, the shape of such a proof is outlined in [figure 2.24](#). An hypothetical compilation function rewrites program P into P' . The semantics of each of these two programs consists of an infinite series of reactions, each rewriting the internal state. However, the states associated with P and P' may not be equal, but only related by some bisimulation relation R . To prove the correctness of the source-to-source rewriting, we would need to find and define this R relation such that (i) the initial states St_0 and St'_0 are related by R , (ii) the transition function preserves R , and (iii) if input states St_i and St'_i are related by R , the values produced during the reaction are indeed equal. A specific R relation would be necessary for each rewriting step in the compiler. Although we have not tried this approach in Vélus, and do not know how much work it would require in practice, we believe that our approach of defining the semantics as a set of constraint, and proving that these constraints are preserved by source-to-source rewriting is easier to implement.

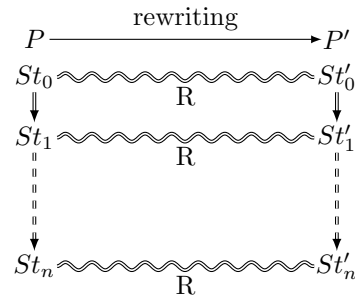


Figure 2.24: Bisimulation for source-to-source rewriting

2.11.3.2 Synchronous stream semantics in Coq

The stream semantics that we adopt for the core dataflow language are inspired by the definitions of [CP03, §3.2]. This semantic model has also been mechanized as a shallow embedding of the Lucid Sychrone language in Coq [BH01]. Lucid Sychrone expressions are represented directly by Coq terms that calculate sampled streams. The `sampleStr` type (reproduced below) is parameterized by a boolean clock, which indicates explicitly when the stream is present or absent. This dependent typing allows the authors to define most of the streams operators presented in this section as total coinductive functions.

```

Inductive sampleElt (A : Type) : bool -> Type :=
| Abs : sampleElt A false
| Pres : A -> sampleElt A true.

Definition clock := Stream bool.

CoInductive sampleStr (A : Type) : clock -> Type :=
| sp_cons (c : clock) :
  sampleElt A (hd c) -> sampleStr A (tl c) -> sampleStr A c.

```

Listing 2.22: Sampled streams from [BH01]

One advantage of this approach is that, as the title of the paper indicates, the clock correctness property comes essentially “for free”. While this is an interesting idea, shallow embedding does not seem practical in the context of a compiler, where we need to manipulate the AST freely. Additionally, our experience of working with dependently-typed streams is that statements and proofs are harder to write.

2.11.4 Modeling the Semantics of State Machines

2.11.4.1 StateCharts

The use of state machines for programming complex behaviors is rooted in a long history that begins with StateCharts [Har87]. In this work, D. Harel introduces a visual language for the specification of complex systems. These state machines are hierarchical: a state

can be refined by a number of sub-states. Two state machines can be composed in parallel to represent two sub-systems running independently. Transitions between states are triggered by external events, like the push of a button, and guarded by conditions expressed on global variables. Additionally, processing a transition may execute an action, like changing the value of a global variable. Such actions are not supported in our work, where transitions only indicate when to exit a state. A state may be entered “by history”, meaning that its sub-states will be restored to the configuration they had when the state was last exited. This is the inspiration for transitions with the `continue` keyword. [Har+87] establishes the operational semantics of this formalism as a transition system. In [HN96], the authors propose executable, reaction-based semantics for an implementation of StateCharts.

2.11.4.2 Stateflow

The Stateflow language [22] is used to specify controllers using StateCharts-like automata in the Simulink [13] environment. Transitions in Stateflow are more complicated than in Statecharts. A transition may lead not to a state, but to a junction of several transitions. A compound transition succeeds only if a sequence of these simple transitions eventually leads to a state. Actions on simple transitions can be set to trigger either if the simple transition is taken, or if the whole compound transition succeeds. The semantics of Stateflow have been studied by Hamon. [HR04] presents an operational model using reaction rules. These reaction rules are defined inductively on the structure of the automaton, represented textually as a program. When an event is received, the program reacts by rewriting, and possibly updating the environment through actions. In [Ham05], the author proposes a denotational semantic model, by representing the execution of an automaton as transition functions. This model is also used as a basis for a prototype compiler for Stateflow.

2.11.4.3 Lustre with modes: Mode-Automata

The idea of combining hierarchical state machines à la Statecharts with dataflow equations à la Lustre came first in the Mode-Automata language [MR98; MR03]. The language of states, or modes, is based on previous work by the same authors [MR01], itself based on StateCharts, with the major simplification of removing multi-level arrows, where a transition could specify which sub-state of a nested automaton is entered. Each mode may contain dataflow equations written in a subset of the Lustre language. This subset allows to define combinatorial computations and to access the previous global value of a variable using the `pre` operator (which is equivalent to `last`). Explicit sampling operators are not available.

The semantics of Mode-Automata are defined by translation. In [MR03], the authors show that a well-formed mode-automaton with parallel or hierarchical composition can be translated into a “flat” mode-automaton. In [MR98], they show that, in turn, any flat mode-automaton can be translated into a dataflow Lustre program. This gives a translation semantics to any well-formed mode-automaton.

2.11.4.4 Lucid Synchrone and Scade 6

The state machines treated in Vélus are most similar to the ones implemented in Lucid Synchrone. A first presentation of these state machines is given in [CPP05]. This paper proposes a translation-based semantic model for state machines and other control operators: the semantics of a high-level operator is exactly the semantics of lower-level operators in which it is compiled. Our work is similar to this proposition: the semantics of `switch`, for instance, is given using the `when` operator. We abstract somewhat from this idea by basing our definitions on semantic operators, rather than syntactic constructs. This eliminates some noise inherent to a compilation function that appear in the translation semantics of [CPP05]. A following paper, [CHP06] introduces a reaction-based relational semantics for the same language. In this model, the reset of blocks is handled by adding an extra bit k in reaction rules. $k = 0$ indicates that a block should react as if it were in its initial state. In particular, the reaction rules for the `fbv` and the initialization arrow depends on this bit. This bit is set to 0 whenever a block needs to be reset. This work on Lucid Synchrone, as well as the ReLuC prototype compiler, led to the addition of state machines to the Scade 6 language [CPP17].

Verified Dependency Analysis

A Vélus program is defined as a set of parallel equations under control blocks. The order of equations in a node does not matter. Each equation may depend on any of the streams defined by another. However, some dependencies are problematic. For instance, the equation $x = x + 1$ does not admit any solution, although it is well typed, because of its self-dependency. Conversely, the equation $x = x$ admits an infinity of solutions. Moreover, neither of these equations can be compiled to imperative code where a value must be computed before it is assigned. This is also the case for any system of equations containing a cycle of *instantaneous dependencies*, such as $x = y + 1$; $y = x * 2$. We say that a dependency is *instantaneous* when it is not broken by a `fb`. For instance, in $x = x0 \text{ fb } (x + 1)$, x depends instantaneously on $x0$, but not on itself.

In most compilers, programs that contain cyclic dependencies are detected statically by a *dependency analysis*, and rejected. The usual approach [Hal+91; Bie+08] is to analyse each node to calculate a graph of instantaneous dependencies between its inputs, local and output declarations. In this chapter, we show how this approach is adapted to Vélus source programs. In particular, we discuss the modifications necessary to handle control blocks. After formally defining the dependency rules of the language, we present a Coq implementation, and verification, of the algorithm used to check that the dependency graph is acyclic. The remainder of the chapter is focused on proving properties of well-formed programs. We first develop an induction scheme for acyclic programs. We then apply it to prove two major properties of the semantic model: determinism and clock correctness.

3.1 Dependency Graph of a Vélus Program

We first present the functions that build a dependency graph from a Vélus node. In a node, several distinct variables may use the same name (for instance, in parallel local declaration blocks, or in different branches of a state machines). To be able to distinguish them, each declaration x in the node is associated to a unique label α_x . Each label becomes a vertex in the dependency graph. The existence of an edge $\alpha_y \leftarrow \alpha_x$ in the

$$\begin{array}{c}
\frac{\Gamma(x) = \alpha_x}{\alpha_x \in \text{UsedInst}_\Gamma(x)[0]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(e)[0] \quad k < \text{numstreams}(\text{if } e \text{ then } es_0 \text{ else } es_1)}{\alpha \in \text{UsedInst}_\Gamma(\text{if } e \text{ then } es_0 \text{ else } es_1)[k]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(es_0)[k]}{\alpha \in \text{UsedInst}_\Gamma(\text{if } e \text{ then } es_0 \text{ else } es_1)[k]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(es_1)[k]}{\alpha \in \text{UsedInst}_\Gamma(\text{if } e \text{ then } es_0 \text{ else } es_1)[k]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(e)[k]}{\alpha \in \text{UsedInst}_\Gamma(e :: es)[k]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(es)[k]}{\alpha \in \text{UsedInst}_\Gamma(e :: es)[(\text{numstreams}(e) + k)]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(es_0)[k]}{\alpha \in \text{UsedInst}_\Gamma(es_0 \text{ fby } es_1)[k]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(es)[k'] \quad k < \text{numstreams}(f(es))}{\alpha \in \text{UsedInst}_\Gamma(f(es))[k]}
\end{array}$$

Figure 3.1: Instantaneously used labels  [Lustre/LCausality.v:31](#)

graph indicates that the variable associated to α_y depends instantaneously on the one associated to α_x . We define a function computing the set of labels used instantaneously in an expression and then use it to establish the set of dependencies induced by blocks.

3.1.1 Analysis of Expressions

In order to detect dependency cycles in the node, we must first determine the set of variables used instantaneously in an expression. [Bie+08, Figure 3] proposes a similar function, but it applies only to normalized nodes: for instance, the equation $(x, y) = (0, x)$ would be rejected by this analysis. We want to extend this function to non-normalized nodes. We also want our function to return the labels of the variables used instantaneously, rather than the variable names themselves. We therefore propose the function $\text{UsedInst}_\Gamma(e)[k]$ which returns the labels of variables used instantaneously to define the k th stream of expression e . In the abstract syntax, each variable is declared with a label. Environment Γ carries these associations.

We present UsedInst using inference rules. Writing $\alpha \in \text{UsedInst}_\Gamma(k)[e]$ means that label α is used instantaneously to define the k th stream of e . This property is decidable:

$$\forall \Gamma k e \alpha, \alpha \in \text{UsedInst}_\Gamma(k)[e] \vee \alpha \notin \text{UsedInst}_\Gamma(k)[e]$$

In our Coq mechanization, UsedInst is defined as an inductive relation, which is practical for proofs. We also defined an efficient function to collect the sets of labels used instantaneously in an expression. This function is proven complete with regards to the inductive definition.

Figure 3.1 presents a few interesting cases of the inductive definition of UsedInst . A variable always uses its own label, which is recovered from the environment. For the

$$\frac{\Gamma(x_i) = \alpha_{x_i}}{\alpha_{x_i} \in \text{Def}_\Gamma([x_i]^i = es)} \quad \frac{\Gamma(x_i) = \alpha_{x_i} \quad \alpha \in \text{UsedInst}_\Gamma(es)[i]}{\Gamma \vdash [x_i]^i = es \mid \alpha_{x_i} \stackrel{dep}{\leftarrow} \alpha}$$

(a) DefEq 🐦 [Lustre/LCausality.v:293](#) (b) DepOnEq 🐦 [Lustre/LCausality.v:333](#)

Figure 3.2: Dependencies for equations

compound expression **if-then-else**, we must consider all the labels used in the condition, as well as the labels used for the k th stream of either branch. An auxiliary definition is used to lift `UsedInst` to a list of expressions. In order to pick the right sub-expression in the list, that is, the one generating the k th stream, it needs to consider the number of streams `numstreams(e)` generated by each sub-expression. As discussed, for a **fb**y, we only consider the labels used in the left sub-expressions, as the ones used in the right are not used instantaneously. The case of node instantiation is also interesting. We choose to treat node instantiations as atomic: all outputs of the node depend instantaneously on all inputs. This is an over-abstraction, as some outputs may only depend on some, or none of the inputs. In [section 3.6.1](#), we discuss other approaches to dependency analysis that handle node instantiations modularly.

3.1.2 Dependencies induced by blocks

We now define the dependency constraints induced by each block. In the following, we write $\Gamma \vdash blk \mid \alpha_y \stackrel{dep}{\leftarrow} \alpha_x$ for “under environment Γ , α_y depends on α_x in block blk ”. In order to define this judgment, we will need to refer to the labels defined by a block blk , which we collect with the function $\text{Def}_\Gamma(blk)$. Like `UsedInst`, we define this function using inference rules, as it is defined as a relation in Coq.

Equations The labels defined by an equation are those associated to the variables at left of the equation. Additionally, an equation induces dependencies between the k th variable it defines, and the variables used instantaneously in the k th stream of the expressions at right. For instance, in the equation $(x, y) = (0, x \text{ fb}y z)$, x does not depend on anything, and y depends solely on x . [Figure 3.2](#) presents the formalization of these intuitions. In the dependency rule, we use the function `UsedInst` to collect the labels used instantaneously in the k th stream of the expressions.

Local Declarations Each local declaration block introduces new labels *locs*. They should be accounted for when treating the blocks in the scope of the declaration. The environment is extended with the append operation, $+$, defined as

$$\forall x, (\Gamma + locs)(x) = \alpha_x \iff (\Gamma(x) = \alpha_x \vee locs(x) = \alpha_x)$$

The rules for defined labels and dependencies for a local declaration are given in [figure 3.3](#). They simply consists in looking recursively at the sub-blocks, after augmenting

$$\begin{array}{c}
\frac{\alpha \in \text{Def}_{(\Gamma + \text{locs})}(\text{blks})}{\alpha \in \text{Def}_{\Gamma}(\text{var } \text{locs } \text{let } \text{blks } \text{tel})} \\
\text{(a) DefScope1 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:279}
\end{array}
\qquad
\begin{array}{c}
\frac{(\Gamma + \text{locs}) \vdash \text{blks} \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x}{\Gamma \vdash \text{var } \text{locs } \text{let } \text{blks } \text{tel} \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x} \\
\text{(b) DepOnScope1 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:318}
\end{array}$$

Figure 3.3: Dependencies for local declarations

$$\begin{array}{c}
\frac{\Gamma \vdash \text{blks} \mid \alpha_1 \xleftarrow{\text{dep}} \alpha_2}{\Gamma \vdash \text{reset } \text{blks } \text{every } e \mid \alpha_1 \xleftarrow{\text{dep}} \alpha_2} \\
\text{(a) DepOnReset1 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:344}
\end{array}
\qquad
\begin{array}{c}
\frac{\alpha_x \in \text{UsedInst}_{\Gamma}(e)[0] \quad \alpha_y \in \text{Def}_{\Gamma}(\text{blks})}{\Gamma \vdash \text{reset } \text{blks } \text{every } e \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x} \\
\text{(b) DepOnReset2 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:347}
\end{array}$$

Figure 3.4: Dependencies for `reset` blocks

the environment. The labels defined in the local declaration escape its scope. This is necessary to build a dependency graph global for the whole node.

Reset blocks Two types of dependencies are induced by a `reset` block, presented in [figure 3.4](#). First, there are the dependencies induced by the sub-blocks. Second, we must take into account the labels used in the control expression. Indeed, the values of all variables defined by the sub-blocks may be reset depending on the value of this expression. This control dependency is expressed in the rule of [figure 3.4b](#).

Switch blocks The treatment of `switch` blocks is more complex. Consider the node of [listing 3.1](#). In the first branch, `y` depends instantaneously on `x`. In the second branch, `x` depends instantaneously on `y`. This apparent cycle is not a problem: the two branches are exclusive, meaning that at each step, at most one of them is active to define the values of `x` and `y`. Moreover, the compilation scheme that we present in [section 4.8](#) produces schedulable code for this program. As such, there is no reason to reject this node.

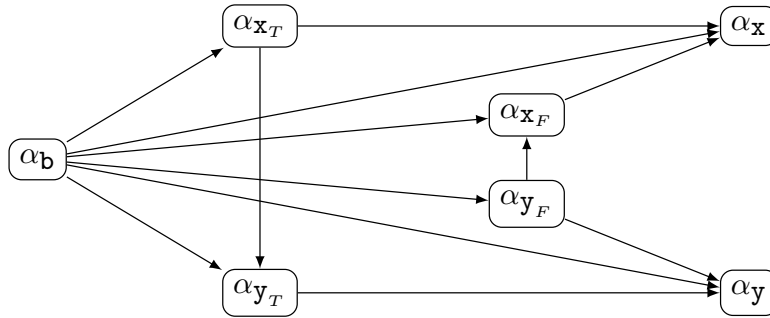
```

node switch_dep(b : bool) returns (x, y : int)
let
  switch b
  | true do x = 0 fby (x + 1); y = x * 2;
  | false do y = 0 fby (y - 1); x = y * 2;
  end
tel

```

Listing 3.1: Node with switched dependencies

To accept such definition, we need a special treatment in the dependency analysis. Indeed, if we simply associate labels α_x and α_y to `x` and `y`, then the dependency analysis would find a cycle between these two labels. Instead, we associate different labels to the definitions of `x` and `y` in each branch. For instance, α_{x_T} would be the label associated

Figure 3.5: Dependency graph for the node `switch_dep`

to the first definition of x , and α_{x_F} the one associated to its second. Reading variable x in the first branch means using α_{x_T} , but not α_{x_F} , or α_x . The global value of x depends itself on its values local to branches. In other words, α_x depends on α_{x_T} and α_{x_F} . All of these values also depend on the value of the condition on the switch, which dictates when they are active. The graph associated with node `switch_dep` is presented in [figure 3.5](#). It is indeed acyclic.

To encode these changes in labels, each branch of a `switch` is annotated, in the AST, by a function σ that associates new labels to some variables. A static invariant not presented here ensures that the labels renamed by σ are only the ones of variables defined by the `switch`. Applying σ to an environment yields:

$$\sigma(\Gamma)(x) = \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \\ \Gamma(x) & \text{otherwise.} \end{cases}$$

[Figure 3.6](#) presents the dependency rules for a `switch` based on this idea. Two rules are used for `Def`. The first one is usual and inspects the sub-blocks after having applied σ to the environment. The second ensures that the “global” labels of defined variables, that is the ones that were renamed by σ , are also considered defined by the `switch`. For instance, in the example discussed above, α_{x_T} , α_{x_F} , and α_x are all defined by the `switch`. The first two dependency rules are similar to the ones for `reset`: they inspect the dependencies of sub-blocks, as well as control dependencies between the control expression and variables defined by the `switch` block. The last rule establishes, for each variable, the dependency between its branch-local definitions and its global definition for the whole `switch`.

The dependency rules for state machines are defined in the same manner, by replacing the dependencies on the control expression by dependencies on the initialization conditions and on the strong transitions. For concision, we do not present these rules here.

Last variables Consider the node below: x depends instantaneously on `last x`, which does not depend on anything. Even if using `last` in this example is not particularly useful compared to `fby`, the same kind of dependencies appear in more complex nodes like

$$\begin{array}{c}
\frac{\alpha \in \text{Def}_{\sigma_i(\Gamma)}(\text{blks}_i)}{\alpha \in \text{Def}_{\Gamma}(\text{switch } e [C_i \text{ do}_{\sigma_i} \text{ blks}_i]^i \text{ end})} \\
\text{(a) DefBranch1 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:284}
\end{array}
\qquad
\frac{x \in \text{dom}(\sigma_i) \quad \Gamma(x) = \alpha}{\alpha \in \text{Def}_{\Gamma}(\text{switch } e [C_i \text{ do}_{\sigma_i} \text{ blks}_i]^i \text{ end})} \\
\text{(b) DefBranch2 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:287}$$

$$\frac{\sigma_i(\Gamma) \vdash \text{blks}_i \mid \alpha_1 \xleftarrow{\text{dep}} \alpha_2}{\Gamma \vdash \text{switch } e [C_i \text{ do}_{\sigma_i} \text{ blks}_i]^i \text{ end} \mid \alpha_1 \xleftarrow{\text{dep}} \alpha_2} \\
\text{(c) DepOnBranch1 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:324}$$

$$\frac{\alpha_x \in \text{UsedInst}_{\Gamma}(e)[0] \quad \alpha_y \in \text{Def}_{\Gamma}(\text{switch } e [C_i \text{ do } \text{ blks}_i]^i \text{ end})}{\Gamma \vdash \text{switch } e [C_i \text{ do } \text{ blks}_i]^i \text{ end} \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x} \\
\text{(d) DepOnSwitch2 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:355}$$

$$\frac{\Gamma(x) = \alpha_y \quad \sigma_i(x) = \alpha_x}{\Gamma \vdash \text{switch } e [C_i \text{ do}_{\sigma_i} \text{ blks}_i]^i \text{ end} \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x} \\
\text{(e) DepOnBranch2 } \color{red}{\text{🐦}} \text{ Lustre/LCausality.v:327}$$

Figure 3.6: Dependencies for `switch` blocks

`drive_sequence` presented in the introduction. In these cases, associating the same label to `x` and `last x` would create a false dependency cycle. An extra label must therefore be introduced to denote the instantaneous usage of the `last` value of a variable.

```

node last_dep(i : int) returns (x : int)
let
  last x = 0;
  x = last x + i;
tel

```

To support this, we extend the environment Γ . We note $\Gamma(\text{last } x)$ for the label associated with `last x` in Γ . [Figure 3.7](#) presents the dependency rules for `last` variables. As expected, the expression `last x` uses the label associated with `last x` in the environment. An equation of the form `last x = e` defines the label associated with `last x`. This label depends instantaneously on all labels used in `e`.

Partial Definitions The dependency analysis of partial definitions is more complicated. Consider the `register` node below. The value of `x` is not declared in the second branch. This program is valid, as the second branch is implicitly completed with `x = last x`. This means that this program induces an implicit dependency from `last x` to `x`.

$$\begin{array}{c}
\frac{\Gamma(\mathbf{last}\ x) = \alpha_{\mathbf{last}\ x}}{\alpha_{\mathbf{last}\ x} \in \text{UsedInst}_{\Gamma}(\mathbf{last}\ x)[0]} \\
\text{(a) IUlast } \color{red}{\text{Lustre/LCausality.v:35}}
\end{array}
\qquad
\begin{array}{c}
\frac{\Gamma(\mathbf{last}\ x) = \alpha_{\mathbf{last}\ x}}{\alpha_{\mathbf{last}\ x} \in \text{Def}_{\Gamma}(\mathbf{last}\ x = e)} \\
\text{(b) DefLast } \color{red}{\text{Lustre/LCausality.v:297}}
\end{array}$$

$$\frac{\Gamma(\mathbf{last}\ x) = \alpha_{\mathbf{last}\ x} \quad \alpha_y \in \text{UsedInst}_{\Gamma}(e)[0]}{\Gamma \vdash \mathbf{last}\ x = e \mid \alpha_{\mathbf{last}\ x} \xleftarrow{\text{dep}} \alpha_y}$$

(c) DepOnLast 🐔 Lustre/LCausality.v:339

Figure 3.7: Dependencies for `last` variables

```

node register(i : int; b : bool) returns (x : int)
let
  last x = 0;
  switch b
  | true do x = i;
  | false do
  end
tel

```

Such implicit dependencies are more difficult to specify, analyse, and reason about. To simplify our mechanization, we decided to not treat partial definitions in the dependency analysis. In practice, this means that the compilation pass that completes implicit definitions (discussed in [section 4.5](#)) runs before the dependency analysis.

3.2 Verified Graph Analysis

The dependency rules described in the previous section are used to build a dependency graph for each node in the program. The graph contains a vertex for each causality label in the node. Each dependency from α_x to α_y is represented by an edge in the graph. In Coq, this graph is represented by a map associating each label to the list of labels it depends on (its predecessors). Of course, this representation does not guarantee the absence of dependency cycles in the graph.

```

Definition graph := Env.t (list ident).

```

Listing 3.2: Dependency graph as a map from a label to its predecessors

To characterize acyclic graphs, we introduce the inductive predicate `AcyGraph VA`, where V is a set of vertices (`PS.t` in Coq), and A a set of edges, represented as a map associating each vertex to its set of successors (`Env.t PS.t` in Coq). It is defined by the three inductive rules presented in [figure 3.8](#). An empty graph is acyclic. Adding a vertex

$$\begin{array}{c}
\frac{}{\text{AcyGraph } \emptyset \emptyset} \qquad \frac{\text{AcyGraph } V E}{\text{AcyGraph } (V \cup \{\alpha\}) A} \\
\frac{\text{AcyGraph } V E \quad \alpha_x, \alpha_y \in V \quad \alpha_x \neq \alpha_y \quad \alpha_y \not\rightarrow_E^+ \alpha_x}{\text{AcyGraph } V (E \cup \{\alpha_x \rightarrow \alpha_y\})}
\end{array}$$

Figure 3.8: Inductive representation of an Acyclic Graph 🐔 [AcyGraph.v:150](#)

to a graph maintains its acyclicity. However, an edge from α_x to α_y may only be added if α_x and α_y are distinct, and if there is no existing transitive return arc. This simple constraint is enough to guarantee the acyclicity property: for any label α , there can be no transitive arc from α to α . This is stated formally below.

Lemma 3 (Acyclicity of an Acyclic Graph 🐔 [AcyGraph.v:235](#))

$$\text{if } \text{AcyGraph } V E \text{ then } \forall \alpha, \alpha \not\rightarrow_E^+ \alpha$$

We now need to develop a function to analyse the dependency graph `gr` built from a node and represented by the type in [listing 3.2](#). If this function succeeds, it should produce a witness of `AcyGraph`, which we can later use to reason about the node. We implement, in Coq, a classic depth-first search algorithm to analyse the dependency graph. To justify its correctness and termination, we use dependant types to encode invariants of the algorithm. We use the `Program` extension [[Soz07](#)] which allows us to define the algorithm in Gallina, Coq’s programming language, while handling proofs obligations separately, with the tactic language Ltac.

We now describe this implementation, presented in [listing 3.3](#). For readability, this presentation employs set notations, monadic notations and implicit coercions from a subset type to its value type. It is otherwise identical to the function implemented in our mechanization. A call to `dfs' s x v` recursively traverses the predecessors of `x` in the graph `gr` (given as a fixed parameter). Input `v` represents a set of vertices already visited. Input `s` represents the *state* of the search, that is, the set of vertices encountered during the traversal. The subset type `dfs_state` specifies that `s` must be a subset of the vertices of the dependency graph `gr`. At the end of its execution, `dfs'` returns the new set of visited vertices `v'`. It is a superset of `v`. We use the `error` monad to represent the possible failure of the algorithm. The function fails if it encounters a vertex that was already in `s` ([line 14](#)): this would mean there is a cycle in the graph, as this vertex was already visited during this traversal. If $x \in v$, then we know that all the predecessors of `x` are already in `v`, and we can simply return `v` ([line 17](#)). Otherwise, the list of predecessors of `x` is read from `gr` ([line 19](#)), and `dfs'` is called recursively on each of them ([line 24](#)) with the extended state `s'`. If all recursive calls succeed, `x` is added to the set of visited vertices, which is then returned.

```

1 Variable gr : graph.
2
3 Definition dfs_state := { p | ∀ x, x ∈ p -> x ∈ gr }.
4 Definition num_remaining (s : dfs_state) : nat := | gr | - | s |.
5
6 Definition visited (s : PS.t) (v : PS.t) : Prop :=
7   (∀ x, x ∈ s -> x ∈ v)
8   ∧ ∃ e, AcyGraph v e
9     ∧ (∀ x, x ∈ v -> ∃ zs, find x graph = Some zs ∧ (∀ y, y ∈ zs -> has_edge e y x)).
10
11 Program Fixpoint dfs' (s : dfs_state) (x : ident) (v : { v | visited s v })
12 {measure (num_remaining s)} : res { v' | visited s v' ∧ x ∈ v' ∧ v ⊆ v' } :=
13   match mem x s with
14   | true => Error (msg "dependency cycle")
15   | false =>
16     match mem x v with
17     | true => OK (exist _ v _)
18     | false =>
19       match find x gr with
20       | None => Error (CTX x :: msg " not found")
21       | Some zs =>
22         let s' := exist _ ({ x } ∪ s) _ in
23         match
24         fold_left (fun v w => do v' <- dfs' s' w v; OK (exist _ v' _)) zs (OK v)
25         with
26         | Error msg => Error msg
27         | OK v' => OK (exist _ ({ x } ∪ v') _)
28         end
29       end
30     end
31   end.
32
33 Definition dfs x (v : { v | visited ∅ v }) :
34   res { v' | visited ∅ v' ∧ x ∈ v' ∧ v ⊆ v' } :=
35   dfs' ∅ x v.

```

Listing 3.3: Certifying graph analysis 🐔 [AcyGraph.v:619](#)

At lines 13 and 16, we use the construction `match/with` rather than `if/then/else` to handle a boolean condition. This is because, when using `if/then/else`, `Program` loses some information about the value of the condition when generating proof obligations.

Termination Coq functions must always terminate. For recursive functions, this is enforced by the guarded recursion criterium: one of the arguments must be of an inductive type, and the function may only be called recursively on strict sub-terms of this argument. The `dfs'` function does not respect this criteria. To sidestep this issue, we use the `Program Fixpoint` command, which allow us to prove the termination of the function by providing a *measure* on one of the arguments. The measure returns a natural number (type `nat`), which should be strictly decreasing on each recursive call. Since (\mathbb{N}, \leq) is well

founded, this is enough to prove that the function terminates.

For this algorithm, we use the measure `num_remaining`, which computes the difference between the number of vertices in the original graph `gr`, and the number of vertices in the current traversal, represented by set `s`. Intuitively, it represents the maximum number of vertices that could still be traversed by recursive calls to `dfs'`. This measure is indeed decreasing, because, before each recursive call, we add vertex `x` to `s`. We know from the tests at `lines 13 and 19` that `x` was not previously in `s`, and that it is in `gr`, and so the measure decreases by 1 at each recursive call.

Correctness The algorithm is correct if its success implies the existence of an `AcyGraph` associated with the graph `gr`. It is difficult to reason a posteriori on a function defined using `Program Fixpoint`, because the corresponding Coq term contains dependant types and proofs justifying its termination. These terms often do not reduce properly, which makes proofs cumbersome. Instead, we integrate the invariants of the algorithm into the type of the function.

The invariant `visited s v` gives two guarantees. First, the set of visited edges `v` and currently traversed edges `s` are disjoint. This is a technical detail necessary for the proof. More importantly, it ensures the existence of a set of edges `e` forming an `AcyGraph` together with `v`. This set of edges must be complete, in the sense that any edge in `gr` going to a vertex of `v` should be represented in `e`. This ensures that `v` indeed forms an (acyclic) prefix of the dependency graph. It is not necessary to compute the set of edges of the `AcyGraph` being built. Indeed, on `line 8`, `e` is given as an existential, which can be erased when extracting the function to OCaml code, and does not incur any runtime cost.

The type of `dfs'` ensures that, given a set of visited vertices `v`, the algorithm, if it succeeds, returns a set of visited vertices `v'` which subsumes `v`, and contains the traversed vertex `x`. The `dfs` function specializes `dfs'`, starting with an empty traversed set. Iterating this function on every vertex in the dependency graph ultimately provides a witness of `AcyGraph` that is complete with regard to the dependency graph, and therefore to the analysed node. We say that the node is *causal*, as formalized below.

Definition 3 (*Causal node* 🐔 [Lustre/LCausality.v:585](#))

$$\text{node_causal } n \quad \text{iff} \quad \exists VE, \text{AcyGraph } VE \wedge (\forall \alpha_x \alpha_y, \vdash n \mid \alpha_y \stackrel{dep}{\leftarrow} \alpha_x \implies \alpha_x \rightarrow_E \alpha_y)$$

Complexity This algorithm is time-efficient as each vertex is only traversed once before being placed in the visited set. We have not tried to verify the running time complexity of this algorithm, as we were more interested in verifying its functional correctness. We believe that, after instrumenting the code, it would be easy to prove that this algorithm runs in linear time with respect to the number of edges in the graph.

3.3 Induction Schemes for Causal Programs

In the following sections, we will be interested in proving properties of the semantic model for programs that do not contain dependency cycles. To do so, we first use the definitions

$$\begin{array}{c}
 \overline{\text{TopoOrder}(\text{AcyGraph } VE) \quad []} \\
 \\
 \frac{\text{TopoOrder}(\text{AcyGraph } VE) \text{ lord} \quad \alpha_y \in V \quad \neg \alpha_y \in \text{lord} \quad (\forall \alpha_x, \alpha_x \xrightarrow{+}_E \alpha_y \implies \alpha_x \in \text{lord})}{\text{TopoOrder}(\text{AcyGraph } VE) (\alpha_y :: \text{lord})}
 \end{array}$$

 Figure 3.9: Topological Ordering of Vertices of a graph 🐔 [AcyGraph.v:690](#)

of the two previous sections to build two induction schemes that will form the backbone of these proofs.

3.3.1 Induction on the labels of a node

Suppose that we want to establish a property of the semantic model. In general, we can express such a property as a predicate on streams $\text{Pstream} : \text{Stream svalue} \rightarrow \text{Prop}$ that should hold for each stream associated to a syntactic element of the program by the semantic model. For now, consider that there is a global history H that associates every variable in a node to a stream; we will later see how to adapt this approach to support histories associated with local declarations. The Pstream property should hold for all streams in H . As a first approximation, we could state this as:

if $G, H, bs \vdash blk$ **and** $H(x) \equiv vs$ **then** $\text{P_stream } vs$

How should we prove that this is true for every variable? Consider the equation $y = x + 1$. In order to prove any interesting property of the stream associated with y , we need to know about the stream associated with x . More generally, the stream associated with a variable y *depends*, in the semantic sense, on the streams associated with the variables that y *depends on*, in the syntactic sense defined in the previous sections. For a node with no dependency cycle, we may prove Pstream for every named stream in the node by *following* these dependencies: we first prove Pstream for a named stream that does not depend on any other, then for a second one that only depends on that first one, and so on by induction. We now discuss the details of this approach.

If a node n is `node_causal`, then, by definition, it can be associated with an acyclic graph $\text{AcyGraph } VE$. To facilitate inductive reasoning, we extract a topological ordering of the vertices from this graph, that is a list where each vertex only depends on vertices that appear earlier in the list. In [figure 3.9](#), we present an inductive definition for this order, where the last premise of the second rule takes into account transitive edges/dependencies. This premise can be weakened to only take into account immediate edges ($\alpha_x \xrightarrow{+}_E \alpha_y$). We have proven that this definition would be equivalent, but it is less convenient in mechanized proofs. The lemma below establishes that, for any `AcyGraph`, there exists at least one complete topological ordering of the vertices.

Lemma 4 (Existence of a Topological Ordering 🐉 [AcyGraph.v:921](#))

iff $\text{AcyGraph } VE$
then $\exists \text{lord}, (\forall \alpha_x \in V, \alpha_x \in \text{lord}) \wedge \text{TopoOrder } (\text{AcyGraph } VA) \text{ lord}$

Recall that, in a dependency graph, each vertex corresponds to one label in the node. The proof proceeds by induction over the topologically ordered list of labels. We pose

$$\text{Pvar } \Gamma \alpha_x := \forall x \text{ vs}, \Gamma(x) = \alpha_x \implies H(x) \equiv \text{vs} \implies \text{Pstream } \text{vs}$$

Since reasoning uses labels rather than variables, $\text{Pvar } \Gamma \alpha_x$ recovers the variable associated with label α_x , and ensures that Pstream holds for the corresponding stream. This indirection complicates our proofs slightly, but is necessary to handle the full expressivity of the language.

The induction principle on the ordered list of labels is presented in [lemma 5](#). As expected, it applies only to causal nodes. Function `locals` collects the associations between local variables of the block and their labels. This includes the labels for `last` variables, and the labels local to a `switch` branch. The last premise gives the shape of the inductive hypothesis: for each label α_y , if $\text{Pvar } \Gamma$ holds for every predecessor of α_y , then $\text{Pvar } \Gamma$ must hold for α_y . Applying the induction shows that $\text{Pvar } \Gamma$ holds for any label of the node.

Lemma 5 (Induction on labels of a causal node)

if `node_causal (node f (ins) returns (outs) blk)`
and $\Gamma = \text{ins} + \text{outs} + \text{locals } \text{blk}$
and $(\forall \alpha_y, (\forall \alpha_x, (\text{ins} + \text{outs}) \vdash \text{blk} \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x \implies \text{Pvar } \Gamma \alpha_x) \implies \text{Pvar } \Gamma \alpha_y)$
then $(\forall \alpha_y, \text{Pvar } \Gamma \alpha_y)$

We now sketch the proof of this lemma. By the definition of `node_causal`, the first hypothesis implies the existence of an acyclic graph $\text{AcyGraph } VE$ from which, by [lemma 4](#), we extract a topological order of labels $\text{TopoOrder } (\text{AcyGraph } VE) \text{ lord}$. The set of vertices V and the list of labels lord correspond exactly to the set of labels in Γ . Proving that Pvar is true for every label in Γ is therefore equivalent to proving it for every label in lord . The goal becomes $\forall \alpha_y, \alpha_y \in \text{lord} \implies \text{Pvar } \Gamma \alpha_y$.

We use the usual induction mechanism of Coq on the witness of `TopoOrder`. The initial case is trivial: $\text{Pvar } \Gamma$ is obviously true for every label in the empty list. The inductive case proceeds as follows. We have $\text{TopoOrder } (\text{AcyGraph } VE) (\alpha_y :: \text{lord})$. By induction, $\text{Pvar } \Gamma$ holds for all labels in lord . To prove that it holds for all labels in $\alpha_y :: \text{lord}$ it remains to prove $\text{Pvar } \Gamma \alpha_y$. Suppose α_x , a label on which α_y depends. By the definition of `node_causal`, we know that there is an edge from α_x to α_y in E . By the definition of `TopoOrder`, α_x appears in lord . By the inductive hypothesis, $\text{Pvar } \Gamma \alpha_x$ holds. Therefore, the last premise of the lemma applies, which concludes the induction.

The lemma on the next page is an alternative scheme which turns out to be more practical than [lemma 5](#). It manipulates the predicate `Pvars` which applies to lists of labels.

The predicate must be true for the empty list. In the inductive case, there is one more hypothesis: for every label α_x on which the head label α_y depends, α_x appears in the tail of the list. The conclusion of this lemma is that `Pvars` holds for a permutation of the list of all labels in the node.

If `Pvars` simply consists in lifting `Pvar` to the labels in the list, then [lemma 5](#) is a corollary of [lemma 6](#).

Lemma 6 (Induction on the list of labels of a causal node 🐔 [Lustre/LCausality.v:2580](#))

```

if   node_causal (node f (ins) returns (outs) blk)
and   Pvars []
and   (∀lord αy, Pvars lord ⇒
        (∀αx, (ins + outs) ⊢ blk | αy ←dep αx ⇒ αx ∈ lord) ⇒
        Pvars (αy :: lord))
then  ∃lord, Permutation lord (ins + outs + locals blk) ∧ Pvars lord
    
```

3.3.2 Induction on the syntax of blocks and local declarations

Until now, we have reasoned globally about a node, with a single environment Γ that associates labels to variable names, and a single history H that associates variable names to streams. However, Vélu programs may contain arbitrarily nested local declarations. Since labels are globally unique, and escape their scope, we can still work with a single Γ that takes into account every label in the node. This is not so simple for histories. As shown previously in [figure 2.19a](#), the semantics of the blocks under a local declaration is relative to an extended history $H + H'$. This local history H' associates each local variable to a stream, and is quantified existentially in the rule. In a sense, it is “hidden” by this rule, and it is therefore not possible to reason globally on the content of H' . We now detail two solutions we have considered for this problem, and explain why the first fails.

Globalizing the History A first solution would be to expose the local histories globally. Practically, this would mean defining a new version of the semantic model, with the rule for local declarations of [figure 2.19a](#) changed to the one presented in [figure 3.10](#), at left. Since variable names are not necessarily unique in the node, we must change the history so that, instead of associating variable names to streams, it associates labels to streams. This level of indirection has a cost. As the labels are not directly accessible in the syntax, an environment needs to be added to the semantic rule. For instance, the rule for variables, presented in [figure 3.10](#) at right, must be modified to recover the label α_x associated with variable x before reading the stream associated with α_x in the global history. These changes need to be propagated to the rest of the rules, and we therefore have to define a new, globalized version of the whole semantic model defined in the previous chapter.

$$\frac{G, H, bs, \Gamma \vdash_{\text{glob}} \text{blks}}{G, H, bs, \Gamma + \text{locs} \vdash_{\text{glob}} \text{var} \text{ locs} \text{ let} \text{ blks} \text{ tel}} \quad \frac{\Gamma(x) = \alpha_x \quad H(\alpha_x) \equiv vs}{G, H, bs, \Gamma \vdash_{\text{glob}} x \Downarrow [vs]}$$

Figure 3.10: Alternate rule for local declarations with globalized histories

The additional definitions are not the only problem. To use this alternative semantic model, we must first derive it from the original semantic model. When trying to build the globalized semantics of blocks from the (local) semantics of blocks, we would need to prove the conjecture below. It states the existence of a globalized history H' that gives a globalized semantics to the block blk , and that corresponds to the initial history H .

Conjecture 1 (Globalizing the semantics of a block)

$$\begin{aligned} & \text{if } G, H, bs \vdash \text{blk} \\ & \text{then } \exists H', (G, H', bs, \Gamma \vdash_{\text{glob}} \text{blk}) \\ & \quad \wedge (\forall x \alpha_x, \Gamma(x) = \alpha_x \implies H(x) \equiv vs \implies H'(\alpha_x) \equiv vs) \end{aligned}$$

Unfortunately, it is not so easy to inductively construct the globalized history H' . Consider the case of the `reset` block, whose semantic rule was presented in [figure 2.16](#). The last premise states that $\forall k, G, \text{mask}_{rs}^k(H, bs) \vdash \text{blks}$, that is, the semantics of the underlying block can be given under an infinite family of instances of H . Applying our hypothesis inductively on each history of this family, we would have the following result:

$$\begin{aligned} & \forall k, \exists H'_k, (G, H'_k, bs, \Gamma \vdash_{\text{glob}} \text{blks}) \\ & \quad \wedge (\forall x \alpha_x, \Gamma(x) = \alpha_x \implies \text{mask}_{rs}^k H(x) \equiv vs \implies H'_k(\alpha_x) \equiv vs) \end{aligned}$$

There is a different H'_k for each instance of H . In order to give a semantics to the `reset` block, we need to build a unique history H' of which all H'_k are instances (formally, $\forall k, H'_k \equiv \text{mask}^k_{rs} H'$). This is only possible if the streams in each H'_k “respect” the masking, that is, they are all of the form $\text{mask}^k_{rs} xs$. Although this is a corollary of the clock-correctness property, reasoning on causal nodes is necessary to prove it. This cycle defeats the approach.

Instrumenting the semantic model The globalization approach tried to move the local histories “out” of the local scopes, so as to state the `Pstream` property globally. The alternative that we now present consists in reasoning more locally about `Pstream`, while still using a global induction scheme. It is implemented by introducing an “instrumented” semantic rule for local declarations, shown in [figure 3.11](#). It takes an extra parameter, *lord*, which corresponds to the ordered list of labels manipulated by induction when proving [lemma 5](#). Compared to the original rule for local declarations, it has one extra premise. It specifies that, for each label α_x in *lord*, `Pvar` holds. By the definition of `Pvar`, this means that, for each variable x associated with α_x in the local declarations *locs*, `Pstream` is true for the stream associated with x in the local history H' .

$$\frac{\forall x, x \in \text{dom}(H') \iff x \in \text{locs} \quad G, H + H', bs, \text{lord} \vdash_p \text{blks} \quad \forall \alpha_x, \alpha_x \in \text{lord} \implies \text{Pvar locs } \alpha_x}{G, H, bs, \text{lord} \vdash_p \text{var locs let blks tel} }$$

Figure 3.11: Instrumented semantic rules for local declarations

These changes do not affect the existing parameters of the judgment. Therefore, the semantic rules for variables, and more generally for expressions, do not need to be changed. We only need to redefine the semantic rules for blocks.

Instrumented semantic and induction on labels The instrumented semantic model is used as part of the Pvars predicate of [lemma 6](#). The initial case is easy to prove: if *lord* is empty, the third premise of [figure 3.11](#) holds trivially; therefore, a non-instrumented semantics implies the existence of an instrumented semantics, as stated below.

Lemma 7 *Existence of an instrumented semantic model*

$$\text{if } G, H, bs \vdash \text{blk} \text{ then } G, H, bs, [] \vdash_p \text{blk}$$

The proof for the inductive case proceeds by induction on the syntax of the block, to find the equation defining the variable y associated to the label α_y at the head of *lord*. Proving that Pstream holds for the stream associated to α_y requires reasoning on the expression defining the value of y , as explained below.

3.3.3 Induction on the k th stream of an expression

Consider the equation $(x_1, \dots, x_n) = \text{es}$. The stream associated to x_k is the k th stream produced by *es*. To establish Pstream for this stream, we must first reason by induction on the k th stream generated by *es*. This is the purpose of the induction principle we develop in this section. We pose

$$\text{Pexp } e \ k := \forall vs, G, H, bs \vdash e \Downarrow vs \implies \text{Pstream}(vs[k])$$

This predicate states that Pstream holds for the k th stream of an expression e . We also define a secondary predicate, Pexps $es \ k$, which lifts Pexp to a list of expressions. The idea is the following: if Pvar Γ holds for all variables used instantaneously to define the k th stream of expression e , then Pexp $e \ k$ must hold. The induction scheme corresponding to this intuition is presented on the next page. It is derived directly from the definition of UsedInst presented in [figure 3.1](#).

Lemma 8 (Induction on the k th stream of an expression 🐔 [Lustre/LCausality.v:2450](#))

if $\text{Pexp } c \ 0$
and $\forall x \alpha_x, \Gamma(x) = \alpha_x \implies \text{Pvar } \Gamma \ \alpha_x \implies \text{Pexp } x \ 0$
and $\forall k, \text{Pexp } e \ 0 \wedge \text{Pexps } es_0 \ k \wedge \text{Pexps } es_1 \ k \implies \text{Pexp } (\text{if } e \ \text{then } es_0 \ \text{else } es_1) \ k$
and $\forall k, \text{Pexps } es_0 \ k \implies \text{Pexp } (es_0 \ \text{fby } es_1) \ k$
and $(\forall k', \text{Pexps } es \ k') \implies \forall k, \text{Pexp } f(es) \ k$
and \dots
then $\forall e \ k, (\forall \alpha_x, \alpha_x \in \text{UsedInst}_\Gamma(e)[k] \implies \text{Pvar } \Gamma \ \alpha_x) \implies \text{Pexp } e \ k$

The conclusion of this scheme states that Pstream is valid for the k th stream of any expression, if all labels used instantaneously in this expression to define the k th stream satisfy Pvar . Each premise of this lemma represents either a base case, or an inductive step. Using this induction scheme requires proving these premises for a property Pstream . We have only presented here a few of the premises: there is one for each constructor in the syntax of expressions. Pstream should always be satisfied by the stream generated by a constant expression. For a variable x , if Pvar is satisfied for the label associated with x , then Pstream should be satisfied for the stream associated with the variable. By the definition of Pvar , this is immediate. For the inductive **if-then-else** step, the available hypotheses are that Pexp holds for the condition stream, as well as the k th streams of both alternatives. Proving this premise amounts to proving that the case semantic operator preserves Pstream .

Following the definition of UsedInst , in the **fby** case, the only hypothesis of the premise is that Pexp holds for the k th stream of the left sub-expressions. This is because the right sub-expressions may depend arbitrarily on other streams. Therefore, **fby** should preserve Pstream even if only its left operand respects Pstream :

$$\forall vs_0 \ vs_1 \ vs, \text{Pstream } vs_0 \implies \text{fby } vs_0 \ vs_1 \equiv vs \implies \text{Pstream } vs$$

The last interesting case is the one of a node instantiation. Since we impose that all outputs of a node depend instantaneously on all inputs, the inductive case assumes that Pexp holds for each sub-expression used as a parameter, and concludes that it holds for each stream generated by the instantiation. Establishing it requires proving that the instantiated node also preserves the Pstream property, which we can state as:

$$\text{Pnode } f \ xs \ ys := (\forall k, \text{Pstream } xs[k]) \implies G \vdash f(xs) \Downarrow ys \implies (\forall k, \text{Pstream } ys[k])$$

In practice, we can prove that Pnode holds for every node in a program by induction on the list of nodes. Indeed, typing invariants ensure that each node only instantiates the nodes defined earlier in the list.

Concluding the induction on expressions also concludes the induction on blocks, labels, and finally the induction on nodes. The final result is that Pnode is valid for all nodes in the program. In the following sections, we describe two core properties of the semantic model, and how they can be established using the combination of inductions we just described. We outline how this general scheme needs to be adapted to fit these proofs.

3.4 Determinism of the Semantic Model

The semantics of a node is deterministic if, for a given list of input streams xs , only one list of output streams ys is possible. Formally, we can state this as [theorem 2](#), where the equivalence relation on streams \equiv is lifted to lists of streams.

Theorem 2 (Determinism 🐔 [Lustre/LSemDeterminism.v:2722](#))

if `node_causal` $G(f)$
and $G \vdash f(xs) \Downarrow ys_1$ **and** $G \vdash f(xs) \Downarrow ys_2$
then $ys_1 \equiv ys_2$

The theorem applies only to causal nodes. Indeed, as discussed, the equation $x = x$ admits any stream for x , and is obviously not deterministic.

To prove [theorem 2](#), we need to adapt the induction schemes described in the previous section. First, we are concerned with not one semantic model, but a pair of semantic models. Indeed, each of the two semantic hypotheses of the theorem yields a different history. The goal of the proof is to establish that these histories are actually identical. The `Pstream`, `Pvar`, `Pexp`, ... predicates described in the previous sections must therefore be instantiated with predicates that relate two streams, list of streams, histories, ...

The predicate on streams must be preserved by `fbv`, even in the absence of an induction hypothesis for the right operand. If we were to use the equivalence relation on streams \equiv directly, this would translate to:

$$xs_1 \equiv xs_2 \implies \text{fbv } xs_1 \ ys_1 \equiv vs_1 \implies \text{fbv } xs_2 \ ys_2 \equiv vs_2 \implies vs_1 \equiv vs_2$$

which is simply not true: streams vs_1 and vs_2 contain, respectively, elements from ys_1 and ys_2 . If these streams are completely unrelated, we cannot prove that vs_1 and vs_2 are equal.

We solve this problem by introducing an equivalence of two streams *up to* n . Intuitively, $xs \equiv_n ys$ means that the first n values of xs and ys correspond. This equivalence is defined inductively below.

Definition 4 (Stream equivalence up to n 🐔 [CoindStreams.v:310](#))

$$xs \equiv_0 ys \qquad x \cdot xs \equiv_{n+1} y \cdot ys \quad \text{iff} \quad x = y \quad \text{and} \quad xs \equiv_n ys$$

This partial equivalence is implied by the stronger equivalence \equiv . Conversely, quantifying n universally recovers the original equivalence from the partial equivalence.

Lemma 9 (Correspondance of partial and total equivalence 🐔 [CoindStreams.v:335](#))

$$(\forall n, xs \equiv_n ys) \quad \text{iff} \quad xs \equiv ys$$

This new definition suffices to prove [lemma 10](#). It states that the `fbv` relation preserves the equivalence up to $n + 1$ of the left operand, given equivalence up to n of the right operand. This lemma is easily proven by coinduction on the definition of the `fbv` and `fbv1` predicates.

Lemma 10 (Equivalence up to n for `fbv` 🐔 [Lustre/LSemDeterminism.v:734](#))

$$\begin{array}{ll} \text{if} & xs_1 \equiv_{n+1} xs_2 & \text{and} & ys_1 \equiv_n ys_2 \\ \text{and} & \text{fbv } xs_1 \ ys_1 \equiv vs_1 & \text{and} & \text{fbv } xs_2 \ ys_2 \equiv vs_2 \\ \text{then} & vs_1 \equiv_{n+1} vs_2 \end{array}$$

This new equivalence relation, and [lemma 9](#), structure the proof of determinism. We reason (globally) by induction on n , proving that every node is deterministic up to n . The corresponding `Pnode` predicate is therefore defined as follows:

$$\begin{aligned} \text{Pnode } n \ f \ xs_1 \ ys_1 \ xs_2 \ ys_2 & := xs_1 \equiv_n xs_2 \implies \\ & G \vdash f(xs_1) \Downarrow ys_1 \implies G \vdash f(xs_2) \Downarrow ys_2 \implies \\ & ys_1 \equiv_n ys_2 \end{aligned}$$

The proof that this `Pnode` holds for every node in the program proceeds as described in the previous sections. The instrumented semantic model described in [figure 3.11](#) needs to be adapted and takes a pair of histories and base clocks as parameters. The instrumented judgement has the form $G, H_1, H_2, bs_1, bs_2, \text{lord} \vdash_P \text{blk}$. The resulting Coq proof contains a lot of administrative details, but no major surprise. It consists in around 2700 lines of Coq script to state and prove all the invariants necessary to establish [theorem 2](#).

3.5 Clock Correctness

The clock correctness property is a core prerequisite to the proof of compiler correctness. It states that “the sampling of generated stream corresponds with the clock-type annotations”. Formally, we define this property by first defining the semantics of clock-type annotations, as presented in [figure 3.12a](#). Under a given history and base-clock stream, the interpretation of a clock type produces a (boolean) clock stream. The stream produced by the base clock is the base-clock stream of the context. For a sampled clock `ck on C(x)`, the underlying clock `ck` is interpreted, and the resulting stream is sampled by the `when` operator, according to the stream associated to the condition x . This definition corresponds to the semantics of `when` expressions.

For any syntactic element annotated by a clock type, the stream of the clock type should be equal to the clock of the stream produced by the syntactic element. The clock of a stream is given by the `clock-of` function presented in [figure 3.12b](#). Presences are associated to `true` and absences to `false`. The rule for annotated variables presented in [figure 3.12c](#) is an example of the kind of clock correctness judgement we want to express. If variable x is associated to stream vs in the history, we expect the clock-type annotation

$$\begin{array}{c}
\frac{}{H, bs \vdash \bullet \Downarrow bs} \\
\frac{H, bs \vdash ck \Downarrow bs_1 \quad H(x) \equiv vs \quad \text{when}^C bs_1 vs \equiv bs_2}{H, bs \vdash ck \text{ on } C(x) \Downarrow bs_2} \\
\text{(a) sem_clock } \color{red}{\text{🐔 CoindStreams.v:1851}} \\
\text{clock-of } (\langle v \rangle \cdot vs) \triangleq \mathbf{F} \cdot \text{clock-of } vs \quad \frac{H(x) \equiv vs \quad H, bs \vdash ck \Downarrow (\text{clock-of } vs)}{H, bs \vdash_{\text{ck}} x^{ck} \Downarrow vs} \\
\text{clock-of } (\langle v \rangle \cdot vs) \triangleq \mathbf{T} \cdot \text{clock-of } vs \\
\text{(b) abstract_clock } \color{red}{\text{🐔 CoindStreams.v:1777}} \quad \text{(c) sc_var } \color{red}{\text{🐔 Lustre/LClockCorrectness.v:169}}
\end{array}$$

Figure 3.12: Semantics of clock-type annotations

ck associated to x to produce stream $\text{clock-of } vs$. The same idea applies to expressions and node instantiations.

As for semantic determinism, our proof of this property assumes the absence of dependency cycles. Indeed, the equation $x = x$ admits any stream, including streams that do not correspond to the declared clock type of x . The clock correctness theorem for nodes, stated in [theorem 3](#) below, only holds for causal, well-clocked nodes. It states that, if such a node has a semantics relating input streams xss to output streams yss , and if the clock streams of xss correspond to the declared clocks of the inputs of the node, then the clock streams of yss correspond to the declared clocks of its outputs. The semantics of the clock-type annotations of inputs and outputs may be given in any history H that associates the correct streams to the inputs and outputs of the node. Indeed, the clock-type annotations of the outputs only depend (syntactically) on input and output variables of the node.

Theorem 3 (Clock Correctness [🐔 Lustre/LClockCorrectness.v:2610](#))

if $G \vdash_{\text{wc}} f$ **and** $\text{node_causal } G(f)$
and $G \vdash f(xss) \Downarrow yss$
and $G(f) = \text{node } f([x_i : ck_i]^i) \text{ returns } ([y_j : ck'_j]^j) \text{ blk}$
and $\forall i, H, (\text{base-of } xss) \vdash_{\text{ck}} x_i^{ck_i} \Downarrow xss_i$ **and** $\forall j, H(y_j) \equiv yss_j$
then $\forall j, H, (\text{base-of } xss) \vdash_{\text{ck}} y_j^{ck'_j} \Downarrow yss_j$

The proof of this theorem is more straightforward than is the proof of determinism. Indeed, this time, `fby` does preserve the clock-correctness property: [lemma 11](#) states that the clock-stream of the stream produced by `fby` is the same as the clock stream of its left operand. This is a direct consequence of the definition of the `fby`: in essence, the operator “forces” its two operands to have the same clock, which means it is not necessary to check the clock of the right operand. This idea was first exploited in the first version of the clock-correctness proof for the core dataflow language [[Jea19](#)]. We apply this result

to prove the `fbby` case of [lemma 8](#). The clock-typing rule for `fbby` states that the k -th clock type of `e0 fbby e1` is equal to the k -th clock-type annotation of `e0`. Using this fact, and the lemma described above, we can prove that, if the k -th stream produced by `e0` corresponds to the interpretation of the k -th clock-type annotation of `e0`, then this is also the case for the k -th stream produced by `e0 fbby e1`.

Lemma 11 (`fbby` preserves clock streams 🐔 [Lustre/LSemantics.v:2235](#))

if `fbby xs ys ≡ vs` **then** `clock-of vs ≡ clock-of xs`

Most other inductive cases of [lemma 8](#) are proven by a similar reasoning: we establish that the syntactic relation between clock-type annotations induced by clock-typing rules correctly reflects the relations induced by semantic stream operators.

The most difficult case is for node instantiations, because of the node subsampling dependencies described in [section 2.4.4](#). The proof of this case proceeds as follows: consider the expression `f(es)`. By the induction hypothesis on expressions, we know that all the streams `xss` of the argument expressions `es` have the correct clock stream. We also know, by induction on the list of program nodes, that node `f` preserves clock correctness, as stated in [theorem 3](#). To use this hypothesis, we have to prove that the input streams `xss` correspond to the declared input clock types of `f` (fourth premise of the theorem). This is not trivial, because the clock-type annotations of expressions `es` are not the same as the input clock types (indeed, they are instantiated, see the clock-typing rules for node instantiations in [figure 2.12c](#)). The same difficulty arises when proving that the output streams `yss` of the node, which we know correspond to the declared output clock types of the node (conclusion of the theorem), also correspond to the clock types of the node instantiation. These proofs are even more complicated when dealing with a resettable node, because the `mask` affects sampling.

Despite these complications, the rest of the proof proceeds as expected, and [theorem 3](#) can be proven without any other surprise. In the following chapter, we will see how this result is crucial in proving the correctness of compilation, and how it affects the overall “proof architecture” of the compiler.

3.6 Discussion and Related Work

3.6.1 Causality Type Systems for Dataflow Languages

The dependency analysis implemented in `Vélus` has a major limitation: it considers node instantiations as atomic, where all inputs have to be determined before output can be computed. This requirement does not stem from the semantic model. Indeed, consider the program in [figure 3.13](#). It is perfectly deterministic: its output is always equal to its input. However, this program would be refused by our compiler because, syntactically, the `use_plumbing` node contains a cycle: `b` depends immediately on itself.

To accept such a program, we would need two adaptations. The first is a modular causality analysis that can express the finer grained dependencies between node inputs

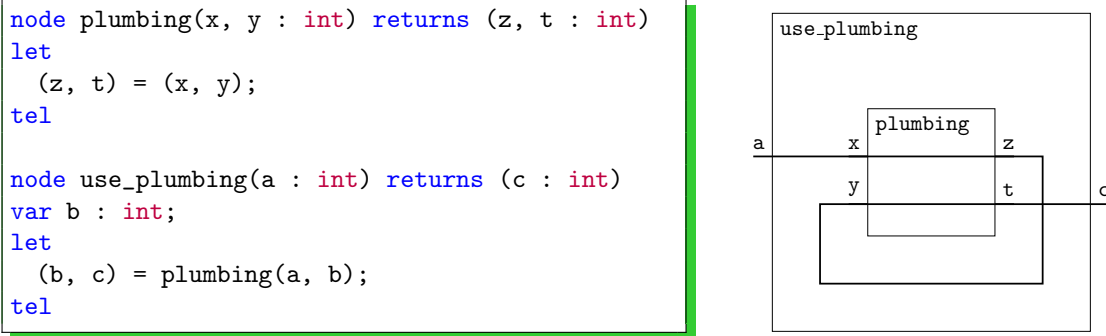


Figure 3.13: A node with a dependency cycle

and outputs. This problem has been attacked by introducing type-based modular analyses. In such a system, the `plumbing` node would have type $\forall \alpha_1 \alpha_2, (\alpha_1 \times \alpha_2) \rightarrow (\alpha_1 \times \alpha_2)$. There are several ways to encode the dependencies between variables using types. [CP01] uses row-polymorphism. This approach was first implemented in Lucid Synchronic. Each expression is given a “causality type” ϕ with rows indicating the variables on which the expression depends. For instance, in the `plumbing` node, the type scheme given above would be instantiated with $\alpha_1 = \{a : p; b : a; \rho_1\}$ and $\alpha_2 = \{a : p; b : p; c : a; \rho_2\}$, where p indicates that a variable is used (present), a indicates that a variable is not used (absent), and row-variables ρ allow the type to be further specialized. This instantiation is well typed, because (i) a is present in α_1 and b is present in α_2 : this respects the variable rule, and (ii) b is absent in α_1 and c is absent in α_2 : this respects the equation rule. Another approach is presented in [HSCC14]. A label representing a *date* is associated to each syntactic element in the program. These labels abstract away from the named variables; this was the inspiration behind the labels in Vélus. Each equation induces precedence constraints between labels. In `use_plumbing`, if α_a, α_b and α_c are the labels for `a`, `b` and `c` respectively, then the type scheme is instantiated with $\alpha_1 = \alpha_a$ and $\alpha_2 = \alpha_b$. The equation induces the precedence constraint $\alpha_a < \alpha_b$ and $\alpha_b < \alpha_c$. These constraints are valid because they define a strict partial order. This system is implemented in Zélus [Pou10], and has also inspired the system used in Scade 6 [CPP17].

The second adaptation required to support richer node instantiations is to the compilation scheme. Indeed, the compilation of nodes implemented in Vélus is strictly modular [Bie+08]: each node is compiled into a single imperative step function. With this technique, the node `use_plumbing` cannot be scheduled, as `y` needs to be calculated before the call. Two solutions exist to compile such programs: the simplest is to inline the definitions of nodes that display such cycles; this is done in Scade 6. This results in schedulable code, and may, coupled with other optimizations like constant propagation, generate more efficient code. However, this approach can also lead to code duplication. An alternative approach, proposed in [PR09], is to generate not one, but several step functions for a node. The calls to these step functions may then be scheduled appropriately, depending on the requirements of the caller. Decomposing of a node into the maximal

number of step functions is, in the general case, NP-hard. A simplified algorithm that works for most real-world program is proposed. When it is not sufficient, an external solver may be needed.

For now, Vélus does not implement any of these ideas, and the program of [figure 3.13](#) cannot be scheduled. The dependency analysis that we have described in this chapter reflects this constraint. We believe that implementing and verifying on-demand inlining of nodes in Vélus would not be too difficult. However, we do not know how a type-based causality analysis would impact our proofs of determinism and clock correctness.

3.6.2 Verified Graph Analysis

A depth-first search graph analysis was previously formalized in Coq [[Pot15](#)] as part of an algorithm for calculating strongly-connected components. The second part of the paper proposes a specification of the depth first search algorithm. Its termination is specified using dependant types. The algorithm program itself is not given, as it is written directly in the tactic language of Coq. The author states that using [Program](#), as we proposed in this dissertation, may make for a more readable code. Finally, two possible improvements are proposed that also apply to our work:

- Our algorithm is formulated using concrete datatypes for vertices, sets of vertices, and graphs (`ident`, `PS.t`, `Env.t`). We could use a more generic formulation with abstract types, which would allow for a more reusable algorithm.
- The function we implement is not tail-recursive, therefore the extracted OCaml function is not either. On large graphs with “deep” dependencies, the execution of this implementation may lead to a stack overflow. In practice, this has not yet happened, but we have not tested our compiler on any large, real-world industrial program. A way to mitigate this issue would be to implement a tail-recursive function using an explicit stack, but the specification of such a function would be more intricate.

Front-End Compilation

Now that we have introduced the specification of the Vélus front-end language, and some of its core properties, we can start discussing how the compiler is implemented and proved correct. This chapter is dedicated to the front-end of the compiler. The front-end first parses and elaborates the source program. Then, it successively simplifies each of the control blocks through a series of source-to-source transformation passes. This pipeline is summarised in [figure 4.1](#). The following sections describe these passes. [Appendix B](#) shows the effect of each successive pass on the `drive_sequence` node presented in the introduction.

4.1 Parsing of Source Programs

Vélus includes a parser that transforms the content of a text file into an AST. Like in CompCert, the parser is implemented using the Coq back-end of the Menhir parser generator [\[PR16\]](#). Menhir takes as input a formal grammar of the language to be parsed, where each derivation rule is associated with a “semantic action”, that is, a Coq expression used to combine the parsed sub-terms of the rule. Menhir then generates an LR(1) automaton corresponding to the grammar. This automaton is passed to an interpreter implemented in Coq that takes as input a stream of tokens from a textual program and produces an AST, or fails in case of a syntax error. The Coq port of Menhir is validated, using a proof-carrying code approach [\[JPL12\]](#). Along with the automaton, the parser

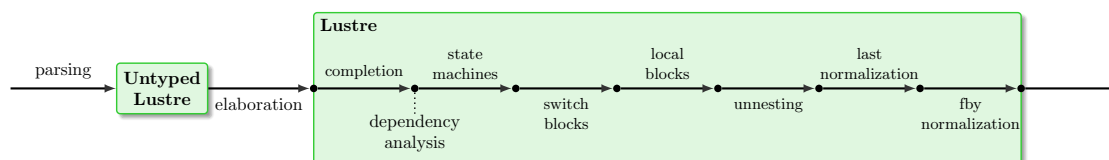


Figure 4.1: Architecture of the Vélus front-end

generates a certificate that is checked by a validator implemented and verified in Coq. The validator ensures three properties of the automaton:

- Correctness: only programs described by the grammar are accepted.
- Safety: interpretation of the automaton does not raise an internal error.
- Completeness: a sequence of tokens that adheres to the grammar is parsed.

The first property holds, by construction, for any input grammar. The latter two only hold for unambiguous grammars. While Menhir only normally signals conflicts in the grammar as warnings, this validator forces us to make the grammar unambiguous.

Using the Coq back-end also comes with some limitations. Precedence and associativity declarations cannot be used to solve conflicts: the grammar has to be LR(1) without annotations. The semantic type of every nonterminal symbol must be explicitly specified. Parameterized nonterminals, such as `list(t)` or `separated_list(sep, t)` are not available. Without this “polymorphism”, we need to explicitly define lists of expressions, lists of blocks, lists of nodes, etc. This makes the grammar a bit cumbersome to define, especially for the syntax of expressions, and their precedence rules. The complete grammar for the language is therefore expressed as 600 lines of code including semantic actions.

The parser generates a term in the Untyped Lustre AST. This AST differs from the one presented in [section 2.1.1](#) in a few ways. The most important is that expressions are not annotated by their types and clock types; the elaboration pass described below adds these annotations. Enumerated constructors are represented by their names (appearing in the program) and not by their tag (the number of the constructor in the type). Finally, every term is annotated by location information, which allows for the printing of precise error messages during elaboration.

4.2 Generating Fresh Identifiers

Most of the compilation algorithms used in the front-end require the introduction of new variables. These variables are used at left of new equations introduced by the compiler. For instance, during the normalization pass, the equation `f = 0 fby (f + (1 fby f))` is compiled to `f = 0 fby (f + f$1); f$1 = 1 fby f`. Here, `f$1` is a new identifier introduced by the compiler. For the compiler to be correct, each of the newly introduced identifiers must be unique. In the context of a compiler implemented in a language with side-effects, this would be easy to implement: we could increment a global counter every time an identifier is generated, and use its value in the name of the identifier. However, Coq is a pure functional language, so using side-effects is not possible.

One possible solution is to *axiomatize* a function, that is, suppose its existence without defining it in Coq. At extraction, this axiomatized function may then be defined by an OCaml expression, using the whole language, including its imperative features. However, just using axiomatized functions is not sufficient. Indeed, to prove properties of the compiler, we need to reason about the identifiers being generated: specifically, we need to

know that they are distinct from previously generated identifiers. One solution would be to axiomatize, in Coq, some properties of the OCaml function. Of course, one needs to be extremely careful not to introduce a false property as an axiom; to reduce this risk, these properties must be as simple and obvious as possible. In the following, we describe how we tackled this issue in Vélus, by axiomatizing a simple `gensym` function written in OCaml, and then by building more complex identifier-generation functions and reasoning using a monadic approach.

4.2.1 Gensym Axiomatization

Identifiers are represented as positive integers in the Coq formalization of both CompCert and Vélus. However, these identifiers also need to be related to a textual representation for printing error messages and intermediate programs. CompCert implements this relation using two hash tables at the OCaml level. These tables associate OCaml strings with identifiers and vice-versa. For Vélus to generate identifiers consistently with CompCert, we need to take into account these tables. We now describe this OCaml code, and how our Coq formalization interacts with it.

The hash tables are manipulated through the `intern_string` and `extern_positive` functions, reproduced in [listing 4.1](#). The first takes an OCaml string, and tries to recover the associated positive integer from the table. If the string is not in the table, a fresh positive integer is generated, using a global counter stored in a mutable reference. An association between the string and the new positive integer is added in both tables, and the new positive integer is returned. The second takes as input a positive integer, and returns the string associated with it. If it does not exist, it simply returns a string containing the value of the positive integer, preceded by a `$` character. Note that these two functions are not symmetrical: only the first one may introduce new associations in the tables. Functions `str_to_pos` and `pos_to_str` simply compose these functions with translations between OCaml and Coq strings.

In Vélus, we axiomatize these last two functions in Coq, as shown in [listing 4.2](#). This allows us to (i) insert new strings in the table, from the Coq code, and (ii) define some predicates on the textual name of the identifier. For instance, the `atom` predicate ensures that the string associated with an identifier does not contain the `$` character.

We assume only two properties of these functions. First, `str_to_pos` should be injective. This is true of the OCaml implementation if (i) `camlstring_of_coqstring` is injective, and (ii) `intern_string` is injective. The second point is true as long as the `pos_of_string` table is itself injective, which is true if it has only been manipulated through the `intern_string` function. The second property is that calling `pos_to_str` after `str_to_pos` is the identity function. This is true because (i) composing `camlstring_of_coqstring` and `coqstring_of_camlstring` gives the identity function, and (ii) calling `extern_positive` after `intern_string` necessarily returns the same string, even if it was just inserted.

There is an issue with `pos_to_str`: it is not a mathematical function, in the sense that two calls to `pos_to_str` with the same arguments may return different values. Consider, for instance, the sequence of calls `pos_to_str 2`, `str_to_pos "hello"`, `pos_to_str 2`.

```

let pos_of_string = (Hashtbl.create 17 : (string, positive) Hashtbl.t)
let string_of_pos = (Hashtbl.create 17 : (positive, string) Hashtbl.t)

let next_positive = ref Coq_xH
let fresh_positive () =
  let p = !next_positive in
  next_positive := Pos.succ !next_positive;
  p

let intern_string s =
  try Hashtbl.find pos_of_string s
  with Not_found ->
    let p = fresh_positive () in
    Hashtbl.add pos_of_string s p;
    Hashtbl.add string_of_pos p s;
    p

let extern_positive p =
  try Hashtbl.find string_of_pos p
  with Not_found ->
    Printf.sprintf "$%d" (to_int p)

let str_to_pos str = intern_string (camlstring_of_coqstring str)
let pos_to_str pos = coqstring_of_camlstring (extern_positive pos)

```

Listing 4.1: Manipulating identifier tables [CompCert]

The first call will return "\$2" if the tables start empty. However, if the call to `str_to_pos` creates the association $2 \leftrightarrow \text{"hello"}$, then the last call will return "hello". This is an issue because Coq assumes that all functions are pure, which allows for substituting syntactically equal terms. Here, this is not necessarily the case. However, this does not introduce bugs in Vélus: `pos_to_str` is only used to define the `atom` predicate, and the equality between two calls to `pos_to_str` or `atom` is never tested by the compiler or reasoned about in a proof.

Nonetheless, manipulating axiomatized functions is risky. The explanations as to why our axioms are correct lack the rigor of mechanized proofs. One must be particularly careful and limit interactions with axiomatized functions and the introduction of new axioms. Earlier versions of Vélus stated two wrong axioms on these functions.

First, `pos_to_str` is *not* injective. Consider the sequence of calls `pos_to_str 42`, `str_to_pos "$42"`, `pos_to_str 2`, started with empty hash tables and the global counter set to 1. The first call returns "\$42", because of the exception in `extern_positive`. The second associates "\$42" and 2 in both tables. This means that the last call returns "\$42", even if its argument is different from the first call.

Second, calling `str_to_pos` on the result `pos_to_str` is *not* necessarily the identity function. Consider the call `str_to_pos (pos_to_str 42)`, with a pair of tables where the only association is $2 \rightarrow \text{"$42"}$. The first call returns "\$42", following the exceptional behavior of `extern_positive`. However, the second call returns 2, because

```

Axiom str_to_pos: string -> ident.
Axiom pos_to_str: ident -> string.

Axiom str_to_pos_injective: ∀ x x',
  str_to_pos x = str_to_pos x' ->
  x = x'.
Axiom pos_to_str_equiv: ∀ x,
  pos_to_str (str_to_pos x) = x.

[...]
Definition local := str_to_pos "local".
Definition norm1 := str_to_pos "norm1".
[...]

Definition atom x := ~In_str "$" (pos_to_str x).

```

Listing 4.2: Conversion between positive integers and strings 🐔 Ident.v:10

the association already exists.

Using these erroneous axioms in the formalization eventually led to bugs in our first approach. In practice, there were some conflicts in the names used in the table, which meant that the generated assembly code did not always use the correct names for function symbols. We discovered this problem when the assembler `ld` refused the generated assembly code. To solve this problem, we removed the two incorrect axioms, and reworked identifier generation using a new `gensym` function.

This function is implemented in OCaml as shown in [listing 4.3](#). It takes three parameters. The first is a prefix string which will appear at the start of the new string. It should always be an atom (that is, it should never contain the `$` character). If not, the compiler aborts; in practice, we always use this function properly. The second is optional, and serves to add some text in the string hinting at the origin of the new identifier. If it is provided, it appears in the middle of the string. The third is a positive integer. The text of the number itself is placed at the end of the string. These three “segments” of the new strings are separated by `$` characters. The new string is inserted in the table using the `intern_string` function, and a new positive integer is returned.

```

let gensym pref hint x =
  let pre = extern_atom pref in
  if String.contains pre '$' then invalid_arg "gensym";
  match hint with
  | None -> intern_string (pre^"$"~string_of_int (to_int x))
  | Some hint ->
    intern_string (pre^"$"~extern_atom hint^"$"~string_of_int (to_int x))

```

Listing 4.3: `gensym` in OCaml


This function was designed so that the axioms presented in [listing 4.4](#) hold. First, the

produced identifier should never be an atom. This is obviously true: the string associated with the identifier contains at least one \$ character. Second, `gensym` is (weakly) injective, in the sense that, if at least one of the `pref` and positive integer parameters is changed, then the result is changed. This is true, because the resulting string is always of the form "`pref$x`" or "`pref$hint$x`". Since neither `pref` nor `x` contain a \$ character, the decomposition of such a string is unique. We do not have to assume that `pref` is an atom in the statement of the axiom, because the compiler will abort if it is not. This simplifies proofs.

```
Axiom gensym : ident -> option ident -> ident -> ident.

Axiom gensym_not_atom: ∀ pref hint x,
  ~atom (gensym pref hint x).

Axiom gensym_injective': ∀ pref hint id pref' hint' id',
  pref <> pref' \ / id <> id' ->
  gensym pref hint id <> gensym pref' hint' id'.
```

Listing 4.4: `gensym` in Coq  [Ident.v:387](#)

These axioms are sufficient to ensure the correctness of identifier generation. We now describe the more general mechanism and reasoning that exploits these definitions to generate identifiers through each pass of the compiler.

4.2.2 The Fresh Monad

As we stated in the introduction to this chapter, the Vélus front-end is structured as a series of passes that rewrite programs into smaller and smaller subsets of the source language. Most of these passes may generate new identifiers. For each pass, we need to prove that the identifiers introduced during the pass do not interfere with the identifiers introduced in previous passes, or with the ones appearing in the source node.

Our approach is the following: each pass introduces identifiers using the `gensym` function. The prefix `pref` parameter passed to `gensym` is unique to each pass. The postfix number is generated through a counter that is local to the pass. Informally, we know that since each pass has its own prefix, then the identifiers generated between passes do not interfere with one another. In Coq, this is mechanized by adding an `n_good` field to the node `Record`, as presented in [listing 4.5](#). It specifies that (i) the name of the node is an atom (this is useful when eventually generating C code), and (ii) that (2) the name of global and local variables are either atoms or generated using `gensym` with a prefix in a given set `prefs`. The set of prefixes is given as a parameter of the `node` type. This parameter varies during compilation: intuitively, after each compilation pass, one new prefix is added to the set. Since `n_good` is used pervasively to prove the preservation of other node invariants, it is easier to include it directly in the type of nodes rather than to have it as an external predicate.

```

Definition AtomOrGensym (prefs : PS.t) (id : ident) :=
  atom id \ / PS.Exists (fun p => exists n hint, id = gensym p hint n) prefs.

Record node {prefs : PS.t} : Type :=
mk_node {
  n_name : ident;
  n_in : list (ident * (type * clock * ident));
  n_out : list decl;
  n_block : block;
  [...]
  n_good : Forall (AtomOrGensym prefs) (map fst n_in ++ map fst n_out)
    /\ GoodLocals prefs n_block
    /\ atom n_name;
}.

```

Listing 4.5: Invariant specifying the well-formedness of identifiers 🐔 [Lustre/LSyntax.v:497](#)

To ensure non-interference during a pass, we reason about the counter that provides the second parameter of `gensym`. To do so, we use a monadic approach. The `Fresh` monad described in [listing 4.6](#) is essentially a specialization of the state monad [[Wad92](#)].

The state of identifier generation is captured by the `fresh_st` type. First, it contains the counter `st_next`. Second, the list `st_anns` keeps track of the generated identifiers, along with some optional annotation (of type `B`) for each identifier. Two invariants should always be preserved on states. First, all generated identifiers must be distinct (`st_nodup`). Second, they must all have been created by calling `gensym` with a postfix that was inferior to the current value of the counter. This allows us to prove that future generated identifiers are also distinct from the existing ones. In our development, the definition of type `fresh_st` is abstracted under an opaque module. The module provides operations and lemmas on this abstracted type that is sufficient to reason on states.

A `Fresh` value is simply a function that takes a state as input, and returns a value and an updated state. As usual for monads, the `ret` function wraps a pure value in `Fresh`, and `bind` sequences two monadic operations. Contrary to the `fresh_st` type, the `Fresh` type is left transparent to the user. This way, monadic expressions appearing in the context of a proof may be reduced, which facilitates mechanized reasoning.

The operation that actually generates names and modifies the state is the `fresh_ident` function. It has two implicit parameters: the prefix `pref` and type of annotations `B`. Its inputs are the optional `hint`, and the annotation `b` to be associated with the new identifier. The returned monadic value is a function that takes a state as input, calls `gensym` with the prefix, `hint` and current counter. The resulting identifier is added to the state, and the counter is incremented. We omit from the listing the proofs that the `st_nodup` and `st_prefs` are valid for the new state. Both are straightforward, and rely on the fact that these properties are true of the predecessor state. The proof of `st_nodup` also depends on the axiomatized injectivity of `gensym`. Like the definition of `fresh_st`, `fresh_ident` is abstracted, and only a few well-chosen properties of its execution are exposed.

```

Record fresh_st (pref : ident) (B : Type) : Type :=
{ st_next : ident;
  st_anns : list (ident * B);
  st_nodup : NoDupMembers st_anns;
  st_prefs : Forall (fun id => exists x hint, id = gensym pref hint x
                    /\ Pos.lt x st_next)
              (map fst st_anns)
}.

Definition Fresh pref (A B : Type) : Type :=
  fresh_st pref B -> A * fresh_st pref B.
Definition ret (a : A) : Fresh pref A B := fun st => (a, st).
Definition bind (x : Fresh pref A B) (k : A -> Fresh pref A' B)
  : Fresh pref A' B := fun st => let '(a, st') := x st in k a st'.

Program Definition fresh_ident {B} {pref} hint (b : B) : Fresh pref ident B :=
  fun st =>
    let id := gensym pref hint (st_next _ _ st) in
    (id, {| st_next := Pos.succ (st_next _ _ st);
          st_anns := (id, b)::st_anns st |}).

Definition st_follows (st st' : fresh_st pref B) :=
  (st_anns st) ⊆ (st_anns st').

```

Listing 4.6: Fresh Monad definitions 🐔 Fresh.v:31

Finally, we establish an order relation on states: `st_follows st st'` states that all identifiers generated in `st` also appear in `st'`. This is true if `st'` is the result of a sequence of applications of `fresh_ident` on `st`; we say that `st'` *follows* `st`. This property facilitates reasoning by inclusion, in particular in proofs of type and clock-type preservation.

We use a tactic to simplify `bind` expressions. If the proof context contains the expression `bind v1 f st1 = (v3, st3)`, then the tactic introduces intermediate values and the following equalities: `v1 st1 = (v2, st2)` and `f v2 st2 = (v3, st3)`. This automatically decomposes monadic functions, which facilitates inductive reasoning. Overall, we believe that the method we adopted is the most suited to our needs, and that the difficulties that sometimes arise are inevitable for a realistic verified compiler.

4.3 Elaboration of Lustre Programs

We now discuss the elaboration which adds type and clock-type annotations to the parsed AST. The elaborator may fail for programs that are not well formed. It is therefore implemented as a Coq function that returns a value in the error monad (`OK/Error`).

In the following, we focus on how the elaborator infers clock-type annotations for untyped expressions. This is not as easy as adding the type annotations, which can be checked directly. We also discuss the verification of this elaborator.

4.3.1 Clock-Type Elaboration by Monadic Unification

Inferring clock-type annotations for a Lustre program is not trivial, even if all variables are already declared with their clocks. This is due to the interaction of two features that increase the expressivity and ease of use of the language. First, the node subsampling described in [section 2.4.4](#). Second, the fact that `whens` for sampling constants may be added implicitly. Consider the example in [listing 4.7](#). In the first node, `f`, it is easy to infer that the constant `0` should be sampled on the clock `• on false(b)`; it is a direct consequence of the `merge` clock-typing rule. The elaborator replaces this constant with `0 when false(b)`. Inferring the clock type of the `4` constant in node `g` is not so simple. Indeed, it depends on the clock-typing rule for node application. By considering the clock type of `f`, and the fact that the first argument passed to `f` has clock type `• on true(b1)`, we can deduce that this constant should have clock type `• on true(b1) on true(b2)`, and replace the constant by `4 when true(b1) when true(b2)`.

```
node f(b : bool; x : int when b) returns (y : int)
let
  y = merge b (true => x) (false => 0);
tel

node g(b1 : bool; b2 : bool when b1) returns (z : int when b1)
let
  z = f(b2, 4);
tel
```

Listing 4.7: Lustre node that requires general clock-type inference

More generally, a unification-based algorithm can be used to infer the clock types of expressions in a program [[CP03](#)]. We adopt a monadic approach by defining an ad-hoc composition of the state and error monads that we call the elaboration monad. The type `sclock` shown in [listing 4.8](#) is used to represent a clock that may contain an unresolved clock-type variable. The state of the monad contains a counter used to generate fresh clock-type variables through `gensym`. We do not use the `Fresh` monad described in the previous section, because we do not need to keep track of the list of generated identifiers in this pass. The state also contains a substitution that associates existing clock-type variables to more precise clock types. This substitution is enriched every time two clock types are unified. A monadic value of type `Elab A` takes as input a state and returns either a value of type `A` and a new state, or an error. The definitions of the `ret` and `bind` functions for `Elab A` are unsurprising; we do not reproduce them here. For brevity, we do not present the definitions of the substitution and unification functions either; they are straightforward.

The elaboration of expressions occurs over two passes. The first function, `elab_exp`, takes as input an untyped expression and constructs an “elaboration expression” of type `eexp`. This intermediate type is identical to the type of expressions presented in [section 2.1.1](#), except (i) the clock-type annotations in the AST use the `sclock` type, and

```

Inductive sclock :=
| Sbase : sclock
| Son : sclock -> ident -> (type * enumtag) -> sclock
| Svar : ident -> sclock.

Definition elab_state : Type := (ident * Env.t sclock).

Inductive res (A: Type) : Type :=
| OK : A -> res A
| Error : errmsg -> res A.

Definition Elab A := elab_state -> res (A * elab_state).

```

Listing 4.8: Elaboration Monad 🐔 [Lustre/LustreElab.v:242](#)

```

Fixpoint elab_exp (ae: expression) {struct ae} : Elab (eexp * astloc) :=
  match ae with
  | CONSTANT ac loc =>
    do x <- fresh_ident;
    do c <- elab_constant loc ac;
    ret (Econst c (Svar x), loc)
  [...]
  | APP f aes loc =>
    (* elaborate arguments *)
    do eas <- mmap elab_exp aes;
    (* instantiate node interface *)
    do (tyck_in, tyck_out) <- find_node_interface loc f;
    let nanns := lnanns eas in
    let sub := instantiating_sub tyck_in nanns in
    do xbase <- fresh_ident;
    do ianns <- mmap (inst_annot loc (Svar xbase) sub) tyck_in;
    do oanns <- mmap (inst_annot loc (Svar xbase) sub) tyck_out;
    do _ <- unify_params loc ianns nanns;
    ret (Eapp f (map fst eas) oanns, loc)
  end

```

Listing 4.9: Elaborating an expression 🐔 [Lustre/LustreElab.v:871](#)

(ii) constants are also annotated with an `sclock`. For constants, the clock-type annotation introduced initially is always `Svar`, because we do not know if implicit `whens` must be added. The other interesting case is that of a node instantiation, where the base clock of the instantiation is first introduced as a clock-type variable. The `instantiating_sub` function builds a substitution `sub` of parameter names to argument names; the argument names can only be the names of “named clocks” as described in [section 2.4.4](#). The resulting substitution is then used to instantiate the input and output parameters of the node. If one of the clocks of these parameters depends on a variable that is not in `sub`, then `inst_annot` returns an error: this corresponds to having a dependency on an anonymous input or on an output, which is not allowed. Finally, the instantiated parameters are

unified with the annotations of the arguments. The case of a node instantiation appearing directly at right of an equation, which allows for dependencies between outputs, is similar.

The second function, `freeze_exp`, is partially shown in [listing 4.10](#). It takes as input an elaboration expression, and “freezes” its annotations: that is, it turns unification clocks into fixed clocks without clock-type variables. This is accomplished by function `freeze_clock`, which first propagates the current substitution through an `sclock`, and then converts it to the `clock` type. The interesting case of `freeze_exp` is the one for constants, where the implicit sampling must be added depending on the inferred clock of the constant. The recursive function `add_whens` traverses the clock-type annotation, and adds `when` for each `Con` constructor.

```

Fixpoint sclk' (ck : sclock) : clock :=
  match ck with
  | Sbase | Svar _ => Cbase
  | Son ck' x b => Con (sclk' ck') x b
  end.

Definition freeze_clock (sck : sclock) : Elab clock :=
  do sck <- subst_sclock sck;
  ret (sclk' sck).

Fixpoint add_whens (e : Syn.exp) (tys: list type) (ck: clock) : Elab Syn.exp :=
  match ck with
  | Cbase => ret e
  | Con ck' x (tx, k) =>
    do e' <- add_whens e tys ck';
    if Env.mem x env then ret (Syn.Ewhen [e'] (x, tx) k (tys, ck)) else error [...]
  end.

Fixpoint freeze_exp (e : eexp) : Elab Syn.exp :=
  match e with
  | Econst c ck =>
    let ty := ctype_cconst c in
    do ck' <- freeze_clock ck;
    add_whens (Syn.Econst c) [Tprimitive ty] ck'
    [...]
  end.

```

Listing 4.10: Freezing an expression 🐔 [Lustre/LustreElab.v:1014](#)

When elaborating an equation, as presented in [listing 4.11](#), the expressions at right of the equation are first elaborated using `elab_exp`. The resulting annotations are then unified with the declared annotations of the variables at left of the equation. Finally, the expressions are frozen using `freeze_exp`. The elaboration of blocks is defined as a single pass that traverses the blocks and elaborates each individual sub-equation or sub-expression in the block.

```

Definition elab_equation (aeq : LustreAst.equation) : Elab Syn.equation :=
  let '(xs, es), loc) := aeq in
  do es' <- mmap elab_exp es;
  do _ <- unify_pat loc xs (lannots es');
  do es' <- mmap freeze_exp (map fst es');
  ret (xs, es')
end.

```

Listing 4.11: Elaborating an equation 🐔 [Lustre/LustreElab.v:1114](#)

4.3.2 Translation Validation of the Elaboration

What does it mean for this elaborator to be correct? First, the elaborated program should “correspond” semantically to the untyped program. Second, the elaborated program should be well typed and well clocked, according to the rules outlined in [chapter 2](#) and presented completely in [appendix A](#).

The first point cannot be formally verified, as there is no semantic model for untyped programs. However, the elaborator is defined as a simple recursive function that constructs type terms of the same form as the source untyped terms. Therefore, the correctness of the pass can be easily checked by inspection of the elaborator’s source.

The second point could be proven directly as a theorem that states that, if the elaboration returns an `OK` value, then the term returned is well typed and well clocked. Earlier versions of Vélus took this approach. At the time, the clock-type elaboration algorithm was simpler than the one described here, but the proof was still quite intricate. In the generalized monadic approach, the proof would be even more complex, so we decided to abandon direct verification.

Instead, we rely on translation validation. Two decision procedures are applied to the elaborated term to check that it is well typed and well clocked. If both return `true`, compilation continues; otherwise, the compiler aborts with an error message. Both of these procedures are simpler than the elaboration function, and it is easy to prove their correctness relative to the inductive definitions of the type and clock-type systems. We expect that, for an elaborated term, these functions never return `false`. If they did, it would mean that either the elaborator adds incorrect annotations, or that it is missing some checks.

In addition to typing and clock-typing, we need to validate the other static node invariants used to prove compilation correctness. Indeed, as we stated in [section 2.1.1](#), these invariants, listed in [appendix A.1](#) are “stored” in fields of the dependent `Record` node. This means that we cannot build a value of type `node` without first proving that they hold. Each invariant is stated as an inductive relation between the inputs, outputs, and body of a node. It is checked by a separate procedure that may return error messages and is relatively easy to verify.

$$\begin{aligned}
e &::= c \quad | \quad C \quad | \quad x \quad | \quad \text{last } x \quad | \quad \diamond e \quad | \quad e \oplus e \quad | \quad e \text{ when } C (x) \\
ce &::= \text{merge } x (C \Rightarrow ce)^+ \quad | \quad \text{case } e \text{ of } (C \Rightarrow ce)^+ \quad | \quad e \\
sc &::= c \quad | \quad C \quad | \quad sc \text{ when } C (x) \\
blk &::= x = ce \\
&\quad | \quad x = sc \text{ fby } e \\
&\quad | \quad x^+ = f (e^+) \quad | \quad x^+ = (\text{reset } f \text{ every } x) (e^+) \\
&\quad | \quad \text{last } x = sc \\
&\quad | \quad \text{reset } blk \text{ every } x \\
var &::= x : ty \text{ on } ck \\
nodedecl &::= \text{node } f (var^+) \text{ returns } (var^+) \text{ var } var^* \text{ let } blk^+ \text{ tel}
\end{aligned}$$
Figure 4.2: Restricted syntax of normalized Lustre 🐔 [Lustre/LSyntax.v:1500](#)

4.4 Structure of the source-to-source rewriting passes

We now move on to the source-to-source rewriting passes that transform a general Lustre program into one that only uses a subset of the language. We first introduce some general ideas behind all of the passes, before describing each pass in more detail.

4.4.1 Normalized subset of the language

We first describe the normalized subset of the language that the source-to-source rewriting passes target. The right subset should be restricted enough to simplify the middle-end compilation passes, but expressive enough to define optimizations and produce efficient code. We present the restricted syntax in [figure 4.2](#). The language is structured with two tiers of expressions: simple expressions, that only contain constants, variables, arithmetic and subsampling operators, and control expressions which may contain conditional operators ([merge](#) and [case](#)). Finally, stateful constructs ([fby](#), node instantiations) may only appear directly under an equation. Note also that all operators have been distributed over their arguments. There can only be one expression under any [when](#), branch of a [merge](#), etc. These transformations are treated by the Unnesting pass described in [section 4.10](#). The restricted language does not contain state machines ([section 4.7](#)) or [switch](#) blocks ([section 4.8](#)). In addition, local declarations can only appear at top of the block; their flattening is described in [section 4.9](#).

4.4.1.1 Reset Blocks

The normalized language still contains (possibly nested) [reset](#) blocks. Indeed, we have found that it is not possible to efficiently compile these resets into other operations of the Lustre language. Previous work [[CPP05](#)] suggests using conditionals to

compile the `reset` of a `fb`. For instance, `reset x = e0 fb e1 every r` would become `x = if r then e0 else (e0 fb e1)`. However, recall that the clock of a `reset` condition is independent from the clock of the underlying equations. This means that even if the source program is well clocked, a program compiled this way may not be. If `r` has a slower clock than `x`, this is not a problem: `r` can be merged with `false` to get a well-clocked and semantically equivalent program. If `r` is faster, however, we cannot simply sample it, as we may lose some resets that can still happen while the clock of `x` is inactive, and must be taken into account. It would be possible to add delays to memorize the occurrence of a `reset` signal and process it later, when the clock of `x` is active, but this is an intricate transformation [HP00].

Instead, we treat `reset` as a primitive operation in the intermediate languages, and thus in the normalized form. This generates efficient code, especially in the case where a stateful construct has several different reset conditions as described in [section 5.3](#). In the normalized syntax, the existence of multiple reset conditions is modeled by the possibility of nested `reset` blocks. The Transcription pass transforms these nested blocks into simpler equations in the NLustre language, as described in [section 5.2.2](#).

4.4.1.2 Shared Variables

There are two ways of treating `last` variables in the compiler. The first is to eliminate them early by compiling them using `fb`. The declaration `last x = e` can be compiled by introducing a new identifier `lx` defined by the equation `lx = e fb x`. In the rest of the code, all occurrences of `last x` are replaced with `lx`. This yields a program that is semantically equivalent, and does not contain any `last` variables. The following compilation passes are simpler to define and verify, as they do not need to handle `last` variables and declarations.

However, the code generated by this scheme is not always optimal. Consider the program of [figure 4.3](#). It contains a partial definition for output `x`, which is authorized since `x` is declared with a `last` value. The generated C code, if we compile `last` with `fb`, is given at the bottom left. Notice that it copies the content of the state variable `(*self).lx` into temporary variable `x`, and then copies `x` back into `(*self).lx`. These extra copies are not ideal, and cannot be optimized away without either a complex analysis of the whole program, or additional invariants whose preservation would have to be proven across all intermediate compilation passes. We would like the code presented at right to be generated instead.

To achieve this, we treat `last` as a primitive construction in intermediate languages, and therefore keep it in the normalized syntax. This requires more work in the whole compiler chain, which we detail in [chapter 5](#).

4.4.2 Implementation and Notations

Most of the compilation passes detailed below implement the compilation scheme proposed in earlier work [CPP05; Bie+08]. We show how to adapt these schemes and prove them correct within a proof assistant.

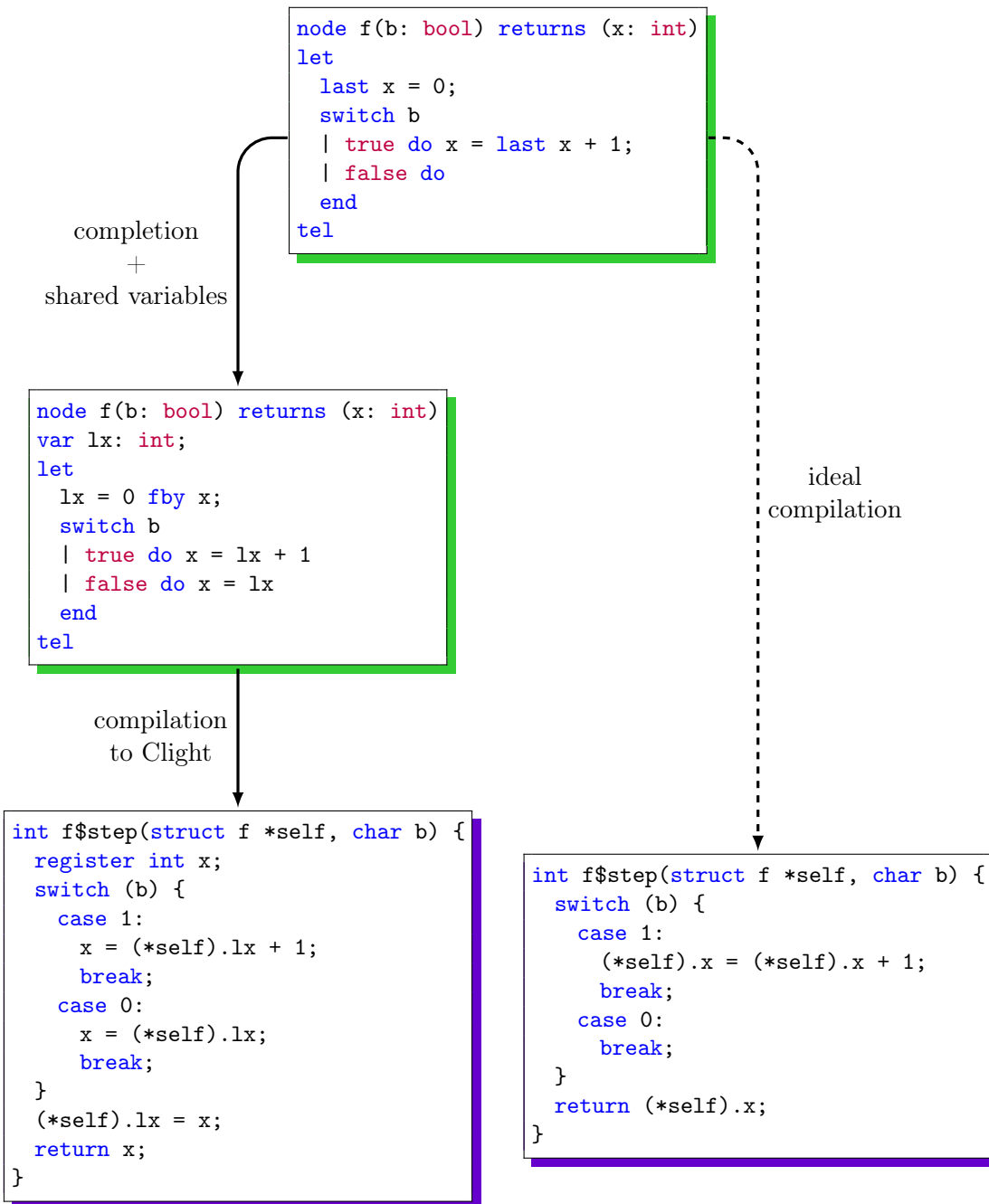


Figure 4.3: A node with partial definition and its compiled code

Each compilation pass is implemented as a distinct recursive function on the syntax of programs. All compilation functions are total. In this dissertation, we will use a unique notation to denote them. $\llbracket S \rrbracket_p$ denotes the compilation of syntactic element S (which can be an expression, block, program) under some additional parameters p . In most cases, this function returns a syntactic element of the same type as S . In some cases, it may instead return a list of syntactic elements, or a pair with syntactic elements and other values.

Each compilation pass is accompanied by proofs of type, clock, and semantics preservation. The proofs all proceed by induction on the definitions in a program, then on the syntax of blocks. We outline the invariants and core properties but omit the tedious syntactic properties that are usually also necessary, for example, typing invariants, uniqueness of declared identifiers, etc.

4.5 Completing Partial Definitions

As seen in [figure 1.6](#), if a variable has a defined `last` value, then it can be defined across a `switch` or state machine, even if some branches lack an equation for it. The source semantics completes any such partial definitions with an implicit $x = \text{last } x$ equation. To facilitate the compilation of `switch` block in a later pass, the compiler makes these equations explicit.

4.5.1 Compilation Function

The interesting cases of the compilation function are presented in [figure 4.4](#). we focus on `switch` and state machines, where definitions may need to be completed. For each branch, the function computes the difference between the set of all variables defined by the block and the set of variables defined by the branch. For each variable x in this difference, that is, for each missing definition in the branch, an equation $x = \text{last } x$ is added. In the notation for this pass, we take some liberties by conflating sets and lists. In the Coq implementation, we use type `PS.t` to represent sets of identifiers efficiently, and convert to lists using the `PS.elements : PS.t -> list ident` function.


4.5.2 Correctness

Proving the correctness of this compilation function is straightforward. The proof is articulated around the correctness invariant presented below. It states that if the original block has a semantics under history H , then the transformed block has a semantics under the same history.

Invariant 1 Completion of Partial Definitions 🐔 [Lustre/Complete/CompCorrectness.v:91](#)

$$\text{if } G, H, bs \vdash blk \text{ then } G, H, bs \vdash \llbracket blk \rrbracket$$

The interesting cases of the proof are the same ones shown in [figure 4.4](#): `switch` and state machines. In these cases, we must show that the added equations $x = \text{last } x$ have

$$\begin{aligned}
& \llbracket \text{switch } e \text{ } [C_i \text{ do } \text{blks}_i]^i \text{ end} \rrbracket \triangleq \\
& \text{let } \text{missing}_i := \bigcup_j \text{Def}(\text{blks}_j) / \text{Def}(\text{blks}_i) \text{ in} \\
& \text{let } \text{lasteqs}_i := \{x = \text{last } x \mid x \in \text{missing}_i\} \text{ in} \\
& \text{switch } e \text{ } [C_i \text{ do } \text{lasteqs}_i; \llbracket \text{blks}_i \rrbracket^i \text{ end} \\
\\
& \llbracket \text{automaton initially } \text{ini} \text{ } [\text{state } C_i \text{ var } \text{locs}_i \text{ do } \text{blks}_i \text{ until } \text{tr}_i]^i \text{ end} \rrbracket \triangleq \\
& \text{let } \text{missing}_i := \bigcup_j \text{Def}(\text{blks}_j) / \text{Def}(\text{blks}_i) \text{ in} \\
& \text{let } \text{lasteqs}_i := \{x = \text{last } x \mid x \in \text{missing}_i\} \text{ in} \\
& \text{automaton initially } \text{ini} \text{ } [\text{state } C_i \text{ var } \text{locs}_i \text{ do } \text{lasteqs}_i; \llbracket \text{blks}_i \rrbracket \text{ until } \text{tr}_i]^i \text{ end} \\
\\
& \llbracket \text{automaton initially } \text{ini} \text{ } [\text{state } C_i \text{ do } \text{blks}_i \text{ unless } \text{tr}_i]^i \text{ end} \rrbracket \triangleq \\
& \text{let } \text{missing}_i := \bigcup_j \text{Def}(\text{blks}_j) / \text{Def}(\text{blks}_i) \text{ in} \\
& \text{let } \text{lasteqs}_i := \{x = \text{last } x \mid x \in \text{missing}_i\} \text{ in} \\
& \text{automaton initially } \text{ini} \text{ } [\text{state } C_i \text{ do } \text{lasteqs}_i; \llbracket \text{blks}_i \rrbracket \text{ unless } \text{tr}_i]^i \text{ end}
\end{aligned}$$
Figure 4.4: Completion function  `Lustre/Complete/Complete.v:50`

a semantics under the sampled history of the branch. This is proven using the implicit completion premise introduced in [figure 2.23](#). The overall proof of semantic correctness for this pass is the shortest of the whole compiler, with around 250 LoC.

4.6 Dependency Analysis and Clocked Semantic Model

In the previous chapter, we defined the dependency analysis of Vélus. This analysis runs after the completion of partial definitions so that it does not have to take into account implicit `x = last x` equations. The analysis proceeds by constructing a dependency graph according to the rules of [section 3.1.2](#), and checks the absence of cycles in this graph using the algorithm presented in [section 3.2](#). If the analysis succeeds, we know that each node in the program is `node_causal`. This means, in particular, that the streams associated to these nodes respect the clock-correctness property described in [section 3.5](#).

While working on the front-end compiler, we realized that this clock correctness property is a necessary prerequisite in the correctness proofs for several of the compilation passes detailed in the following chapters, for three reasons.

First, most of the semantic operators (`fb`, `when`, `merge`, ...) are partial, and only allow arguments with the same, or complementary, clock streams. When building a composed expression from existing sub-expressions, we must prove that the streams produced by these sub-expressions indeed have appropriate clocks. In some cases, this can be deduced from other hypotheses in the context, but not always.

Second, the decomposition and recomposition of semantic hypotheses around the `reset` operator require that the calculated streams respect the expected masking, which is a corollary of the clock-correctness property.

Finally, the property must be preserved up to the generation of imperative code. Indeed, following the model proposed in [Bie+08], the imperative code generated from a dataflow program is “clock-directed”. The general idea is that clock-type annotations are used to generate conditionals that control the activation of instructions. Hence, the semantics of the imperative program depend directly on these clock-type annotations. Therefore, the correctness proof of the generation of imperative code requires that the semantics of intermediate languages respect the clock-correctness property.

For all of these reasons, we have to prove that the semantic model associated with each intermediate program respects the clock-correctness properties. Considering the number of separated compilation passes, this poses a proof-engineering issue. We describe below the three strategies that we considered to tackle this problem.

Repeating the Dependency Analysis To establish the clock-correctness property for a given program, we need to ensure that every node of the program is causal. The `node_causal` property depends on the syntax of the node, which means that it is not trivially preserved by compilation passes. One solution would be to re-analyze the program after each compilation pass. If the analysis fails (which should never happen if the initial analysis succeeded), then the compiler would simply raise an error message. In practice, the graph algorithm we implemented is efficient, but the algorithm building the graph from the program is less so, as it needs to visit the whole syntax tree of the program. This means that the dependency analysis is still relatively costly, and we prefer to run it only once.

Causality Preservation Instead of re-running the dependency analysis, we could directly prove that every compilation pass preserves the causality of the compiled node: if `node_causal n`, then `node_causal [n]`. We tried this approach on one of the simpler compilation passes: the `fbv` normalization. This proof is described briefly in [EMSOFT21], and at more length in [Pes20]. In our experience, writing this proof was very difficult. Indeed, as we have described, the `node_causal` predicate depends on the existence of an `AcyGraph` that is global to the node, while the proof must be built inductively on local transformations of the program. This discrepancy leads to a proof involving global invariants that are difficult to reason about. Additionally, this work was done in an earlier version of Vélus, before the introduction of causality labels, where the dependency graph was built directly from the variable names in the program. We believe that the level of indirection introduced by these labels would make reasoning even more cumbersome. For this reason, we decided to abandon this approach.

Instrumented Semantic Model The approach we finally adopted stems from the observation that we do not actually need to prove that each intermediate program is causal. We only need to establish that their semantic models respect the clock-correctness property, which is a consequence of causality. Therefore, we can simply prove that each compilation pass preserves this property, which is far easier than showing that they preserve causality. In practice, we define an extended semantic model that contains

$$\begin{array}{c}
 \frac{\forall x \text{ ck}, \Gamma(x) = \text{ck} \implies H, bs \vdash_{\text{ck}} x^{\text{ck}} \Downarrow vs}{\Gamma, bs \vdash_{\text{ck}} H} \\
 \text{(a) sc_vars } \color{red}{\text{Lustre/LClockedSemantics.v:73}} \\
 \\
 \frac{\begin{array}{l} G(f) = \mathbf{node} \ f(x_1, \dots, x_n) \ \mathbf{returns} \ (y_1, \dots, y_m) \ \mathbf{blk} \\ \forall i \in 1..n, H(x_i) \equiv xs_i \quad \forall i \in 1..m, H(y_i) \equiv ys_i \\ G, H, (\mathbf{base-of} \ (xs_1, \dots, xs_n)) \vdash_{\text{ck}} \mathbf{blk} \quad (\mathbf{ins} + \mathbf{outs}), (\mathbf{base-of} \ (xs_1, \dots, xs_n)) \vdash_{\text{ck}} H \end{array}}{G \vdash_{\text{ck}} f(xs_1, \dots, xs_n) \Downarrow (ys_1, \dots, ys_m)} \\
 \text{(b) sem_node_ck } \color{red}{\text{Lustre/LClockedSemantics.v:302}} \\
 \\
 \frac{\forall x, x \in \text{dom}(H') \iff x \in \text{locs} \quad G, H + H', bs \vdash_{\text{ck}} \mathbf{blks} \quad \text{locs}, bs \vdash_{\text{ck}} H}{G, H, bs \vdash_{\text{ck}} \mathbf{var} \ \text{locs} \ \mathbf{let} \ \mathbf{blks} \ \mathbf{tel}} \\
 \text{(c) sem_scope_ck } \color{red}{\text{Lustre/LClockedSemantics.v:89}} \\
 \\
 \frac{G, H, bs \vdash_{\text{ck}} es \Downarrow xss \quad G \vdash_{\text{ck}} f(xss) \Downarrow yss}{G, H, bs \vdash_{\text{ck}} f(es) \Downarrow yss} \\
 \text{(d) Sapp } \color{red}{\text{Lustre/LClockedSemantics.v:196}}
 \end{array}$$

Figure 4.5: Some core semantic rules of the clocked semantic model

enough additional information to deduce the clock-correctness property, and we prove that compilation passes preserve this model, rather than the reference model presented in [chapter 2](#). Most of the semantic rules of this model are the same as in the reference model, except for the ones presented in [figure 4.5](#). The model encodes the clock-correctness property at the level of declarations. We say that a history H has well-clocked semantics under environment Γ if, for each variable x with clock type ck in Γ , then x annotated by ck has well-clocked semantics, as per the definition of [figure 3.12c](#). The semantic rules for node and local scopes ensure that all histories have well-clocked semantics. Based on this hypothesis, it is easy to prove that the streams produced by any expression also respect the clock-correctness property.

Even if only these two rules are modified, the entire model needs to be redefined for this approach to work. Indeed, the inductive definition of the semantic rules for expressions depends on the rule for nodes (because of the node instantiation case, presented in [figure 4.5d](#)). Unfortunately, this means that almost identical definitions and lemmas are duplicated for the instrumented semantic model.

The theorem on the next page states that the existence of a source semantic model for a well-clocked and causal node implies the existence of an instrumented semantic model. It is proven using the causal inductive reasoning described in [section 3.5](#). The induction scheme allows to deduce that all input, output and local variables of the node

are associated to streams that respect the clock-correctness property. We use this fact to build witnesses for the new premises of the node and local declaration rules.

Theorem 4 (Existence of a Clocked Semantic 🦊 [Lustre/LClockCorrectness.v:2610](#))

if $G \vdash_{wc} f$ **and** `node_causal` $G(f)$
and $G \vdash f(xss) \Downarrow yss$
and $G(f) = \text{node } f([x_i : ck_i]^i) \text{ returns } ([y_j : ck'_j]^j) \text{ blk}$
and $\forall i, H, (\text{base-of } xss) \vdash_{ck} x_i^{ck_i} \Downarrow xss_i$
then $G \vdash_{ck} f(xss) \Downarrow yss$

4.7 Compiling State Machines

After the dependency analysis, compilation continues with the elimination of state machines. This pass follows the scheme proposed in [CPP05], by transforming a state machine into a combination of `switch` and `reset` blocks. This transformation is simplified because we separate state machines with strong and weak transitions, proposing a dedicated compilation scheme for each type.

4.7.1 Compilation Function

The compilation function is presented in [figure 4.6](#). It takes the block to be compiled as an input and returns two values: the compiled block and a set of type declarations to be added to the program. Indeed, the list of states of each state machine must be converted to a new type so that the state labels may be manipulated as values. We choose to collect these new types in the same function that compiles blocks because it simplifies reasoning on the well-typedness of the resulting program.

State machines with weak transitions are treated by generating a single `switch`, with `reset` blocks in each branch. The condition of the `switch` and `reset` corresponds to the state stream described by the semantics for this construction. It is stored in fresh variables x_{st} , x_{res} . The next value of the state stream is calculated by the compiled transitions, and is stored in x_{st1} and x_{res1} . This corresponds exactly to the semantic rules of [section 2.9](#). In the same way, the compilation of initialization and transitions reifies the semantics of these constructs. In the compiled code, the initialization expression must be sampled on the clock of the state machine. We write C^{ck} for the constant C sampled (using `when`) on the clock ck .

The compilation of state machines with strong transitions is a bit more involved. It requires the generation of two `switch` blocks. The first one handles the transitions and controls which branch of the second one will be active in the same cycle. Again, this reflects the semantic model exactly.

$$\begin{aligned}
& \llbracket \text{automaton initially } ini^{ck} \text{ [state } C_i \text{ var } locs_i \text{ do } blks_i \text{ until } tr_i \text{]}^i \text{ end} \rrbracket \triangleq \\
& \text{let } blks'_i, tys_i := \llbracket blks_i \rrbracket \text{ in} \\
& \text{var } x_{st}, x_{res}, x_{st1}, x_{res1}; \\
& \text{let} \\
& \quad (x_{st}, x_{res}) = \llbracket ini \rrbracket_{ck} \text{ fby } (x_{st1}, x_{res1}); \\
& \quad \text{switch } x_{st} \llbracket C_i \text{ do reset var } locs_i \text{ let } (x_{st1}, x_{res1}) = \llbracket tr_i \rrbracket_{C_i}; blks'_i \text{ tel every } x_{res} \rrbracket^i \text{ end} \\
& \text{tel, } (\{\text{type } ty = (C_i)^i\} \cup (\bigcup_i tys_i)) \\
\\
& \llbracket \text{automaton initially } C^{ck} \text{ [state } C_i \text{ do } blks_i \text{ unless } tr_i \text{]}^i \text{ end} \rrbracket \triangleq \\
& \text{let } blks'_i, tys_i := \llbracket blks_i \rrbracket \text{ in} \\
& \text{var } x_{st}, x_{res}, x_{st1}, x_{res1}; \\
& \text{let} \\
& \quad (x_{st}, x_{res}) = (C^{ck}, \text{false}^{ck}) \text{ fby } (x_{st1}, x_{res1}); \\
& \quad \text{switch } x_{st} \llbracket C_i \text{ do reset } (x_{st1}, x_{res1}) = \llbracket tr_i \rrbracket_{C_i} \text{ every } x_{res} \rrbracket^i \text{ end}; \\
& \quad \text{switch } x_{st1} \llbracket C_i \text{ do reset } blks'_i \text{ every } x_{res1} \rrbracket^i \text{ end} \\
& \text{tel, } (\{\text{type } ty = (C_i)^i\} \cup (\bigcup_i tys_i)) \\
\\
& \llbracket \text{otherwise } C \rrbracket_{ck} \triangleq (C^{ck}, \text{false}^{ck}) \\
& \llbracket \text{if } e \text{ then } C \text{ ini} \rrbracket_{ck} \triangleq \text{if } e \text{ then } (C^{ck}, \text{false}^{ck}) \text{ else } \llbracket ini \rrbracket_{ck} \\
\\
& \llbracket \epsilon \rrbracket_{C_d} \triangleq (C_d, \text{false}) \\
& \llbracket \text{! } e \text{ then } C; tr \rrbracket_{C_d} \triangleq \text{if } e \text{ then } (C, \text{true}) \text{ else } \llbracket tr \rrbracket_{C_d} \\
& \llbracket \text{! } e \text{ continue } C; tr \rrbracket_{C_d} \triangleq \text{if } e \text{ then } (C, \text{false}) \text{ else } \llbracket tr \rrbracket_{C_d}
\end{aligned}$$
Figure 4.6: Compilation of State Machines 🐔 [Lustre/CompAuto/CompAuto.v:66](#)

4.7.2 Correctness

The invariant for the correctness proof is presented below; it states that the semantics of the transformed block is preserved under the same history. The simplicity of this invariant comes from the fact that the transformation is local to a block. The variables introduced when compiling a state machine are captured by the scope and have no influence on the semantics of parents or sub-blocks.

Invariant 2 Compilation of State Machines 🐔 [Lustre/CompAuto/CACorrectness.v:558](#)

$$\text{if } G, H, bs \vdash blk \quad \text{and} \quad \llbracket blk \rrbracket = (blk', tys) \quad \text{then} \quad G, H, bs \vdash blk'$$

In the case of the proof where the blk is a state machine, the newly introduced variables are associated with streams that reflect the state streams appearing in the semantic rules. To establish a semantic for the `switch` and `reset` blocks, we rely on the correspondence of select with when and mask described in [lemma 2](#). We have omitted some details from the proof, which we detail further in [appendix C](#). Still, the proof of semantic correctness for this pass is not too complex. This is mostly because the semantic definitions are directly

mirrored in the source-to-source transformation. Still, the relative complexity of state machines is only partially resolved by this pass; `switch`, `reset`, and local declarations remain, and we will see further that compiling them is more difficult.

4.8 Compiling Switch Blocks

In the previous section, we showed how state machines are compiled by introducing `switch` blocks. We now show how these blocks are compiled into the core dataflow language. We follow the compilation scheme from [CPP05], which transforms `switch` blocks using the `when` and `merge` operators.

4.8.1 Compilation Function

The compilation function for `switch` blocks is presented in figure 4.7. It is parameterized by a substitution σ for renaming variables and a clock type ck for resampling constants. The parameters account for enclosing `switch` blocks. Compilation, renaming, and resampling thus occur together in a single pass. This complicates certain invariants but satisfies the Coq syntactic criterion for checking function termination. We only present the interesting case of the function. To compile a `switch`, we start by introducing new local variables. First, x_{cond} corresponds to the condition stream. Second, a sampled variable $x_j^{C_i}$ is introduced for any variable or `last` variable x_j that is free (FV) in any block, for each branch label C_i . Finally, a sampled variable $y_j^{C_i}$ is introduced for any variable y_j that is defined (DV) in any block, for each branch label C_i . Each of the new free variables $x_j^{C_i}$ samples the original variable x_j with `when` according to the condition variable and the appropriate branch label. The defined variables y_j are reconstituted by a `merge` over their branch-specific versions $y_j^{C_i}$. Finally, the compilation function is applied recursively on each branch, adding substitutions for branch-specific free and defined variables, and a deeper, branch-specific clock type. The bottom part of figure 4.7 presents three important cases from the expression compilation function: constants are resampled on the given clock, variables are renamed, and recursively so, through `when` and other expressions.

Simplification and Optimization The Coq implementation does not actually compute $FV(blks_i)$. It instead samples all variables not defined within the `switch` and on the clock of the condition, whether they are used or not. The `switch` clock typing rule guarantees that this gives a superset of the free variables. This approach simplifies the proof since we do not have to reason about the FV function. A later pass (section 5.2.3.2) eliminates variables and equations that are not required to calculate an output variable, whether they are introduced by the compiler or a programmer.

4.8.2 Correctness

As we have seen, the compilation of `switch` involves a substitution for renaming variables. For this compilation pass and some of the following, we will express invariants using

$$\begin{aligned}
& \left[\text{switch } e \text{ } [C_i \text{ do } \text{blks}_i]^i \text{ end} \right]_{\sigma, ck} \triangleq \\
& \text{var } x_{cond}, \dots; \text{let} \\
& \quad x_{cond} = \left[e \right]_{\sigma, ck}; \\
& \quad x_j^{C_i} = \sigma(x_j) \text{ when } C_i(x_{cond}); \quad \forall x_j \in \text{FV}(\text{blks}_i), \forall C_i \\
& \quad \sigma(y_j) = \text{merge } x_{cond} \left[(C_i \Rightarrow y_j^{C_i}) \right]^i; \quad \forall y_j \in \text{DV}(\text{blks}_i) \\
& \quad \left[\text{blks}_i \right]_{\{x_j \mapsto x_j^{C_i}\} \circ \{y_j \mapsto y_j^{C_i}\} \circ \sigma, ck \text{ on } C_i(x_{cond})} \\
& \text{tel} \\
& \quad \left[c \right]_{\sigma, \bullet} \triangleq c \\
& \quad \left[c \right]_{\sigma, ck \text{ on } C(x)} \triangleq \left[c \right]_{\sigma, ck} \text{ when } C(x) \\
& \quad \left[x \right]_{\sigma, ck} \triangleq \sigma(x) \\
& \quad \left[e \text{ when } C(x) \right]_{\sigma, ck} \triangleq \left[e \right]_{\sigma, ck} \text{ when } C(\sigma(x))
\end{aligned}$$

Figure 4.7: Compiling `switch` blocks 🐔 [Lustre/ClockSwitch/ClockSwitch.v:127](#)

history extension modulo substitution; $H_1 \sqsubseteq_{\sigma} H_2$ signifies that “ H_2 extends H_1 after renaming by σ ”. When σ is the identity function, we write $H_1 \sqsubseteq H_2$.

Definition 5 (History extension modulo substitution)

$$H_1 \sqsubseteq_{\sigma} H_2 \quad \text{iff} \quad \forall x \text{ vs}, H_1(x) \equiv \text{vs} \rightarrow H_2(\sigma(x)) \equiv \text{vs}$$

The inductive proof of correctness is structured around the invariant below. It assumes that the source block `blk` has a semantics for history H_1 and base clock bs . The base clock corresponds to the evaluation of clock type ck under a faster base clock bs' . The new history H_2 extends H_1 modulo the substitution σ . If these conditions hold, then the compiled block has a semantics under H_2 and bs' .

Invariant 3 (Compilation of `switch` blocks 🐔 [Lustre/ClockSwitch/CSCorrectness.v:597](#))

$$\begin{aligned}
& \text{if } G, H_1, bs \vdash \text{blk} \quad \text{and} \quad H_1, bs' \vdash ck \Downarrow bs \quad \text{and} \quad H_1 \sqsubseteq_{\sigma} H_2 \\
& \text{then } G, H_2, bs' \vdash \left[\text{blk} \right]_{\sigma, ck}
\end{aligned}$$

For the case of a switch, we know that the source block has a semantics under a history H . To prove that the generated local scope has a semantics, each introduced variable is associated to a stream satisfying the corresponding equation. The variable x_{cond} is associated with a condition stream cs whose correctness follows from a lemma on expressions. For each branch, a sampled free variable $x_j^{C_i}$ is associated with the stream $\text{when}^{C_i} cs H(x_j)$. The semantics of the new equations for defined variables, the $\sigma(y_j)$, follows from the relation between `when` and `merge` stated in [lemma 12](#). The semantics for the blocks in branches follows inductively from the invariant, setting H_1 to H , H_2 to H extended with the new variables, σ to the substitution introduced by the compilation function, and ck to the clock for the corresponding branch. Compared to a source block

```

node f(x: int) returns (y, z: bool)
let
  var t: int;
  let t = x fby (t + 1);
    y = t > 10;
tel;
var t: int;
let t = x + 1;
  z = t < 0;
tel
tel

node f(x: int) returns (y, z: bool)
var t : int; local$t$1 : int;
let
  t = x fby (t + 1);
  y = t > 10;
  local$t$1 = x + 1;
  z = local$t$1 < 0
tel
tel

```

Figure 4.8: Example of the flattening of local scopes

within a branch with base clock bs , the compiled version is activated on the faster base clock of the new context bs' . The slower clock bs corresponds to the clock type given as a parameter of the compilation function.

Lemma 12 Correspondence of when and merge 🐔 [Lustre/ClockSwitch/CSCorrectness.v:111](#)

$$\text{merge } cs \text{ (when}^{C_1} vs \text{ } cs, \dots, \text{when}^{C_n} vs \text{ } cs) \equiv vs$$

4.9 Flattening Local Scopes

Nested local declarations may be present in a source node or introduced by the two previous compilation passes. A normalized program, however, may only have a single local declaration block as the top-most block. We now describe the pass that flattens nested local scopes into a single one.

The main difficulty of this pass arises when a node uses the same identifier for two local declarations, as is the case in [figure 4.8](#). In that case, to produce a well-formed node, at least one of the two declarations must be renamed to a fresh identifier, and the renaming must be propagated in all sub-blocks of the declaration.

4.9.1 Compilation Function

The transformation function is shown in [figure 4.9](#). Its parameter is a substitution σ which encodes renamings. Compiling an equation applies the substitution to each variable at left of the equation, and to all identifiers in each expression at right, which we write $\sigma(e)$. Reset blocks are traversed recursively. For local declarations, the `fresh_idents` function builds a substitution from the existing declarations to new identifiers; we detail below how this function is implemented. The compilation function is then called recursively with the extended substitution. In addition to the sub-blocks, the list of declarations is also returned. Compiling a node means compiling its body, and constructing a unique top-level scope with this list of flattened and renamed declarations.

$$\begin{aligned}
\llbracket xs = es \rrbracket_{\sigma} &\triangleq \sigma(xs) = \sigma(es), [] \\
\llbracket \text{last } x = e \rrbracket_{\sigma} &\triangleq \text{last } \sigma(x) = \sigma(e), [] \\
\llbracket \text{reset } blks \text{ every } e \rrbracket_{\sigma} &\triangleq \text{let } blks', xs := \llbracket blks \rrbracket_{\sigma} \text{ in reset } blks' \text{ every } \sigma(e), xs \\
\llbracket \text{var } locs \text{ let } blks \text{ tel} \rrbracket_{\sigma} &\triangleq \text{let } \sigma' := \text{fresh_idents}(locs) \text{ in} \\
&\quad \text{let } blks', xs := \llbracket blks \rrbracket_{(\sigma+\sigma')} \text{ in } blks', xs + \sigma'(locs) \\
\llbracket \text{node } f(ins) \text{ returns } (outs) \text{ blk} \rrbracket &\triangleq \text{let } blks', xs := \llbracket blk \rrbracket_{id} \text{ in} \\
&\quad \text{node } f(ins) \text{ returns } (outs) \text{ var } xs \text{ let } blks' \text{ tel}
\end{aligned}$$

Figure 4.9: Flattening of local scopes 🐔 [Lustre/InlineLocal/InlineLocal.v:513](#)

4.9.2 Fresh identifiers and the Reuse monad

The `fresh_idents` function needs to generate fresh identifiers to replace existing declarations. An obvious way to implement this would be to use the `Fresh` monad described in [section 4.2.2](#). This is, however, not ideal: using this monad would, by default, rename all local identifiers, and thus hinder traceability between the source and produced nodes; in particular, if we use the scheme of [figure 4.9](#) as is, this would even rename the top-level declarations of the source node. Instead, we define an extended version of the `Fresh` monad, which we call the `Reuse` monad. It only generates new identifiers when strictly necessary, that is when an identifier is used twice. The definition of the monad is given in [listing 4.12](#). Its state contains both a `fresh_st` for identifier generation, and a set of identifiers that are already used. When `reuse_ident x` is called, either the identifier has not yet been used, in which case it is returned directly, or it has, and a fresh identifier is returned.

```

Record reuse_st := {
  fresh_st: fresh_st local unit;
  used: PS.t;
}.
Definition Reuse A := reuse_st -> (A * reuse_st).

Definition reuse_ident (x : ident) : Reuse ident :=
  fun st =>
    if PS.mem x st.(used) then
      let (y, st') := fresh_ident (Some x) tt st.(fresh_st) in
      (y, {| fresh_st := st'; used := st.(used) |})
    else
      (x, {| fresh_st := st.(fresh_st); used := PS.add x st.(used) |}).

```

Listing 4.12: Reuse monad 🐔 [Lustre/InlineLocal/InlineLocal.v:242](#)

4.9.3 Correctness

The correctness invariant for the compilation of local scopes is presented below. The semantics of the source block are given under history H_1 . The second hypothesis states the existence of a history H_2 which refines H_1 modulo substitution σ . H_2 gives a semantic to the blocks after application of the substitution σ . We know the exact domain of this history and that it is well clocked. We then prove that there exists an history H_3 which gives a semantics to the compiled block. In addition to the associations in H_2 , H_3 associates a stream to each variable returned by the compilation function.

Invariant 4 (Compilation of local scopes 🦉 [Lustre/InlineLocal/ILCorrectness.v:516](#))

if $G, H_1, bs \vdash blk$
and $H_1 \sqsubseteq_{\sigma} H_2$ **and** $\text{dom}(H_2) = \Gamma$ **and** $\Gamma, bs \vdash_{\text{ck}} H_2$
and $\llbracket blk \rrbracket_{\sigma} = (blks', xs)$
then $\exists H_3, H_2 \sqsubseteq H_3 \wedge \text{dom}(H_3) = \Gamma + xs \wedge (\Gamma + xs), bs \vdash_{\text{ck}} H_3 \wedge G, H_3, bs \vdash blks'$

There are two difficulties in this proof. Both are related to the construction of the new history H_3 . First, when encountering a block of local declarations, a new history actually needs to be built. From the context, we know that $G, H_1, bs \vdash \text{var } locs \text{ let } blks \text{ tel}$, and, from the second hypothesis of the invariant, that we have a history H_2 that refines H_1 modulo substitution. By the first hypothesis, the semantic rules for local blocks tell us that there exists a local history H'_1 such that $G, (H_1 + H'_1), bs \vdash blks$. We must construct H'_2 , the projection of H'_1 by substitution $\sigma' = \text{fresh_idents}(locs)$ to continue the induction on sub-blocks. To do so, we use the inverse of a substitution σ^{-1} , and define $H'_2(x) = H'_1(\sigma'^{-1}(x))$. We then need to prove that all the pre-conditions of the invariant hold for $(H_2 + H'_2)$.

The second difficulty was alluded to in [section 3.3.2](#). When treating the case of a [reset](#) block, applying the induction hypothesis gives us an infinite family of histories H'_k :

$$\begin{aligned} \forall k, \exists H'_k, H'_k \sqsubseteq \text{mask}^k rs H_2 \wedge \text{dom}(H'_k) = \Gamma + xs \\ \wedge (\Gamma + xs), (\text{mask}^k rs bs) \vdash_{\text{ck}} H'_k \\ \wedge G, H'_k, (\text{mask}^k rs bs) \vdash blks' \end{aligned}$$

However, to prove the existence of a semantic model for the compiled [reset](#) block, we need to construct a single history H_3 such that $\forall k, G, \text{mask}^k_{rs} (H_3, bs) \vdash blks'$. This means “combining” the infinite family of H'_k into a single history. This is possible, since we know that each H'_k is correctly sampled by the masked base-clock. We cannot, however, give a constructive proof that H_3 exists. Indeed, we need to construct a single value from an infinite number of values; which requires introducing the axiom of choice in our formalization. This axiom is presented, as it is implemented in Coq, in [listing 4.13](#). Given a relation R between types A and B , the axiom posits that, if every element of A is in relation with at least one element of B , then there exists a function which “chooses” the element of B for any element of A .

```
Axiom functional_choice: ∀ (A B : Type) (R : A -> B -> Prop),
  (∀ (x : A), ∃ (y : B), R x y) ->
  (∃ f : (A -> B), ∀ (x : A), R x (f x)).
```

Listing 4.13: Axiom of Choice in Coq

Using this axiom, we can prove [lemma 13](#). For a given relation P , if, for all k , there exists a history H such that the k th instance of H satisfies P , then there exists a single history H such that for all instances of H , P is satisfied. We can finally use this lemma to build a history H_3 that gives a semantics to the compiled `reset` block.

Lemma 13 (Axiom of Choice applied to `mask` 🐔 [CoindStreams.v:2570](#))

$$\text{if } \forall k, \exists H, P \ k \ (\text{mask}^k \text{ rs } H) \text{ then } \exists H, \forall k, P \ k \ (\text{mask}^k \text{ rs } H)$$

4.10 Unnesting and Distribution

The compilation passes presented in the previous sections have eliminated each of the complex block structures from the program. To fit the normalized syntax of the language, it remains to simplify the syntax of expressions and equations. The unnesting and distribution pass achieves a first simplification step by isolating `fb` and node instantiations in their own equations, and by distributing operators over their operands.

4.10.1 Compilation Function

The unnesting of an expression $[e]$ produces a list of expressions due to distribution. For example, the expression $(x, y) \text{ when } ck$, would become $(x \text{ when } ck, y \text{ when } ck)$. It also produces a list of auxiliary equations due to unnesting.

Several cases of this function are presented in [figure 4.10](#). Constants and variables are not changed and do not add any new equation. Binary operators are treated recursively. The expressions at left of a `when` are treated recursively and the original `when` is distributed over the resulting expressions. For `fb`s, the recursively generated expressions are combined pair-by-pair into new equations defining fresh variables. The case for node instantiations, $[f(es)]$, is similar but does not require distribution after unnesting. We do not show the cases for the `merge` or `if`, as they are similar to that of the `fb`. In the full definition, we add special cases for subexpressions where unnesting is not required by the grammar of [figure 4.2](#). The aim is to minimize transformations to the original program. For instance, if a `fb` or node instance already appears directly in an equation, there is no need to unnest it. Finally, `reset` blocks are transformed by unnesting the condition in its own equation, and distributing the `reset` operation over each unnested sub-block.

$$\begin{aligned}
\llbracket c \rrbracket &\triangleq ([c], []) \\
\llbracket x \rrbracket &\triangleq ([x], []) \\
\llbracket e_1 \oplus e_2 \rrbracket &\triangleq \text{let } ([e'_1], eqs'_1) := \llbracket e_1 \rrbracket \text{ in} \\
&\quad \text{let } ([e'_2], eqs'_2) := \llbracket e_2 \rrbracket \text{ in} \\
&\quad ([e'_1 \oplus e'_2], eqs'_1 \cup eqs'_2) \\
\llbracket es \text{ when } C(x) \rrbracket &\triangleq \text{let } [e_i]^i, eqs' := \llbracket es \rrbracket \text{ in} \\
&\quad ([e_i \text{ when } C(x)]^i, eqs') \\
\llbracket es_0 \text{ fby } es_1 \rrbracket &\triangleq \text{let } ([e_{0i}']^i, eqs'_0) := \llbracket es_0 \rrbracket \text{ in} \\
&\quad \text{let } ([e_{1i}']^i, eqs'_1) := \llbracket es_1 \rrbracket \text{ in} \\
&\quad ([x_i]^i, [x_i = e_{0i}' \text{ fby } e_{1i}']^i \cup eqs'_1 \cup eqs'_2) \\
\llbracket f(es) \rrbracket &\triangleq \text{let } (es', eqs') := \llbracket es \rrbracket \text{ in} \\
&\quad ([x_i]^i, [[x_i]^i = f(es')] \cup eqs') \\
\llbracket \text{reset } blk_i \text{ every } e \rrbracket &\triangleq \text{let } [blk_i]^i := \llbracket blk_i \rrbracket \text{ in} \\
&\quad \text{let } e', eqs' := \llbracket e \rrbracket \text{ in} \\
&\quad [\text{reset } blk_i \text{ every } x]^i \cup [x = e'] \cup eqs'
\end{aligned}$$

Figure 4.10: Unnesting of expressions 🐔 [Lustre/Unnesting/Unnesting.v:303](#)

4.10.2 Correctness

As with the previous pass, reasoning about the correctness of this transformation revolves around history extension. The central correctness invariant is presented below. Given the existence of a semantics for the source expression under history H_1 , it states the existence of a history H_2 extending H_1 which gives a semantics to the new expressions and equations. In the proof, every time an unnesting happens, this history is constructed by associating the newly introduced variables to the streams produced by the unnested expression. The proof is straightforward but tedious due to the number of cases in the syntax and the special cases defined to minimize the number of unnestings.

Invariant 5 Unnesting of expressions 🐔 [Lustre/Unnesting/UCorrectness.v:763](#)

$$\begin{aligned}
&\mathbf{if} \quad G, H_1, bs \vdash e \Downarrow vs \quad \mathbf{and} \quad [e] = (es', eqs') \\
&\mathbf{then} \quad \exists H_2, H_1 \sqsubseteq H_2 \wedge G, H_2, bs \vdash es' \Downarrow vs \wedge G, H_2, bs \vdash eqs'
\end{aligned}$$

4.11 Normalization of shared variables

While the NLustre language accepts shared (`last`) variables, it places three restrictions on their placement and definition. First, shared variables may only be initialized by constant expressions. This ensures that the state of the generated imperative code is initialized in

constant time by a function that does not depend on inputs. Second, outputs may not be declared with shared variables. This simplifies the definition of later languages, where state variables cannot be used as outputs. Finally, equations may define variables used with `last` with simple or control expressions, but not with `fbys` or node calls. Doing so simplifies the compilation of later languages by making explicit each update of a state variable. The first requirement is reflected by the normalized syntax presented in [figure 4.2](#). The other two are characterized as inductive predicates on the node.

The three requirements we describe above are simple, but they may overlap. Consider the equations `last y = y0; y = f(last y)`, where `y` is an output of the node being defined. They violate each of the requirements defined above. However, simply applying the transformation that eliminates `last` initialized by non-constants values (defined below) is sufficient to fit all the restrictions. In the case of equations `last y = 0; y = f(last y)`, where only the second and third requirements are violated, two separate transformations are necessary to eliminate the output `last` and the definition by `y` by a node application. As we see, these combinations can make the compilation subtle, especially if we want to produce efficient code. To avoid simplify the algorithms and proofs, we separate this pass into three transformations, each treating one requirement. All of these passes are structured in the same way. They (i) analyze the node to find which `last` needs to be removed or renamed, (ii) generate new identifiers (using the `Fresh` monad) for these `lasts` at the top-level scope and a substitution from old to new identifiers, and (iii) apply this substitution to the sub-blocks of the top-level scope.


4.11.1 Initializing lasts with constants

First, we eliminate `lasts` initialized by non-constant expressions. There is no way to compile such declarations using `last` only: they are instead transformed into `fbys` equations. This introduces unavoidable copies in the code and prevents the ideal compilation scheme described in [section 4.4.1.2](#). This is not so much a problem in real-world programs: ideally, programmers initialize their shared variables with constants.


The compilation function is presented in [figure 4.12](#). First, it calls `nconst-lasts blk`, which returns the set of `last` variables initialized by a non-constant expression in block `blk`. These are the `lasts` which must be transformed. The substitution σ associates old identifiers that appear in this set to fresh identifiers. By construction, σ only renames `last` variables; other variables in the node are unchanged. These fresh identifiers are declared in the top-level scope. The sub-blocks are compiled by applying the substitution recursively to every sub-expression. The `last` initialization case is particular: if the variable of interest appears in the substitution, then a `fbys` equation is generated; otherwise, the `last` equation is not changed.

Correctness The invariant for the compilation of local blocks is stated below. It relies on the refinement of histories modulo substitution. This refinement is enough to give a semantics to sub-expressions on which σ was applied. The most interesting case is for an initialization equation `last x = e`, when `last x` appears in the substitution and a `fbys`

$$\begin{aligned}
\text{is-constant } c &\triangleq \text{ true} \\
\text{is-constant } C &\triangleq \text{ true} \\
\text{is-constant } (e \text{ when } C(X)) &\triangleq \text{ is-constant } e \\
\text{is-constant } e &\triangleq \text{ false} \quad (\text{otherwise}) \\
\text{nconst-lasts } (xs = e) &\triangleq \emptyset \\
\text{nconst-lasts } (\text{last } x = e) &\triangleq \text{ if } (\text{is-constant } e) \text{ then } \emptyset \text{ else } \{x\} \\
\text{nconst-lasts } (\text{reset } blk \text{ every } x) &\triangleq \text{ nconst-lasts } blk
\end{aligned}$$

Figure 4.11: Non-constant `last` initializations  [Lustre/NormLast/NormLast.v:80](#)

$$\begin{aligned}
& \llbracket \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} \text{ let } \text{blks} \text{ tel} \rrbracket \triangleq \\
& \text{let } \sigma := \{\text{last } x \mapsto \text{last } \$x \mid x \in \text{nconst-lasts } \text{blks}\} \text{ in} \\
& \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} + [\sigma(x) \mid x \in \text{nconst-lasts } \text{blks}] \text{ let } \llbracket \text{blks} \rrbracket_{\sigma} \text{ tel} \\
& \llbracket xs = e \rrbracket_{\sigma} \triangleq xs = \sigma(e) \\
& \llbracket \text{last } x = e \rrbracket_{\sigma} \triangleq \begin{cases} \text{if } \text{last } x \in \sigma \text{ then } \sigma(\text{last } x) = \sigma(e) \text{ fby } x \\ \text{else } \text{last } x = \sigma(e) \end{cases} \\
& \llbracket \text{reset } blk \text{ every } x \rrbracket_{\sigma} \triangleq \text{reset } \llbracket blk \rrbracket_{\sigma} \text{ every } x
\end{aligned}$$

Figure 4.12: Explication of `last` initializations  [Lustre/NormLast/NormLast.v:136](#)

equation is generated. Giving a semantics to the `fbv` operator is direct, since the same `fbv` semantic operator is used to characterized both `last` and `fbv`. The other obligations are deduced directly from the refinement hypothesis.

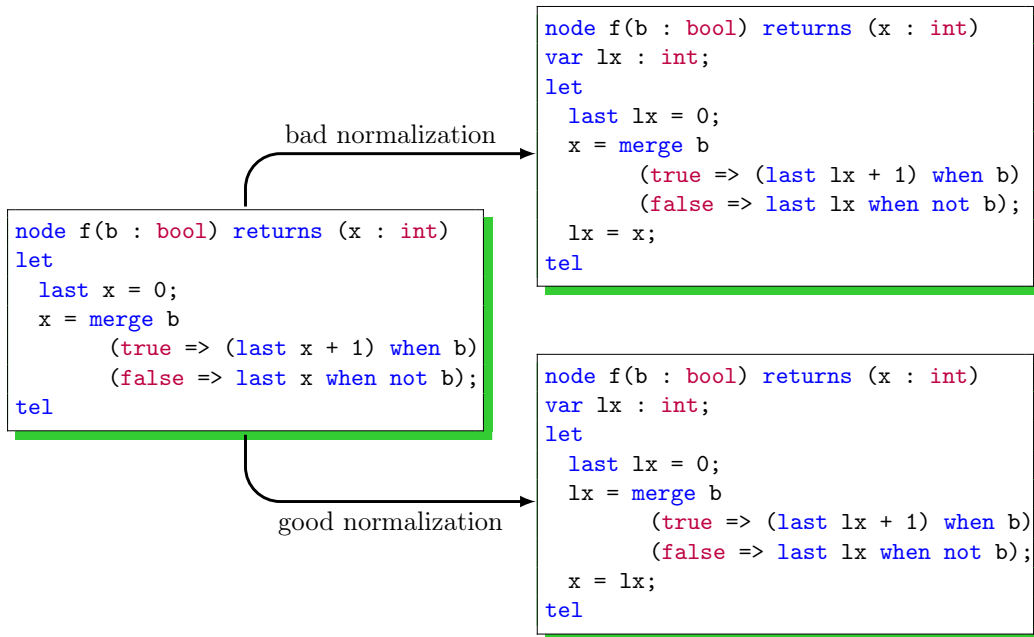
Invariant 6 Explication of `last` initializations  [Lustre/NormLast/NLCorrectness.v:59](#)

$$\text{if } G, H_1, bs \vdash blk \text{ and } H_1 \sqsubseteq_{\sigma} H_2 \text{ then } G, H_2, bs \vdash \llbracket blk \rrbracket_{\sigma}$$

While H_1 is the history of the source program, to instantiate the invariant, we need to provide H_2 , as well as a proof that semantic refinement holds. Again, we may build H_2 by applying reverse substitution on H_1 : we give $H_2(x) = H_1(\sigma^{-1}(x))$.

4.11.2 Removing lasts on outputs

The second pass removes `lasts` on node outputs. To do so, we introduce new local variables that will be used as `lasts` instead of the outputs, and add appropriate copies. The direction in which we introduce these copies is important. Consider the program in [figure 4.13](#) at left. There are two ways to compile it. The first, at top, inserts a new local variable `lx` and copies output `x` to it. The left-hand-side of the equation defining `x` is unchanged, while the right-hand-side is transformed, to change `last x` to `last lx`. We thus loose the syntactic relation between `x` and its last value presented in the `merge` equation of the source program. With this program, we would generate code akin to the

Figure 4.13: Two choices for normalization of `last` on output

one at bottom-left of figure 4.3, which does not allow us to remove the useless update of the state variable in the branch where `b` is `false`. For this reason, it is better to generate the program at the bottom, where `x` is also replaced with `lx` at left of the `merge` equation, and where output `x` is defined as equal to `lx`.

The compilation function for this pass is presented in figure 4.14. In the compiler, each declaration of a variable that is used with `last` is annotated so as to build the environment of typing and clock-typing judgements. This means that the function does not need to analyze definitions to find out which outputs are defined and used with `last`. For each of these, the function introduces a fresh local variable and builds a substitution σ . It then transforms the sub-block, and their sub-expressions. In expressions, the substitution is applied only at points where the `last` value of a variable is read. Other variables are unchanged. Applying $[blk]_{\sigma}$ produces a list of blocks: the original block where renaming by σ has been applied, and possibly some `x = lx` equations. The most interesting case is the one for equations. Renaming is applied to the left-hand-side variable `x` and the right-hand-side expression. If `x` is in the domain of σ , that is, if it was indeed renamed, it also introduces an equation between the new and old identifiers. Renaming is also applied at left of `last` initialization equations.

Correctness Refinement modulo substitution is not enough alone to state the correctness of this pass. Indeed, while `last` variables and declarations are systematically renamed, other variables are only renamed when they appear at left of equations. This transformation must be reflected in the history used to give a semantics to the new block. We show the necessary relation between an old history H_1 and a new history H_2 in the

$$\begin{aligned}
& \llbracket \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} \text{ let } \text{blks} \text{ tel} \rrbracket \triangleq \\
& \text{let } \sigma := \{x \mapsto \text{last } x \mid \text{last } x \in \text{outs}\} \text{ in} \\
& \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} + [\sigma(x) \mid \text{last } x \in \text{outs}] \text{ let } \llbracket \text{blks} \rrbracket_{\sigma} \text{ tel} \\
\\
& \llbracket \text{last } x \rrbracket_{\sigma} \triangleq \text{last } \sigma(x) \\
& \llbracket x \rrbracket_{\sigma} \triangleq x \\
\\
& \llbracket x = e \rrbracket_{\sigma} \triangleq \begin{cases} \text{if } x \in \sigma \text{ then } \sigma(x) = \llbracket e \rrbracket_{\sigma}; x = \sigma(x) \\ \text{else } x = \llbracket x \rrbracket_{\sigma} \end{cases} \\
& \llbracket \text{last } x = c \rrbracket_{\sigma} \triangleq \text{last } \sigma(x) = c \\
& \llbracket \text{reset } \text{blk} \text{ every } x \rrbracket_{\sigma} \triangleq \text{let } [\text{blk}'_i]^i := \llbracket \text{blk} \rrbracket_{\sigma} \text{ in } \llbracket \text{reset } \text{blk}'_i \text{ every } x \rrbracket^i
\end{aligned}$$

Figure 4.14: Elimination of output **lasts** 🐔 [Lustre/NormLast/NormLast.v:225](#)

invariant below. H_2 must refine H_1 modulo substitution, and also refine H_1 directly for every non-**last** variable. Under these preconditions, we prove that H_2 gives a semantics to the transformed block.

Invariant 7 (Elimination of output **lasts** 🐔 [Lustre/NormLast/NLCorrectness.v:402](#))

$$\begin{aligned}
& \text{if } G, H_1, bs \vdash \text{blk} \\
& \text{and } H_1 \sqsubseteq_{\sigma} H_2 \quad \text{and } (\forall x \text{ vs}, H_1(x) \equiv \text{vs} \implies H_2(x) \equiv \text{vs}) \\
& \text{then } G, H_2, bs \vdash \llbracket \text{blk} \rrbracket_{\sigma}
\end{aligned}$$

Building a H_2 that respects these constraints is a bit more involved, since **last** and non-**last** variables are not transformed in the same way. We define H_2 by

$$\begin{aligned}
H_2(\text{last } x) &= H_1(\text{last } (\sigma^{-1}(x))) \\
H_2(x) &= \begin{cases} \text{if } x \in \sigma^{-1} \text{ then } H_1(\sigma^{-1}(x)) \\ \text{else } H_1(x) \end{cases}
\end{aligned}$$


It fits the requirements: H_2 contains the streams associated to (i) renamed **last** variables, (ii) renamed variables, and (iii) old variables. Using this history to instantiate the invariant concludes the proof of semantics preservation.

4.11.3 Stateless definitions for lasts


The final pass removes **last** variables defined by either **fbys** or node instances. For example, the equations **last** $x = 0$; $x = 0$ **fbf** (**last** $x + 1$) are not allowed. They are transformed into **last** $1x = 0$; $x = 0$ **fbf** (**last** $1x + 1$); $1x = x$. Equations with node instantiations are treated similarly.

The transformation function, presented in [figure 4.16](#), first analyzes the blocks of the node with the **stateful-defs** function, which returns the set of variables defined by stateful equations. The set of variables to be renamed is the intersection of this set with the set

$$\begin{aligned}
\text{stateful-defs } (x = e_0 \text{ fby } e_1) &\triangleq \{x\} \\
\text{stateful-defs } (xs = f(es)) &\triangleq \{x \mid x \in xs\} \\
\text{stateful-defs } (x = e) &\triangleq \emptyset \\
\text{stateful-defs } (\text{last } x = e) &\triangleq \emptyset \\
\text{stateful-defs } (\text{reset } blk \text{ every } x) &\triangleq \text{stateful-defs } blk
\end{aligned}$$

Figure 4.15: Stateful definitions  [Lustre/NormLast/NormLast.v:246](#)

$$\begin{aligned}
& \llbracket \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} \text{ let } \text{blks} \text{ tel} \rrbracket \triangleq \\
& \text{let } xs := \{x \mid \text{last } x \in \text{locs} \wedge x \in \text{stateful-defs } \text{blks}\} \text{ in} \\
& \text{let } \sigma := \{\text{last } x \mapsto \text{last } \$x \mid x \in xs\} \text{ in} \\
& \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} + [\sigma(x) \mid x \in xs] \\
& \text{let } \llbracket \text{blks} \rrbracket_{\sigma}; [\sigma(x) = x \mid x \in xs] \text{ tel} \\
& \llbracket xs = e \rrbracket_{\sigma} \triangleq xs = \llbracket e \rrbracket_{\sigma} \\
& \llbracket \text{last } x = e \rrbracket_{\sigma} \triangleq \text{last } \sigma(x) = \llbracket e \rrbracket_{\sigma} \\
& \llbracket \text{reset } blk \text{ every } x \rrbracket_{\sigma} \triangleq \text{reset } \llbracket blk \rrbracket_{\sigma} \text{ every } x
\end{aligned}$$

Figure 4.16: Unnesting of `last` definitions  [Lustre/NormLast/NormLast.v:272](#)

of local variables defined with `last`. From this set, the function generates fresh local variables that will be used with `last`; as usual, we build a substitution σ from old to new identifiers. This substitution is applied to sub-expressions and sub-blocks. Only `last` variables are renamed in expressions.

The recursive function treating blocks is more straightforward than the one of the previous pass: the variables at left of an equation are never renamed, because there is no useful case where we would like to keep a relation between the left and right sides of a stateful equation. Instead of renaming these left-hand-side variables, we introduce copy equations from old to new identifiers directly at the top level.

Correctness Despite the differences between this pass and the previous, the correctness invariant for this pass ends-up being identical to the one presented in [invariant 7](#). This makes sense for two reasons. First, as before, only `last` variables are renamed in expressions; other variables are still read and keep their original streams. Second, the direction of equations between old and new variables does not matter semantically. Indeed, in our semantic model, equations $x = y$ and $y = x$ are equivalent, because they are just constraints on the streams associated with x and y in the history.

Conclusion We have shown that each of the three transformation preserves the semantics of a program. To define the normalization of `last`, we compose them to get a pass that also preserves the semantics of a program. Our presentation of the semantic correctness proof for these passes omitted, as always, a lot of administrative details around the well-formedness of input programs. In the full Coq proof, these details turn out to

$$\llbracket x = (e_0 \text{ fby } e_1)^{ck} \rrbracket \triangleq \begin{cases} x = \text{if } xinit \text{ then } e_0 \text{ else } px \\ xinit = \llbracket \text{true} \rrbracket_{ck} \text{ fby } \llbracket \text{false} \rrbracket_{ck} \\ px = \llbracket def_{ty} \rrbracket_{ck} \text{ fby } e_1 \end{cases}$$

Figure 4.17: Normalization of `fby` equations 🐔 [Lustre/NormFby/NormFby.v:80](#)

be cumbersome, as we must also show that they are preserved by the three sub-passes. Having three separate proofs also means repeating a lot of boilerplate proof scripts. Combining the passes directly might avoid this, but it would probably make the overall proof much more complex.

4.12 Normalization of `fby` equations


As with `last` variables, there are additional constraints on the definition of `fby` equations. First, as indicated in [figure 4.2](#), `fby`s may only be initialized by constant expressions. Second, the outputs of the node may not be defined directly by a `fby`. The next compilation function normalizes the `fby` equations so that they respect these constraints.

4.12.1 Compilation Function

The compilation function is presented in [figure 4.17](#). It transforms an equation of the form $x = e_0 \text{ fby } e_1$, where e_0 is not already a sampled constant, into the three equations shown at right. The equation for `xinit` is only true at the first instant of presence of streams with clock type ck . Compiling $\llbracket c \rrbracket_{ck}$ adds `whens` around the constant c so that it has the same clock as the initial `fby`. The equation for `px` delays the stream associated with e . Its initial value is def_{ty} , an arbitrary constant of type ty . Since our language only manipulates boolean, integer, and floating-point values, it is always possible to choose such a constant (`false`, `0`, `0.0`). The value of the constant is never used in the definition of x . These three equations are in normal form because only sampled constants are used at left of the `fby`s. The subsequent transcription pass removes the `whens` from the constants.


Efficiency This schema is quite naive. For two equations containing `fby`s with the same clock type, it produces two identical initialization equations. For example, the normalization of $(x, y) = (x0, y0) \text{ fby } (y, x)$ would give rise to two equations identically defined by `true fby false`. This is a performance issue, because each `fby` requires its own state memory in the generated imperative code. Our first approach to avoid this inefficiency was to use the state monad to memoize and reuse generated initialization equations. However, this complicated the correctness proof, especially after the introduction of `reset` blocks in the language. Instead, we keep the naive approach of adding initialization equations at will, in the knowledge that the NLustre register deduplication optimization, presented in [section 5.2.3.3](#), will remove duplicates.

$$\begin{aligned} \text{fby-co } v_0 (\langle \rangle \cdot xs) &\triangleq \langle \rangle \cdot \text{fby-co } v_0 xs \\ \text{fby-co } v_0 (\langle v \rangle \cdot xs) &\triangleq \langle v \rangle \cdot \text{fby-co } v xs \end{aligned}$$

Figure 4.18: Coinductive fby function  [CoindStreams.v:1410](#)

4.12.2 Correctness

The core of the proof is to show that the semantics of an equation $x = e_0 \text{ fby } e_1$ is preserved in the new equations generated from it. As in previous proofs, the history is extended to incorporate the new variables. An additional difficulty is to reason from the semantics of the initial **fby** to show that expressions involving sampled constants, two new **fbys**, and an **if** construction each also have a semantics, and that their composition matches the original one.

Invariant 8 (Normalization of **fby** equations  [Lustre/NormFby/NFCorrectness.v:436](#))

$$\begin{aligned} &\text{if } G, H_1, bs \vdash eq \quad \text{and } G, \Gamma \vdash_{wc} eq \\ &\text{then } \exists H_2, H_1 \sqsubseteq H_2 \wedge G, H_2, bs \vdash [eq] \end{aligned}$$

Slow constants c^{ck} The stream associated to a constant c can be given directly. Obtaining the stream for a sampled constant $[c]_{ck}$ is more difficult because it requires associating the clock type ck with a stream. For instance, to prove that $c \text{ when } C1(b1) \text{ when } C2(b2)$ has a semantics, we must first establish that the clock $\bullet \text{ on } C1(b1) \text{ on } C2(b2)$ has one. Thankfully, the invariants of the instrumented semantic model, presented in [section 4.6](#) allow us to show the existence of this stream. Using these invariants and lemmas still complicate the proof, as they require that the compiled program be well clocked.

Delay equations We must obtain streams for the **fby** expressions that define x_{init} and px in the produced equations. To do so, we apply the coinductive **fby-co** function, presented in [figure 4.18](#), on the streams obtained for the slow constants or associated with the expression e_1 in the original **fby** equation. Note that this function is total because the **fby** is initialized by a constant, and therefore there is no alignment issue.

Choosing a value with if Finally, we must show that the initial equation $x = e_0 \text{ fby } e_1$ and its replacement $x = \text{if } x_{init} \text{ then } e_0 \text{ else } px$ denote the same stream. Applying rule inversion to the semantic predicate for the former gives $\text{fby } ys_0 \text{ } ys_1 \equiv zs$, that is, the relevant semantic operator applied to streams associated with, respectively, e_0 , e_1 , and x . From this and the results presented above, and a semantic operator for the conditional operator, we can show that the constructed streams fulfill the semantic relation associated with the **if** operator, and therefore give a semantics to this last equation.

4.13 Discussion and Related Works

4.13.1 Translation validation of synchronous dataflow programs

Auger [Aug13] describes a formally verified compiler for a Lustre-like language, based on translation validation. Each transformation pass is specified by a relation between source and target programs, implemented with two algorithms, and verified with two proofs. First, an unverified analysis of the source program produces a certificate, that is, a hint on how the program should be transformed. Then, an algorithm checks and applies the certificate to transform the source program. The algorithm may fail if the certificate, or source program, are ill-formed. If it succeeds, it is proven that the source and target programs are related, according to the specification of the pass. In some cases, the certificate may be the target program to generate itself, and the verified checker just has to check that it is indeed in relation with the source program. Finally, a separate proof establishes that two related programs have the same semantics.

Let's give an example to make this more concrete. The first transformation implemented is an unnesting pass, similar to the one we describe in [section 4.10](#), minus the distribution of operators which is implemented in a later pass. For this pass, the certificate is a list of names to introduce in the node, a list of new equations binding the unnested sub-expressions, and the list of old equations where sub-expressions have been unnested. This first list of equations can also be seen as a substitution from patterns (left of equation) to expressions (right of equation). To check that the certificate is correct, the checker simply applies this substitution in the list of old equations. The result should be equal to the list of equations of the source program. The relation that specifies this transformation is the equality modulo substitution: intuitively $e_1 \approx e_2$ if applying σ to e_1 produces e_2 . This relation is defined inductively over the syntax of expressions. It is then easy to prove that, if the substitution corresponds to valid equations, then e_1 and e_2 have the same semantics.

Auger describes two ways in which this approach shines. First, the checker is simple, and easy to verify against the relational specification of the pass. The proof of semantic correctness is also easier to verify from a relational specification, without being burdened by details of the implementation of an algorithm. By separating the proof in two, each part is made simpler. Second, since the certificate generator is not formally verified, it is easier to maintain and modify. In particular, this gives more freedom for experimenting with different code-generation schemes.

There are two main drawbacks to this approach. First, having to recheck the certificate causes a slight overhead in compilation-time; that being said, the checker algorithms are often simple and efficient. More importantly, while the compiler may never silently generate incorrect code, the checker might fail on a particular program, which would cause the compiler to abort. This may be either because of a bug in the certificate generator or because the checker cannot verify the certificate because it is too complex. This is not critical, but may still be problematic for a compiler used in production. The only way to mitigate these kinds of issues is with intensive testing. In Vélus, the two passes implemented using translation validation are elaboration, and scheduling (which

we discuss in the next chapter). We have made efforts to test these passes and have found some bugs in previous versions that led to the compiler aborting.

4.13.2 Generating Fresh Identifiers

4.13.2.1 Protecting the axiomatization

As discussed in [section 4.2](#), the use of axiomatized functions may be unsound, and introduce bugs in a compiler. In particular, the assumption made by Coq that all functions are observationally pure, that is, $\forall f x, (fx) = (fx)$, does not necessarily hold for axiomatized OCaml functions. Boulmé [\[Bou21\]](#) proposes a strategy to avoid this unsoundness. It consists in wrapping the type of a nondeterministic calculation in a monad type: “ $A \rightarrow ??B$ represents the type of impure functions from type A into type B ”. Type $??B$ is axiomatized, but may be interpreted as $B \rightarrow \mathbf{Prop}$, the type of predicates over B . Under this interpretation, $A \rightarrow ??B$ returns a predicate that characterizes all the possible results of evaluating the impure function. During extraction, $??B$ is treated like B . In other words, an OCaml function of type $A \rightarrow B$ implements an axiom of type $A \rightarrow ??B$.

From this type of impure computations, Boulmé defines the “may-return” relation of type $\rightsquigarrow: ??A \rightarrow A \rightarrow \mathbf{Prop}$, where $k \rightsquigarrow a$ states that “ k may return a ”. In addition, he axiomatizes and specifies the usual monadic operators with respect to the \rightsquigarrow relation.

- $\mathbf{ret}: A \rightarrow ??A$
such that $\mathbf{ret} x \rightsquigarrow y \implies x = y$
- $\mathbf{bind}: ??A \rightarrow (A \rightarrow ??B) \rightarrow ??B$
such that $\mathbf{bind} k_1 k_2 \rightsquigarrow b \implies \exists a, k_1 \rightsquigarrow a \wedge (k_2 a) \rightsquigarrow b$

Of course, any Coq function that uses the result of a non-deterministic computation becomes non-deterministic, and this monad propagates to the whole code-base, as monads do. However, the Vélus compiler passes that generate identifiers already use the **Fresh** monad defined earlier. It would be interesting to combine our approach with this non-determinism monad, to better protect the calls to `gensym`.

4.13.2.2 Generalized Monadic Approach

Nigron [\[Nig22, Chap.3\]](#) describes the use of a state monad to generate fresh identifiers. The state of the monad is represented by a record with two fields: a counter and a trail of already-generated identifiers; we did the same in our implementation. Unlike ours however, the state does not encode any invariant of these fields by construction. Instead, a monadic function is treated as an imperative program. Pre-and-post-conditions are given as Hoare triples with separation logic [\[Cha20\]](#). The non-duplication invariants on the state are treated by this logic.

This approach is used to reimplement the `SimplExpr` pass of `CompCert`, which simplifies the input language `CompCert-C` to the more restricted `Clight` language. This pass is specified in two ways: an executable, monadic function, and a relational specification. A first lemma establishes that the function respects the specification, and a second that

programs related by the specification have the same semantics. Monadic reasoning is only used for the first proof. The author states that transforming the original proof, which reasons on generated identifiers explicitly, into a proof using separation logic reduces proof size by around 35%. It is not obvious if following this approach would lead to the similar gains in Vélus. First, our use of dependent invariants for monadic states is another way to avoid administrative proofs, that would be redundant with the use of separation logic. Second, the correctness of our passes are proven directly on the function, and not using an intermediate relational specification of the transformation; we are not sure how straightforward the proofs would be without this separation.

Middle-End Compilation

In this final technical chapter, we describe the middle-end of the Vélus compiler, up to the translation to the imperative Obc language. The first section describes this language, its semantics, and the optimizations that may be defined at the imperative level. The remainder describes the compilation of normalized Lustre programs into Obc programs. The compilation scheme uses two other intermediate languages: Nlustré and Stc, each with its own semantic model. The sequence of compilation passes after our modifications is presented in [figure 5.1](#). The grayed-out passes are not discussed in this dissertation, as they are not impacted by our changes.

Since all of these languages and compilation passes have already been described in [[PLDI17](#); [POPL20](#); [Bru20](#)], we focus on the adaptations necessary to support the new features added in this thesis: the generalized `reset` and the efficient compilation of `last` variables. We repeat some of the central definitions and lemmas from previous work when necessary and otherwise simply refer to the relevant section or figure in the previous work.

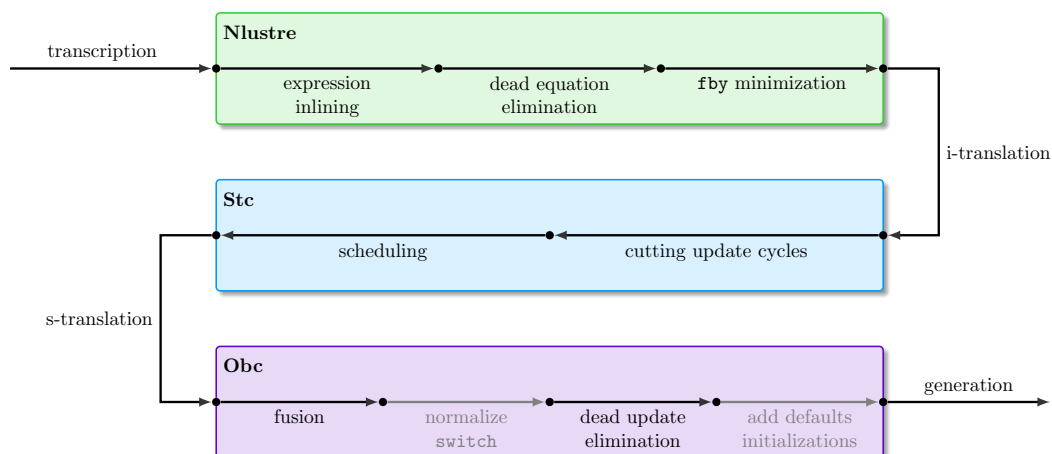


Figure 5.1: Architecture of the Vélus middle-end

```

class ::= class cls { statedec* instdec* method* }
statedec ::= state x : ty
instdec ::= instance x : cx
method ::= method mx ( var+ ) returns ( var+ ) var var* { s }
e ::= c | C | x | state(x) |  $\diamond$  e | e  $\oplus$  e
s ::= skip
   | x := e
   | state(x) := e
   | x* := cls(x).mx(e+)
   | s ; s
   | switch e { ( | C => s )+ }

```

Figure 5.2: Abstract Syntax of the Obc Language 🐔 [Obc/ObcSyntax.v:249](#)

5.1 The Obc target language

5.1.1 Syntax of Obc

Obc is a simple object-oriented language similar to the one introduced in [Bie+08, §4]. The full abstract syntax of Obc is presented in figure 5.2. An Obc class may contain **state** variables and **instances** of other objects. Each class exposes some **methods**, which take some inputs, return some outputs, and may update **state** variables and call methods of sub-**instances**. The body of a **method** is composed of a single statement.

The simplest Obc statement is **skip**, which does nothing but is useful for compilation. Statements that assign the value of an expression to a variable or state variable are syntactically distinct. An Obc expression is either an enumerated or scalar constant, an access to a variable or state variable, or an application of an operator on sub-expressions. Variables can also be assigned to values returned by a call to a method of a sub-instance. Sequencing statements $s_1; s_2$ means that s_1 is executed first, followed by s_2 . Finally, the **switch** contains an enumerated condition used to choose between its branches. In the compiler, it is possible that some branches of the **switch** do not contain a statement, in which case the execution falls back on a default branch. This facilitates an optimization that removes the last branching instruction from the generated code. For concision, we omit this feature from our presentation of Obc.

5.1.2 A compiled example

To introduce the compilation schemes described in this chapter, we give an example of the compilation of a normalized Lustre program to Obc. Recall the `drive_sequence` node presented in the introduction. Its normalized form is shown in figure 5.3, at top. We


```

node drive_sequence (step : bool) returns (mA, mB : bool)
var l$mB, l$mA : bool;
let
  mB = l$mB;
  l$mB = merge step
    (false => last l$mB when not step)
    (true => last l$mA when step);
  mA = l$mA;
  l$mA = merge step
    (false => last l$mA when not step)
    (true => not last l$mB when step);
  last l$mB = true;
  last l$mA = true;
tel

node motor(pause : bool) returns (ena, mA, mB : bool)
var step : bool;
let
  (ena, step) = feed_pause(pause);
  (mA, mB) = drive_sequence(step);
tel

```

```

class drive_sequence {
  state l$mB : bool;
  state l$mA : bool;

  method step(step : bool)
  returns (mA, mB : bool)
  var stc$l$mB : bool {
    stc$l$mB := state(l$mB);
    switch step {
    | false => state(l$mB) := state(l$mB)
    | true => state(l$mB) := state(l$mA)
    };
    switch step {
    | false => state(l$mA) := state(l$mA)
    | true => state(l$mA) := not stc$l$mB
    };
    mB := state(l$mB);
    mA := state(l$mA)
  }

  method reset() {
    state(l$mB) := true;
    state(l$mA) := true
  }
}

```

```

class motor {
  instance mA : drive_sequence;
  instance ena : feed_pause;

  method step(pause : bool)
  returns (ena, mA, mB : bool)
  var step : bool {
    ena, step := feed_pause(ena).step(pause);
    mA, mB := drive_sequence(mA).step(step)
  }

  method reset() {
    drive_sequence(motorA).reset();
    feed_pause(enable).reset()
  }
}

```

Figure 5.3: Compilation of drive_sequence and motor to Obc

also include the `motor` node that composes `drive_sequence` with the `feed_pause` node. This is not exactly the program produced by the front-end compiler. For readability, we have shortened the name of some variables and eliminated some aliases. The actual, step-by-step compilation of the `drive_sequence` node is detailed in [appendix B](#). The corresponding Obc code is presented underneath. Each `node` in the source program is compiled to a `class` in the Obc code. Each instantiation of a `node` in the source program is compiled to an `instance` in the Obc program. Each variable associated with a stateful construct in the source program (`last` or `fbv` equation) becomes a `state` variable of the `class`. A `class` generated from a Lustre `node` contains two `methods`. The first, `step`, takes the same inputs as the original `node`, executes one cycle of the dataflow program which updates the state, and returns the corresponding outputs. The second, `reset`, (re)initializes the `state` variables and sub-`instances` of the `class`.

Lustre equations are transformed into updates of variables or `state` variables. Node instantiations are transformed into calls to the `step` method of the corresponding instance. The control represented by sampling in Lustre is implemented by imperative `switch` statements on enumerated conditions. Most importantly, while the order of equations in the Lustre program does not matter, the statements of the Obc program have been *scheduled* into a sequence where each instruction only uses variables that were assigned before its evaluation.

5.1.3 Semantics of Obc

We now recall the formal semantics of Obc described in [Bru20, §4.1.2]. [Figure 5.4](#) presents the important rules of this big-step semantic model. Judgement $me, ve \vdash_{obc} e \Downarrow v$ indicates that expression e produces value v under memory me and environment ve . In Obc, the notions of explicit presences and absences from the dataflow language are not relevant. However, a value may be either defined, written $\lfloor v \rfloor$ or undefined, written $\lfloor \rfloor$. The distinction between the notions of presence and definedness becomes clear when looking at the first two rules of [figure 5.4](#). A local variable x can always be read from an environment ve , which we write $ve(x)$, but this may return an undefined value. In a program compiled from a dataflow source, the value of a variable should only be defined in cycles where the corresponding synchronous value is present. In other words, this notion of undefinedness encodes the absence of local variables. The behavior of state variables is stricter. Reading from a state variable x is only possible if its value is defined by the memory me . Indeed, since state variables are persistent through cycles, the value of `state`(x) is always defined if it has been initialized: it is the last value assigned to `state`(x).

The semantic judgment for Obc statements is written $P, me, ve \vdash_{obc} s \Downarrow me', ve'$, and states that under global context P , statement s updates memory me and environment ve to me' and ve' . The rules for variable and state variable assignment are central to this update. In both cases, if the expression at right of an assignment evaluates to a defined value v , then the value of x in the environment (resp. memory) is updated, which we note $ve\{x \mapsto v\}$ (resp. $me\{x \mapsto v\}$). When evaluating a sequence of two statements, the first one is evaluated in the original context ve, me , and the second in the updated

$$\begin{array}{c}
\frac{}{me, ve \vdash_{obc} x \downarrow ve(x)} \qquad \frac{me(x) = v}{me, ve \vdash_{obc} \mathbf{state}(x) \downarrow [v]} \\
\\
\frac{me, ve \vdash_{obc} e \downarrow [v]}{P, me, ve \vdash_{obc} x := e \downarrow me, ve\{x \mapsto v\}} \qquad \frac{me, ve \vdash_{obc} e \downarrow [v]}{P, me, ve \vdash_{obc} \mathbf{state}(x) := e \downarrow me\{x \mapsto v\}, ve} \\
\\
\frac{P, me, ve \vdash_{obc} s_1 \downarrow me_1, ve_1 \quad P, me_1, ve_1 \vdash_{obc} s_2 \downarrow me_2, ve_2}{P, me, ve \vdash_{obc} s_1; s_2 \downarrow me_2, ve_2} \\
\\
\frac{me, ve \vdash_{obc} e \downarrow [C_i] \quad P, me, ve \vdash_{obc} s_i \downarrow me', ve'}{P, me, ve \vdash_{obc} \mathbf{switch} e\{[C_i \Rightarrow s_i]^i\} \downarrow me', ve'}
\end{array}$$

Figure 5.4: Selected semantic rules for Obc 🐔 [Obc/ObcSemantics.v:110](#)

context ve_1, me_1 ; the final context ve_2, me_2 is the result of the evaluation of the second statement. Finally, the condition of a `switch` statement evaluates to a constructor C_i . The corresponding statement s_i is then evaluated. We do not present the rules for method calls, as they are not directly relevant to our work.

The definitions we have presented above are almost identical to the ones presented in earlier work. The only major difference is the extension of `if-then-else` into the more general `switch` statement; this was part of the work of L.Brun on enumerated types. We did not need to change Obc to support the generalized `reset` and `last` variables. This also means that we did not have to update the function that compiles Obc into Clight, nor its proof of semantics preservation.

5.1.4 Optimizations

The Obc code presented in [figure 5.3](#) can be optimized to reduce branching and eliminate redundant update instructions. Understanding these optimizations is key to understanding some of the design choices in the middle-end of Vélus.

Fusion of Conditionals A first optimization aims at reducing the number of `switch` statements in the program [[Bie+08](#)]. Indeed, each `switch` incurs branching in the generated assembly code, which may increase execution time. The transformation that reduces the number of `switches` is straightforward: two adjacent `switches` on the same condition are *fused* into one. In the case of [figure 5.3](#), the body of the `step` method of `drive_sequence` contains two `switches`; fusion produces the code of [listing 5.1](#), with only one `switch`. It turns out that the definition and verification of this pass is affected by our compilation scheme for `last` variables. Since the required changes are intertwined with the definition of the compilation from Stc to Obc, we will come back to this pass in [section 5.4.4](#).

```

switch step {
| false => state(l$mB) := state(l$mB); state(l$mA) := state(l$mA)
| true  => state(l$mB) := state(l$mA); state(l$mA) := not stc$l$mB
}

```

Listing 5.1: Obc code after fusion

Dead Update Elimination The second optimization removes useless `state` variable updates. In the program of [listing 5.1](#), there are two: `state(l$mA) := state(l$mA)` and `state(l$mB) := state(l$mB)`. Replacing these instructions by `skip` yields the program presented in [listing 5.2](#). All this pass does is remove instructions of the form `state(x) := state(x)`. The semantic correctness of this transformation is obvious, since such instructions overwrite the memory with the value that was already in it.

In this example, the instructions removed correspond to equations defining variables declared with `last` in the `false` branch of each `merge`. In turn, this corresponds to the equations that were automatically added by shared variable completion. The simplicity of this optimization is what motivates the inclusion of `last` variables throughout the compiler, in contrast to the possibility of replacing them with `fbys` directly.

```

class drive_sequence {
  state l$mB : bool;
  state l$mA : bool;

  method step(step : bool)
  returns (mA, mB : bool)
  var stc$l$mB : bool {
    stc$l$mB := state(l$mB);
    switch step {
    | false => skip; skip
    | true  => state(l$mB) := state(l$mA); state(l$mA) := not stc$l$mB
    };
    mB := state(l$mB);
    mA := state(l$mA)
  }

  method reset() {
    state(l$mB) := true;
    state(l$mA) := true
  }
}

```

Listing 5.2: drive-sequence step method after optimizations

$$\begin{aligned}
e &::= c \quad | \quad C \quad | \quad x \quad | \quad \text{last } x \quad | \quad \diamond e \quad | \quad e \oplus e \quad | \quad e \text{ when } C (x) \\
ce &::= \text{merge } x (C \Rightarrow ce)^+ \quad | \quad \text{case } e \text{ of } (C \Rightarrow ce)^+ \quad | \quad e \\
eq &::= x = ce \\
&\quad | \quad x = \text{reset } c \text{ fby } e \text{ every } x^* \\
&\quad | \quad \text{last } x = c \text{ every } x^* \\
&\quad | \quad x^+ = (\text{reset } f \text{ every } x^*) (e^+) \\
var &::= x : ty \text{ on } ck \\
nodedecl &::= \text{node } f (var^+) \text{ returns } (var^+) \text{ var } var^* \text{ let } eq^+ \text{ tel}
\end{aligned}$$

Figure 5.5: Abstract Syntax of the NLustre Language 🐔 NLustre/NLSyntax.v:24

5.2 NLustre: a normalized dataflow language

We now go back to the first intermediate language used in Vélus, NLustre. It is more or less a restricted form of the Lustre syntax described earlier. Its syntax is presented in [figure 5.5](#).

As in the normalized Lustre syntax presented in the previous chapter, stateless expressions are split in two levels. First, a simple expression may be a constant, variable, **last** variable, operation, or sampling by **when**. Second, a control expression may be either a **merge**, a **case**, or a simple expression. An additional constraint, modeled by the type-system, is that the branches of control expressions should be ordered by constructors, that is, they should appear in the same order as in the type declaration. This restriction simplifies the typing and semantic definitions of these intermediate languages, as well as later compilation passes.

The language does not contain structured blocks anymore. Instead, four types of equations are possible. First, equations with a control expressions at right. These equations are compiled to code that does not update any state. The next three equations are stateful: **fbby**, **last** initialization, and node call. All of these equations may be reset by a disjunction of **reset** conditions. These conditions are independent, and may be on different clocks. The list of **reset** conditions may also be empty.

Compared to [Bru20, Figure 2.5], this version of NLustre has three major differences. First, the presence of **last**, and **last** initialization equations. Second, the possibility of resetting **fbby** equations (as well as **last** initialization). Finally, the possibility of having multiple **reset** conditions on a single stateful equation.

5.2.1 Semantic Models

The NLustre language might seem redundant with the more general Lustre language, but having a separate, simpler AST has two benefits. First, it simplifies the definition and verification of dataflow optimizations. [Section 5.2.3](#) discusses two passes that optimize NLustre programs.

$$\frac{H, bs \vdash_{\omega} e \Downarrow vs \quad H, bs \vdash ck \Downarrow (\text{clock-of } vs)}{H, bs \vdash_{\omega} e^{ck} \Downarrow vs}$$

Figure 5.6: Coinductive semantics of annotated expressions

Second, it simplifies the definition of the semantics and therefore facilitates defining multiple semantic models for the language. Indeed, the semantics of NLustre are specified by three distinct relational models. These models are shown to be equivalent, and progress from a dataflow-based view of the language, to a transition-based view. This decomposes and simplifies the proof of correspondence with the semantics of later intermediate languages. These semantic models have already been presented in [POPL20; Bru20]. We only give a brief presentation of each below and highlight the changes necessary to integrate the resetting of `fb` and `last` variables.

5.2.1.1 Coinductive Semantics

The first semantic model uses coinductive streams and is similar to the one of Lustre. The semantic judgement for expressions is $H, bs \vdash_{\omega} e \Downarrow vs$. It does not require a global environment G because expressions can not contain node calls. Additionally, vs represents a single stream, as there are no lists of expressions in the normalized language. The rules defining this judgement are similar to the ones presented in [chapter 2](#).

To simplify correctness proofs, some of the rules in this model encode the clock-correctness property discussed in [section 3.5](#). The idea is similar to the instrumented semantic model for Lustre from [section 4.6](#), but implemented a bit differently. Instead of encoding the property at variable declarations, this model encodes them at equations, using a judgement for annotated expressions, which we write $H, bs \vdash_{\omega} e^{ck} \Downarrow vs$. The unique rule for this judgement is presented in [figure 5.6](#). An annotated expression e^{ck} produces stream vs if e without its annotation produces vs and the clock of vs corresponds to the interpretation of the clock type ck .

Our modifications to the compiler require only one change to the semantics of expressions: adding a rule for `last` variables. Just as in the Lustre semantics, we do this by adding the streams associated with `last x` to the history.

Equations require more work. Judgement $G, H, bs, \vdash_{\omega} eq$ states that history H respects the constraints induced by equation eq . The judgement is defined by four rules, one for each type of equation. Our modifications did not affect the rules for stateless equations and node calls presented in [Bru20, Figure 2.7]. We will focus on the rules for `fb` and `last` initialization, which are both defined using the `fb-co` delay operator. This operator was already present in [Bru20, Definition 2.3.3]. We recall the details in [definition 6](#). It takes as input an initialization value v_0 and a stream vs . Whenever the value of vs is present, v_0 is produced and the value at head of vs is kept as the new initialization value. Contrary to the operator used in Lustre, which takes two streams which must have the

xs	$\langle \rangle$	1	$\langle \rangle$	2	$\langle \rangle$	1	$\langle \rangle$	$\langle \rangle$	3	$\langle \rangle$	4	...
rs	F	F	F	T	F	F	T	F	F	F	F	...
$\text{reset-co}_F^0 xs rs$	$\langle \rangle$	1	$\langle \rangle$	0	$\langle \rangle$	1	$\langle \rangle$	$\langle \rangle$	0	$\langle \rangle$	4	...

Figure 5.7: Example behavior of reset-co

$$\frac{
 \begin{array}{l}
 H, bs \vdash_{\infty} e^{ck} \Downarrow vs \\
 \forall i, H(x_i) \equiv xs_i \quad \text{bools-ofs } [xs_i]^i \equiv rs \quad H(x) \equiv \text{reset-co}_F^c (\text{fby-co } c \text{ vs}) rs
 \end{array}
 }{
 G, H, bs \vdash_{\infty} x =_{ck} \text{reset } c \text{ fby } e \text{ every } [x_i]^i
 }$$

$$\frac{
 \begin{array}{l}
 H(x) \equiv vs \\
 \forall i, H(x_i) \equiv xs_i \quad \text{bools-ofs } [xs_i]^i \equiv rs \quad H(\text{last } x) \equiv \text{reset-co}_F^c (\text{fby-co } c \text{ vs}) rs
 \end{array}
 }{
 G, H, bs \vdash_{\infty} \text{last } x = c \text{ every } [x_i]^i
 }$$

Figure 5.8: Coinductive semantics for stateful equations 🐔 NLustre/NLCoindSemantics.v:281

same clock as input, this operator is total. In Coq, we define it as a coinductive function that produces a stream.

To define the semantics of the new resettable **fby** and **last**, we also define an operator **reset-op** that resets a stream, that is, that inserts the initial value v_0 every time the reset signal is **true**. Additionally, a reset must occur even if it is triggered in an instant when the value is absent. The operator **reset-co** in [definition 6](#) implements this behavior: if a **true** arrives on the rs stream while xs is absent, the pending **true** is passed to the corecursive call to **reset-co**, and will trigger a reset when xs is present again. An example of this behavior is illustrated in [figure 5.7](#).


Definition 6 (fby-co and reset-co operators 🐔 NLustre/NLCoindSemantics.v:192)

$$\begin{aligned}
 \text{fby-co } v_0 (\langle \rangle \cdot xs) &\triangleq \langle \rangle \cdot \text{fby-co } v_0 xs \\
 \text{fby-co } v_0 (\langle v \rangle \cdot xs) &\triangleq \langle v \rangle \cdot \text{fby-co } v xs \\
 \text{reset-co}_{r_0}^{v_0} (\langle \rangle \cdot xs) (r \cdot rs) &\triangleq \langle \rangle \cdot \text{reset-co}_{(r_0 \vee r)}^{v_0} xs rs \\
 \text{reset-co}_{r_0}^{v_0} (\langle v \rangle \cdot xs) (r \cdot rs) &\triangleq \langle \text{if } (r_0 \vee r) \text{ then } v_0 \text{ else } v \rangle \cdot \text{reset-co}_F^{v_0} xs rs
 \end{aligned}$$

Since **fby-co** and **reset-co** are both total functions, we can compose them to give a semantics to both **fby** and **last** equations, as presented in [figure 5.8](#). The partial **bools-ofs** function transforms a list of value streams that only contain boolean values into a stream of the point-wise disjunction of these booleans.

The two rules are similar: they relate the semantics of the initialization constant, the stream associated with the delayed expression/variable, and the delayed and initialized stream. This reflects how closely related the **fby** and **last** constructions are.

$$\frac{\forall n, (H(n)), (bs(n)) \vdash_{inst} e \downarrow (vs(n))}{H, bs \vdash_{ind} e \Downarrow vs}$$


Figure 5.9: Lifting instantaneous semantics  [IndexedStreams.v:447](#)

5.2.1.2 Indexed Semantics

In the indexed semantic model, streams are represented as functions of type `nat -> svalue`, where `nat` is the type of Peano natural numbers. Since these streams are just functions, the n th element of stream xs is simply $xs(n)$.

Furthermore, an expression is not related to an infinite stream of values, but rather to an instantaneous synchronous value. These instantaneous semantics are specified by the judgement $R, b \vdash_{inst} e \downarrow v$, where R is an instantaneous environment that maps a variable name to a synchronous value, b is a boolean instantaneous clock, and v is a synchronous value produced by the expression. The rules defining this judgement are presented in [Bru20, Figure 2.8]. We simply extend them with a rule for `last` expressions; again, this is treated by adding the instantaneous `last` values of variables in the environment R . Using instantaneous semantics is possible because, since expressions are stateless, the value of an expression at an instant only depends on the context at that instant. For stateful equations, the semantic model still depends on the history of the node. The instantaneous semantics of expressions are lifted to infinite stream semantics by the rule presented in [figure 5.9](#). History H and streams bs and vs are indexed to give the instantaneous semantics for the n th instant.

The semantic rules for equations are similar to the coinductive ones, with the semantic operators replaced by their indexed counterparts. The definition for the indexed `fby-ind` was presented in [Bru20, Definition 2.3.7]. In the same style, we propose an indexed `reset-ind` operator in [definition 7](#). This operator returns a stream that is equal to its input xs , except when xs is present and a reset must be performed. The `do-reset` auxiliary function specifies that there is a “pending” reset at instant n either if $rs(n)$ is `true`, or if a `true` arrived since the previous presence of xs . Formally, `do-reset xs rs n = true` if and only if there exists $m \leq n$ such that $rs(m) = \text{true}$ and $\forall k \in [m; n[, xs(k) = \langle \rangle$.

Definition 7 (Indexed specification of `reset-ind`  [NLustre/NLIndexedSemantics.v:67](#))

$$\begin{aligned} \text{do-reset } xs \text{ } rs \ 0 &\triangleq rs(0) \\ \text{do-reset } xs \text{ } rs \ (n + 1) &\triangleq rs((n + 1)) \vee ((xs \ n) = \langle \rangle \wedge \text{do-reset } xs \text{ } rs \ n) \\ \text{reset-ind}^{v_0} \text{ } xs \text{ } rs &\triangleq \lambda n. \begin{cases} \text{if } xs(n) = \langle \rangle & \text{then } \langle \rangle \\ \text{if } xs(n) = \langle x \rangle \wedge \text{do-reset } xs \text{ } rs \ n & \text{then } \langle v_0 \rangle \\ \text{if } xs(n) = \langle x \rangle & \text{then } \langle x \rangle \end{cases} \end{aligned}$$

Correspondence between coinductive and indexed semantics These two semantic models are equivalent. However, the compiler correctness proof only requires establishing

that the indexed semantics simulates the coinductive semantics, as stated by the lemma below. In the premise, the input and output streams are coinductive. To state the conclusion, they must be converted to indexed streams; this conversion uses the indexing operation on coinductive streams defined in [listing 2.9 \(page 26\)](#).

Lemma 14 (Coinductive to indexed semantics 🐔 [NLustre/NLCoindToIndexed.v:601](#))

if $G \vdash_{co} f([xs_i]^i) \Downarrow [ys_j]^j$ **then** $G \vdash_{ind} f([\text{tr-Stream } xs_i]^i) \Downarrow [\text{tr-Stream } ys_j]^j$
where $\text{tr-Stream } xs \triangleq \lambda n.(xs \# n)$

The proof of this result, already present in earlier versions of Vélus, is largely unchanged by our modifications to the NLustre language and its semantic models. The only novel obligation concerns the correspondence between the `reset-co` and `reset-ind` operators. The lemma below states that the conversion of the stream produced by `reset-co` is equivalent to the application of `reset-ind` to converted streams.

Lemma 15 (Coinductive to indexed reset 🐔 [NLustre/NLCoindToIndexed.v:201](#))

$\text{tr-Stream } (\text{reset-co}_F^{v_0} xs rs) \approx \text{reset-ind}^{v_0} (\text{tr-Stream } xs) (\text{tr-Stream } rs)$
where $xs \approx ys$ **iff** $(\forall n, xs(n) = ys(n))$

5.2.1.3 Indexed Semantics with Memory

The final semantic model extends the indexed model by adding explicit state handling. This model was designed as a bridge between the semantics of NLustre and that of the next intermediate language.

The new elements in this semantic model are “memories”. An instantaneous memory has two parts: (i) an environment storing the values of state variables, and (ii) an environment storing the memories of sub-node instances. We mostly discuss the former, as it is the most relevant for the `fby` and `last` constructs. The semantics of nodes is given against a stream of instantaneous memories, which we note M . We write $M_n(x)$ for looking up the value of x at the n th instant in memory M , which is a partial function. The memory is not the same as the history, for two reasons. First, the memory only tracks the values of variables associated with stateful constructs (`fby`, `last`). Second, the memory contains values, not synchronous values. The chronogram in [figure 5.10](#) illustrates this distinction, for an execution of equation $y = 0 \text{ fby } (y + x)$. Memory $M(y)$ starts with the value at left of the `fby`, and keeps it until x becomes present. In the next cycle, it updates to the value of $y + x$. In a sense, the memory stores the “next” value of y . It corresponds to an imperative behavior where the value of y would be updated at the end of an instant, so that it may be read in the next instant.

In this model, the semantic judgement for nodes is written $G, M \vdash_{mem} f(xs) \Downarrow ys$. It exposes the memory M of the node. In the next intermediate languages, the memory is an object that is directly manipulated during the execution of the program. It is passed as input, updated, and returned as an output. Exposing the memory in the semantic rules is a first step in this direction.

x	⟨⟩	1	⟨⟩	⟨⟩	2	4	⟨⟩	5	...
M(y)	0	0	1	1	1	3	7	7	...
y	⟨⟩	0	⟨⟩	⟨⟩	1	3	⟨⟩	7	...

Figure 5.10: Difference between history and memory, for $y = 0 \text{ fby } (y + x)$

r	F	F	F	T	F	F	F	T	F	F	...
x	⟨⟩	1	2	4	⟨⟩	1	2	⟨⟩	⟨⟩	1	...
M(y)	0	0	1	3	4	4	5	7	0	0	...
y	⟨⟩	0	1	0	⟨⟩	4	5	⟨⟩	⟨⟩	0	...

Figure 5.11: Example trace for $y = \text{reset } 0 \text{ fby } (x + y) \text{ every } r$

Our modification to this semantic model amounts to defining a semantic operator for resetting stateful constructs. Resetting the value of a stream should also reset the value in memory; this corresponds to the expected imperative behavior. Unlike in the previous model, we did not find a way to cleanly separate the `fby` and `reset` operators, or to define them as functions. This is because it is difficult to untangle their effects, and constraints, on the memory. We instead follow the approach of [Bru20, Figure 3.7.a], and provide a `mfbyreset` relation between the streams manipulated by a resettable `fby`, and the memory.

Definition 8 (Resettable `fby` in memory semantics 🐔 [NLustre/NLMemSemantics.v:77](#))

$$\begin{aligned}
 & \text{mfbyreset}_{v_0}^x M \text{ } xs \text{ } rs \approx ys \\
 & \text{iff } M_0(x) = v_0 \\
 & \text{and } \forall n, \begin{cases} \text{if } xs(n) = \langle v \rangle \wedge rs(n) = F \text{ then } M_{n+1}(x) = v & \wedge ys(n) = \langle M_n(x) \rangle \\ \text{if } xs(n) = \langle \rangle \wedge rs(n) = F \text{ then } M_{n+1}(x) = M_n(x) & \wedge ys(n) = \langle \rangle \\ \text{if } xs(n) = \langle v \rangle \wedge rs(n) = T \text{ then } M_{n+1}(x) = v & \wedge ys(n) = v_0 \\ \text{if } xs(n) = \langle \rangle \wedge rs(n) = T \text{ then } M_{n+1}(x) = v_0 & \wedge ys(n) = \langle \rangle \end{cases}
 \end{aligned}$$

This definition specifies how the value associated with x is updated in memory M . Value v_0 comes from the initialization constant, xs is the stream of the expression at right of the `fby`, rs is the reset stream, and ys is the resulting stream. The relation constrains $M_0(x)$ to be the initial value v_0 . For each instant n , there are four possibilities, depending on the presence of $xs(n)$ and the value of $rs(n)$. Let's first consider the two cases where no reset occurs. If $xs(n)$ is present, its value is put in memory for the next instant, and the current value in memory is produced. If it is absent, then the value in memory is kept for the next instant. If a reset happens, the changes are more involved. If $xs(n)$ is present, the reset is processed immediately: v_0 is produced on the output stream, but the value of $xs(n)$ is stored in memory. If it is absent, then the initial value v_0 overrides the value in memory. An example trace for equation $y = \text{reset } 0 \text{ fby } (x + y) \text{ every } r$ is presented in [figure 5.11](#). The first reset applies to y directly, but the second applies to its memory at the next instant.

We do not present the formal semantic rules for the **fb**y and **last** initialization equations here. They closely follow their counterparts from the coinductive and indexed models, with the composition of **fb**y and **reset** operators replaced by the **mfbyreset** relation.

Correspondence with the indexed model The memory semantic model has been proven equivalent to the indexed semantic model in both directions. Going from the model with memory to the model without memory is direct: it suffices to erase the memory constraints. The other direction is necessary to the correctness proof of the compiler and more involved. To go from the indexed model to the memory model, we must exhibit the existence of a memory stream that satisfies all the semantic constraints. The overall lemma is presented in [lemma 16](#).

Lemma 16 (Indexed to memory semantics 🐔 [NLustre/NLMemSemantics.v:695](#))

$$\text{if } G \vdash_{md} f(xss) \downarrow yss \text{ then } \exists M, G, M \vdash_{mem} f(xss) \downarrow yss$$

The overall structure of our updated proof follows the one discussed in [[Bru20](#), §3.3.1.3]. It proceeds by induction on the list of equations, building the memory stream variable-by-variable. In the case of **fb**y or **last** equations, the proof constructively provide the values in memory at each instant (see line $M(y)$ in [figure 5.11](#)) from the existing synchronous streams. To do so, we define two intermediate constructive operators **fb**yreset and **hold**reset, presented in [definition 9](#).

Definition 9 (**hold**reset and **fb**yreset 🐔 [NLustre/NLIndexedSemantics.v:298](#))

$$\begin{aligned} \text{holdreset}^{v_0} \ xs \ rs \ 0 &\triangleq v_0 \\ \text{holdreset}^{v_0} \ xs \ rs \ (n+1) &\triangleq \begin{cases} \text{if } xs(n) = \langle x \rangle & \text{then } x \\ \text{if } xs(n) = \langle \rangle \wedge rs(n) = \text{F} & \text{then } \text{holdreset}^{v_0} \ xs \ rs \ n \\ \text{if } xs(n) = \langle \rangle \wedge rs(n) = \text{T} & \text{then } v_0 \end{cases} \\ \text{fb}yreset^{v_0} \ xs \ rs &\triangleq \lambda n. \begin{cases} \text{if } xs(n) = \langle \rangle & \text{then } \langle \rangle \\ \text{if } xs(n) = \langle x \rangle \wedge rs(n) = \text{F} & \text{then } \text{holdreset}^{v_0} \ xs \ rs \ n \\ \text{if } xs(n) = \langle x \rangle \wedge rs(n) = \text{T} & \text{then } v_0 \end{cases} \end{aligned}$$

In particular, **fb**yreset is equivalent to composing **fb**y-ind and **reset**-ind, as stated below. The proof of this lemma requires an induction on the index of streams, and some intricate case analysis on the values of xs and rs .

Lemma 17 (Composing **fb**y-ind and **reset**-ind explicitly 🐔 [NLustre/NLIndexedSemantics.v:306](#))

$$\text{fb}yreset^{v_0} \ xs \ rs \approx \text{reset-ind}^{v_0} (\text{fb}y\text{-ind}^{v_0} \ xs) \ rs$$

In the proof of memory existence, we use this form of the stream. We then use [lemma 18](#) to show that the **mfbyreset** relation holds. This lemma is almost immediate by unfolding of the definitions of indexed stream equivalence (\approx), **mfbyreset** and **fb**yreset, and by case analysis on $xs(n)$ and $rs(n)$.

$$\begin{array}{lcl}
[c]^e & \triangleq & c \\
[C]^e & \triangleq & C \\
[x]^e & \triangleq & x \\
[\text{last } x]^e & \triangleq & \text{last } x \\
[\diamond e_1]^e & \triangleq & \diamond [e_1]^e \\
[e_1 \oplus e_2]^e & \triangleq & [e_1]^e \oplus [e_2]^e \\
[e \text{ when } C(x)]^e & \triangleq & [e]^e \text{ when } C(x) \\
[\text{merge } x [(C_i \Rightarrow e_i)]^i]^{\text{ce}} & \triangleq & \text{merge } x (\text{sort } [(C_i \Rightarrow [e_i]^{\text{ce}})]^i) \\
[(\text{case } e \text{ of } [(C_i \Rightarrow e_i)]^i)_{ty}^{\text{ck}}]^{\text{ce}} & \triangleq & \text{case } [e]^e \text{ of } (\text{sort } [(C_i \Rightarrow [e_i]^{\text{ce}})]^i) \\
[e]^{\text{ce}} & \triangleq & [e]^e
\end{array}$$

Figure 5.12: Transcription of expressions 🐔 [Transcription/Tr.v:109](#)

Lemma 18 (Building memory semantics for `fby` 🐔 [NLustre/NLMemSemantics.v:572](#))

if $(\forall n, M_n(x) = \text{holdreset}^{v_0} \text{ } xs \text{ } rs \text{ } n)$ then $\text{mfbyreset}_{v_0}^x M \text{ } xs \text{ } rs \approx \text{fbyreset}^{v_0} \text{ } xs \text{ } rs$

5.2.2 Transcription: From Lustre to NLustre

The transcription pass translates the Lustre AST into the NLustre AST. The proof of correctness for this transformation targets the coinductive semantics of NLustre, which is the most similar to the Lustre source semantics. In earlier versions of Vélus [EMSOFT21], the transcription algorithm was trivial, as the program was already compatible with the NLustre representation. For our more expressive language, both expressions and blocks need to be transformed.

5.2.2.1 Transcription of Expressions


The functions that transcribe expressions are presented in [figure 5.12](#). They both take as input a Lustre expression. Function $[e]^e$ transcribes it into a simple expression, and $[e]^{\text{ce}}$ into a control expression. Both of these functions are partial: they fail for Lustre expressions that are not compatible with the syntax of simple or control expressions, respectively. In Coq, these functions return in the `Error` monad.

The transcription of simple expressions is trivial. The transcription of `merge` and `case` is more involved. Indeed, in NLustre, the branches of control expressions (`merge`, enumerated `case`) must be ordered by constructors, that is, they must be in the same order as in the type declaration. Since this restriction is not present in the source Lustre language, the transcription must sort these branches. To do so, the function uses `Sorting.Mergesort`, provided by the Coq standard library.

5.2.2.2 Transcription of Blocks


The function that transcribes Lustre blocks into NLustre equations is shown in [figure 5.13](#). For a stateless equation, it simply transcribes the control expression. The function

$$\begin{aligned}
 [c]^c &\triangleq c \\
 [C]^c &\triangleq C \\
 [sc \text{ when } C(x)]^c &\triangleq [sc]^c \\
 [x = ce]_{xrs} &\triangleq x = [ce]^{ce} \\
 [\text{reset } blk \text{ every } xr]_{xrs} &\triangleq [blk]_{(xr :: xrs)} \\
 [x = sc \text{ fby } e]_{xrs} &\triangleq x = \text{reset } [sc]^c \text{ fby } [e]^e \text{ every } xrs \\
 [\text{last } x = sc]_{xrs} &\triangleq \text{last } x = [sc]^c \text{ every } xrs \\
 [xs = f(es)]_{xrs} &\triangleq xs = (\text{reset } f \text{ every } xrs)([es]^e) \\
 [xs = (\text{reset } f \text{ every } xr)(es)]_{xrs} &= xs = (\text{reset } f \text{ every } (xr :: xrs))([es]^e)
 \end{aligned}$$

 Figure 5.13: Transcription of blocks  [Transcription/Tr.v:258](#)

must also flatten the Lustre `reset` blocks into `reset` conditions for the stateful NLustre equations. To do so, the condition of each traversed `reset` block is accumulated into the extra parameter `xrs`. When encountering a `fbby`, a `last` equation, or a node instantiation, `xrs` is added to the `reset` conditions of the transformed equation. In the case of `fbby` and `last` equations, the sampled initialization constant is simplified into either a scalar or enumerated constant.

The invariant below states correctness for the transcription of blocks. Its conclusion states that block `blk` compiled with `xrs` has a semantics under history H . Here, `xrs` is the list of the conditions of the `reset` blocks surrounding `blk` in the source program. Since `blk` is located under `reset` blocks, its source semantics must be given under H masked by a reset stream `rs`, which is the disjunction of all boolean streams associated with the conditions `xrs`. This invariant is intricate because in Lustre, `reset` blocks are nested, while in NLustre, individual equations are reset. The invariant bridges the two representations.

Invariant 9 (Transcription of blocks  [Transcription/Correctness.v:1545](#))

$$\begin{aligned}
 &\mathbf{if} \quad \forall i, H(xrs_i) \equiv rss_i \quad \mathbf{and} \quad \mathbf{bools\text{-}ofs} \quad rss \equiv rs \\
 &\mathbf{and} \quad \forall k, G, (\text{mask}^k \quad rs \quad (H, bs)) \vdash blk \\
 &\mathbf{then} \quad G, H, bs, \vdash_{co} [blk]_{xrs}
 \end{aligned}$$

The proof proceeds by induction on the syntax of Lustre blocks and normalized equations. Each of the four major syntactic cases handled by the block compilation function (stateless equation, `fbby/last`, node call, nested `reset` block) raises a specific difficulty in the correctness proof.

Case 1: Stateless equations Compiling a stateless equation ignores the reset conditions. To prove that the scheme is correct, we must prove that the semantics of masking can also be ignored for a stateless NLustre expression.

Lemma 19 (Unmasking of stateless expressions 🐔 [Transcription/Correctness.v:944](#))

$$\mathbf{iff} \quad (\forall k, (\mathbf{mask}^k rs (H, bs)) \vdash_{\infty} e \Downarrow vs) \quad \mathbf{then} \quad H, bs \vdash_{\infty} e \Downarrow vs$$

This lemma is stated on the coinductive semantics of NLustre, but it is easier to prove in the indexed semantics, in which we can reason about the n th execution cycle. The `mask` operator is simplified using an equivalent, indexed definition based on the `count` function that returns the number of `true` values along a stream.

Lemma 20 (Indexed specification of `mask` 🐔 [CoindStreams.v:2427](#))

$$(\mathbf{mask}^k rs xs) \# n = \begin{cases} xs \# n & \text{if } (\mathbf{count}_0 rs) \# n = k \\ \langle \rangle & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathbf{count}_k (\mathbf{F} \cdot rs) &\triangleq k \cdot \mathbf{count}_k rs \\ \mathbf{count}_k (\mathbf{T} \cdot rs) &\triangleq (k + 1) \cdot \mathbf{count}_{(k+1)} rs \end{aligned}$$

To prove [lemma 19](#), we must show that e has a semantics under H at any instant n . Instantiating the hypothesis with $k = (\mathbf{count}_0 rs) \# n$ makes the masking transparent for instant n , which completes the proof.

Case 2: `fbym` and `last` For the case of `fbym` equations, we must show the equivalence between the masked `fbym` operations from the Lustre semantics, and the composition of the `fbym-co` and `reset-co` operations from the NLustre semantics. This correspondence is stated below: if the Lustre `fbym` operator can be applied on all maskings of xs by rs , with constant initial value v_0 , then we can give a semantics to the composition of `fbym-co` and `reset-co` with these same streams and initial value.

Lemma 21 (Masked `fbym` to reset of `fbym-co` 🐔 [Transcription/Correctness.v:661](#))

$$\begin{aligned} \mathbf{if} \quad &\forall k, \mathbf{fbym} (\mathbf{const} (\mathbf{clock-of} (\mathbf{mask}^k rs xs)) v_0) (\mathbf{mask}^k rs xs) \equiv (\mathbf{mask}^k rs ys) \\ \mathbf{then} \quad &ys \equiv \mathbf{reset-co}_F^{v_0} (\mathbf{fbym-co} v_0 xs) rs \end{aligned}$$

The proof of this lemma also exploits the correspondence between coinductive and indexed specification. Specifically, we use the indexed specification of \equiv from the Coq standard library, presented below. Two streams are coinductively equivalent if they are point-wise equivalent.

Lemma 22 (Indexed specification of `EqSt`)

$$xs \equiv ys \quad \mathbf{iff} \quad (\forall n, xs \# n = ys \# n)$$

The proof then proceeds by induction on n , and by case-analysis on the heads of the xs and rs streams. We use some rewriting lemmas to simplify the applications of coinductive operators (`mask`, `const`, etc...).

Case 3: Node instantiation For node instantiations that do not yet have a reset condition, the proof is straightforward, since the coinductive semantics of NLustre use the same `mask` operator as that of Lustre. If the source equation already has a reset condition, there is an additional complication. The masking of the source history must be composed with that of the inputs and outputs of the instantiation. We discuss this composition problem in more details in the paragraph on nested `reset` blocks.

Case 4: Reset blocks Finally, we must prove that [invariant 9](#) is preserved when traversing a `reset` block. In this case, the reset condition is accumulated in the list `xrs`. Therefore, the application of the induction hypothesis to the sub-block must take into account this new condition. Skipping some of the bookkeeping of the proof, if rs_1 is the stream associated with the reset conditions passed as parameters, and rs_2 is the stream associated with the reset condition of the block being compiled, reconstructing the inductive invariant for the sub-block requires proving the lemma below.

Lemma 23 (Decomposition of `mask` 🐔 [Transcription/Correctness.v:1259](#))

$$\forall rs_1 rs_2 k, \exists k_1 k_2, \text{mask}^k (rs_1 \vee rs_2) xs \equiv \text{mask}^{k_2} rs_2 (\text{mask}^{k_1} rs_1 xs)$$

It states that, for any instance k of the masking of a stream xs by a disjunction of reset streams $rs_1 \vee rs_2$, there are two indexes k_1 and k_2 such that the composition of the masking by k_1/rs_1 and k_2/rs_2 produces the same instance. In the indexed setting, k_1 and k_2 must be chosen such that $(\text{mask}^{k_2} rs_2 (\text{mask}^{k_1} rs_1 xs)) \# n = (\text{mask}^k (rs_1 \vee rs_2) xs) \# n$. This equation is simplified using the indexed specification of `mask` from [lemma 20](#).

Finding k_1 and k_2 constructively is not possible. Indeed, it would require a procedure that, given rs and k , decides if there exists n such that $(\text{count}_0 rs) \# n = k$. This would be equivalent to writing an `index-of` function which finds the index of a given value in a stream. In Coq, this is not possible, because streams are infinite, and therefore the `index-of` function would not necessarily terminate. We have not found any constructive way of sidestepping this issue. Instead, we use the excluded middle principle from classical logic. By applying this axiom, we prove the lemma below: either there exists an n such that the count reaches k at index n , or there is none.

Lemma 24 (Excluded middle applied to `count` 🐔 [Transcription/Correctness.v:1167](#))

$$\forall rs k, (\exists n, (\text{count}_0 rs) \# n = k) \vee (\forall n, (\text{count}_0 rs) \# n \neq k)$$

We apply this lemma to k and $rs_1 \vee rs_2$. In the first case, we can instantiate $k_1 = (\text{count}_0 rs_1) \# n$ and $k_2 = (\text{count}_0 rs_2) \# n$. The second case implies that k is too big, that is, there are only a finite number of `true`s in $rs_1 \vee rs_2$, and k is bigger than this number. Therefore, k is bigger than the number of `true`s in both rs_1 and rs_2 . Choosing any k_1 and k_2 that are superior or equal to k results in an equivalent opaque masking.

We are somewhat disappointed to need classical logic in this proof, while most of the other proofs in the compiler only use constructive reasoning. In addition, we have seen in

section 4.9.3 that the treatment of `reset` blocks also requires the axiom of choice. Both of the issues stem from the use of universal quantification in the semantic rules for `reset` blocks, which prevents constructive reasoning. We have not found a formalization for the semantics of `reset` blocks that does not need this quantification and integrates with the source semantics of Vélus. For now, our conclusion is that since our semantics is based on relational predicates that manipulate infinite objects, some non-constructive reasoning may be unavoidable. This is the same conclusion reached in [LG09], where some of the proofs on a coinductive semantic model require classical reasoning.

5.2.3 NLustre Optimizations

The AST and semantic definitions of NLustre are simpler than those of Lustre. This simplicity facilitates the definition and verification of optimizations on dataflow programs. In this section, we describe three dataflow optimizations implemented in Vélus.

5.2.3.1 Expression Inlining

This first pass is not an optimization in-and-of itself. Instead, it facilitates later optimizations. The *expression inlining* pass consists in replacing local variables defined by simple expressions with their definition. For instance, the equations $y = x$; $x = x1 + x2$ can be rewritten into $y = x1 + x2$; $x = x1 + x2$. The equation defining x is not removed by this pass, but will be removed by the following dead equation elimination pass. The pass applies this transformation for any local variable x such that either x is used only once in the node or x is defined by an expression that does not induce any computation: either a constant or variable, possibly sampled by `when`, but not applications of arithmetic or logic operators. These criteria guarantee that calculations are never duplicated.

This transformation has several benefits. First, it reduces the number of variables and equations, and therefore assignments in the generated Obc/C program. This does not necessarily make the generated code more efficient, because the register allocation used in the C compiler usually eliminates useless assignments. However, it does make the code more readable. More importantly, this transformation facilitates the elimination of dead updates implemented in Obc. Indeed, suppose the equations $x = \text{merge } c \text{ (true} \Rightarrow x\$1) \text{ (false} \Rightarrow x\$2)$; $x\$2 = \text{last } x \text{ when not } c$. Compiling these equations to Obc generates the statements $x\$2 = \text{state}(x)$; $\text{state}(x) = x\$2$. If we apply the transformation, and replace $x\$2$ by its definition, the generated Obc statement would simply be $\text{state}(x) := \text{state}(x)$, which can be optimized away easily. This example is relevant, because this is exactly the shape of the equations generated by our compiler for a `switch` block, such as the one used in the `drive_sequence` node. This transformation is therefore necessary in order to easily compile away the dead updates generated from partial definitions in the source program.

$$\begin{aligned}
\text{Free}(x) &\triangleq \{x\} \\
\text{Free}(\text{last } x) &\triangleq \{\text{last } x\} \\
\text{Free}(\diamond e_1) &\triangleq \text{Free}(e_1) \\
\text{Free}(e_1 \oplus e_2) &\triangleq \text{Free}(e_1) \cup \text{Free}(e_2) \\
\text{Free}(e \text{ when } C(x)) &\triangleq \text{Free}(e) \cup \{x\} \\
\text{Free}(\text{merge } x [(C_i \Rightarrow e_i)]^i) &\triangleq \{x\} \cup (\bigcup_i \text{Free}(e_i)) \\
\text{Free}(\text{case } e \text{ of } [(C_i \Rightarrow e_i)]^i (_ \Rightarrow e_d)) &\triangleq \text{Free}(e) \cup (\bigcup_i \text{Free}(e_i)) \cup \text{Free}(e_d) \\
\\
\text{Free}(x = e) &\triangleq \text{Free}(e) \\
\text{Free}(xs = (\text{reset } f \text{ every } xrs)(es)) &\triangleq xrs \cup \text{Free}(es) \\
\text{Free}(x = \text{reset } c_0 \text{ fby } e \text{ every } xrs) &\triangleq \text{Free}(e) \cup xrs \\
\text{Free}(\text{last } x = c_0 \text{ every } xrs) &\triangleq xrs
\end{aligned}$$

Figure 5.14: Free variables in NLustre 🐔 [NLustre/IsFree.v:30](#)

$$\begin{aligned}
\text{Def}(x = e) &\triangleq \{x\} \\
\text{Def}(xs = (\text{reset } f \text{ every } xrs)(es)) &\triangleq xs \\
\text{Def}(x = \text{reset } c_0 \text{ fby } e \text{ every } xrs) &\triangleq \{x\} \\
\text{Def}(\text{last } x = c_0 \text{ every } xrs) &\triangleq \{\text{last } x\}
\end{aligned}$$

Figure 5.15: Variables defined by an NLustre equation 🐔 [NLustre/IsDefined.v:35](#)

5.2.3.2 Dead Equation Elimination

The compilation of `switch` blocks described in [section 4.8](#) introduces sampling equations, some of which may be useless. For efficiency, we need to remove these useless equations, as well as possible useless, or *dead*, equations and variables from the original source program. A variable is dead if (i) it is not an output of the node, and (ii) it is not read in any other equation of the node. An equation is dead if it only defines dead variables. And so on by transitive closure. The simplicity of these definitions highlights why we implement this pass in NLustre: in a dataflow program, the uselessness of a variable in defining the output of a node is syntactical. In an imperative language, where an instruction may modify a global state, this definition is not as simple.

Node dependency analysis The function that determines the dead variables of a node uses a simple graph analysis. It first builds the dependency graph of the node. The function `Free` presented in [figure 5.14](#) calculates the set of free variables in NLustre expressions and equations. Since the goal is to determine which variables are unused, we are not interested in instantaneous dependencies, but on any dependency between variables of the node. Therefore, the free variables at right of a `fbby` are taken into account. The `Free` function also makes a distinction between the `last` and current values of variables even though this distinction is not important for this pass; we reuse these definitions other contexts. The `Def` function collects the set of variables defined by an

equation. Again, the distinction between `x` and `last x` does not matter for this pass.

These functions are used to build a dependency graph for the node. The vertices of the graph are the variables of the node. For each equation `eq`, for each pair (x, y) where `x` is defined with or without `last` by `eq` and `y` is free with or without `last` in `eq`, there is an arc from `x` to `y`.

```

Variable (gr : Env.t PS.t).

Definition traverse_one (wh gr : PS.t) (x : ident) :=
  let ps := match Env.find x gr with
    | Some ps => ps
    | None => PS.empty
  end in
  PS.fold (fun y '(wh, gr) => (wh / y, if y ∈ wh then { y } ∪ gr else gr))
    ps (wh / x, gr / x).


Function unreachable whgr
{measure (fun '(wh, gs) => (PS.cardinal wh + PS.cardinal gs)) whgr} :=
  match choose (snd whgr) with
  | None => (fst whgr)
  | Some x => unreachable (traverse_one (fst whgrs) (snd whgrs) x)
end.

```

Listing 5.3: Unreachable vertices 🐔 [NLustre/DeadCodeElim/DCE.v:169](#)

Graph Analysis A variable is dead if and only if the corresponding vertex in the graph is unreachable from the output variables. To compute the set of unreachable vertices we implement a simple reachability analysis by traversal of the graph. We use some common set notations to make the code more readable. The graph to be analysed `gr` is passed as an invariant parameter. It is represented as an associative map from each vertex to the set of variables it depends on (`Env.t PS.t`). The main function is `unreachable`. Its argument, `whgr`, is a pair of a set of *white* vertices that have not been visited, and *grey* vertices that are queued to be visited. Initially, the white set contains all vertices, and the grey set contains all output variable vertices. The algorithm proceeds by choosing one vertex `x` in the grey set and calling the `traverse_one` function on it. This function removes `x` from the grey set. Then, for each `y` that `x` depends on, if `y` is still in the white set, it is transferred from the white to the grey set. The `unreachable` function is then called recursively on the new white and grey sets. The algorithm ends when the grey set is empty. At this point, all the remaining vertices in the white set are unreachable from the original grey set and the corresponding variables are therefore dead.

The `unreachable` function is defined using the Coq `Function` command. It is similar to the `Program Fixpoint` extension that we used to define the Lustre dependency analysis, as it sidesteps the guarded recursion criteria of Coq by defining a decreasing measure on one of the arguments. It is a bit more limited than `Program Fixpoint`: only proof obligations related to the termination of the algorithm may be solved separately. This


$$\begin{array}{l}
\boxed{\text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs} \text{ let } \text{eqs} \text{ tel}} \triangleq \\
\text{let } \text{dead} := \text{dead-in-node } \text{outs} \text{ locs} \text{ eqs} \text{ in} \\
\text{let } \text{locs}' := [x \mid x \in \text{locs} \wedge x \notin \text{dead}] \text{ in} \\
\text{let } \text{eqs}' := [\text{eq} \mid \text{eq} \in \text{eqs} \wedge (\forall x, (x \in \text{Def}(\text{eq}) \vee \text{last } x \in \text{Def}(\text{eq})) \implies x \notin \text{dead})] \text{ in} \\
\text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ var } \text{locs}' \text{ let } \text{eqs}' \text{ tel}
\end{array}$$
Figure 5.16: Dead Equation Elimination in a node  NLustre/DeadCodeElim/DCE.v:584

means that `Function` does not facilitate reasoning with dependant types. However, it does generate a functional induction scheme that we can use to reason a posteriori on the function. This is useful since we need to establish several properties of `unreachable` to prove the correctness of the pass.

Node transformation The full transformation for a node is described in [figure 5.16](#). Function `dead-in-node` first determines the set of dead variables in the node by applying the `unreachable` analysis. Then, it filters the local variables, keeping only those that do not appear in the dead set, and also the list of equations, keeping only the ones that do not define variables in the dead set.

Correctness The proof of semantic preservation for this pass is trivial, and requires less than 30 lines of proof. This is unsurprising: the only transformation is removing variables and equations. Since each equation defines semantic constraints on a history, removing equations can only remove constraints, and never add new ones, and the implication of source to target semantics is trivially true.

While the semantics preservation proof is trivial, some effort is still required to prove the preservation of static invariants of a node, such as typing and clock typing. In particular, removing local variables may make an expression ill-typed if it refers to variables that were removed. To prove that this never happens, we first need to prove that the node dependency analysis is complete, that is, that it covers every possible dependency. Then we need to prove that the reachability analysis is correct, as specified by the lemma below. For each vertex x in the dead set, if another vertex y depends on x , then y must also be in the dead set. This ensures that any equation that previously used variables that have been removed has also been removed. This result is used in most of the proofs of preservation of static invariants for this pass.

Lemma 25 (Correctness of `unreachable`  NLustre/DeadCodeElim/DCE.v:257)

$$\begin{array}{l}
\text{if } \text{unreachable } G(\text{wh}, \text{gr}) = \text{dead} \text{ and } x \in \text{dead} \\
\text{then } \forall y, x \rightarrow_G y \implies y \in \text{dead}
\end{array}$$

```

node f (x, y : int) returns (z : int)
var t1, t2, fby$2, fby$4 : int;
    fby$1, fby$3 : bool;
let
  fby$1 = true fby false;
  fby$2 = 0 fby t1 + 1;
  t1 = if fby$1 then x else fby$2;
  fby$3 = true fby false;
  fby$4 = 0 fby t2 + 1;
  t2 = if fby$3 then y else fby$4;
  z = t1 + t2;
tel

```

```

node f (x, y : int) returns (z : int)
var t1, t2, fby$2, fby$4 : int;
    fby$1 : bool;
let
  fby$1 = true fby false;
  fby$2 = 0 fby t1 + 1;
  t1 = if fby$1 then x else fby$2;
  fby$4 = 0 fby t2 + 1;
  t2 = if fby$1 then y else fby$4;
  z = t1 + t2;
tel

```


Figure 5.17: Optimizing duplicate `fby` equations

5.2.3.3 Fby Minimization


The `fby`-normalization pass of section 4.12 may introduce duplicate `true fby false` equations when normalizing `fbys` that are not initialized by constants. For instance, `fby`-normalization could generate the code shown in figure 5.17 at left, where variables `fby$1` and `fby$3` are defined by the same expression and have the same clock type. Additionally, we have observed that source programs may also contain redundant `fbys`, for example, in expressions of the form `0 fby x` that access the previous value of `x` at multiple points in a program. Since each `fby` equation induces a separate state variable in the generated code, and uses thus more memory, it is useful to minimize their number.

Computing equivalence classes If two `fby` equations generate the same stream, only one need to be kept. For instance, in figure 5.17, the program at left should be transformed into the one at right. In the compiler, this semantic equivalence is approximated syntactically: two `fby` equations are considered equivalent if they have the same initialization constant, update expression and set of reset conditions. The function presented in figure 5.18 analyses the equations in a node to compute the equivalence classes. The function `fbys cls eq` processes equation `eq` in a context where the set of classes `cls` have already been determined from previous equations. Each class in `cls` is represented by the variable defined by the *primary* `fby` equation of the class along with its initialization constant, update expression and reset conditions. The function returns an updated set of classes, and a substitution from the variables defined by *secondary* `fbys` to the ones defined by primary `fbys`. If `eq` is a `fby` equation, the function checks if its parameters correspond to any of the classes in `cls`. If so, this `fby` is secondary, because it is a duplicate from another `fby` that has already been analysed. A substitution from its identifier to the primary one is returned. If not, this `fby` is primary: the list of classes is extended with it and the empty substitution is returned. In all cases where `eq` is not a `fby` equation, the classes `cls` are unchanged and the empty substitution is returned.

$$\begin{aligned}
 \text{fbys } cls \ (x = \text{reset } c_0 \ \text{fby } e \ \text{every } xrs) &\triangleq \begin{cases} \text{if } (y, c_0, e, xrs) \in cls \ \text{then } cls, \{x \mapsto y\} \\ \text{else } cls \cup \{(x, c_0, e, xrs)\}, \text{id} \end{cases} \\
 \text{fbys } cls \ eq &\triangleq cls, \text{id} \\
 \text{fbys } cls \ \epsilon &\triangleq cls, \text{id} \\
 \text{fbys } cls \ (eq; eqs) &\triangleq \text{let } cls_1, \sigma_1 := \text{fbys } cls \ eq \ \text{in} \\
 &\quad \text{let } cls_2, \sigma_2 := \text{fbys } cls_1 \ eqs \ \text{in} \\
 &\quad cls_2, (\sigma_1 \circ \sigma_2)
 \end{aligned}$$

 Figure 5.18: Computing **fby** equivalence classes  NLustre/DupRegRem/DRR.v:53

$$\begin{aligned}
 &[\text{var } locs \ \text{let } eqs \ \text{tel}] \triangleq \\
 &\text{let } _, \sigma := \text{fbys } \emptyset \ eqs \ \text{in} \\
 &\text{let } locs' := [x \mid x \in locs \wedge x \notin \text{dom}(\sigma)] \ \text{in} \\
 &\text{let } eqs' := [[eq]_\sigma \mid eq \in eqs \wedge (\forall x, x \in \text{Def}(eq) \implies x \notin \text{dom}(\sigma))] \ \text{in} \\
 &\text{var } locs' \ \text{let } eqs' \ \text{tel}
 \end{aligned}$$

 Figure 5.19: Removing duplicate registers  NLustre/DupRegRem/DRR.v:136

The **fbys** function is applied to a list of equations by abuse of notation. When the list is empty, the classes are left unchanged and the substitution is empty. The head of the list is treated first, potentially adding a new class, the new set of classes cls_2 is then passed to the recursive call for the tail of the list. The substitution from the head and tail of the list are then combined.

Transformation of the node The node transformation is presented in [figure 5.19](#). It first computes the substitution of secondary to primary **fby** equations. It then filters the local variables to keep only those that do not correspond to secondary **fby**s. Finally, it filters the equations, and applies the substitution to the expression of each remaining equation. An alternative to applying a substitution would be to simply introduce copy equations from primary to secondary variables, but this would make the code longer without necessarily simplifying the proofs.

Correctness of the transformation The semantic preservation proof for this pass consists in composing two main lemmas. The first, [lemma 26](#), is similar to many lemmas from the previous chapter. It states that, if an equation eq has a semantics under history H , and if H refines itself modulo substitution σ , then the renamed equation also has a semantics under H .

Lemma 26 (Renaming correctness  NLustre/DupRegRem/DRRCorrectness.v:252)

$$\text{if } G, H, bs \vdash_{\text{ind}} eq \ \text{and} \ H \sqsubseteq_\sigma H \ \text{then} \ G, H, bs \vdash_{\text{ind}} [eq]_\sigma$$

Since this pass does not introduce any new variables, the history does not change. This makes the second hypothesis of the lemma quite strong: to establish it, we need to prove that for each association $\sigma(x) = y$, x and y are associated to the same stream in H . To do so, we first establish in [lemma 27](#) a syntactic property of the `fbys` function: any association between variables x and y in the substitution means that the source program contains equivalent `fbys` equations for x and y . This property is proven by induction on the list of equations and by case-analysis on the definition of the `fbys` function.

Lemma 27 (Correctness of `fbys` 🐔 [NLustre/DupRegRem/DRR.v:241](#))

$$\begin{array}{l} \text{if } \text{fbys } \emptyset \text{ eqs} = (cls', \sigma) \quad \text{and} \quad \sigma(x) = y \\ \text{then } \exists c_0 e \text{ xrs}, (x = \text{reset } c_0 \text{ fbys } e \text{ every } \text{xrs}) \in \text{eqs} \\ \quad \wedge (y = \text{reset } c_0 \text{ fbys } e \text{ every } \text{xrs}) \in \text{eqs} \end{array}$$

After obtaining this result, we use a result on the semantic determinism for `fbys` equations to establish the second hypothesis of [lemma 26](#) and complete the overall proof of semantic preservation.

5.2.3.4 Other possible optimizations

The two optimization passes that we just described are fairly uncomplicated, the main aim being to eliminate inefficiency introduced by previous compilation passes to simplify the corresponding correctness proofs. The fact that the optimizations may also remove some inefficiencies from the source code is a bonus. More optimizations could be treated at the NLustre level. We describe three possible optimizations, and the difficulties we think their verification may pose.

Node inlining Each node instantiation in the dataflow program induces a function call in the generated imperative code, and possibly some manipulations of the data structure used for outputs. This incurs a run-time cost. To avoid it, the compiler could inline the definitions of some nodes at their instantiation site.

The listing at left of [figure 5.20](#) shows a normalized form of the `count_up` node described in the introduction, and a node that specializes it to count the number of `true`s of input `b`. The listing at right shows the result of inlining the `count_up` instance in `cnt_true`. The input parameter of `count_up`, `inc`, is bound to the argument expression `1 when b`. The local variable `c` is bound to the output parameter `o`. This example highlights one subtlety of this transformation: since the call to `count_up` is sampled by `b`, all the variable declarations and constants in the inlined code must be sampled as well. This is essentially the same as specializing the type of a polymorphic function. Another subtlety not represented here is that some of the declarations of the inlined node may need to be renamed to avoid conflicts with names used in the instantiating node or other inlined

```

node count_up(inc:int) returns (o:int)
var norm$1 : int32;
let
  norm$1 = 0 fby o;
  o = norm$1 + inc;
tel

node cnt_true(b:bool) returns (y:int)
var c:int when b; py:int;
let
  c = count_up(1 when b);
  y = merge b
      (true => c)
      (false => py when not b)
  py = 0 fby y;
tel

```

```

node cnt_true(b:bool) returns (y:int)
var inc, norm$1, o, c: int when b;
  py: int;
let
  inc = 1 when b;
  norm$1 = (0 when b) fby o;
  o = norm$1 + inc;
  c = o;
  y = merge b
      (true => c)
      (false => py when not b)
  py = 0 fby y;
tel

```

Figure 5.20: Inlining of the count_up node

nodes. Finally, if the node to be inlined is called with `reset`, inlining also needs to distribute its reset conditions over the stateful equations of the inlined node.

We believe that these three difficulties would create most of the complications in any semantic preservation proof. Indeed, this would be akin to combining the difficulties of (i) the sampling from `switch` compilation (section 4.8), (ii) the renaming from local scope flattening (section 4.9), and (iii) the distribution and application of `reset` from unnesting/transcription (sections 4.10 and 5.2.2). To avoid this, a solution could be to treat inlining earlier, in the Lustre language, where the pass could at least introduce local scopes and `reset` blocks to shift the last two difficulties to the corresponding compilation passes.

Additionally, not all node instantiations should be inlined. Indeed, in some cases, inlining leads to an increase in code size that outweighs the gains of eliminating a function call. To judge if a node should be inlined or not, we would need to implement a heuristic based on the complexity of the node to be inlined, the number of parameters, etc. This heuristic would not need to be formally verified, as it does not modify the code but only decides whether or not it should be modified. It could therefore be implemented separately, possibly as an OCaml function.

Constant propagation Another useful optimization would be to statically evaluate expressions when possible. For example, programmers commonly write bit-wise manipulations using bit-shift operators: we could for example reduce `1 << 4` into `16` at compile time. This may be particularly useful when combined with the inlining optimization: if a node is instantiated with constant parameters, its inlined body may contain a lot of reducible expressions.

It is not clear if implementing this optimization in Vélus would improve performances directly; indeed, CompCert already performs constant propagation at the Register Transfer Language (RTL) level [Ler09a, §7.2]. However, it might still facilitate other optimizations by producing simpler expressions and therefore improving the number of successful syntactic comparisons.

Implementing this optimization in NLustre would not pose too many difficulties: recall that the semantics of CompCert operators are specified as partial functions, as presented in [listing 2.3](#). To simplify an operation of the form $c_1 \oplus c_2$, we can simply apply the corresponding operator at compile time on the values associated with constants c_1 and c_2 . Since we are using the semantic operator itself, the semantic correspondance proof for this reduction would be trivial. In order to fully reduce all possible expressions, this reduction would also need to be interleaved with the expression inlining pass. For instance, to simplify $x = 1 + 2$; $y = x * 3$, we would first reduce $1 + 2$ into 3 , then replace x with 3 everywhere it is used, including the equation for y , and then reduce $3 * 3$ into 9 . This mechanism of reduction/rewriting needs to be repeated until a fixed point is reached: this means defining the compilation function as a Coq recursive function with a decreasing measure on the body of the node being transformed. We would most likely need to use a function of the numbers of operators and variables appearing in the node.

Dataflow minimization The optimization we introduced to deduplicate redundant **fb** equations is limited. Indeed, consider the program presented below. The streams associated with **t1** and **t2** are both the same stream of natural numbers. However, the optimization pass described in the previous section is not able to remove one of them, as their update expressions are not syntactically equal.

```
node f(x : int) returns (z : int)
var t1, t2 : int;
let
  t1 = 0 fby (t1 + 1);
  t2 = 0 fby (t2 + 1);
  z = t1 + t2 + x;
tel
```

Listing 5.4: Node with redundant **fb**s

To treat this kind of program, we need a more general dataflow minimization pass. Such a transformation generalizes common sub-expression elimination. One possible algorithm consists in iteratively refining the most general possible equivalence relation between variables of the node. It starts with R_0 , a relation where all variables are equivalent. Relation R_{n+1} is built from R_n in the following way: two variables x and y are related by R_{n+1} iff they are defined by equations that are equal modulo R_n . The algorithm continues until a fixed point is reached, that is $R_{n+1} = R_n$. At this point, one equation is introduced for each equivalence class. In the example of [listing 5.4](#),

the equivalence classes of R_0 would be $\{ t1, t2, z \}$ and those of R_1 and R_2 would be $\{ t1, t2 \}$ and $\{ z \}$, which means one of the two `fb` equations can be removed.

This algorithm is implemented in the Heptagon compiler [Dev17], which is written in OCaml. From what we could gather, the worst-case complexity of this implementation is $\mathcal{O}(n^2 \log n)$ with n the number of variables in the node. This complexity seems close to the theoretical limit: in the worst case, we need n iterations before finding the fixed point. Each iteration needs to sort each of the n variables into a new equivalence class. Finding which class is the correct one, with an efficient data structure, requires $\log n$ comparisons. Additionally, the OCaml program exploits some imperative features. It is not clear if a purely functional program would have the same complexity. Because of this, and the possible intricacy of the proof of semantic preservation for this algorithm, it might be better to treat this optimization using a translation-validation approach. The fixed-point algorithm may be implemented in OCaml, with a Coq verified validator checking that the classes obtained at the end of the algorithm are correct with respect to the program.

5.3 Generalizing the Stc language

The usual compilation scheme for synchronous dataflow languages directly translates normalized and scheduled dataflow equations to imperative code [Bie+08]. In Vélus, there is an extra step: the intermediate language Stc, was introduced in [POPL20]. The main purpose of this language is to improve the effectiveness of equation scheduling before translation to imperative code. Recall that, in Obc, two adjacent `switch` statements on the same condition can be fused. The `switch` statements in Obc code are generated from the clocks and control expressions of the dataflow program. In order to maximize the number of `switch` fusions, and thereby minimize branching in the generated program, the scheduler should place equations with the same or similar clocks next to one another. In early versions of Vélus [PLDI17], scheduling was performed on NLustre equations, but with the addition of modular reset [POPL20], this became less effective. The extra complication is that `reset` operations are compiled into `switch` statements, whose guard expressions may differ from the activation clocks of the corresponding equations. To facilitate the fusion optimization, it is therefore better to schedule `reset` operations independently from the equations themselves. This is especially important for programs with state machines, where the compilation may generate several `reset` operations with the same condition, which should be scheduled together if possible. The Stc language, which we now present, allows independent scheduling of `reset` operations.

5.3.1 Example and informal semantics

The example of figure 5.21 illustrates the separation of `reset` and `update` operations in Stc. The NLustre node at left is compiled into the Stc transition system at right. Variable `x` becomes a state variable in Stc, since it is used with `last` in the source. Its `last` initialization is used to specify the initial value of `x` in the header of the system. The body of the system contains *transition constraints*. Since the `last` initialization may

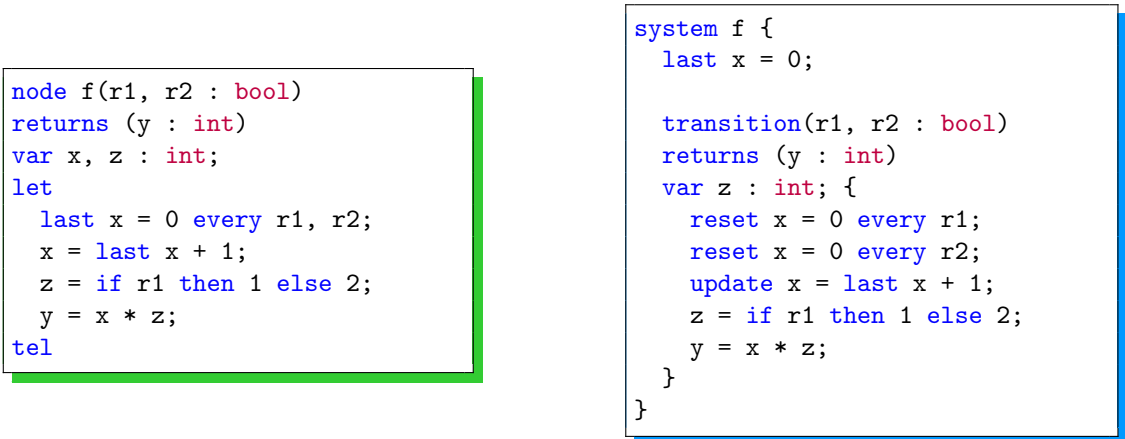


Figure 5.21: An Stc system compiled from an NLustre node

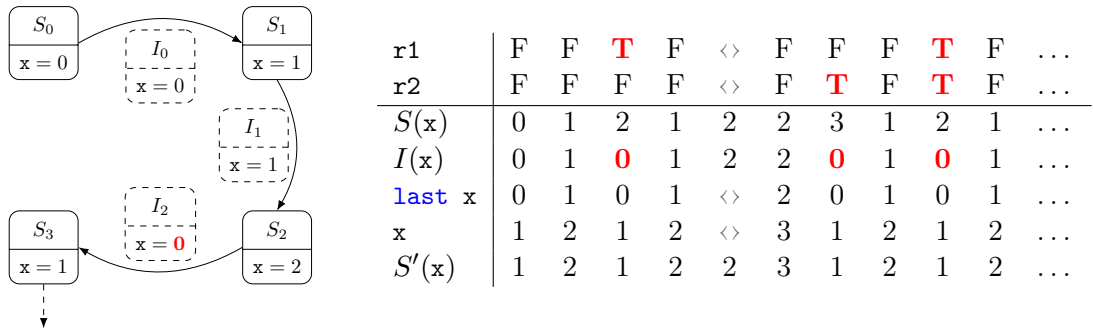


Figure 5.22: Example execution of the program of figure 5.21

be `reset`, it generates a `reset` constraint for each of its reset conditions. The equation defining `x` is translated into an update constraint. The equations defining `z` and `y` are translated into simple constraints. The first `reset` constraint and the constraint defining `z` should be scheduled next to each other, since they have the same condition `r1`.

Figure 5.21 illustrates the behavior of this system, focusing on the state variable `x`. An Stc system specifies the initial values and transition relation for its state. The diagram at left shows the first four states for the example system, for the inputs indicated in the table at right. In the initial state, S_0 , the value of `x` is set to 0, according to its declaration. The update equation specifies the value of `x` in the next state S_1 . The intermediate states I , shown with dashed rectangles, represent the value of state variables after a reset but before an update. For state variables manipulated with `last`, this corresponds to the `last` value, if it is present. For instance, $I_2(x) = 0$ because, at the third reaction, `r1` is `true`, and therefore the value of `last x` must be reset to 0.

Our formal description of the language focuses on the modifications necessary to support the features seen in this example, namely generalized resetting and `last` variables.

```

system ::= system f { init statedecl* ; last statedecl* ; sub subdecl* ; transition }
statedecl ::= x = c
subdecl ::= x : f
transition ::= transition ( var+ ) returns ( var+ ) var var* { tc+ }
tc ::= x =ck ce
    | reset x = c every ck
    | next x =ck e
    | update x =ck ce
    | reset f<x> every ck
    | x* =ck f<x> ( e* )

```

Figure 5.23: Abstract syntax of Stc

5.3.2 Syntax of Stc

The abstract syntax of Stc is presented in [figure 5.23](#). An Stc program is a list of (transition) systems. The **init** and **last** keywords declare two types of state variables, with their initialization by a constant value. Intuitively, the former corresponds to variables defined by **fbv** equations in NLustre. The latter corresponds to the **last** variables of NLustre; in particular, their **last** value may be accessed in expressions. We detail below how their semantics differ. The **sub** keyword declares a list of sub-systems instances, with the corresponding system names. Then, the **transition** keyword specifies how the state is updated. The transition is declared with input, output, and possibly local variables. Its body is a list of transition constraints.

The simplest transition constraint assigns a value to an output or local variable. The value is defined by a control expression, using the same language of expressions as in NLustre. The constraint is annotated by a clock type, which indicates when it is active. A **reset** constraint specifies that the value of a state variable x is reset every time a clock ck evaluates to **true**. A **next** (respectively **update**) constraint specifies how a state variable declared with **init** (respectively **last**) is updated. The clock-type annotations of these constraints indicate when the state is updated. A sub-system can also be reinitialized by a **reset** constraint. The final constraint updates a sub-system using inputs calculated from expressions, and associates its output values to variables.

5.3.3 Formal semantics of Stc

State and state variables In the semantic model, a state S has the same form as the instantaneous memory used in the NLustre memory semantics described in [section 5.2.1.3](#). We focus on the treatment of state variables, as the treatment of sub-system instances is mostly unchanged from [[Bru20](#), §3.4.2].

$$\frac{
\begin{array}{l}
P(f) = \text{system } f \{ \text{init } stds_n; \text{last } stds_l; \text{sub } subds; \text{trans } \} \\
\forall(x = c) \in stds_n, S(x) = c \quad \forall(x = c) \in stds_l, S(x) = c \\
\forall(x : f') \in subds, \text{initial-state } P f' S[x]
\end{array}
}{
\text{initial-state } P f S
}$$

Figure 5.24: Initial state of an Stc system 🐔 [Stc/StcSemantics.v:61](#)

State initialization The semantic judgment `initial-state P f S` specifies that the initial state of system f in program P is S ; the formal rule defining it is presented in [figure 5.24](#). For every state variable x declared in the system, $S(x)$ contains the corresponding constant. The initialization of each sub-system yields the corresponding sub-state, accessed by $S[x]$.

Transition constraints Each transition constraint constrains part of the next state S' , depending on the current state S . These constraints are formalized by the semantic judgement $P, R, b, S, I, S' \vdash_{stc} tc$. In addition to the current and next states, it takes as parameters the program P for evaluation of sub-systems, an environment R that stores the values of variables, a base clock b , and an intermediate state I which specifies the value of state variables after a potential reset but before an update.

The semantics of a stateless constraint $x =_{ck} e$ is specified by the first rule in [figure 5.25](#). The expression e is evaluated instantaneously under environment R . The resulting synchronous value sv must be associated with the left-hand variable in R . This is similar to the rules for equations in the dataflow languages. As in NLustre, the clock type of the equation is evaluated and must correspond to the presence or absence of sv . This overspecification simplifies the correctness proof for the translation to Obc.

There are two rules for a constraint `reset x = c every ck`. The first applies when ck evaluates to `true` and associates x with c in I . The rule where ck evaluates to `false` does not constrain the value of x in I . This is because a state variable may have several `reset` constraints associated with it. This is highlighted by the example presented in [figure 5.22](#), where x is reset whenever either `r1` or `r2` is `true`. With these rules, all active resets constrain I , and they must not be contradictory. If none of the `reset` constraints are active, then the value of $I(x)$ is not specified by the `reset` rules. Instead, it is specified by the rules controlling the update of x , which we detail below. This design choice is practical because, syntactically, there is only one update constraint for x , while there may be several independent reset constraints.

We now look at the rule which specifies the semantics of `next x =ck e` when e produces a present value. The constraint is annotated with a list of clock types $ckrs$, which must correspond to the clock types of the `reset` constraints for x in the system. The first premise of the rule specifies that, if all the clocks in $ckrs$ evaluate to false, then $I(x) = S(x)$. This is complementary to the `reset` rules, and completes the specification of $I(x)$. The other premises specify the value of x in the subsequent state S' and in the environment R . The value associated with x in R is the one in I : when accessing x in

$$\begin{array}{c}
\frac{R, b \vdash_{\text{inst}} e^{ck} \downarrow sv \quad R(x) = sv}{P, R, b, S, I, S' \vdash_{\text{stc}} x =_{ck} e} \\
\\
\frac{R, b \vdash_{\text{inst}} ck \downarrow \mathbf{T} \quad I(x) = c}{P, R, b, S, I, S' \vdash_{\text{stc}} \mathbf{reset} \ x = c \ \mathbf{every} \ ck} \quad \frac{R, b \vdash_{\text{inst}} ck \downarrow \mathbf{F}}{P, R, b, S, I, S' \vdash_{\text{stc}} \mathbf{reset} \ x = c \ \mathbf{every} \ ck} \\
\\
\frac{(\forall ck \in ckrs, R, b \vdash_{\text{inst}} ck \downarrow \mathbf{F}) \implies I(x) = S(x) \quad R, b \vdash_{\text{inst}} e^{ck} \downarrow \langle v \rangle \quad R(x) = \langle I(x) \rangle \quad S'(x) = v}{P, R, b, S, I, S' \vdash_{\text{stc}} \mathbf{next}_{ckrs} \ x =_{ck} \ e} \\
\\
\frac{(\forall ck \in ckrs, R, b \vdash_{\text{inst}} ck \downarrow \mathbf{F}) \implies I(x) = S(x) \quad R, b \vdash_{\text{inst}} e^{ck} \downarrow \langle \rangle \quad R(x) = \langle \rangle \quad S'(x) = I(x)}{P, R, b, S, I, S' \vdash_{\text{stc}} \mathbf{next}_{ckrs} \ x =_{ck} \ e} \\
\\
\frac{(\forall ck \in ckrs, R, b \vdash_{\text{inst}} ck \downarrow \mathbf{F}) \implies I(x) = S(x) \quad R, b \vdash_{\text{inst}} e^{ck} \downarrow \langle v \rangle \quad R(\mathbf{last} \ x) = \langle I(x) \rangle \quad R(x) = \langle v \rangle \quad S'(x) = v}{P, R, b, S, I, S' \vdash_{\text{stc}} \mathbf{update}_{ckrs} \ x =_{ck} \ e} \\
\\
\frac{(\forall ck \in ckrs, R, b \vdash_{\text{inst}} ck \downarrow \mathbf{F}) \implies I(x) = S(x) \quad R, b \vdash_{\text{inst}} e^{ck} \downarrow \langle \rangle \quad R(\mathbf{last} \ x) = \langle \rangle \quad R(x) = \langle \rangle \quad S'(x) = I(x)}{P, R, b, S, I, S' \vdash_{\text{stc}} \mathbf{update}_{ckrs} \ x =_{ck} \ e}
\end{array}$$

Figure 5.25: Transition Semantics of Stc 🐔 [Stc/StcSemantics.v:78](#)

expressions of the system, its value is the one after any reset, but before update; this is similar to the behavior of a resettable **fb**y. The *next* value of x , stored in S' , is that produced by expression e .

The second rule for **next** $x =_{ck} e$ applies if the value produced by e is absent. Its first premise is the same as that of the previous rule. The rule specifies that the value in S' is not updated: it is the same as the value in I .

The next two rules concern **update** constraints. They both start with the premise completing the specification of $I(x)$. Then, they generalize the rule for stateless constraints by specifying the value of **last** x , which is the same as $I(x)$. If e evaluates to a present value $\langle v \rangle$, the value of x in S' is set to v .

We do not show the semantic rules for sub-system reset and update, as they are almost identical to the ones presented in [Bru20, Figure 3.6]. The only necessary addition to the update rule is a premise relating $I[x]$ with $S[x]$ in the case where all reset constraints are inactive, similarly to the state variable update rules.

$$\begin{array}{c}
P(f) = \text{system } f\{\dots\}; \text{transition } ([x_i]^i) \text{ returns } ([y_j]^j) \text{ var } locs; \{tcs\} \\
\forall i, R(x_i) = vx_i \quad \forall j, R(x_j) = vx_j \quad P, R, (\text{ibase-of } [vx_i]^i), S, I, S' \vdash_{stc} tcs \\
\hline
P, S, S' \vdash_{stc} f([vx_i]^i) \downarrow [vy_j]^j \\
\\
P, S, S' \vdash_{stc} f(xss(k)) \downarrow yss(k) \quad P, S' \vdash_{stc} f(xss) \overset{k+1}{\circlearrowleft} yss \\
\hline
P, S \vdash_{stc} f(xss) \overset{k}{\circlearrowleft} yss
\end{array}$$

Figure 5.26: Stc system semantics 🐦 [Stc/StcSemantics.v:229](#)

Iterated system semantics The instantaneous semantics of a full Stc system are given in the first rule of [figure 5.26](#). The judgement $P, S, S' \vdash_{stc} f(xs) \downarrow ys$ states that in a program P , given inputs xs , system f transitions from state S to state S' , producing outputs ys . The rule defining this judgment is similar to the ones for nodes in Lustre and NLustre. It supposes the existence of a local environment R in which the inputs and outputs of the system are given, and an intermediate state I , and ensures that all transition constraints apply to R, S, I and S' .


The infinite behavior of the system is specified by the judgment $P, S \vdash_{stc} f(xss) \overset{k}{\circlearrowleft} yss$, which states that the iterated application of transitions of f starting from state S on streams xss produce streams yss . This judgment is defined coinductively by the second rule of [figure 5.26](#). Its definition uses an extra parameter k to specify the index of streams xss and yss , which is implicitly 0 if not written.

5.3.4 From NLustre to Stc

We have already shown an example of the compilation of NLustre to Stc, initially presented in [\[Bru20, Figure 3.3\]](#). Each NLustre equation is compiled into one or several transition constraints. To support the compilation of **last** variables and generalized resetting, we extend the compilation function. The updated function $[eq]_{\text{lasts}}$ is presented in [figure 5.27](#). It has a parameter **lasts** which associates each **last** variable to the list of its reset conditions. This association is determined by analysing all the **last** initialization equations in a node before applying the compilation function.

Compiling a stateless equation requires an additional case analysis compared to the previous version. If the equation defines a stateless variable x , then it is compiled to a stateless constraint. However, if x is declared with **last** then the equation is compiled to an **update** constraint. The **lasts** table is used both to determine whether or not x is declared with **last** and to recover the list of reset clocks used to annotate the **update** constraint. The compilation of a **last** initialization equation generates one state-variable **reset** constraint for each **reset** condition of the equation. Each source **reset** condition is expressed as a variable x_i annotated with its clock ck_i . In the Stc code, this condition is simply represented as the sampled clock $ck_i \text{ on } T(x_i)$. As expected, this clock evaluates


$$\begin{aligned}
[x = e]_{\text{lasts}} &\triangleq \begin{cases} \text{if } x \notin \text{lasts} & \text{then } x = e \\ \text{if } \text{lasts}(x) = \text{ckrs} & \text{then } \text{update}_{\text{ckrs}} x = e \end{cases} \\
[\text{last } x = c \text{ every } [r_i^{ck_i}]^i]_{\text{lasts}} &\triangleq [\text{reset } x = c \text{ every } (r_i \text{ on } T(ck_i))]^i \\
[x = \text{reset } c \text{ fby } e \text{ every } [r_i^{ck_i}]^i]_{\text{lasts}} &\triangleq \begin{aligned} &\text{next } x = [r_i \text{ on } T(ck_i)]^i e; \\ &[\text{reset } x = c \text{ every } (r_i \text{ on } T(ck_i))]^i \end{aligned} \\
[xs = (\text{reset } f \text{ every } [r_i^{ck_i}]^i)(es)]_{\text{lasts}} &\triangleq \begin{aligned} &xs = [r_i \text{ on } T(ck_i)]^i f\langle x \rangle(es); \\ &[\text{reset } f\langle x \rangle \text{ every } (r_i \text{ on } T(ck_i))]^i \end{aligned}
\end{aligned}$$

Figure 5.27: Translation of NLustre equations  [NLustreToStc/Translation.v:50](#)

to **true** iff x_i is present and evaluates to **true**.


The changes to the compilation of **fbv** and node instantiations are related to the generalization of **reset**. For each type of equation, the function generates the corresponding update constraint (**next** and instance update), and one **reset** constraint for each reset condition of the equation. These transitions may be scheduled independently, since their overall semantics is independent of their syntactic order.

Correctness The proof of semantic correspondence between NLustre and Stc is based on the NLustre semantics with memory. Having the memory of the NLustre node facilitates stating and proving the correctness lemma below. If the NLustre node has a semantics with memory M , then (i) M_0 is the initial state for the compiled system, and (ii) iterating the transition relation of the compiled system with the original inputs xss produces the same output streams yss .

Lemma 28 (NLustre to Stc translation  [NLustreToStc/Correctness.v:811](#))

if $G, M \vdash_{\text{mem}} f(xss) \Downarrow yss$ **then** initial-state $[G] f M_0 \wedge [G], M_0 \vdash_{\text{stc}} f(xss) \circlearrowleft yss$

Proving that M_0 is the initial state is direct, and our modifications to the compilation pass do not add any difficulties compared to the proof described in [Bru20, §3.2.2]. The proof of the second conclusion requires more work. To establish the iterated semantics presented in figure 5.26, it exploits a result unstated in the previous lemma: the sequence of states of the Stc system corresponds exactly to M . In other words, the transition relation of the system, applied on the n th inputs, relates M_n with M_{n+1} . This is stated formally by the lemma below.

Lemma 29 (Instantaneous NLustre to Stc translation  [NLustreToStc/Correctness.v:731](#))

if $G, M \vdash_{\text{mem}} f(xss) \Downarrow yss$ **then** $\forall n, [G], M_n, M_{n+1} \vdash_{\text{stc}} f(xss(n)) \Downarrow yss(n)$

Recall that the instantaneous semantic rule for systems requires the existence of an environment R and an intermediate state I . Providing R is easy: it is exactly $H(n)$,

where H is the history of the source node. Constructing I is more difficult, as it depends on the stateful equations in the node. The original proof presented in [Bru20, §3.2.2] proceeded by induction on the list of source equations, building I value-by-value and proving simultaneously that it respects the generated constraints. This type of local reasoning was possible because each state variable of the generated system was only subject to the transition constraints generated from a single equation. This is no longer the case. Consider the NLustre equations `last x = 0 every r; ...; x = last x + 1`. The first generates the constraint `reset x = 0 every r`; the second generates the constraint `update x = last x + 1`. The semantics of both depend on $I(x)$. Since the equations are compiled separately, reasoning on I must be global.

To simplify the proof, we adopt the following strategy: in a first step, we build I by induction on the list of equations, along with a strong invariant relating the syntax and semantics of the node with the values of I . In a second step, we use this invariant to prove that I respects the generated transition constraints. The part of the first step relevant to `last` equations is presented in lemma 30. For any `last x = ...` in the source equations, there exists streams associated with x and `last x` in history H , as well as some streams associated with the reset conditions of the equation; all of these streams are related by the `mfbyreset` operator of definition 8 (page 128). The first five conjuncts correspond exactly to the definition of the memory semantics of a `last` initialization equation. Restating them in the invariant allows us to fulfill the semantic constraints in the second part of the proof. The last conjunct relates these streams to the values actually stored in I . Notice that, if we ignore absence and presence, these values correspond to the definition of `fbyreset` in definition 9 (page 129). In the second part, we use the correspondence between `mfbyreset` and `fbyreset` from lemma 18 (page 130), to conclude the proof. The cases of the invariant for `fby` and node instantiation are similar and not shown here.

Lemma 30 (Building intermediate state 🐔 NLustreToStc/Correctness.v:413)

$$\begin{array}{l}
 \text{if } G, H, bs, M \vdash_{mem} eqs \\
 \\
 \text{then } \exists I, \left\{ \begin{array}{l}
 \forall (\text{last } x = c \text{ every } xrs) \in eqs, \\
 \exists xs \ ls \ ys \ rs, \left\{ \begin{array}{l}
 H(x) \approx xs \\
 \wedge H(\text{last } x) \approx ls \\
 \wedge \forall i, H(xrs_i) \approx ys_i \\
 \wedge \text{bools-ofs } [ys_i]^i \approx rs \\
 \wedge \text{mfbyreset}_c^x M \ xs \ rs \approx ls \\
 \wedge \forall n, \left\{ \begin{array}{l}
 \text{if } rs(n) = \top \text{ then } I_n(x) = c \\
 \text{else } I_n(x) = \text{holdreset}^c \ xs \ rs \ n
 \end{array} \right.
 \end{array} \right. \\
 \wedge \forall (x = \text{reset } c \ \text{fby } e \ \text{every } xrs) \in eqs, \\
 \dots \\
 \wedge \forall (x = (\text{reset } f \ \text{every } xrs)(es)) \in eqs, \\
 \dots
 \end{array} \right.
 \end{array}
 \end{array}$$

5.4 Translation to imperative Obc code

We now describe the remaining compilation passes which translate transition systems to imperative code. We first detail the translation from the Stc to Obc syntax, since it is important for understanding the specifics of the previous and subsequent passes. We then detail the scheduling of Stc programs, and how the scheduling invariant is used to prove the correctness of the Stc to Obc translation. Finally, we discuss the fusion optimization that is applied to the generated Obc. As these passes were already present in earlier versions of the compiler; we focus on the changes necessary to support the extensions of Stc (generalized resetting, `last` variables and updates).

5.4.1 From Stc to Obc

We first describe the translation of an Stc system into an Obc class. The compilation function for a system, presented in [figure 5.28](#) at top, is broadly unchanged. Both types of state variables, declared with `init` and `last`, are translated into Obc state variables (`state`). The sub-systems, declared with `sub`, are translated into instances of the corresponding classes (`instance`). The compilation of a system generates two methods. The first, `reset`, reinitializes all state variables and instances, based on the declared initializations in the Stc system. The second, `step`, implements the transition relation specified by the Stc `transition`. Its body is the sequence of the compiled constraints.

The compilation function for transition constraints $[tc]_{sts}$, presented in [figure 5.29](#), takes an extra input *sts*, specifying the set of state variables of the system. This parameter is necessary for the function to decide whether to translate an Stc variable as a local or state variable in Obc. The compilation scheme for transition constraints is presented first. Recall that each transition constraint is activated on a given clock *ck*. The Obc language does not have clocks. The transformation `controlsts ck stmt` encodes the conditional activation using `switch` statements. If *ck* is the base clock, *stmt* is activated unconditionally. Otherwise, if *ck* is a clock sampled on $C(x)$, a `switch` on *x* is produced, where only branch *C* contains statement *stmt* and the other branches do nothing.

The compilation of a stateless constraint produces an assignment to the corresponding variable. However, since the Obc language does not contain control expressions, the compilation of the expression at right of the assignment may require generating `switch` statements. The compilation function for control expressions, $[e]_{sts}^{stmt}$ therefore takes as a parameter a partial assignment instruction *stmt*. This function traverses the control expressions `merge` and `case`, generating a `switch` for each. When encountering a simple expression, the assignment is completed, and inserted under the corresponding branch of the `switches`. Compiling an `update` constraint is similar, but generates updates of state variables, as exemplified by [figure 5.3](#) (page 119). The compilation of `next` constraints is simpler: since the expression at right may only be a simple expression, it generates a single assignment. The compilation of state-variable `reset` constraints generates assignments of the initialization constant *c* to the corresponding state variable. These assignments are controlled by the condition clock of the `reset` constraint. Finally, compiling sub-system

<pre style="margin: 0;"> system f { init [$x_i^n : ty_i^n = c_i^n$]^{i} last [$x_i^l : ty_i^l = c_j^l$]^{j} sub [$x_k^s : f_k$]^{k} transition (ins) returns ($outs$) var $locs$ {tcs} }</pre>	\triangleq	<pre style="margin: 0;"> class f { [state $x_i^n : ty_i^n$]^{i} [state $x_j^l : ty_j^l$]^{j} [instance $x_k^s : f_k$]^{k} method reset() returns() { [state (x_i^n) := c_i^n]^{i} [state (x_j^l) := c_j^l]^{j} [$_ := f_j(x_j)$.reset()]^{k} } method step(ins) returns($outs$) var $locs$ { [tcs]^{$([x_i^n]^i + [x_j^l]^j)$} } }</pre>
--	--------------	---

Figure 5.28: Translation of Stc systems 🐔 [StcToObc/Translation.v:176](#)

$\text{control}_{sts} \bullet \text{stmt} \triangleq \text{stmt}$ $\text{control}_{sts} (\text{ck on } C(x)) \text{ stmt} \triangleq \text{switch } [x]_{sts} \{ C \Rightarrow \text{stmt} \}$
$[x =_{ck} e]_{sts} \triangleq \text{control}_{sts} \text{ ck } ([e]_{sts}^{x :=})$ $[\text{update}_{ck} x = e]_{sts} \triangleq \text{control}_{sts} \text{ ck } ([e]_{sts}^{\text{state}(x) :=})$ $[\text{next } x =_{ck} e]_{sts} \triangleq \text{control}_{sts} \text{ ck } (\text{state}(x) := [e]_{sts})$ $[\text{reset } x = c \text{ every } ck]_{sts} \triangleq \text{control}_{sts} \text{ ck } (\text{state}(x) := c)$ $[xs =_{ck} f\langle x \rangle (es)]_{sts} \triangleq \text{control}_{sts} \text{ ck } (xs := f(x) . \text{step}([es]_{sts}))$ $[\text{reset } f\langle x \rangle \text{ every } ck]_{sts} \triangleq \text{control}_{sts} \text{ ck } (_ := f(x) . \text{reset}())$
$[\text{merge } x [(C_i \Rightarrow e_i)]^i]_{sts}^{stmt} \triangleq \text{switch } [x]_{sts} \{ C_i \Rightarrow [e_i]_{sts}^{stmt} \}^i$ $[\text{case } e \text{ of } [(C_i \Rightarrow e_i)]^i (_ \Rightarrow e_d)]_{sts}^{stmt} \triangleq \text{switch } [e]_{sts} \{ C_i \Rightarrow [e_i]_{sts}^{stmt} \}^i$ $[e]_{sts}^{stmt} \triangleq \text{stmt}([e]_{sts})$
$[c]_{sts} \triangleq c$ $[x]_{sts} \triangleq \begin{cases} \text{if } x \in sts \text{ then } \text{state}(x) \\ \text{else } x \end{cases}$ $[\text{last } x]_{sts} \triangleq \text{state}(x)$ $[\diamond e_1]_{sts} \triangleq \diamond [e_1]_{sts}$ $[e_1 \oplus e_2]_{sts} \triangleq [e_1]_{sts} \oplus [e_2]_{sts}$ $[e \text{ when } C(x)]_{sts} \triangleq [e]_{sts}$

Figure 5.29: Translation of Stc constraints 🐔 [StcToObc/Translation.v:98](#)

updates and resets generates, respectively, calls to the `step` and `reset` methods of the corresponding class.

The compilation of simple expressions has two subtleties. First, a variable x may be compiled either to an access to a temporary variable x , or to an access to a state variable `state(x)`, based on whether or not x is declared as a state variable in *sts*. Second, `last x` is simply compiled to `state(x)`. This means that, if x is a state variable, both expressions x and `last x` are compiled to `state(x)`. In particular, the constraint `x = last x` is compiled into a `state(x) := state(x)` statement that we may optimize away in Obc. Intuitively, in the imperative program, both `last x` and x refer to the same state variable, or memory cell, but before and after its update. We will see below how this affects the scheduling of Stc constraints.

5.4.2 Scheduling of Stc Constraints

This compilation scheme is only correct for scheduled Stc systems. In this section, we define what it means for a system to be well scheduled. We then discuss the algorithm that schedules an Stc system, its translation-validation, and the special pre-treatment necessary to schedule some programs.

5.4.2.1 Scheduling rules

To define the scheduling rules for a system, we first define the set of variables read or defined by a transition. The free variables of a constraint are those free in its expressions (see [figure 5.14, page 135](#)), or clock. The variables defined by a constraint are specified by the `Def` function presented in [figure 5.30](#). A variable is either defined by a stateless equation, the update of a `last` variable, or the update of a sub-system.

The `WellScheduled` predicate is presented in [figure 5.31](#). It is defined inductively over the list of transition constraints *tcs*. It takes two extra invariant parameters: *ins*, the set of inputs of the system, and *nexts*, the set of state variables declared with `init` and updated with `next`. An empty system is always well scheduled. If the system contains a list of constraints *tcs* followed by a single constraint *tc*, then *tcs* must be well scheduled, and the following constraints relate *tc* and *tcs*.

The second premise specifies which free variables may be read in *tc*. To read the current value of a variable x , it must have been previously written by either (i) being defined in *tcs*, (ii) being an input of the transition, and (iii) being in the *nexts* set (that is, be a state variable updated previously).

The next two premises account for state variable updates. The previous value of a state variable x can only be read before the x is updated. Specifically, x may not be read after it is updated with `next`, and `last x` may not be read after it is update with `update`.

The final three premises account for `reset` constraints. The value of a state variable x (respectively `last x`) is read after its possible reset. Therefore, all `reset` constraints must be scheduled before reading or updating their state variables, or sub-system instances. In the inductive definition, this is encoded contrapositively.

$$\begin{aligned}
\text{Def}(x = e) &\triangleq \{x\} \\
\text{Def}(\text{update } x = e) &\triangleq \{x\} \\
\text{Def}(\text{next } x = e) &\triangleq \emptyset \\
\text{Def}(\text{reset } x = c \text{ every } ck) &\triangleq \emptyset \\
\text{Def}(xs = f\langle x \rangle(es)) &\triangleq xs \\
\text{Def}(\text{reset } f\langle x \rangle \text{ every } ck) &\triangleq \emptyset
\end{aligned}$$

Figure 5.30: Variables defined by transition constraints 🐔 [Stc/StcSyntax.v:935](#)

$$\frac{\text{WellScheduled } ins \ nexts \ \epsilon}{\text{WellScheduled } ins \ nexts \ tcs}$$

$$\begin{aligned}
&\forall x \in \text{Free}(tc), x \in \text{Def}(tc) \vee x \in ins \vee x \in nexts \\
&\forall x \in \text{Free}(tc), (\text{next } x = _) \notin tcs \quad \forall (\text{last } x) \in \text{Free}(tc), (\text{update } x = _) \notin tcs \\
&\forall x, tc = (\text{reset } x = _ \text{ every } _) \implies x \notin \text{Free}(tcs; tc) \wedge (\text{next } x = _) \notin tcs \\
&\forall x, tc = (\text{reset } x = _ \text{ every } _) \implies (\text{last } x) \notin \text{Free}(tcs; tc) \wedge (\text{update } x = _) \notin tcs \\
&\forall x, tc = (\text{reset } _ \langle x \rangle \text{ every } _) \implies (_ = _ \langle x \rangle (_)) \notin tcs
\end{aligned}$$

$$\text{WellScheduled } ins \ nexts \ (tcs; tc)$$

Figure 5.31: Scheduling rules for Stc constraints 🐔 [Stc/StcWellDefined.v:43](#)

This definition completely specifies a scheduling of translation constraints that, when compiled to Obc, yields a correct program. However, systems may have several possible schedulings, or none. In the next two sections, we present a pre-processing step that make some non-schedulable systems schedulable, and then describe the scheduling algorithm used in Vélus, and the heuristics designed to increase its effectiveness with regard to the Obc fusion optimization.

5.4.2.2 Cutting update cycles

Consider the following Lustre equations, and the corresponding transition constraints.

- $x = 0 \text{ fby } y; y = 0 \text{ fby } x$ compiles to $\text{next } x = y; \text{next } y = x$
- $x = \text{last } y; y = \text{last } x$ compiles to $\text{update } x = \text{last } y; \text{update } y = \text{last } x$
- $y = x + \text{last } x$ compiles to $y = x + \text{last } x$

None of these definitions contain dependency cycles according to the rules described in [chapter 3](#). However, none of the generated translation constraints are schedulable. In the first instance, according to the rule for `next` state variables, both x and y must be updated after they are used. However, each is used in the update of the other, which

means that `next x = y` would have to be scheduled before `next y = x`, and vice-versa. This is of course impossible. The same reasoning applies in the second instance. The last instance is a bit different: since the definition of `y` uses both `x` and `last x`, it would have to be scheduled both before and after `x` is updated, which is also impossible.

In all of these instances, the issue is that a value stored in a state variable must be updated while it is still needed. A simple solution is to copy this value into a temporary variable; that way, it is still accessible after the state variable is updated. Of course, copying has a run time and memory cost, which we want to minimize.

Algorithm To illustrate the heuristics and algorithm, we use the Stc code produced for the `drive_sequence` example, presented in [listing 5.5](#). It is not immediately schedulable, as updating `l$mA` depends on `last l$mB`, and vice-versa. A copy for either `last l$mA` or `last l$mB` should be added. Here, choosing either does not make any difference performance-wise.

```

system drive_sequence {
  last l$mA = true; l$mB = true;
  transition(step : bool) returns (mA, mB : bool) {
    update l$mA =
      merge step
        (false => (last l$mA when not step))
        (true => (not last l$mB when step))
    mA = l$mA
    update l$mB =
      merge step
        (false => (last l$mB when not step))
        (true => (last l$mA when step))
    mB = l$mB
  }
}

```

Listing 5.5: Stc code for `drive_sequence`

In more complex cases, where there are several inter-locking cycles, the pass should minimize the number of copies introduced. To do so, it builds a graph of the dependencies between state-variable updates. It then uses an algorithm to calculate the minimal feedback arc set [ELS93]. Given a directed graph, the algorithm returns a set of arcs such that the graph without these arcs is acyclic. Finding a minimum set is NP-hard problem: the algorithm is based on a heuristics and runs in linear time. Since this analysis does not need to be formally verified, we implement it in OCaml using the OCamlgraph [CFS07] library which includes an implementation of the algorithm contributed by Timothy Bourke.

Before we describe the OCaml analysis and the Coq transformation, we show the interface between them in [listing 5.7](#). The axiomatized function `cutting_points` takes as inputs the list of `last` and `next` state variables and the list of constraints in the system, and returns a list of state variables for which a copy must be introduced before the update.

```

system drive_sequence {
  last l$mA = true; l$mB = true;
  transition(step : bool) returns (mA, mB : bool)
  var stc$l$mB : bool; {
    stc$l$mB = last l$mB
    update l$mA =
      merge step
      (false => (last l$mA when not step))
      (true => (not stc$l$mB when step))
    mA = l$mA
    update l$mB =
      merge step
      (false => (last l$mB when not step))
      (true => (last l$mA when step))
    mB = l$mB
  }
}

```

Listing 5.6: `drive_sequence` node after cutting update cycles

```

Parameter cutting_points : list ident -> list ident -> list Syn.trconstr -> list ident.

```

Listing 5.7: Interface with OCaml analysis 🐔 [Stc/CutCycles/CC.v:26](#)

OCaml analysis The function is implemented in OCaml. It builds the graph of update dependencies according to the rules specified in [figure 5.32](#). For each constraint tc , the function collects the sets of state variables that must be updated before ($\text{Bef}_{next}(tc)$) or after ($\text{Aft}_{next}(tc)$) the constraint is applied. Both of these functions are defined recursively over the syntax of constraints and expressions; we only show the base cases for variables and `last` variables. The functions are parameterized by the set of next state variables $next$. A state variable that is updated with `update` (not in $next$) must necessarily be updated before it is read. Conversely, a state variable updated by `next` may only be updated after it is read. On the other hand, `last` variables may only be read before they are updated.

The sets returned by these functions are used to compute the edges of the graph. We write $x \rightarrow y$ to indicate that x must be updated before y . Copying the previous value of y into a buffer removes this scheduling constraint, but it is only possible if y is a state variable. Therefore, in the case of constraints defining a variable, we introduce a stronger arc $x \Rightarrow y$ which may not be cut by the feedback arc set algorithm.

The graph built following this rules is then passed to the `Fashwo.feedback_arc_set` function of OCamlgraph [[CFS07](#)], which returns the list of edges to cut; the `cutting_points` function simply returns the destination vertices of these edges.

Coq transformation and correctness For each variable x in the list of cutting points, the Coq transformation function introduces a copy constraint. We focus on the case where

$$\text{Bef}_{nxt}(x) \triangleq \begin{cases} \text{if } x \in nxt \text{ then } \emptyset \\ \text{else } \{x\} \end{cases} \quad \text{Aft}_{nxt}(x) \triangleq \begin{cases} \text{if } x \in nxt \text{ then } \{x\} \\ \text{else } \emptyset \end{cases}$$

$$\text{Bef}_{nxt}(\text{last } x) \triangleq \emptyset \quad \text{Aft}_{nxt}(\text{last } x) \triangleq \{x\}$$

constraint	$y \in \text{Bef}_{nxt}(tc)$	$y \in \text{Aft}_{nxt}(tc)$
<code>update</code> $x = e$, <code>next</code> $x = e$, <code>reset</code> $x = _$ <code>every</code> ck	$y \rightarrow x$	$x \rightarrow y$
$x = e$, $xs = f\langle i \rangle(es)$	$y \Rightarrow x$	$x \rightarrow y$

Figure 5.32: Graph Analysis for cutting state-variable update cycles

x is declared with `last`, the case for `next` being similar. A constraint $stc\$x = \text{last } x$ is added, and `last` x is replaced by $stc\$x$ in the rest of the body. This transformation is illustrated in [listing 5.6](#), where we show the `drive_sequence` system after cutting the update cycle between `1$mB` and `1$mA`. Here, the FASH algorithm has indicated that `1$mB` should be copied. This leads to adding an $stc\$1\$mB = \text{last } 1\$mB$ constraint, and renaming in `last` `1$mB` into $stc\$1\mB in the transition constraint for `1$mA`.

The function does not rename `last` `1$mB` in the constraint for `1$mB`; if it did, we would lose the syntactic relation that allows us to remove the `state(1$mB) := state(1$mB)` statement in the generated Obc code. In the Coq implementation, this is handled by an extra case that prevents renaming `last` x in a constraint that updates x . This extra case can never prevent the removal of a cycle, because updating a state variable x may always depend on the previous value of x .

The Coq implementation is relatively simple. To generate fresh identifiers for the new copies, it reuses the `Fresh` monad described in the previous chapter. To handle renaming, the semantic preservation proof reuses the notion of refinement of environment modulo substitution.

Possible improvement The transformation we outlined is still sub-optimal. In the example of [listing 5.6](#), $stc\$1\mB is declared on the base clock. This means that in the generated Obc code, the copy will be calculated in every cycle, while it is only necessary when $c = \text{true}$. To avoid unnecessary copies on cycles where $c = \text{false}$, a sampled variable could be introduced instead: adding constraint $stc\$1\$mB = \text{last } 1\$mB \text{ when } c$, and replacing the corresponding expression in the constraint updating `1$mA`. While this is straightforward in this specific example, more general case may have arbitrary combinations of sampling with other operators, which complicates not only the transformation and its verification, but also the interaction between the Coq program and the OCaml analysis.

To avoid mixing the complexities of this optimization with the transformation, we could also isolate it in a separate pass. A program analysis could infer, for each local variable x , the fastest clock ck on which it is used, and thereby downsample x to clock ck . This would cover the case explained above and possibly optimize some source programs. Vélus does not yet implement this optimization.

5.4.2.3 Scheduling algorithm

OCaml implementation Like the graph analysis used to cut cycles between state variables, the scheduling algorithm is implemented in OCaml and afterward validated by a Coq decision procedure. The interface between Coq and OCaml consists in the `schedule` function presented in [listing 5.8](#). It takes as input the name of the system (for printing error messages) and the list of the constraints in the system, and returns a list of positive integers indicating the position of each constraint in the scheduled system.


```
Parameter schedule : ident -> list tconstr -> list positive.
```

Listing 5.8: Interface between Coq and OCaml scheduling  [Stc/StcSchedule.v:52](#)

The underlying OCaml algorithm proceeds by building a graph of dependencies between transition constraints. Each vertex of the graph corresponds to a transition constraint and is annotated with its activation clock. Each edge corresponds to a dependency between constraints. This includes the dependencies between state variable updates that we highlighted in the previous section, as well as dataflow dependencies. A topological sort algorithm then chooses a scheduling from the graph. The heuristics used in the algorithm try to keep constraints with the same or similar clock types together, in order to improve the efficiency of the fusion optimization. We did not need to modify this algorithm in our work; the only necessary changes to support the new constructions were adding cases to the function that builds the dependency graph.

Coq implementation and semantic preservation The list of positions returned by `schedule` is used in the Coq code to sort the transition constraints. The body of the resulting `system` is thus a permutation of the source translation constraints. Since the semantics of Stc systems does not depend on the order of translation constraints in the system, the proof of semantic preservation is trivial.

Scheduling validator Finally, we must establish that the scheduled program respects the `WellScheduled` predicate. We cannot reason directly on the OCaml program. Instead, we use verified translation validation by implementing a function `well_sch` that takes the list of inputs, next variables and constraints of the scheduled program, and returns `true` iff the constraints are indeed well scheduled. This function is shown to be equivalent to the `WellDefined` predicate.

Lemma 31 (Correctness and completeness of `well_sch`  [Stc/StcSchedulingValidator.v:322](#))

$$\text{well_sch } ins \ nexts \ tcs = \text{true} \quad \text{iff} \quad \text{WellScheduled } ins \ nexts \ tcs$$

Overall, the definition of `well_sch` and the proof of its specification have the same structure as in previous work. The function is defined recursively on the list of constraints, with accumulators that track the sets of used, defined and updated variables in the

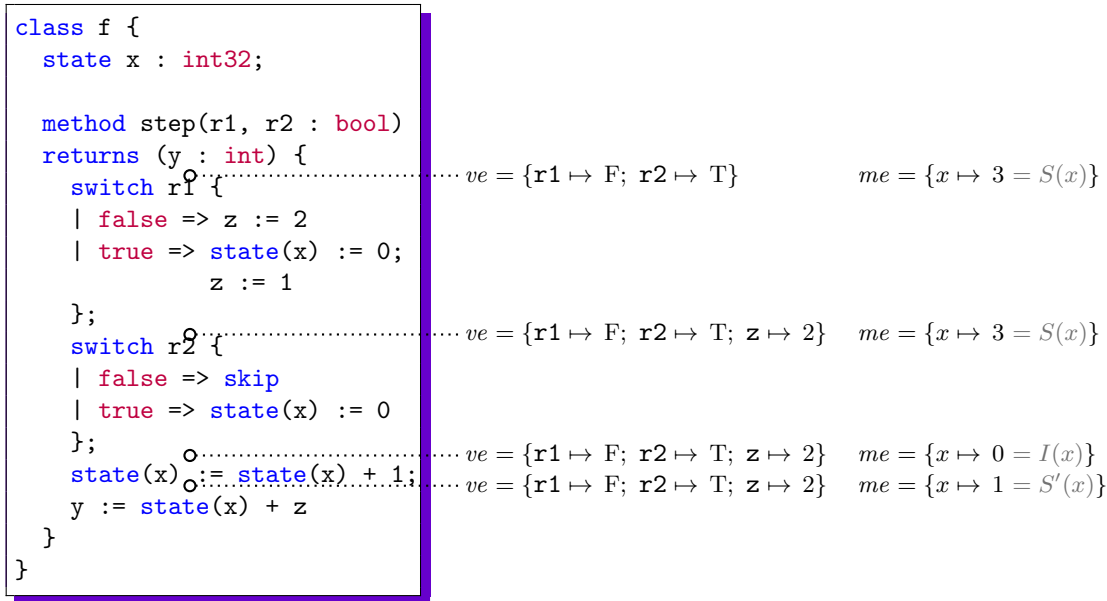


Figure 5.33: Relating Stc and Obc semantics

constraints already traversed, and checks that the rules of `WellDefined` are fulfilled for the last constraint in the list. The specification of [lemma 31](#) is shown by induction on the list, and case analysis on this definition. We simply extended the definition and proof with additional accumulators and checks for the new cases.

5.4.3 Stc to Obc Correctness proof

The semantic preservation proofs for the compilation of Stc to Obc relate the constraint-based semantics of Stc with the imperative, big-step semantics of Obc. The proof has been discussed in detail in [\[Bru20, §1.4.3\]](#). Rather than describe formally the whole proof, we outline the core ideas behind it using a specific example. We focus on the relation between the state of the Stc program and the memory of the Obc class, which is the component the most affected by our changes to the Stc language.

An Obc execution example Recall the example of [figure 5.22](#), which contains a state variable manipulated with `last`, `update`, and two `reset` constraints. To illustrate the relation between Stc and Obc behavior, we focus specifically on the 7th transition step of the execution shown in [figure 5.22](#). At this step, we have $r1 = F$, $r2 = T$, and $S(x) = 3$. By applying the semantic rules for each transition constraint, we deduce that $I(x) = 0$ since the second reset constraint is active, and therefore that $R(\text{last } x) = 0$, and finally $S'(x) = R(x) = 1$.

That program, once scheduled, compiles to the Obc class shown in [figure 5.33](#) at left (the `reset` method is omitted for concision). We now show how the transition system semantics acts as a specification for the memory updates in Obc. The contents of the

environment ve and memory me after each statement are indicated at right. Initially, the environment only contains the values of inputs and the memory contains the previous value of x . We assume it is 3, so that at the start of the execution we have $me(x) = S(x)$. Since $r1 = F$, the first `switch` statement does not modify the value of `state(x)`. The second `switch` statement executes the `state(x) := 0` statement. According to the semantics of state-variable assignments, the value of x in me is set to 0. The next statement updates the value of x in me to be 1. Finally, the last statement associates y to 3 in the environment.

At the end of the execution, the Obc environment corresponds to the Stc environment, and the memory corresponds to the next state S' . This means that the execution of the Obc step method correctly simulates the transition of the Stc system. This is stated formally on the next page, where \approx is an equivalence relation between Stc states and Obc memories.

Lemma 32 (Stc to Obc correctness 🐔 [StcToObc/Correctness.v:1651](#))

$$\begin{array}{l} \text{if } P, S, S' \vdash_{stc} f(xs) \downarrow ys \quad \text{and} \quad me \approx S \\ \text{then } \exists me', [P], me \vdash_{obc} f.step(xs) \downarrow (me', ys) \quad \wedge \quad me' \approx S' \end{array}$$

While this lemma specifies the expected memory at the start and end of the execution, it does not say anything about the memory throughout the execution. In particular, it does not specify its relation with I , the intermediate state that contains values after `reset` and before `update`. In the example, we have $I(x) = 0$, which matches the content of me after the second `switch` statement, that is, after the `reset` has occurred.

Proof by induction and memory invariant We now outline the proof of [lemma 32](#). It proceeds by induction on the list of transition constraints tcs of the source Stc system. However, the reasoning needs to be global. Indeed, we have seen in the previous example that the values in Obc memory me evolve at every step, while the values in the Stc states S , I , and S' are fixed. To bridge this gap, we express a correspondance invariant between Obc memory and Stc states. We write this invariant `MemoryCorres R b tcs S I S' me`, and present it on the next page. It relates the content of me with that of S , I and S' after the execution of the statements compiled from a given list of constraints tcs . It is also parameterized by the environment R and base clock b that give a semantics to these constraints.

Invariant 10 (Memory correspondance invariant 🦉 [Stc/StcMemoryCorres.v:53](#))

$$\begin{array}{l}
 \text{MemoryCorres } R \ b \ tcs \ S \ I \ S' \ me \\
 \text{iff } \forall x, \left\{ \begin{array}{l}
 \text{if } (\text{update } x = _) \notin tcs \\
 \text{and } \forall ck, (\text{reset } x = _ \text{ every } ck) \in tcs \implies R, b \vdash ck \downarrow F \\
 \text{then } me(s) = S(x) \\
 \text{and if } (\text{update } x = _) \notin tcs \\
 \text{and } \exists ck, (\text{reset } x = _ \text{ every } ck) \in tcs \wedge R, b \vdash ck \downarrow T \\
 \text{then } me(s) = I(x) \\
 \text{and if } (\text{update } x = _) \in tcs \\
 \text{then } me(x) = S'(x)
 \end{array} \right. \\
 \text{and } \forall x, \dots \\
 \text{and } \forall s, \dots
 \end{array}$$

The invariant comprises three conjuncts. The first, on which we focus, concerns the values associated with **last** state variables. The other two concern the values associated with **next** state variables and sub systems. We do not show them since they are structured similarly to the first. For each state variable x , there are three possible cases; in each case, the value of $me(x)$ is related to one of the three states S , I or S' .

1. If tcs does not contain the **update** constraint for x , and all the **reset** constraints for x in tcs are inactive (their clocks evaluate to F), then, in the imperative code, x has not yet been reset or updated, and therefore $me(x) = S(x)$. In the example, this is the case before the second **switch** statement.
2. If tcs does not contain the **update** constraint for x , but there exists an active **reset** constraint for x in tcs , then, in the imperative code, x has been reset, and therefore $me(x) = I(x)$. In the example, this is the case just after the second **switch** statement.
3. If tcs does contain the **update** constraint for x , then, in the imperative code, x has been updated, and therefore $me(x) = S'(x)$. In the example, this is the case after the **state(x) := state(x) + 1** instruction.

We now show how this invariant enables the proof of semantic correctness for this pass. Suppose that the constraints of the system are $tcs_1; tc_2; tcs_3$. Compiling them produces statements $stmts_1; stmt_2; stmts_3$. If we have already established $\llbracket P \rrbracket, me, ve \vdash_{obc} stmts_1 \downarrow me_1, ve_1$, where me and ve are the initial memory and environment, we need to prove $\llbracket P \rrbracket, me_1, ve_1 \vdash_{obc} stmt_2; stmts_3 \downarrow me', ve'$. This requires the existence of memory me_2 and environment ve_2 such that $\llbracket P \rrbracket, me_1, ve_1 \vdash_{obc} stmt_2 \downarrow me_2, ve_2$ and $\llbracket P \rrbracket, me_2, ve_2 \vdash_{obc} stmts_3 \downarrow me', ve'$.

Consider the first requirement. Suppose that the transition being compiled, tc_2 , accesses the value of **last** x . This access is compiled into the Obc expression **state**(x). To show that the Obc expression produces the same value as the Stc source expression, we need to show that $me_1(x)$ is equal to $R(\text{last } x)$. From the semantics of the **update**

constraint for x in the source program, we already know that $R(\text{last } x)$ is equal to $I(x)$. All we need to show is that $me_1(x) = I(x)$.

This is possible if we know that $\text{MemoryCorres } R b tcs_1 S I S' me_1$. Indeed, since $\text{last } x$ is read by tc_2 , we know, by the scheduling rules, that (i) there cannot be an **update** constraint in tcs_1 , and (ii) all potential **reset** constraints are in tcs_1 . The first point means that we are in case (1) or (2) of the invariant. If we are in case (2), we have $me_1(x) = I(x)$ trivially. If we are in case (1), we know that all the **reset** constraints are in tcs_1 , and we know that all their clocks evaluate to **false**. Since we know (syntactically) that one **update** constraint for x must exist in the system, we can use the first premise of the semantic rule for **update** which states that, if all **reset** constraints are inactive, then $I(x) = S(x)$. From this we can conclude that $me_1(x) = S(x) = I(x)$, which finally allows us to associate the correct value to $\text{state}(x)$.

To apply the inductive hypothesis and complete the inductive step, we must also prove that the MemoryCorres invariant is preserved for the updated memory me_2 under the new list of previous constraints, which includes tc_2 : $\text{MemoryCorres } R b (tcs_1; tc_2) S I S' me_2$. This is not trivial if tc_2 is a **reset** or **update** constraint, in which cases me_2 may be different from me_1 , and each relevant case of the invariant must be shown to be preserved.

5.4.4 Changes to the Fusion Optimization in Obc

The Obc generated from Stc has too many conditionals: one for each constraint that is not on the base clock. This simplifies the proof of semantic preservation, but such naive code is not efficient and must be optimized by fusing adjacent **switch** statements when their conditions are syntactically equal. In general, fusing **switch** statements does not preserve the program semantics, since an earlier statement may modify a variable or state variable tested by a later one. The previous version of Vélus defined a **Fusable** predicate [Bru20, Figure 4.6] formalizing the above condition. Any program translated from a well-scheduled Stc system was **Fusable**.

The example presented in figure 5.34 shows that this predicate no longer holds under our modified compilation scheme. In the Stc program (at left), the y is updated depending on its **last** value, and the value of z depends on the value of y . In the Obc program, this induces two **switches**, both on condition $\text{state}(y)$. Although these conditions are syntactically equal, their values differ, and the **switches** must not be fused.

Refining the fusion optimization The central case for the function that fuses **switches** is presented in figure 5.35. Compared to the earlier definition in [Bru20, Definition 4.5.1], we have generalized the function to **switch** on enumerated types, and changed the presentation somewhat, but the definition is essentially the same: when encountering two adjacent **switches** with the same condition, they are fused to produce one **switch**, and the statements in each branch are recursively fused. This definition is pleasantly simple, but, as we have seen in the above example, it may incorrectly translate Obc programs generated from Stc systems. To fix this issue, there are two possible solutions. First, the function could analyse the branches of the first **switch** to check that

<pre> system f { init y = true; transition(x:int) returns (z:int) { update y = case last y of (false => true) (true => false) z = case y of (false => 0) (true => x) } } </pre>	<pre> class f { state y : bool; method step(x:int) returns (z:int) { switch state(y) { false => state(y) := true true => state(y) := false }; switch state(y) { false => z := 0 true => z := x } } } </pre>
--	--

Figure 5.34: Non-fusible program

$$\begin{aligned}
 & [\text{switch } e \{ [| C_i \Rightarrow s_{1,i}]^i \}; \text{switch } e \{ [| C_i \Rightarrow s_{2,i}] \}] \triangleq \\
 & \text{switch } e \{ [| C_i \Rightarrow [s_{1,i}; s_{2,i}]^i \}
 \end{aligned}$$

Figure 5.35: Performing the fusion 🐔 [Obc/Fusion.v:44](#)

none of them update a variable used in e ; if they do, the fuse would not happen. This would prevent the issue raised in the previous example, but incur a compile-time cost and complicate the definition of the transformation.

The intuition for a better solution can be found by examining the example of [figure 5.34](#). Notice that the condition for the first `switch` is generated from expression `last y`, while that of the second is generated from expression `y`: at the `Stc` level, it is possible to syntactically differentiate the values of `y` before and after its update. The solution is to import this distinction into the syntax of `Obc`, by adding an optional annotation to the reading of state variables. In other words, we compile `last y` into `statelast(y)` and `y` into `statecur(y)`, where the annotation indicates whether the value of `y` is read before or after its update. This annotation is not interpreted in the `Obc` semantics. However, the equality test on expressions does take the annotations into account which suffices to prevent fusing the two `switches` from the above example, while still allowing fusion of two `switches` that both appear either before or after the update.

New Fusible invariant We now discuss the proof of correctness for the new implementation of the optimization. As in previous work, the proof depends on the `Fusible` invariant, which characterizes programs that may be fused safely. This invariant should hold for any `Obc` class compiled from a well-formed `Stc` system. It essentially transfers some of the information from the `Stc` scheduling invariant to the `Obc` syntax. With our

$$\begin{array}{c}
\overline{\text{Fusible } (x := e)} \qquad \overline{\text{Fusible } (\text{state } (x) := e)} \qquad \overline{\text{Fusible } ([x_i]^i := f(x).mx(es))} \\
\\
\frac{\text{Fusible } s_1 \quad \text{Fusible } s_2 \quad \forall x, x_{\text{upd}} \in \text{CanWrite}(s_1) \implies x_{\text{last}} \notin \text{Free}(s_2)}{\text{Fusible } (s_1; s_2)} \\
\\
\frac{\forall i, \text{Fusible } s_i \quad \forall x, x_{\text{cur}} \in \text{Free}(e) \implies \forall i, x \notin \text{CanWrite}(s_i) \quad \forall x, x_{\text{last}} \in \text{Free}(e) \implies \forall i, x_{\text{res}} \notin \text{CanWrite}(s_i)}{\text{Fusible } (\text{switch } e \{ [C_i \Rightarrow s_i]^i \})} \\
\\
\begin{array}{ll}
\text{Free}(x) \triangleq \{x_{\text{cur}}\} & \text{CanWrite}(x := e) \triangleq \{x_{\text{upd}}\} \\
\text{Free}(\text{state}_{\text{last}}(x)) \triangleq \{x_{\text{last}}\} & \text{CanWrite}(\text{state}(x) :=_{\text{res}} e) \triangleq \{x_{\text{res}}\} \\
\text{Free}(\text{state}_{\text{cur}}(x)) \triangleq \{x_{\text{cur}}\} & \text{CanWrite}(\text{state}(x) :=_{\text{upd}} e) \triangleq \{x_{\text{upd}}\}
\end{array}
\end{array}$$

Figure 5.36: Fusible invariant 🐔 [Obc/Fusion.v:505](#)

modifications to the Obc syntax and the Stc-to-Obc compilation pass, we modified the invariant greatly. We now motivate its new definition, presented in [figure 5.36](#). The invariant is defined inductively over the syntax of Obc statements. The base statements (assignment, state assignment, method call) are all trivially fusible. The more interesting cases are those for sequence and `switch` statements.

First, the rule for a sequence $s_1; s_2$ specifies that both s_1 and s_2 are themselves fusible. Then, it specifies a relation between the variables written in s_1 and those read in s_2 . Function $\text{Free}(s)$ specifies the set of variables read in a statement s , annotated with either `cur` or `last`, indicating whether the value read is that of x or `last` x . It is defined recursively over the syntax of statements and expressions; the base cases are presented at the bottom left of [figure 5.36](#). Only the current value of a local variable may be read. In the case of a state variable, the function is guided by the syntactic annotations. Function $\text{CanWrite}(s)$ characterizes the set of variables that can be written by statement s . The rule for a sequence specifies that the variables whose `last` value is read in s_2 cannot be written by s_1 . At first glance, this rule seems to correspond to the scheduling invariant, but it is actually too strong. Indeed, recall that Stc `reset` constraints are compiled to assignments, and must be scheduled before any read of `last`. Clearly, we must treat the compiled `reset` and `update` constraints differently in the invariant; to do so, we add annotations `res` and `upd` to the assign statements compiled from these two types of constraints. Function $\text{CanWrite}(s)$ also adds these annotations to the variable written by s . The premise for fusibility of a sequence $s_1; s_2$ then becomes “if x is updated in s_1 , then it must not be read as `last` in s_2 ”.

The rule for `switch` statements is, unsurprisingly, the most involved. First, it specifies that the statements of all branches are themselves `Fusible`. Then, it relates the variables read in the expression with the ones written by each branch. The first premise specifies

that the current value of x cannot be read if x is updated or reset in the branches (we write $x_$ to signify that the annotation may be either `upd` or `res`). The second specifies that the `last` value of x cannot be read if x is reset by the branches.

Fusibility of programs translated from Stc The `Fusible` invariant is respected for the translation of well-scheduled Stc programs. This is stated formally below: if a list of constraints tcs is `WellScheduled`, then its translation to Obc is `Fusible`.

Lemma 33 (Fusibility of translated Stc constraints 🐔 [StcToObc/Stc2ObcInvariants.v:293](#))

if `WellScheduled` $ins\ nexts\ tcs$ **then** `Fusible` $\llbracket tcs \rrbracket_{sts}$

The proof of this lemma proceeds by induction on the list of constraints and inversion of the scheduling hypothesis. There are two non-trivial obligations in the proof. The first is encountered when compiling a non-empty list of constraints $tcs_1; tc_2$, which produces a sequence of Obc statements $\llbracket tcs_1 \rrbracket_{sts}; \llbracket tc_2 \rrbracket_{sts}$. To prove that this sequence is `Fusible`, we must prove that the variables updated in tcs_1 are not read with `last` in tc_2 . This is a direct consequence of the third premise of `WellScheduled` (cf. [figure 5.31](#)). The other concerns the compilation of a constraint tc that produces a `switch` statement, either because of its clock (with `control`), or when compiling a `merge` or `case`. In all of these cases, we must prove that the variables read in the condition of the `switch` are not updated by the generated statement. This is always true, by the scheduling hypothesis: a variable x defined with `last` may only be read after it is reset and updated; in particular, if x is reset or updated by tc , then it cannot be read by tc or its clock; therefore, it cannot be read by the Obc statement generated from tc . Similarly, `last` x may not be read before x is reset; in particular, if x is reset by tc , then it cannot be read by tc or its clock.

This informal reasoning is relatively easy to mechanize in Coq. [Lemma 33](#) is proven with the help of a few additional syntactic invariants of Stc systems.

Correctness of fusion Having proven that the `Fusible` invariant holds for Obc programs compiled from Stc, we now show that it suffices to prove the semantic correctness of the fusion optimization. The main correctness lemma is presented below. It states that, if the source statement is `Fusible`, then the compiled statement has the same semantics as the source.

Lemma 34 (Correctness of fusion 🐔 [Obc/Fusion.v:895](#))

if `Fusible` s **and** $P, me, ve \vdash_{obc} s \downarrow me', ve'$ **then** $P, me, ve \vdash_{obc} \llbracket s \rrbracket \downarrow me', ve'$

The non-trivial case of the proof is the one that actually fuses two `switches`, following the transformation presented in [figure 5.35](#). The semantic hypothesis for this case is

$$P, me, ve \vdash_{obc} \text{switch } e \{ [| C_i => s_{1,i}]^i \}; \text{switch } e \{ [| C_i => s_{2,i}] \} \downarrow me', ve'$$

Inverting it reveals an intermediate memory ve_1 and environment me_1 that have been updated by one of the branches of the first `switch`, but not by the second. In both the

source and compiled statement, the statements contained in the branches of the second `switch` are executed in the context me_1, ve_1 . Therefore, to prove that the fused `switch` has the same semantics, it suffices to prove that, in the source program, both evaluations of condition e produce the same value. Since the evaluation of expressions is deterministic, this amounts to proving that e has the same semantics under contexts me_1, ve_1 and me, ve . This is true if none of the variables read in e are updated in memory by the first `switch`, which we can prove using the Fusible invariant. There are two possible cases:

- if x is read as `last` in e , then, by inversion of the Fusible rule for `switch`, it cannot be updated as `res` in the first `switch`. Additionally, by inversion of the rule for sequences, this means it cannot be updated as `upd` either.
- if x is read as `cur` in e , then, by inversion of the rule for `switch`, it cannot be updated as `res` or `upd` in the first `switch`.

5.5 Discussion and Related Work

5.5.1 Compilation to CompCert Clight and Beyond

The ultimate target of Vélus is Clight, the input language of the CompCert verified C compiler [Ler09a]. We did not need to modify the existing pass that generates Clight from Obc [PLDI17]. The Clight program is compiled by CompCert into assembly language by a verified compilation chain. It goes through several intermediate languages, each with its own semantic model. They can be divided broadly into three categories. First, subsets of C with simplified control. The semantics of early languages is given with a relational big-step model and later languages are specified by small-step semantics. Second, Control Flow Graph (CFG) languages which are defined as graphs where nodes are instructions. A node may have several successors (conditional branching), and predecessors. The first of these languages, RTL, uses pseudo registers. RTL programs are compiled into LTL which uses machine registers allocated by the register allocation pass. Last, the CFG representation is linearized into a textual, linear representation of the assembly code.

Most of the optimizations implemented in CompCert occur around the RTL language: function inlining, constant propagation, common subexpression elimination, dead code elimination, optimizations related to register allocation, etc. The optimizations implemented in Vélus may be redundant with those of CompCert for some programs, but in other cases, the simplicity and strong static invariants of the dataflow language allows NLustre-level optimizations to be more aggressive than those defined at the CFG level.

Some of the passes of CompCert use verified translation-validation. In particular, register allocation is implemented using a graph-coloring algorithm implemented in OCaml. The non-verified pass takes as input an RTL program and returns a compiled LTL program where the machine registers have been allocated. A verified validator then checks that the two are equivalent. This is more complex than the approach we use for scheduling, where the scheduler simply returns the list of transition numbers; this makes sense, since

the CompCert pass translates to a new intermediate language with a completely different AST.

5.5.2 Verified Compilation in CakeML

The back-end of CakeML [Tan+19] translates a subset of SML into machine code. Like CompCert and Vélus, it includes several intermediate languages, each with their own semantics, specified in a functional big-step style. Most of the compilation passes of CakeML are proven directly with respect to the big-step functional semantics of these intermediate languages. Some passes, such as one implementation of the register allocation, use translation validation.

The abstract values used in the source semantics of CakeML may be immediate numbers, allocated blocks or closures. Later, closures are represented by allocated blocks. One of the most delicate passes of CakeML is the one that transforms these abstract values into their concrete, machine representation. Vélus avoids this difficulty because it is more restricted than CakeML, and therefore does not need allocated blocks, and because the scalar values of Vélus are those of the host compiler, CompCert. Only enumerated values require a special treatment in the pass that compiles Obc into Clight code.

5.5.3 Translation Validation of Dataflow Programs

In [Aug13], the author describes the verified scheduling of a dataflow language. As in Vélus, scheduling is based on translation validation, where the validator checks that the returned set of equations is a permutation of the source. Notably, the validator does not check that the equations are correctly scheduled; this check is done by the following pass, which generates imperative Obc code. To do this, and to facilitate the proof of correctness for this pass, the validator manipulates the sets of defined and used variables explicitly.

The dissertation also describes some imperative optimizations, and sketches possible correctness proofs. In particular, it discusses the fusion optimization. Since scheduling is defined on NLustre equations directly, which means `resets` of nodes may not be scheduled separately from updates; this might lead to the generation of imperative code where conditionals cannot be fused optimally. The author proposes several solutions to improve this situation. First, he suggests that the order of Obc statements may still be changed to try to bring similar `switches` close together. This transformation preserves semantics as long as dependencies between instructions are respected; however, we are not convinced that the correctness proof would be straightforward, as this means changing the intermediate memories and environments. Another suggested transformation is to add `switch` statements to previously uncontrolled statements. For example, the statement `y = 1` could become `switch x { | True => y = 1 | False => y = 1 }`. If the statement appears in between two `switch` statements on condition `x`, then this transformation allows for fusion of the three. We believe that this transformation would be easy to verify using our Obc semantics.

Conclusion

In this dissertation, we presented an extension of the Vélus language with control blocks including hierarchical state machines. We introduced a novel semantics for these constructions. We showed the proof techniques used to establish fundamental properties of the semantics. Finally, we proposed some adaptations to the well-established compilation scheme for these constructions, and showed how the Vélus semantic model facilitates the correctness proof of each compilation pass.

The compilation passes are chained with the compilation function that generates Clight from an Obc program, and finally with the compilation function from Clight to assembly provided by CompCert. This results in a `compile` (🐦 [Velus.v:147](#)) function that transforms an untyped Vélus AST into an assembly program. [Theorem 1](#) shows that, for each elaborated Vélus program G for which a semantics exists compiled into assembly code P , the iterated semantics of P simulates the dataflow semantics of G . The theorem is shown by composing the proofs of semantic preservation for each transformation pass, along with the proof of correctness for the CompCert compiler [[Ler09a](#)].

6.1 Experimental Evaluation

The Coq `compile` function is extracted to an OCaml program, and compiled to an executable compiler for Vélus. [Theorem 1](#) assures that the code generated by this compiler is correct, but does not say anything about its efficiency. For safety-critical embedded software, we are interested in minimizing the required memory and estimated Worst Case Execution Time (WCET). The efficiency of the generated code, with regards to these metrics, depends on the compilation scheme, the optimizations, and the back-end compiler. Like the Scade KCG [[CPP17](#), Figure 4] and the academic Heptagon [[Gér+12](#), §6] compilers, Vélus implements *clock-directed modular code generation* [[Bie+08](#)] extended with source-to-source transformations in the front-end. We have implemented this scheme without compromising to simplify its formalization and proof. In addition, Vélus provides some optimizations that exploit properties of the source or intermediate languages; other optimizations that rely on machine-level details are provided by CompCert.

Memory Usage The compilation scheme guarantees compile-time bounds on the memory required by the generated code: recursion is not permitted, which bounds the stack size, and a heap is not used. Therefore, the main focus should be the stack size and the size of the states which are statically allocated. The former mostly depends on the number of local variables, which will typically be optimized during register allocation in the back-end. To minimize the latter, a front-end should concentrate on the state variables, since a back-end cannot normally optimize them. The *fbj-minimization* pass presented in [section 5.2.3.3](#) reduces the number of **fbj** equations in the NLustre program, and therefore the number of state variables in the compiled program.

Another optimization that could reduce the amount of memory used by state variables is related to state machines. Suppose an automaton with only **then** transitions. In this automaton, states are always reset on entry. Since the values of state variables are always “forgotten” by transitions, the state variables of the different states could be allocated in overlapping blocks of memory, for example by using a **union** in Clight. We do not implement this optimization, and believe that it would require significant changes to the whole Vélus development. Some exclusivity annotations would need to be added to each intermediate language to inform the generation of Clight code. A mutual-exclusion invariant that ensures the correctness of these annotations would then need to be expressed on the source language, where it is obvious from the structure, and carried through intermediate passes, where the structure is obscured. Finally, this invariant would justify the correctness of the generation of Clight code.

Worst Case Execution Time The WCET of a function is an estimation of the worst-case number of CPU cycles necessary to execute it. In the context of synchronous dataflow programs, we measure the WCET of the **step** function generated for the main node of a program. Unfortunately, there is no established set of benchmarks for the Lustre language. Instead, we use our own programs and programs from the literature as our benchmark. We estimated the WCET using OTAWA v2.3.0 [[Bal+10](#)]. [Table 6.1](#) presents the WCET of the assembly program generated by Vélus and CompCert, compared to that of the program generated by Heptagon with either CompCert or GCC. The first column displays the WCET of the programs generated by Vélus with all optimizations active. To evaluate the effect of these optimizations, we also estimated the WCET of each program when disabling some of them. These intermediate results are not included in the table to keep it simple.

As we hypothesized, disabling the fusion optimization has the most impact on performances, incurring a +63% increase of the overall WCET for these benchmarks. Indeed, these particular programs make heavy use of block-based constructs which induce branching in the generated code. We also tried replacing the scheduling heuristics described in [section 5.4.2.3](#) with a simpler algorithm that returns a valid scheduling of Stc constraints, without trying to maximize the effectiveness of fusion. Surprisingly, this only incurs a +8% WCET increase overall. There are two explanations for this relatively small performance loss. First, the scheduling algorithm is a heuristic and does not necessarily find the best possible schedule, which is an NP-hard problem. In some cases, it might

	<i>Vélus</i>	<i>Hept+CC</i>	<i>Hept+gcc</i>	<i>Hept+gcci</i>
buttons [CPP17]	1005	1430 (42 %)	625 (-37 %)	625 (-37 %)
chrono [CPP05]	500	970 (94 %)	570 (14 %)	570 (14 %)
cruisecontrol [Pou06]	1385	1680 (21 %)	880 (-36 %)	830 (-40 %)
heater [Pou06]	2415	3110 (28 %)	725 (-69 %)	500 (-79 %)
stepper motor	930	1185 (27 %)	605 (-34 %)	520 (-44 %)
stopwatch [CPP17]	1255	1970 (56 %)	1280 (1 %)	1280 (1 %)

Table 6.1: Estimated WCET in cycles by OTAWA 2.3.0 [Bal+10] for armv7-a using CompCert 3.11 (CC) and GCC 12.2.1 at -O1 without inlining (gcc) and with inlining (gcci); percentages are relative to the *Vélus* column

even produce a worst schedule than the naive algorithm. Second, the body of some benchmarked nodes (stopwatch, buttons) are defined as a single state machine where each state only contain simple equations. In the code generated from such a state machine, the top-level conditionals may always be fused; therefore, the scheduling heuristic has less importance.

Not removing dead NLustre equations as described in section 5.2.3.2 increases the WCET by 6%. There is no dead equation in the benchmark sources, which means the equations removed by these optimizations are those introduced by the compilation of `switch` blocks.

Finally, we evaluate the improved compilation of `last` variables, which permits the optimization of dead update equations of the form `state(x) := state(x)`. On the benchmarks that use `last` variables, using the simpler compilation scheme that compiles `last` to `fbv` early raises the WCET by 4%. Interestingly, deactivating only the optimization that actually removes the dead updates does not change the WCET. It seems that, when using the improved compilation for `last`, the dead-code elimination of CompCert is able to remove the code generated from these updates.

We then compare the WCET of the code generated by *Vélus* with that generated using Heptagon 1.05 and different back-end C compilers. The most favourable comparison is with Heptagon using CompCert as a back-end; in that case, *Vélus* is systematically faster than Heptagon. This is due to the combination of the optimizations implemented in *Vélus* but not in Heptagon. However, our experimental results also show that the gains from front-end optimizations are largely outdone by the back-end optimizations of GCC. Indeed, while one of CompCert’s advantages is to provide optimizations with formal correctness guarantees, these optimizations are less aggressive than that of GCC, even when limited by the -O1 flag. In particular, CompCert’s inlining heuristic is not fine-tuned to handle the many small functions generated by *Vélus*.

6.2 Proof Engineering and Practical Concerns

A major challenge of our work was to implement and verify an efficient compilation scheme for a realistic language while keeping the language specification understandable

Feature	Executable	Specification	Proofs	Total	Compile time (min) Effort (months)	
Vélus v3 (w/ datatypes)	3965	21 439	46 996	72 400	35	
<code>reset</code> blocks	+199	+1829	+4590	+6618	42	3
remove anonymous variables	+4	-793	-2015	-2804	49	1
local blocks	+142	+1647	+5048	+6837	50	4
<code>switch</code> blocks	+134	+1126	+3680	+4940	53	2
<code>last</code> variables	+246	+770	+254	+1270	55	2
state machines	+465	+1776	+2563	+4794	60	5
completion	+88	+581	+1718	+2387	62	2
better <code>last</code> compilation	+504	+1754	+4007	+6265	65	3
Vélus v4	5747	30 119	66 841	102 707	65	

Table 6.2: Number of Coq LoC for each added feature in Vélus

and the correctness proofs simple. While we think we have achieved the first two points, we are not completely satisfied with the last.

To discuss the size of the proof, and the effort behind it, we present in [table 6.2](#) the “cost” in terms of number of Coq Lines of Code (LoC) for each major feature added to Vélus and discussed in this dissertation. We split the LoC into three categories: (i) the definitions that are extracted to OCaml code and actually executed in the compiler, (ii) the specification which includes the dynamic and static semantic definitions as well as the statements of all intermediate lemmas, and (iii) the proofs written in Ltac. The fifth column displays the sum of these three categories. We omit from this total the administrative (module definition and instantiation), comments and blank lines. The number of LoC was measured using a tool based on `coqwc`¹. The table also presents the cumulative time necessary to compile Vélus at each step. These measurements were done using 4 cores (`make -j4`) on a system with an Intel i7 running at a base clock speed of 1.80GHz, and 16Go of RAM. This does not include the time necessary to compile CompCert, which is invariant. The last column shows an estimate of the effort that was necessary to implement the feature, in months, for a single developer. This does not include the language design, which was done before the start of this thesis.

The first row corresponds to the version of Vélus that handles just the core dataflow language, with normalization [EMSOFT21] and enumerated types. In this version of Vélus, the executable code represents around 6% of the total LoC. In the subsequent rows, we distinguish two kind of features: those that affect only the front-end of Vélus (local and `switch` blocks, state machines, basic `last` variables, completion), and those that require changes to the whole compiler (`reset` blocks, improved compilation of `last` variables).

¹<https://github.com/coq/coq/blob/master/tools/coqwc.ml1>

The latter usually require more effort. However, features that only affect the front-end may also be costly, especially in terms of specification and proof. For instance, arbitrarily nested local blocks are an unassuming feature that required a lot of work. This is due, in part, to the generalization of the dependency analysis and the proof schemes described in [chapter 3](#). Recall also that local blocks are used extensively in the compilation of other features; in a sense, the time spent on this construction is capitalized on by later, more complex constructs.

The removal of anonymous variables is a simplification of the clock typing of node instantiations that resulted in the rules presented in [section 2.4.4](#). In the previous version of Vélus, instantiations of nodes with clock-type dependencies between outputs could be nested. Specifying the clock typing of such instantiations required the use of unique, *anonymous* variables at the instantiation point. Handling these variables became difficult, especially with the introduction of nested local declarations, and we decided to remove them. This limits the language somewhat, but we believe avoiding this compromise and introducing the subsequent constructions would have increased the complexity of the proofs to an unmanageable level.

In the previous section, we showed that the WCET impact of using the more optimized compilation scheme for `last` variables, on our benchmarks, is just 4%. We now see that the effort necessary to achieve this gain accounts for 6% of the total LoC of the compiler. Considering the small performance gains, it is unclear if this effort was worth it. Indeed, in addition to the existing effort, this modification complicates the intermediate languages used in the compiler, which may make future extensions more difficult. On the other hand, independently of the performance gains, the extension of intermediate languages clarified some design choices, especially regarding `Stc`.

The final Vélus project, with all these features, still has a code-to-total ratio of 6%. This ratio is considerably lower than that of CompCert, which is around 17%. This is not an issue in and of itself, since this discrepancy may be due to several justified factors: the differences between the semantic models of the intermediate languages of Vélus, the number of static invariants used for each language, the complexity of the source language itself, etc. However, we are more concerned about the maintainability of Vélus. We have seen that even adding or removing a small feature (anonymous variables, completion) has a significant impact on the code size. In addition, [table 6.2](#) only shows the relative diff. The absolute number of LoC modified to implement each feature may be between 2 and 10 times as large. In addition, the amount of work necessary for future extensions of Vélus would not be constant. Suppose that we were to add a new control block, that compiles after `switch` blocks but before local declarations. This would impact all the compilation passes before its elimination: completion, state machines, `switch`. Even if the treatment of this block by these passes is trivial, the corresponding proofs may not be, due to the complexity of some inductive invariants. This is a serious engineering issue, which may severely limit, or at least slow down, future extensions of Vélus. We see three possible mitigations to this problem, which we discuss from least to most invasive.

A first avenue might be to dedicate more time to proof automation. With well thought-out tactics designed to prove recurring obligations related to the different intermediate

languages, trivial cases that currently require a lot of busy work could be automated. This kind of automation has two downsides. First, it generally involves automatic proof search, which may increase proof-checking time. Second, and more importantly, while an automated proof may be more resistant to changes in the definitions and implementation, if it “breaks”, then “fixing” the proof may be more difficult, as it is not clear which part of the automated tactic needs to be modified to fit the new definitions.

A complementary approach would be to rewrite some of the definitions of Vélus to simplify the proofs. While we do not want to modify the semantic definitions, some of the static invariants (typing, clock-typing, other well-formedness invariants) may be combined to form definitions that are easier to manipulate in proofs. Another possibility would be to define a generic framework for program transformations, which specific compilation passes would be instances of. The repetitive, boilerplate proofs could then be established once for this framework, and the proof for each compilation pass would focus on the interesting details. This is an attractive idea, but we do not yet have a concrete idea of what such a framework might look like.

A final idea stems from the observation that a significant amount of the Vélus proofs concern the complex static invariants used in the source and intermediate languages. Indeed, the correct typing and clock-typing of a Vélus program are checked at the start of the compilation chain, and are necessary up to the very end of the chain. Therefore, each of the numerous compilation passes must provably preserve typing and clock typing. This is not necessary in CompCert: when a semantic correctness proof requires the source program to be well-typed, a decision procedure simply re-checks the types of the program. If the procedure succeeds, the program is proven to be well-typed; if it fails, then the compiler aborts. This is a kind of translation validation. Doing the same in Vélus would reduce the proof effort to proving that (i) the decision procedures are correct, and (ii) the dynamic semantics are preserved by each compilation pass. Adopting this approach would have the usual downsides of translation validation. It would increase the time to compile programs, due to the numerous re-checks. Also, if a compilation pass contains an error, it may generate code that is not statically well-formed, which will be rejected by a later check, aborting the compilation. To find such bugs, more testing of the compiler would be required.

6.3 Open Questions

If the proof complexity can be kept in check, using one or several of the above techniques, Vélus could still be a fertile ground for experimentation on the verification of synchronous languages and programs. We are currently considering several projects that could leverage the existing work.

Functional Semantics The relational semantics currently used in Vélus has proven to be useful for proving the correctness of program transformations. However, this model is sometimes too abstract. We have seen that establishing its determinism requires a complex proof. Conversely, we have not proven the existence of a semantic model: we do

not know if all (or even if any) statically well-formed programs have a semantics under this model. Proving this last property is especially difficult, as it requires constructing a witness of the history that is at the heart of our model. Executable semantic models might answer these issues. We have experimented with implementing a constructive state-based interpreter for the Vélus language, following [Col+23], but have not invested enough time into refining the definitions or proving its correspondence with the relational model. Other works are currently investigating a stream-based denotational model for Vélus, where the semantics of an expression is given as a Coq function that produces a (possibly finite) stream. The correspondence proof between this model and the current relational model has made significant progress [BJP22]. This model also shows promise for proving program properties. It is not yet clear how easy it will be to extend this model to the control blocks discussed in this dissertation.

Additional Optimizations In the previous chapter, we outlined a few optimizations that Vélus does not yet implement: constant propagation, node inlining, generalized dataflow minimization. Implementing and verifying these optimizations at the NLustre level should not be too difficult. In order to evaluate their benefits, it would be useful to develop a larger set of benchmarks, possibly compatible with other dataflow-synchronous compilers (Heptagon, Lustre v6, Scade 6).

Type-Based Causality Analysis Having implemented node inlining would also allow for the compilation of node applications with non-atomic dependencies, such as the ones described in [section 3.6.1](#). To reason on the dependencies of such programs, we would also need to implement and verify a type-based modular causality analysis. It is not clear how this analysis would impact the proofs of determinism and clock correctness of Vélus.

Arrays and Records One major feature of Scade 6 is the support of functional arrays and iterators [Mor07; CPP17]. An array of size n may be used to define n parallel computations, n -ary delays, etc. They also allow for the definition of matrix operations. Compiling arrays poses two major difficulties. First, the size of arrays must be known statically; this requires a separate analysis, which may be complex for an expressive language with type inference [CPP23]. Second, while these arrays are manipulated functionally, their efficient compilation requires at least eliminating intermediate copies, and at best compiling functional updates into in-place modification. We are aware of work in that direction, but it is not yet clear how this would be mechanized in a proof assistant.

6.4 Concluding Remarks

Although the language described in this dissertation is not novel, we believe that treating it in an ITP has made some of the design choices and challenges more explicit. The relational dataflow semantics highlights the correspondence between dataflow primitives and state machines, which was already reflected in the first compilation scheme for these

constructs. Working with an ITP has also shown that, despite the apparent simplicity of the definitions, the mechanized reasoning requires a quantity of technical details that can become difficult to manage. These details are unavoidable when treating a realistic language without compromise, and should inform future design decisions.

We have also felt the inertia that comes with complex and time-consuming proofs: once a design choice is made, going back on it is costly. In a sense, this is the same problem encountered when developing industrial qualified software (like Scade), where the cost of qualification prevents rapid iteration. For this reason, it seems to us that verified compilers implemented using proof assistants are best suited for well-established, well-formalized languages. In that regard, our methodology for Vélus could have been more efficient. Indeed, we worked iteratively, by adding features to the language one by one, which leads to rewriting some proofs several times. If we were to start Vélus again from scratch, it would be more efficient to start directly with the full language, and designing the compiler from front-end to back-end.

We believe that our work could be applied to an industrial context. While it seems unrealistic to completely verify an existing industrial compiler, we can see several ways of integrating ITPs in the development and qualification process. Defining the semantics of the language, and proving its desirable properties may increase confidence in the language design, and be used for documentation purposes. The Ott tool [Sew+07], which was used to typeset inference rules in this thesis, may also be used as a common source of truth to generate both the documentation for a language and mechanized definitions for an ITP. An interpreter verified with regard to these semantics could be used for testing the compiler by comparing the interpretation of the source and compiled programs. A complex compilation pass or optimization algorithm could be implemented and verified in the ITP, and either integrated with the compiler, or used separately for testing and comparison with the compiler. All of these ideas exploit mechanized formalization to increase the confidence in the correctness of the compiler, facilitate the qualification process, and clarify language and compiler design.

Type Systems and Static Predicates of Vélus

A.1 Node Invariants

In this section, we detail the static node predicates expected of valid Vélus programs. These invariants are used extensively in compilation proofs. They appear as dependent fields of the `node` record, that is, directly in the Lustre AST.

A.1.1 Variables Defined

The first invariant, presented in [figure A.1](#), ensures that the variables defined by equations correspond exactly to the declared outputs and local variables of the node, with no duplication. The variables defined by a block are modeled by a list. The rule for equations simply exports the variables at left of the equation. A `last` equation does not define the current value of any variable. A `reset` blocks defined the same variables as its underlying blocks. The underlying blocks of a local declaration should define at least the declared local variables.

The rule for `switch` and state machines are more complicated: we decompose them in two. Each branch of a `switch` should define a subset Γ' of the defined variables of the whole `switch` Γ . The variables that do not belong in this subset, that is Γ_l , are the implicitly defined ones, and so they should all be declared with a `last`. To ensure they are, the lists indicate, for each variable, whether or not it is declared with a `last` using a boolean, that is accessed by reading $\Gamma(x)$. Additionally, the domain of the causality label substitution σ should be included in the defined variables; this is imperative when proving properties of causal nodes.

The same rule applies for the state of a state machines, composed with the prerequisite of local declarations. Finally, a node is well-formed if the variables defined by its block are a permutation of the declared outputs of the node.

We also need to constrain the definitions of initial values of `last` variables. This is accomplished by the `LastsDefined` invariant of [figure A.2](#). Only `last` equations declare

$$\begin{array}{c}
\frac{}{\text{VarsDefined}([x_i]^i = es)[x_i]^i} \qquad \frac{}{\text{VarsDefined}(\mathbf{last} \ x = e)[]} \\
\frac{\text{VarsDefined} \ blks \ \Gamma}{\text{VarsDefined}(\mathbf{reset} \ blks \ \mathbf{every} \ e) \ \Gamma} \qquad \frac{\text{Permutation}(\Gamma + locs) \ \Gamma' \quad \text{VarsDefined} \ blks \ \Gamma'}{\text{VarsDefined}(\mathbf{var} \ locs \ \mathbf{let} \ blks \ \mathbf{tel}) \ \Gamma} \\
\frac{\text{Permutation} \ \Gamma \ (\Gamma' + \Gamma_l) \quad \text{VarsDefined} \ blks \ \Gamma' \quad \forall x, x \in \Gamma_l \implies \Gamma_l(x) = \mathbf{T} \quad \sigma \subseteq \Gamma}{\text{VarsDefined}(C \ \mathbf{do}_\sigma \ blks) \ \Gamma} \\
\frac{\forall i, \text{VarsDefined} \ br_i \ \Gamma}{\text{VarsDefined}(\mathbf{switch} \ e \ [(br_i)]^i \ \mathbf{end}) \ \Gamma} \\
\frac{\text{Permutation} \ \Gamma \ (\Gamma' + \Gamma_l) \quad \text{Permutation}(\Gamma' + locs) \ \Gamma'' \quad \text{VarsDefined} \ blks \ \Gamma'' \quad \forall x, x \in \Gamma_l \implies \Gamma_l(x) = \mathbf{T} \quad \sigma \subseteq \Gamma}{\text{VarsDefined}(\mathbf{state} \ C \ \mathbf{var}_\sigma \ locs \ \mathbf{do} \ blks \ \mathbf{until} \ trunt \ \mathbf{unless} \ trunl) \ \Gamma} \\
\frac{\forall i, \text{VarsDefined} \ autst_i \ \Gamma}{\text{VarsDefined}(\mathbf{automaton} \ \mathbf{initially} \ autinits \ [(autst_i)]^i \ \mathbf{end}) \ \Gamma} \\
\frac{\text{Permutation} \ outs \ \Gamma \quad \text{VarsDefined} \ blk \ \Gamma}{\text{VarsDefined}(\mathbf{node} \ f(ins) \ \mathbf{returns} \ (outs) \ blk)}
\end{array}$$

Figure A.1: Checking Defined Variables 🐦 [Lustre/LSyntax.v:276](#)

last variables. The rule for local declarations only take into account the variables that are declared as **last**. These declarations may “cross” **reset** blocks, but not **switch** or state machines. This is because, for the compilation model to be efficient, a **last** should be initialized with a constant. This is not the case if the initialization appears under a switched block, and so we forbid this possibility; this also simplifies slightly some proofs.

A.1.2 No Duplication in Declarations, No Shadowing

The **NoDupLocals** invariants presented in [figure A.3](#) ensures both that there is no name duplication in any local declaration, and that, as explained in [section 2.7](#), there is no shadowing of global variables by local declarations. It is trivially true for equations and **last** equations. The **NoDupMembers** predicate ensures that there is no name duplication in a list of declarations. When encountering a local declaration, we need to check that (1) its declarations do not contain duplicates, (2) they are disjoint from the global ones in context Γ , and (3) the underlying blocks are also **NoDupLocals**, under the augmented

$$\begin{array}{c}
\frac{}{\text{LastsDefined}([x_i]^i = es) []} \qquad \frac{}{\text{LastsDefined}(\mathbf{last} \ x = e) [x]} \\
\\
\frac{\text{Permutation}(\Gamma + \text{filter haslast } locs) \Gamma' \quad \text{LastsDefined } blk s \Gamma'}{\text{LastsDefined}(\mathbf{var} \ locs \ \mathbf{let} \ blk s \ \mathbf{tel}) \Gamma} \\
\\
\frac{\text{LastsDefined } blk s \Gamma}{\text{LastsDefined}(\mathbf{reset} \ blk s \ \mathbf{every} \ e) \Gamma} \qquad \frac{\forall i, \text{LastsDefined } blk s_i []}{\text{LastsDefined}(\mathbf{switch} \ e \ [(C_i \ \mathbf{do} \ blk s_i)]^i \ \mathbf{end}) []} \\
\\
\frac{\text{LastsDefined } blk s (\text{filter haslast } locs)}{\text{LastsDefined}(\mathbf{state} \ C \ \mathbf{var}_\sigma \ locs \ \mathbf{do} \ blk s \ \mathbf{until} \ trunt \ \mathbf{unless} \ trunt)} \\
\\
\frac{\forall i, \text{LastsDefined } autst_i}{\text{LastsDefined}(\mathbf{automaton} \ \mathbf{initially} \ autinits \ [(autst_i)]^i \ \mathbf{end}) []} \\
\\
\frac{\text{Permutation}(\text{filter haslast } outs) \Gamma \quad \text{LastsDefined } blk \Gamma}{\text{LastsDefined}(\mathbf{node} \ f(ins) \ \mathbf{returns} \ (outs) \ blk)}
\end{array}$$

Figure A.2: Checking Defined Last Variables 🐔 [Lustre/LSyntax.v:378](#)

context $locs + \Gamma$. The other rules for blocks are unsurprising; note that the predicate also ensures that there is no duplication in the keys of causality label substitutions. Again, this is necessary to prove properties of causal nodes. Finally, a node is well formed if there is no duplication in its global variables (inputs and outputs), and if its body is well formed.

A.1.3 Shape of identifiers

The final static node obligations concern the identifiers used within a node. It is used to justify the freshness of identifiers generated between each pass, as discussed in [section 4.2](#). It essentially consists in applying the `AtomOrGensym` predicate, presented in [listing 4.5](#), to all declarations in the node. The rules defining this judgement are given in [figure A.4](#).

A.2 Type System

The type system of Vélus is straightforward. It includes scalar types from the back end, and enumerated types. We write $[C_i]^i$ for the enumerated type declared with constructors C_i . To be well formed, enumerated types must be declared in the source file, with no duplicate constructors. This rule is specified in [figure A.5](#). It also gives the

$$\begin{array}{c}
\frac{}{\text{NoDupLocals } \Gamma ([x_i]^i = es)} \qquad \frac{}{\text{NoDupLocals } \Gamma (\text{last } x = e)} \\
\\
\frac{\text{NoDupLocals } \Gamma \text{ blks}}{\text{NoDupLocals } \Gamma (\text{reset } \text{blks every } e)} \\
\\
\frac{\text{NoDupMembers } locs \quad \forall x, x \in locs \implies x \notin \Gamma \quad \text{NoDupLocals } (\Gamma + locs) \text{ blks}}{\text{NoDupLocals } \Gamma (\text{var } locs \text{ let } \text{blks tel})} \\
\\
\frac{\text{NoDupMembers } \sigma \quad \text{NoDupLocals } \Gamma \text{ blks}}{\text{NoDupLocals } \Gamma (C \text{ do}_\sigma \text{ blks)} \qquad \frac{\forall i, \text{NoDupLocals } \Gamma \text{ br}_i}{\text{NoDupLocals } \Gamma (\text{switch } e [(br_i)]^i \text{ end})} \\
\\
\frac{\text{NoDupMembers } locs \quad \forall x, x \in locs \implies x \notin \Gamma \quad \text{NoDupMembers } \sigma \quad \text{NoDupLocals } (\Gamma + locs) \text{ blks}}{\text{NoDupLocals } \Gamma (\text{state } C \text{ var}_\sigma \text{ locs do } \text{blks until } \text{trunt unless } \text{truntl})} \\
\\
\frac{\forall i, \text{NoDupLocals } \Gamma \text{ autst}_i}{\text{NoDupLocals } \Gamma (\text{automaton initially } \text{autinits } [(autst_i)]^i \text{ end})} \\
\\
\frac{\text{NoDupMembers } (ins + outs) \quad \text{NoDupLocals } (ins + outs) \text{ blk}}{\text{NoDupLocals } (\text{node } f(ins) \text{ returns } (outs) \text{ blk})}
\end{array}$$

Figure A.3: Checking Non-Duplication 🐔 [Lustre/LSyntax.v:416](#)

typing rules for clock annotations: for a sampled clock ck **on** $C(x)$, x must be declared with an enumerated type of which C is a constructor.

The judgement $G, \Gamma \vdash_{\text{wt}} e : tys$ specifies that an expression e is well-typed with types tys . [Figure A.6](#) details the corresponding typing rules. Scalar constants are always well typed. Enumerated constructors are well typed if the constructor belongs to a declared enumerated type. The type of a variable or **last** variable is looked up in the environment. Unary and binary operators are annotated with their input and output types. All other operators preserve the types of their operands. In addition, the condition of **when**, **merge**, and **case** must be of an enumerated type matching the constructors used in the expression.

[Figure A.7](#) details the typing rules for blocks and nodes. An equation is well typed if the types of variables at left correspond to those of expressions at right. The condition of a **reset** block must be boolean; we write **bool** as a shortcut for the enumerated type with constructors **false** | **true**, which is implicitly defined in every program. The blocks under a local declaration are typed with an extended environment. The constructors of a **switch** must be a permutation of those of the enumerated type of the condition. All the transition conditions of an automaton must be boolean, and all the transition targets

$$\begin{array}{c}
\frac{\text{'\$'} \notin \text{str_to_pos } x}{\text{AtomOrGensym}_{\text{prefs}} x} \qquad \frac{\text{pref} \in \text{prefs}}{\text{AtomOrGensym}_{\text{prefs}} (\text{gensym } \text{pref } \text{hint } x)} \\
\\
\frac{}{\text{GoodLocals}_{\text{prefs}} (x_1, \dots, x_n) = \text{es}} \qquad \frac{}{\text{GoodLocals}_{\text{prefs}} \text{last } x = e} \\
\\
\frac{\text{GoodLocals}_{\text{prefs}} \text{blks}}{\text{GoodLocals}_{\text{prefs}} \text{reset } \text{blks } \text{every } e} \\
\\
\frac{\forall x, x \in \text{locs} \implies \text{AtomOrGensym}_{\text{prefs}} x \quad \text{GoodLocals}_{\text{prefs}} \text{blks}}{\text{GoodLocals}_{\text{prefs}} \text{var } \text{locs } \text{let } \text{blks } \text{tel}} \\
\\
\frac{\forall i, \text{GoodLocals}_{\text{prefs}} \text{blks}_i}{\text{GoodLocals}_{\text{prefs}} \text{switch } e [C_i \text{ do } \text{blks}_i]^i \text{ end}} \\
\\
\frac{\forall x, x \in \text{locs} \implies \text{AtomOrGensym}_{\text{prefs}} x \quad \text{GoodLocals}_{\text{prefs}} \text{blks}}{\text{GoodLocals}_{\text{prefs}} (\text{state } C \text{ var}_{\sigma} \text{locs } \text{do } \text{blks } \text{until } \text{trunt } \text{unless } \text{truntl})} \\
\\
\frac{\forall i, \text{GoodLocals}_{\text{prefs}} \text{autst}_i}{\text{GoodLocals}_{\text{prefs}} \text{automaton } \text{initially } \text{autinits } [\text{autst}_i]^i \text{ end}} \\
\\
\frac{\forall x, x \in (\text{ins} + \text{outs}) \implies \text{AtomOrGensym}_{\text{prefs}} x \quad \text{GoodLocals}_{\text{prefs}} \text{blk}}{\text{GoodLocals}_{\text{prefs}} (\text{node } f(\text{ins}) \text{returns } (\text{outs}) \text{blk})}
\end{array}$$

Figure A.4: Checking Non-Duplication 🐓 [Lustre/LSyntax.v:416](#)

$$\begin{array}{c}
\frac{\text{type } tx = [| C_i]^i \in G \quad \text{NoDup } [C_i]^i}{G \vdash_{\text{wt}} [| C_i]^i} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} ck \quad \Gamma(x) = [| C_i]^i \quad G \vdash_{\text{wt}} [| C_i]^i \quad C \in [C_i]^i}{G, \Gamma \vdash_{\text{wt}} ck \text{ on } C(x)} \\
\\
\frac{}{G, \Gamma \vdash_{\text{wt}} \bullet}
\end{array}$$

Figure A.5: Typing rules for clocks 🐓 [Lustre/LTyping.v:41](#)

$$\begin{array}{c}
\frac{}{G, \Gamma \vdash_{\text{wt}} c : [\text{type_const } c]} \quad \frac{G \vdash_{\text{wt}} [! C_i]^i \quad C \in [C_i]^i}{G, \Gamma \vdash_{\text{wt}} C : [[! C_i]^i]} \quad \frac{\Gamma(x) = ty}{G, \Gamma \vdash_{\text{wt}} x : [ty]} \\
\\
\frac{\Gamma(\text{last } x) = ty}{G, \Gamma \vdash_{\text{wt}} \text{last } x : [ty]} \quad \frac{G, \Gamma \vdash_{\text{wt}} e_1 : [ty_1] \quad \vdash \diamond_{ty_1} : ty}{G, \Gamma \vdash_{\text{wt}} \diamond e_1 : [ty]} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} e_1 : [ty_1] \quad G, \Gamma \vdash_{\text{wt}} e_2 : [ty_2] \quad \vdash \oplus_{ty_1 \times ty_2} : ty}{G, \Gamma \vdash_{\text{wt}} e_1 \oplus e_2 : [ty]} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} es_0 : [ty_j]^j \quad G, \Gamma \vdash_{\text{wt}} es_1 : [ty_j]^j}{G, \Gamma \vdash_{\text{wt}} es_0 \text{ fby } es_1 : [ty_j]^j} \quad \frac{G, \Gamma \vdash_{\text{wt}} es_0 : [ty_j]^j \quad G, \Gamma \vdash_{\text{wt}} es_1 : [ty_j]^j}{G, \Gamma \vdash_{\text{wt}} es_0 \rightarrow es_1 : [ty_j]^j} \\
\\
\frac{\Gamma(x) = [! C_i]^i \quad G \vdash_{\text{wt}} [! C_i]^i \quad C \in [C_i]^i \quad G, \Gamma \vdash_{\text{wt}} es : [ty_j]^j}{G, \Gamma \vdash_{\text{wt}} es \text{ when } C(x) : [ty_j]^j} \\
\\
\frac{\Gamma(x) = [! C_i]^i \quad G \vdash_{\text{wt}} [C_i]^i \quad \text{Permutation } [C_i]^i [C_i']^i \quad \forall i, G, \Gamma \vdash_{\text{wt}} es_i : [ty_j]^j}{G, \Gamma \vdash_{\text{wt}} \text{merge } x [C_i' \Rightarrow es_i]^i : [ty_j]^j} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} e : [[! C_i]^i] \quad G \vdash_{\text{wt}} [C_i]^i \quad \text{Permutation } [C_i]^i [C_i']^i \quad \forall i, G, \Gamma \vdash_{\text{wt}} es_i : [ty_j]^j}{G, \Gamma \vdash_{\text{wt}} \text{case } e \text{ of } [C_i' \Rightarrow es_i]^i : [ty_j]^j} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} es : [ty_i]^i \quad G(f) = \text{node } f([x_i : ty_i]^i) \text{ returns } ([y_j : ty_j']^j) \text{ blk}}{G, \Gamma \vdash_{\text{wt}} f(es) : [ty_j']^j}
\end{array}$$

Figure A.6: Typing rules for expressions 🐔 [Lustre/LTyping.v:63](#)

must be declared states of the automaton. Finally, a node is well typed if its body is well typed under the declared input and output variables.

A.3 Clock-Type System

We have already discussed the clock-type system of Vélus in [chapter 2](#). [Figures A.8](#) and [A.9](#) complete the clock-typing rules for expressions and blocks. In particular, they show the rules for the **fby** and initialization arrow, which both preserve the clock types of their sub-expressions. The **case** operators forces all the branches to be on the same clock type as the condition.

Blocks under a local declaration are typed in an extended environment. The rule also requires that the local clocks be well formed under the extended environment. The

$$\begin{array}{c}
\frac{\forall i, \Gamma(x_j) = ty_j \quad G, \Gamma \vdash_{\text{wt}} es : [ty_j]^j}{G, \Gamma \vdash_{\text{wt}} [x_j]^j = es} \qquad \frac{\Gamma(\text{last } x) = ty \quad G, \Gamma \vdash_{\text{wt}} e : [ty]}{G, \Gamma \vdash_{\text{wt}} \text{last } x = e} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} blks \quad G, \Gamma \vdash_{\text{wt}} e : [\text{bool}]}{G, \Gamma \vdash_{\text{wt}} \text{reset } blks \text{ every } e} \qquad \frac{G, (\Gamma + locs) \vdash_{\text{wt}} blks}{G, \Gamma \vdash_{\text{wt}} \text{var } locs \text{ let } blks \text{ tel}} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} e : [[| C_i]^i] \quad G \vdash_{\text{wt}} [| C_i]^i \quad \text{Permutation } [C_i]^i [C'_i]^i \quad \forall i, G, \Gamma \vdash_{\text{wt}} blks_i}{G, \Gamma \vdash_{\text{wt}} \text{switch } e [C'_i \text{ do } blks_i]^i \text{ end}} \\
\\
\frac{\text{NoDup } [C_i]^i \quad G, \Gamma, [C_i]^i \vdash_{\text{wt}} \text{autinits} \quad \forall i, G, \Gamma, [C_i]^i \vdash_{\text{wt}} \text{autscope}_i}{G, \Gamma \vdash_{\text{wt}} \text{automaton initially } \text{autinits } [\text{state } C_i \text{ autscope}_i]^i \text{ end}} \\
\\
\frac{C \in [C_i]^i \quad \text{NoDup } [C_i]^i \quad \forall i, G, \Gamma \vdash_{\text{wt}} blks_i \quad \forall i, G, \Gamma, [C_i]^i \vdash_{\text{wt}} \text{trans}_i}{G, \Gamma \vdash_{\text{wt}} \text{automaton initially } C [\text{state } C_i \text{ do } blks_i \text{ unless } \text{trans}_i]^i \text{ end}} \\
\\
\frac{G, (\Gamma + locs), \text{constrs} \vdash_{\text{wt}} \text{trans} \quad G, (\Gamma + locs) \vdash_{\text{wt}} blks}{G, \Gamma, \text{constrs} \vdash_{\text{wt}} \text{var } locs \text{ do } blks \text{ until } \text{trans}} \\
\\
\frac{C \in \text{constrs}}{G, \Gamma, \text{constrs} \vdash_{\text{wt}} \text{otherwise } C} \qquad \frac{G, \Gamma \vdash_{\text{wt}} e : [\text{bool}] \quad C \in \text{constrs}}{G, \Gamma, \text{constrs} \vdash_{\text{wt}} C \text{ if } e; \text{autinits}} \\
\\
\frac{}{G, \Gamma, \text{constrs} \vdash_{\text{wt}} \epsilon} \qquad \frac{G, \Gamma \vdash_{\text{wt}} e : [\text{bool}] \quad C \in \text{constrs}}{G, \Gamma, \text{constrs} \vdash_{\text{wt}} \text{if } e \text{ then } C \text{ trans}} \\
\\
\frac{G, \Gamma \vdash_{\text{wt}} e : [\text{bool}] \quad C \in \text{constrs}}{G, \Gamma, \text{constrs} \vdash_{\text{wt}} \text{if } e \text{ continue } C \text{ trans}} \\
\\
\frac{G, (\text{ins} + \text{outs}) \vdash_{\text{wt}} blk}{G \vdash_{\text{wt}} \text{node } f(\text{ins}) \text{ returns } (\text{outs}) \text{ blk}}
\end{array}$$


Figure A.7: Typing rules for blocks and nodes 🐔 [Lustre/LTyping.v:203](#)

$$\begin{array}{c}
\frac{}{G, \Gamma \vdash_{\text{wc}} c : [\bullet]} \quad \frac{\Gamma(x) = ck}{G, \Gamma \vdash_{\text{wc}} x : [ck]} \quad \frac{\Gamma(x) = ck}{G, \Gamma \vdash_{\text{wc}} x : [(x : ck)]} \quad \frac{G, \Gamma \vdash_{\text{wc}} e_1 : [ck]}{G, \Gamma \vdash_{\text{wc}} \diamond e_1 : [ck]} \\
\\
\frac{G, \Gamma \vdash_{\text{wc}} e_1 : [ck] \quad G, \Gamma \vdash_{\text{wc}} e_2 : [ck]}{G, \Gamma \vdash_{\text{wc}} e_1 \oplus e_2 : [ck]} \quad \frac{G, \Gamma \vdash_{\text{wc}} es_0 : [ck_j]^j \quad G, \Gamma \vdash_{\text{wc}} es_1 : [ck_j]^j}{G, \Gamma \vdash_{\text{wc}} es_0 \text{ fby } es_1 : [ck_j]^j} \\
\\
\frac{G, \Gamma \vdash_{\text{wc}} es_0 : [ck_j]^j \quad G, \Gamma \vdash_{\text{wc}} es_1 : [ck_j]^j}{G, \Gamma \vdash_{\text{wc}} es_0 \rightarrow es_1 : [ck_j]^j} \quad \frac{\Gamma(x) = ck \quad G, \Gamma \vdash_{\text{wc}} es : [ck]^j}{G, \Gamma \vdash_{\text{wc}} es \text{ when } C(x) : [ck \text{ on } C(x)]^j} \\
\\
\frac{\Gamma(x) = ck \quad \forall i, G, \Gamma \vdash_{\text{wc}} es_i : [ck \text{ on } C_i(x)]^j}{G, \Gamma \vdash_{\text{wc}} \text{merge } x [C_i \Rightarrow es_i]^i : [ck]^j} \\
\\
\frac{G, \Gamma \vdash_{\text{wc}} e : [ck] \quad \forall i, G, \Gamma \vdash_{\text{wc}} es_i : [ck]^j}{G, \Gamma \vdash_{\text{wc}} \text{case } e \text{ of } [C_i \Rightarrow es_i]^i : [ck]^j} \\
\\
\frac{G, \Gamma \vdash_{\text{wc}} es : [nck_i]^i \quad G(f) = \text{node } f([x_i \text{ on } ick_i]^i) \text{ returns } ([y_j \text{ on } ock_j]^j) \text{ blk} \\ \forall i, \text{WellInstantiated}_{\sigma}^{bck}(x_i : ick_i) nck_i \quad \forall j, \text{WellInstantiated}_{\sigma}^{bck}(y_j : ock_j) (_ : ck'_j)}{G, \Gamma \vdash_{\text{wc}} f(es) : [ck'_j]^j} \\
\\
\frac{\Gamma(\text{last } x) = ck}{G, \Gamma \vdash_{\text{wc}} \text{last } x : [ck]}
\end{array}$$

Figure A.8: Clock-typing rules for expressions 🐔 [Lustre/LClocking.v:55](#)

clock-typing rules for state machines follow the same ideas as the ones for `switch` blocks: the sub-blocks are clock typed in a sampled environment. Additionally, the environment of state machines with weak transitions is extended with the local declarations that may be used in transitions.

$$\begin{array}{c}
\frac{\forall j, \Gamma(x_j) = ck_j \quad G, \Gamma \vdash_{wc} es : [ck_j]^j}{G, \Gamma \vdash_{wc} [x_j]^j = es} \\
\\
\frac{\begin{array}{c} \forall j, \Gamma(y'_j) = ck'_j \\ G, \Gamma \vdash_{wc} es : [nck_i]^i \quad G(f) = \mathbf{node} f([x_i \mathbf{on} ick_i]^i) \mathbf{returns} ([y_j \mathbf{on} ock_j]^j) \mathit{blk} \\ \forall i, \text{WellInstantiated}_{\sigma}^{bck}(x_i : ick_i) nck_i \quad \forall j, \text{WellInstantiated}_{\sigma}^{bck}(y_j : ock_j) (y'_j : ck'_j) \end{array}}{G, \Gamma \vdash_{wc} [y'_j]^j = f(es)} \\
\\
\frac{\Gamma(\mathbf{last} x) = ck \quad G, \Gamma \vdash_{wc} e : [ck]}{G, \Gamma \vdash_{wc} \mathbf{last} x = e} \quad \frac{G, \Gamma \vdash_{wc} e : [ck] \quad G, \Gamma \vdash_{wc} \mathit{blks}}{G, \Gamma \vdash_{wc} \mathbf{reset} \mathit{blks} \mathbf{every} e} \\
\\
\frac{\vdash_{wc} (\Gamma + \mathit{locs}) \quad G, (\Gamma + \mathit{locs}) \vdash_{wc} \mathit{blks}}{G, \Gamma \vdash_{wc} \mathbf{var} \mathit{locs} \mathbf{let} \mathit{blks} \mathbf{tel}} \\
\\
\frac{G, \Gamma \vdash_{wc} e : [ck] \quad \forall x ck', \Gamma'(x) = ck' \implies \Gamma(x) = ck \wedge ck' = \bullet \quad \forall i, G, \Gamma' \vdash_{wc} \mathit{blks}_i}{G, \Gamma \vdash_{wc} \mathbf{switch} e [C_i \mathbf{do} \mathit{blks}_i]^i \mathbf{end}} \\
\\
\frac{\begin{array}{c} \forall x ck', \Gamma'(x) = ck' \implies \Gamma(x) = ck \wedge ck' = \bullet \\ G, \Gamma' \vdash_{wc} \mathit{autinits} \quad \forall i, G, \Gamma' \vdash_{wc} \mathit{autscope}_i \end{array}}{G, \Gamma \vdash_{wc} \mathbf{automaton} \mathbf{initially} \mathit{autinits}^{ck} [\mathbf{state} C_i \mathit{autscope}_i]^i \mathbf{end}} \\
\\
\frac{\begin{array}{c} \forall x ck', \Gamma'(x) = ck' \implies \Gamma(x) = ck \wedge ck' = \bullet \\ \forall i, G, \Gamma' \vdash_{wc} \mathit{blks}_i \quad \forall i, G, \Gamma' \vdash_{wc} \mathit{trans}_i \end{array}}{G, \Gamma \vdash_{wc} \mathbf{automaton} \mathbf{initially} C^{ck} [\mathbf{state} C_i \mathbf{do} \mathit{blks}_i \mathbf{unless} \mathit{trans}_i]^i \mathbf{end}} \\
\\
\frac{G, (\Gamma + \mathit{locs}) \vdash_{wc} \mathit{trans} \quad G, (\Gamma + \mathit{locs}) \vdash_{wc} \mathit{blks}}{G, \Gamma \vdash_{wc} \mathbf{var} \mathit{locs} \mathbf{do} \mathit{blks} \mathbf{until} \mathit{trans}} \\
\\
\frac{}{G, \Gamma \vdash_{wc} \mathbf{otherwise} C} \quad \frac{G, \Gamma \vdash_{wc} e : [\bullet]}{G, \Gamma \vdash_{wc} C \mathbf{if} e; \mathit{autinits}} \\
\\
\frac{}{G, \Gamma \vdash_{wc} \epsilon} \quad \frac{G, \Gamma \vdash_{wc} e : [\bullet]}{G, \Gamma \vdash_{wc} \mathbf{if} e \mathbf{then} C \mathit{trans}} \quad \frac{G, \Gamma \vdash_{wc} e : [\bullet]}{G, \Gamma \vdash_{wc} \mathbf{if} e \mathbf{continue} C \mathit{trans}} \\
\\
\frac{\vdash_{wc} \mathit{ins} \quad \vdash_{wc} (\mathit{ins} + \mathit{outs}) \quad G, (\mathit{ins} + \mathit{outs}) \vdash_{wc} \mathit{blk}}{G \vdash_{wc} \mathbf{node} f(\mathit{ins}) \mathbf{returns} (\mathit{outs}) \mathit{blk}}
\end{array}$$

Figure A.9: Clock-typing rules for blocks  [Lustre/LClocking.v:167](#)

Full Compilation of the Introductory Example

In this appendix, we illustrate the compilation scheme used in the compiler by applying it to the `drive_sequence` example presented in the introduction. We show the intermediate programs after each separate pass. The source program is reproduced in the listing below.

```
node drive_sequence(step : bool)
returns (mA, mB : bool)
let
  last mA = true;
  last mB = true;
  switch step
  | true do (mA, mB) = (not (last mB), last mA)
  | false do
  end;
tel
```

Listing B.1: Source node

The completion pass described in [section 4.5](#) completes the definitions of `mA` and `mB` by adding `mA = last mA` and `mB = last mB` equations in the second branch. This transformation makes explicit the implicit equations that already exist in the semantics.

```

node drive_sequence (step : bool) returns (mA, mB : bool)
let
  last mA = true;
  last mB = true;
  switch step
  | true do
    (mA, mB) = (not (last mB), last mA)
  | false do
    mA = last mA;
    mB = last mB;
  end
tel

```

Listing B.2: Completed

The pass that compiles state machines does not do anything to this node, since it does not contain any. The next pass, described in [section 4.8](#) compiles away the `switch` block. Each variable used in the `switch` is sampled by `when`, and the variables defined by the `switch` are now defined by a `merge`. Some of the sampling equations are actually useless; they are greyed-out in the listing and will be optimized away by a later pass.

```

node drive_sequence (step : bool) returns (mA, mB : bool)
let
  last mA = true;
  last mB = true;
  var
    swi$step$1, swi$mA$2, swi$mB$3, swi$mA$4, swi$mB$5 : bool when true(step);
    swi$step$6, swi$mA$7, swi$mB$8, swi$mA$9, swi$mB$10 : bool when false(step);
  let
    mA = merge step (true => swi$mA$2) (false => swi$mA$7);
    mB = merge step (true => swi$mB$3) (false => swi$mB$8);
    (swi$mA$2, swi$mB$3) = (not swi$mB$5, swi$mA$4);
    swi$mA$4 = last mA when true(step);
    swi$mB$5 = last mB when true(step);
    swi$mA$7 = swi$mA$9;
    swi$mB$8 = swi$mB$10;
    swi$mA$9 = last mA when false(step);
    swi$mB$10 = last mB when false(step);
    swi$step$1 = step when true(step);
    swi$step$6 = step when false(step);
  tel
tel

```

Listing B.3: Removed `switch`

The nested local scope that was introduced by the previous pass is “flattened” into a top-level local scope by the pass described in [section 4.9](#). Since the node does not contain

any duplicate variable names, no renaming is necessary.

```
node drive_sequence (step : bool) returns (mA, mB : bool)
var
  swi$step$1, swi$mA$2, swi$mB$3, swi$mA$4, swi$mB$5 : bool when true(step);
  swi$step$6, swi$mA$7, swi$mB$8, swi$mA$9, swi$mB$10 : bool when false(step);
let
  last mA = true;
  last mB = true;
  mA = merge step (true => swi$mA$2) (false => swi$mA$7);
  mB = merge step (true => swi$mB$3) (false => swi$mB$8);
  (swi$mA$2, swi$mB$3) = (not swi$mB$5, swi$mA$4);
  swi$mA$4 = last mA when true(step);
  swi$mB$5 = last mB when true(step);
  swi$mA$7 = swi$mA$9;
  swi$mB$8 = swi$mB$10;
  swi$mA$9 = last mA when false(step);
  swi$mB$10 = last mB when false(step);
  swi$step$1 = step when true(step);
  swi$step$6 = step when false(step);
tel
```

Listing B.4: Flattened local scope

The unnesting pass described in [section 4.10](#) breaks the fifth equation in two simpler ones.

```
node drive_sequence (step : bool) returns (mA, mB : bool)
var
  swi$step$1, swi$mA$2, swi$mB$3, swi$mA$4, swi$mB$5 : bool when true(step);
  swi$step$6, swi$mA$7, swi$mB$8, swi$mA$9, swi$mB$10 : bool when false(step);
let
  last mA = true;
  last mB = true;
  mA = merge step (true => swi$mA$2) (false => swi$mA$7);
  mB = merge step (true => swi$mB$3) (false => swi$mB$8);
  swi$mA$2 = not swi$mB$5;
  swi$mB$3 = swi$mA$4;
  swi$mA$4 = last mA when true(step);
  swi$mB$5 = last mB when true(step);
  swi$mA$7 = swi$mA$9;
  swi$mB$8 = swi$mB$10;
  swi$mA$9 = last mA when false(step);
  swi$mB$10 = last mB when false(step);
  swi$step$1 = step when true(step);
  swi$step$6 = step when false(step);
tel
```

Listing B.5: Unnested

The outputs of the nodes are used with `last`, which must be simplified by the pass described in [section 4.11](#). This pass introduces new `last` variables, `lastmA1` and `lastmB2`, which replace all occurrences of `mA` and `mB`.

```

node drive_sequence (step : bool) returns (mA, mB : bool)
var
  swi$step$1, swi$mA$2, swi$mB$3, swi$mA$4, swi$mB$5 : bool when true(step);
  swi$step$6, swi$mA$7, swi$mB$8, swi$mA$9, swi$mB$10 : bool when false(step);
  last$mA$1 : bool; last$mB$2 : bool;
let
  last last$mA$1 = true;
  last last$mB$2 = true;
  last$mA$1 = merge step (true => swi$mA$2) (false => swi$mA$7);
  mA = last$mA$1;
  last$mB$2 = merge step (true => swi$mB$3) (false => swi$mB$8);
  mB = last$mB$2;
  swi$mA$2 = not swi$mB$5;
  swi$mB$3 = swi$mA$4;
  swi$mA$4 = last last$mA$1 when true(step);
  swi$mB$5 = last last$mB$2 when true(step);
  swi$mA$7 = swi$mA$9;
  swi$mB$8 = swi$mB$10;
  swi$mA$9 = last last$mA$1 when false(step);
  swi$mB$10 = last last$mB$2 when false(step);
  swi$step$1 = step when true(step);
  swi$step$6 = step when false(step);
tel

```

Listing B.6: Normalized `lasts`

There are no `fbv` equations to be simplified in this program. The translation of this program to NLustre only reorders the branches of the `merge`.

The next pass, [section 5.2.3.1](#), replaces some variables by the simple expressions that define them. In particular, it reconstructs the syntactic relation between `lastmA1` and `last lastmA1` which was lost during the compilation of `switch`.

```
node drive_sequence (step : bool) returns (mA, mB : bool)
var
  swi$step$1, swi$mA$2, swi$mB$3, swi$mA$4, swi$mB$5 : bool when true(step);
  swi$step$6, swi$mA$7, swi$mB$8, swi$mA$9, swi$mB$10 : bool when false(step);
  last$mA$1 : bool; last$mB$2 : bool;
let
  last last$mB$2 = true;
  last last$mA$1 = true;
  last$mA$1 =
    merge step
      (false => (last last$mA$1 when false(step)))
      (true => (not (last last$mB$2 when true(step))));
  mA = last$mA$1;
  last$mB$2 =
    merge step
      (false => (last last$mB$2 when false(step)))
      (true => (last last$mA$1 when true(step)));
  mB = last$mB$2;
  swi$mA$2 = not (last last$mB$2 when true(step));
  swi$mB$3 = last last$mA$1 when true(step);
  swi$mA$4 = last last$mA$1 when true(step);
  swi$mB$5 = last last$mB$2 when true(step);
  swi$mA$7 = last last$mA$1 when false(step);
  swi$mB$8 = last last$mB$2 when false(step);
  swi$mA$9 = last last$mA$1 when false(step);
  swi$mB$10 = last last$mB$2 when false(step);
  swi$step$1 = step when true(step);
  swi$step$6 = step when false(step);
tel
```

Listing B.7: Inlined simple expressions

Then, the dead equations are eliminated by the pass described in [section 5.2.3.2](#). In particular, it removes the equations introduced during the compilation of `switch`, as well as those “inlined” by the previous pass.

```

node drive_sequence (step : bool) returns (mA, mB : bool)
var last$mA$1 : bool; last$mB$2 : bool;
let
  last last$mB$2 = true;
  last last$mA$1 = true;
  last$mA$1 =
    merge step
      (false => (last last$mA$1 when false(step)))
      (true => (not (last last$mB$2 when true(step))));
  mA = last$mA$1;
  last$mB$2 =
    merge step
      (false => (last last$mB$2 when false(step)))
      (true => (last last$mA$1 when true(step)));
  mB = last$mB$2;
tel

```

Listing B.8: Removed dead equations

The NLustre node is then translated into an Stc system following the scheme presented in [section 5.3.4](#).

```

system drive_sequence {
  init last$mA$1 = true, last$mB$2 = true;
  transition(step : bool) returns (mA, mB : bool)
  {
    update last$mA$1 =
      merge step
        (false => (last last$mA$1 when false(step)))
        (true => (not (last last$mB$2 when true(step))))
    mA = last$mA$1
    update last$mB$2 =
      merge step
        (false => (last last$mB$2 when false(step)))
        (true => (last last$mA$1 when true(step)))
    mB = last$mB$2
  }
}

```

Listing B.9: Translated to Stc

The node contains an update cycle between state variables `lastmA1` and `lastmB2`. It is cut by adding a copy of `last lastmB2`, using the algorithm of [section 5.4.2.2](#).

```

system drive_sequence {
  init last$mB$2 = true, last$mA$1 = true;
  transition(step : bool) returns (mA : bool; mB : bool)
  var stc$last$mB$2$1 : bool;
  {
    update last$mA$1 =
      merge step
        (false => (last last$mA$1 when false(step)))
        (true => (not (stc$last$mB$2$1 when true(step))))
    stc$last$mB$2$1 = last last$mB$2
    mA = last$mA$1
    update last$mB$2 =
      merge step
        (false => (last last$mB$2 when false(step)))
        (true => (last last$mA$1 when true(step)))
    mB = last$mB$2
  }
}

```

Listing B.10: Cut update cycles

Then, the node is scheduled by the algorithm described in [section 5.4.2](#).

```

system drive_sequence {
  init last$mB$2 = true, last$mA$1 = true;
  transition(step : bool) returns (mA, mB : bool)
  var stc$last$mB$2$1 : bool;
  {
    stc$last$mB$2$1 = last last$mB$2
    update last$mB$2 =
      merge step
        (false => (last last$mB$2 when false(step)))
        (true => (last last$mA$1 when true(step)))
    update last$mA$1 =
      merge step
        (false => (last last$mA$1 when false(step)))
        (true => (not (stc$last$mB$2$1 when true(step))))
    mA = last$mA$1
    mB = last$mB$2
  }
}

```

Listing B.11: Scheduled

The node is then translated to Obc, following the scheme described in [section 5.4](#). In the body of the dissertation, we omitted the default branches of `switches`. The actual compiler places the statement that would be under the last branch of the `switch` in the default branch. With this scheme, the generated C `switch` does one less test.

```
class drive_sequence {
  state last$mB$2 : bool;
  state last$mA$1 : bool;

  method step(step : bool) returns (mA, mB : bool)
  var stc$last$mB$2$1 : bool
  {
    stc$last$mB$2$1 := state(last$mB$2);
    switch step {
    | false => state(last$mB$2) := state(last$mB$2)
    | true => _
    | _ => state(last$mB$2) := state(last$mA$1)
    };
    switch step {
    | false => state(last$mA$1) := state(last$mA$1)
    | true => _
    | _ => state(last$mA$1) := not stc$last$mB$2$1
    };
    mA := state(last$mA$1);
    mB := state(last$mB$2);
    skip
  }

  method reset() {
    state(last$mB$2) := true;
    state(last$mA$1) := true;
  }
}
```

Listing B.12: Translated to Obc

The two `switch` are fused into one by the pass described in [section 5.4.4](#).

```
class drive_sequence {
  state last$mB$2 : bool;
  state last$mA$1 : bool;

  method step(step : bool) returns (mA, mB : bool)
  var stc$last$mB$2$1 : bool {
    stc$last$mB$2$1 := state(last$mB$2);
    switch step {
    | false => state(last$mB$2) := state(last$mB$2);
              state(last$mA$1) := state(last$mA$1)
    | true  => _
    | _    => state(last$mB$2) := state(last$mA$1);
              state(last$mA$1) := not stc$last$mB$2$1
    };
    mA := state(last$mA$1);
    mB := state(last$mB$2)
  }

  method reset() {
    state(last$mB$2) := true;
    state(last$mA$1) := true
  }
}
```

Listing B.13: Fused

The final optimisation removes two dead update statements in the `false` branch of the `switch`. The specification of this pass is trivial, as explained in [section 5.1.4](#).

```
class drive_sequence {
  state last$mB$2 : bool;
  state last$mA$1 : bool;

  method step(step : bool) returns (mA, mB : bool)
  var stc$last$mB$2$1 : bool {
    stc$last$mB$2$1 := state(last$mB$2);
    switch step {
    | false => skip; skip
    | true => _
    | _ => state(last$mB$2) := state(last$mA$1);
          state(last$mA$1) := not stc$last$mB$2$1
    };
    mA := state(last$mA$1);
    mB := state(last$mB$2)
  }

  method reset() {
    state(last$mB$2) := true;
    state(last$mA$1) := true
  }
}
```

Listing B.14: Removed dead updates

Finally, Vélus generates Clight code from the Obc program. There are a few complex details in this generation, in particular relating to the memory model of Clight. These have been presented in [Bru20, Chapter 5].

```
void fun$step$drive_sequence(struct drive_sequence *obc2c$self,
                             struct fun$step$drive_sequence *obc2c$out,
                             unsigned char step) {
    register unsigned char stc$last$mB$2$1;
    stc$last$mB$2$1 = (*obc2c$self).last$mB$2;
    switch (step) {
        case 0:
            /*skip*/;
            break;
        default:
            (*obc2c$self).last$mB$2 = (*obc2c$self).last$mA$1;
            (*obc2c$self).last$mA$1 = !stc$last$mB$2$1;
    }
    (*obc2c$out).mA = (*obc2c$self).last$mA$1;
    (*obc2c$out).mB = (*obc2c$self).last$mB$2;
    return;
}

void fun$reset$drive_sequence(struct drive_sequence *obc2c$self) {
    (*obc2c$self).last$mB$2 = 1;
    (*obc2c$self).last$mA$1 = 1;
    return;
}
```

Listing B.15: Generated C code

A Semantic Preservation Proof

This appendix gives more detail about the semantic preservation proof for the compilation of state machines. It focuses on one specific case: the compilation of state machines with strong transitions, which was presented in [section 4.7](#). Through this example, we try to give a taste of the mechanized Coq proofs of Vélus. We start by presenting the complete inductive invariant for blocks, which was summarized in [invariant 2](#) ([page 99](#)).

Invariant 11 Compilation of State Machines 🐓 [Lustre/CompAuto/CACorrectness.v:558](#)

if	$G, \Gamma_{ty} \vdash_{wt} blk$	(Hwt)
and	$G, \Gamma_{ck} \vdash_{wc} blk$	(Hwc)
and	$\text{dom}(\Gamma_{ck}) \subseteq \text{dom}(\Gamma_{ty})$	$(Hincl)$
and	$\text{NoDupLocals } \Gamma_{ty} \ blk$	(Hnd)
and	$\forall x, x \in \text{dom}(\Gamma_{ty}) \implies \text{AtomOrGensym}_{\emptyset} x$	(Hat)
and	$\text{GoodLocals}_{\emptyset} blk$	$(Hgood)$
and	$G, H, bs \vdash_{ck} blk$	$(Hsem)$
and	$\text{dom}(H) \subseteq \text{dom}(\Gamma_{ty})$	$(Hdom)$
and	$\Gamma_{ck}, bs \vdash_{ck} H$	(Hsc)
then	$G, H, bs \vdash_{ck} [blk]$	

The first two hypotheses indicate that the block must be well typed and well clock-typed. The domain of typing environment Γ_{ty} and clock-typing environment Γ_{ck} may be different. Indeed, recall that the clock-typing rules for [switch](#) and state machines restrict the domain of the clock-typing environment; this is not the case in the typing rules. All the same, the domain of Γ_{ck} should be included in that of Γ_{ty} .

The next three premises concern the uniqueness of existing and generated identifiers. The [Hnd](#) premise indicates the absence of shadowing and duplication in local variables, according to the rules of [appendix A.1.2](#). The [Hat](#) and [Hgood](#) premises indicate that all global and local variables use well-formed identifiers, according to the rules of [appendix A.1.3](#): they are all either atomic, or formed using `gensym` with a prefix in the set

prefs. Here, since the compilation of state machines is the first pass that introduces fresh identifiers, the set of source prefixes is the empty set \emptyset .

The **Hsem** premise indicates that *blk* has a semantics under history *H*. The next two premises add to the specification of *H*. First, its domain should be included in the domain of Γ_{ty} . This facilitates the extension of *H* with fresh variables. Second, *H* must be well-clocked according to the rule of [figure 4.5a](#): the streams associated with each variable in *H* must adhere to the corresponding clock annotations. This facilitates the proof of obligations related to the clocked semantic model.

We now describe the proof for the case a state machine with strong transitions, that is, when *blk* = **automaton initially** *C* [**state** C_i^{ck} **do** *blks_i* **unless** *tr_i*]^{*i*} **end**. The proof is shown in [listing C.1](#), as a simplified Coq proof script using the notations of this dissertation. It starts by inverting the clock-typing and semantics hypotheses, which exhibits some existential variables that are useful further in the proof. Inverting the **Hsem** hypothesis for this block gives the following four premises.

$$\begin{array}{ll}
 H, bs \vdash ck \Downarrow bs' & (\text{Hck}) \\
 \text{fby} (\text{const } bs' (C, F)) \text{ } sts_1 \equiv sts & (\text{Hfby}) \\
 \forall i, \forall k, G, (\text{select}_0^{C_i, k} sts (H, bs)), C_i \vdash tr_i \Downarrow (\text{select}_0^{C_i, k} sts sts_1) & (\text{Htr}) \\
 \forall i, \forall k, G, (\text{select}_0^{C_i, k} sts_1 (H, bs)), C_i \vdash_{ck} blks_i & (\text{Hblks})
 \end{array}$$

Inverting the **Hwc** hypothesis gives a new clock-typing environment Γ'_{ck} and the following three premises.

$$\begin{array}{ll}
 \forall x ck', \Gamma'_{ck}(x) = ck' \implies \Gamma_{ck}(x) = ck \wedge ck' = \bullet & (\text{Henv}) \\
 \forall i, G, \Gamma'_{ck} \vdash_{wc} blks_i & (\text{Hwcblks}) \\
 \forall i, G, \Gamma'_{ck} \vdash_{wc} tr_i & (\text{Hwctr})
 \end{array}$$

The proof begins ([lines 4-19](#)) with some forward reasoning using the **assert** tactic to establish auxiliary facts needed in several later sub-goals. **Hnin1** establishes that the new identifiers do not appear in Γ_{ty} . This is true because all identifiers in Γ_{ty} are atoms and the fresh identifiers have been generated by **gensym**. **Hnin2** states that these identifiers cannot appear in the domain of *H* either, since it is a subset of the domain of Γ_{ty} . By reasoning on the **Fresh** monad, we can also prove that these identifiers are all distinct. **Hincl'** proves that the domain of the inner clock-typing environment Γ'_{ck} is included in that of Γ_{ck} ; it follows directly from **Henv**.

The next two **asserts** exhibit properties of the semantic models. The state streams *sts* and *sts₁* are both aligned with the base clock of the state machine *bs'*, because the **fby** operator used in the **Hfby** hypothesis forces alignment of its arguments.

```

1 inversion Hsem as [| | | | | |Hck Hfby Htr Hblks].
2 inversion Hwc as [| | | | | |Γck' Henv Hwcbllks Hwctr].
3
4 assert (Forall (fun id => id ∉ dom(Γty)) [xst; xres; xst1; xres1]) as Hnin1.
5 { ... apply Hat. ... }
6 assert (Forall (fun id => id ∉ dom(H)) [xst; xres; xst1; xres1]) as Hnin2.
7 { ... apply Hdom. ... }
8 assert (NoDup [xst; xres; xst1; xres1]) as Hnd.
9 { ... }
10 assert (dom(Γck') ⊆ dom(Γck)) as Hincl'.
11 { ... apply Henv. ... }
12
13 assert (clock-of sts ≡ bs') as Hac.
14 { ... apply ac_fby1 in Hfby. ... }
15 assert (clock-of sts1 ≡ bs') as Hac1.
16 { ... apply ac_fby2 in Hfby. ... }
17
18 remember {xst ↦ π1(sts); xres ↦ π2(sts); xst1 ↦ π1(sts1); xres1 ↦ π2(sts1)} as H'.
19 assert (H ⊆ H + H') as Href'.
20 { ... apply Hnin2. }
21
22 eapply Sscope with (H' := H'). (* figure 4.5c *)
23 - (* ∀x, x ∈ dom(H') ⇔ x ∈ {xst, xres, xst1, xres1} *)
24   split; intros * Hin; ...
25 - (* {xst : ck, xres : ck, xst1 : ck, xres1 : ck}, bs ⊢ck (H + H') *)
26   constructor. (* figure 4.5a *)
27   intros * Hxck. (* {xst : ck, xres : ck, xst1 : ck, xres1 : ck}(x) = ck' *)
28   assert (ck' = ck); [|subst]. { ... }
29   destruct Hxck as [Heq1|Heq1|Heq1|Heq1].
30   + (* (H + H'), bs ⊢ck xstck ↓ π1(sts) *)
31     constructor. (* figure 3.12c *)
32     * (* (H + H')(xst) ≡ π1(sts) *) ...
33     * (* (H + H'), bs ⊢ck ck ↓ clock-of (π1(sts)) *)
34     rewrite clockof_proj1, Hac. apply Hck.
35     ... (* 3 more cases *)
36 - (* G, (H + H'), bs ⊢ck eqfby; sw1; sw2 *)
37   repeat constructor.
38   +{ (* G, (H + H'), bs ⊢ck (xst, xres) = (Cck, falseck) fby (xst1, xres1) *)
39     repeat econstructor.
40     - (* (H + H')(xst) ≡ π1(sts) *) ...
41     - (* (H + H')(xres) ≡ π2(sts) *) ...
42     - (* G, (H + H'), bs ⊢ck Cck ↓ [π1(const bs' (C, F))] *)
43     eapply add_whens_sem. ... apply Hck.
44     - (* G, (H + H'), bs ⊢ck falseck ↓ [π2(const bs' (C, F))] *)
45     eapply add_whens_sem. ... apply Hck.
46     - (* (H + H')(xst1) ≡ π1(sts1) *) ...
47     - (* (H + H')(xres1) ≡ π2(sts1) *) ...
48     - (* fby (π1(const bs' (C, F))) (π1(sts1)) ≡ π1(sts) *)
49     eapply fby_proj1, Hfby.
50     - (* fby (π2(const bs' (C, F))) (π2(sts1)) ≡ π2(sts) *)
51     eapply fby_proj2, Hfby.
52   }
53 +{ (* G, (H + H'), bs ⊢ck switch xst [Ci do reset (xst1, xres1) = [tri]Ci every xresi end *)

```

```

54   econstructor; intros.
55   - (* (H + H')(xst) ≡ π1(sts) *) ...
56   - (* G, whenCi (π1(sts)) (H + H', bs) ⊢ck reset (xst1, xres1) = [tri]Ci every xres *)
57   econstructor; intros.
58   + (* whenCi (π1(sts)) (H + H')(xres) ≡ whenCi (π1(sts)) (π2(sts)) *)
59     rewrite when_hist. (* figure 2.14c *)
60     ...
61   + (* G, mask0k (whenCi (π1(sts)) (π2(sts))) (whenCi (π1(sts)) (H + H', bs)) ⊢ck ... *)
62     rewrite <- select_mask_when. (* lemma 2 *)
63     (* G, select0Ci,k sts (H + H', bs) ⊢ck (xst1, xres1) = [tri]Ci *)
64     repeat econstructor.
65     * (* select0Ci,k sts (H + H')(xst1) ≡ select0Ci,k sts (π1(sts1)) *)
66       rewrite select_hist. ...
67     * (* select0Ci,k sts (H + H')(xres1) ≡ select0Ci,k sts (π2(sts1)) *)
68       rewrite select_hist. ...
69     * (* G, select0Ci,k sts (H + H', bs) ⊢ck [tri]Ci ↓ [π1(sts1); π2(sts1)] *)
70     eapply trans_exp_sem.
71     ...
72     apply Htr.
73   }
74 +{ (* G, (H + H'), bs ⊢ck switch xst1 [Ci do reset blks'i every xres1]i end *)
75   econstructor; intros.
76   - (* (H + H')(xst1) ≡ π1(sts1) *) ...
77   - (* G, whenCi (π1(sts1)) (H + H', bs) ⊢ck reset blks'i every xres1 *)
78   econstructor; intros.
79   + (* whenCi (π1(sts1)) (H + H')(xres1) ≡ whenCi (π1(sts1)) (π2(sts1)) *)
80     rewrite when_hist. ...
81   + rewrite <- select_mask_when.
82     (* G, select0Ci,k sts (H + H', bs) ⊢ck (xst1, xres1) = [blks]i *)
83     apply sem_block_refines. ...
84     (* G, select0Ci,k sts H, bs ⊢ck (xst1, xres1) = [blks]i *)
85     apply Hind.
86     * (* G, Γty ⊢w blksi *)
87       inversion Hwt. auto.
88     * (* G, Γ'ck ⊢w blksi *)
89       apply Hwcblks.
90     * (* dom(Γ'ck) ⊆ dom(Γty) *)
91       intros. apply Hincl, Hincl'.
92     * (* NoDupLocals Γty blksi *)
93       inversion Hnd. auto.
94     * (* ∀x, x ∈ dom(Γty) ⇒ AtomOrGensym0 x *)
95       apply Hat.
96     * (* GoodLocals0 blksi *)
97       inversion Hgood. auto.
98     * (* G, select0Ci,k sts H, bs ⊢ck blksi *)
99       apply Hblk.
100    * (* dom(select0Ci,k sts H) ⊆ dom(Γty) *)
101      rewrite select_dom. apply Hdom.
102    * (* Γ'ck, select0Ci,k sts bs ⊢ck select0Ci,k sts H *)
103      apply select_sc_vars, Hsc.
104  }

```

Listing C.1: Proof of semantic correctness for compilation of state machines

Recall the compilation scheme presented in [figure 4.6](#). Compiling a state machine with strong transitions produces a scope `var $x_{st}, x_{res}, x_{st1}, x_{res1} : ck$ let $eqfby; sw_1; sw_2$ tel` where

- $eqfby$ is `(x_{st}, x_{res}) = ($C^{ck}, false^{ck}$) fby (x_{st1}, x_{res1})`
- sw_1 is `switch x_{st} [C_i do reset (x_{st1}, x_{res1}) = $[tr_i]_{C_i}$ every x_{res}] i end`
- sw_2 is `switch x_{st1} [C_i do reset $blks'_i$ every x_{res1}] i end`

To give a semantics to this scope, the proof must exhibit a local history H' such that (i) H' has domain $\{x_{st}, x_{res}, x_{st1}, x_{res1}\}$, (ii) $H + H'$ is well-clocked, and (iii) history $H + H'$ gives a semantics to the new blocks.

We take $H' = \{x_{st} \mapsto \pi_1(sts); x_{res} \mapsto \pi_2(sts); x_{st1} \mapsto \pi_1(sts_1); x_{res1} \mapsto \pi_2(sts_1)\}$, where π_1 and π_2 are the projections from the state-and-reset stream to enumerated value stream and boolean stream. We then show that $H + H'$ refines H . This is not completely trivial, because history concatenation $+$ gives priority to the right operand, as defined in [figure 2.19b](#). If H and H' were in conflict, then H' would overwrite some values of H , and the refinement would not hold. Using `Hnin2`, we prove that such a conflict never occurs.

The domain of H' is trivially correct ([lines 23-24](#)). Proving that $H + H'$ is well-clocked under the local environment ([lines 25-35](#)) requires proving that, for each variable x associated with a clock ck' in the local environment, then the clock of the stream associated with x is the interpretation of ck' . We reason by case analysis on the set of new identifiers. Since all new identifiers are declared with the clock type of the state machine ck , we can prove that $ck' = ck$. By `Hck`, we know that ck evaluates to bs' . For instance, if x is x_{st} , then its stream is $\pi_1(sts)$, which has the same clock as sts . By `Hac`, this is indeed bs' . The three other cases are similar.

The remaining obligations ([lines 36-104](#)) concern the semantics of the compiled sub-blocks. The proof that $eqfby$ has a semantics under $H + H'$ is relatively simple. It exploits the definition of H' to give a semantics to the variables x_{st}, x_{res} , etc. The semantics of sampled constants is given using an auxiliary lemma `add_whens_sem`. Finally, the `fby` semantics follow from the `Hfby` hypothesis.

Giving a semantics for the sw_1 block ([lines 53-73](#)) requires a bit more work. First, the stream associated to the `switch` condition is the one associated to the x_{st} variable, that is $\pi_1(sts)$. Then, the semantics of each branch must be given under a sampled history. In particular, this requires giving a semantics to the `reset` condition by finding the value of the x_{res} variable in the sampled history. By definition, this is the sampling of the $\pi_2(sts)$ stream, which is defined since $\pi_2(sts)$ has the same clock as $\pi_1(sts)$. Under the `reset` block, the history and base clock are sampled by `when` and `mask`. Using [lemma 2 \(page 47\)](#) transforms this sampling into sampling by `select`, which moves the goal closer to the source semantics. Then, we apply the constructor for equation semantics. Again, the semantics of variables are given by sampling the sts_1 stream. Finally, the semantics of the compiled transitions is shown by the auxiliary lemma `trans_exp_sem` which proceeds by induction over the list of transitions.

Finally, the proof must establish the semantics of the compiled sub-blocks under sw_2 ([lines 74-104](#)). Again, the subgoal starts by showing the correspondence between the

generated `switch` and `reset` blocks and the source `select` semantics. Then, the semantics of a compiled block $[blks_i]$ must be given under sampled H , and not sampled $H + H'$, to stay consistent with the inductive hypotheses `Hdom` and `Hat`. The `sem_block_refines` lemma states that, if a block has a semantics under a history H_1 , and a history H_2 refines H_1 , then the block has a semantics under H_2 . It can be applied because the `select` operator preserves the refinement stated in `Href`. Then, we apply the induction hypothesis `Hind`. It remains to prove all of the premises of `invariant 11` for $blks_i$. Typing, clock-typing, `NoDupLocals`, `GoodLocals` and semantics obligations are proven using the original inverted hypotheses. Since the new clock-typing environment is Γ_{ck} , the `Hinc1` hypothesis must be proven again by combining the original `Hinc1` with `Hinc12`. The obligation `Hat` is shown immediately. The domain of the sampled history is still included in that of Γ_{ty} , since `select` may only reduce history. Finally, the `select_sc_vars` lemma shows that `select` preserves the well-clocking of the history under a sampled environment. This concludes the proof for this case.

Although we have shown the overall structure of the proof, this presentation still omits some complex details: the case that we summarized requires around 300LoC in Coq. In particular, we have treated the application of sampling to histories as a partial function. In our actual Coq mechanization, it is specified as a relation, which complicates the proofs somewhat by introducing new existential histories which the proof needs to make “more concrete” to reason about. Unfortunately, we have not yet found the right definitions to eliminate these painful details.

Bibliography

- [13] *Simulink: User's Guide*. R2013a. Release 2013a. The Mathworks. Natick, MA, U.S.A., Mar. 2013.
- [22] *Stateflow User's Guide*. 10.7. Matlab & Simulink R2022b. The Mathworks. Natick, MA, U.S.A., Sept. 2022.
- [Abr+20] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. "Proof-Producing Synthesis of CakeML from Monadic HOL Functions". In: *Journal of Automated Reasoning (JAR)* (June 2020). URL: <https://rdcu.be/b4FrU>.
- [Ana+17] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Z. Weaver. "CertiCoq : A verified compiler for Coq". In: *CoqPL'17: The Third International Workshop on Coq for Programming Languages*. Jan. 2017.
- [And95] Charles André. *SYNCCHARTS: A Visual Representation of Reactive Behaviors*. Technical Report. RR 95-52. Sophia-Antipolis, France: I3S, Oct. 1995. URL: <http://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/SYNCCHARTS/SyncCharts.pdf>.
- [App03] A. W. Appel. "Foundational Proof-Carrying Code". In: *Foundations of Intrusion Tolerant Systems*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2003, p. 25.
- [Aug13] Cédric Auger. "Compilation certifiée de SCADE/LUSTRE". PhD thesis. Orsay, France: Univ. Paris Sud 11, Apr. 2013. URL: <https://tel.archives-ouvertes.fr/tel-00818169/document>.
- [Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *8th IFIP WG 10.2 Int. Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*. Vol. 6399. LNCS. Waidhofen an der

- Ybbs, Austria: Springer, Oct. 2010, pp. 35–46. URL: <https://hal.inria.fr/hal-01055378/document>.
- [BB91] Albert Benveniste and Gérard Berry. “The Synchronous Approach to Reactive and Real-Time Systems”. In: *Proc. IEEE* 79.9 (Sept. 1991), pp. 1270–1282. URL: <https://ptolemy.berkeley.edu/projects/chess/design/2010/discussions/Pdf/synclang.pdf>.
- [BC84] Gérard Berry and Laurent Cosserat. “The ESTEREL Synchronous Programming Language and its Mathematical Semantics”. In: *Seminar on Concurrency*. Ed. by Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel. Vol. 197. LNCS. Pittsburg, USA: Springer, July 1984, pp. 389–448.
- [Ber00] Gérard Berry. *The Esterel v5 Language Primer*. 5.91. Ecole des Mines and INRIA. July 2000. URL: <http://www-sop.inria.fr/members/Gerard.Berry/Papers/primer.zip>.
- [BH01] Sylvain Boulmé and Grégoire Hamon. “Certifying Synchrony for Free”. In: *Proc. 8th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*. Ed. by Robert Nieuwenhuis and Andrei Voronkov. Vol. 2250. LNCS. Havana, Cuba: Springer, Dec. 2001, pp. 495–506. URL: <https://hal.archives-ouvertes.fr/hal-01571762>.
- [Bie+08] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. “Clock-directed modular code generation for synchronous data-flow languages”. In: *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*. Tucson, AZ, USA: ACM Press, June 2008, pp. 121–130. URL: <https://www.di.ens.fr/~pouzet/bib/lctes08a.pdf>.
- [BJP22] Timothy Bourke, Paul Jeanmaire, and Marc Pouzet. “Towards a denotational semantics of streams for a verified Lustre compiler (short talk)”. In: *28th International Conference on Types for Proofs and Programs*. Nantes, France, June 2022. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_paper_28.pdf.
- [BL09] Sandrine Blazy and Xavier Leroy. “Mechanized Semantics for the Clight Subset of the C Language”. In: *J. Automated Reasoning* 43.3 (Oct. 2009), pp. 263–288. URL: <https://hal.inria.fr/inria-00352524/document>.
- [Bou21] Sylvain Boulmé. “Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)”. See also <http://www-verimag.imag.fr/~boulme/hdr.html>. Habilitation à diriger des recherches. Université Grenoble-Alpes, Sept. 2021. URL: <https://hal.science/tel-03356701>.
- [Bru20] Lélío Brun. “Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset”. PhD thesis. PSL Research University, June 2020. URL: <https://tel.archives-ouvertes.fr/tel-03068862>.

- [Cas+87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. “LUSTRE: A declarative language for programming synchronous systems”. In: *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1987)*. Munich, Germany: ACM Press, Jan. 1987, pp. 178–188. DOI: [10.1145/41625.41641](https://doi.org/10.1145/41625.41641). URL: https://www.cse.unsw.edu.au/~plaice/archive/JAP/P-ACM_POPL87-lustre.pdf.
- [CFS07] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. “Designing a Generic Graph Library Using ML Functors”. In: *Symposium on Trends in Functional Programming*. 2007.
- [Cha20] Arthur Charguéraud. “Separation Logic for Sequential Programs (Functional Pearl)”. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: [10.1145/3408998](https://doi.org/10.1145/3408998). URL: <https://doi.org/10.1145/3408998>.
- [CHP06] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. “Mixing Signals and Modes in Synchronous Data-flow Systems”. In: *Proc. 6th ACM Int. Conf. on Embedded Software (EMSOFT 2006)*. Ed. by Sang Lyul Min and Yi Wang. Seoul, South Korea: ACM Press, Oct. 2006, pp. 73–82. URL: <https://www.di.ens.fr/~pouzet/bib/emsoft06.pdf>.
- [Col+23] J.-L. Colaço, M. Mendler, B. Pauget, and M. Pouzet. “A Constructive State-based Semantics and Interpreter for a Synchronous Data-flow Language with State Machines”. In: *ACM TECS* same issue (2023).
- [CompCert] Xavier Leroy. *CompCert*. <https://github.com/AbsInt/CompCert>. 2023.
- [Coq] Coq Development Team. *The Coq proof assistant reference manual*. Inria. 2020. URL: <https://coq.inria.fr/distrib/current/refman/>.
- [CP01] Pascal Cuoq and Marc Pouzet. “Modular Causality in a Synchronous Stream Language”. In: *10th European Symposium on Programming (ESOP 2001), part of European Joint Conferences on Theory and Practice of Software (ETAPS 2001)*. Ed. by David Sands. Vol. 2028. LNCS. Genova, Italy: Springer, Apr. 2001, pp. 237–251. DOI: [10.1007/3-540-45309-1_16](https://doi.org/10.1007/3-540-45309-1_16).
- [CP03] Jean-Louis Colaço and Marc Pouzet. “Clocks as First Class Abstract Types”. In: *Proc. 3rd Int. Conf. on Embedded Software (EMSOFT 2003)*. Ed. by R. Alur and I. Lee. Vol. 2855. LNCS. Philadelphia, PA, USA: Springer, Oct. 2003, pp. 134–155. DOI: [10.1007/978-3-540-45212-6_10](https://doi.org/10.1007/978-3-540-45212-6_10).
- [CP04] Jean-Louis Colaço and Marc Pouzet. “Type-based initialization analysis of a synchronous dataflow language”. In: *Int. J. Software Tools for Technology Transfer* 6.3 (Aug. 2004), pp. 245–255. URL: <https://www.di.ens.fr/~pouzet/bib/sttt04.pdf>.

- [CP96] Paul Caspi and Marc Pouzet. “Synchronous Kahn Networks”. In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 226–238. ISBN: 0897917707. DOI: [10.1145/232627.232651](https://doi.org/10.1145/232627.232651). URL: <https://doi.org/10.1145/232627.232651>.
- [CP98] Paul Caspi and Marc Pouzet. “A Co-iterative Characterization of Synchronous Stream Functions”. In: *First Workshop on Coalgebraic Methods in Computer Science (CMCS'98)*. Vol. 11. ENTCS. Lisbon, Portugal: Elsevier Science, Mar. 1998, pp. 1–21. DOI: [10.1016/S1571-0661\(04\)00050-7](https://doi.org/10.1016/S1571-0661(04)00050-7).
- [CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “A Conservative Extension of Synchronous Data-flow with State Machines”. In: *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. Ed. by Wayne Wolf. Jersey City, USA: ACM Press, Sept. 2005, pp. 173–182. DOI: [10.1145/1086228.1086261](https://doi.org/10.1145/1086228.1086261). URL: <https://www.di.ens.fr/~pouzet/bib/emsoft05b.pdf>.
- [CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “Scade 6: A Formal Language for Embedded Critical Software Development”. In: *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*. Nice, France: IEEE Computer Society, Sept. 2017, pp. 4–15. URL: <https://hal.inria.fr/hal-01666470/document>.
- [CPP23] Jean-Louis Colaço, Baptiste Pauget, and Marc Pouzet. “Polymorphic Types with Polynomial Sizes”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2023. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 36–49. ISBN: 9798400701696. DOI: [10.1145/3589246.3595372](https://doi.org/10.1145/3589246.3595372). URL: <https://doi.org/10.1145/3589246.3595372>.
- [Dev17] Heptagon Developers. *Heptagon/BZR manual*. Apr. 2017. URL: <http://heptagon.gforge.inria.fr/pub/heptagon-manual.pdf>.
- [ELS93] Peter Eades, Xuemin Lin, and W.F. Smyth. “A fast and effective heuristic for the feedback arc set problem”. In: *Information Processing Letters* 47.6 (1993), pp. 319–323. ISSN: 0020-0190. DOI: [https://doi.org/10.1016/0020-0190\(93\)90079-0](https://doi.org/10.1016/0020-0190(93)90079-0). URL: <https://www.sciencedirect.com/science/article/pii/0020019093900790>.
- [EMSOFT21] Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. “Verified Lustre Normalization with Node Subsampling”. In: *ACM Trans. Embedded Computing Systems*. International Conference on Embedded Software 20.5s (Oct. 2021), Article 98. ISSN: 1539-9087. DOI: [10.1145/3477041](https://doi.org/10.1145/3477041).

-
- [EMSOFT23] Timothy Bourke, Basile Pesin, and Marc Pouzet. “Verified Compilation of Synchronous Dataflow with State Machines”. In: *ACM Transactions on Embedded Computing Systems, special issue for EMSOFT 2023*. International Conference on Embedded Software. Oct. 2023.
- [GC04] Eduardo Giménez and Pierre Castéran. *A Tutorial on (Co-)Inductive Types in Coq*. 2004. URL: <https://hal.science/hal-00344325>.
- [Gér+12] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. “A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler”. In: *Proc. 13th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2012)*. Ed. by Reinhard Wilhelm, Heiko Falk, and Wang Yi. Beijing, China: ACM Press, June 2012, pp. 51–60. URL: <https://www.di.ens.fr/~guatto/papers/lctes12.pdf>.
- [GTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. “Polychrony for system design”. In: *J. Circuits, Systems and Computers* 12.3 (2003), pp. 261–303. URL: <http://www.worldscinet.com/123/12/1203/S0218126603000763.html>.
- [Hal+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous dataflow programming language LUSTRE”. In: *Proc. IEEE* 79.9 (Sept. 1991), pp. 1305–1320. URL: <http://www-verimag.imag.fr/~halbwach/lustre-ieee.html>.
- [Ham05] Grégoire Hamon. “A Denotational Semantics for Stateflow”. In: *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*. Ed. by Wayne Wolf. Jersey City, USA: ACM Press, Sept. 2005, pp. 164–172. DOI: [10.1145/1086228.1086260](https://doi.org/10.1145/1086228.1086260).
- [Har+87] D. Harel, A. Pnuelli, J.P. Schmidt, and R. Sherman. “On the Formal Semantics of Statecharts”. In: *2nd IEEE Symposium on Logic in Computer Science*. 1987.
- [Har87] David Harel. “Statecharts: A Visual Formalism for Complex Systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274. DOI: [10.1016/0167-6423\(87\)90035-9](https://doi.org/10.1016/0167-6423(87)90035-9). URL: http://www.inf.ed.ac.uk/teaching/courses/seoc/2005_2006/resources/statecharts.pdf.
- [HN96] David Harel and Amnon Naamad. “The STATEMATE Semantics of Statecharts”. In: *ACM Trans. Software Engineering and Methodology (TOSEM)* 5.4 (Oct. 1996), pp. 293–333. DOI: [10.1145/235321.235322](https://doi.org/10.1145/235321.235322).
- [HP00] Gégioire Hamon and Marc Pouzet. “Modular Resetting of Synchronous Data-Flow Programs”. In: *Proc. 2nd ACM SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming (PPDP 2000)*. Ed. by Frank Pfenning. Montreal, Canada: ACM, Sept. 2000, pp. 289–300. DOI: [10.1145/351268.351300](https://doi.org/10.1145/351268.351300). URL: <https://www.di.ens.fr/~pouzet/bib/ppdp00.ps.gz>.

- [HR04] Grégoire Hamon and John Rushby. “An Operational Semantics for State-flow”. In: *Proc. 7th Int. Conf. on Fundamental Approaches to Software Engineering (FASE’04)*. Ed. by M. Wermelinger and T. Margaria-Steffen. Vol. 2984. LNCS. Barcelona, Spain: Springer, Apr. 2004, pp. 229–243. URL: <http://www.csl.sri.com/users/rushby/papers/sttt07.pdf>.
- [HSCC14] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. “A Type-Based Analysis of Causality Loops in Hybrid Modelers”. In: *Proc. 17th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2014)*. Ed. by Martin Fränzle and John Lygeros. Berlin, Germany: ACM Press, Apr. 2014, pp. 71–82. DOI: [10.1145/2562059.2562125](https://doi.org/10.1145/2562059.2562125). URL: <http://www.tbrk.org/papers/abstracts.html#hsc14>.
- [Jea19] Paul Jeanmaire. *Propriétés dynamiques du système d’horloges de Lustre*. Master’s thesis. 2019.
- [JFLA21] Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. “Normalisation vérifiée du langage Lustre”. In: *Journées Francophones des Langages Applicatifs (Apr. 2021)*. URL: <https://hal.archives-ouvertes.fr/hal-03190426>.
- [JFLA23] Timothy Bourke, Basile Pesin, and Marc Pouzet. “Analyse de dépendance vérifiée pour un langage synchrone à flot de données”. In: *Journées Francophones des Langages Applicatifs (Jan. 2023)*. Ed. by Timothy Bourke and Delphine Demange, pp. 101–120. URL: <https://hal.inria.fr/hal-03936656>.
- [JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. “Validating LR(1) parsers”. In: *21st European Symposium on Programming (ESOP 2012), part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012)*. Ed. by Helmut Seidl. Vol. 7211. LNCS. Tallinn, Estonia: Springer, Mar. 2012, pp. 397–416. URL: <https://hal.inria.fr/hal-01077321/document>.
- [Kah74] Gilles Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *Proc. Int. Federation for Information Processing (IFIP) Congress 1974*. Ed. by Jack L. Rosenfeld. Stockholm, Sweden: North-Holland, Aug. 1974, pp. 471–475. URL: https://perso.ensta-paristech.fr/~chapoutot/various/kahn_networks.pdf.
- [Kum+14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. “CakeML: A Verified Implementation of ML”. In: *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*. San Diego, CA, USA: ACM Press, Jan. 2014, pp. 179–191. URL: <https://cakeml.org/pop14.pdf>.

- [LCTES11] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. “Divide and Recycle: Types and Compilation for a Hybrid Synchronous Language”. In: *Proc. 12th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2011)*. Ed. by Jan Vitek and Bjorn De Sutter. Chicago, USA: ACM Press, Apr. 2011, pp. 61–70. DOI: [10.1145/1967677.1967687](https://doi.org/10.1145/1967677.1967687). URL: <http://www.tbrk.org/papers/abstracts.html#lctes11>.
- [LeG+91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. “Programming Real-Time Applications with SIGNAL”. In: *Proc. IEEE* 79.9 (Sept. 1991), pp. 1321–1336. URL: <https://hal.inria.fr/inria-00540460/document>.
- [Ler09a] Xavier Leroy. “A formally verified compiler back-end”. In: *J. Automated Reasoning* 43.4 (Dec. 2009), pp. 363–446. URL: <http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf>.
- [Ler09b] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Comms. ACM* 52.7 (2009), pp. 107–115. URL: <https://hal.inria.fr/inria-00415861/document>.
- [LG09] Xavier Leroy and Hervé Grall. “Coinductive big-step operational semantics”. In: *Information and Computation* 207.2 (2009), pp. 284–304. URL: <http://xavierleroy.org/publi/coindsem-journal.pdf>.
- [Mor07] Lionel Morel. “Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation”. In: *EURASIP Journal on Embedded Systems* (2007), p. 59130. URL: <https://hal.science/hal-00292876>.
- [MR01] Florence Maraninchi and Yann Rémond. “Argos: an automaton-based synchronous language”. In: *Computer Languages* 27.1–3 (2001), pp. 61–92. URL: <https://hal.archives-ouvertes.fr/hal-00273055/document>.
- [MR03] Florence Maraninchi and Yann Rémond. “Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems”. In: *Science of Computer Programming* 46.3 (2003), pp. 219–254. DOI: [10.1016/S0167-6423\(02\)00093-X](https://doi.org/10.1016/S0167-6423(02)00093-X).
- [MR98] F. Maraninchi and Y. Rémond. “Mode-Automata: About Modes and States for Reactive Systems”. In: *7th European Symposium on Programming (ESOP 1998), part of European Joint Conferences on Theory and Practice of Software (ETAPS 1998)*. Vol. 1381. LNCS. Lisbon, Portugal: Springer, Mar. 1998, pp. 185–189. DOI: [10.1007/BFb0053571](https://doi.org/10.1007/BFb0053571).
- [Nec97] George C. Necula. “Proof-carrying code”. In: *ACM-SIGACT Symposium on Principles of Programming Languages*. 1997.
- [Nig22] Pierre Nigron. “Programmes avec effets et leurs preuves dans la théorie des types : application à la compilation certifiée et aux traitements de paquets certifiés”. Theses. Sorbonne Université, Nov. 2022. URL: <https://theses.hal.science/tel-04028224>.

- [Owe+16] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. “Functional Big-Step Semantics”. In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 589–615. ISBN: 978-3-662-49498-1.
- [Pau08] Lawrence C. Paulson. *The Isabelle Reference Manual*. Uni. of Cambridge. June 2008.
- [Pau09] Christine Paulin-Mohring. “A constructive denotational semantics for Kahn networks in Coq”. In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin. Cambridge, UK: CUP, 2009, pp. 383–413. URL: <https://hal.inria.fr/inria-00431806/document>.
- [Pes20] Basile Pesin. *Normalisation du langage Lustre dans l’assistant de preuve Coq*. Master’s thesis. 2020.
- [PLDI17] Timothy Bourke, Lélío Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. “A Formally Verified Compiler for Lustre”. In: *Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain: ACM Press, June 2017, pp. 586–601. DOI: [10.1145/3062341.3062358](https://doi.org/10.1145/3062341.3062358). URL: <http://www.tbrk.org/papers/abstracts.html#pldi2017>.
- [POPL20] Timothy Bourke, Lélío Brun, and Marc Pouzet. “Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset”. In: *Proc. of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–29. DOI: [10.1145/3371112](https://doi.org/10.1145/3371112). URL: <http://www.tbrk.org/papers/abstracts.html#popl2020>.
- [Pot15] François Pottier. “Depth-First Search and Strong Connectivity in Coq”. In: *Journées Francophones des Langages Applicatifs (JFLA)*. Jan. 2015. URL: <http://cambium.inria.fr/~fpottier/publis/fpottier-dfs-scc.pdf>.
- [Pou06] Marc Pouzet. *Lucid Synchronic, v. 3. Tutorial and reference manual*. Université Paris-Sud. Apr. 2006. URL: <https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf>.
- [Pou10] Marc Pouzet. *Zélus*. <https://github.com/INRIA/zelus>. 2010.
- [PR09] Marc Pouzet and Pascal Raymond. “Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation”. In: *Proc. 9th ACM Int. Conf. on Embedded Software (EMSOFT 2009)*. Grenoble, France: ACM Press, Oct. 2009, pp. 215–224. URL: <https://www.di.ens.fr/~pouzet/bib/emsoft09.pdf>.
- [PR16] François Pottier and Yann Régis-Gianas. *Menhir Reference Manual*. Inria. Aug. 2016. URL: <https://gallium.inria.fr/~fpottier/menhir/manual.pdf>.

- [PSS98a] Amir Pnueli, M. Siegel, and Ofer Shtrichman. “Translation Validation for Synchronous Languages”. In: *Proc. 25th Int. Colloq. on Automata, Languages and Programming*. Ed. by Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Vol. 1443. LNCS. Springer, 1998, pp. 235–246. DOI: [10.1007/BFb0055057](https://doi.org/10.1007/BFb0055057).
- [PSS98b] Amir Pnueli, Michael Siegel, and Eli Singerman. “Translation Validation”. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 1998.
- [RB22] Lionel Rieg and Gérard Berry. *Towards Coq-verified Esterel Semantics and Compiling*. arXiv. Sept. 2022. DOI: [10.48550/ARXIV.1909.12582](https://doi.org/10.48550/ARXIV.1909.12582). arXiv: [1909.12582v3](https://arxiv.org/abs/1909.12582v3) [cs.FL].
- [RL10] Silvain Rideau and Xavier Leroy. “Validating register allocation and spilling”. In: *Compiler Construction (CC 2010)*. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 224–243. URL: <http://xavierleroy.org/publi/validation-regalloc.pdf>.
- [Sew+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. “Ott: Effective Tool Support for the Working Semanticist”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 1–12. ISBN: 9781595938152. DOI: [10.1145/1291151.1291155](https://doi.org/10.1145/1291151.1291155). URL: <https://doi.org/10.1145/1291151.1291155>.
- [Shi+17] Gang Shi, Yuanke Gan, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew. “A Formally Verified Sequentializer for Lustre-Like Concurrent Synchronous Data-Flow Programs”. In: *Proc. 39th Int. Conf. on Software Engineering Companion (ICSE-C’17)*. Buenos Aires, Argentina: IEEE Press, May 2017, pp. 109–111. DOI: [10.1109/ICSE-C.2017.83](https://doi.org/10.1109/ICSE-C.2017.83).
- [Shi+19] Gang Shi, Yucheng Zhang, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew. “A formally verified transformation to unify multiple nested clocks for a Lustre-like language”. In: *Science China Information Sciences* 62.1 (Jan. 2019), article 12801. DOI: [10.1007/s11432-016-9270-0](https://doi.org/10.1007/s11432-016-9270-0).
- [SN08] Konrad Slind and Michael Norrish. “A Brief Overview of HOL4”. In: *Proc. 21st Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2008)*. Ed. by Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. LNCS. Montreal, Canada: Springer, Aug. 2008, pp. 28–32. URL: https://ts.data61.csiro.au/publications/nicta_full_text/1482.pdf.
- [Soz07] Matthieu Sozeau. “Subset Coercions in Coq”. In: *Lecture Notes in Computer Science* 4502 (2007), pp. 237–252. DOI: [10.1007/978-3-540-74464-1_16](https://doi.org/10.1007/978-3-540-74464-1_16). URL: <https://hal.inria.fr/inria-00628869>.

- [Str02] Martin Strecker. “Formal Verification of a Java Compiler in Isabelle”. In: *Automated Deduction—CADE-18*. 2002, pp. 63–77. DOI: [10.1007/3-540-45620-1_5](https://doi.org/10.1007/3-540-45620-1_5).
- [Tan+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. “A New Verified Compiler Backend for CakeML”. In: *Proc. 21st ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2016)*. Nara, Japan: ACM Press, Sept. 2016, pp. 60–73. URL: <https://cakeml.org/icfp16.pdf>.
- [Tan+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. “The verified CakeML compiler backend”. In: *Journal of Functional Programming* 29 (2019).
- [Tec05] Esterel Technologies. *The Esterel v7 Reference Manual*. v7_30. Esterel Technologies. Villeneuve-Loubet, France, Nov. 2005.
- [VB14] Jérôme Vouillon and Vincent Balat. “From Bytecode to JavaScript: the Js_of_ocaml Compiler”. In: *Software: Practice and Experience* 44 (Aug. 2014). DOI: [10.1002/spe.2187](https://doi.org/10.1002/spe.2187).
- [Wad92] Philip Wadler. “Monads for functional programming”. In: *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. Ed. by Manfred Broy. Vol. 118. NATO ASI Series. Springer, 1992, pp. 233–264. URL: <https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf>.
- [Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proc. 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA: ACM Press, June 2011, pp. 283–294. URL: <https://www.flux.utah.edu/paper/yang-pldi11>.

Index

clock-type system	90	graph analysis	78, 136, 157
clock	36	AcyGraph	64
lustre	182	optimizations	
merge	34	dataflow	134
reset	44	constant propagation	141
switch	41	dead equation elimination .	135
when	34	inlining	140
constant	34	minimization	138, 142
equation	38	imperative	121
node	36	dead code elimination	122
node instantiation	38	fusion of conditionals .	121, 162
operator	34	semantics	
variable	34	clock	75
compilation		history	28, 97
fby normalization	138	lustre	29
last normalization	106	fby	34
switch block	100, 135	last	46
back-end	166	merge	32
completion	94	reset	43, 44, 130
cutting update cycles	154	switch	40
elaboration	10, 86	when	32, 40
generating identifiers	80, 115	clock correctness	74
local declaration	102	clocked semantics	95, 97
nlustre to stc	148	constant	31
parsing	10, 79	determinism	73
scheduling	153, 158, 167	equation	29
state machine	98	initialization arrow	35
stc to obc	151, 152, 159, 165	local declaration	45, 97
transcription	130		
unnesting	105		

node	29, 97
node instantiation	35, 97
operator	31
state machine	48, 49
variable	29
nlustre	
fbyreset	129
holdreset	129
mfbyreset	128
reset-ind	126
coinductive model	124, 130
indexed model	126
memory model	127, 149
obc	120
stc	145, 149
initialization	146
iterated semantics	148
transition constraint	147
streams	23
coinductive	25, 52, 124
EqSt	26
Str_nth	26
init_from	26
map	26
indexed	24, 126

RÉSUMÉ

Les systèmes embarqués critiques sont souvent spécifiés par des formalismes schéma-bloc. SCADe Suite est un environnement de développement pour ces systèmes utilisé depuis vingt ans dans l'industrie avionique, nucléaire, automobile, et autres domaines critiques. Son formalisme graphique se traduit en une représentation textuelle basée sur le langage synchrone à flots de données Lustre, et incorpore des fonctionnalités de langages plus récents comme Lucid Synchrone. En Lustre, un programme est défini comme un ensemble d'équations qui spécifie la relation entre entrées et sorties du programme à chaque instant. Le langage des expressions inclut des opérateurs arithmétiques et logiques, des opérateurs de délais qui permettent d'accéder à la précédente valeur d'une expression, et des opérateurs d'échantillonnage qui permettent à certaines valeurs d'être calculées moins souvent que d'autres.

Le projet Vélus est une formalisation d'un sous-ensemble du langage Scade 6 dans l'assistant de preuves Coq. Il propose une formalisation de la sémantique dynamique du langage sous forme de relations entre flots infinis d'entrées et de sorties. Il inclut aussi un compilateur qui utilise CompCert, un compilateur vérifié pour C, pour produire du code assembleur. Enfin, il fournit une preuve de bout-en-bout que ce compilateur préserve la sémantique à flots de données des programmes sources.

Cette thèse étend Vélus en y ajoutant les blocs de contrôles de Scade 6 et Lucid Synchrone, ce qui inclut une construction qui contrôle l'activation des équations selon une condition (switch), une construction permettant d'accéder à la valeur précédente d'une variable (last), une construction qui réinitialise les opérateurs de délai (reset), et, enfin, des machines à états hiérarchiques, qui permettent la spécification de comportements modaux complexes. Toutes ces constructions peuvent être arbitrairement imbriquées dans un programme. Nous étendons la sémantique de Vélus avec une nouvelle spécification pour ces constructions qui encode leur comportement par l'échantillonnage. Nous proposons un schéma d'induction générique pour les programmes bien formés qui permet de prouver certaines propriétés du modèle sémantique, comme son déterminisme ou l'adhérence des valeurs aux types déclarés. Enfin, nous décrivons la compilation de ces constructions telle qu'implémentée dans Vélus. Nous montrons que le modèle de compilation qui réécrit ces constructions dans le langage noyau peut être implémenté, spécifié et vérifié dans Coq. La compilation de last et reset nécessite des changements plus profonds dans les langages intermédiaires de Vélus.

MOTS CLÉS

langages synchrones à flots de données, Lustre, Scade, compilation vérifiée, machines à états, Vélus, Coq

ABSTRACT

Safety-critical embedded systems are often specified using block-diagram formalisms. SCADe Suite is a development environment for such systems which has been used industrially in avionics, nuclear plants, automotive and other safety-critical contexts for twenty years. Its graphical formalism translates to a textual representation based on the Lustre synchronous dataflow language, with extensions from later languages like Lucid Synchrone. In Lustre, a program is defined as a set of equations that relate inputs and outputs of the program at each discrete time step. The language of expressions at right of equations includes arithmetic and logic operators, delay operators that access the previous value of an expression, and sampling operators that allow some values to be calculated less often than others.

The Vélus project aims at formalizing a subset of the Scade 6 language in the Coq Proof Assistant. It proposes a specification of the dynamic semantics of the language as a relation between infinite streams of inputs and outputs. It also includes a compiler that uses CompCert, a verified compiler for C, as its back end to produce assembly code, and an end-to-end proof that compilation preserves the semantics of dataflow programs.

In this thesis, we extend Vélus to support control blocks present in Scade 6 and Lucid Synchrone, which includes a construction that controls the activation of equations based on a condition (switch), a construction that accesses the previous value of a named variable (last), a construction that re-initializes delay operators (reset), and finally, hierarchical state machines, which allow for the definition of complex modal behaviors. All of these constructions may be arbitrarily nested in a program. We extend the existing semantics of Vélus with a novel specification for these constructs that encodes their behavior using sampling. We propose a generic induction principle for well-formed programs, which is used to prove properties of the semantic model such as determinism and type system correctness. Finally, we describe the extension of the Vélus compiler to handle these new constructs. We show that the existing compilation scheme that lowers these constructs into the core dataflow language can be implemented, specified and verified in Coq. Compiling the reset and last constructs requires deeper changes in the intermediate languages of Vélus.

KEYWORDS

synchronous dataflow languages, Lustre, Scade, verified compilation, state machines, Vélus, Coq