Compilation certifiée de SCADE/LUSTRE

Cédric Auger

Université Paris Sud

École Doctorale d'informatique de Paris Sud

Équipe PARKAS

THÈSE présentée par :

Cédric AUGER

soutenue le : 7 février 2013

pour obtenir le grade de : Docteur de l'Université Paris Sud

Discipline : Informatique

Compilation Certifiée de SCADE/LUSTRE

THÈSE DIRIGÉE PAR :

POUZET Marc	Professeur, École Normale Supérieure de Paris

PRÉSIDENT DU JURY :

BENZAKEN Véronique	Professeur, Université Paris-Sud

RAPPORTEURS :

BLAZY Sandrine	Professeur, Université de Rennes 1
DE SIMONE Robert	Directeur de Recherche, INRIA Sophia-Antipolis

EXAMINATEURS :

PAGANO Bruno	Directeur Scientifique, ESTEREL TECHNOOLOGIES

Résumé

Les langages synchrones sont apparus autour des années quatre-vingt, en réponse à un besoin d'avoir un modèle mathématique simple pour implémenter des systèmes temps réel critiques. Dans ce modèle, le temps est découpé en instants discrets durant lesquels tous les composants du système reçoivent et produisent une donnée. Cette modélisation permet des raisonnements beaucoup plus simples en évitant de devoir prendre en compte le temps de calcul de chaque opération. Dans le monde du logiciel critique, la fiabilité du matériel et de son fonctionnement sont primordiaux, et on accepte d'être plus lent si on devient plus sûr.

Afin d'augmenter cette fiabilité, plutôt que de concevoir manuellement tout le système, on utilise des machines qui synthétisent automatiquement le système souhaité à partir d'une description la plus concise possible. Dans le cas du logiciel, ce mécanisme s'appelle la compilation, et évite des erreurs introduites par l'homme par inadvertance. Elle ne garantit cependant pas la bonne correspondance entre le système produit et la description donnée.

Des travaux récents menés par une équipe INRIA dirigée par Xavier Leroy ont abouti en 2008 au compilateur CompCert d'un sous-ensemble large de C vers l'assembleur PowerPC pour lequel il a été prouvé dans l'assistant de preuve Coq que le code assembleur produit correspond bien à la description en C du programme source. Un tel compilateur offre des garanties fortes de bonne correspondance entre le système synthétisé et la description donnée. De plus, avec les compilateurs utilisés pour le temps réel critique, la plupart des optimisations sont désactivées afin d'éviter les erreurs qui y sont liées. Dans CompCert, des optimisations elles aussi prouvées sont proposées, ce qui pourrait permettre ces passes dans la production de systèmes temps réel critiques sans en compromettre la fiabilité.

Le but de cette thèse est d'avoir une approche similaire mais spécifique à un langage synchrone, donc plus approprié à la description de systèmes temps réel critiques que ne l'est le C. Un langage synchrone flots de données semblable à Lustre, nommé Ls, et un langage impératif semblable au langage C, nommé Obc y sont proposés ainsi que leur sémantique formelle et une chaîne de compilation avec des preuves de préservation de sémantique le long de cette chaîne.

Mots clés : Compilation, sémantique, synchrone, preuve de programmes, traduction par validation, langage flots de données, langage impératif, certification.

Abstract

Synchronous languages first appeared during the 80's, in order to provide a mathematical model for safety-critical systems. In this model, time is discrete. At each instant, all components of the system simultaneously receive and produce some data. This model allows simpler reasonning on the behaviour of the system, as it does not involve the time required for each of the operations for every component. In safety-critical systems, safety is the rule, so a poor performance behaviour can be allowed if it improves safety.

In order to improve safety, rather than conceiving directly the system, machines are used to automatically design the system from a given concise description. In the case of software, this machine is called a compiler, and avoids issues due to some human inadvertence. But it does not ensure that the produced system and the description specification really shows the same behaviour.

Some recent work from an INRIA team lead by Xavier Leroy achieved in 2008 the realisation of the CompCert compiler from a large subset of C to PowerPC assembly, for which it was proven inside of the Coq proof assistant that the produced system fits its source description. Such a compiler offers strong guarantees that the produced system and its given description by the programmer really fit. Furthermore, most current compiler's optimizations are disabled when dealing with safety-critical systems in order to avoid tedious compilation errors that optimizations may introduce. Proofs for optimizations may allow their use in this domain without affecting the faith we could place in the compiler.

The aim of this thesis is to follow a similar path, but this one on a language which would be more suited for safety-critical systems than the C programming language. Some dataflow synchronous programming language very similar to Lustre, called Ls is described with its formal semantics, as well as an imperative programming language similar to a subset of C called Obc. Furthermore some compilation process is described as well as some proofs that the semantics is preserved during the compilation process.

Keywords : Compilation, semantics, synchronous, proofs of programs, translation validation, dataflow languages, imperative languages, certification.

Table des matières

Table des figures

Introduction

0.1 Quel est le titre de cette thèse ?

0.1.1 Langages, systèmes formels, sémantique, compilation

Un langage est un ensemble de suites de caractères qui sont soumises à des contraintes. Dans les langages naturels, ainsi que dans les langages de programmation, ces contraintes sont appelées les règles syntaxiques ou encore les règles de grammaire du langage.

Un système formel est une modélisation mathématique d'un langage, dans le but de le définir sans ambiguïté ainsi que de donner la possibilité d'y vérifier des propriétés de façon automatique. Il explique donc comment construire les objets qui nous intéressent. Dans cette thèse de nombreux systèmes formels apparaissent sous la forme de règles d'inférence [1]. Les langues naturelles, par exemple, ne sont jamais décrites dans un système formel alors que les expressions arithmétiques le sont. De telles règles seraient trop complexes à écrire. De plus une langue peut être amenée à évoluer et il existe des dissensions sur le fait de savoir si une phrase donnée est ou n'est pas correcte.

Une sémantique est le sens que l'on veut donner aux objets étudiés. Par exemple :
- Un dictionnaire expose une liste de mots et leur définition ; le mot est l'objet étudié, et sa sémantique est donnée par la définition.
- La phrase «le chat vert vole le soleil» est une phrase correcte en français (sujet, verbe, complément), mais dénuée de sens. Elle n'a donc pas de sémantique.
- $2 + 2$ est une expression mathématique valide, dont la sémantique la valeur quatre ; alors que $\frac{2}{0}$ n'a pas de sémantique.

Les sémantiques des langages de programmations se décrivent assez naturellement par des systèmes formels, contrairement aux langues naturelles.

Enfin, la compilation est le nom que les informaticiens ont donné à la traduction. Comme les anglais, les machines ne comprennent pas le français. Il faut donc traduire ce que l'on veut exprimer pour se faire comprendre. Les machines ont leur propre langage, et donner un ordre à une machine se fait généralement en deux temps. Dans un premier temps, il faut traduire formellement ce que l'on souhaite dans un langage adapté (le langage de programmation), ensuite on fait appel à un traducteur extérieur appelé compilateur, capable de comprendre le langage de programmation et le langage de la machine. Un bon compilateur est un compilateur capable de traduire tout en gardant le sens de ce que l'on veut. C'est là la problématique de cette thèse : « Comment traduire pour la machine en s'assurant que la traduction n'altère pas le sens ? »

0.1.2 Systèmes critiques et temps réel

Les systèmes critiques temps réel sont utilisés pour des systèmes où des vies humaines, l'environnement ou du matériel sont en jeu (comme dans le domaine médical, l'aviation ou le nucléaire). Dans ces situations, on n'accepte pas qu'un temps de réponse trop long ait pour conséquence une divergence de comportement vis-à-vis du comportement prévu, c'est pourquoi on veut pouvoir estimer et maintenir en dessous d'un certain seuil le temps de réponse ; une faille dans le système provoquant son arrêt est donc inacceptable, ceci

1. Une règle d'inférence est la donnée d'une propriété voulue et d'un ensemble de conditions qui impliquent cette propriété.

se traduisant par un temps de réponse infini. Pour ces raisons de sûreté, de prévisibilité et de rapidité, ces dispositifs sont souvent conçus pour être des circuits électroniques, qui n'interagissent donc pas avec un système d'exploitation. En particulier, la création dynamique de processus, ainsi que l'allocation dynamique de mémoire y sont généralement interdits.

0.1.3 Les langages synchrones [5]

Il existe plusieurs formes de systèmes, selon leur interaction avec l'environnement dans lequel ils sont plongés :

- les systèmes transformationnels, qui ne requièrent aucune notion de temps ; une fois les entrées lues, le système évolue en vase clos et rend une sortie. La composition de tels systèmes est assez simple, on envoie une requête, on attend la réponse et on continue.
  Exemples : les compilateurs, le calcul des décimales de pi, ...
- les systèmes interactifs, qui ne requièrent pas de notion de temps non plus, mais qui ne fonctionnent pas en vase clos, l'environnement extérieur au système communique en permanence avec le programme. La composition devient plus complexe car on ne peut plus se contenter d'attendre une réponse qui peut ne jamais venir avant de continuer, et peut donc être oubliée ou traitée ultérieurement.
  Exemples : un navigateur web, le système d'exploitation, ...
- les systèmes synchrones, qui requièrent une notion de temps partagé, comme pour les systèmes interactifs, on a une communication avec l'environnement extérieur, mais cette communication est discrète (par opposition à continue), et ce qui compte n'est pas nécessairement un temps physique, mais l'ordre dans lequel des requêtes sont émises. Dans ce modèle, il n'y a pas de temps qui s'écoule entre deux communications avec l'environnement extérieur, et la réponse est donnée instantanément (ie. avant la communication suivante avec l'environnement). La composition est à nouveau simple, mais soumise à des contraintes de causalité.
  Exemples : certaines applications multimédia qui découpent un film en instants et s'assurent ainsi qu'il n'y a pas pour l'utilisateur de décalage entre l'image et le son, les systèmes de contrôles qui doivent s'assurer que tous les moteurs travaillent ensemble, ...

  Les systèmes synchrones sont très jeunes, puisqu'ils ne datent que des années 80.

Une des premières applications industrielles des langages synchrones a été le lancement de l'Airbus A320 en 1984. Jusque là, les avions de grandes lignes contenaient beaucoup de systèmes mécaniques, tels que le « manche à balais » qui actionnait les ailerons de l'avion par un système mécanique de roues et poulies. Les systèmes « fly by wire » qui transmettaient les commandes du pilote à des moteurs via des signaux électriques étaient moins courants et jusqu'alors réservés à des avions expérimentaux à quelques exceptions près, telles que le Concorde en 1969. Les systèmes mécaniques peuvent souvent être considérés comme synchrones. Le temps qui s'écoule entre le moment où le pilote incline le « manche à balais » et le moment où les deux ailerons bougent est du à l'élasticité des matériaux et peut être négligé. En revanche dans un système digitalisé, les informations sont analysées et traitées avant de demander aux moteurs de se mettre en marche, ce qui fait que les ai-

lerons peuvent s'actionner avec un décalage. Dans certains cas, ce genre de décalage peut avoir de graves conséquences, par exemple si un moteur traite un ancien ordre « baisse » alors qu'un autre moteur traite le nouvel ordre « lève », la pièce actionnée peut se tordre, se déchirer ou se casser, ou les moteurs peuvent tomber en panne.

Une autre application est celle de la construction de réacteurs nucléaires de seconde génération la même année. Les premiers réacteurs nucléaires datent de 1950 et ont toujours été l'objet des plus hautes normes de sécurité. Les centrales nucléaires tournent en permanence et nécessitent plusieurs heures pour être arrêtées. Il y est nécessaire de repérer les écarts pour les corriger le plus tôt possible. Là encore, pour savoir si un calcul relève ou non d'un écart, il faut s'assurer que les mesures sur lesquelles le calcul a été effectué aient bien eu lieu au même instant.

De fait les systèmes critiques sont souvent modélisés par des systèmes synchrones, et la notion de temps coïncide pratiquement avec celle du temps physique auquel nous sommes habitués, c'est-à-dire qu'on exige que le temps physique qui s'écoule entre deux communications soit inférieur à une durée fixe.

L'un des points forts des langages synchrones et d'avoir une modélisation la plus simple possible qui permet d'augmenter la confiance que l'on peut placer dans ses programmes et dans les outils d'analyse de ces programmes.

Il y a plusieurs grandes familles de langages synchrones :

- **Les langages flot de contrôle**, comme Esterel [6], qui restent assez proches des langages impératifs ; les boucles y sont explicites, de même que l'ordre dans lequel les instructions sont exécutées.
- **Les langages flot de données**, qui sont de nature déclarative (on ne peut en effet pas calculer un flot puis un autre, ces calculs étant souvent entrelacés), et qui se prête assez bien aux représentations schéma-blocs. Dans cette famille, on trouve des langages tels que Signal [22], Lustre [27] ou Lucid Synchrone [10].
  Dans ces langages on écrit des relations que l'on souhaite voir vérifier entre les flots considérés, c'est au compilateur de décider comment calculer pour réaliser ces contraintes.
- **Les langages instantanés**, qui sont une sorte de mélange des deux précédents. Ces langages décrivent juste la relation entre les valeurs reçues, les valeurs retournées et l'état du système avant et après réaction. Il s'agit donc d'automates finis.

0.1.4 Certification, qualification, validation et vérification

Tout logiciel critique, de par sa nature, est amené à être expertisé pour recevoir des acceptations de mise en production par les autorités compétentes du domaine, qui engagent dès lors leur propre responsabilité.

Il existe de nombreuses normes dans les secteurs liés aux systèmes critiques, telles que la norme DO178B dans l'avionique (maintenant remplacée par la norme DO178C), la norme IEC-60880 pour les centrales nucléaires ou encore EN-50128 pour le transport ferroviaire.

Ces normes définissent un certain nombre de niveaux de criticité, et pour chaque niveau de criticité, des points à remplir pour son développement. Ainsi, les niveaux de criticité

les plus bas, comme par exemple le bon fonctionnement des lampes des passagers d'un système de transport ne sont souvent soumises à aucune règle, alors que le système de freins qui peut amener à la mort des usagers et la destruction du véhicule va être soumis aux règles les plus strictes de la norme.

Pour la mise en production de système critique, il faut faire appel à une autorité de certification qui va délivrer un certificat attestant que le système a été produit dans les normes. Cette certification est longue et coûteuse, c'est pourquoi certaines entreprises telles que Esterel Technologies ou CLEARSY proposent des outils de développement (comme SCADE ou l'Atelier B) qui ont reçu l'aval de ces autorités, afin d'accélérer et d'alléger la certification.

La **certification** d'un système est le processus d'acceptation du système par les autorités compétentes.

La **qualification** d'un logiciel est un processus en vue d'alléger le travail de certification ; normalement, tout le système doit être analysé, mais si des parties du systèmes sont automatisés, la qualification du processus d'automatisation permet de ne pas avoir à certifier ces parties. C'est typiquement le cas des compilateurs, on utilise un compilateur qualifié afin de ne pas avoir à certifier le code binaire qu'il produit ; bien entendu, il faut quand même certifier le code source, mais c'est en général bien plus aisé.

La **certification par validation** est une technique de certification [48, 44]. On définit une propriété qui doit être vérifiée par un système S, et un système certifié C qui décide si S vérifie ou non cette propriété. C est alors une autorité de certification. Le système S n'a alors plus nécessairement besoin d'avoir été engendré par un logiciel qualifié, s'il est systématiquement certifié par validation. Cette technique a été notamment exploitée par le projet CompCert.

La **vérification** consiste en la formalisation d'une spécification et la démonstration que cette spécification est bien vérifiée. Vérifier directement un générateur de code est une tâche souvent bien trop ardue pour être envisageable. Il est cependant courant de coupler un compilateur avec des outils de vérification. Cependant les outils de vérification sont appelés sur chaque code produit, ce qui peut ralentir le processus de développement, et remonter à la source d'une erreur n'est pas toujours facile.

0.2 Motivation

De nombreux outils existent pour évaluer le temps de réponse dans le pire cas (WCET pour "worst case execution time") [63]. Ces outils sont là pour assurer le bon comportement du matériel vis-à-vis de contraintes physiques, mais n'apportent aucune garantie quant à la correction fonctionnelle du code.

On dispose aussi d'outils pour l'aspect sémantique d'un système, par exemple basés sur le model-checking[13] (TLA+ [35] par exemple), qui permettent de vérifier que deux systèmes dans le même formalisme ont le même comportement observationnellement, afin de choisir le plus efficace des deux, ou encore de voir si un système répond à une spécification donnée.

Ces outils permettent de se donner un formalisme sur lequel raisonner, et s'ils ont été

qualifiés, produisent des systèmes qui peuvent ne pas avoir à être certifiés. Il est alors souvent beaucoup plus facile de certifier le travail en amont.

Cependant une problématique apparaît : il est plus facile de raisonner sur le formalisme et les modèles que l'on se donne pour programmer les systèmes, et par conséquent avoir un niveau de confiance élevé dans ce que devrait faire le système ; mais au final, seul comptera le système produit, donc lui seul a réellement besoin d'être certifié, malheureusement, ce travail est beaucoup plus ardu, long est susceptible d'erreurs que celui que l'on peut faire en amont.

C'est donc là qu'entre en jeu la qualification logicielle. Les experts qualifiant le logiciel doivent s'assurer que le système engendré suit exactement la spécification initiale. À l'heure actuelle ce travail de qualification repose sur des méthodes de développement éprouvées, prenant en compte divers critères tels que la traçabilité (possibilité, entre autres de pouvoir expliquer une anomalie de comportement du système produit par une faiblesse des spécifications initiales), une documentation précise du logiciel, une batterie de tests couvrant les spécifications données, ... L'usage de méthodes formelles commence cependant à apparaître dans la qualification logicielle, notamment dans la norme DO178C, qui succède à la norme DO178B utilisée à l'heure actuelle dans l'avionique. Ceci devrait apporter une forte garantie supplémentaire à la confiance que l'on peut dans le système certifié, puisqu'il suffit dès lors de n'avoir confiance que dans la spécification initiale et la validité de la méthode formelle utilisée.

Dans cet esprit, la qualification de SCADE V6 a été simplifiée en écrivant ce générateur de code dans une version certifiée d'OCaml pour la norme DO-178B plutôt que dans une version qualifiée de C comme c'était fait précédemment.

Cet intérêt pour les méthodes formelles s'est aussi traduit par la réalisation de CompCert, un compilateur d'un gros fragment du langage C vers l'assembleur PowerPC prouvé correct dans l'assistant de preuves Coq [36].

Dans ce contexte, il serait intéressant de disposer de générateurs de code muni d'une preuve formelle de correspondance entre la spécification souhaitée et le comportement du code engendré.

C'est dans cette optique qu'a été écrite cette thèse ; proposer une sémantique formelle pour un langage synchrone appelé Ls, proche de Lustre, ainsi que pour un langage assez bas niveau, appelé Obc, aisément compilable vers du C, de donner une compilation du premier langage dans le second, et enfin de démontrer que cette compilation respecte bien la sémantique des deux langages.

Ls est un langage qui a évolué à partir d'un autre langage qui servait initialement de noyau au langage Heptagon [23]. Heptagon est en essence une extension du langage Lustre. Ce langage contient des automates et gère efficacement les tableaux.

0.3 Structure de la thèse

Cette thèse se décompose en quatre parties.

La première partie expose l'architecture globale du compilateur.

Le premier chapitre présente la programmation synchrone avec le langage Lustre sur un exemple. L'exemple y subit toutes les passes de compilations dévelopées dans les parties suivantes. Les passes de compilations y sont expliquées informellement.

Le second chapitre introduit les notations et notions de bases (nœuds, horloges, ...), utilisées tout au long de la thèse.

Le troisième chapitre présente les différents sens que l'on peut attacher à l'expressions « préservation de la sémantique ». On y voit aussi quels outils et méthodes ont été choisis pour démontrer une préservation de sémantique.

La seconde partie présente la première phase de la compilation. Cette phase se déroule dans le monde flots de données.

Le chapitre quatre donne une description formelle des primitives flots de données.

Le chapitre cinq présente le premier langage de la chaîne de compilation, nommé Ls. Sa syntaxe y est donnée ainsi que sa sémantique formelle. Un exemple complet de programme y est donné, qui sera suivi tout au long de la thèse.

Le chapitre six présente Lsn, le premier langage intermédiaire. Il s'agit d'une version simplifiée de Ls, où les écritures dans la mémoire du nœud sont isolées. Sa syntaxe et sa sémantique y sont données.

Le chapitre sept présente la première passe de compilation de Ls vers Lsn. La relation attendue entre un nœud s dans Ls et son compilé t dans Lsn y est donnée, ainsi que l'énoncé de préservation de la sémantique. Les preuves figurent en annexe.

La troisième partie présente la seconde phase de la compilation. Cette phase quitte le monde flot de données pour le monde instantané, où on ne manipule plus de suites, mais des valeurs.

Le chapitre huit présente le premier langage instantané de la chaîne, il s'agit de Lsni. Ce langage est, comme Ls et Lsn, déclaratif. La seule vraie distinction avec Lsn est l'introduction d'instances de nœuds pour identifier les mémoires associées. Syntaxe et sémantique formelle y sont données.

Le chapitre neuf expose comment réinterpréter les sémantiques flot de données dans les sémantiques instantanées, et présente la compilation de Lsn à Lsni. Les preuves sont aussi données en annexe.

Le chapitre dix présente Obc, le langage final de la chaîne de compilation. Ce langage est adapté à la génération de code pour les architectures de Von Neumann traditionnelles. C'est un langage impératif, similaire au langage C.

Le chapitre onze présente la dernière passe de compilation, c'est-à-dire comment passer de Lsni à Obc. L'ordre des instructions y est donc fixé, et là encore les preuves sont données en annexe.

La quatrième partie est la conclusion de cette thèse. On y voit comment notre exemple pourrait être informellement compilé vers le langage C, et quelles sont les fonctionnalités qui manquent au langage Ls afin d'en faire un langage d'utilisation plus agréable.

Le chapitre douze étudie la compilation vers le langage C et s'intéresse à ce qui pourrait être fait afin de se brancher formellement au compilateur CompCert.

Le chapitre treize propose quelques extensions qui pourraient être ajoutées à Ls pour obtenir un langage de plus haut niveau, toujours doté d'une sémantique formelle.

Enfin le chapitre quatorze apporte une conclusion à cette thèse.

En annexes figurent quelques suppléments.

L'annexe A présente des optimisations qui ne sont pas implémentées dans la chaîne de compilation donnée, mais qui peuvent être assez facilement prouvée. Certaines optimisations sont même gratuites en termes de preuve car validées de la même façon que des programmes non optimisés.

L'annexe B regroupe tous les codes intermédiaires produits.

L'annexe C discute de certains choix techniques d'implémentation.

0.4 À propos des dessins

Cette thèse comporte des diagrammes colorés. Cependant, il se peut que la version consultée ait été imprimée en niveaux de gris. Si c'est le cas, il faut comprendre « vert » par « gris très clair », « bleu », par « gris un peu moins clair » et « rouge » par « gris encore moins clair ».

Figure 1: Étalonnage des couleurs.

Première partie

Architecture générale du compilateur

Chapitre 1

Exemple introductif

Ce chapitre se veut très informel. Il introduit un exemple qui sera utilisé tout au long de la thèse.

1.1 Un petit programme flot de données

Les langages flots de données tels que LUSTRE manipulent des suites. Dans ce chapitre, on va s'intéresser à un petit programme qui prend en entrée une suite de valeurs booléennes, et qui entrelace deux flots d'entiers partants de 0, l'un croissant, l'autre décroissant, selon la valeur du booléen.

1.1.1 Définition d'une suite récurrente

Une suite récurrente se définit en mathématiques par des valeurs de base, et une formule qui lie les valeurs précédentes à la valeur suivante. Un exemple classique est la suite des entiers impairs :

$$\begin{aligned} u_0 &= 1 \\ u_{n+1} &= u_n + 2 \end{aligned}$$

Il existe deux opérateurs sur les suites en LUSTRE qui permettent de construire une telle suite :

- l'opérateur pw [1] d'application point à point qui prend plusieurs suites $(u_n)_{n\in\mathbf{N}}$, $(v_n)_{n\in\mathbf{N}}$, $(w_n)_{n\in\mathbf{N}}$, etc. et une fonction sur des valeurs f et qui retourne une suite $(f\,\mathrm{pw}(u,v,w,etc.)_n)_{n\in\mathbf{N}}$ telle que $f\,\mathrm{pw}(u,v,w,etc.)_n = f(u_n,v_n,w_n,etc.)$ pour tout n. Par abus de notation, une telle suite est notée $f\,(u,v,w,etc.)$, voire même $u+v$, $u-v$, etc. lorsque f est l'opérateur arithmétique $+$, $-$, etc. . Si de plus f est une fonction constante, la suite sera simplement notée f.
- l'opérateur `->pre` [2] qui prend deux suites $(u_n)_{n\in\mathbf{N}}$ et $(v_n)_{n\in\mathbf{N}}$ et construit une nouvelle suite $((u\texttt{->pre}v)_n)_{n\in\mathbf{N}}$ telle que $(u\texttt{->pre}v)_0 = u_0$ et $(u\texttt{->pre}v)_{n+1} = v_n$ pour tout n.

1. pour « point wise » ; en réalité cet opérateur n'existe pas à proprement parler dans le langage, car systématiquement remplacé par l'abus de notation dont il est fait référence

2. pour «previous» ; en réalité, dans LUSTRE, cet opérateur se décompose en deux opérateurs, `->` et `pre`

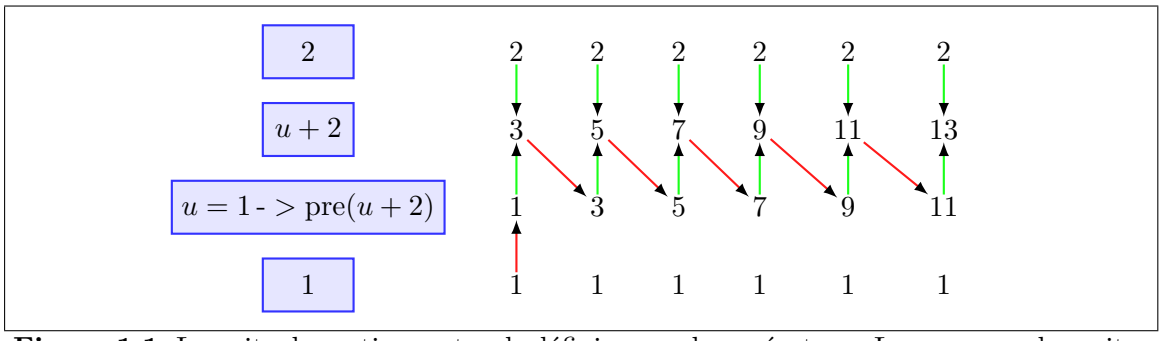


Figure 1.1: La suite des entiers naturels définie avec des opérateurs LUSTRE sur des suites

En utilisant ces deux opérateurs, on peut construire une relation caractéristique de la suite des entiers impairs :

$$u = (1\texttt{->pre}(u + 2))$$

En effet, on doit avoir, par définition de l'opérateur `->pre` les deux propriétés suivantes :

- $(1\texttt{->pre}(u + 2))_0 = 1_0$, donc $u_0 = 1$
- $(1\texttt{->pre}(u + 2))_{n+1} = (u + 2)_n$, donc $u_{n+1} = u_n + 2_n = u_n + 2$

Ce système correspond bien à la définition de la suite des entiers impairs. La figure 1.1 illustre ces manipulations sur les suites, avec en rouge le fonctionnement de l'opérateur `->pre` et en vert celui de $+$.

1.1.2 Échantillonnement de flots

Le second concept important en LUSTRE est celui de sous-suite. En mathématiques, une sous-suite $\left(u_{\phi(n)}\right)_{n\in\mathbf{N}}$ est la donnée d'une suite $(u_n)_{n\in\mathbf{N}}$ et d'une suite $(\phi_n)_{n\in\mathbf{N}}$ strictement croissante dans les entiers naturels ; on dit alors que la suite $(\phi_n)_{n\in\mathbf{N}}$ échantillonne $(u_n)_{n\in\mathbf{N}}$. En LUSTRE, une suite ϕ strictement croissante des entiers dans les entiers peut être représentée comme une suite de booléens $(\mathrm{ck}_{\phi_n})_{n\in\mathbf{N}}$ qui vérifie :

$$\forall n, m.\ \phi_n = m \Rightarrow \mathrm{ck}_{\phi_m} = \texttt{true}$$
$$\forall n, m.\ \phi_n < m < \phi_{n+1} \Rightarrow \mathrm{ck}_{\phi_m} = \texttt{false}$$

Une sous-suite $\left(u_{\phi(n)}\right)_{n\in\mathbf{N}}$ s'écrit alors (u `when` ck_ϕ) en LUSTRE.

Par exemple, si on s'intéresse aux valeurs d'une suite sur les entiers pairs, la suite $(\phi_n)_{n\in\mathbf{N}}$ est la suite qui vaut 0, 2, 4, 6, 8, etc., alors que $(\mathrm{ck}_{\phi_n})_{n\in\mathbf{N}}$ est la suite `true`, `false`, `true`, `false`, etc.

En LUSTRE, on peut ainsi définir des sous-suites d'une suite. La figure 1.2 illustre l'échantillonnage. Les flèches rouges indiquent les décalages de flots, alors que les flèches bleues sont les échantillonnages.

Dans le contexte

$$\begin{aligned} \phi_n &= 2n \\ u_0 &= 0 \\ u_{n+1} &= 1 + u_n \end{aligned}$$

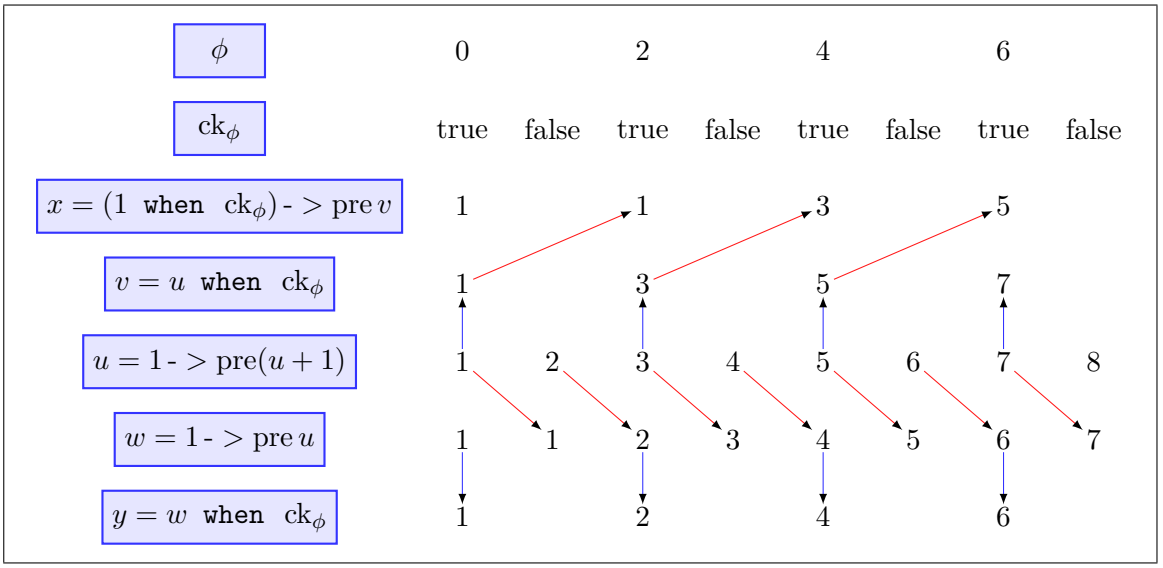


Figure 1.2: Exemple d'échantillonnage de suites

On remarque que les suites $(x_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{aligned} v_n &= u_{\phi_n} \\ x_0 &= 1 \\ x_{n+1} &= v_n \end{aligned}$$

et $(y_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{aligned} w_0 &= 1 \\ w_{n+1} &= u_n \\ y_n &= w_{\phi_n} \end{aligned}$$

sont distinctes, c'est à dire que les suites représentées par `(1 ->pre` u`) when` ck_ϕ et `(1 when` ck_ϕ `) ->pre (`u `when` ck_ϕ`)` sont généralement différentes.

Comme ces définitions ne définissent pas les valeurs en tout point de la suite, on a besoin d'un autre opérateur qui permet de remplir ces trous. En Lustre, cet opérateur est l'opérateur `current` ; il remplace les trous crées par le dernier échantillonnage par la dernière valeur calculée, à l'exception des premiers qui restent indéfinis. La figure 1.3 présente le comportement de `current`, avec les valeurs indéfinies représentées par *nil*.

1.1.3 Un petit programme flot de données

Nous pouvons à présent écrire notre premier (et dernier) programme en Lustre (version 4) [53].

Un programme en Lustre se présente comme un système d'équations paramétrées par des suites et définissant des suites. Parmis les suites du systèmes, certaines doivent être retournées afin de pouvoir exploiter la solution du système trouvé. Toutes les autres suites du système sont alors inaccessibles de l'extérieur.

Ces équations doivent respecter les propriétés suivantes :

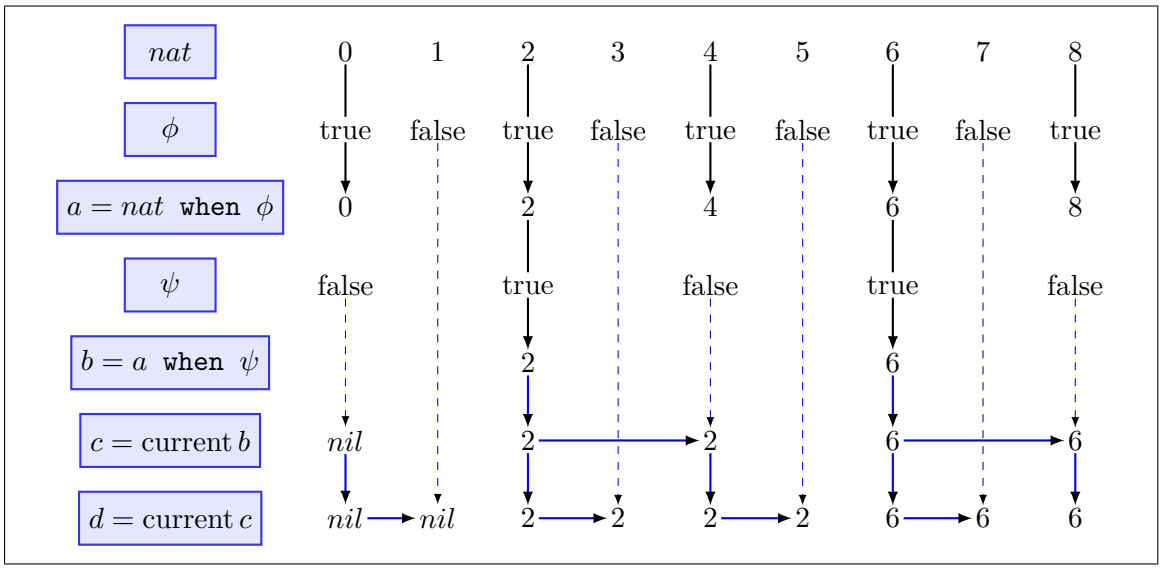


Figure 1.3: L'opérateur current

```
node deltas (sign : bool) returns (delta : int);
var     nsign:bool;
        dp:int when  sign;
        dn:int when nsign;
let     dp = (0 when  sign) ->pre ((delta+1) when  sign);
        dn = (0 when nsign) ->pre ((delta-1) when nsign);
        delta = if sign then (current dp) else (current dn);
        nsign = not sign;
tel;
```

Figure 1.4: Un petit programme LUSTRE

Cloture Toute suite utilisée dans le système doit être définie par une équation du système à l'exception des paramètres du système.

Concision Toute suite définie dans le système ne peut l'être qu'au plus une fois.

Causalité Tout terme de suite ne peut se calculer à partir de lui même.

La propriété de cloture garantit que tout est sous contrôle et qu'on ne dépend pas d'un facteur extérieur au programme en dehors de ses paramètres.

La propriété de concision n'apporte rien en soi, mais si on la retirait, il faudrait être en mesure de pouvoir décider si les deux équations définissant bien la même suite, ce qui est trop complexe à réaliser, en plus d'être inutile en pratique.

La propriété de causalité assure existence et unicitié d'une solution pour un système clos [3]. Elle exclut des équations de la forme $u_n = u_n$, $u_n = u_n + 1$ ou des systèmes comme $u_n = v_n + 1 \ \wedge \ v_n = u_n - 1$, mais autorise des équations de la forme $u_n = 0\texttt{->pre}(1 + u_n)$.

Ce programme gère deux compteurs, l'un croissant et l'autre décroissant sur deux

3. En supposant que les fonctions utilisées soient toujours bien définies pour toute valeur, afin d'éviter par exemples des problèmes de division par zéro.

```
node deltas (sign:bool) returns (delta:int32);
var dp:int32 when sign;
    dn:int32 when not sign;
let dp = 0 fby ((delta+1) when sign);
    dn = 0 fby ((delta-1) when not sign);
    delta = merge sign (True -> dp) (False -> dn);
tel;
```

Figure 1.5: Un petit programme Ls

sous-suites complémentaires.

entrée	sign	true	true	false	true	false	false	true
	dp	0	1		2			3
	dn			0		−1	−2	
sortie	delta	0	1	0	2	−1	−2	3

Dans le langage Ls qui est le langage source de notre chaîne de compilation, l'opérateur `->pre` sera remplacé par un l'opérateur `fby` qui a le même comportement, mais dont l'argument gauche doit impérativement être une constante, et non une suite. L'opérateur `current` est remplacé par un opérateur `merge` qui prend en entrée deux suites complémentaires, c'est-à-dire ne pouvant pas simultanément définir une valeur au même indice, et retourne une suite qui remplit les trous de l'une avec la valeur de l'autre. Ceci évite, d'une part de se retrouver avec des valeurs indéfinies, et d'autre part de devoir mémoriser la valeur précédente. La traduction naturelle du programme Lustre dans Ls donne le programme défini en figure 1.5.

1.2 Compilation d'un petit programme

Le processus de compilation a deux problèmes à traiter. Le premier est de réussir à ne plus avoir de notion de suite afin de ne manipuler que des données que la machine sait utiliser, c'est-à-dire afin de ne traiter que des valeurs simples. Le second est comment organiser les calculs pour les rendre effectifs.

1.2.1 Passer des suites de valeurs aux valeurs simples

Un premier travail consiste à simplifier les équations en jeu, pour que le compilateur puisse traiter par la suite une seule information à la fois.

Pour cela, on va introduire de nouvelles variables, comme montré en figure 1.6.

Bien que ce soit à première vue évident quand on connaît la sémantique du langage, il faut encore se persuader que ce nouveau programme définit bien la même suite `delta` que l'ancien. C'est la partie II de cette thèse. Ici, il suffit d'utiliser un théorème de référence transparentielle ; en remplaçant toutes les nouvelles suites introduites par leur définition, on retrouve à la virgule près le programme d'origine, et le théorème permet de conclure.

```
node deltas (sign:bool) returns (delta:int32);
var dp_pre0, dn_pre0:int32;
    dp, dp_pre:int32 when sign;
    dn, dn_pre:int32 when not sign;
let dp_pre0 = delta+1;
    dp_pre = dp_pre0 when sign;
    dn_pre0 = delta-1;
    dn_pre = dn_pre0 when not sign;
    dp = 0 fby dp_pre;
    dn = 0 fby dn_pre;
    delta = merge sign (True -> dp) (False -> dn);
tel;
```

Figure 1.6: Un petit programme Ls après introduction de variables

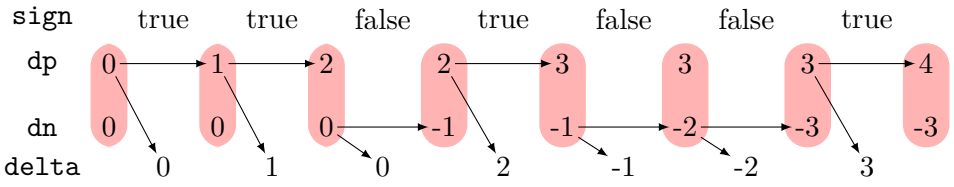


Figure 1.7: Calcul des suites en utilisant des mémoires

Après cette étape, on peut facilement identifier quelles valeurs sont passées entre les suites, il s'agit des variables `dp` et `dn`. Ces deux valeurs auront un statut particulier, celui de mémoires du programme. Une façon d'éliminer les suites pour se focaliser sur les valeurs est d'appeler le programme avec une valeur de la suite et une mémoire reflétant les calculs effectués jusqu'à présent, et de récupérer la sortie correspondante, ainsi que la nouvelle mémoire.

La figure 1.7 illustre cette idée ; on voit en particulier que le calcul de la mémoire suivante ne se fait qu'à partir de la mémoire précédente et de l'entrée. Il n'y a jamais accès ni au futur ni à des mémoires calculées plusieurs étapes auparavant.

1.2.2 Expliquer comment calculer

Maintenant, le compilateur doit transformer le programme pour expliquer à la machine comment calculer. Comme nous avons déjà simplifié les équations et qu'elles sont toutes de la forme $x = f(...)$ où f est une fonction des valeurs à calculer, il suffit d'effectuer un tri topologique des équations en suivant les dépendances de données.

Cependant certaines valeurs peuvent en apparence dépendre d'elles mêmes. C'est notamment le cas de `dn` qui dépend de `dn_pre` qui dépend de `dn_pre0` qui dépend de `delta` qui dépend de `dn`. Ce problème est résolu en remarquant que `dn` est une mémoire, et donc que sa valeur précédente est déjà connue. On peut donc la diviser en deux versions, `dn` qui est la valeur avant de commencer l'itération et `dn'` qui est la valeur à la fin de l'itération.

On part donc du principe selon lequel `delta`, `dp` et `dn` sont déjà définis. La seule équation alors bien définie est `delta = merge sign (True -> dp) (False -> dn);`. Ce

```
node deltas (sign:bool) returns (delta:int32);
var dp_pre0, dn_pre0:int32;
    dp, dp_pre:int32 when sign;
    dn, dn_pre:int32 when not sign;
let
    delta = merge sign (True -> dp) (False -> dn);
    dp_pre0 = delta+1;
    dn_pre0 = delta-1;
    dp_pre = dp_pre0 when sign;
    dp = 0 fby dp_pre;
    dn_pre = dn_pre0 when not sign;
    dn = 0 fby dn_pre;
tel;
```

Figure 1.8: Un petit programme pas encore impératif...

sera donc la première instruction à exécuter.

Maintenant, `delta` est aussi défini, et les équations `dp_pre0 = delta+1;` ainsi que `dn_pre0 = delta-1;` deviennent aussi bien définies et peuvent être exécutées.

Une fois les variables `dp` et `dn` devenues inutiles, on peut les écraser avec les nouvelles valeurs de $\texttt{dp}'$ et $\texttt{dn}'$. On peut donc ajouter vers la fin du programme les équations $\texttt{dp}' = 0 \texttt{ fby dp\_pre}$ et $\texttt{dn}' = 0 \texttt{ fby dn\_pre}$.

Il n'y a pas une unique façon d'ordonner les équations. Le programme donné en figure 1.8 correspond à l'un des multiples ordonnancements possibles.

1.2.3 Traduction vers du code impératif

Le code obtenu n'est pas encore tout à fait impératif. En effet, les opérateurs `merge` et `when` n'ont pas d'équivalent dans la plupart des langages de programmation impérative. ; il faut encore les encoder dans des structures de contrôle.

De plus, la compilation doit produire une procédure qui attend des entrées et une mémoire qu'elle doit modifier à chaque itération. Il est donc nécessaire d'identifier précisément quelle est la mémoire nécessaire et comment l'initialiser.

La compilation doit donc produire une mémoire initiale et une procédure d'itération, que l'on regroupe dans un objet.

Le langage vers lequel est compilé notre langage synchrone s'appelle Obc.

Le code brut après traduction est donné en figure 1.9.

Ce code est séquentiel, et la compilation peut s'arrêter ici. Cependant le code est particulièrement inefficace, et on peut encore lui appliquer des passes d'optimisation pour obtenir un code plus léger et rapide. En particulier, on peut regrouper les `switch` entre eux et faire de la propagation de constantes pour obtenir le code en figure 1.10.

La chaîne de compilation est résumée en figure 1.11.

```
class deltas {
  memory {
    dp : int32 = 0;
    dn : int32 = 0;
  }
  step (sign:bool) returns (delta:int32) {
    dp_pre0, dn_pre0 : int32;
    dp_pre : int32;
    dn_pre : int32;
    --
    switch(sign) {
      True -> { delta = mem.dp;
              }
      False -> { delta = mem.dn;
               }
    }
    dp_pre0 = delta+1;
    dn_pre0 = delta-1;
    switch(sign) { True -> { dp_pre = dp_pre0; } }
    switch(sign) { True -> { mem.dp = dp_pre; } }
    switch(sign) { False -> { dn_pre = dn_pre0; } }
    switch(sign) { False -> { mem.dn = dn_pre; } }
  }
}
```

Figure 1.9: ... sa version impérative...

```
class deltas {
  memory {
    dp : int32 = 0;
    dn : int32 = 0;
  }
  step (sign:bool) returns (delta:int32) {
    switch(sign) {
      True -> { delta = mem.dp;
                mem.dp = delta+1;
              }
      False -> { delta = mem.dn;
                 mem.dn = delta-1;
               }
    }
  }
}
```

Figure 1.10: ... et sa version optimisée.

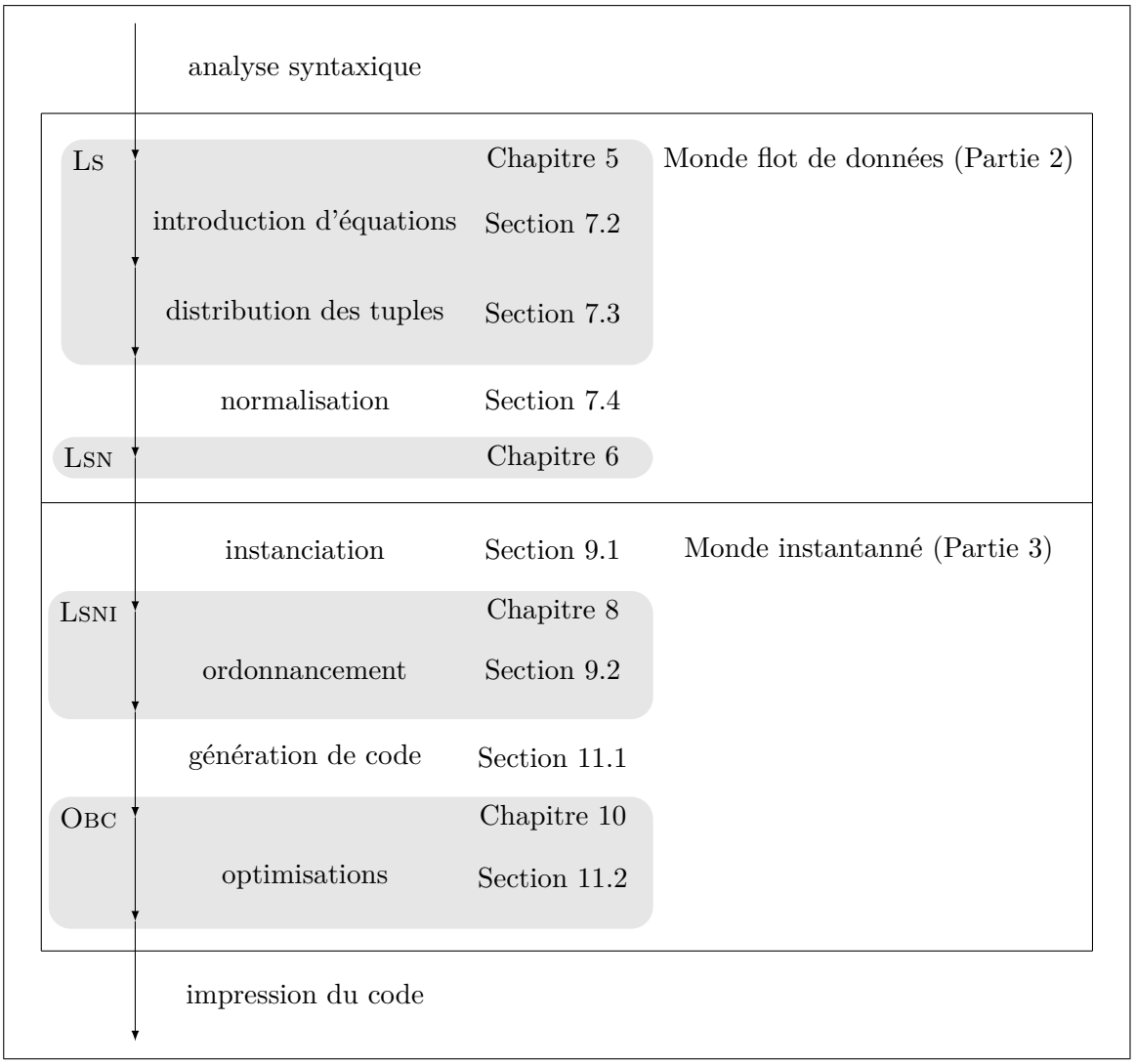


Figure 1.11: Chaîne de compilation de Ls à Obc

```c
#include "deltas.h"
#include <unistd.h>
#include <stdio.h>

deltas_t instance; // la memoire du programme

int main () {
  int32_t input;
  int32_t output;
  deltas_reset(&instance); // initialisation
  while(1) {
    scanf("%d", &input); // recuperation des entrees
    deltas_step(&instance, input, &output); // iteration
    printf("%d\n", output); // utilisation de la sortie
  }
}
```

Figure 1.12: Utilisation du programme traduit au sein d'un programme exécutable.

1.2.4 Compilation C

La certification s'arrête ici. Certifier formellement vers un langage usuel, tel que C ou Java, impliquerait la définition d'une sémantique formelle de ce langage, ainsi que la formalisation de ce que l'on entendrait par préservation de la sémantique par compilation.

En effet, la compilation présentée ne produit pas un code exécutable, mais une bibliothèque de fonctions à appeler dans une boucle. Or certains langages compilables, comme le sous ensemble de C utilisé dans CompCert [36] ont bien été formalisés, mais donnent une sémantique non pas à une bibliothèque de fonctions, mais à la fonction principale d'un programme.

De plus, il est préférable de s'arrêter dans un langage qui puisse être facilement traduit dans d'autres, ce qui est le cas de Obc.

Une traduction possible vers C est tout de même donnée dans les figures 1.13 et 1.14.

La mémoire du programme se traduit en une structure contenant toutes les valeurs à passer entre appels (ici `dp` et `dn`), ainsi qu'une fonction d'initialisation de cette mémoire (qui, ici, met à 0 ces deux valeurs).

La fonction d'itération prend en arguments une référence vers une mémoire, les entrées du programme (ici `sign`) et des références vers les sorties (ici `delta`).

Pour utiliser le programme généré, il faut alors commencer par réserver une mémoire et l'initialiser. Ensuite, il faut placer la fonction d'itération dans une boucle qui récupère la suite des valeurs d'entrées et calcule les valeurs de sorties pour les utiliser, comme en figure 1.12.

À titre de comparaison, voici aussi les versions compilées à partir du compilateur Lustre (version 4), et réédité à la main pour le rendre plus lisible et lui donner une interface comparable.

On y voit en particulier que la mémoire doit contenir une information supplémentaire d'initialisation. Avoir une expression à gauche de `->pre` au lieu d'une constante à gauche

```
#ifndef __DELTAS_H__
#define __DELTAS_H__

#include <stdint.h>

// DELTAS
typedef struct {
  int32_t dp;
  int32_t dn;
} deltas_t;
void deltas_reset (deltas_t*);
void deltas_step (deltas_t*, int32_t, int32_t*);

#endif
```

Figure 1.13: Interface du programme traduit.

```
#include "deltas.h"

// DELTAS
void deltas_step (deltas_t* this, int32_t sign, int32_t* delta) {
  if(sign) {
    *delta = this->dp;
    this->dp = *delta + 1;
  }
  else {
    *delta = this->dn;
    this->dn = *delta - 1;
  }
}

void deltas_reset (deltas_t* this) {
  this->dp = 0;
  this->dn = 0;
}
```

Figure 1.14: Implémentation du programme traduit.

```
#ifndef __DELTAS_H__
#define __DELTAS_H__

#include <stdint.h>

// DELTAS
typedef struct {
  int32_t dp;
  int32_t dn;
  int32_t dn_init; // as boolean
  int32_t dp_init; // as boolean
} deltas_t;
void deltas_reset (deltas_t*);
void deltas_step (deltas_t*, int32_t, int32_t*);

#endif
```

Figure 1.15: Interface du programme traduit (version LUSTRE).

de `fby` oblige calculer la valeur initiale au bon moment, et donc à tester si on est dans l'instant initial.

```
#include "deltas.h"

// DELTAS
void deltas_step (deltas_t* this, int32_t sign, int32_t* delta) {
  int dp_;
  int dn_;
  if (this->dn_init) dn_ = 0;
  else               dn_ = this->dn;
  if (this->dp_init) dp_ = 0;
  else               dp_ = this->dp;
  if (sign) {
    *delta = dp_;
    this->dp = *delta+1;
  }
  else {
    *delta = dn_;
    this->dn = *delta-1;
  }
  this->dn_init = this->dn_init && sign;
  this->dp_init = this->dp_init && !sign;
}

void deltas_reset (deltas_t* this) {
  this->dp_init = 1; // true
  this->dn_init = 1; // true
}
```

Figure 1.16: Implémentation du programme traduit (version LUSTRE).

Chapitre 2

Notions de base

Avant d'aborder les différents langages et leur sémantique, nous allons présenter les notions communes à tous nos langages.

2.1 Notations mathématiques

Cette thèse repose, comme beaucoup de théories classiques de mathématiques, sur la théorie des ensembles de Zermelo-Fraenkel.

Dans cette théorie, la plupart des symboles sont courants et les notations ne varient pas d'une source à l'autre. C'est le cas des symboles $\cup$, $\cap$, $\subset$, $\times$ qui désignent respectivement l'union, l'intersection, l'inclusion et le produit de deux ensembles.

Cependant, d'autres symboles sont moins courants et les notations utilisées varient d'une source à l'autre. C'est le cas des symboles suivants :

- l'union disjointe de deux ensembles (parfois appelée coproduit de deux ensembles) $\mathcal{A}$ et $\mathcal{B}$ est notée $\mathcal{A} + \mathcal{B}$ en référence aux notations utilisées en théorie des types et en théorie des catégories. D'autres sources utilisent la notation $\mathcal{A} \sqcup \mathcal{B}$.
- l'ensemble des fonctions de $\mathcal{A}$ dans $\mathcal{B}$ est noté $\mathcal{A} \to \mathcal{B}$ également en référence à la théorie des types. En théorie des ensembles, cet ensemble est souvent noté $\mathcal{B}^{\mathcal{A}}$.
- l'ensemble des parties d'un ensemble $\mathcal{A}$ est noté $\mathfrak{P}(\mathcal{A})$. D'autres sources utilisent $\mathcal{P}(\mathcal{A})$, $\wp(\mathcal{A})$ ou encore $\mathbf{2}^{\mathcal{A}}$ du fait d'une identification entre les fonctions de $\mathcal{A}$ dans un ensemble à deux éléments $\mathbf{2}$ et les parties de $\mathcal{A}$.
- étant donnés un ensemble $\mathcal{A}$ et une propriété P sur les éléments de $\mathcal{A}$, on note $\{x{\in}\mathcal{A} \mid P(x)\}$, $\{x : \mathcal{A} \mid P(x)\}$ ou encore $\{x \mid P(x)\}$ lorsque l'ensemble $\mathcal{A}$ est implicite, l'ensemble des éléments de $\mathcal{A}$ qui vérifient la propriété P.

Certaines formules pouvant être longues, ou contenir des parenthèses imbriquées les rendant difficiles à lire, des « blocs » de propositions pourront être constitués, ainsi :

$$(A \vee B \vee (C \Rightarrow (D \wedge E \wedge F))) \Rightarrow ((G \vee H \vee I) \wedge J)$$

pourra être écrit :

$$\left(\bigvee \left\{ \begin{array}{l} A \\ B \\ \\ C \Rightarrow \bigwedge \left\{ \begin{array}{l} D \\ E \\ F \end{array} \right. \end{array} \right. \right) \Rightarrow \bigwedge \left\{ \begin{array}{l} G \vee H \vee I \\ J \end{array} \right.$$

2.2 Types, constantes, opérateurs et interfaces

2.2.1 Interfaces

Tous les langages que nous considérons doivent pouvoir s'interfacer avec un autre langage afin de pouvoir importer des fonctions et types déjà présents.

Pour ce, on demande un environnement initial de types qui peuvent être abstraits ou définis par une énumération finie de valeurs, et de symboles de fonctions non interprétés, c'est-à-dire dont le compilateur ne connaît que le nom et le type de ses paramètres et de sa valeur de retour.

2.2.2 Types

Dans le temps réel critique, de nombreuses restrictions apparaissent dans les langages utilisés du fait des fortes contraintes que l'on s'impose afin de garantir une exécution sans erreurs.

En particulier, on interdit l'allocation de mémoire dynamique, qui peut échouer ; et donc les seuls types autorisés sont les types dont on connaît la place en occupation mémoire, ce qui exclut les chaînes de caractères, les listes, les arbres, etc.

Par simplicité, les seuls types considérés dans cette thèse seront les types énumérés, et les types abstraits qui permettent de relâcher un peu les contraintes que l'on s'impose sur les types. De plus on interdit à une valeur d'appartenir à deux types différents. On n'a donc pas en particulier de type natif pour les tableaux mêmes non modifiables (on peut cependant déclarer un type abstrait pour ces tableaux), ni la possibilité de donner une même valeur par défaut à tous les types.

Les types utilisés ne sont pas « compilés », dans le sens où leur définition ne subit pas de modification au cours de la traduction de Ls vers Obc. Cela permet de pouvoir comparer les données manipulées dans le code source avec les données produites dans le code final. Comme cet environnement de types est commun à toute la chaîne, il sera toujours implicite par la suite et au besoin, on pourra y faire référence par T.

Par la suite, les types seront notés τ_i où i est un indice permettant de distinguer les différents types d'une même expression.

Les types énumérés

Les types énumérés sont assez peu utiles dans des langages tels que C, en effet, ce ne sont que des entiers, et la macro `#define` pourrait très bien suffire. En revanche, dès

que l'on veut vérifier certaines conditions de typage, et s'assurer de l'exhaustivité des traitements par cas, on a besoin d'un type spécifique.

La plupart des langages synchrones, qu'il soient ou non flot de données disposent d'un système de déclarations de types énumérés, notamment afin d'encoder des modes qui explicitent sous quelles conditions des variables doivent être actives, et quelles opérations effectuer.

Pour des questions de typage, on interdit d'avoir deux constructeurs de même nom, de même pour les types. En particulier, on ne peut pas écraser un type déjà défini. On demande également à tout type énuméré d'avoir au moins un constructeur.

Par la suite K dénote l'ensemble de tous les constructeurs ; et étant donné un constructeur k, on note $\mathrm{Ktype}\,(k)$ le type de ce constructeur.

Définition 2.2.1 (Type unité)**.** *On définit le type unité, noté* `unit` *comme étant le type énuméré de seul constructeur* `Only`*.*

Les types abstraits

Quand on veut interfacer un langage de programmation avec d'autres, on a besoin de pouvoir traduire les structures de données d'un langage vers un autre. Cependant certains types peuvent ne pas avoir d'équivalent, on peut alors vouloir informer le compilateur qu'un tel type existe et qu'il n'a pas à se préoccuper de sa structure.

On peut aussi vouloir disposer d'une approche modulaire et pouvoir manipuler ses données comme on l'entend, sans avoir à connaître la structure des données, ce qui peut permettre au programmeur de réorganiser son type sans avoir à réécrire tout son code.

Dans ces deux situations on peut utiliser des types abstraits . Cependant, dans le cas de l'interfaçage avec un autre langage, il faut s'assurer que les données ne puissent être modifiées par les fonctions de l'interface par effet de bord, sans quoi, la sémantique du programme n'est plus déterministe, et la transparence référentielle est perdue.

Comme on ne connaît pas la taille d'un type abstrait, il faut garder à l'esprit, qu'on ne voudra pas le stocker dans un nœud ; une valeur abstraite ne peut survivre plus d'un instant, ce qui limite leur champ d'application.

2.2.3 Valeurs, valeurs non étendues et valeurs étendues

Une valeur non étendue est un habitant d'un type du langage. Pour un type énuméré, c'est donc un constructeur ; et pour un type abstrait, c'est une référence.

Le type de la valeur non étendue α est noté $\mathrm{Ttype}\,(\alpha)$.

Une valeur étendue est soit une valeur non étendue, soit une absence de valeur non étendue. Ainsi, quand un mode est actif, ses valeurs étendues associées sont toutes présentes (non étendues), alors que lorsqu'il est inactif, elles sont absentes. On peut ainsi garantir que rien n'est calculé dans les modes inactifs.

Notation 2.2.1 (valeurs étendue)**.** *On note abs la valeur étendue absente et α la valeur étendue présente qui correspond à la valeur non étendue α.*

Une valeur est une valeur étendue ou une valeur non étendue selon le contexte.

Par convention, les valeurs non étendues utilisent la lettre grecque α, et les valeurs étendues utilisent la lettre grecque β.

Notation 2.2.2 (valeurs d'un type)**.** *On note $[\![\tau]\!]$ le domaine des valeurs non étendues du type τ, et $[\![\tau]\!]^{abs}$ celui des valeurs étendues.*

2.2.4 Flots

Comme les langages de début de chaîne de compilation sont des langages flots de données, il faut étendre ces notions à celles de flots.

Notation 2.2.3 (monoïdes)**.** *Étant donné un ensemble E, on note E^* le monoïde libre sur E, ϵ dénote son élément neutre (le mot vide), $e_1 \circ e_2$ le produit de e_1 et e_2 (la concaténation) et $[x]$ le plongement de x dans E^* (le mot x), $e\,\langle i \rangle$ dénote la $i$ème lettre du mot e, et enfin $x_1, ..., x_n$ dénote le mot formé par les lettres x_1 à x_n (c'est-à-dire $[x_1] \circ ... \circ [x_n]$). L'ajout d'une valeur en fin se note $e \cdot x$ et se définit comme $e \circ [x]$.*

Dans la littérature, il existe plusieurs notions de flots, la plupart acceptent la notion de flot fini et certains n'acceptent que les flots infinis.

Dans cette thèse, seuls les flots finis (mais de taille arbitraires) sont considérés pour des raisons techniques liées à des lourdeurs sur l'utilisation des types coinductifs de Coq.

Les conventions de nommage pour les flots de valeurs non étendues est ϕ_i ; alors que pour les valeurs étendues, on utilisera ψ_i.

On notera que $\psi\,\langle i \rangle$ peut tout aussi bien être une valeur absente, qu'une valeur présente ; on ne « saute » pas les absences.

Remarque 2.2.1 (Représentation des flots)**.** *Il existe principalement deux façons de représenter un flot par une liste.*

La première façon consiste à considérer que la première valeur accessible d'une liste est la plus vieille. Par exemple, en Ocaml*, la liste* `0::(1::(2::(3::[])))` *pourrait représenter le début de la suite des entiers naturels. La fin de liste* (`[]`) *n'en est pas vraiment une, car on pourrait vouloir l'étendre (en* `(4::(5::[]))` *par exemple).*

La seconde consiste à considérer que la première valeur la plus accessible d'une liste est la plus récente. C'est cette convention qui sera utilisée, la même liste sera donc représentée par $((\epsilon \cdot 1) \cdot 2) \cdot 3$. Le constructeur de début de liste (ϵ) est un vrai début de liste cette fois ci.

Dans ces deux conventions, la valeur la plus vieille est toujours la plus à gauche, mais sa profondeur dépend de la convention adoptée.

L'un des avantages de cette seconde convention est qu'on peut étendre plus facilement un flot en lui ajoutant une valeur récente, alors que dans la première, il faut s'enfoncer tout au bout dans la liste pour ajouter une valeur récente.

Un autre avantage concerne la définition et l'utilisation de l'itérateur sur les fonctions de flots. En Ocaml*, on aurait les définitions suivantes :*

```
let iter1 (f : 'a -> 'etat -> 'etat) : 'etat -> 'a list -> 'etat =
    let rec iter acc al =
        match al with
        | [] -> acc
        | a :: al -> iter (f a acc) al
    in iter

type 'a tsil = Lin | Snoc of 'a tsil * 'a (* type des listes inversees *)

let iter2 (f : 'a -> 'etat -> 'etat) (base : 'etat) : 'a tsil -> 'etat =
    let rec iter al =
        match al with
        | Lin -> base
        | Snoc (al, a) -> f a (iter al)
    in iter
```

`iter1` (resp. `iter2`) existe déjà dans la bibliothèque standard de OCaml *sous le nom `fold_left` (resp. `fold_right`) modulo l'ordre dans lequel sont passés les arguments.*

La version `iter2` est plus simple puisque la fonction récursive `iter` qu'elle utilise en interne n'a qu'un seul argument, qui est la liste en cours. Elle est d'ailleurs associée au principe d'induction naturel sur les listes :

$$((\forall x, l.\ P(l) \Rightarrow P(l \cdot x)) \wedge P(\epsilon)) \Rightarrow \forall l.\ P(l)$$

La version `iter1` est moins agréable à utiliser puisque la récursion interne utilise deux arguments, la liste et un état souvent appelé accumulateur. Elle est associée naturellement associée au principe de réduction suivant :

$$(\forall x, l.\ P(x :: l) \Rightarrow P(l)) \Rightarrow \forall l.\ P(l) \Rightarrow P([])$$

2.2.5 Produit de types et signatures

Un produit de types, que nous noterons π par convention, est un élément du monoïde libre engendré par les types (T^*).

Notation 2.2.4 (domaine d'un produit de types)**.** *$[\![\pi]\!]$ (resp. $[\![\pi]\!]^{abs}$) dénote le domaine associé au produit de types via le morphisme $\tau_1, ..., \tau_n \to [\![\tau_1]\!], ..., [\![\tau_n]\!]$ (resp. $\tau_1, ..., \tau_n \to [\![\tau_1]\!]^{abs}, ..., [\![\tau_n]\!]^{abs}$) .*

Une signature, que nous noterons σ par convention, est un couple d'un produit de types π_{ent} et d'un type τ_{sort}. La première composante est appelée le produit de types des entrées de σ alors que la seconde est appelée le type de sortie de σ.

Notation 2.2.5 (signatures)**.** *Étant donnés un produit de types π_{ent} et un type τ_{sort}, on note $\pi_{ent} \to \tau_{sort}$ la signature σ de types des entrées π_{ent} et de type de sortie τ_{sort}.*

2.2.6 Opérateurs

Pour s'interfacer avec d'autres langages, on peut déclarer des fonctions qui permettent de manipuler les types abstraits.

Pour que la sémantique des programmes utilisant ces fonctions soit bien définie, il est impératif que l'implémentation associée soit sans effet de bord, c'est-à-dire définisse une fonction au sens mathématique de ses entrées vers ses sorties, ne dépendant pas d'un état interne.

Si une telle fonction a des effets de bords non critiques, c'est-à-dire contient des accès ou des modifications d'un état interne qui n'est utilisé que par des composants non critiques du système et qui n'affecte pas les sorties produites indépendamment des entrées, alors la sémantique des nœuds reste bien définie, même si le comportement global du système peut avoir un comportement surprenant bien que non critique.

Cette thèse ne s'intéresse qu'au comportement des nœuds et non au comportement du système, on supposera donc par la suite que toutes ces fonctions sont sans effet de bord.

Par la suite, ces fonctions seront appelées opérateurs, car elles ne renvoient qu'une unique valeur en résultat.

Là encore, on interdit aux déclarations d'écraser une déclaration précédente.

Comme pour les types, les opérateurs sont déclarés en début de chaîne de compilation. L'ensemble des opérateurs est noté O et sera implicite par la suite.

Par convention les opérateurs seront notés o_i.

À chaque opérateur o est associé une signature $O_{sig}(o) = \pi_{ent} \to \tau_{sort}$ (qui correspond à celle indiquée dans la syntaxe concrète), et une sémantique $O_{sem}(o)$, qui est une fonction de $[\![\pi_{ent}]\!]$ dans $[\![\tau_{sort}]\!]$.

2.3 Nœuds et sémantiques

Nous avons vu pour l'instant deux composantes statiques de l'environnement global, il s'agit de T et O. Il existe une troisième composante qui elle est dynamique, et décrit les nœuds que l'on construit.

Cette composante ne sera pas implicite, en effet, dans la sémantique on aura besoin de l'étendre ; cette composante sera notée N et ses éléments seront notés n_i.

Par simplification, on impose que la signature des entrées soit un type simple, plutôt qu'un produit de types. Les nœuds n'ont donc qu'une seule entrée et une seule sortie. Si on veut définir un nœud à plusieurs entrées et plusieurs sorties, il faut passer par un type abstrait, avec un opérateur qui combine plusieurs entrées en une seule et autant d'opérateurs de projections qu'il y a de composantes dans le tuple.

Comme pour les opérateurs, chaque nœud n dispose d'une signature $N_{sig}(n) = \tau_{ent} \to \tau_{sort}$ et d'une sémantique qui est différente de celle des opérateurs.

En effet, cette sémantique se décline en deux variantes.

- La sémantique flots de données $N^{\mathcal{F}}_{sem}(n)$ de n est un élément de $\mathfrak{P}([\![\tau_{ent}]\!] \times [\![\tau_{sort}]\!])$.
- La sémantique instantannée $N^{\mathcal{I}}_{sem}(n)$ de n, qui nécessite une composante supplémentaire appelée la mémoire du nœud, et notée $N_{mem}(n)$. La sémantique instantanée est alors un élément de $[\![\pi]\!] \times \mathfrak{P}(([\![\tau_{ent}]\!] \times [\![\pi]\!]) \times ([\![\tau_{sort}]\!] \times [\![\pi]\!]))$ où π est la mémoire du nœud ($N_{mem}(n)$). Le premier élément du couple est appelé mémoire initiale du

nœud et noté $N^{\mathcal{I}}_{init}(n)$, l'autre est la relation d'itération et noté $N^{\mathcal{I}}_{iter}(n)$.

On vera dans le chapitre suivant qu'on demande aussi une condition sur ces sémantiques qui permette de les unifier.

2.4 Environnement local et horloges

Un langage synchrone déclaratif nomme les primitives qu'il considère ; ce nommage et les primitives associées forment son environnement local.

Dans les dernières passes de compilation, les langages ne manipuleront plus des flots de données, afin de mieux correspondre au paradigme impératif. De tels langages seront qualifiés d'instantanés par opposition à flot de données. Dans ces langages instantanés, ces primitives sont les valeurs étendues.

Dans les langages flots de données, ces primitives sont des flots de valeurs étendues qui doivent être de même longueur. On peut alors transposer un environnement de flots en une suite d'environnements instantanés et réciproquement.

Comme dans ML, on ne peut pas parler de variables au sens de la programmation impérative, car les primitives associées ne sont pas amenées à être modifiées ; les équations posées ont un sens mathématique et ne sont pas des affectations.

En plus de la primitive associée, l'environnement associe à chaque nom un type, qui est le type de toutes ses valeurs, et une expression symbolique qui énonce sous quelles conditions une valeur est présente ou absente, cette expressions est appelée l'horloge du flot.

En Lustre, une horloge est soit `when x`, soit `base` où `x` est un nom de flot booléen ; et la condition de présence d'une valeur en un instant i est que la valeur du flot `x` à l'instant i doit être présente et de valeur `true`.

`y`	*true*	*true*	*false*	*false*	*false*	*true*
`x`	0	1	2	3	4	5
`x when y`	0	1				5

Définition 2.4.1. *Horloge*

Dans les langages utilisés dans cette thèse, une horloge est un couple formé d'un nom de flot x et d'une constante k, et est notée `when` $k(x)$.

On a la correspondance suivante entre les horloges de Lustre et celles utilisées dans cette thèse :

Lustre	Horloges définies dans cette thèse
`when b`	`when True(b)`
`when x=A`	`when A(x)`
horloge de base	`when Only(base)`

`base` est ici un nom de flot spécial, de type `unit`, ce nom de flot sera défini au chapitre 5.

L'ensemble des horloges est noté CK par la suite.

Pour l'encodage de l'horloge de base, on impose dans notre environnement local l'existence d'un flot nommé `base`, de type `unit` et dont l'horloge est `on Only(base)`.

Cet encodage permet d'éviter de distinguer les deux cas qui peuvent se présenter.

Dans les langages de notre chaîne de compilation, les horloges ne sont pas plus expressives, mais on s'autorise à considérer plus de deux états. en Lustre, une horloge est simplement une expression booléenne ; dans les langages que nous considérons, une horloge est un couple formé d'un nom de flot et d'une constante. Une horloge Lustre active les instants où son expression s'évalue à `true` ; les horloges que nous considérons active les instants où le flot nommé s'évalue à la constante indiquée.

`y`	*A*	*A*	*C*	*B*	*A*	*B*
`x`	0	1	2	3	4	5
`x when A(y)`	0	1			4	
`x when B(y)`				3		5
`x when C(y)`			2			

2.5 Typage, causalité et bonne formation

Le typage, la bonne formation et la causalité sont les trois règles qui décident si un nœud doit ou non être accepté par le compilateur.

Le rejet par le compilateur d'un programme vérifiant ces trois règles ne devrait pas se produire ; cependant il n'existe pas de preuve selon laquelle ceci est vérifié. Un tel rejet n'est cependant pas problématique du point de vue de la sécurité du code produit, puisque justement dans ce cas aucun code n'est produit, mais relève d'une mauvaise analyse du code.

2.5.1 Typage et vérification des horloges

Tous les langages de la chaîne de compilation envisagée sont fortement typés ; et ce typage est statique. De même tout nom de flot se voit donné une horloge.

Il n'y a aucune inférence de type ou d'horloge.

Les expressions ont un type à deux composantes. La première composante est le type des valeurs représentées, c'est-à-dire un tuple de types abstraits ou concrets tels que présentés dans le chapitre précédent.

La seconde composante est une horloge qui s'applique à tous les types du tuple de la première composante ; en particulier, une paire d'expressions sur des horloges différentes sera mal typée. Cette composante disparaît dans le code Obc final, les horloges n'y ayant plus de sens.

2.5.2 Bonne formation

La règle de bonne formation impose que toutes les flots déclarés de l'environnement (hors flots d'entrée du nœud) doivent être associées à une et une seule équation du nœud,

et n'apparaître qu'une seule fois à gauche de l'égalité de cette équation. L'équivalent de cette propriété pour le langage Obc et de dire que dans toute exécution du nœud, si cette exécution accède à une variable, alors cette variable est définie une et une seule fois dans cette exécution. Cette contrainte rappelle celle des langages à affectation unique (SSA en anglais pour Static Single Assignement).

Le but de cette règle est d'interdire les deux situations suivantes :

- Une variable indéfinie dans l'environnement, ce n'est pas problématique d'en avoir une si elle n'est jamais lue (mais alors, elle n'a aucune raison d'apparaître).
- Une variable sur-définie, d'un point de vue sémantique, si les deux expressions définissant la variable produisent bien la même valeur, alors il n'y a pas de problème, mais il faudrait alors s'assurer que c'est le cas, ce qui peut être complexe, voire incalculable en pratique (même si c'est théoriquement décidable, puisqu'on travaille avec un automate à états finis).

2.5.3 Causalité

Pour qu'un nœud ait du sens, il ne suffit pas que toutes les équations soient bien typées et que le nœud soit bien formé ; on doit aussi demander que la sémantique soit déterministe, c'est-à-dire qu'il n'y ait qu'au plus un unique environnement de flots qui soit solution du système posé. Par exemple `x = x;` est bien typé et est bien formé dans un environnement ne contenant que le flot `x`, mais admet autant de solutions qu'il existe de flots compatibles avec le type de `x` ; une telle équation est donc à interdire. De plus, certaines équations, telles que `x = x+1;` ne peuvent pas avoir de solutions avec la définition usuelle de `+` ; alors que dans un contexte où tous les opérateurs et tous les nœuds sont totaux, il est raisonnable de penser que tout nœud causal, bien formé et bien typé admet une unique solution. Cette conjecture n'a cependant pas été démontrée pour nos langages.

Un nœud est dit causal si on peut trouver un ordre de calcul dans les valeurs des suites du nœud à partir de son flot d'entrée. On interdit donc à un nom de flot de dépendre directement ou indirectement de lui même (en s'autorisant cependant à dépendre de son passé).

Une fois le nœud traduit en code impératif, la causalité se traduit par le fait que toutes les variables auxquelles on accède ont été initialisées.

2.6 Conclusion

Les langages synchrones définissent donc, en plus des notions usuelles des langages de programmation, la notion d'horloge. Cette notion d'horloge matérialise dans les langages impératifs le fait qu'une valeur ait ou non besoin d'être calculée.

La sémantique qui nous intéresse est celle des nœuds, c'est-à-dire la suite obtenue à partir d'une suite en entrée, le seul point de haut niveau qu'il reste encore à expliciter est comment exprimer la préservation de la sémantique tout au long de la chaîne de compilation ; c'est ce que décrit le chapitre suivant.

Chapitre 3

Modèles de compilation préservant la sémantique

3.1 Sémantique de Kahn

Dans [33], Gilles Kahn définit une sémantique sur de petites unités de calcul qui doivent vérifier une propriété de continuité. L'espace considéré par Kahn est celui des suites finies ou infinies sur un ensemble E.

3.1.1 Cadre formel

Définition 3.1.1 (Suites finies ou infinies)**.** *Les suites finies ou infinies sont les suites à valeurs dans $E \sqcup \{\bot\}$ stationnaires à leur première occurrence de $\bot$.*

Une suite $(u_n)_{n\in\mathbb{N}}$ est dite infinie si elle ne contient pas $\bot$ ($\forall 0 \leq k, u_k \neq \bot$).

Elle est dite finie dans le cas contraire ; il existe alors un entier $long\left((u_n)_{n\in\mathbf{N}}\right)$ appelé la longueur de la suite $(u_n)_{n\in\mathbf{N}}$ tel que $\forall 0 \leq k < long\left((u_n)_{n\in\mathbf{N}}\right).\ u_k \neq \bot$ et $\forall k \geq long\left((u_n)_{n\in\mathbf{N}}\right).\ u_k = \bot$.

On note E^ω cet espace.

Définition 3.1.2 (Concaténation de suites finies ou infinies, préfixes)**.** *La concaténation de deux suites, u et v, notée $u \circ^\omega v$ est la suite des valeurs de u, suivies de celles de v.*

Si u est infinie, $u \circ^\omega v = u$.

Si u est finie, $(u \circ^\omega v)_n = u_n$ si $n < long(u)$ et $(u \circ^\omega v)_{long(u)+n} = v_n$ sinon.

Si y est une suite finie ou infinie et que x est une suite finie, on note $y = x \circ^\omega \star$ et on dit que x est préfixe de y s'il existe une suite z telle que $y = x \circ^\omega z$.

Les ouverts de la topologie [1] associée à E^ω sont les ensembles O qui vérifient les deux propriétés suivantes :

– si $x \in O$ et que x est préfixe de y alors $y \in O$ (propriété de monotonie)

1. Il s'avère que cette topologie est une topologie de Scott [55] associée à l'ordre préfixe.

– si $x \in O$ est une suite *infinie*, alors il existe une suite *finie* y, préfixe de x tel que $y \in O$ (propriété d'accessibilité)

Une relation R entre suites dénombrables finies ou infinies est dite continue si pour tout ouvert O, l'ensemble $\{x \mid \exists\, y \in O.\, R(x,y)\}$ est un ouvert. Une fonction f entre suites dénombrables finies ou infinies est dite continue si la relation R_f définie par $R_f(x,y) \Leftrightarrow f(x) = y$ est continue.

L'intérêt de la propriété d'accessibilité est d'interdire à la fonction une analyse de la totalité de ses entrées avant de retourner une valeur.

Il ressort de cette définition que les fonctions continues sont monotones, c'est-à-dire que si f est continue, $\forall x, y.\, y = x \circ^\omega \star \Rightarrow f(y) = f(x) \circ^\omega \star$. Pour un exemple de fonction monotone non continue, on peut considérer une fonction qui retournerait la suite vide si ses entrées sont finies et une suite arbitraire non vide si ses entrées sont infinies.

Ce cadre sémantique est commun à plusieurs langages synchrones, notamment Lustre et Lucid Synchrone. Tous les nœuds définis dans les langages de la chaîne de compilation décrite par la suite seront aussi continus.

3.1.2 Machines à états

Une question qui vient alors naturellement est la suivante : « Quelles sont les machines auxquelles on peut rattacher une sémantique continue ? ».

Une réponse est donnée par les machines à états.

Définition 3.1.3 (Machine à états)**.** *Une machine M à états est la donnée d'un ensemble S appelé ensemble des états, d'un ensemble I appelé ensemble des entrées, d'un ensemble O appelé ensemble des sorties, d'une fonction trans de $I \times S$ dans $O^\omega \times S$ appelée fonction de transition, d'un état initial s_0 et d'une production initiale o_0 dans O^ω.*

Définition 3.1.4 (Exécution d'une machine à états)**.** *Une machine à états lit une suite finie ou infinie $(u_n)_{n\in\mathbf{N}}$, et produit une suite finie ou infinie $(v_n)_{n\in\mathbf{N}}$ à valeurs dans $O^\omega \times S$ définies par :*

$$\begin{array}{rcl} v_0 & = & (o_0, s_0) \\ v_{n+1} & = & (o_n \circ^\omega o, t) \\ & & \quad \textit{avec } (o_n, s_n) = v_n \textit{ et } \textit{trans}(u_n, s_n) = (t, o) \textit{ si } u_n \neq \bot \\ v_{n+1} & = & v_n \textit{ si } u_n = \bot \end{array}$$

On appelle « exécution de la machine M sur $(u_n)_{n\in\mathbf{N}}$ » la suite v_n ainsi obtenue.

De façon plus informelle, la n-ième valeur de l'exécution de la machine à états M est l'état atteint après lecture de la n-ième valeur de $(u_n)_{n\in\mathbf{N}}$ et la suite de toutes les valeurs produites jusqu'à la lecture de la n-ième valeur de $(u_n)_{n\in\mathbf{N}}$.

La seconde composante de cette suite (les états) n'est en général pas convergente et n'est donc pas intéressante d'un point de vue dénotationnel.

La première composante en revanche (les suites) est croissante pour la relation préfixe. On peut alors définir la suite w_n par :

$$\begin{array}{lrcl} \text{si } \forall m, o, s.\ v_m = (o,s) \Rightarrow o_n = \bot \text{ alors} & w_n & = & \bot \\ \text{si } \exists m, o, s.\ v_m = (o,s) \wedge o_n \neq \bot \text{ alors} & w_n & = & o_n \end{array}$$

Par monotonie de la seconde composante de v_n, cette suite est définie de façon unique.

On peut alors montrer que la fonction qui à $(u_n)_{n\in\mathbf{N}}$ associe $(w_n)_{n\in\mathbf{N}}$ ainsi calculée est continue au sens de la topologie de Kahn.

Réciproquement, toute fonction continue au sens de la topologie de Kahn peut s'implémenter par une machine à états, en prenant pour S l'ensemble I^ω, pour pouvoir stocker toutes les valeurs de $(u_n)_{n\in\mathbf{N}}$ lues jusque là.

3.1.3 Machines à états finis

Les machines à états capturent bien les fonctions que l'on veut définir. Cependant elles capturent aussi des fonctions qui ne sont pas raisonnables pour le temps réel critique.

La première restriction à imposer est de travailler en mémoire bornée, donc l'ensemble des états doit être fini.

La seconde restriction est de travailler en temps d'exécution borné, les sorties produites à chaque étape doivent donc être des suites nécessairement finies, et même de longueur bornée.

Définition 3.1.5 (Machine à états finis)**.** *Une machine à états finis est une machine dont l'ensemble des états est fini et dont la fonction de transition ne produit que des suites de longueur inférieure à N pour un N donné. La suite initiale o_0 doit aussi être bornée.*

Le langage Lustre impose encore comme restriction que le N de la définition ci-dessus soit inférieur ou égal à 1, et la suite initiale vide. Ceci s'interprète en énonçant que les sorties d'un nœud sont toujours plus lentes (au sens large) que les entrées. C'est-à-dire que pour produire une valeur, il faut toujours en consommer au moins une.

Les langages dont il sera question dans cette thèse demanderont en fait d'avoir $N = 1$ et une suite initiale vide, c'est-à-dire que la k-ième entrée correspond à la k-ième sortie et réciproquement. Les machines à états finis de cette forme sont donc les cibles de la chaîne de compilation qui va être exposée dans cette thèse. De telles machines sont appelées machines de Mealy [41]. La section 2.3 fait d'ailleurs référence à une telle machine à états finis comme variante de la sémantique flot de données.

3.2 Unification des sémantiques

Comme annoncé précédemment en section 2.3, on a deux sémantiques alternatives pour les nœuds. On veut pouvoir utiliser tantôt l'une des deux, tantôt l'autre. Pour pouvoir parler de préservation de sémantique tout au long de la chaîne, il faut pouvoir unifier les deux sémantiques.

3.2.1 Contraintes sur la sémantique instantanée

On veut que cette sémantique soit le moins contrainte possible afin de pouvoir compiler le plus possible de sémantiques flot de données. En fait il n'existe aucune contrainte sur cette sémantique pour qu'elle soit unifiable avec une sémantique flot de données.

3.2.2 Contraintes sur la sémantique flots de données

Définition 3.2.1 (sémantique point à point)**.** *Soit $N^{\mathcal{F}}_{sem}(n)$ une sémantique flot de données d'un nœud de signature $\sigma_{ent} \to \sigma_{sort}$; on dit qu'elle est point à point si :*

$$\forall \vec{\phi_i}, \vec{\phi_o}. \quad (\vec{\phi_i}, \vec{\phi_o}) \in N^{\mathcal{F}}_{sem}(n) \Rightarrow \bigvee \left\{ \begin{array}{ll} (\vec{\phi_i}, \vec{\phi_o}) = (\epsilon, \epsilon) & \\ \exists \phi_{i'}, \phi_{o'}, \alpha_{i'}, \alpha_{o'}. & \bigwedge \left\{ \begin{array}{l} \phi_i = \phi_{i'} \circ [\alpha_{i'}] \\ \phi_o = \phi_{o'} \circ [\alpha_{o'}] \\ (\phi_{i'}, \phi_{o'}) \in N^{\mathcal{F}}_{sem}(n) \end{array} \right. \end{array} \right.$$

Pour que la sémantique flot de données soit unifiable avec une sémantique instantanée, il faut qu'elle soit définie point à point.

En d'autres termes, il faut que tous les flots soient de même longueur, et que si on a un élément dans cette sémantique, alors en lui enlevant tous ses derniers instants, on obtient un élément qui reste dans la sémantique. La propriété d'être point à point a deux grosses implications :

- On peut se passer des suites infinies dans les définitions, car toute sémantique point à point admet un prolongement par continuité dans la sémantique de Kahn. La propriété d'accessibilité pour les ouverts devient alors inutile, et les fonctions sont alors réduites aux fonctions monotones.
- On perd la possibilité d'avoir des entrées et des sorties qui vont sur des rythmes différents, comme c'est le cas en Lustre (et Lucid Synchrone).

La condition nécessaire et suffisante pour qu'une sémantique flot de données soit unifiable avec une sémantique instantanée est en fait donnée par un relèvement de sémantique.

3.2.3 Relèvement de sémantiques

Définition 3.2.2 (relèvement de sémantiques)**.** *Soit (M_0, S) une sémantique instantanée sur une signature $\sigma_{ent} \to \sigma_{sort}$; on peut alors construire un élément* $\mathrm{Lift}_M(M_0, S)$ *de $\mathfrak{P}((\llbracket \sigma_{ent} \rrbracket^* \times \llbracket \sigma_{sort} \rrbracket^*) \times \llbracket N_{mem}(n) \rrbracket)$ tel que :*

$$\begin{array}{l} \forall \phi_i, \phi_o, \vec{\alpha_m}. \ ((\phi_i, \phi_o), \vec{\alpha_m}) \in \mathrm{Lift}_M(M_0, S) \\ \quad \Rightarrow \bigvee \left\{ \begin{array}{ll} \phi_i = \epsilon \wedge \phi_o = \epsilon \wedge \vec{\alpha_m} = M_0 & \\ \exists \phi_{i'}, \phi_{o'}, & \bigwedge \left\{ \begin{array}{l} \phi_i = \phi_{i'} \cdot \alpha_{i'} \\ \phi_o = \phi_{o'} \cdot \alpha_{o'} \\ ((\phi_{i'}, \phi_{o'}), \vec{\alpha_{m'}}) \in \mathrm{Lift}_M(M_0, S) \\ ((\alpha_{i'}, \vec{\alpha_{m'}}), (\alpha_{o'}, \vec{\alpha_m})) \in S \end{array} \right. \\ \alpha_{i'}, \alpha_{o'}, \vec{\alpha_{m'}}. & \end{array} \right. \end{array}$$

Cette existence peut nous être donnée par le théorème de point fixe de Kleene en montrant que l'ensemble des parties de $E = (\llbracket \sigma_{ent} \rrbracket^* \times \llbracket \sigma_{sort} \rrbracket^*) \times \llbracket N_{mem}(n) \rrbracket$ peut être muni d'une structure d'ordre partiel complet pour la relation d'inclusion. On note $ext(\phi_{i'}, \phi_{o'}, \vec{\alpha_{m'}})$ l'ensemble des triplés $(\phi_i, \phi_o, \vec{\alpha_m})$ tels qu'il existe $\alpha_{i'}$ et $\alpha_{o'}$ pour lesquels $\phi_i = \phi_{i'} \cdot \alpha_i$, $\phi_o = \phi_{o'} \cdot \alpha_o$ et $((\alpha_{i'}, \vec{\alpha_{m'}}), (\alpha_{o'}, \vec{\alpha_m})) \in \mathrm{Lift}_M(M_0, S)$. L'application qui à toute partie p de E associe $\bigcup_{((\phi_{i'}, \phi_{o'}), \vec{\alpha_{m'}}) \in p} ext(\phi_{i'}, \phi_{o'}, \vec{\alpha_{m'}}) \cup p \cup \{((\epsilon, \epsilon), M_0)\}$ est alors continue au sens de Scott. Elle admet donc un plus petit point fixe qui est la relation cherchée entre des flots d'entrée, ceux de sortie et le dernier état interne calculé.

Intuitivement, cela revient à "faire tourner" la sémantique instantanée en donnant à l'itération suivante la mémoire précédemment calculée.

On appelle alors relèvement sémantique de la sémantique instantanée de (M_0, S) *et on note* $\mathrm{Lift}\,(M_0, S)$ *la projection de* $\mathrm{Lift}_M\,(M_0, S)$ *sur sa première coordonnée (on oublie la mémoire instantanée).*

Définition 3.2.3 (sémantiques unifiées)**.** *On dit que les sémantiques flot de données et instantanées sont unifiées, dès lors que le relèvement de la sémantique instantanée est extensionnellement égal à la sémantique flots de données.*

Le relèvement de sémantique est alors toujours une relation continue.

En revanche une sémantique continue flot de données ne peut pas toujours être ramenée à une sémantique instantanée. Les cas où ce n'est pas possible impliquent en réalité des programmes qui auraient besoin d'une mémoire non bornée (d'où l'impossibilité de proposer un $N_{mem}\,(n)$). Par construction cependant, tous les nœuds que l'on pourra définir par les syntaxes présentées dans les parties suivantes ne nécessiteront pas de mémoire infinie.

Dans les chaînes de compilations, l'environnement contiendra toujours des nœuds à sémantique unifiée. L'un des principaux buts de cette thèse et de montrer que la compilation préserve la sémantique ; et donc ici que les sémantiques flots de données des nœuds seront unifiées avec leurs sémantiques instantanées.

3.3 Préservations de sémantique

3.3.1 Compilateurs

Comme présenté en introduction, les compilateurs sont simplement des traducteurs d'un langage vers un autre.

Tout comme la traduction des langues naturelles, on doit s'attendre à ce que la traduction "ait le même sens" que le texte original ; et tout comme la traduction des langues naturelles, ce n'est pas toujours tout à fait le cas, mais on peut s'en accommoder si la traduction est "suffisamment proche".

La sous sections suivante est là pour donner un sens précis à "ait le même sens que" et "suffisamment proche".

Enfin, comme toute traduction, on peut se permettre d'échouer, surtout lorsque le texte original n'a pas de sens.

Un compilateur n'est donc pas simplement une fonction d'un code source vers du code cible, mais une fonction d'un code source vers du code cible ... ou une erreur.

Comme seules les compilations qui n'échouent pas nous intéressent, nous introduisons une notation :

Notation 3.3.1 (compilation sans erreur)**.** $n_s \overset{trad}{\Rightarrow} n_t$ *signifie que la compilation de* n_s *par trad a réussi et a produit* n_t*.*

3.3.2 Complétude et correction de compilateurs

Tous les langages considérés dans nos passes de compilation ont un cadre sémantique commun afin de pouvoir mettre en relation le code source et le code produit.

Ce type de sémantique est dite dénotationnelle [56] ; elle abstrait le contenant (le programme) pour se focaliser sur le contenu (ce qu'il représente), cela permet de raisonner modulo tous les programmes qui représente le même objet mathématique.

En contre partie, on ne peut plus raisonner sur d'autres aspects tels que la complexité du programme ou son occupation mémoire. Ce n'est de toutes façons pas dans les objectifs de cette thèse, et ces notions n'ont que peu de valeur dans Ls puisque sa sémantique n'expose ni de modèle mémoire ni de modèle de complexité. La complexité et l'occupation mémoire d'un programme ont cependant du sens dans le cadre du logiciel critique (assurance de réalisabilité du code avec les contraintes d'occupation mémoire et de pire temps d'exécution), et devront être analysés après compilation (on ne tente pas ici de pouvoir les estimer en analysant le source et vérifier que le code produit satisfait à ces estimations).

À partir de là se pose notre problématique générale : «Que veut dire être bien compilé ?».

Un nœud n_s est bien compilé en un programme n_t si il existe une certaine relation entre les sémantiques $N^{\mathcal{F}}_{sem}(n_s)$ et $N^{\mathcal{F}}_{sem}(n_t)$.

Les relations simples envisageable sont l'inclusion, la contenance et l'égalité.

3.3.3 Nature des compilateurs

Complétude

C'est la préservation liée à la relation d'inclusion. En termes de traductions de langues naturelles, cela reviendrait à donner un sens plus large que le sens d'origine ; par exemple traduire «The sky is blue.» par «Le ciel a une couleur.».

Définition 3.3.1 (complétude)**.** *Étant donnée une fonction de compilation trad, on dit que trad est complète si :*

$$\forall n_s, n_t. \left(s \overset{trad}{\Rightarrow} t\right) \Rightarrow N^{\mathcal{F}}_{sem}(n_s) \subseteq N^{\mathcal{F}}_{sem}(n_t)$$

Si cette propriété est vérifiée, alors si la sémantique $N^{\mathcal{F}}_{sem}(n_s)$ autorise un nœud n_s à produire des sorties o sur des entrées i, alors sa sémantique compilée $N^{\mathcal{F}}_{sem}(n_t)$ autorise aussi le nœud compilé n_t à produire o sur i.

Il n'y a pas de réaction autorisée par le source qui ne soit pas capturée par son compilé.

Notation 3.3.2 (traduction complète)**.** *On note $n_s \lesssim n_t$ la relation $N^{\mathcal{F}}_{sem}(n_s) \subseteq N^{\mathcal{F}}_{sem}(n_t)$.*

On note $\overset{\lesssim}{}trad$ le prédicat selon lequel trad préserve complètement la sémantique.

Bien qu'en apparence moins intéressante que la correction, parce que la sémantique des nœuds compilés est déterministe, qu'elle est souvent plus facile à démontrer, plus naturelle à énoncer et qu'au pire empêcherait une erreur d'exécution, c'est celle qui nous intéressera.

Correction

C'est la préservation liée à la relation de contenance. En termes de traduction de langues naturelles, cela revient à donner un sens plus précis que le sens d'origine (par exemple traduire «The sky is blue.» par «Le ciel est bleu et nuageux.»).

Définition 3.3.2 (correction)**.** *Étant donnée une fonction de compilation trad, on dit que trad est correcte si :*

$$\forall n_s, n_t. \left(n_s \overset{trad}{\Rightarrow} n_t\right) \Rightarrow N^{\mathcal{F}}_{sem}(n_s) \supseteq N^{\mathcal{F}}_{sem}(n_t)$$

Si cette propriété est vérifiée, alors si la sémantique compilée $N^{\mathcal{F}}_{sem}(n_t)$ autorise le nœud compilé n_t à produire des sorties o sur des entrées i, alors la sémantique $N^{\mathcal{F}}_{sem}(n_s)$ autorisait déjà le nœud n_s à produire o sur i.

Le compilé n'a pas le droit d'inventer un nouveau sens au programme, mais il peut éventuellement en perdre.

Notation 3.3.3 (traduction correcte)**.** *On note $n_s \gtrsim n_t$ la relation $N^{\mathcal{F}}_{sem}(n_s) \supseteq N^{\mathcal{F}}_{sem}(n_t)$.*

On note $\overset{\gtrsim}{}trad$ le prédicat selon lequel trad préserve correctement la sémantique.

La propriété de correction est souvent la première à exiger d'un programme vis-à-vis d'un comportement, refuser des programmes corrects parce qu'on manque de confiance dans leur comportement est assez naturel, cependant il est souvent plus difficile de démontrer la correction que la complétude.

Perfection

C'est la préservation liée à la relation d'égalité.

Définition 3.3.3 (perfection)**.** *Étant donnée une fonction de compilation trad, on dit que trad est parfaite si :*

$$\forall n_s, n_t. \left(n_s \overset{trad}{\Rightarrow} n_t\right) \Rightarrow N^{\mathcal{F}}_{sem}(n_s) = N^{\mathcal{F}}_{sem}(n_t)$$

Ce serait la relation idéale à vérifier ; en effet, d'un point de vue sémantique n_s et son compilé n_t seraient indissociables ; cette relation cumule cependant toutes les contraintes des deux précédentes.

Notation 3.3.4 (traduction parfaite)**.** *On note $n_s \sim n_t$ la relation $N^{\mathcal{F}}_{sem}(n_s) = N^{\mathcal{F}}_{sem}(n_t)$.*

On note $\overset{\sim}{}trad$ le prédicat selon lequel trad préserve parfaitement la sémantique.

Ces définitions sont analogues à celles des simulations. La complétude correspond à la simulation en avant (forward simulation), alors que la correction correspond à la simulation en arrière (backward simulation) et la perfection à la bisimulation [36].

3.3.4 Natures de sémantiques

Pour justifier le choix d'une compilation complète plutôt que correcte ou que correcte et complète, on peut chercher à quelles conditions sur $N^{\mathcal{F}}_{sem}(n_s)$ et $N^{\mathcal{F}}_{sem}(n_t)$, la correction et la complétude coïncident.

Déterminisme

Définition 3.3.4 (déterminisme)**.** *Étant donnée une sémantique $N^{\mathcal{F}}_{sem}(n_s)$, on dit qu'elle est déterministe si :*

$$\forall i, o_1, o_2.\, (i, o_1) \in N^{\mathcal{F}}_{sem}(n_s) \wedge (i, o_2) \in N^{\mathcal{F}}_{sem}(n_s) \Rightarrow o_1 = o_2$$

Notation 3.3.5 (déterminisme)**.** *Étant donnée une sémantique $N^{\mathcal{F}}_{sem}(n_s)$, on note $Det\left(N^{\mathcal{F}}_{sem}(n_s)\right)$ le prédicat selon lequel la sémantique $N^{\mathcal{F}}_{sem}(n_s)$ est déterministe.*

Le déterminisme n'est pas toujours une propriété souhaitée dans les langages déclaratifs ; par exemple `Prolog` n'est pas déterministe.

Cependant, parmi nos objectifs, il y en a deux qui font que nos langages seront déterministes :

- la transparence référentielle fait que l'on peut faire de la factorisation de code. Sans déterminisme, ce n'est plus possible, car alors factoriser le code restreint la sémantique ; des programmes non déterministes acceptables peuvent donner des programmes non acceptables après factorisation.
  Par exemple si on suppose qu'un nœud $n()$ peut produire 0 ou 1, alors le système :
  $$\begin{cases} x = n() \\ y = n() \\ z = \frac{1}{(1-x)\times y} \end{cases}$$
  peut être acceptable, mais le programme suivant ne l'est plus :
  $$\begin{cases} x = n() \\ y = x \\ z = \frac{1}{(1-x)\times y} \end{cases}$$
- le code final est pour des raisons d'efficacité proche du code machine et ne doit pas faire de choix. On verra par la suite que la complétude de la compilation a pour conséquence le déterminisme de toute la chaîne de langages.

Totalité

Définition 3.3.5 (totalité)**.** *Étant donnée une sémantique $N^{\mathcal{F}}_{sem}(n_s)$, on dit qu'elle est totale si :*

$$\forall i. \exists o.\, (i, o) \in N^{\mathcal{F}}_{sem}(n_s)$$

Notation 3.3.6 (totalité)**.** *Étant donnée une sémantique $N^{\mathcal{F}}_{sem}(n_s)$, on note $Tot\left(N^{\mathcal{F}}_{sem}(n_s)\right)$ le prédicat selon lequel la sémantique $N^{\mathcal{F}}_{sem}(n_s)$ est totale.*

En pratique, les fonctions considérées ne sont pas toujours totales [2]. On peut alors considérer le domaine des entrées pour lesquelles la sémantique est définie. Alternativement, on peut détecter les entrées qui posent problèmes et retourner une valeur par défaut accompagnée d'un code d'erreur afin de rendre la fonction totale [3].

2. Par exemple, la division qui pose problème lorsque son argument est nul.

3. Dans le cas de la division, par zéro, on pourra retourner un tuple contenant un booléen indiquant le bon déroulement, et le résultat avec zéro pour valeur par défaut.

Il n'y a pas de preuve que les langages de la chaîne de compilation soient totaux. Cependant il est raisonnable de penser que dans un contexte où toutes les fonctions abstraites importées sont totales sur les entrées bien typées, alors les nœuds que l'on peut définir sont eux aussi totaux sur les entrées bien typées.

3.3.5 Lemmes de sémantique élémentaires

Lemme 3.3.1 (Lemmes de transferts)**.** *Les propriétés suivantes sont vérifiées :*

1. $\forall n_s, n_t.s \gtrsim t \wedge Det\left(N^{\mathcal{F}}_{sem}(n_s)\right) \Rightarrow Det\left(N^{\mathcal{F}}_{sem}(n_t)\right)$
2. $\forall n_s, n_t.s \gtrsim t \wedge Tot\left(N^{\mathcal{F}}_{sem}(n_t)\right) \Rightarrow Tot\left(N^{\mathcal{F}}_{sem}(n_s)\right)$
3. $\forall n_s, n_t.s \gtrsim t \wedge Det\left(N^{\mathcal{F}}_{sem}(n_s)\right) \wedge Tot\left(N^{\mathcal{F}}_{sem}(n_t)\right) \Rightarrow n_s \lesssim n_t$
4. $\forall n_s, n_t.s \lesssim t \wedge Tot\left(N^{\mathcal{F}}_{sem}(n_s)\right) \Rightarrow Tot\left(N^{\mathcal{F}}_{sem}(n_t)\right)$
5. $\forall n_s, n_t.s \lesssim t \wedge Det\left(N^{\mathcal{F}}_{sem}(n_t)\right) \Rightarrow Det\left(N^{\mathcal{F}}_{sem}(n_s)\right)$
6. $\forall n_s, n_t.s \lesssim t \wedge Det\left(N^{\mathcal{F}}_{sem}(n_t)\right) \wedge Tot\left(N^{\mathcal{F}}_{sem}(n_s)\right) \Rightarrow n_s \gtrsim n_t$

En démontrant que toutes les passes de compilation sont complètes, et en démontrant que le code cible (Obc) est déterministe, on en déduira par (5) que chaque langage intermédiaire est déterministe.

En se restreignant au programmes sur lesquels on donne des entrées qui produisent des sorties, par (4), on en déduira que les sémantiques de tous les langages intermédiaires de la chaîne sont totales sur les entrées productives du langage source (Ls).

En combinant ces deux points et (6), on en déduit que la traduction est parfaite sur les entrées productrices de Ls.

Les propriétés (1), (2) et (3) ne sont pas utilisées, elles sont juste données pour compléter la symétrie ; leur démonstration n'est pas donnée ici, mais les preuves sont tout aussi simples.

Les lemmes (4) et (5) ne sont que des propriétés de composition qui assurent la préservation d'une propriété tout au long de la chaîne.

Le lemme (6) est le seul véritable lemme d'importance, et qui justifie le choix d'une préservation complète des sémantique.

Preuve des lemmes de transfert (4)*,* (5) *et* (6)*.* Les preuves sont directes et n'utilisent aucune astuce.

- *Preuve du lemme* (4)
  - Soient n_s et n_t tels que $s \lesssim t$ et $Tot\left(N^{\mathcal{F}}_{sem}(n_s)\right)$, et i une entrée ; cherchons alors à démontrer que $\exists o.(i,o) \in N^{\mathcal{F}}_{sem}(n_t)$.
  - Par totalité de $N^{\mathcal{F}}_{sem}(n_s)$, nous avons l'existence de o tel que $(i,o) \in N^{\mathcal{F}}_{sem}(n_s)$; c'est notre témoin existentiel. Nous devons alors montrer que $(i,o) \in N^{\mathcal{F}}_{sem}(n_t)$, ce qui est donné par $n_s \lesssim n_t$.
- *Preuve du lemme* (5)

  - Soient n_s et n_t tels que $s \lesssim t$ et $Det\left(N^{\mathcal{F}}_{sem}(n_t)\right)$, i une entrée, o_1 et o_2 deux sorties telles que $(i, o_1) \in N^{\mathcal{F}}_{sem}(n_s)$ et $(i, o_2) \in N^{\mathcal{F}}_{sem}(n_s)$; cherchons alors à démontrer que $o_1 = o_2$.
  - Comme $n_s \lesssim n_t$ et $(i, o_1) \in N^{\mathcal{F}}_{sem}(n_s)$ (resp. $(i, o_2) \in N^{\mathcal{F}}_{sem}(n_s)$), nous obtenons $(i, o_1) \in N^{\mathcal{F}}_{sem}(n_t)$ (resp. $(i, o_2) \in N^{\mathcal{F}}_{sem}(n_t)$).
  - En appliquant l'hypothèse de déterminisme de $Det\left(N^{\mathcal{F}}_{sem}(n_t)\right)$, on conclut.
- *Preuve du lemme* (6)
  - Soient n_s et n_t tels que $n_s \lesssim n_t$, $Tot\left(N^{\mathcal{F}}_{sem}(n_s)\right)$ et $Det\left(N^{\mathcal{F}}_{sem}(n_t)\right)$, i une entrée et o une sortie telles que $(i, o) \in N^{\mathcal{F}}_{sem}(n_t)$; cherchons alors à démontrer que $(i, o) \in N^{\mathcal{F}}_{sem}(n_s)$.
  - Par totalité de $N^{\mathcal{F}}_{sem}(n_s)$, nous avons l'existence de o_1 tel que $(i, o_1) \in N^{\mathcal{F}}_{sem}(n_s)$. Pour conclure il nous suffit donc de prouver que $o = o_1$.
  - Comme $n_s \lesssim n_t$, on en déduit $(i, o_1) \in N^{\mathcal{F}}_{sem}(n_t)$.
  - Par le déterminisme de n_t, comme $(i, o) \in N^{\mathcal{F}}_{sem}(n_t)$ et $(i, o_1) \in N^{\mathcal{F}}_{sem}(n_t)$, on peut conclure.

□

3.4 Extraction d'un compilateur certifié

Avant d'aborder la section sur la certification, voici une brève présentation du langage Coq et du mécanisme d'extraction.

3.4.1 Présentation du langage Coq [7, 11]

Coq est un langage de programmation fonctionnelle basé sur le calcul des constructions inductives (CoIC). Historiquement, il est issu du projet Formel commencé en 1982, et s'appuyait initialement sur le calcul des constructions inductives (CoC), d'où son nom. Ce langage réunit de nombreuses fonctionnalités des langages de programmation modernes (typeclasses *à la* Haskell, modules *à la* Ocaml), ainsi que deux fonctionnalités phares du langage, à savoir les types dépendants de termes, et la séparation des types en deux sortes, l'une qui représente les objets avec lesquels on peut calculer, et l'autre qui représente des informations sur des objets.

Coq pour le programmeur

Coq en tant que langage de programmation comprend :
- des directives de chargement de modules *à la* Haskell.
  Exemple : `Require Import List.`
- la définition de fonctions anonymes
  Exemple : `fun (x : nat) => 2 * x + 1`
- l'ordre supérieur, c'est-à-dire la possibilité d'écrire des fonctions qui prennent en argument des fonctions et/ou dont la valeur de retour est elle même une fonction.
  Exemple : `Let o(A B:Type)(f:A->B):=fun (C:Type)(g:C->A)c => f (g c).`

```
(* Un exemple de type de donnees recursif *)
Inductive List (A : Type) : Type :=
  | Nil : List A
  | Cons : A -> List A -> List A
.
```

Figure 3.1: Types algébriques simples en Coq

```
(* Un type de donnees algebrique generalise *)
Inductive Expr : Type -> Type :=
  | Etrue : Expr bool
  | Efalse : Expr bool
  | E0 : Expr nat
  | ES : Expr nat -> Expr nat
  | Eplus : Expr nat -> Expr nat -> Expr nat
  | Eif : forall A, Expr bool -> Expr A -> Expr A -> Expr A
.
```

Figure 3.2: Types algébriques généralisés en Coq

- l'inférence de types [4].
  Exemple : `fun x => 2 * x + 1` (le type de x ne peut être que *nat*, et Coq arrive à le deviner automatiquement).
- des types de données algébrique (figure 3.1), ainsi que le filtrage (pattern-matching) dans les expressions.
- un système de modules *à la* OCaml qui occupent une place beaucoup moins importante que dans OCaml puisqu'ils peuvent souvent être remplacés par des types de données algébriques, un peu de la même manière que les modules OCaml peuvent être aisément simulés en Haskell [62].
- un système de classes de types *à la* Haskell

En plus de ces fonctionnalités, les structures de données algébriques utilisées peuvent dépendre de types comme dans les types de données algébriques généralisés [3] (figure 3.2), comme en Haskell et qui sont apparus dans la version 4.0 d'OCaml. Mais ces types peuvent aussi dépendre directement de termes (figure 3.3). Le filtrage sur les types dépendants est plus complexe que le fitrage sur les types non dépendants. Un exemple en sera donné en figure 3.6 dans la section suivante.

Maintenant, donnons quelques raisons pour lesquelles personne n'utilise Coq en tant que langage de programmation pour développer des programmes de la vie de tous les jours.

- Coq est dépourvu de tout effet de bord et son exécution est indépendante d'un environnement extérieur. Il n'est donc pas possible de récupérer une valeur donnée par l'utilisateur durant l'exécution du programme, ni de modifier le contenu d'une variable globale.

4. le système de types de Coq étant plus complexe que celui de OCaml ou de Haskell, il est cependant parfois nécessaire de donner explicitement des informations de typage.

```
(* Un type dependant de terme (autre qu'un type) *)
Inductive Vecteur (A : Type) : nat -> Type :=
  | Fin : Vecteur A 0
  | Deb : forall n, Vecteur A n -> Vecteur A (S n)
.
```

Figure 3.3: Types dépendants en Coq

- Coq n'est pas Turing-complet ; c'est-à-dire qu'il existe des algorithmes que l'on peut implémenter en utilisant un langage de programmation traditionnel que l'on ne peut pas implémenter en Coq [5], car toute fonction écrite en Coq termine. Ainsi par exemple, on n'est pas en mesure à l'heure actuelle d'écrire un programme en Coq qui retourne `true` si et seulement si la suite de Collatz [6] atteint la valeur 1, alors que cela se fait sans difficulté en C, Ocaml, Haskell, ... sous hypothèse de ressources infinies.
  De plus, même dans le cas où on sait prouver la terminaison du programme, il faut le faire pour définir le programme souhaité, et c'est souvent lourd à mettre en place.
- Coq ne dispose pas de mécanisme d'exceptions ; les fonctions écrites en Coq doivent être totales. En conséquence, il faut soit produire une valeur artificielle dans les cas impossibles, soit démontrer que le cas considéré n'est pas possible, soit modifier le type de retour pour prendre en compte les cas que l'on ne veut pas considérer. La première solution rendrait difficile la détection d'erreur (qui par ailleurs ne pourraient pas être rattrapées) ; la seconde est lourde à mettre en place, et n'est pas toujours envisageable ; la troisième oblige à rattraper systématiquement les erreurs éventuelles et peut s'avérer d'une utilisation pénible.
- le typage dépendant est une fonctionnalité intéressante, mais sa mise en œuvre est complexe et n'en vaut souvent pas la peine ; en retirant cette fonctionnalité on se retrouve essentiellement avec un langage plus contraignant que ceux utilisés habituellement.

On a donc à peu de choses près les mêmes points forts et les mêmes points faibles que dans le langage Agda 2 [45].

Coq pour le logicien

En mathématiques, on peut considérer deux sortes d'objets ; les objets mathématiques d'une part et les propriétés (objets méta-mathématiques) d'autre part. Par exemple l'expression $2+2=4$, est une propriété mettant en relation deux objets mathématiques $2+2$ et 4. Il arrive également de travailler sur des ensembles vérifiant une certaine propriété, comme $\{x|x>0\}$; un tel ensemble est un objet mathématique, et à chaque élément de cet ensemble est associé une propriété.

Ces deux objets sont cependant très liés, d'une part parce que l'un comme l'autre peut

5. Cependant, si on sait prouver dans le formalisme de Coq qu'un tel algorithme termine, alors on peut l'implémenter en Coq

6. Une suite définie par $u_{k,0}=k$ et $u_{k,n+1}=u_{k',n}$ avec $k'=\frac{k}{2}$ si k est pair et $k'=3\times k+1$ si k est impair

```
Inductive nat : Set  := (* les entiers naturels          *)
  | O : nat             (* O est un entier naturel       *)
  | S : nat -> nat      (* S est la fonction successeur *)
.
Notation "0" := O.
Notation "n '+1'" := (S n) (at level 0).
```

Figure 3.4: Un exemple d'objets mathématique en Coq : les entiers naturels, le zéro, la fonction successeur

être crée à partir de l'autre comme l'un ($\{x|x > 0\}$ est un objet mathématique qui utilise une propriété dans sa définition ; et $2 + 2 = 4$ est une propriété qui met en relation deux objets mathématiques) ; d'autre part du fait d'une correspondance entre objets mathématiques et preuves, connue sous le nom de correspondance de Curry-Howard [30]. Cette correspondance permet d'associer à chaque ensemble mathématique E un énoncé P_E, et à chaque élément de E une preuve de P_E et vice-versa, de façon bijective. Typiquement, on aura $P_{A\times B} = P_A \wedge P_B$ et de même que tout élément de $A \times B$ est une paire (a, b) avec $a \in A$ et $b \in B$, toute preuve de $P_A \wedge P_B$ est en fait composée d'une preuve de P_A et d'une preuve de P_B. Un autre exemple est celui de l'ensemble $A{\rightarrow}B$ des fonctions de A dans B, qui correspond à l'énoncé $P_A \Rightarrow P_B$: à chaque élément de A (resp. à chaque preuve de P_A), on associe un élément de B (resp. une preuve de P_B).

Il existe une raison cependant de garder la distinction entre ces deux objets ; c'est le « principe d'indifférence aux preuve » (proof-irrelevance en anglais) qui stipule que la construction d'un objet donné ne peut dépendre de sa preuve. En particulier, d'un énoncé de la forme $\exists x, x > 0$ on ne peut extraire un entier. L'un des intérêts de ce principe est le mécanisme d'extraction ; les preuves doivent être vues comme des garanties et non des calculs ; on ne veut pas « calculer » une preuve pour fournir un résultat.

L'assistant de preuve Coq se base sur ces remarques avec un même langage pour ces deux objets (figures 3.4 et 3.5), mais avec pour restriction l'interdiction d'inspecter une propriété (`Prop` dans Coq) pour produire un objet mathématique[7] (`Set` dans Coq). La figure 3.7 donne un cas concret de cette restriction.

Le mot clé `Notation` rencontré en figure 3.4 permet d'étendre la syntaxe de Coq. Cette fonctionnalité n'est pas fondamentale, mais permet une lecture souvent plus agréable. Sans ce système, il faudrait écrire `strictp (S n)` en dernière ligne de l'exemple de la figure 3.5 à la place de `strictp (n +1)`.

filtrage dépendant Les preuves (et définitions) qui doivent filtrer sur des types dépendants doivent être annotés pour indiquer comment le filtrage affecte le type de retour. Un exemple en est donné en figure 3.6. Pour prouver que `strictp 0` est absurde, on inspecte cette hypothèse, le seul cas possible est `PosStr m` pour un certain `m` qui doit vérifier `m +1` = `0`. Un tel `m` n'existe pas bien entendu, mais le système de typage de Coq ne s'en aperçoit pas naturellement. En réalité, dans le cas général savoir si ce genre d'égalité est

7. sauf dans certains cas particuliers où la propriété ne contient pas d'objet mathématique, ces cas sont décris dans la section 4.5.4, partie « empty and singleton elimination » du manuel de référence [39]

```
Inductive False : Prop := (* la propriete fausse                    *)
.                         (* elle n'a aucune regle de construction *)
Inductive True : Prop := (* la propriete trivialement vraie *)
  | I : True             (* I est la preuve triviale        *)
.
Inductive eq (S : Set) (s : S) : S -> Prop := (* propriete egal a s *)
  | eq_refl : eq S s s                        (* s est egal a s     *)
.
Notation "x = y" := (@eq _ x y) (at level 70).

Inductive strictp : nat -> Prop :=    (* etre strictement positif *)
  | PosStr : forall n, strictp (n +1) (* tout successeur l'est    *)
.
```

Figure 3.5: Quelques propriétés en Coq

```
Definition strictp_n_spec :
 forall n, strictp n -> match n with 0 => False | _ +1 => True end :=
  fun n (H : strictp n) => match H with PosStr m => I end.
  (* 'H' = 'PosStr m', donc 'n' = 'm +1', donc on doit prouver 'True' *)

Definition strictp_0_absurde1 : strictp 0 -> False :=
  strictp_n_spec 0.
  (* En utilisant la definition precedente, avec 'n' = '0' *)

Definition strictp_0_absurde2 : strictp 0 -> False :=
  fun s => match s in strictp x return match x return Prop
                                      with | 0 => False
                                           | _ +1 => True
                                      end (* = False car x = 0 *)
           with | PosStr m => I (* Ce cas impose x = m+1,
                                   on doit donc demontrer True *)
           end
  (* Sans utiliser de definition intermediaire, on doit donner des
     informations supplementaires *)
.
```

Figure 3.6: Programmation avec types dépendants en Coq

ou non vérifiable est indécidable. On demande donc à l'utilisateur d'admettre cette égalité (impossible), et d'en déduire ce qu'il peut.

Ici, l'astuce est de dire :« Si 0 est de la forme $x+1$, je renvoie n'importe quoi, sinon je renvoie une preuve de $False$ ».

Comme 0 est de la forme 0, on en déduit alors une preuve de `False`, ce qui est le but recherché. Or, l'égalité `m +1 = 0` nous autorise à retourner n'importe quoi (ici une preuve de `True`), et au final, on a bien obtenu la preuve recherchée.

indifférence aux preuves Considérons une fonction au sens mathématique f de $\{x{\in}\mathbb{N}|x > 0\}$ dans $\mathbb{N}$, alors on a $f(3) = f(3)$. En Coq, une valeur de $\{x{\in}\mathbb{N}|x > 0\}$ est un couple d'un entier et d'une preuve que cet entier est strictement positif. A-t-on alors $f(3, \text{une preuve de } 3 > 0) = f(3, \text{une autre preuve de } 3 > 0)$?

Pour répondre à cette question, on s'intéresse à la fonction `predecesseur` donnée en figure 3.7.

Un premier essai retourne une erreur : on a voulu inspecter la preuve de stricte positivité et retourner l'entier contenu dans cette preuve. Le contenu du message est clair, on n'a pas le droit de regarder le contenu d'une preuve pour donner une valeur calculatoire. Il s'agit là d'un comportement similaire aux monades, une fois qu'on inspecte une preuve, on ne peut rien produire d'autre qu'une preuve.

Pour pouvoir produire une valeur, il faut donc inspecter l'entier donné. Si cet entier est strictement positif, on sait retourner son prédécesseur. Sinon, on se retrouve avec une preuve de $0 > 0$, on peut donc en déduire une preuve de `False`.

`False` est une propriété particulière, il n'est pas possible *a priori* de construire une valeur de ce type, et, pour cette raison, ce type fait partie des preuves qu'on s'autorise à inspecter pour produire une valeur. Il existe d'autres propriétés qui vérifient cette exception, elles sont données dans le manuel de référence de Coq [39].

tactiques La grosse différence entre Coq et Agda 2 est la possibilité de manipuler et de construire des termes à l'aide d'un langage spécialisé dans cette tâche, souvent de manière automatisée. Typiquement, le théorème $A \Rightarrow B \Rightarrow A$ est évident, devoir le prouver en écrivant `fun (a : A) (_ : B) => a` est un peu long alors qu'il suffirait de demander à l'assistant de preuve de trouver la preuve par lui même. Les preuves tendent aussi à être longues si elles ne sont pas artificiellement découpées et il est parfois dur de s'y retrouver.

Il existe en Coq deux langages spécialisés dans l'écriture de preuve. Le premier (figure 3.8), est un langage assez naturel, mais verbeux et peu utilisé. Le second (figure 3.9) est le langage de tactiques qui donne une impression d'obfuscation et qui n'est pas intelligible pour de longues preuves sans utilisation de l'interpréteur Coq. La bibliothèque standard de Coq propose plus d'une centaine de tactiques.

La logique de Coq

Coq n'implémente pas la logique classique, mais la logique intuitionniste au travers du calcul des constructions inductives.

```
Definition predecesseur : forall n, strictp n -> nat :=
  fun n strp => match strp with PosStr m => m end
.
(*
>>>
Error:
Incorrect elimination of "strp" in the inductive type "strictp":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop
is not allowed on a predicate in sort Set
because proofs can be eliminated only to build proofs.
*)
Definition predecesseur : forall n, strictp n -> nat :=
  fun n => match n as x return strictp x -> nat
           with | 0 => fun absurd =>
                         match strictp_0_absurde absurd with end
                | S m => fun _ => m
           end
.
```

Figure 3.7: Indifférence aux preuves en Coq

```
Lemma predecesseur_specification
 : forall n H, eq _ (S (predecesseur n H)) n.
proof.
  let n be such that (strictp n).
  per cases on _hyp.
    suppose it is (PosStr m).
      thus thesis.
  end cases.
Qed.

Print predecesseur_specification.
(*
>>>
fun (n : nat) (_hyp : strictp n) =>
(fun subcase_
 : forall m : nat, (predecesseur (m) +1 (PosStr m)) +1 = (m) +1 =>
 match _hyp as s in strictp n0 return (predecesseur n0 s) +1 = n0 with
 | PosStr n0 => subcase_ n0
 end)
  (fun m : nat =>
   (fun _fact : (predecesseur (m) +1 (PosStr m)) +1 = (m) +1 => _fact)
     (eq_refl nat (m) +1))
     : forall (n : nat) (H : strictp n), (predecesseur n H) +1 = n
*)
```

Figure 3.8: Le langage mathématique de Coq

```
Lemma predecesseur_specification2: forall n H, S (predecesseur n H)=n.
Proof.
  intros _ [m]. (* fun _ (PosStr m) => ... *)
  split.        (* eq_refl _ _ (m +1) *)
Qed.
Print predecesseur_specification2.
(*
>>>
fun (_tmp : nat) (H : strictp _tmp) =>
match H as s in strictp n return (predecesseur n s) +1 = n with
| PosStr m => eq_refl nat (m) +1
end
     : forall (n : nat) (H : strictp n), (predecesseur n H) +1 = n
*)
```

Figure 3.9: Le langage de tactiques de Coq

En particulier le raisonnement par l'absurde n'est *a priori* pas exploitable en Coq sans avoir recours à des axiomes supplémentaires. Une des raisons est que l'axiome du tiers exclu entre en contradiction avec d'autres axiomes que certains logiciens utilisent. Une autre raison est que l'on peut vouloir faire preuves constructives de théorèmes.

À titre d'exemple, voici trois énoncés qui sont démontrables classiquement, mais dont la preuve intuitionniste n'est pas évidente, voire impossible :

- la loi de Peirce : « $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ ». Cet énoncé est équivalent à l'axiome du tiers exclu (toute proposition est soit vraie soit fausse), et n'est pas démontrable en logique intuitionniste.
- le paradoxe des buveurs : « Dans tout bar non vide, il existe une personne telle que si cette personne boit alors tout le monde boit. » Le fait que cet énoncé soit ou non démontrable en logique intuitionniste dépend de ce qu'on appelle un bar ; si on sait par exemple qu'un bar ne contient jamais plus de mille personnes, alors c'est démontrable (mais la preuve pourra être très longue). Dans le cas général, quand il y a une infinité de personnes, l'énoncé n'est plus démontrable en logique intuitionniste sans axiome supplémentaire.
- « Il existe deux nombres irrationnels a et b tels que a^b soit rationnel. » La preuve classique consiste à considérer $b = \sqrt{2}$ [8] ; si b^b est rationnel, alors $a = b$ donne une solution ; sinon $a = b^b$ est solution (on a alors $a^b = 2$). Cette démonstration est typiquement non constructive, on peut trouver b, mais on hésite entre deux valeurs possibles pour a. En réalité, des théorèmes d'analyse permettent de savoir laquelle des deux possibilités est la bonne [9], mais les preuves sont bien plus complexes.

Parmi les autres axiomes des mathématiques classiques qui ne font pas partie de la théorie de Coq, on peut citer :

- L'axiome du choix :

```
Inductive habite : Type -> Prop := Hab : forall T, T -> habite T.
```

8. on rappelle que $\sqrt{2}$ est irrationnel

9. Le théorème de Gelfond-Schneider, qui par ailleurs apporte une solution au septième problème de Hilbert, permet de choisir $a = \sqrt{2}^{\sqrt{2}}$ et $b = \sqrt{2}$.

```
Axiom choix : forall A (T : A -> {U | habite U}),
              forall a, proj1_sig (T a).
```

« Le produit cartésien d'une famille d'ensembles tous non vides est toujours non vide. »

– L'axiome d'extensionnalité :

```
Axiom extensionnalite :
      forall A (T : A -> Type) (f g : forall a, T a),
             (forall a, f a = g a) -> f = g.
```

« Si deux fonctions calculent les mêmes choses, alors elles sont égales. »

– L'indifférence aux preuves (proof-irrelevance) :

```
Axiom indifference : forall (P : Prop) (p q : P), p = q.
```

« Deux preuves d'un même énoncé sont toujours égales (comprendre indiscernables). »

L'ajout de tels axiomes doit se faire de façon prudente, car certains sont incompatibles entre eux ; c'est-à-dire qu'avec certaines combinaisons d'axiomes, on peut déduire une preuve de `False`, et par la même de n'importe quel énoncé ; une telle théorie est alors dite incohérente.

3.4.2 Extraction de programmes Coq vers Ocaml

Au travers de la correspondance de Curry-Howard [30], Coq est donc à la fois un langage de programmation et un langage de preuve. Il est alors naturel de vouloir traduire les programmes écrits en Coq vers des programmes exécutables. Comme le résultat de ces programmes est indépendant des preuves utilisées, dans des soucis d'efficacité, on veut pouvoir retirer les preuves de ces programmes. La seule connaissance du fait que les programmes ont été prouvés doit être suffisante pour s'assurer que les programmes sont corrects.

Une telle procédure existe, et a été entièrement réecrite dans la version 7.0 par Pierre Letouzey [37].

En reprenant sur la fonction prédécesseur, voilà ce que l'on obtient :

```
Recursive Extraction predecesseur.
(*
type nat =
| O
| S of nat

(** val predecesseur : nat -> nat **)

let predecesseur = function
| O -> assert false (* absurd case *)
| S m -> m

*)
```

Le type `nat` a été traduit de façon identique ; par contre l'argument de type `strictp n`, étant dans `Prop`, a été effacé et le filtrage sur un type vide (`strictp O`) a été remplacé par une levée d'exception (`assert false`). L'extraction de programmes Coq peut se faire vers Ocaml, Haskell et Scheme ; bien que Ocaml soit le mieux supporté à l'heure actuelle.

Voici quelques autres exemples d'extraction pour donner une idée du fonctionnement :

```
(* La version de False dans Set;
   comme le type vide n'existe pas en Ocaml,
   l'extraction se fait vers un type par defaut.
*)
Inductive empty : Set := .
Definition empty_fun (e : empty) : forall A, A :=
  match e with end.
Recursive Extraction empty_fun.
(* >>>
type empty = unit
let empty_fun e = assert false
*)

(* Quand le type ne contient qu'un seul constructeur,
   une optimisation de l'extraction enleve cet encapsulation;
   les parties dans Prop, comme 'n>0' sont egalement effacees
   du type.
*)
Inductive positive : Set := Pos : forall n, n > 0 -> positive.
Extraction positive. (* >>> nat *)

(* Ocaml n'implemente pas encore les GADTs, quand ce sera
   le cas, peut etre aura-t-on une meilleure traduction de
   ces types.
*)
Inductive bool_or_nat : Set -> Set :=
 | Bool : bool -> bool_or_nat bool
 | Nat : nat -> bool_or_nat nat
.
Extraction bool_or_nat.
(* type 'x bool_or_nat = | Bool of bool | Nat of nat *)

(* Enfin un exemple d'un type Coq intraduisible dans Ocaml;
   le module Obj d'Ocaml permet de creer des valeurs de
   n'importe quel type et de les interpreter comme etant
   de n'importe quel autre type. Le systeme de typage de
   Coq permet de s'assurer de la bonne utilisation du module.
*)
Definition weird_type (b : bool) := if b then bool else nat.
Definition f : forall b, weird_type b :=
  fun b => if b as b0 return weird_type b0 then true else 0.
Recursive Extraction f.
(*
type __ = Obj.t
```

```
type bool =
| True
| False

type nat =
| O
| S of nat

type weird_type = __

(** val f : bool -> weird_type **)

let f = function
| True -> Obj.magic True
| False -> Obj.magic O
*)
```

3.4.3 Procédures de décision et de vérification

Définition 3.4.1 (Décidabilité)**.** *Étant donné un type T et une propriété P portant sur les éléments du type T, on appelle procédure de décision tout algorithme retournant « vrai » chaque fois qu'un élément de type T qui vérifie la propriété P lui est donné, et qui retourne « faux » chaque fois qu'un élément de type T qui ne vérifie pas la propriété lui est donné.*

On dit alors que la propriété P est décidable.

En Coq, on appelle procédure de décision de la propriété $P : T \to Prop$ toute fonction de type $\forall(t : T), \{P(t)\} + \{P(t) \to False\}$. Une telle fonction décide bien de la propriété P comme défini ci-dessus, et fourni en plus une preuve de vérification (ou de non vérification) de la propriété.

En pratique cependant, on cherche souvent à savoir si une propriété P est vérifiée, mais on ne cherche pas toujours à savoir si elle n'est pas vérifiée. Il y a plusieurs raisons à cela :

- Souvent, si la propriété n'est pas vérifiée, on abandonne le processus dans lequel on s'est engagé. Une preuve de non vérification est alors inutile.
- Il arrive que la propriété considérée soit semi-décidable, c'est-à-dire que si elle est vérifiée on arrive toujours à le découvrir, mais si elle n'est pas vérifiée, on n'arrive pas toujours à produire une preuve de non vérification de la propriété. On ne peut alors pas définir de procédure de décision.
- Il arrive que la propriété soit indécidable en général, c'est-à-dire qu'il existe des cas pour lesquels on n'est pas capable de dire si oui ou non la propriété est vérifiée. Par contre, il est fréquent d'être capable de dire « vrai » pour une famille assez vaste de cas particuliers favorables et de dire « peut être » dans tous les autres cas (voire de dire « non » dans une famille assez vaste de cas particuliers qui ne vérifient pas la propriété).

Définition 3.4.2 (Vérifiabilité)**.** *Étant donné un type T et une propriété P portant sur les éléments du type T, on appelle procédure de vérification tout algorithme qui termine*

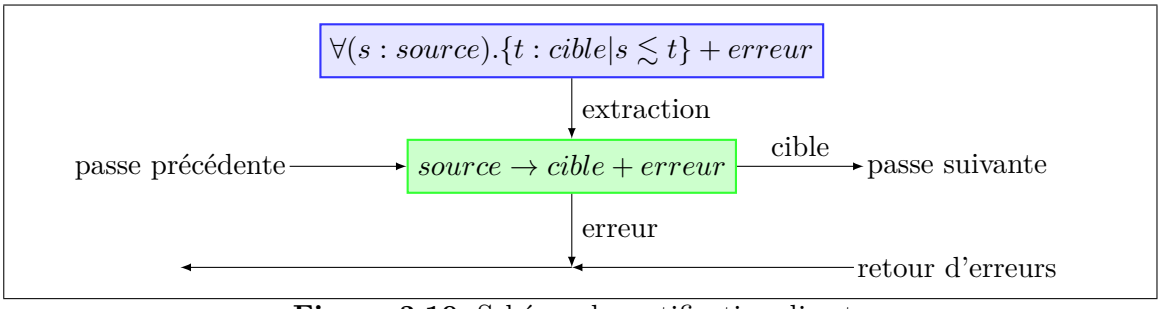


Figure 3.10: Schéma de certification directe

toujours et ne peut retourner « vrai » que si l'élément de type T passé en argument vérifie la propriété P, et qui ne peut retourner que « peut être » s'il ne retourne pas « vrai ».

Toute propriété est « vérifiable » en ce sens puisqu'il suffit de retourner systématiquement « peut être ». On réservera donc le qualificatif « vérifiable » aux propriétés pour lesquelles on dispose d'un algorithme qui vérifie la propriété P et qui a l'air de décider de la propriété (c'est-à-dire pour lesquels on a l'impression que les « peut être » ont lieu uniquement quand la propriété n'est pas vérifiée).

Dans cette thèse, on appelle procédure de vérification de la propriété $P : T \rightarrow Prop$ toute fonction de type $\forall(t : T), \{P(t)\} + erreur$. Une telle fonction vérifie bien la validité de la propriété P comme défini ci-dessus, et fourni en plus une preuve de vérification de la propriété.

La définition de décidabilité est une définition classique de la théorie de la calculabilité [34]. La définition de vérifiabilité est en revanche propre à cette thèse. Elle est différente de la semi-décidabilité en ce sens qu'un programme qui vérifie la propriété P peut être rejeté dans le cas de la vérifiabilité (quand « peut être » est retourné), mais pas dans le cas de la semi-décidabilité.

La vérifiabilité est donc sujette au principe suivant : « On peut rejeter en utilisant le bon sens, mais on ne peut accepter que si on sait prouver ».

3.5 Certificats et validations

Maintenant que le langage Coq a été introduit, nous allons voir comment ce langage peut être utilisé dans une optique de certification afin de produire un compilateur formellement prouvé.

3.5.1 Certification directe

On démontre que pour tout programme $n_s : source$, on peut construire un programme $n_t : cible$ tel que n_t préserve la sémantique de n_s, ou retourner une erreur. Ce principe est illustré en figure 3.10.

Retourner une erreur n'indique pas en soi que n_s ne peut avoir de sémantique ou de programme équivalent, mais simplement que dans la preuve, un passage a été jugé trop dur (éventuellement indécidable) pour savoir si on pouvait produire un programme équivalent.

Cette approche est conceptuellement la plus simple, cependant dans le cas général, la preuve de préservation de sémantique peut être difficile, et on peut avoir des problèmes de maintenabilité, car à la moindre modification du code du compilateur, on peut être amené à refaire entièrement la preuve.

Raffinement de la certification directe

Pour rendre le système plus facile à maintenir, et également faciliter certaines preuves, on peut alors recourir à une relation entre la syntaxe du programme source et celle du programme cible, notée ici $P(n_s, n_t)$. On attend deux propriétés de cette relation. La première est que si on se donne un programme n_s, on doit être en mesure de produire un programme n_t tel que $P(n_s, n_t)$, ou bien annoncer que l'on n'a pas pu produire de tel programme. La seconde est que si on se donne deux programmes n_s et n_t, la condition $P(n_s, n_t)$ implique que $n_s \lesssim n_t$.

Formellement ces deux propriétés s'expriment par :

$$\begin{array}{ll} (V_1) & \forall n_s.\ \{n_t | P(n_s, n_t)\} + erreur \\ (V_2) & \forall n_s, n_t. P(n_s, n_t) \Rightarrow n_s \lesssim n_t \end{array}$$

De ces deux propriétés, il est aisé d'en déduire $\forall n_s.\ \{n_t | n_s \lesssim n_t\} + erreur$.

Cette façon de procéder n'apporte rien d'un point de vue théorique à l'approche directe, comme on peut le voir en choisissant $P(n_s, n_t) = n_s \lesssim n_t$. Cependant elle permet de spécifier de façon plus précise quelles relations raisonnables doivent vérifier les codes sources et les codes cibles pour pouvoir facilement donner des garanties de la préservation de la sémantique. Voici quelques intérêts de cette variante :

- le système gagne en clarté ; il est plus facile de comprendre la spécification d'un programme que le programme lui même
- si la relation est bien choisie, la preuve de préservation est découpée en V_1 et V_2 qui sont plus simples.
- modifier le générateur de code peut n'obliger à retoucher que la preuve V_1 qui est souvent la plus facile, et non à refaire toute la preuve de préservation

et quelques inconvénients :

- si on veut changer suffisamment la fonction de traduction, alors elle peut produire du code qui ne vérifie plus la relation P. Il faut alors définir une nouvelle relation, et démontrer V_1 et V_2 pour cette relation.
- il y a besoin de faire deux preuves, qui peuvent se ressembler fortement et être redondantes

Dans les preuves que j'ai réalisées en Coq, toutes les certifications directes utilisent cette variante par choix.

3.5.2 Une forme de validation

La technique précédente présente cependant des inconvénients ; en voici quelques uns :

- La maintenance du code passe par une maîtrise de l'assistant de preuve utilisé.

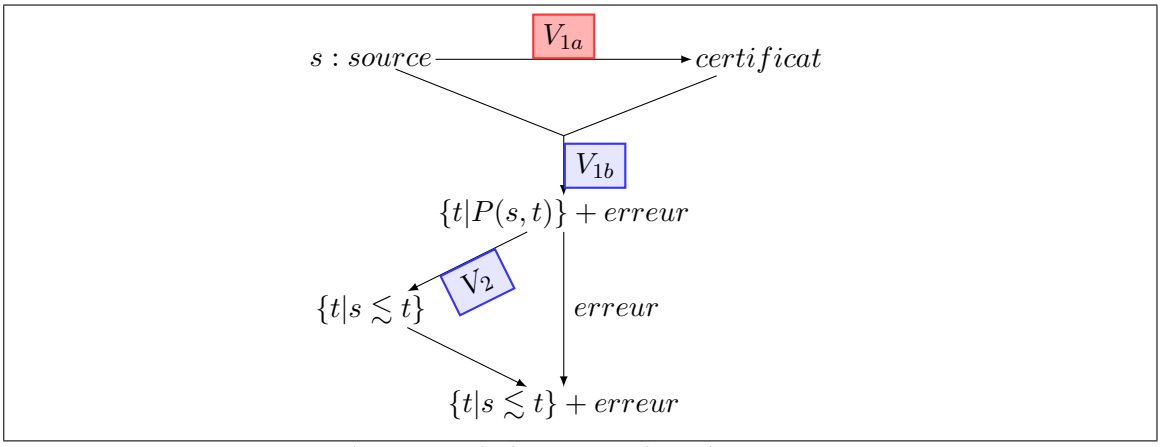


Figure 3.11: Découpage de la preuve de préservation en trois points

- Une amélioration des fonctions de compilation peut amener à modifier tout ou partie des preuves établies concernant le point (V_1), alors que la propriété P peut ne pas avoir changé (auquel cas le point (V_2) n'est pas affecté).
- Dans le cadre de la qualification, modifier le code amène à devoir tout requalifier ; pour cela il faut indiquer ce qui a été modifié et justifier ces choix ; c'est un processus long et coûteux.

Pour contrer certains de ces points, il suffirait de découper le point (V_1) en deux sous points (voir la figure 3.11) :

- (V_{1a}) être capable à partir d'un code source de produire des directives de compilations, c'est-à-dire des données qui vont aider l'assistant de preuve à produire le code cible à partir du code source. Par la suite ces directives seront appelées certificat, et peuvent être utilisées pour servir de trace à la compilation.
- (V_{1b}) être capable à partir du certificat de produire un code cible qui vérifie la relation P que l'on s'est fixée.

Maintenant, l'idée de la traduction par validation est que le point (V_{1a}) peut être écrit sans aucune spécification, par exemple dans le fragment F_ω de Coq. Comme l'extraction vers `Ocaml` de ce fragment est l'identité, on peut, de façon tout à fait sécurisée écrire le générateur de certificat directement en `Ocaml`. On gagne alors en maintenabilité, car le générateur de certificat n'a plus besoin d'être certifié, et n'importe qui peut le modifier sans aucune connaissance de l'assistant de preuve et sans avoir à faire requalifier toute l'architecture. Dans le pire des cas, le certificat donnera de mauvais indices qui aboutiront à l'échec de la compilation, mais en aucun cas ne produira de code faux en termes de préservation de la sémantique.

Ceci nous amène au diagramme 3.12.

- En rouge le code qui n'a pas besoin d'être certifié.
- En bleu, ce qui doit être prouvé.
- En vert, ce qui est automatique.

On parle de traduction par validation lorsque le certificat est égal au code produit. Le point (V_{1b}) consiste alors à être capable de dire si oui ou non la relation P est vérifiée entre deux programmes donnés ; c'est à peu de choses près le cadre de travail d'articles de Amir Pnueli, Ofer Strichman et Michael Siegel [50, 48] avec pour P la relation de

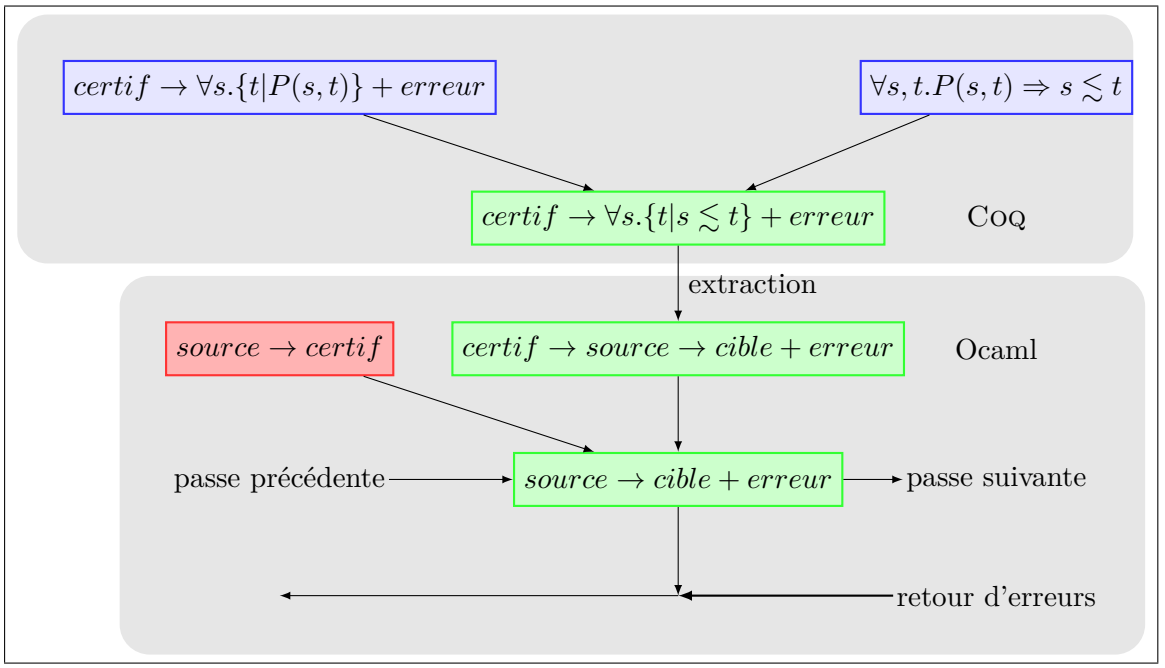


Figure 3.12: Schéma de compilation certifiée assistée par programme externe.

bisimulation.

Cette technique a été utilisée également dans le compilateur `CompCert`.

Par abus de langage, par la suite on parlera de traduction par validation, même dans le cas où le certificat est une directive et non pas directement le code produit.

3.5.3 Compilation assistée par serveur certifié

L'idée précédente d'avoir un outil externe qui permette de produire du code et de faire des preuves de préservation en se reposant sur des indices (les certificats) produits par l'outil externe peut être poussé encore un peu plus loin.

En effet, jusque là, on a présenté une façon de réaliser une passe de compilation, mais l'enchaînement des différentes peut imposer quelques lourdeurs. La figure 3.13 montre un enchaînement naturel. En pointillés apparaissent les connexions entre les différentes passes. Ces connexions n'ont pas été obtenues par extraction, et ont du être réalisées à la main. C'est une opération triviale, mais qui ne profite pas des garanties de sûreté d'un code extrait. Pour éviter ce problème, on peut réaliser l'enchaînement directement dans l'assistant de preuve, et fournir d'un coup au code extrait tous les outils, ce qui amènerait à la figure 3.14.

Une alternative intéressante à cette architecture est celle d'un système client-serveur (figure 3.19). Le client serait programmé en Ocaml, alors que le serveur lui serait en Coq.

Protocole Présentons d'abord le protocole client-serveur d'un compilateur assisté.

Le serveur doit maintenir un état interne dépendant du code source s ($ei(s)$), et im-

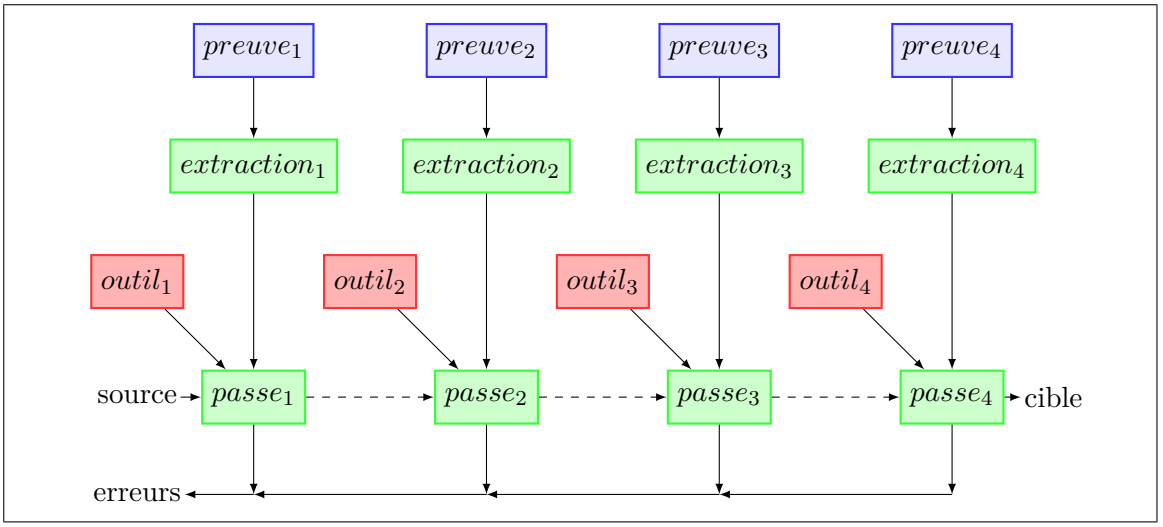


Figure 3.13: Enchainement possible des passes.

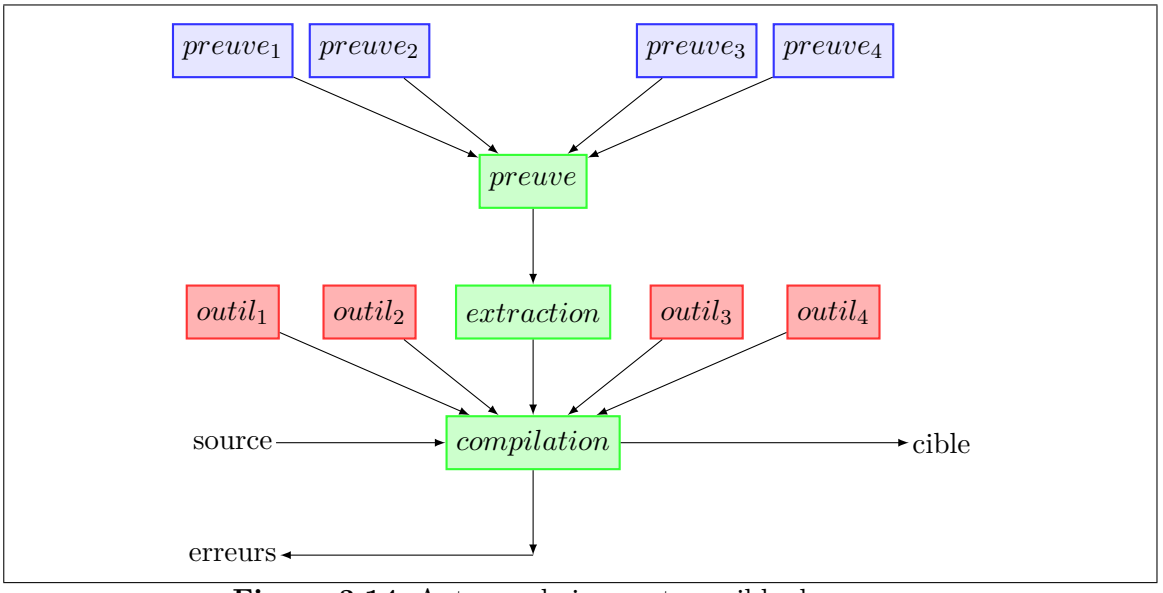


Figure 3.14: Autre enchainement possible des passes.

```
Definition requete : Type := ... (* le type des requetes *)
Definition reponse : Type := ... (* le type des reponses *)
```

Figure 3.15: Les types utilisés dans le protocole client-serveur

plémente un certain nombre de primitives de compilation avec leurs preuves associées.

Par exemple, le serveur peut être en mesure étant donnée une ligne de code l et un certificat c censé indiquer que la ligne l se trouve dans du code mort, de vérifier la validité du certificat c, de retirer la ligne l, et de montrer que le nouveau programme est équivalent à l'ancien.

Le client est l'outil externe qui va diriger la compilation. Il va donc devoir produire des requêtes en fournissant les certificats et les donner au serveur.

Le serveur doit ensuite essayer de satisfaire cette requête, et informer en retour le client si cette requête a échoué ou a réussi. Dans un cas comme dans l'autre, il peut éventuellement donner au client des informations supplémentaires sur son état actuel.

La figure 3.15 présente les deux types utilisés dans le protocole : le type `requete` qui décrit les arguments de chaque primitive, et le type `reponse` qui informe du déroulement de la requête précédente (succès ou échec, avec informations supplémentaires éventuelles).

Client Le client est donc un programme écrit en OCaml pour s'interfacer avec le code extrait du serveur.

Il doit donc commencer par une requête, puis répondre à chaque réponse du serveur par une nouvelle requête, ou indiquer la fin de la compilation.

Typiquement, on voudrait le représenter en Coq par :

```
Record client : Type :=
  { etatInterne : Type
  ; etatInitial : etatInterne
  ; requeteInitiale : requete
  ; prog : (reponse * etatInterne) -> option (requete * etatInterne)
           (* retourne None en fin de compilation *)
  }.
```

Cette représentation est possible, mais présente deux inconvénients.

Le premier inconvénient est que l'extraction va produire le type suivant :

```
type client = { etatInitial : Obj.t;
                requeteInitiale : requete;
                prog : (reponse * Obj.t) -> (requete * Obj.t) option
              }
```

Ce code ne s'identifie pas exactement avec celui de Coq. En particulier l'état interne n'est plus vraiment typé. La bonne utilisation du module `Obj` dans le code extrait est garanti par le système de typage de Coq, or ici, c'est le programmeur qui va devoir écrire le client en OCaml.

Le second inconvénient est que le système client-serveur ne pourra pas être écrit dans

```
Inductive client : Set :=
| Communication : requete ->
                  (reponse -> client) ->
                  client
| Fin : client.
```

Figure 3.16: Le type client écrit en Coq

Coq, car on ne donne aucune garantie que la composante `prog` du client termine toujours (en retournant `None`).

L'idée va donc être de transformer le type client en un type récursif. On voudrait donc avoir un type de la forme

```
requete * (reponse -> (requete * (reponse -> ...) option) option)
```

Un tel type cache l'état interne du client, ce qui évite le recours au module `Obj`, et peut s'écrire sous forme inductive, comme en figure 3.16, ce qui va permettre d'écrire le système client-serveur dans Coq. De plus, ce type s'extrait à l'identique en OCaml, ce qui permet de faire le lien entre le code Coq et le code OCaml sans difficulté supplémentaire.

Serveur Enfin, nous pouvons décrire le serveur de compilation.

Ce serveur a quatre composantes essentielles (en bleu sur la figure 3.19).

La première composante, *amorce*, est celle qui va charger le code s et produire l'état interne du serveur $ei(s)$. Une fois l'état interne initialisé, le serveur va soit recevoir l'ordre de terminer la compilation, soit recevoir une requête.

La seconde composante, $rafraichis$, est celle qui va mettre à jour l'état interne en fonction de la requête. En cas d'échec de la requête, le nouvel état interne peut être le précédent accompagné d'un rapport d'erreur. En cas de réussite, le nouvel état pourra simplement être la nouvelle représentation du code avec une preuve de préservation de sa sémantique.

La troisième composante, *reponds*, inspecte l'état interne pour voir si la requête précédente a réussi, et crée une réponse adéquate pour le client. Une fois la réponse émise, le serveur va recevoir une nouvelle requête ou une demande de fin de compilation.

La quatrième et dernière composante, *termine*, est utilisée lorsqu'une demande de fin de compilation est reçue. Si l'état interne contient du code cible avec la preuve de préservation de la sémantique du code source, le code cible peut être retourné. Sinon, le serveur peut retourner une erreur.

Ces quatre composantes sont récapitulées avec leur type en figure 3.17.

Le système client-serveur Toutes les pièces du système sont maintenant introduites, et le code du système client-serveur en Coq peut s'écrire simplement comme en figure 3.18.

Le type du client et le serveur extrait sont donnés en figure 3.20. Accessoirement, le client est plus facile à écrire en OCaml. En effet, en Coq se pose le problème de la terminaison ; il faut être en mesure de prouver que le client finit toujours par produire

```
(* L'etat interne du serveur, qui depend du code source initial *)
Definition ei : source -> Type := ...

(* Construire l'etat interne initial *)
Definition amorce : forall s, ei s := ...

(* Retourne le programme compile *)
Definition termine
: forall s, ei s -> { t : cible | s <~ t } + erreur := ...

(* La communication cote serveur *)
Definition rafraichis : requete -> forall s, ei s -> (ei s) := ...
Definition reponds : forall s, ei s -> reponse := ...
```

Figure 3.17: Les composantes du serveur

```
Definition serveur (s : source)
: client -> { t : cible | s <~ t } + erreur
:= (fix _serveur_ (e0 : ei s) c :=
    match c with
    | Communication req cc =>
      let e1 := rafraichis req s e0 in _serveur_ e1 (cc (reponds s e1))
    | Fin => termine s e0
    end) (amorce s).
```

Figure 3.18: Le système client-serveur en Coq

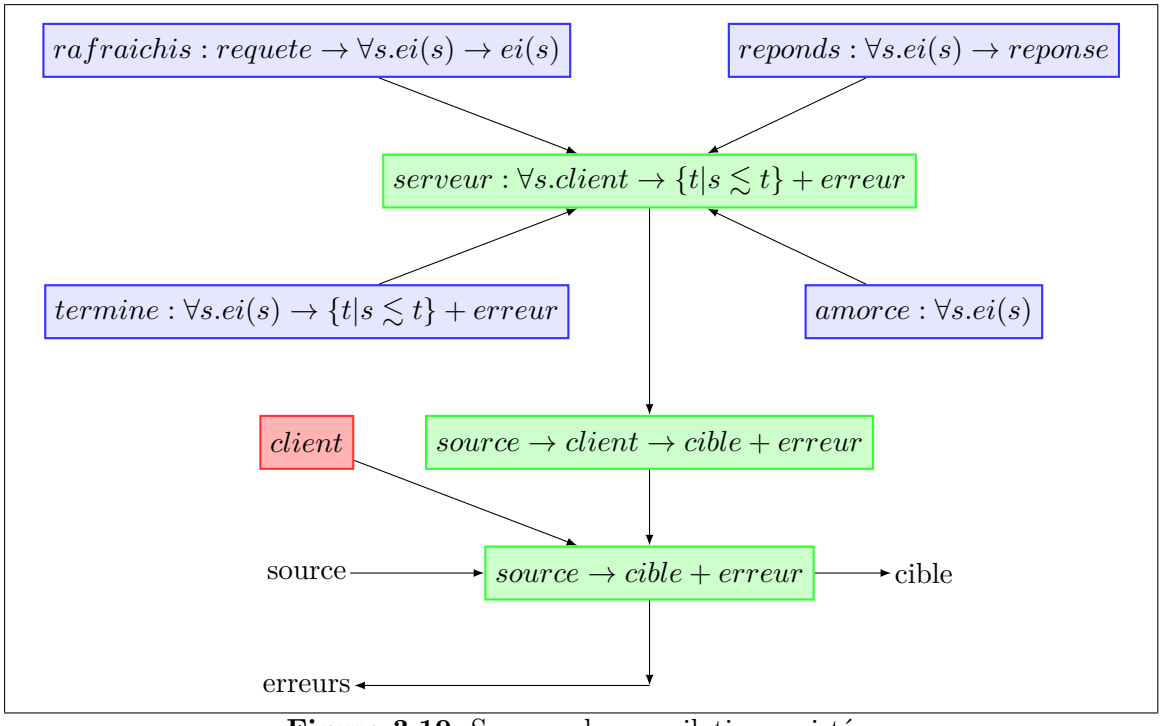


Figure 3.19: Serveur de compilation assistée.

```
type client =
| Communication of requete * (reponse -> client)
| Fin

let serveur s =
  let rec _serveur_ e0 = function
  | Communication (req, cc) ->
    let e1 = rafraichis req s e0 in _serveur_ e1 (cc (reponds s e1))
  | Fin -> termine s e0
  in _serveur_ (amorce s)
```

Figure 3.20: Type client, et code du serveur extraits vers OCaml.

`Fin`. En OCaml, cette restriction est levée ; sans pour autant poser de problème quant à la validité du code produit. En effet, si l'exécution du compilateur termine, alors le client aurait pu être écrit en Coq en utilisant une liste (potentiellement très longue) de toutes les requêtes produites par le compilateur. Si au contraire, l'exécution ne termine pas, il n'y a pas de problème puisqu'aucun code (ni aucune erreur) ne sera produit. Dans CompCert, comme alternative à ce problème, une boucle avec un nombre d'itérations maximal (très grand) est utilisé dans le module `Iteration`.

Un autre intérêt de cette architecture est du point de vue de la traçabilité de la compilation. En effet, on peut encapsuler un client dans un autre afin d'enregistrer la communication client-serveur dans un fichier.

```
let log (f : out_channel) =
        let rec _log_ c =
            match c with
            | Fin -> Fin
            | Communication (req, cc) ->
              let () = output_value f req in
              let ncc rep = output_value f rep; _log_ (cc rep) in
              Communication (req, ncc)
        in (_log_ : client -> client)
```

Enfin dernier intérêt, comme tout est centralisé, il est possible de définir la sémantique du langage source, celle du langage cible et une interface client-serveur dans un premier module qui doit être certifié. Les preuves et les définitions intermédiaires peuvent alors être définies dans des modules qui ne nécessitent pas de certification, et le client peut aussi être écrit en OCaml sans avoir besoin d'être certifié. Cette architecture est plus facile à faire évoluer et à maintenir, puisque seul un petit fragment a vraiment besoin de certification.

3.6 Conclusion

Nous avons défini dans cette partie tout le formalisme « haut niveau » dont nous aurons besoin par la suite, et expliqué dans quelle mesure on peut avoir confiance dans ce formalisme, ainsi que comment passer des preuves mathématiques aux programmes réels. Ce formalisme est assez léger et sa description ne fait que quelques centaines de lignes dans un assistant de preuves tel que Coq.

Les parties suivantes présenteront les aspects plus techniques, à savoir les vraies sémantiques et leurs preuves de préservation.

Deuxième partie

Le monde flot de données

Chapitre 4

Primitives flots de données

Afin de pouvoir exprimer le plus possible de programmes sur les flots de données, avec une mémoire bornée en taille, on va chercher à se donner un jeu d'opérateurs complets, mais minimal. Un chronogramme récapitulatif de toutes ces primitives est donné en figure 4.1.

Ces opérateurs (à l'exception de celui du séquençage de flots) sont issus de Lustre [1] (**lift**$^{\#}$ et **when**$^{\#}$) et Lucid Synchrone [51] (**fby**$^{\#}$ et **merge**$^{\#}$). L'opérateur **fby**$^{\#}$ était déjà employé dans le langage Lucid [61].

Comme énoncé en remarque 2.2.1, la représentation utilisée des flots par des listes n'est pas conventionnelle. Les valeurs les plus récentes se trouvent en tête de représentation alors que les plus vieilles se trouvent en bout de queue.

Ce choix a été fait pour simplifier l'écriture des fonctions récursives afin qu'elles n'aient plus d'accumulateur. Avoir un accumulateur permet du code souvent implémenté plus efficacement avec des techniques de compilation tirant profit de la récursion terminale. En revanche, pour ce qui est des preuves, il faut prendre en considération cet accumulateur et en trouver une bonne généralisation pour définir de bons invariants lors des inductions. Ces considérations d'efficacité ne concernent pas cette thèse cependant. En effet, la modélisation des flots est indépendante du compilateur, dont le temps d'exécution est lui même indépendant des performances du code compilé.

	b	Only	Only	Only	Only	Only	Only	Only
	$tics$	False	True	True	False	True	False	True
	$nat(b)$	0	1	2	3	4	5	6
	$\mathbf{lift}^{\#}_{(\boldsymbol{\lambda x.2})}\ (b)$	2	2	2	2	2	2	2
$nat_T =$	$\mathbf{when}^{\#}_{\boldsymbol{True}}\ (nat(b), tics)$	*abs*	1	2	*abs*	4	*abs*	6
$nat_F =$	$\mathbf{when}^{\#}_{\boldsymbol{False}}\ (nat(b), tics)$	0	*abs*	*abs*	3	*abs*	5	*abs*
$foo =$	$\mathbf{fby}^{\#}_{\mathbf{100}}\ (nat_T)$	*abs*	100	1	*abs*	2	*abs*	4
	$\mathbf{merge}^{\#}\ (\{nat_F, foo\}, tics)$	0	100	1	3	2	5	4

Figure 4.1: Chronogramme des primitives flot de données (hors séquençage)

4.1 Primitives sans mémoire

Définition 4.1.1 (relèvement point à point)**.** *Étant donnés un produit de types $\tau_1 \times ... \times \tau_n$, un type τ et une fonction f de $[\![\tau_1]\!] \times ... \times [\![\tau_n]\!]$ dans $[\![\tau]\!]$, on définit inductivement le relèvement point à point* $\mathbf{lift}^{\#}_{f}$ *comme une fonction partielle par :*

$$\mathbf{lift}^{\#}_{f} : [\![\tau_1]\!]^{abs^*} \times ... \times [\![\tau_n]\!]^{abs^*} \rightarrow [\![\tau]\!]^{abs^*}$$

$$\begin{array}{rcl} \mathbf{lift}^{\#}_{f}\ (\epsilon, ..., \epsilon) & = & \epsilon \\ \mathbf{lift}^{\#}_{f}\ (\psi_1 \cdot abs, ..., \psi_n \cdot abs) & = & \mathbf{lift}^{\#}_{f}\ (\psi_1, ..., \psi_n) \cdot abs \\ \mathbf{lift}^{\#}_{f}\ (\psi_1 \cdot \alpha_1, ..., \psi_n \cdot \alpha_n) & = & \mathbf{lift}^{\#}_{f}\ (\psi_1, ..., \psi_n) \cdot f\ (\alpha_1, ..., \alpha_n) \end{array}$$

Cette primitive permet de définir les flots d'une valeur constante k par $\mathbf{lift}^{\#}_{(\boldsymbol{\lambda x.k})}$, ou encore de relever les opérateurs importés.

Définition 4.1.2 (échantillonnage)**.** *Étant donnés deux types τ_1 et τ_2, deux flots ψ_1 de type τ_1 et ψ_2 de type τ_2, ainsi qu'une constante k de type τ_1, on définit inductivement l'échantillonnage* $\mathbf{when}^{\#}_{k}\ (\psi_2, \psi_1)$ *de ψ_2 par k sur ψ_1 comme une fonction partielle par :*

$$\mathbf{when}^{\#}_{k} : [\![\tau_2]\!]^{abs^*} \times [\![\tau_1]\!]^{abs^*} \rightarrow [\![\tau_2]\!]^{abs^*}$$

$$\begin{array}{rcll} \mathbf{when}^{\#}_{k}\ (\epsilon, \epsilon) & = & \epsilon & \\ \mathbf{when}^{\#}_{k}\ (\psi_2 \cdot abs, \psi_1 \cdot abs) & = & \mathbf{when}^{\#}_{k}\ (\psi_2, \psi_1) \cdot abs & \\ \mathbf{when}^{\#}_{k}\ (\psi_2 \cdot \alpha, \psi_1 \cdot k) & = & \mathbf{when}^{\#}_{k}\ (\psi_2, \psi_1) \cdot \alpha & \\ \mathbf{when}^{\#}_{k}\ (\psi_2 \cdot \alpha_2, \psi_1 \cdot \alpha_1) & = & \mathbf{when}^{\#}_{k}\ (\psi_2, \psi_1) \cdot abs & \textit{si } k \neq \alpha_1 \end{array}$$

Cette primitive permet de définir des modes, c'est à dire des états dans lesquels on calcule qui n'interfèrent pas avec les autres modes. Le fait de pouvoir échantillonner des flots déjà échantillonnés permet de définir des sous modes.

Cependant l'échantillonnage est inutile si on ne dispose pas de son dual pour revenir sur le flot de base en remplaçant des valeurs absentes par des valeurs présentes.

Il existe plusieurs solutions. Dans Lustre on a recours à un opérateur noté `current` qui remplaçait les valeurs absentes par la dernière valeur présente. Deux problèmes apparaissent alors, le premier est que cela introduit une mémoire par utilisation de l'opérateur ; le second est que les premières valeurs absentes ne peuvent pas être remplacées par leur valeur présente précédente (qui n'existe pas).

Pour y remédier, on utilisera plutôt une primitive de fusion de flots qui entrelace les valeurs de plusieurs petits flots en un seul gros flot.

Définition 4.1.3 (horloges disjointes)**.** *Étant donnés deux flots de même type ψ_1 et ψ_2, on dit qu'ils sont sur des horloges disjointes, si à tout instant leur valeur ne peut être simultanément présente.*

Exemple 4.1.1. *Si on considère les flots :*
- $\psi_0 = \epsilon \cdot abs \cdot abs$

– $\psi_1 = \epsilon \cdot 1 \cdot abs$
– $\psi_2 = \epsilon \cdot abs \cdot 2$
– $\psi_3 = \epsilon \cdot 1 \cdot 2$

Le flot ψ_0 et tout autre flot de l'exemple sont sur des horloges disjointes.

Le flot ψ_3 et tout autre flot de l'exemple (sauf ψ_0) ne sont pas sur des horloges disjointes.

Les flots ψ_1 et ψ_2 sont sur des horloges disjointes.

Définition 4.1.4 (fusion de flots)**.** *Étant donné un type τ de constructeurs k_1, ..., k_n, un flot ψ_τ de ce type et une famille $(\psi_{k_i})_{k_i \in [\![\tau]\!]}$ de flots de même type τ_0 et deux à deux sur des horloges disjointes, on définit inductivement la fusion de cette famille de flots le long de ψ_τ,* $\mathbf{merge}^{\#}\left((\psi_{k_i})_{k_i \in [\![\tau]\!]}, \psi_\tau\right)$ *comme une fonction partielle par :*

$$\mathbf{merge}^{\#} : \left([\![\tau_0]\!]^{abs^*}\right)^n \times [\![\tau]\!]^{abs^*} \to [\![\tau_0]\!]^{abs^*}$$

$$\begin{aligned}
\mathbf{merge}^{\#}\left((\psi_{k_i})_{k_i \in [\![\tau]\!]}, \epsilon\right) &= \epsilon \\
&\quad si\ \forall k_i \in [\![\tau]\!].\, \psi_{k_i} = \epsilon \\
\mathbf{merge}^{\#}\left((\psi_{k_i})_{k_i \in [\![\tau]\!]}, \psi_\tau \cdot abs\right) &= \mathbf{merge}^{\#}\left((\psi'_{k_i})_{k_i \in [\![\tau]\!]}, \psi_\tau\right) \cdot abs \\
&\quad si\ \forall k_i \in [\![\tau]\!].\, \psi_{k_i} = \psi'_{k_i} \cdot abs \\
\mathbf{merge}^{\#}\left((\psi_{k_i})_{k_i \in [\![\tau]\!]}, \psi_\tau \cdot k_j\right) &= \mathbf{merge}^{\#}\left((\psi'_{k_i})_{k_i \in [\![\tau]\!]}, \psi_\tau\right) \cdot \alpha \\
&\quad si \bigwedge \begin{cases} \forall k_i \in [\![\tau]\!].\, k_i \neq k_j \Rightarrow \psi_{k_i} = \psi'_{k_i} \cdot abs \\ \psi_{k_j} = \psi'_{k_j} \cdot \alpha \end{cases}
\end{aligned}$$

Le second argument de cette primitive est un flot qui sélectionne la valeur à prendre, à la manière d'un `switch` en C.

4.2 Primitives avec mémoire

La première primitive avec mémoire dont on ait besoin est celle qui permet de garder en mémoire la dernière valeur présente du flot. En Lustre, cette primitive est celle de l'opérateur `pre`. Elle a cependant un léger inconvénient du point de vue de la compilation ; en effet, elle "vide" son premier instant présent. Le compilateur doit alors s'assurer que son emploi se fait dans une construction qui re-remplit ce premier instant présent ; c'est pourquoi il est courant de rencontrer `0 -> pre (f x)`. Pour contourner cette difficulté, on impose au programmeur d'initialiser systématiquement ces décalages. La primitive de décalage demande alors une constante comme argument.

Définition 4.2.1 (décalage)**.** *Étant donnés un type τ, une constante k un flot ψ de type τ_1 , on peut définir le décalage de ψ initialisé à k comme une fonction partielle par :*

$$\mathbf{fby}_{\boldsymbol{k}}^{\#} : [\![\tau]\!]^{abs^*} \to \left([\![\tau]\!]^{abs^*} \times [\![\tau]\!]\right)$$

$$\begin{aligned}
\mathbf{fby}_{\boldsymbol{k}}^{\#}\ (\epsilon) &= (\epsilon, k) \\
\mathbf{fby}_{\boldsymbol{k}}^{\#}\ (\psi_1 \cdot abs) &= (\psi_2 \cdot abs, k_0)\ \textit{ avec } \mathbf{fby}_{\boldsymbol{k}}^{\#}\ (\psi_1) = (\psi_2, k_0) \\
\mathbf{fby}_{\boldsymbol{k}}^{\#}\ (\psi_1 \cdot k_1) &= (\psi_2 \cdot k_0, k_1)\ \textit{ avec } \mathbf{fby}_{\boldsymbol{k}}^{\#}\ (\psi_1) = (\psi_2, k_0)
\end{aligned}$$

Avec toutes ces primitives, on retrouve la puissance des systèmes d'équations de Lustre. On va cependant ajouter une nouvelle primitive de nature différente des autres, qui réassemble une liste de flots non étendus en un flot étendu selon un autre flot qui contrôle l'insertion des absences.

Définition 4.2.2 (séquençage)**.** *Étant donnés deux type τ_1 et τ_2, une liste de flots non étendus l de type τ_1, un flot étendu ψ de type τ_2 et une constante k de type τ_2, on définit inductivement le séquençage de l tous les k par ψ comme une fonction partielle par :*

$$\mathbf{seq}_{\boldsymbol{k}}^{\#} : (\llbracket\tau_1\rrbracket^*)^* \times \llbracket\tau_2\rrbracket^{abs^*} \rightarrow \llbracket\tau_1\rrbracket^{abs^*}$$

$$\begin{aligned}
\mathbf{seq}_{\boldsymbol{k}}^{\#}\ ([\epsilon], \epsilon) &= \epsilon \\
\mathbf{seq}_{\boldsymbol{k}}^{\#}\ (l, \psi_1 \cdot abs) &= \mathbf{seq}_{\boldsymbol{k}}^{\#}\ (l, \psi_1) \cdot abs \\
\mathbf{seq}_{\boldsymbol{k}}^{\#}\ (l \cdot (m \cdot \alpha), \psi_1 \cdot k_0) &= \mathbf{seq}_{\boldsymbol{k}}^{\#}\ (l \cdot m, \psi_1) \cdot \alpha\ \textit{ avec } k \neq k_0 \\
\mathbf{seq}_{\boldsymbol{k}}^{\#}\ (l \cdot (\epsilon \cdot \alpha), \psi_1 \cdot k) &= \mathbf{seq}_{\boldsymbol{k}}^{\#}\ (l, \psi_1) \cdot \alpha
\end{aligned}$$

L'idée de cette définition est de concaténer la liste de flots non étendus en un seul flot non étendu, puis d'insérer les absences du flot de contrôle, de sorte que le flot obtenu soit sur la même horloge que le flot de contrôle. Cette définition est suffisamment contraignante pour être inversible ; si on connaît le résultat du séquençage et le flot séquenceur, on peut reconstruire de façon unique la liste des flots séquencés.

Exemple 4.2.1. *Les flots $\phi_1 = \epsilon \cdot a \cdot b$; $\phi_2 = \epsilon \cdot p$ et $\phi_3 = \epsilon \cdot x \cdot y \cdot z$ sont séquençables tous les $True$ par $\psi = \epsilon \cdot abs \cdot False \cdot False \cdot abs \cdot True \cdot True \cdot abs \cdot False \cdot False$:*

$$\mathbf{seq}_{\boldsymbol{True}}^{\#}\ ((\phi_1, \phi_2, \phi_3), \psi) = \epsilon \cdot abs \cdot a \cdot b \cdot abs \cdot p \cdot x \cdot abs \cdot y \cdot z$$

Contre-exemple 4.2.1. *Les flots $\phi_1 = \epsilon \cdot a \cdot b$ et $\phi_2 = \epsilon \cdot p$ ne sont pas séquençables tous les $True$ par $\psi = abs \cdot False \cdot True \cdot abs \cdot False$, car ψ impose que le premier flot ait exactement une seule valeur (car il n'y a qu'une seule valeur présente avant la première occurrence de $True$).*

Définition 4.2.3 (Projeté d'un flot non étendu vers un flot étendu)**.** *Étant donné un flot non étendu ϕ, on définit son projeté ϕ^{abs} comme étant le flot étendu qui ne contient que des valeurs présentes et qui sont celles de ϕ. Formellement, on a donc :*

$$\begin{aligned}
\epsilon^{abs} &= \epsilon \\
(\phi \cdot \alpha)^{abs} &= \psi \cdot \alpha\ \textit{ avec } \psi = \phi^{abs}
\end{aligned}$$

4.3 Conclusion

Ce chapitre a exprimé un jeu de primitives pour être en mesure d'encoder toute machine à états finis. Avec ces primitives, il est maintenant possible de donner une sémantique à tous les langages flots de données de la chaîne de compilation.

Toutes ces primitives satisfont à une certaine propriété de morphisme ; si F_i est une fonction des flots dans les flots qui se contente d'insèrer une absence à l'instant i, alors :

- $\mathbf{lift}^{\#}_{\boldsymbol{op}}\,(F_i(\psi_1),...,F_i(\psi_n)) = F_i(\mathbf{lift}^{\#}_{\boldsymbol{op}}\,(\psi_1,...,\psi_n))$
- $\mathbf{when}^{\#}_{\boldsymbol{k}}\;(F_i(\psi_2),F_i(\psi_1)) = F_i(\mathbf{when}^{\#}_{\boldsymbol{k}}\;(\psi_2,\psi_1))$
- $\mathbf{merge}^{\#}\;\left(\left(F_i(\psi_{k_j})\right)_{k_j\in[\![\tau]\!]},F_i(\psi_\tau)\right) = F_i(\mathbf{merge}^{\#}\;\left(\left(\psi_{k_j}\right)_{k_j\in[\![\tau]\!]},\psi_\tau)\right)$
- $\mathbf{fby}^{\#}_{\boldsymbol{k_1}}\;(F_i(\psi_1)) = (F_i(\psi_2),k_2)$ avec $\mathbf{fby}^{\#}_{\boldsymbol{k_1}}\;(\psi_1) = (\psi_2,k_2)$
- $\mathbf{seq}^{\#}_{\boldsymbol{k}}\;(l,F_i(\psi)) = F_i(\mathbf{seq}^{\#}_{\boldsymbol{k}}\;(l,\psi))$

On a une propriété similaire pour le retrait d'absence à instants donnés.

Cette propriété est une propriété clé des langages synchrones à temps logique, on veut toujours pouvoir plonger des flots sur des échelles de temps plus fines (en insérant des absences) ou plus grossières (en enlevant des absences) sans en altérer la sémantique.

Le jeu de primitives présentées est complet pour exprimer toute machine à états finis selon la définition 3.1.5 avec $N \leq 1$.

En effet la suite $(w_n)_{n\in\mathbf{N}}$ calculée à partir de la suite $(u_n)_{n\in\mathbf{N}}$ par une machine à états finis pour $N \leq 1$ est définie par le système suivant :

$$\left\{\begin{array}{rcl}(s',w) & = & \mathbf{lift}^{\#}_{\boldsymbol{etape}}(s,u)\\ s & = & \mathbf{fby}^{\#}_{\boldsymbol{s_0}}\;(s')\end{array}\right.$$

Ce système définit également $(s_n)_{n\in\mathbf{N}}$, la suite des états lors de l'exécution et $(s'_n)_{n\in\mathbf{N}}$, la suite des états après le premier état (donc s privé de l'état initial s_0).

Les primitives de fusion et d'échantillonage de flots permettent de définir hiérarchiquement plusieurs machines à états finis qui ne seraient pas actives simultanément, et de combiner leurs sorties en une seule. L'entrelacement des exécutions de deux machines M_1 et M_2 selon un flot booléen $(b_n)_{n\in\mathbf{N}}$ et sur une entrée $(u_n)_{n\in\mathbf{N}}$ est défini par $(w_n)_{n\in\mathbf{N}}$, solution du système :

$$\left\{\begin{array}{rcl}u_{True} & = & \mathbf{when}^{\#}_{\boldsymbol{True}}\;(u,b)\\ w_{True} & = & M_1(u_{True})\\ u_{False} & = & \mathbf{when}^{\#}_{\boldsymbol{False}}\;(u,b)\\ w_{False} & = & M_2(u_{False})\\ w & = & \mathbf{merge}^{\#}\;(\{w_{True},w_{False}\},b)\end{array}\right.$$

La primitive de séquençage permet d'abandonner complètement l'exécution d'une machine à états finis, et de l'exécuter à nouveau sur son état initial. Certains langages de programmation, tels que Lustre ne proposent pas cette fonctionnalité. Il faut alors concevoir des machines à états finis admettant une entrée supplémentaire qui commande la réinitialisation. Il faut faire attention à ce que le système, après exécution de la routine de réinitialisation soit bel et bien dans l'état initial. Étant donnée une machine M, une suite d'entrées $(u_n)_{n\in\mathbf{N}}$, et une suite de réinitialisations $(b_n)_{n\in\mathbf{N}}$, on peut définir la suite $(w_n)_{n\in\mathbf{N}}$ des valeurs calculées par M sur $(u_n)_{n\in\mathbf{N}}$ réinitialisé tous les $(b_n)_{n\in\mathbf{N}}$ comme solution du

système :

$$\begin{cases} \mathbf{seq}^{\#}_{\boldsymbol{True}}\ ((x_n)_{n\in\mathbf{N}}, b) &=& u \\ \forall i.\ y_i &=& M(x_i) \\ w &=& \mathbf{seq}^{\#}_{\boldsymbol{True}}\ ((y_n)_{n\in\mathbf{N}}, b) \end{cases}$$

Il n'y a donc pas lieu d'ajouter de nouvelle primitive, mais si cela devait se faire, il serait naturel d'exiger de la nouvelle primitive qu'elle vérifie aussi la propriété de morphisme mentionnée ci-dessus.

Chapitre 5

Ls : syntaxe et sémantique

5.1 Syntaxe du langage Ls

Ls est le langage source de notre chaîne de compilation ; sa syntaxe simplifiée est donnée en figure 5.1. Les programmes écrits dans ce langage s'appellent des nœuds ; ils sont constitués de paramètres d'entrés, de paramètres de sorties, de déclarations de flots intermédiaires, et d'une suite d'équations mettant en relation des noms de flots avec des expressions utilisant les primitives de la section précédente.

Notations Afin de ne pas trop surcharger les notations, du sucre syntaxique sera utilisé.

- Les opérateurs arithmétiques usuels (`+,-,*,/`) seront infixes.
- `when Only(base)` pourra être omis.
- `when` x sera un raccourci pour `when True`(x).
- `when not` x sera un raccourci pour `when False`(x).
- `every` x sera un raccourci pour `every True`(x).
- `every not` x sera un raccourci pour `every False`(x).
- Quand une application de nœud n'est pas suivie de `every`, c'est qu'elle est implicitement suivie de `every False(True when ...)`.

Dans les exemples, des commentaires pourront figurer dans le code, ils commencent par `--` et se finissent à la fin de la ligne, comme en Haskell et en Lustre.

Les exemples suivants pourront également être des nœuds à plusieurs entrées et plusieurs sorties sur la même horloge, la restriction à un seul argument n'étant donnée que

n	::=	`node` f `(` x `:` τ `) returns (` x `:` τ `);`
		`var` x `:` τ `when` k `(` x `),` ..., x `:` τ `when` k `(` x `)`
		`let` sys `tel;`

sys	::=	eq `;` ...`;` eq
eq	::=	`(` x `,` ..., x `) =` e
e	::=	x | k | k `fby` e | o `(` e `)` | `(` e, ..., e `)` | f `(` e `)` `every` k `(` e `)` | `merge` x `(` k `->` e `)` ... `(` k `->` e `)` | e `when` k `(` x `)`

Figure 5.1: Syntaxe des nœuds et expressions dans Ls

```
-- type declarations
type mode = Left | Right
type button = Pushed | Released
type chess_clock
;--end of type declarations, begin of operators declarations

-- returns True iff 1st argument is Released and 2nd is Pushed
fun trigger: button * button -> bool

-- returns the opposite mode
fun flip: mode -> mode

-- compute a chess clock to display it, only the active part is given
fun make_chess_clock: mode * int32 * int32 -> chess_clock

;--end of operators declarations, begin of nodes implementations
```

Figure 5.2: Déclaration des types et opérateurs utilisés

pour des raisons de simplifications d'écriture.

Enfin, pour les définitions formelles portant sur les systèmes, le cas de base des inductions porte sur le système vide qui sera représenté par ε_s. Cette notation sera aussi utilisée dans les langages suivants de la chaîne de compilation.

5.1.1 Exemples

Tout au long des chapitres suivants, nous allons développer un exemple complet de programme qui prend en compte tous les aspects de cette compilation.

Ce programme est un petit système dont le but est d'afficher le temps de réflexion de deux joueurs d'échecs ; l'horloge doit donc mémoriser deux temps ; à tout moment un seul temps est actif (s'écoule) pendant que l'autre est passif (figé). Lorsque le premier temps est actif, on est dans le mode `Left` de l'horloge, et dans l'autre cas, dans le mode `Right`. Ce système contient un bouton qui peut avoir deux états (`Pushed` et `Released`) ; dès l'instant où le bouton passe de `Released` à `Pushed`, l'horloge change de mode.

Ce programme utilise les opérateurs `trigger`, `flip` et `make_chess_clock`, l'interface de notre programme correspond à la figure 5.2. `trigger` et `flip` auraient pu être programmés dans Ls comme illustré en figure 5.3, mais n'illustrent aucune particularité intéressante du point de vue de la compilation.

Tous les codes intermédiaires sont récapitulés en annexe, afin de retrouver rapidement le nœud source de la passe de compilation en cours.

Le programme en soi est donné en figure 5.4. Sa sémantique sera donnée plus loin.

```
node trigger (old : button, new : button) returns (trig : bool);
let trig = (old == Released) && (new == Pushed);
tel;

node flip (m : mode) returns (f : mode);
let f = merge m
          (Left -> Right when Left(m))
          (Right -> Left when Right(m));
tel;
```

Figure 5.3: Définition des opérateurs `trigger` et `flip`

```
-- some node to count time when active
node chrono_base (tic:unit) returns (time:int32);
let time = 0 fby (1+time);
tel;

-- to get the current mode
node get_mode (flip_button:button) returns (current_mode:mode);
var real_flip:bool;
    keep:mode when not real_flip; switch_:mode when real_flip;
let real_flip = trigger(Pushed fby flip_button, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
    current_mode =
      Left fby merge real_flip (False -> keep) (True -> switch_);
tel;

-- node to count time in minutes and seconds
node ms (tic:unit) returns (mins, secs: int32);
var reset:bool;
let mins = (chrono_base(tic) every reset)/60;
    secs = (chrono_base(tic) every reset)%60;
    reset = False fby ((mins==59) && (secs==59));
tel;

-- node to compute the chess clock in order to display it
node chess_clock (flip_button:button) returns (cc:chess_clock);
var current_mode:mode; m, s: int32;
let current_mode = get_mode(flip_button);
    (m,s) = merge current_mode
              (Left -> ms(Only when Left(current_mode)))
              (Right -> ms(Only when Right(current_mode)));
    cc = make_chess_clock (current_mode, m, s);
tel;
```

Figure 5.4: Code Ls de la pendule pour les échecs

5.2 Typage de Ls

Tous les programmes écrits dans la syntaxe précédemment présentée n'ont pas toujours de sens. Cependant, il existe des règles de typage qui sont là pour justifier le rejet de programmes dont on ne veut pas donner de sens.

Le typage n'ajoute pas d'annotation à l'arbre de syntaxe abstraite ; mais crée l'environnement statique local (ou échoue si le programme n'est pas bien typé).

Le bon typage est un gage fort que le programme ait un sens, mais ne le garantit pas ; en effet, il faut aussi s'assurer de la bonne fondation des définitions (ce qui s'appelle la causalité ; nous y reviendrons au chapitre 9).

L'idéal serait d'effectuer plusieurs analyses (non nécessairement certifiées) du code pour qu'on puisse espérer qu'aucune erreur ne soit provoquée par le compilateur, plutôt que d'accepter tous les programmes et de devoir échouer au milieu de la compilation ; car il faut alors pouvoir remonter les erreurs pour les expliquer afin que le programmeur les corrige.

Il est donc de bon ton de faire une passe d'analyse de causalité en plus du typage d'emblée, mais nous ne le faisons pas ici puisque ce n'est pas nécessaire.

Il n'y a pas d'inférence de types dans notre langage, puisque toute variable déclarée est déclarée avec son type ; le typage consiste donc à :

- lire toutes les déclarations de flots et ajouter leur type (horloge comprise) à l'environnement de typage local tout en vérifiant la cohérence avec l'environnement en cours
- lire toutes les équations du système et vérifier qu'elles satisfont toutes les contraintes de typage données en figures 5.5 et 5.6.

Définition 5.2.1 (environnement de typage local)**.** *Un environnement de typage $\mathcal{E}$ est la donnée :*

- *d'un domaine de noms de flots $dom_{\mathcal{E}}$*
- *d'une application qui à tout nom de flot x de $dom_{\mathcal{E}}$ associe son type* $\text{type}_{\mathcal{E}}(x)$ *dans* T
- *d'une application qui à tout nom de flot x de $dom_{\mathcal{E}}$ associe son horloge* $\text{ck}(x)$ *dans* CK
- *d'une preuve de :*

$$\texttt{base} \in dom_{\mathcal{E}}$$

- *d'une preuve de :*

$$\forall x \in dom_{\mathcal{E}}, x_{ck}, k_{ck}.\, \text{ck}(x) = \texttt{when}\ k_{ck}(x_{ck}) \Rightarrow \bigwedge \begin{cases} x_{ck} \in dom_{\mathcal{E}} \\ \text{type}_{\mathcal{E}}(x_{ck}) = \text{Ktype}(k_{ck}) \end{cases}$$

L'un des avantages de cette définition est de pouvoir manipuler des environnements intrinsèquement cohérents (dans le sens où un flot ne peut être filtré que par une constante du type du flot filtrant).

Exemple 5.2.1 (Bonne déclaration de variable)**.**

```
var x : int;
```

(C'est-à-dire `var x : int when Only(base);`*) définit l'environnement :*

$$\begin{cases} \text{type}_{\mathcal{E}}(\texttt{base}) = \texttt{unit} & \text{ck}(\texttt{base}) = \texttt{when Only}(\texttt{base}) \\ \text{type}_{\mathcal{E}}(x) = \texttt{int} & \text{ck}(x) = \texttt{when Only}(\texttt{base}) \end{cases}$$

Contre-exemple 5.2.1 (Mauvaise déclaration de variable)**.**

```
var x : int when base;
```

(C'est-à-dire `var x : int when True(base);`*) ne définit pas d'environnement, car on aurait d'une part* $\text{type}_{\mathcal{E}}(\texttt{base}) = \texttt{unit}$*, et d'autre part* $\text{Ktype}(\texttt{True}) = \texttt{bool}$*, ce qui contredirait une des preuves véhiculées par l'environnement.*

Un autre avantage est qu'en construisant progressivement l'environnement, on a la garantie que toutes les horloges sont bien fondées (c'est-à-dire sont issues de l'horloge de base), et qu'il n'y a pas de cycle de dépendances entre noms de flots (hormis le flot `base`).

5.2.1 Création de l'environnement de typage

L'environnement local de typage avant ajout des déclarations ne contient que le flot `base` d'horloge `when Only(base)` et de type `unit`. On vérifie aisément qu'il définit un environnement.

L'ajout d'une variable passe par une vérification que son nom n'est pas déjà défini, puis par une vérification que son horloge est compatible avec l'environnement en cours, et enfin par l'ajout de son nom, de son type et de son horloge à l'environnement (ainsi que les preuves données par les vérifications et celles déjà existantes).

On note $TEnv(i, o, l)$ l'environnement ainsi construit à partir des de l'entrée i, de la sortie o et des variables locales l et d'un environnement initial ne contenant que l'horloge de base.

5.2.2 Vérification du typage

La vérification du typage se fait dans le cadre d'un environnement de typage. Elle repose essentiellement sur quatre prédicats : un prédicat pour les expressions, un pour les équations, un pour les systèmes d'équations et un pour le nœud entier.

Vérification du typage pour une expression

Les expressions se doivent d'être toutes sur une même horloge [1].

La relation « l'expression e a pour produit de type π et pour horloge ck dans l'environnement $\mathcal{E}$ » est notée $\mathcal{E} \vdash_{\text{Ls}} e : \pi, \text{ck}$. Les règles d'inférences définissant cette relation sont données en figure 5.5.

Les constantes sont toutes sur l'horloge de base. Le type et l'horloge d'un nom de flot sont donnés directement par l'environnement de typage. Le décalage de flot ne change ni

1. Lever cette restriction n'aurait d'intérêt que si un nœud ou un opérateur pouvait renvoyer plusieurs flots sur des horloges différentes, comme c'est le cas en Lustre et Lucid Synchrone.

$$\textsc{Const}\ \frac{}{\mathcal{E} \vdash_{\text{Ls}} k : [\text{Ktype}\,(k)]\,, \texttt{when Only(base)}} \qquad \textsc{Var}\ \frac{x \in dom_{\mathcal{E}}}{\mathcal{E} \vdash_{\text{Ls}} x : [\text{type}_{\mathcal{E}}\,(x)]\,, \text{ck}\,(x)}$$

$$\textsc{Fby}\ \frac{\mathcal{E} \vdash_{\text{Ls}} e : [\text{Ktype}\,(k)]\,, \text{ck}}{\mathcal{E} \vdash_{\text{Ls}} k\ \texttt{fby}\ e : [\text{Ktype}\,(k)]\,, \text{ck}} \qquad \textsc{Op}\ \frac{O_{sig}\,(o) = \pi_i \to \tau_o \qquad \mathcal{E} \vdash_{\text{Ls}} e : \pi_i, \text{ck}}{\mathcal{E} \vdash_{\text{Ls}} o\texttt{(}e\texttt{)} : [\tau_o]\,, \text{ck}}$$

$$\textsc{Every}\ \frac{N_{sig}\,(f) = \tau_i \to \tau_o \qquad \mathcal{E} \vdash_{\text{Ls}} e_1 : [\tau_i]\,, \text{ck} \qquad \mathcal{E} \vdash_{\text{Ls}} e_2 : [\text{Ktype}\,(k)]\,, \text{ck}}{\mathcal{E} \vdash_{\text{Ls}} f\texttt{(}e_1\texttt{)}\ \texttt{every}\ k\texttt{(}e_2\texttt{)} : [\tau_o]\,, \text{ck}}$$

$$\textsc{When}\ \frac{x \in dom_{\mathcal{E}} \qquad \text{type}_{\mathcal{E}}\,(x) = \text{Ktype}\,(k) \qquad \mathcal{E} \vdash_{\text{Ls}} e : \pi, \text{ck}\,(x)}{\mathcal{E} \vdash_{\text{Ls}} e\ \texttt{when}\ k\texttt{(}x\texttt{)} : \pi, \texttt{when}\ k\,(x)}$$

$$\textsc{Tuple}\ \frac{\mathcal{E} \vdash_{\text{Ls}} e_1 : \pi_1, \text{ck} \qquad \mathcal{E} \vdash_{\text{Ls}} e_2 : \pi_2, \text{ck}}{\mathcal{E} \vdash_{\text{Ls}} e_1\texttt{,}\ e_2 : \pi_1 \circ \pi_2, \text{ck}}$$

$$\textsc{Merge}\ \frac{\begin{array}{c} x \in dom_{\mathcal{E}} \\ \forall \texttt{(}k_i\texttt{->}e_i\texttt{)} \in cases.\ \mathcal{E} \vdash_{\text{Ls}} e_i : \pi, k_i(x) \qquad \forall k \in \text{type}_{\mathcal{E}}\,(x)\,.\ \exists!e_k.\ \texttt{(}k\texttt{->}e_k\texttt{)} \in cases \end{array}}{\mathcal{E} \vdash_{\text{Ls}} \texttt{merge}\ x\ cases : \pi, \text{ck}\,(x)}$$

Figure 5.5: Règles de typage des expressions de Ls

le type ni l'horloge de l'expression. L'échantillonage ne change pas le type mais donne une horloge plus lente. La fusion de flots a le même type que chacun des flots fusionnés, et la même horloge que le flot de sélection.

Vérification du typage pour un nœud

Ls étant déclaratif, la syntaxe des équations la plus simple serait juste une paire d'expressions, et la vérification du typage consisterait en la vérification de l'égalité des deux types. Pour des raisons pratiques, l'expression gauche est en fait un tuple de noms de flots, et non une expression quelconque. Autoriser des expressions quelconques amènerait à faire soit des analyses complexes pour savoir si l'équation $x + y = z + t$ peut être transformée en $x = z + t - y$, $y = z + t - x$, $z = x + y - t$ ou $t = x + y - z$ afin de pouvoir calculer, soit à rejeter systématiquement ce genre d'équations.

Comme on ne veut pas définir plusieurs fois un même flot, on va également garder une trace des nom de flots définis.

$\mathcal{E} \vdash_{\text{Ls}} x = y : E$ signifie que l'équation $x = y$ est bien typée sous $\mathcal{E}$ et définit l'ensemble E de noms de flots.

$\mathcal{E} \vdash_{\text{Ls}} x = y; ...; z = t; : E$ signifie que le système d'équations $x = y; ...; z = t;$ est bien typé et définit l'ensemble E de noms de flots (avec définitions uniques).

$\vdash_{\text{Ls}} n : \sigma$ signifie que le nœud n est bien typé et définit un nœud de signature σ ;

$$\textsc{Eq}\ \frac{\forall 1 \leq i \leq n. \bigwedge \begin{cases} x_i \in dom_{\mathcal{E}} \\ \mathrm{ck}\,(x_i) = \mathrm{ck} \\ \forall 1 \leq j < i. x_i \neq x_j \end{cases} \qquad \mathcal{E} \vdash_{\textsc{Ls}} e : \mathrm{type}_{\mathcal{E}}\,(x_i)\,, ..., \mathrm{type}_{\mathcal{E}}\,(x_n)\,, \mathrm{ck}}{\mathcal{E} \vdash_{\textsc{Ls}} (x_1, ..., x_n) = e : \{x_1, ..., x_n\}}$$

$$\textsc{SysO}\ \frac{}{\mathcal{E} \vdash_{\textsc{Ls}} \varepsilon_s : \emptyset} \qquad \textsc{SysS}\ \frac{\mathcal{E} \vdash_{\textsc{Ls}} eq : E_{eq} \qquad \mathcal{E} \vdash_{\textsc{Ls}} sys : E_{sys} \qquad \emptyset = E_{eq} \cap E_{sys}}{\mathcal{E} \vdash_{\textsc{Ls}} eq;\ sys : E_{eq} \cup E_{sys}}$$

$$\textsc{Node}\ \frac{TEnv\,(i, o, l) \vdash_{\textsc{Ls}} sys : \{o, l_1, ..., l_r\}}{\vdash_{\textsc{Ls}} \begin{array}{l} \texttt{node}\ f\ \texttt{(}i : \tau_i\texttt{)} \\ \texttt{returns (}o : \tau_o\texttt{);} \\ \texttt{var}\ l_1 : \tau_{l_1}; ...; l_r : \tau_{l_r}\texttt{;} \\ \texttt{let}\ sys\ \texttt{tel;} \end{array} : (\tau_i, \tau_o)}$$

Figure 5.6: Règles de typage des nœuds de Ls

c'est-à-dire que son système est bien typé sous l'environnement de typage définit par les déclarations du nœud (entrée, sortie et locales) et définit exactement la déclaration de sortie et les déclarations locales. En particulier le bon typage interdit la définition d'une entrée, garantit que toutes les noms de flot utilisés ont été déclarés et interdit les définitions multiples d'un même nom de flot.

Les règles de typage pour les équations, systèmes et nœuds sont données en figure 5.6.

Lemme 5.2.1 (vérifiabilité du typage dans Ls)**.** *Le typage est vérifiable dans* Ls*. Plus précisément, on peut construire une fonction de signature*

$$\forall n.\ \{(\tau_i, \tau_o) | \vdash_{\textsc{Ls}} n : (\tau_i, \tau_o)\} + erreur$$

Démonstration. La preuve se fait sans difficulté particulière par induction sur la structure du nœud. La vérification du typage est dirigée par la syntaxe. □

5.3 Sémantique de Ls

La sémantique de Ls présuppose le bon typage du nœud ; elle repose sur un environnement sémantique global flot de données, et sur un environnement sémantique local qui est une extension de l'environnement de typage local.

Définition 5.3.1 (environnement sémantique local)**.** *Étant donné un environnement de typage local $\mathcal{E}$, un environnement sémantique local $\mathcal{S}^{\mathcal{F}}$ est la donnée :*

- *d'une application qui à tout nom de flot x associe sa sémantique* $\mathrm{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)$ *dans* $[\![\mathrm{type}_{\mathcal{E}}\,(x)]\!]^{abs^*}$
- *d'une preuve de :*

$$\forall i, x, x_{ck}, k_{ck}.\, \mathrm{ck}\,(x) = \texttt{when}\ k_{ck}\,(x_{ck}) \Rightarrow \left(\mathrm{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}\,(x)\,\langle i\rangle = abs \Leftrightarrow k_{ck} \neq \mathrm{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}\,(x_{ck})\,\langle i\rangle\right)$$

Ainsi on a une garantie intrinsèque que dans l'environnement, les flots nommés représentent bien des échantillonnages des flots nommés indiqués dans leur déclaration.

De plus, par construction de l'environnement de typage, tous les flots nommés sont issus de l'horloge de base et ont donc le même nombre d'instants par transitivité.

Sémantique des expressions

La sémantique des expressions est assez naturelle pour les constructions de base en utilisant les primitives flots de données ; elle est formellement définie en figure 5.7. Les constantes, opérateurs, noms de flots et décalages reprennent simplement leur primitive associée. La sémantique des tuples est triviale. L'échantillonnage distribue sa primitive à tous ses flots. La fusion de flots agrège les tuples de flots de chaque branche en un seul tuple de flots (le typage garantit que ce tuple a bien une horloge).

La réinitialisation est cependant plus complexe ; quand on séquence les flots d'entrées et les flots produits par une constante selon le flot de réinitialisation, on veut que chaque groupe d'entrée et sortie fassent partie de la sémantique du nœud considéré.

Sémantique des nœuds

La sémantique d'un nœud consiste en la vérification d'un système d'équations dans un environnement sémantique local compatible avec les flots d'entrée et les flots de sortie.

$\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} x = y$ signifie que l'équation $x = y$ est bien vérifiée sous $\mathcal{S}^{\mathcal{F}}$.

$\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} x = y; ...; z = t;$ signifie que le système d'équations $x = y; ...; z = t;$ est bien vérifié sous $\mathcal{S}^{\mathcal{F}}$.

$\vdash_{\text{Ls}} n \Downarrow (\phi_i, \phi_o)$ signifie que le nœud n admet ϕ_o comme sortie sur ϕ_i ; c'est-à-dire qu'il admet un environnement sémantique local pour lequel la valuation de l'entrée donne ϕ_i et la valuation de la sortie donne ϕ_o, qui vérifie toutes les équations du système.

La définition formelle de la sémantique des équations, des systèmes d'équations et des nœuds est donnée en figure 5.8.

5.3.1 Exemples

Le nœud `chrono_base` et la primitive `fby`

Ce nœud sert à compter le nombre de fois où son entrée est présente. Ici, la seule entrée possible est une liste de `Only`, qui est l'unique valeur de type `unit`.

Par exemple, sur cinq instants, on peut montrer qu'une sortie possible est $0, 1, 2, 3, 4$. En effet, en construisant l'environnement :

`base`	$\mapsto$	Only	Only	Only	Only	Only
`tic`	$\mapsto$	Only	Only	Only	Only	Only
`time`	$\mapsto$	0	1	2	3	4

$$\textsc{Const}\ \frac{}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} k \Downarrow \left[\mathbf{lift}^{\#}_{(\boldsymbol{\lambda x.k})}\left(\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(\texttt{base})\right)\right]} \qquad \textsc{Var}\ \frac{}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} x \Downarrow \left[\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)\right]}$$

$$\textsc{Op}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e \Downarrow \vec{\psi}}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} o\texttt{(}e\texttt{)} \Downarrow \left[\mathbf{lift}^{\#}_{\boldsymbol{O_{sem}(op)}}(\vec{\psi})\right]}$$

$$\textsc{When}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e \Downarrow \psi_0, ..., \psi_n \qquad \psi = \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e\ \texttt{when}\ k\texttt{(}x\texttt{)} \Downarrow \mathbf{when}^{\#}_{\boldsymbol{k}}\ (\psi_0, \psi), ..., \mathbf{when}^{\#}_{\boldsymbol{k}}\ (\psi_n, \psi)}$$

$$\textsc{Fby}\ \frac{\mathcal{S}^{\mathcal{F}} e \vdash_{\text{Ls}} [\psi_0] \Downarrow \qquad \mathbf{fby}^{\#}_{\boldsymbol{k_0}}\ (\psi_0) = (\psi_1, k_1)}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} k_0\ \texttt{fby}\ e \Downarrow [\psi_1]}$$

$$\textsc{Every}\ \frac{\begin{array}{c}\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_r \Downarrow [\psi_r] \qquad \mathbf{seq}^{\#}_{\boldsymbol{k}}\ (\psi_i, \psi_r) = \psi_{i,1}, ..., \psi_{i,m} \qquad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_i \Downarrow [\psi_i] \\ \mathbf{seq}^{\#}_{\boldsymbol{k}}\ (\psi_o, \psi_r) = \psi_{o,1}, ..., \psi_{o,m} \qquad \forall 1 \leq k \leq m.\quad (\psi_{i,k}, \psi_{o,k}) \in N^{\mathcal{F}}_{sem}(f)\end{array}}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} f\texttt{(}e_i\texttt{)}\ \texttt{every}\ k\texttt{(}e_r\texttt{)} \Downarrow [\psi_o]}$$

$$\textsc{Tuple}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_1 \Downarrow \vec{\psi_1} \qquad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_2 \Downarrow \vec{\psi_2}}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_1\texttt{,}\ e_2 \Downarrow \vec{\psi_1} \circ \vec{\psi_2}}$$

$$\textsc{Merge}\ \frac{\begin{array}{c}\forall 1 \leq i \leq n.\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_i \Downarrow \psi_{k_i,1}, ..., \psi_{k_i,m} \\ \forall 1 \leq j \leq m.\ \mathbf{merge}^{\#}\ \left((\psi_{k_i,j})_{k_i}, \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)\right) = \psi'_j\end{array}}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} \texttt{merge}\ x\ \texttt{(}k_1\texttt{->}e_1\texttt{)...(}k_n\texttt{->}e_n\texttt{)} \Downarrow \psi'_1, ..., \psi'_m}$$

Figure 5.7: Sémantique des expressions de Ls

$$\textsc{Eq}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e \Downarrow \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x_1), ..., \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x_n)}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} \texttt{(}x_1, ..., x_n\texttt{)} = e} \qquad \textsc{SysO}\ \frac{}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} \varepsilon_s}$$

$$\textsc{SysS}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} eq \qquad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} eq\texttt{;}\ sys}$$

$$\textsc{Node}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys \qquad \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(i) = {\phi_i}^{abs} \qquad \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(o) = {\phi_o}^{abs}}{\vdash_{\text{Ls}} \begin{array}{l}\texttt{node}\ f\ \texttt{(}i : \tau_i\texttt{)} \\ \texttt{returns (}o : \tau_o\texttt{);} \\ \texttt{var}\ l\texttt{;} \\ \texttt{let}\ sys\ \texttt{tel;}\end{array} \Downarrow (\phi_i, \phi_o)}$$

Figure 5.8: Sémantique des nœuds de Ls

```
-- some node to count time when active
node chrono_base (tic:unit) returns (time:int32);
let time = 0 fby (1+time);
tel;
```

Figure 5.9: Le nœud `chrono_base`

```
-- to get the current mode
node get_mode (flip_button:button) returns (current_mode:mode);
var real_flip:bool;
    keep:mode when not real_flip; switch_:mode when real_flip;
let real_flip = trigger(Pushed fby flip_button, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
    current_mode =
      Left fby merge real_flip (False -> keep) (True -> switch_);
tel;
```

Figure 5.10: Le nœud `get_mode`

le chronogramme suivant vérifie bien l'équation du système :

(t)	`time`	0	1	2	3	4
(i)	`1+`t	1	2	3	4	5
(t)	`0 fby`i	0	1	2	3	4

Le nœud `get_mode` et les primitives `when` et `merge`

Ce nœud reçoit en entrée la pression d'un bouton et retourne le mode en cours qui change après la pression du bouton (ce mode étant initialement `Left`).

Supposons que les joueurs appuient sur la pendule au premier instant, laissent appuyé au deuxième, relâchent au troisième, réappuient au quatrième et relâchent au cinquième.

Selon la spécification, il y aura changement de mode exactement aux instants deux et cinq ; en effet, l'environnement :

`base`	$\mapsto$	Only	Only	Only	Only	Only
`flip_button`	$\mapsto$	Pushed	Pushed	Released	Pushed	Released
`current_mode`	$\mapsto$	Left	Right	Right	Right	Left
`real_flip`	$\mapsto$	True	False	False	True	False
`keep`	$\mapsto$	*abs*	Right	Right	*abs*	Left
`switch_`	$\mapsto$	Right	*abs*	*abs*	Left	*abs*

```
-- node to count time in minutes and seconds
node ms (tic:unit) returns (mins, secs: int32);
var reset:bool;
let mins = (chrono_base(tic) every reset)/60;
    secs = (chrono_base(tic) every reset)%60;
    reset = False fby ((mins==59) && (secs==59));
tel;
```

Figure 5.11: Le nœud `ms`

donne lieu au chronogramme :

(f)	`flip_button`	Push.	Push.	Rel.	Push.	Rel.
(df)	`Pushed fby` f	Push.	Push.	Push.	Rel.	Push.
(rf)	`trigger(df, f)`	True	False	False	True	False
(c)	`current_mode`	Left	Right	Right	Right	Left
(sc)	c `when True(`rf`)`	Left	*abs*	*abs*	Right	*abs*
(s)	`flip(`sc`)`	Right	*abs*	*abs*	Left	*abs*
(k)	c `when False(`rf`)`	*abs*	Right	Right	*abs*	Left
(m)	`merge` rf `(True->`s`) (False->`k`)`	Right	Right	Right	Left	Left
(c)	`Left fby` c	Left	Right	Right	Right	Left

qui vérifie le système.

Le nœud `ms` et les appels de nœuds

Ce nœud sert à décomposer le temps écoulé en minutes et secondes. L'opérateur `%` est celui de calcul du modulo (reste de la division). L'opérateur `&&` est celui du calcul du « et » logique. L'opérateur `==` est le test d'égalité.

Là encore le système est vérifié avec l'environnement :

`base`	$\mapsto$	...	`Only`	`Only`	`Only`	`Only`	`Only`
`tic`	$\mapsto$	...	`Only`	`Only`	`Only`	`Only`	`Only`
`mins`	$\mapsto$	...	59	59	0	0	0
`secs`	$\mapsto$	...	58	59	0	1	2
`reset`	$\mapsto$	...	`False`	`False`	`True`	`False`	`False`

```
-- node to compute the chess clock in order to display it
node chess_clock (flip_button:button) returns (cc:chess_clock);
var current_mode:mode; m, s: int32;
let current_mode = get_mode(flip_button);
    (m,s) = merge current_mode
            (Left -> ms(Only when Left(current_mode)))
            (Right -> ms(Only when Right(current_mode)));
    cc = make_chess_clock (current_mode, m, s);
tel;
```

Figure 5.12: Le nœud `chess_clock`

avec comme chronogramme :

(c)	`tic`	...	Only	Only	Only	Only	Only
(r)	`reset`	...	False	False	True	False	False
(c)	$\mathbf{seq}^{\#}_{True}\ (c', r)$	...	Only	Only	Only	Only	Only
(c')		...	Only	Only][	Only	Only	Only
(t')	`chrono_base(`c'`)`	...	3598	3599][	0	1	2
(t)	$\mathbf{seq}^{\#}_{True}\ (t', r)$	...	3598	3509	0	1	2
(t)	`chrono_base(`c`) every` r	...	3598	3599	0	1	2
(m)	t`/60`	...	59	59	0	0	0
(s)	t`%60`	...	58	59	0	1	2
$(b1)$	m`==59`	...	True	True	False	False	False
$(b2)$	s`==59`	...	False	True	False	False	False
(b)	$b1$`&&`$b2$	...	False	True	False	False	False
(r)	`False fby` b	...	False	False	True	False	False

Les crochets au milieu du chronogramme marquent la fin d'un flot d'exécution et le début du suivant. Le flot c se voit découpé par r en une liste de flots c'. `chrono_base` est exécuté sur chacun des flots de c' pour produire une liste de flots t'. Les flots de t' sont alors concaténés selon r pour donner une sémantique à `chrono_base(`c`) every` r.

Le nœud `chess_clock`

Ce nœud permet la gestion des horloges de chaque joueur en parallèle, conformément au mode en cours et retourne un type abstrait construit sur le mode en cours et le temps (minutes et secondes) actif.

Supposons qu'à un instant donné, le joueur de gauche ait réfléchi en tout dix-sept minutes et dix-huit secondes secondes ; et que le joueur de droite ait réfléchi cinquante-neuf minutes et cinquante-huit secondes ; et que ce soit au joueur de gauche de jouer. Le joueur gauche appuie sur la pendule pendant deux secondes, puis une seconde plus tard le

joueur droit appuie dessus ; l'environnement à considérer est alors le suivant :

`base`	$\mapsto$	...	Only	Only	Only	Only	Only
`flip_button`	$\mapsto$	...	Rel.	Push.	Push.	Rel.	Push.
`m`	$\mapsto$	...	17	59	59	0	17
`s`	$\mapsto$	...	18	58	59	0	19

(`cc` a été omis par simplification).

Le chronogramme devient alors :

(b)	`flip_button`	...	Rel.	Push.	Push.	Rel.	Push.
(x)	`get_mode(`b`) every ...`	...	Right	Left	Left	Left	Right
(tl)	`Only when Left(`x`)`	...	*abs*	Only	Only	Only	*abs*
$\begin{pmatrix} ml \\ sl \end{pmatrix}$	`ms(`tl`)`	... ...	*abs*	59 58	59 59	0 0	*abs*
(tr)	`Only when Right(`x`)`	...	Only	*abs*	*abs*	*abs*	Only
$\begin{pmatrix} mr \\ sr \end{pmatrix}$	`ms(`tr`)`	...	17 18	*abs*	*abs*	*abs*	17 19
$\begin{pmatrix} m \\ s \end{pmatrix}$	`merge (Left->`ml, sl`)` `(Right->`mr, sr`)`	... ...	17 18	59 58	59 59	0 0	17 19

et vérifie bien le système.

Chapitre 6

Lsn : syntaxe et sémantique

6.1 Syntaxe de Lsn

Lsn (pour Ls normalisé) est un sous langage de Ls. Cependant, pour simplifier les algorithmes décrits par la suite, ainsi que les preuves, il est préférable de lui donner une syntaxe propre.

Comme le montre la figure 6.1, les expressions sont plus structurées que dans Ls. En particulier, toutes les structures censées contenir un état (décalage de flots et appel de nœud) sont en tête des équations.

Différences	Motivations
`fby` et `every` au niveau des équations	isoler les mémoires pour le passage vers une sémantique instantanée
disparition des tuples	élimination d'objets de seconde classe
`merge` en tête d'expressions	refléter la structure du code cible ; les `merge` seront traduits vers des blocs de contrôle imbriqués
`when` enrobe uniquement une constante ou un nom de flot	produire du code dans lequel on n'échantillonne pas d'expression calculatoire
`fby` et `every` ne travaillent que sur des noms de flots	des raisons techniques liées à la simplification de preuves dans Lsni

n	$::=$	`node` f `(` x `:` τ `) returns (` x `:` τ `);`
		`var` x `:` τ `on` k `(` x `);` ...`;` x `:` τ `on` k `(` x `)`
		`let` sys `tel;`
sys	$::=$	eq `;` ...`;` eq
eq	$::=$	x `=` ce | x `=` k `fby` x | x `=` f `(` x `) every` $k(x)$
ce	$::=$	e | `merge` x `(`k`->` ce `)` ...`(`k`->`ce`)`
e	$::=$	sa | o `(` e `,` ...`,` e `)`
sa	$::=$	a | sa `when` k `(` x `)`
a	$::=$	x | k

Figure 6.1: Syntaxe des nœuds et expressions dans Lsn

```
node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when not real_flip;
    switch_ : mode on real_flip;
    -- begin of new variables
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
    -- end of new variables
let
    -- begin of new equations
    mode_4_1 = Left fby mode_4_0;
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_);
    button_1_0 = Pushed fby flip_button;
    -- end of new equations
    real_flip = trigger(button_1_0, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
    current_mode = mode_4_1;
tel;
```

Figure 6.2: `get_mode` en LSN

Cette syntaxe est proche des « *A*-normal forms » [20] utilisées en compilation de langages fonctionnels où chaque expression atomique est liée à une variable.

Exemple 6.1.1 (`get_mode` et `ms` normalisés)**.** *Reprenons, en figure 6.2, l'exemple qui sert de fil rouge à cette thèse, et plus particulièrement les nœuds* `get_mode` *et* `ms`*. Sa compilation, c'est-à-dire la méthode pour obtenir ce résultat sera décrite dans le prochain chapitre.*

6.2 Typage de Lsn

Le typage dans LSN se définit dans le même cadre que celui de LS. Cependant il est plus simple dans le cas des expressions sans mémoire, puisqu'un unique type est retourné (et non un tuple).

En vérité, et cela sera encore plus flagrant d'un point de vue sémantique, il vaut mieux voir LS comme une extension de LSN que LSN comme une restriction de LS. Les règles d'inférence sont en fait plus simples dans LSN que dans LS.

Les règles de typage sont données en figure 6.3 et figure 6.4. Elles sont induites par celles de LS. On trouve donc un prédicat $\mathcal{E} \vdash_{\text{LSN}} e : \tau, \text{ck}$ qui exprime le fait que l'expression e a pour type τ[1] et horloge ck sous l'environnement $\mathcal{E}$. On trouve aussi un prédicat $\mathcal{E} \vdash_{\text{LSN}} eq/sys : E$ qui exprime le fait que l'équation eq ou le système sys soit bien typée sous l'environnement de typage $\mathcal{E}$ et définit les noms de flots de E. Enfin, $\vdash_{\text{LSN}} n : (\tau_i, \tau_o)$

1. Contrairement à ce qui se passe dans LS, une expression a un type simple et non un produit de types.

$$\textsc{Const}\ \frac{}{\mathcal{E} \vdash_{\text{Lsn}} k : \text{Ktype}\,(k)\,,\texttt{when Only}\,(\texttt{base})} \qquad \textsc{Var}\ \frac{x \in dom_{\mathcal{E}}}{\mathcal{E} \vdash_{\text{Lsn}} x : \text{type}_{\mathcal{E}}\,(x), \text{ck}\,(x)}$$

$$\textsc{Op}\ \frac{O_{sig}\,(o) = (\tau_1 \times ... \times \tau_n) \to \tau_o \qquad \forall 1 \leq i \leq n. \quad \mathcal{E} \vdash_{\text{Lsn}} e_i : \tau_i, \text{ck}}{\mathcal{E} \vdash_{\text{Lsn}} o\texttt{(}e_1, ..., e_n\texttt{)} : \tau_o, \text{ck}}$$

$$\textsc{When}\ \frac{x \in dom_{\mathcal{E}} \qquad \text{type}_{\mathcal{E}}\,(x) = \text{Ktype}\,(k) \qquad \mathcal{E} \vdash_{\text{Lsn}} e : \tau, \text{ck}\,(x)}{\mathcal{E} \vdash_{\text{Lsn}} e\ \texttt{when}\ k\texttt{(}x\texttt{)} : \tau, k\texttt{(}x\texttt{)}}$$

$$\textsc{Merge}\ \frac{\begin{array}{c} x \in dom_{\mathcal{E}} \\ \forall \texttt{(}k_i\texttt{->}e_i\texttt{)} \in cases.\ \mathcal{E} \vdash_{\text{Lsn}} e_i : \tau, k_i\texttt{(}x\texttt{)} \qquad \forall k \in \text{type}_{\mathcal{E}}\,(x)\,.\ \exists! e_k.\ \texttt{(}k\texttt{->}e_k\texttt{)} \in cases \end{array}}{\mathcal{E} \vdash_{\text{Lsn}} \texttt{merge}\ x\ cases : \tau, \text{ck}\,(x)}$$

Figure 6.3: Règles de typage des expressions de Lsn

$$\textsc{EqExp}\ \frac{x \in dom_{\mathcal{E}} \qquad \mathcal{E} \vdash_{\text{Lsn}} e : \text{type}_{\mathcal{E}}\,(x)\,, \text{ck}\,(x)}{\mathcal{E} \vdash_{\text{Lsn}} x = e : \{x\}}$$

$$\textsc{EqFby}\ \frac{x \in dom_{\mathcal{E}} \qquad x' \in dom_{\mathcal{E}} \qquad \text{ck}\,(x') = \text{ck}\,(x) \qquad \text{Ktype}\,(k) = \text{type}_{\mathcal{E}}\,(x')}{\mathcal{E} \vdash_{\text{Lsn}} x = k\ \texttt{fby}\ x' : \{x\}}$$

$$\textsc{EqApp}\ \frac{\begin{array}{ccccc} x_r \in dom_{\mathcal{E}} & \text{type}_{\mathcal{E}}\,(x_r) = \text{Ktype}\,(k) & N_{sig}\,(x) = \tau_i \to \tau_o & x_i \in dom_{\mathcal{E}} \\ \text{ck}\,(x_i) = \text{ck}\,(x_r) = \text{ck}\,(x_o) & \text{type}_{\mathcal{E}}\,(x_i) = \tau_i & x_o \in dom_{\mathcal{E}} & \text{type}_{\mathcal{E}}\,(x_o) = \tau_o \end{array}}{\mathcal{E} \vdash_{\text{Lsn}} x_o = f\texttt{(}x_i\texttt{)}\ \texttt{every}\ k\texttt{(}x_r\texttt{)} : \{x_o\}}$$

$$\textsc{SysO}\ \frac{}{\mathcal{E} \vdash_{\text{Lsn}} \varepsilon_s : \emptyset} \qquad \textsc{SysS}\ \frac{\mathcal{E} \vdash_{\text{Lsn}} eq : E_{eq} \qquad \mathcal{E} \vdash_{\text{Lsn}} sys : E_{sys} \qquad \emptyset = E_{eq} \cap E_{sys}}{\mathcal{E} \vdash_{\text{Ls}} eq\texttt{;}\ \ sys : E_{eq} \cup E_{sys}}$$

$$\textsc{Node}\ \frac{TEnv\,(i, o, l) \vdash_{\text{Lsn}} sys : \{o, l_1, ..., l_r\}}{\vdash_{\text{Lsn}} \begin{array}{l} \texttt{node}\ f\ \texttt{(}i : \tau_i\texttt{)} \\ \texttt{returns (}o : \tau_o\texttt{);} \\ \texttt{var}\ l_1 : \tau_{l_1}\texttt{;} ...\texttt{;} l_r : \tau_{l_r}\texttt{;} \\ \texttt{let}\ sys\ \texttt{tel;} \end{array} : (\tau_i, \tau_o)}$$

Figure 6.4: Règles de typage des nœuds de Lsn

$$\textsc{Const}\ \frac{}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} k \Downarrow \mathbf{lift}^{\#}_{(\boldsymbol{\lambda x.k})}\left(\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(\texttt{base})\right)} \qquad \textsc{Var}\ \frac{}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} x \Downarrow \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)}$$

$$\textsc{Op}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_1 \Downarrow \psi_1 \qquad ... \qquad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_n \Downarrow \psi_n}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} o(e_1, ..., e_n) \Downarrow \mathbf{lift}^{\#}_{\boldsymbol{O_{sem}(op)}}(\psi_1, ..., \psi_n)}$$

$$\textsc{When}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e \Downarrow \psi' \qquad \psi = \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e\ \texttt{when}\ k(x) \Downarrow \mathbf{when}^{\#}_{\boldsymbol{k}}\ (\psi', \psi)}$$

$$\textsc{Merge}\ \frac{\forall 1 \leq i \leq n.\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_i \Downarrow \psi_{k_i} \qquad \mathbf{merge}^{\#}\left((\psi_{k_i})_{k_i}, \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)\right) = \psi'}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} \texttt{merge}\ x\ \ (k_1\texttt{->}e_1)...(k_n\texttt{->}e_n) \Downarrow \psi'}$$

Figure 6.5: Sémantique des expressions de Lsn

exprime le fait que le nœud n soit bien typé, de type d'entrées τ_i et de type de sortie τ_o.

Contrairement à Ls, on ne cherche plus à savoir si le typage est vérifiable de typage est décidable (il se trouve qu'il l'est, mais c'est anecdotique). En effet, on montre aisément que la compilation qui va être décrite par la suite préserve le typage. Si le code Ls est bien typé et compile, alors le code (Lsn) compilé est aussi bien typé.

6.3 Sémantique de Lsn

La sémantique de Lsn se fait dans le même cadre que celle de Ls. Les règles d'inférences sont présentées en figure 6.5 et figure 6.6 ; les mêmes remarques que celles du typage s'appliquent, cette sémantique est plus simple pour les expressions, et la plupart des expressions traduisent en fait directement la primitive associée (car il n'y a plus de tuples). On note $\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e \Downarrow \psi$ pour exprimer le fait que l'expression e produit le flot ψ sous l'environnement $\mathcal{S}^{\mathcal{F}}$. On note $\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} eq/sys$ pour exprimer le fait que sous l'environnement $\mathcal{S}^{\mathcal{F}}$, l'équation eq ou le système sys est bien vérifié(e). Enfin, $\vdash_{\text{Lsn}} n \Downarrow (\phi_i, \phi_o)$ exprime que le nœud n produit le flot ϕ_o sur l'entrée ϕ_i.

$$\textsc{EqExp}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e \Downarrow \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} \texttt{(}x\texttt{)} = e} \qquad \textsc{EqFby}\ \frac{\mathbf{fby}^{\#}_{k}\left(\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x')\right) = \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x)}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} x = k\ \texttt{fby}\ x'}$$

$$\textsc{EqApp}\ \frac{\begin{array}{c}\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x_r) = \psi_r \qquad \mathbf{seq}^{\#}_{k}\left(\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x_i), \psi_r\right) = \psi_{i,1}, ..., \psi_{i,m} \\ \mathbf{seq}^{\#}_{k}\left(\text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(x_o), \psi_r\right) = \psi_{o,1}, ..., \psi_{o,m} \qquad \forall 1 \leq k \leq m.\quad (\psi_{i,k}, \psi_{o,k}) \in N^{\mathcal{F}}_{sem}(f)\end{array}}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} x_o = f\texttt{(}x_i\texttt{)}\ \texttt{every}\ k\texttt{(}x_r\texttt{)}}$$

$$\textsc{SysO}\ \frac{}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} \varepsilon_s} \qquad \textsc{SysS}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} eq \qquad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} sys}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} eq\texttt{;}\ sys}$$

$$\textsc{Node}\ \frac{\mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} sys \qquad \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(i) = {\phi_i}^{abs} \qquad \text{sem}^{\mathcal{F}}_{\mathcal{S}^{\mathcal{F}}}(o) = {\phi_o}^{abs}}{\vdash_{\text{Lsn}} \begin{array}{l}\texttt{node}\ f\ \texttt{(}i : \tau_i\texttt{)} \\ \texttt{returns (}o : \tau_o\texttt{);} \\ \texttt{var}\ l_1 : \tau_{l_1}\texttt{;} ...\texttt{;} l_r : \tau_{l_r}\texttt{;} \\ \texttt{let}\ sys\ \texttt{tel;}\end{array} \Downarrow (\phi_i, \phi_o)}$$

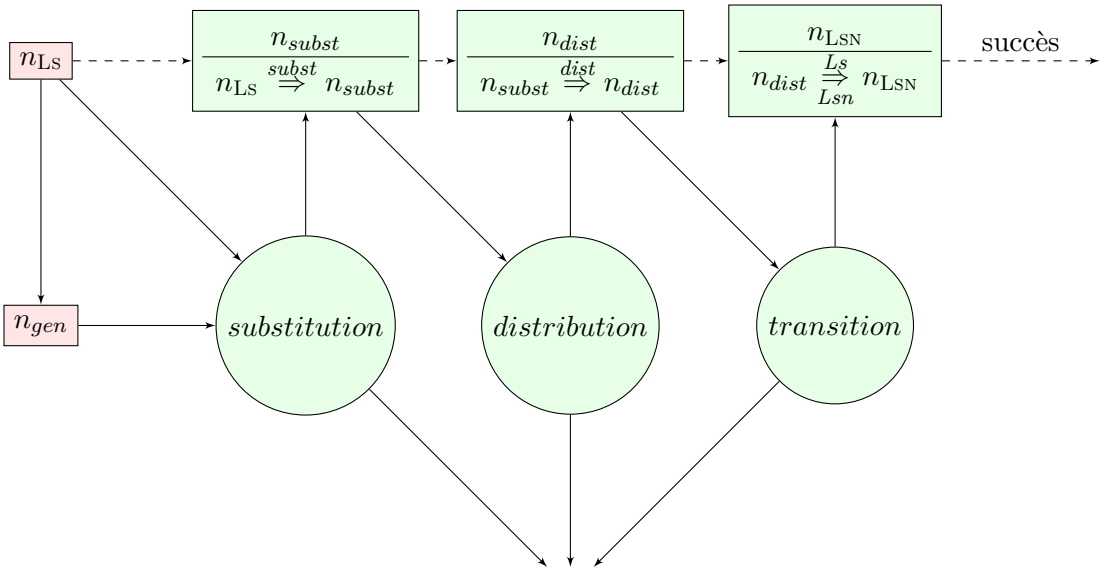
Figure 6.6: Sémantique des nœuds de Lsn

Chapitre 7

Normalisation : de Ls à Lsn

7.1 Schéma de l'algorithme de normalisation

L'algorithme de normalisation se découpe en deux phases ; la première phase introduit de nouvelles équations ; la seconde élimine les tuples. Enfin une passe supplémentaire projette le code normalisé vers la syntaxe de Lsn.



En teinte rose apparaissent les parties du monde extérieur à l'assistant de preuve, elles ne sont pas certifiées et n'ont pas besoin de l'être. Les flèches pointillées représentent la suite des nœuds produits. Les rectangles sont des types sommes dans l'assistant de preuve ; l'extraction ne garde que la partie supérieure, mais on sait qu'on a été capable de produire la partie inférieure (la spécification du code extrait). Ici, cette spécification correspond à trois relations $\overset{subst}{\Rightarrow}$, $\overset{dist}{\Rightarrow}$ et $\underset{Lsn}{\overset{Ls}{\Rightarrow}}$ qui seront définies dans ce chapitre et dont on a prouvé qu'elles préservaient la sémantique. n_{gen} est une indication donnée par un outil externe (par exemple programmé en OCaml) à partir de n_{Ls}, qui donne une indication de comment

compiler le code. Comme indiqué quand la traduction par validation a été présentée, cet outil n'a pas besoin d'être sûr.

7.2 Création de noms de flots intermédiaires

7.2.1 Présentation de la validation de cette passe

Comme annoncé précédemment, cette phase se fait par validation. On utilise un programme extérieur qui va suggérer de nouvelles équations à introduire (V_{1a}) ; une fois toutes ces équations introduites on valide le système par rapport à celui d'origine (V_{1b}) (voir la section 3.5.2 pour les notations V_{1a} et V_{1b}).

Informellement, la relation attendue est que le système produit soit de la forme p=e ; sys_2 où :

- p est le nom de flot introduit [1] ;
- e apparaît dans le système d'origine ;
- e a son type et son horloge qui concordent avec p ;
- en remplaçant toutes les occurrences de p dans sys_2 ; par e, on obtient le système d'origine.

Cette relation s'avère être décidable et peut donc servir à la constitution d'une compilation par validation.

La suite de cette section s'attache à définir formellement les notions d'occurrences et de substitutions, ainsi que de prouver que la validation de cette relation permet de préserver la sémantique.

Sous environnement

Définition 7.2.1 (sous environnement)**.** *Étant donnés deux environnements de typage $\mathcal{E}_1$ et $\mathcal{E}_2$ et deux environnements associés $\mathcal{S}_1^{\mathcal{F}}$ et $\mathcal{S}^{\mathcal{F}}$, on dit que $\mathcal{S}_1^{\mathcal{F}}$ est un sous environnement de $\mathcal{S}_2^{\mathcal{F}}$ s'il vérifie les conditions suivantes :*

- $dom_{\mathcal{E}_1} \subseteq dom_{\mathcal{E}_2}$
- $\forall x \in dom_{\mathcal{E}_1}.\ \mathrm{type}_{\mathcal{E}_1}(x) = \mathrm{type}_{\mathcal{E}_2}(x)$
- $\forall x \in dom_{\mathcal{E}_1}.\ \mathrm{sem}^{\mathcal{F}}_{\mathcal{S}_1^{\mathcal{F}}}(x) = \mathrm{sem}^{\mathcal{F}}_{\mathcal{S}_2^{\mathcal{F}}}(x)$

Cette relation définit une relation d'ordre partiel ; on notera donc par la suite $\mathcal{S}^{\mathcal{F}} \leq \mathcal{S}^{\mathcal{F}}$.

Occurrences et transparence référentielle

Lustre est un langage qui vérifie la propriété de transparence référentielle. En d'autres termes remplacer une ou plusieurs occurrences d'un nom de flot par sa définition ne change pas la sémantique du programme. Cette propriété est souvent utilisée dans les langages qui la vérifient à des fins d'optimisation ; ici, même si on peut l'utiliser pour optimiser le code, sa première fonction va être de déplacer des bouts de code dans une structure plus hiérarchisée.

1. dans le développement effectué, il s'agit d'une liste de noms de flots, ce qui permet de gérer des nœuds qui retournent plusieurs flots.

$$\textsc{Refl}\ \frac{}{e \lhd e} \qquad \textsc{Fby}\ \frac{e \lhd e'}{e \lhd k\ \texttt{fby}\ e'} \qquad \textsc{Op}\ \frac{e \lhd e'}{e \lhd o(e')} \qquad \textsc{Every}\ \frac{e \lhd arg}{e \lhd f(arg)\ \texttt{every}\ k(r)}$$

$$\textsc{When}\ \frac{e \lhd e'}{e \lhd e'\ \texttt{when}\ k(x)} \qquad \textsc{Tuple1}\ \frac{e \lhd e_1}{e \lhd e_1, e_2} \qquad \textsc{Tuple2}\ \frac{e \lhd e_2}{e \lhd e_1, e_2}$$

$$\textsc{Merge}\ \frac{\exists 1 \leq i \leq n.\ e \lhd e_{k_i}}{e \lhd \texttt{merge}\ x\ \ (k_1\texttt{->}e_{k_1})\ \ \ldots\ \ (k_n\texttt{->}e_{k_n})} \qquad \textsc{Eq}\ \frac{e \lhd exp}{e \lhd (x_1, ..., x_n)\texttt{=}exp}$$

$$\textsc{SysS1}\ \frac{e \lhd eq}{e \lhd eq\,;sys} \qquad \textsc{SysS2}\ \frac{e \lhd sys}{e \lhd eq\,;sys}$$

Figure 7.1: Occurrence d'une expression

Définition 7.2.2 (occurrence d'une expression)**.** *On dit qu'une expression exp (ou un système sys) a une occurrence d'une expression e si e apparaît dans exp (ou sys). Formellement, cela se traduit par les règles d'inférences en figure 7.1. On note $e \lhd exp$ (ou $e \lhd sys$) cette relation.*

Définition 7.2.3 (équivalence modulo égalité de deux expressions)**.** *Deux expressions e_1 et e_2 sont équivalentes modulo égalité de deux expressions a et b si en remplaçant certaines occurrences de a dans e_1 par b, on obtient e_2. Cette relation d'équivalence est notée $e_1 \overset{a \to b}{\sim} e_2$. Cette définition s'étend aux équations et systèmes d'équations. Les règles d'inférence sont données en figure 7.2.*

Substitutions

On peut maintenant définir la relation que l'on souhaite voir validée entre deux nœuds pour assurer la préservation de la sémantique.

Définition 7.2.4 (Relation de substitution entre deux nœuds)**.** *On définit la relation*

$$\textsc{Refl1}\ \frac{}{a \overset{a\to b}{\sim} b} \qquad \textsc{Refl2}\ \frac{}{e \overset{a\to b}{\sim} e} \qquad \textsc{Fby}\ \frac{e_1 \overset{a\to b}{\sim} e_2}{k\ \texttt{fby}\ e_1 \overset{a\to b}{\sim} k\ \texttt{fby}\ e_2}$$

$$\textsc{Op}\ \frac{e_1 \overset{a\to b}{\sim} e_2}{o(e_1) \overset{a\to b}{\sim} o(e_2)} \qquad \textsc{Every}\ \frac{e_1 \overset{a\to b}{\sim} e_2 \qquad r_1 \overset{a\to b}{\sim} r_2}{f(e_1)\ \texttt{every}\ r_1 \overset{a\to b}{\sim} f(e_2)\ \texttt{every}\ r_2}$$

$$\textsc{When}\ \frac{e_1 \overset{a\to b}{\sim} e_2}{e_1\ \texttt{when}\ k(x) \overset{a\to b}{\sim} e_2\ \texttt{when}\ k(x)} \qquad \textsc{Tuple}\ \frac{e_1 \overset{a\to b}{\sim} e_2 \qquad f_1 \overset{a\to b}{\sim} f_2}{e_1, e_2 \overset{a\to b}{\sim} f_1, f_2}$$

$$\textsc{Merge}\ \frac{\forall 1 \leq i \leq n.\ e_{k_i 1} \overset{a\to b}{\sim} e_{k_i 2}}{\texttt{merge}\ x\ (k_1\texttt{->}e_{k_1 1})\ \ldots\ (k_n\texttt{->}e_{k_n 1}) \overset{a\to b}{\sim} \texttt{merge}\ x\ (k_1\texttt{->}e_{k_1 2})\ \ldots\ (k_n\texttt{->}e_{k_n 2})}$$

$$\textsc{Eq}\ \frac{e_1 \overset{a\to b}{\sim} e_2}{(x_1, ..., x_n)\texttt{=}e_1 \overset{a\to b}{\sim} (x_1, ..., x_n)\texttt{=}e_2} \qquad \textsc{SysO}\ \frac{}{\varepsilon_s \overset{a\to b}{\sim} \varepsilon_s}$$

$$\textsc{SysS}\ \frac{eq_1 \overset{a\to b}{\sim} eq_2 \qquad sys_1 \overset{a\to b}{\sim} sys_2}{eq_1\,\texttt{;}\,sys_1 \overset{a\to b}{\sim} eq_2\,\texttt{;}\,sys_2}$$

Figure 7.2: Équivalence modulo égalité de deux expressions

$n_s \overset{subst}{\Rightarrow} n_t$ *entre un code d'origine* n_s *et un code produit* n_t *par :*

$$n_s \overset{subst}{\Rightarrow} n_t = \begin{array}{l} \exists b, x_1, ..., x_n, \tau_1, ..., \tau_n, \mathrm{ck}, sys_1, sys_2, locs, i, o, f. \\ \bigwedge \begin{cases} TEnv\,(i, o, locs) \vdash_{\mathrm{Ls}} b : \tau_1, ..., \tau_n, \mathrm{ck} \\ \forall 1 \leq i \leq n.\ x_i \notin dom_{TEnv(i,o,locs)} \\ b \lhd sys_1 \\ sys_2 \overset{(x_1,...,x_n)\to b}{\sim} sys_1 \\ \forall 1 \leq i < j \leq n.\ x_i \neq x_j \\ n_s = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs \\ \texttt{let } sys_1 \texttt{ tel;} \end{array}\right. \\ n_t = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs; x_1 : \tau_1 \texttt{ on } \mathrm{ck}; ...; x_n : \tau_n \texttt{ on } \mathrm{ck}; \\ \texttt{let } (x_1, ..., x_n) = b\texttt{;}sys_2 \texttt{ tel;} \end{array}\right. \end{cases} \end{array}$$

Théorème 7.2.1 (Préservation de la sémantique par ajout de noms de flots)**.** *La validation de la relation* $n_s \overset{subst}{\Rightarrow} n_t$ *implique la préservation de la sémantique de* n_s *après transformation en* n_t*. Plus formellement, en utilisant les notations de la partie précédente :*

$$\forall n_s, n_t.\ \left(n_s \overset{subst}{\Rightarrow} n_t\right) \Rightarrow (n_s \lesssim n_t)$$

Démonstration. Ce théorème est une application directe du corollaire 7.2.2.

Lemme 7.2.1 (préservation de la sémantique par équivalence modulo égalité)**.**

$$\begin{array}{l} \forall \mathcal{E}_1, \mathcal{E}_2, \mathcal{S}^{\mathcal{F}}, \mathcal{S}^{\mathcal{F}}.\ \mathcal{S}^{\mathcal{F}} \leq \mathcal{S}^{\mathcal{F}} \Rightarrow \\ \quad \forall a, b, \vec{\psi}.\ \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} a \Downarrow \vec{\psi} \wedge \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} b \Downarrow \vec{\psi} \Rightarrow \\ \quad\quad \forall e_1, e_2, \vec{\psi'}.\ e_2 \overset{a\to b}{\sim} e_1 \wedge \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} e_1 \Downarrow \vec{\psi'} \Rightarrow \\ \quad\quad\quad \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} e_2 \Downarrow \vec{\psi'} \end{array}$$

De manière plus informelle, si deux expressions e_1 *et* e_2 *sont équivalentes modulo égalité de deux expressions* a *et* b *ayant même sémantique (à extension de l'environnement local près), alors la sémantique de* e_1 *est incluse dans celle de* e_2*.*

L'extension de ce lemme aux systèmes d'équations donne :

$$\begin{array}{l} \forall \mathcal{E}_1, \mathcal{E}_2, \mathcal{S}^{\mathcal{F}}, \mathcal{S}^{\mathcal{F}}.\ \mathcal{S}^{\mathcal{F}} \leq \mathcal{S}^{\mathcal{F}} \Rightarrow \\ \quad \forall a, b, \vec{\psi}.\ \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} a \Downarrow \vec{\psi} \wedge \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} b \Downarrow \vec{\psi} \Rightarrow \\ \quad\quad \forall sys_1, sys_2.\ sys_2 \overset{a\to b}{\sim} sys_1 \wedge \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} sys_1 \Rightarrow \\ \quad\quad\quad \mathcal{S}^{\mathcal{F}} \vdash_{\mathrm{Ls}} sys_2 \end{array}$$

Preuve du lemme 7.2.1. La preuve se fait par induction sur la relation d'équivalence modulo égalité. Tous les cas sont traités trivialement ; sauf Refl2 où on a besoin de faire une induction (triviale) sur $e_1 (= e_2)$ pour démontrer que la sémantique d'une expression est préservée par extension de l'environnement local. (On peut cependant simplifier cette preuve en remplaçant Refl2 par deux règles d'inférence sur les variables et les constantes.) □

En appliquant le lemme précédent dans le cas où a est une variable ou un tuple de variables, on obtient le corollaire suivant :

Corollaire 7.2.1 (Corollaire de substitutions)**.**

$$\begin{array}{l}\forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, b, \tau_1, ..., \tau_n, \text{ck} \,.\, \mathcal{E} \vdash_{\text{Ls}} b : \tau_1, ..., \tau_n, \text{ck} \Rightarrow \\ \quad \forall sys_1.\ b \lhd sys_1 \wedge \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys_1 \Rightarrow \\ \quad\quad \forall sys_2, x_1, ..., x_n.\ (\forall 1 \leq i < j \leq n.\ x_i \neq x_j \wedge x_i \notin dom_{\mathcal{E}}) \wedge sys_2 \overset{(x_1,...,x_n)\to b}{\sim} sys_1 \Rightarrow \\ \quad\quad\quad \exists \psi_1, ..., \psi_n.\ \mathcal{S}^{\mathcal{F}} + [x_i \mapsto \psi_i] \vdash_{\text{Ls}} (x_1, ..., x_n) = b; sys_2\end{array}$$

Preuve du corollaire 7.2.1. Soient $\mathcal{E}$ un environnement de typage local, $\mathcal{S}^{\mathcal{F}}$ un environnement sémantique local, τ_1, ..., τ_n des types, ck une horloge et b une expression de type dont on suppose (H_b) qu'elle est de type $\tau_1, ..., \tau_n$ sur l'horloge ck.

Si maintenant on considère un système sys_1 dans lequel apparaît b $(H_{b \lhd sys_1})$ et qui ait du sens vis-à-vis de l'environnement sémantique local (H_{sys_1}), alors par une induction sur H_{sys_1} qui exploite $H_{b \lhd sys_1}$, on peut en déduire l'existence de $\psi_1, ..., \psi_n$, tels que $\mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} b \Downarrow \psi_1, ..., \psi_n$ (H_ψ).

En se donnant alors x_1, ..., x_n tous distincts et qui n'apparaissent pas dans le domaine de $\mathcal{E}$, on peut étendre $\mathcal{E}$ en $\mathcal{E}' = \mathcal{E} + [x_i \mapsto \tau_i \ \texttt{on}\ \text{ck}]$ et $\mathcal{S}^{\mathcal{F}}$ en $\mathcal{S}^{\mathcal{F}} = \mathcal{S}^{\mathcal{F}} + [x_i \mapsto \psi_i]$; on a alors $\mathcal{S}^{\mathcal{F}} \leq \mathcal{S}^{\mathcal{F}}$ $(H_\leq)$.

En utilisant alors le lemme 7.2.1, et les hypothèses $H_\leq$, H_ψ et H_{sys_1} on en déduit : $\forall sys_2.\ sys_2 \overset{(x_1,...,x_n)\to b}{\sim} sys_1 \Rightarrow \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys_2$.

Finalement, on peut conclure en utilisant les règles SysS de typage et de sémantique pour Ls. □

Corollaire 7.2.2 (Préservation de la sémantique par introduction de nouveaux noms de flots)**.**

$$\begin{array}{l}\forall b, i, o, locs, f, x_1, ..., x_n, \tau_1, ..., \tau_n, \text{ck}, sys_1, sys_2. \\ \quad TEnv\,(i, o, locs) \vdash_{\text{Ls}} b : \tau_1, ..., \tau_n, \text{ck} \wedge \forall 1 \leq i \leq n.\ x_i \notin dom_{TEnv(i,o,locs)} \wedge \\ \quad b \lhd sys_1 \wedge sys_2 \overset{(x_1,...,x_n)\to b}{\sim} sys_1 \wedge (\forall 1 \leq i < j \leq n.\ x_i \neq x_j) \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{node } f(i) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{returns } (o) \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{node } f(i) \qquad\qquad\qquad\qquad\qquad\qquad \texttt{var } locs; \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{returns } (o) \qquad\qquad\qquad\qquad\qquad\qquad\quad x_1 : \tau_1 \ \texttt{on}\ \text{ck}; \\ \qquad\qquad \forall \phi_i, \phi_o.\ \vdash_{\text{Ls}} \ \texttt{var } locs \qquad \Downarrow (\phi_i, \phi_o) \Rightarrow \vdash_{\text{Ls}} \qquad\quad ...; \qquad\qquad\qquad \Downarrow (\phi_i, \phi_o) \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{let } sys_1 \qquad\qquad\qquad\qquad\qquad\qquad\quad x_n : \tau_n \ \texttt{on}\ \text{ck}; \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{tel;} \qquad\qquad\qquad\qquad\qquad\qquad \texttt{let } (x_1, ..., x_n) = b; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad sys_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{tel;}\end{array}$$

Preuve du corollaire 7.2.2. La preuve est une application directe du corollaire 7.2.1 et des règles d'inférences du typage et de sémantique. □

□

Un algorithme vérifiant la relation de substitution entre deux nœuds est facile à implémenter dans l'assistant de preuve Coq.

7.2.2 Algorithme d'introduction de nouvelles équations

Il manque un dernier élément à la première passe de normalisation : l'outil externe (V_{1a}) qui va proposer un candidat à la validation, ou annoncer que l'on peut passer à la passe de compilation suivante.

Grâce au système de compilation par validation, cet outil n'a pas besoin d'être certifié correct (c'est-à-dire qu'on peut l'autoriser à retourner un nœud qui ne vérifie pas la relation souhaitée).

En conséquence, cette section ne va pas proposer de preuve, mais simplement discuter des différentes possibilités de génération de code.

Anticipation des passes suivantes

Il existe des codes de nœuds qui peuvent être validés, mais qui ne sont pas intéressants pour autant.

En effet, la validation garantit la préservation de la sémantique en cas de succès, mais elle ne garantit pas que le code produit soit accepté par les étapes suivantes de la compilation.

Voici une liste des propriétés minimales à exiger pour le bon déroulement de la suite de la compilation ; ces différents points seront discutés dans les passes de compilation correspondantes :

P_1 tous les arguments d'une application de nœud ou d'un décalage de flot doivent être des variables (pour des raisons de simplifications de preuve)

P_2 toutes les applications de nœuds et tous les décalages doivent se faire au niveau des équations (car leurs versions impératives correspondent à des modifications de l'environnement qui dépendent d'un ordre d'exécution)

P_3 un opérateur ne peut pas prendre en argument une fusion de flots (par simplification, et pour éviter une explosion de la taille du code)

P_4 seuls les noms de flots et les constantes peuvent être échantillonnés (pour avoir une représentation interne simplifiée dans Lsn)

P_5 il ne doit pas y avoir de cycle de décalages directs de flots car l'ordonnanceur ne peut pas les gérer sans nom de flot intermédiaire.
Par exemple, `x=0 fby y; y=1 fby x;` n'est pas acceptable pour l'ordonnanceur ; il faut introduire `z=x`, pour obtenir le système `z=x; x=0 fby y; y=1 fby z;` qui peut être ordonnancé. Pour plus de détails, se référer aux sections 9.2.2 et 11.1.1.

P_6 aucune sortie de nœud ne peut être définie directement par un décalage car les décalages correspondront à des flots pré-calculés. Dans la compilation actuelle, retourner un tel flot reviendrait à retourner sa version non décalée. Nous reviendrons sur ce point aux sections 9.2.2 et 11.1.1.

Les quatre premières propriétés sont reflétées par la syntaxe de Lsn.

Deux exemples d'implémentation

Un premier exemple d'implémentation d'un algorithme éligible pour cette première passe de simplification de code, que nous appellerons « implémentation minimale » consiste à repérer la première occurrence d'expression e qui ne vérifie pas une des propriétés demandées, remplacer cette occurrence par un nom de flot x et d'ajouter l'équation $x = e$ au système. Si toute occurrence d'expression vérifie la propriété demandée, on a terminé la première passe.

Un second exemple d'implémentation, que nous appellerons « implémentation maximale » consiste à nommer toute expression du système d'origine qui ne soit pas déjà une variable, et à utiliser ce nom dans toutes les occurrences de cette expression.

Exemple 7.2.1 (`ms` traduit avec l'implémentation minimale)**.**

```
-- node to count time in minutes and seconds
node ms (tic:unit) returns (mins, secs: int32);
var reset:bool;
    -- begin of new variables
    chrono1 : int32;
    chrono2 : int32;
    reset1 : bool;
    -- end of new variables
let reset1 = (mins==59)&&(secs==59);
    chrono2 = chrono_base(tic) every reset;
    chrono1 = chrono_base(tic) every reset;
    mins = chrono1/60;
    secs = chrono2%60;
    reset = False fby reset1;
tel;
```

Exemple 7.2.2 (`ms` traduit avec l'implémentation maximale)**.**

```
-- node to count time in minutes and seconds
node ms (tic:unit) returns (mins, secs: int32);
var reset:bool;
    -- begin of new variables
    mins_aux : int32;
    chrono : int32;
    cst60 : int32;
    secs_aux : int32;
    reset_aux : bool;
    bmins : bool;
    cst59 : int32;
    bsecs : bool;
    -- end of new variables
let
    -- begin of new equations
    bsecs = secs == cst59;
    cst59 = 59;
    bmins = mins == cst59;
    reset_aux = bmins && bsecs;
    secs_aux = chrono % cst60;
```

```
    cst60 = 60;
    chrono = chrono_base(tic) every reset;
    mins_aux = chrono / cst60;
    -- end of new equations
    mins = mins_aux;
    secs = secs_aux;
    reset = reset_aux;
tel;
```

Sur ces deux exemples, on voit qu'en substituant successivement chacune des équations introduites, on retombe sur le nœud original (c'est ce que fait la fonction de validation).

Ces deux implémentations sont très simples. L'implémentation minimale permet d'avoir du code plus facile à reconnaître vis-à-vis du code initial ; alors que l'implémentation maximale tend à produire du code difficilement reconnaissable et très proche du code trois adresses que l'on trouve dans de nombreux compilateurs.

Un des aspects intéressants de la traduction par validation est qu'aucune des deux implémentations n'est plus difficile à vérifier que l'autre, ni même que pour toute autre. On peut donc partir d'une implémentation naïve puis l'améliorer sans avoir peur de casser les preuves, ou de devoir remettre en cause la correction du compilateur.

Optimisations

À ce stade, diverses optimisations sont possibles. En voici trois :
- factorisation de sous expressions communes
- minimisation d'automates
- élimination de code mort

Cette thèse n'a pas pour but de traiter la préservation de la sémantique au travers d'optimisations. Cependant, sur ces trois optimisations, la première peut être obtenue gratuitement (l'implémentation maximale fait une telle factorisation) ; les deux autres nécessiteraient des preuves supplémentaires, et sont commentées en annexe.

Exemple 7.2.3 (`chrono_base`, `get_mode` et `ms` après introduction de nouveaux flots)**.** *On reprend notre exemple « pendule des échecs ». L'implémentation choisie est différente des deux implémentations précédentes ; elle se comporte comme l'implémentation minimale, mais factorise aussi son code.*

Les nouveaux noms de flots sont formés du type du nom de flot introduit, suivi du numéro de l'équation dans lequel figurait l'expression, suivi d'un autre numéro pour différencier deux flots de même type provenant d'une même équation. Cette convention de nommage est arbitraire.

Le nœud `chrono_base` *introduit le nom de flot* `int32_1_0` *pour vérifier la propriété* P_1*, et le nom de flot* `int32_1_1` *pour vérifier la propriété* P_6*.*

Le nœud `get_mode` *introduit* `mode_4_0` *(resp.* `mode_4_1`*,* `button_1_0`*) pour vérifier la propriété* P_1 *(resp.* P_6*,* P_2*).*

Le nœud `chess_clock` *présente un exemple où après introduction d'équations, le nœud n'est pas encore en forme normale ; en effet, il reste des tuples qui sont interdits dans* Lsn *et seront éliminés dans la passe suivante.*

```
node chrono_base (tic:unit) returns (time:int32);
var
    -- begin of new variables
    int32_1_0 : int32;
    int32_1_1 : int32;
    -- end of new variables
let
    -- begin of new equations
    int32_1_1 = 0 fby int32_1_0;
    int32_1_0 = 1+time;
    -- end of new equations
    time = int32_1_1;
tel;
```

Figure 7.3: `chrono_base` après introduction de nouveaux noms de flots

```
node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when not real_flip;
    switch_ : mode on real_flip;
    -- begin of new variables
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
    -- end of new variables
let
    -- begin of new equations
    mode_4_1 = Left fby mode_4_0;
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_);
    button_1_0 = Pushed fby flip_button;
    -- end of new equations
    real_flip = trigger(button_1_0, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
    current_mode = mode_4_1;
tel;
```

Figure 7.4: `get_mode` après introduction de nouveaux noms de flots

```
node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    -- begin of new variables
    int32_1_0 : int32;
    bool_3_0 : bool;
    -- end of new variables
let
    -- begin of new equations
    bool_3_0 = (mins==59) && (secs==59);
    int32_1_0 = chrono_base(tic) every reset;
    -- end of new equations
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    reset = False fby bool_3_0;
tel;
```

Figure 7.5: `ms` après introduction de nouveaux noms de flots

7.3 Distribution des tuples

7.3.1 Présentation de l'algorithme

L'algorithme de distribution ne dépend d'aucun choix, et n'introduit aucun élément nouveau dans la structure du nœud, il n'y a donc aucun intérêt à valider cette passe, et on peut l'écrire directement en Coq.

Comme présenté en section 3.5.1, on définir une relation de distributivité, démontrer que l'on sait fabriquer la version distribuée d'un système (V_1), et démontrer que cette relation préserve la sémantique (V_2).

Distribution d'un système d'équations

Étant donnée une expression e qui dans son contexte produit les flots $\psi_1, ..., \psi_n$, on voudrait la simplifier en une liste d'expressions $l = e_1, ..., e_n$ telle que l'évaluation de e_i donne ψ_i pour tout i.

Après l'introduction de nouveaux noms (c'est-à-dire la passe précédente), on peut toujours le faire sur les expressions obtenues (hormis l'application de nœud qui produit des flots indivisibles).

Cette relation de distribution est notée split $(e) = (l)$. Elle s'étend à une relation entre une équation et un système et à une relation entre deux système ; et est définie en figure 7.7.

La règle de distribution sur le décalage (Fby) aurait pu exploiter le fait qu'à ce stade la seule expression acceptée soit un nom de flot, et aurait pu aussi être placée comme règle sur les équations, comme (Every) ; les preuves n'en auraient été ni plus simples ni plus compliquées *a priori*.

Cette relation est essentiellement fonctionnelle, et il n'est pas difficile d'écrire un algorithme qui construit la liste d'expressions distribuées et avec une preuve que cette relation

```
node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    -- begin of new variables
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
    -- end of new variables
let
    -- begin of new equations
    unit_2_7 = Only when Right(current_mode);
    (int32_2_5, int32_2_6) = ms(unit_2_7) every not bool_2_4;
    bool_2_4 = True when Right(current_mode);
    unit_2_3 = Only when Left(current_mode);
    (int32_2_1, int32_2_2) = ms(unit_2_3) every not bool_2_0;
    bool_2_0 = True when Left(current_mode);
    bool_1_0 = True;
    -- end of new equations
    current_mode = get_mode(flip_button) every not bool_1_0;
    (m,s) = merge current_mode
            (Left -> (int32_2_1, int32_2_2))
            (Right -> (int32_2_5, int32_2_6));
    cc = make_chess_clock (current_mode, m, s);
tel;
```

Figure 7.6: `chess_clock` après introduction de nouveaux noms de flots

$$\textsc{Const}\ \frac{}{\text{split}\,(k) = ([k])} \qquad \textsc{Var}\ \frac{}{\text{split}\,(x) = ([x])} \qquad \textsc{Fby}\ \frac{\text{split}\,(e_1) = ([e_2])}{\text{split}\,(k\ \texttt{fby}\ e_1) = ([k\ \texttt{fby}\ e_2])}$$

$$\textsc{Op}\ \frac{\text{split}\,(e_1) = (e_2)}{\text{split}\,(o\texttt{(}e_1\texttt{)}) = (o\texttt{(}e_2\texttt{)})}$$

$$\textsc{When}\ \frac{\text{split}\,(e) = (e_1, ..., e_n)}{\text{split}\,(e\ \texttt{when}\ k\texttt{(}x\texttt{)}) = (e_1\ \texttt{when}\ k\texttt{(}x\texttt{)}, ..., e_n\ \texttt{when}\ k\texttt{(}x\texttt{)})}$$

$$\textsc{Tuple}\ \frac{\text{split}\,(e_1) = (l_1) \qquad \text{split}\,(e_2) = (l_2)}{\text{split}\,(e_1\texttt{,}\ e_2) = (l_1 \circ l_2)}$$

$$\textsc{Merge}\ \frac{\forall 1 \leq i \leq n.\ \text{split}\,(e_i) = (e_{i,1}, ..., e_{i,m})}{\text{split}\begin{pmatrix} \texttt{merge}\ x \\ \texttt{(}k_1\texttt{->}e_1\texttt{)} \\ ... \\ \texttt{(}k_n\texttt{->}e_n\texttt{)} \end{pmatrix} = \begin{pmatrix} \texttt{merge}\ x & & \texttt{merge}\ x \\ \texttt{(}k_1\texttt{->}e_{1,1}\texttt{)} & ..., & \texttt{(}k_1\texttt{->}e_{1,m}\texttt{)} \\ ... & & ... \\ \texttt{(}k_n\texttt{->}e_{n,1}\texttt{)} & & \texttt{(}k_n\texttt{->}e_{n,m}\texttt{)} \end{pmatrix}}$$

$$\textsc{Eq}\ \frac{\text{split}\,(e) = (e_1, ..., e_n)}{\text{split}\,((x_1, ..., x_n)\texttt{=}e) = (x_1\texttt{=}e_1\texttt{;}...\texttt{;}x_n\texttt{=}e_n\texttt{;})}$$

$$\textsc{Every}\ \frac{}{\text{split}\,(x_o\texttt{=}f\texttt{(}x_i\texttt{)}\ \texttt{every}\ x_r) = (x_o\texttt{=}f\texttt{(}x_i\texttt{)}\ \texttt{every}\ x_r\texttt{;})} \qquad \textsc{SysO}\ \frac{}{\text{split}\,(\varepsilon_s) = (\varepsilon_s)}$$

$$\textsc{SysS}\ \frac{\text{split}\,(eq) = (sys_{eq}) \qquad \text{split}\,(sys_1) = (sys_2)}{\text{split}\,(eq\texttt{;}\ \ sys_1) = (sys_{eq}\texttt{;}\ \ sys_2)}$$

Figure 7.7: Relation de distribution

est vérifiée dans l'assistant de preuve Coq (ce qui a par ailleurs été fait).

Définition 7.3.1 (Relation de distribution entre deux nœuds)**.** *On définit la relation* $n_s \overset{dist}{\Rightarrow} n_t$ *entre un code d'origine* n_s *et un code produit* n_t *par :*

$$n_s \overset{dist}{\Rightarrow} n_t = \begin{array}{l} \exists sys_1, sys_2, locs, i, o, f. \\ \bigwedge \left\{ \begin{array}{l} \text{split}\,(sys_1) = (sys_2) \\ n_s = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs \\ \texttt{let } sys_1 \texttt{ tel;} \end{array} \right. \\ n_t = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs \\ \texttt{let } sys_2 \texttt{ tel;} \end{array} \right. \end{array} \right. \end{array}$$

Preuve de préservation de sémantique

Théorème 7.3.1 (Préservation de la sémantique par distribution des expressions)**.** *La validation de la relation* $n_s \overset{dist}{\Rightarrow} n_t$ *implique la préservation de la sémantique de* n_s *après transformation en* n_t*.*

$$\forall n_s, n_t.\ \left(n_s \overset{dist}{\Rightarrow} n_t\right) \Rightarrow (n_s \lesssim n_t)$$

Démonstration. Ce théorème est une application directe du lemme 7.3.1.

Lemme 7.3.1 (Préservation de la sémantique par distribution)**.**

$$\begin{array}{l} \forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, e, e_1, ..., e_n, \psi_1, ..., \psi_n. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e \Downarrow \psi_1, ..., \psi_n \ \wedge\ \text{split}\,(e) = (e_1, ..., e_n) \Rightarrow \\ \quad\quad \forall 1 \leq i \leq n.\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_i \Downarrow \psi_i \end{array}$$

L'extension de ce lemme aux systèmes d'équations donne :

$$\begin{array}{l} \forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, sys_1, sys_2.\ \text{split}\,(sys_1) = (sys_2) \ \wedge\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys_1 \Rightarrow \\ \quad\quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys_2 \end{array}$$

preuve du lemme 7.3.1. La preuve se fait par induction sur la relation de distribution. Tous les cas sont traités trivialement. □

□

Exemple 7.3.1 (`chess_clock` normalisé)**.** *Dans notre exemple de pendule aux échecs, seul le nœud* `chess_clock` *est affecté par cette passe de compilation (figure 7.8). Les noms de flots* `m` *et* `s` *ont ainsi été séparés l'un de l'autre.*

```
node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
let
    unit_2_7 = Only when Right(current_mode);
    (int32_2_5, int32_2_6) = ms(unit_2_7) every False(bool_2_4);
    bool_2_4 = True when Right(current_mode);
    unit_2_3 = Only when Left(current_mode);
    (int32_2_1, int32_2_2) = ms(unit_2_3) every False(bool_2_0);
    bool_2_0 = True when Left(current_mode);
    bool_1_0 = True;
    current_mode = get_mode(flip_button) every False(bool_1_0);
    --
    m = merge current_mode (Left -> int32_2_1) (Right -> int32_2_5);
    s = merge current_mode (Left -> int32_2_2) (Right -> int32_2_6);
    --
    cc = make_chess_clock (current_mode, m, s);
tel;
```

Figure 7.8: La distribution de `chess_clock`

7.4 Le changement de syntaxe : de Ls vers Lsn

Le changement de syntaxe est la dernière passe qui se fasse dans le monde flot de données. Après ce changement, on se retrouve donc dans le petit noyau flot de données qu'est Lsn. Ce changement se fait uniquement au niveau de la syntaxe abstraite (c'est-à-dire la représentation interne du programme dans le compilateur), la syntaxe concrète (c'est-à-dire le code tel qu'on peut le lire ou l'écrire) n'est pas affectée, en conséquence, aucun exemple de programme produit n'est donné ici (tout programme se traduirait par son identité).

7.4.1 Présentation de l'algorithme

Le changement de syntaxe de Ls à Lsn est une étape facile à implémenter et à prouver après les deux passes précédentes. Comme pour la passe de distribution, une relation de transition syntaxique a été définie cette fois ci non plus entre deux nœuds de Ls, mais entre un nœud de Ls et un nœud de Lsn. Si cette relation est vérifiée, la sémantique est préservée ; sinon c'est que vraisemblablement l'algorithme d'introduction de nouveaux noms n'a pas bien fait son travail et qu'un des propriétés P_1, P_2, P_3 ou P_4 n'est pas vérifiée (les propriétés P_5 et P_6 concernent des propriétés utiles plus tard, mais pas nécessaires au changement de syntaxe).

Le programme obtenu à la fin, contrairement à ce que laisse supposer le nom de cette passe n'est pas nécessairement une forme normale, puisque le code obtenu à la fin pourrait être encore simplifiable en destructurant complètement le code pour que toute équation ne soit qu'une primitive appliquée à des noms de flots (ce que ferait l'implémentation maximale). Avoir une syntaxe qui refléterait cette situation (et qui serait donc plus restrictive que ne l'est celle de Lsn) conduirait à des preuves plus simples pour la transition syntaxique. En effet, il n'y aurait plus la hiérarchie *atom*, *satom*, *exp*, *cexp*, *eq*, mais simplement *eq* ; quant aux preuves, elles n'auraient plus besoin d'utiliser de récursion, puisqu'aucune expression ne pourrait en contenir une autre. En contrepartie, il y aurait beaucoup plus de noms de flots introduits, et le code produit serait plus dur à comprendre, le nombre de noms de flots introduits peut affecter les performances de certaines analyses telles que des calculs sur des graphes de dépendances.

Relation de transition syntaxique de Ls à Lsn

Cette relation se définit comme une identité pour tout nœud vérifiant les propriétés P_1 à P_4 et est indéfinie dans tous les autres cas.

Toute expression e de Ls qui est soit un nom de flot, soit une constante est convertie en une expression atomique a ($e \xrightarrow{atom} a$). Toute expression normalisée e échantillonnée sur une horloge est convertie en une expression atomique e' échantillonnée sur une horloge (propriété P_4, $e \xrightarrow{satom} sa$). Toute expression normalisée e ne faisant intervenir que des applications d'opérateurs est convertie en expression simple e' (propriétés P_2 et P_3, $e \xrightarrow{exp} e'$). Toute expression e pouvant fusionner des flots est convertie en une expression complexe ce ($e \xrightarrow{cexp} ce$).

Les équations normalisées eq de Ls sont converties en équations eq' de Lsn (propriété P_1, $eq \xrightarrow{eq} eq'$). Et les systèmes normalisés sys de Ls sont convertis en systèmes sys' de Lsn ($sys \xrightarrow{equs} sys'$).

$$\textsc{Var}\ \frac{}{x \xrightarrow{atom} x} \qquad \textsc{Const}\ \frac{}{k \xrightarrow{atom} k}$$

$$\textsc{Atom}\ \frac{e \xrightarrow{atom} a}{e \xrightarrow{satom} a} \qquad \textsc{When}\ \frac{e \xrightarrow{satom} sa}{e\ \texttt{when}\ k(x) \xrightarrow{satom} sa\ \texttt{when}\ k(x)}$$

$$\textsc{Satom}\ \frac{e \xrightarrow{satom} sa}{e \xrightarrow{exp} sa} \qquad \textsc{Op}\ \frac{\forall 1 \leq i \leq n.\ e_{1,i} \xrightarrow{exp} e_{2,i}}{o(e_{1,1}, ..., e_{1,n}) \xrightarrow{exp} o(e_{2,1}, ..., e_{2,n})}$$

$$\textsc{Exp}\ \frac{e_1 \xrightarrow{exp} e_2}{e_1 \xrightarrow{cexp} e_2} \qquad \textsc{Merge}\ \frac{\forall 1 \leq i \leq n.\ e_i \xrightarrow{cexp} ce_i}{\begin{array}{l}\texttt{merge}\ x\ \ (k_1\texttt{->}e_1)\\ \quad\ldots\\ \quad(k_n\texttt{->}e_n)\end{array} \xrightarrow{cexp} \begin{array}{l}\texttt{merge}\ x\ \ (k_1\texttt{->}ce_1)\\ \quad\ldots\\ \quad(k_n\texttt{->}ce_n)\end{array}}$$

$$\textsc{Eq}\ \frac{e \xrightarrow{cexp} ce}{x\texttt{=}e \xrightarrow{eq} x\texttt{=}ce} \qquad \textsc{Fby}\ \frac{}{x_1\texttt{=}k\ \texttt{fby}\ x_2 \xrightarrow{eq} x_1\texttt{=}k\ \texttt{fby}\ x_2}$$

$$\textsc{Every}\ \frac{}{x_o\ \texttt{=}\ f(x_i)\ \texttt{every}\ x_r \xrightarrow{eq} x_o\ \texttt{=}\ f(x_i)\ \texttt{every}\ x_r}$$

$$\textsc{SysO}\ \frac{}{\varepsilon_s \xrightarrow{equs} \varepsilon_s} \qquad \textsc{SysS}\ \frac{eq_1 \xrightarrow{eq} eq_2 \qquad sys_1 \xrightarrow{equs} sys_2}{eq_1\texttt{;}\ sys_1 \xrightarrow{equs} eq_2\texttt{;}\ sys_2}$$

Figure 7.9: Relation de transition de Ls vers Lsn

Un programme qui produit du code qui vérifie cette relation (ou retourne une erreur) a été écrit en Coq. Pour terminer cette passe, il suffit donc de démontrer que cette relation préserve la sémantique.

Définition 7.4.1 (Relation de transition de syntaxe de Ls vers Lsn entre deux nœuds)**.**

On définit la relation $n_s \overset{\substack{Ls \\ Lsn}}{\Rightarrow} n_t$ *entre un code d'origine* n_s *et un code produit* n_t *par :*

$$n_s \overset{\substack{Ls \\ Lsn}}{\Rightarrow} n_t = \begin{array}{l} \exists sys_1, sys_2, locs, i, o, f. \\ \bigwedge \left\{ \begin{array}{l} sys_1 \xrightarrow{sys} sys_2 \\ n_s = \left[\begin{array}{l} \texttt{node } f\texttt{(i) returns (o)} \\ \texttt{var } locs \\ \texttt{let } sys_1 \texttt{ tel;} \end{array} \right. \\ n_t = \left[\begin{array}{l} \texttt{node } f\texttt{(i) returns (o)} \\ \texttt{var } locs \\ \texttt{let } sys_2 \texttt{ tel;} \end{array} \right. \end{array} \right. \end{array}$$

Preuve de préservation de sémantique

La preuve de préservation de la sémantique bénéficie du fait que l'on travaille dans le même cadre malgré le changement de syntaxe. Ce ne sera pas le cas lorsque l'on passe du monde flots de données au monde instantanné.

Théorème 7.4.1 (Préservation de la sémantique par transition de Ls vers Lsn)**.** *La validation de la relation* $n_s \overset{\substack{Ls \\ Lsn}}{\Rightarrow} n_t$ *implique la préservation de la sémantique de* n_s *après transformation en* n_t*.*

$$\forall n_s, n_t.\ \left(n_s \overset{dist}{\Rightarrow} n_t\right) \Rightarrow (n_s \lesssim n_t)$$

Démonstration. Ce théorème est une application directe du lemme 7.4.1 portant sur les systèmes d'équations.

Lemme 7.4.1 (Préservation de la sémantique par transition)**.** *La sémantique des constantes et noms de flots est préservée :*

$$\begin{array}{l} \forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, e_1, e_2, \psi. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_1 \Downarrow [\psi] \ \wedge\ e_1 \xrightarrow{atom} e_2 \Rightarrow \\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_2 \Downarrow \psi \end{array}$$

La sémantique des constantes et noms de flots échantillonnés est préservée :

$$\begin{array}{l} \forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, e_1, e_2, \psi. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_1 \Downarrow [\psi] \ \wedge\ e_1 \xrightarrow{satom} e_2 \Rightarrow \\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_2 \Downarrow \psi \end{array}$$

La sémantique des expressions est préservée :

$$\begin{array}{l} \forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, e_1, e_2, \psi. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_1 \Downarrow [\psi] \ \wedge\ e_1 \xrightarrow{exp} e_2 \Rightarrow \\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_2 \Downarrow \psi \end{array}$$

La sémantique des expressions sous contrôle est préservée :

$$\begin{array}{l} \forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, e_1, e_2, \psi. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} e_1 \Downarrow [\psi] \ \wedge\ e_1 \xrightarrow{cexp} e_2 \Rightarrow \\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} e_2 \Downarrow \psi \end{array}$$

La sémantique des équations est préservée :

$$\begin{array}{l}\forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, eq_1, eq_2. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} eq_1 \ \wedge \ eq_1 \xrightarrow{eq} eq_2 \Rightarrow \\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} eq_2\end{array}$$

La sémantique des systèmes est préservée :

$$\begin{array}{l}\forall \mathcal{E}, \mathcal{S}^{\mathcal{F}}, sys_1, sys_2. \\ \quad \mathcal{S}^{\mathcal{F}} \vdash_{\text{Ls}} sys_1 \Downarrow \ \wedge sys_1 \xrightarrow{sys} sys_2 \Rightarrow \\ \mathcal{S}^{\mathcal{F}} \vdash_{\text{Lsn}} sys_2 \Downarrow\end{array}$$

Preuve du lemme 7.4.1. Chacun des énoncés du lemme se démontre en utilisant le précédent, ainsi qu'une induction sur la relation de transition. La sémantique de Ls ayant été conçue comme extension de celle de Lsn, il suffit de reprendre les définitions des deux sémantiques. □

□

7.5 Conclusion

Cette partie a montré une façon de compiler un nœud de Ls dans Lsn sans en changer la sémantique, ou retourner une erreur dans certains cas, au travers d'une petite chaîne de compilation d'un nœud n_{Ls} vers un nœud n_{Lsn} :

$$n_{\text{Ls}} \overset{subst}{\Rightarrow} n_{subst} \overset{dist}{\Rightarrow} n_{dist} \underset{Lsn}{\overset{Ls}{\Rightarrow}} n_{\text{Lsn}}$$

qui amène à :

$$n_{LS} \lesssim n_{subst} \lesssim n_{dist} \lesssim n_{\text{Lsn}}$$

De plus, bien que ce ne soit pas la finalité de cette compilation, la factorisation d'expressions peut être obtenue gratuitement. L'introduction de nouvelles équations repose sur un outil externe que l'on pourrait paramétrer pour obtenir différentes formes de compilation (par exemple, on pourrait choisir l'implémentation minimale si on veut que la compilation soit rapide, ou choisir une implémentation plus complexe, qui offrira probablement de meilleures performances sur le code produit, mais au coût d'une compilation plus lente).

Toute cette partie a bien été formalisée et prouvée dans l'assistant de preuve Coq.

Troisième partie

Le monde instantané

Chapitre 8

Lsni : syntaxe et sémantique

8.1 Syntaxe de Lsni

La syntaxe de Lsni (figure 8.1) est très proche de celle de Lsn, au point que la seule différence dans la syntaxe se situe au niveau des déclarations. En effet, comme on le verra plus en détails par la suite, Lsni a besoin de référencer ses mémoires. On peut rattacher une mémoire à un nom de flot dans le cas des décalages (`fby`). Par contre, dans le cas des applications de nœuds, la mémoire se retrouve plus naturellement rattachée à l'occurrence du nœud. Comme chaque application d'un nœud n doit avoir son propre état interne, on nomme chacune des occurence de n, afin de pouvoir les distinguer les unes des autres. Ces occurences nommées s'appellent les instances du nœud n. Étant donné une instance j d'un nœud n, on appelle « classe de j » le nœud n.

Exemple 8.1.1 (Instanciation du nœud `chess_clock`)**.** *Dans notre exemple, le nœud* `chess_clock` *contient deux appels au nœud* `ms`*. Chacun de ces appels est nommé (*`ms_2` *et* `ms_5` *en figure 8.2). Le nom d'instance créé est ici le nom du nœud suivi du numéro d'équation où l'instance apparaît. La encore cette convention est arbitraire.*

n	$::=$	`node` f `(`x`:` τ`) returns (`x`:` τ`);`
		`var` x`:` τ `on` k`(`x`);` ...`;` x`:` τ `on` k`(`x`)`
		`ins` j`:` f`;` ...`;` j`:` f
		`let` sys `tel;`
sys	$::=$	eq `;` ...`;` eq
eq	$::=$	x `=` ce | x `=` k `fby` x | x `=` f `(` x `)` `every` k`(`x`)`
ce	$::=$	e | `merge` x `(`k`->` ce `)` ...`(`k`->`ce`)`
e	$::=$	sa | o `(` e `,` ...`,` e `)`
sa	$::=$	a | sa `when` k `(` x `)`
a	$::=$	x | k

Figure 8.1: Syntaxe des nœuds et expressions dans Lsni

```
node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
ins
    ms_2 : ms;
    ms_5 : ms;
    get_mode_8 : get_mode;
let
    unit_2_7 = Only when Right(current_mode);
    (int32_2_5, int32_2_6) = ms_2(unit_2_7) every not bool_2_4;
    bool_2_4 = True when Right(current_mode);
    unit_2_3 = Only when Left(current_mode);
    (int32_2_1, int32_2_2) = ms_5(unit_2_3) every not bool_2_0;
    bool_2_0 = True when Left(current_mode);
    bool_1_0 = True;
    current_mode = get_mode_8(flip_button) every not bool_1_0;
    m = merge current_mode (Left -> int32_2_1) (Right -> int32_2_5);
    s = merge current_mode (Left -> int32_2_2) (Right -> int32_2_6);
    cc = make_chess_clock (current_mode, m, s);
tel;
```

Figure 8.2: L'instanciation dans `chess_clock`

8.2 Typage et bonne formantion dans Lsni

Le typage nécessite un environnement de typage légèrement différent ; en effet, les instances deviennent des objets nommés, il faut donc maintenir des informations statiques qui leur sont associées. Dans notre cas, il n'y a qu'une information qui est la classe de l'instance, c'est-à-dire le nœud instancié.

Définition 8.2.1 (environnement statique local)**.** *Un environnement statique $\mathcal{E}^+$ local est la donnée :*

- *d'un environnement de typage local $Env_{\mathcal{E}^+}$;*
- *d'un domaine de noms d'instances $inst_{\mathcal{E}^+}$;*
- *d'une application qui à tout nom d'instance j de $inst_{\mathcal{E}^+}$ associe un nœud* $\mathrm{inst}_{\mathcal{E}^+}(j)$ *dans N (appellé la classe de l'instance j) ;*
- *d'une application qui à tout nom d'instance j de $inst_{\mathcal{E}^+}$ associe* $\mathrm{Res}_{\mathcal{E}^+}(j)$*, un nom de flot de réinitialisation dans $dom_{Env_{\mathcal{E}^+}}$;*
- *d'une application qui à tout nom d'instance j dans $inst_{\mathcal{E}^+}$ associe un constructeur* $\mathrm{Cst}_{\mathcal{E}^+}(j)$ *de réinitialisation dans K ;*
- *d'une application qui à toute instance j de $inst_{\mathcal{E}^+}$ associe son paramètre d'entrée* $\mathrm{In}_{\mathcal{E}^+}(j)$ *dans $dom_{Env_{\mathcal{E}^+}}$;*
- *d'une preuve que* $\forall j \in inst_{\mathcal{E}^+}, \tau_i, \tau_o. \bigwedge \begin{cases} N_{sig}\left(\mathrm{inst}_{\mathcal{E}^+}(j)\right) = \tau_i \to \tau_o \\ \mathrm{Ktype}\left(\mathrm{Cst}_{\mathcal{E}^+}(j)\right) \in \mathrm{type}_{Env_{\mathcal{E}^+}}\left(\mathrm{Res}_{\mathcal{E}^+}(j)\right) \\ \tau_i = \mathrm{type}_{Env_{\mathcal{E}^+}}\left(\mathrm{In}_{\mathcal{E}^+}(j)\right) \\ \mathrm{ck}\left(\mathrm{In}_{\mathcal{E}^+}(j)\right) = \mathrm{ck}\left(\mathrm{Res}_{\mathcal{E}^+}(j)\right) \end{cases}$
- *d'un domaine de noms de registres $regs_{\mathcal{E}^+}$;*
- *d'une application qui à tout registre x de $regs_{\mathcal{E}^+}$; associe un nom de flot* $\mathrm{next}_{\mathcal{E}^+}(x)$ *dans $dom_{Env_{\mathcal{E}^+}}$*
- *d'une preuve que* $\forall x \in regs_{\mathcal{E}^+}. \bigwedge \begin{cases} x \in dom_{Env_{\mathcal{E}^+}} \\ \mathrm{ck}(x) = \mathrm{ck}\left(\mathrm{next}_{\mathcal{E}^+}(x)\right) \\ \mathrm{type}_{Env_{\mathcal{E}^+}}(x) = \mathrm{type}_{Env_{\mathcal{E}^+}}\left(\mathrm{next}_{\mathcal{E}^+}(x)\right) \end{cases}$

L'environnement statique local est donc une extension de l'environnement de typage local. En première lecture, on peut ne garder à l'esprit que l'application qui à toute instance associe son nœud. Le reste n'est là que pour des détails techniques dans la preuve de préservation de la sémantique.

Les règles de typage sont données en figure 8.3 et figure 8.4. Il n'y a pas de changement avec Lsn pour le typage des expressions. Seul le typage des équations est affecté.

$$\textsc{Const}\ \frac{}{\mathcal{E}^+ \vdash_{\text{LSNI}} k : \text{Ttype}\,(k)\,, \texttt{when Only}\,(\texttt{base})}$$

$$\textsc{Var}\ \frac{x \in dom_{Env_{\mathcal{E}^+}}}{\mathcal{E}^+ \vdash_{\text{LSNI}} x : \text{type}_{Env_{\mathcal{E}^+}}\,(x), \text{ck}\,(x)}$$

$$\textsc{Op}\ \frac{O_{sig}\,(o) = \tau_1, ..., \tau_n \to \tau_o \qquad \forall 1 \leq i \leq n.\quad \mathcal{E}^+ \vdash_{\text{LSNI}} e_i : \tau_i, \text{ck}}{\mathcal{E}^+ \vdash_{\text{LSNI}} o\texttt{(}e_1, ..., e_n\texttt{)} : \tau_o, \text{ck}}$$

$$\textsc{When}\ \frac{x \in dom_{Env_{\mathcal{E}^+}} \qquad \text{type}_{Env_{\mathcal{E}^+}}\,(x) = \text{Ktype}\,(k) \qquad \mathcal{E}^+ \vdash_{\text{LSNI}} e : \tau, \text{ck}\,(x)}{\mathcal{E}^+ \vdash_{\text{LSNI}} e\ \texttt{when}\ k\texttt{(}x\texttt{)} : \tau, \texttt{when}\ k\,(x)}$$

$$\textsc{Merge}\ \frac{\begin{array}{c} x \in dom_{Env_{\mathcal{E}^+}} \qquad \forall \texttt{(}k_i\texttt{->}e_i\texttt{)} \in \mathit{cases}.\ \mathcal{E}^+ \vdash_{\text{LSNI}} e_i : \tau, \texttt{when}\ k_i\,(x) \\ \forall k \in \text{type}_{Env_{\mathcal{E}^+}}\,(x)\,.\ \exists! e_k.\ \texttt{(}k\texttt{->}e_k\texttt{)} \in \mathit{cases} \end{array}}{\mathcal{E}^+ \vdash_{\text{LSNI}} \texttt{merge}\ x\ \mathit{cases} : \tau, \text{ck}\,(x)}$$

Figure 8.3: Règles de typage des expressions de LSNI

$$\textsc{EqExp}\ \frac{x \in dom_{Env_{\mathcal{E}^+}} \qquad x \notin regs_{\mathcal{E}^+} \qquad \mathcal{E}^+ \vdash_{\text{LSNI}} e : \text{type}_{Env_{\mathcal{E}^+}}\,(x)\,, \text{ck}\,(x)}{\mathcal{E}^+ \vdash_{\text{LSNI}} x = e : \{x\}}$$

$$\textsc{EqFby}\ \frac{x \in regs_{\mathcal{E}^+} \qquad \text{Ktype}\,(k) = \text{type}_{Env_{\mathcal{E}^+}}\,(x)}{\mathcal{E}^+ \vdash_{\text{LSNI}} x = k\ \texttt{fby}\ \text{next}_{\mathcal{E}^+}\,(x) : \{x\}}$$

$$\textsc{EqApp}\ \frac{\begin{array}{c} N_{sig}\left(\text{inst}_{\mathcal{E}^+}\,(j)\right) = \tau_i \to \tau_o \qquad x_o \in dom_{Env_{\mathcal{E}^+}} \\ x_o \notin regs_{\mathcal{E}^+} \qquad \text{ck}\,(x_o) = \text{ck}\left(\text{Res}_{\mathcal{E}^+}\,(j)\right) \qquad \text{type}_{Env_{\mathcal{E}^+}}\,(x_o) = \tau_o \end{array}}{\mathcal{E}^+ \vdash_{\text{LSNI}} x_o = j\texttt{(}\,\text{In}_{\mathcal{E}^+}\,(j)\,\texttt{)}\ \texttt{every}\ \text{Cst}_{\mathcal{E}^+}\,(j)\,\texttt{(}\,\text{Res}_{\mathcal{E}^+}\,(j)\,\texttt{)} : \{x_o\}}$$

$$\textsc{SysO}\ \frac{}{\mathcal{E}^+ \vdash_{\text{LSNI}} \varepsilon_s : \emptyset} \qquad \textsc{SysS}\ \frac{\mathcal{E}^+ \vdash_{\text{LSNI}} \mathit{eq} : E_{eq} \qquad \mathcal{E}^+ \vdash_{\text{LSNI}} \mathit{sys} : E_{sys} \qquad \emptyset = E_{eq} \cap E_{sys}}{\mathcal{E}^+ \vdash_{\text{LS}} \mathit{eq};\ \mathit{sys} : E_{eq} \cup E_{sys}}$$

$$\textsc{Node}\ \frac{\begin{array}{c} l = l_1 : \tau_{l_1}; ...; l_r : \tau_{l_r} \\ j = j_1 : f_1; ...; j_n : f_n \qquad \mathit{TEnv}\,(x_i, x_o, l, j) \vdash_{\text{LSNI}} \mathit{sys} : \{o, l_1, ..., l_r\} \end{array}}{\vdash_{\text{LSNI}} \begin{array}{l} \texttt{node}\ f\ \texttt{(}x_i : \tau_i\texttt{)} \\ \texttt{returns (}x_o : \tau_o\texttt{);} \\ \texttt{var}\ l\texttt{;}\ \texttt{ins}\ m\texttt{;} \\ \texttt{let}\ \mathit{sys}\ \texttt{tel;} \end{array} : (\tau_i, \tau_o)}$$

Figure 8.4: Règles de typage des nœuds de LSNI

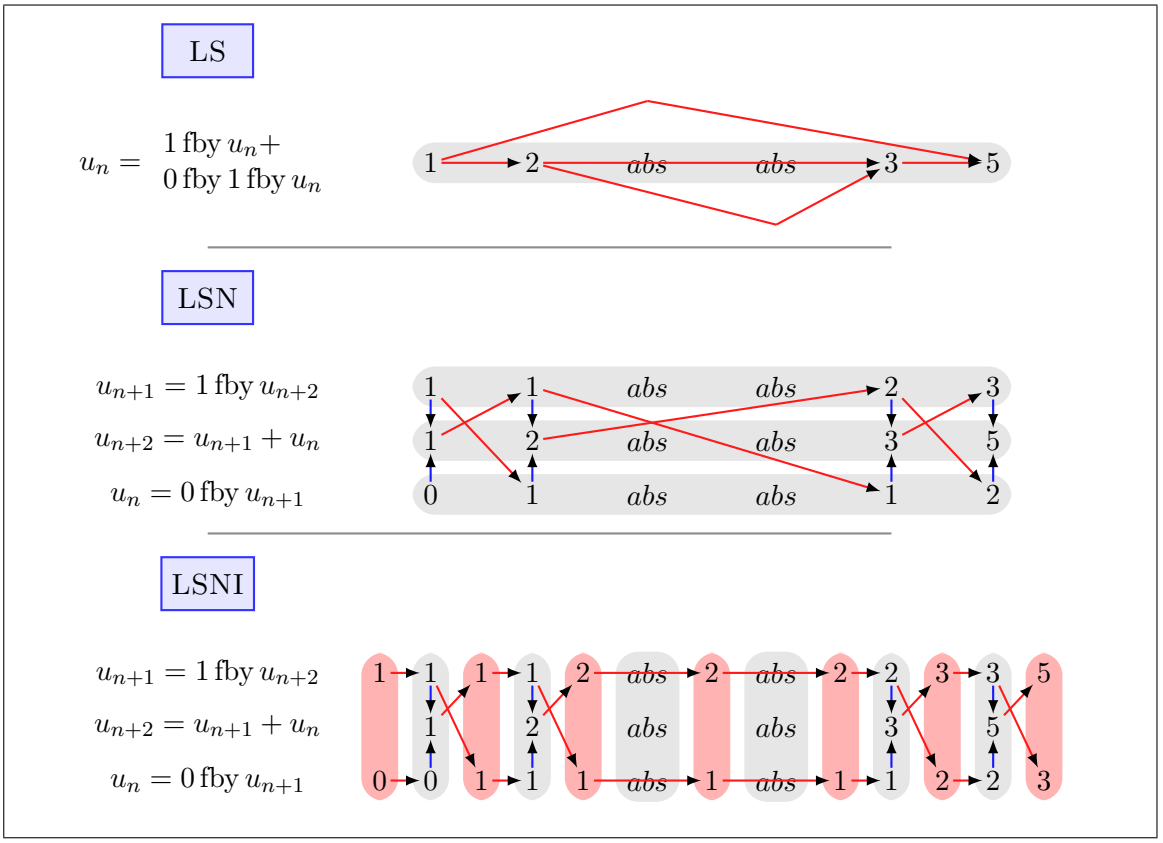


Figure 8.5: La suite de Fibonnacci, exprimée dans LS, LSN et LSNI avec leur représentation sémantique

8.3 Sémantique de Lsni

La sémantique le LSNI marque un changement important. Jusqu'à présent l'objet sémantique de base était le flot de données. Cependant les langages impératifs ne travaillent pas sur des flots de données mais sur des valeurs simples. Il va donc falloir transformer des environnements de flots en flots d'environnements, afin d'exprimer l'évolution du système instant par instant.

Sur la figure 8.5, on peut voir un encodage de la suite de Fibonacci, sur une horloge autre que celle de base afin d'illustrer la gestion des absences. En rouge sont des dépendances par rapport au passé et en bleu des dépendances par rapport au présent (l'instant). Les cadres colorés représentent les entités sémantique considérées. Dans LS et LSN on a un environnement de flots. Dans LSNI, c'est un flot d'environnements. Les cadres roses sont une addition à la sémantique pour LSNI qui complète la perte de l'information de tous les instants passés. C'est une mémoire bornée qui va constituer la mémoire de l'instant. Elle contient les mémoires associées aux délais (`fby`) et aux instances appelées (il n'y en a pas dans la suite de Fibonnacci).

Dans LS, un flot peut faire référence à une valeur arbitrairement lointaine dans le passé (ici on remonte jusqu'à quatre instants en arrière).

Dans Lsn, une conséquence du fait d'avoir remonté les décalages de flots est que cette fois on a juste à remonter un instant présent en arrière, mais à cause des absences, on peut toujours avoir besoin de remonter plusieurs instants absents en arrière.

Dans Lsni, la mémoire partagée entre instants (cadres roses) est matérialisée et insérée entre les instants, cette fois ci on n'a plus besoin de regarder en arrière, il suffit de maintenir à jour cette mémoire pour calculer. On a donc réussi à se défaire du modèle « flots de données », pour passer à un modèle « flots d'instants » dans lequel on peut calculer l'instant suivant sans connaissance du passé.

8.3.1 Environnement et mémoire d'un nœud

Du fait du passage en sémantique instantannée, on redéfinit l'environnement sémantique local de la façon suivante :

Définition 8.3.1. *Étant donné un environnement de typage $\mathcal{E}$, un environnement sémantique local instantané $\mathcal{S}^{\mathcal{I}}$ est la donnée :*
- *d'une application qui à tout nom de flot x dans $dom_{\mathcal{E}}$ associe une valeur présente ou absente* $\mathrm{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x)$ *dans* $[\![\mathrm{type}_{\mathcal{E}}(x)]\!]^{abs}$
- *d'une preuve que :*
  $\forall x, x_{ck}, k_{ck}.\, \mathrm{ck}(x) = \texttt{when}\ k_{ck}(x_{ck}) \Rightarrow \left(\mathrm{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = abs \Leftrightarrow k_{ck} \neq \mathrm{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_{ck})\right)$

Conformément à ce qui a été dit en section 2.3, tout nœud n se voit attribuer une mémoire $N_{mem}(n)$ exprimée comme un produit de types.

Définition 8.3.2. *Étant donné un environnement de typage $\mathcal{E}^+$, une mémoire sémantique locale $\mathcal{M}$ est la donnée :*
- *d'une application qui à tout registre x de $regs_{\mathcal{E}^+}$ associe sa valeur sémantique* $\mathrm{reg}_{\mathcal{M}}(x)$ *dans* $[\![\mathrm{type}_{Env_{\mathcal{E}^+}}(x)]\!]$ *;*
- *d'une application qui à toute instance j de $inst_{\mathcal{E}^+}$ associe la mémoire associée dans* $[\![N_{mem}\left(\mathrm{inst}_{\mathcal{E}^+}(x)\right)]\!]^{abs}$.

Afin de simplifier les preuves et le formalisme, le type de la mémoire sémantique locale est figé dans la suite. Il est donc important d'avoir fait les bons choix de normalisation afin d'optimiser cette mémoire, car il n'est plus question de la modifier par la suite.

Relâcher cette contrainte amènerait à manipuler plus de variantes qu'annoncé en section 2.3.

Comme attendu, on a également besoin de définir une mémoire sémantique locale initiale à partir de la définition du nœud. Du fait des contraintes de typage, si un nœud n est bien typé, il n'existe qu'une seule mémoire sémantique M qui vérifie la propriété inductive $M \vdash_{init} sys$ donnée en figure 8.6, où sys est le système associé au nœud, et $\mathcal{E}^+$ l'environnement de typage local.

Cette mémoire M est la mémoire initiale du nœud.

$$\textsc{EqExp}\ \frac{}{M \vdash_{init} (x) = e} \qquad \textsc{EqFby}\ \frac{}{M \vdash_{init} x = \mathrm{reg}_M(x)\ \ \texttt{fby}\ \ x'}$$

$$\textsc{EqApp}\ \frac{\mathrm{mem}_M(x) = N^{\mathcal{I}}_{init}\left(\mathrm{inst}_{\mathcal{E}^+}(j)\right)}{M \vdash_{init} o = j(x_i)\ \ \texttt{every}\ \ k(x_r)} \qquad \textsc{SysO}\ \frac{}{M \vdash_{init} \varepsilon_s}$$

$$\textsc{SysS}\ \frac{M \vdash_{init} sys \qquad M \vdash_{init} eq}{M \vdash_{init} eq\texttt{;}\ \ sys}$$

Figure 8.6: Spécification de la mémoire initiale d'un nœud

8.3.2 Sémantique des expressions

Les expressions le Lsni, tout comme celles de Lsni, ne font jamais appel à la mémoire sémantique du nœud, la mémoire sémantique sert de glu entre les instants et n'est utile que dans les équations et systèmes d'équation. En conséquence, la sémantique des expressions est celle de Lsn, adaptée aux instants ; elle est donnée en figure 8.7.

8.3.3 Sémantique des équations

Les équations et systèmes d'équations doivent de plus prendre en compte la mémoire sémantique avant et après l'instant ; leur sémantique est donnée en figure 8.8.

On note « $\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\textsc{Lsni}} eq$ » le prédicat selon lequel l'équation eq est vérifiée sur l'environnement sémantique local $\mathcal{S}^{\mathcal{I}}$, la mémoire avant l'instant $\mathcal{M}_b$ et la mémoire après l'instant $\mathcal{M}_a$.

Dans le cas d'une équation simple $x = e$, il y a juste à vérifier que le nom de valeur x et l'expression e s'évaluent à la même valeur dans l'environnement local. Les mémoires n'entrent pas en jeu.

Dans le cas d'un décalage $x_1 = k\texttt{fby}x_2$, du fait des contraintes sur le typage et sur l'environnement sémantique local, x_1 et x_2 sont ou bien tous deux absents, ou bien tous deux présents. Dans le premier cas (règle EqFbyAbs), on demande que le registre associés à x_1 dans la mémoire après l'instant n'ait pas été modifiée par rapport à celle avant l'instant. Dans le second cas (règle EqFbyPres), on demande que x_1 ait la même valeur que le registre qui lui est associé à l'instant précédent, et que x_2 ait la même valeur que le registre associé à x_1 dans l'instant suivant, c'est-à-dire que la mémoire est modifiée pour remplacer l'ancienne valeur de x_1 par la nouvelle valeur de x_2.

Dans le cas de l'application d'une instance $x_o = j(x_i)\ \ \texttt{every}\ \ k(x_r)$, par typage, ou bien x_i, x_o et x_r sont tous simultanément absents, ou bien ils sont tous simultanément présents. Dans le premier cas (règle EqAppAbs), on demande juste que la mémoire associée à l'instance soit conservée avant et après l'instant. Dans le second cas (règle EqAppPres), on se donne une mémoire M, qui est la mémoire initiale dans le cas où x_r s'évalue à k et la mémoire associée à l'instance j dans l'état précédent sinon, et on vérifie que la fonction

$$\textsc{Const}\ \frac{}{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} k \Downarrow k} \qquad \textsc{Var}\ \frac{}{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} x \Downarrow \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x)}$$

$$\textsc{OpAbs}\ \frac{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e_1 \Downarrow abs \qquad ... \qquad \mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e_n \Downarrow abs}{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} o(e_1, ..., e_n) \Downarrow abs}$$

$$\textsc{OpPres}\ \frac{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e_1 \Downarrow \alpha_1 \qquad ... \qquad \mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e_n \Downarrow \alpha_n}{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} o(e_1, ..., e_n) \Downarrow O_{sem}(op)(\alpha_1, ..., \alpha_n)}$$

$$\textsc{WhenAbs}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) \neq k}{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e\ \texttt{when}\ k(x) \Downarrow abs} \qquad \textsc{WhenPres}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = k \qquad \mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e \Downarrow \alpha}{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e\ \texttt{when}\ k(x) \Downarrow \alpha}$$

$$\textsc{MergeAbs}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = abs}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{LSNI}} \texttt{merge}\ x\ (k_1\texttt{->}e_1)...(k_n\texttt{->}e_n) \Downarrow abs}$$

$$\textsc{MergePres}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = k_i \qquad \mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e_i \Downarrow \alpha}{\mathcal{S}^{\mathcal{F}} \vdash_{\text{LSNI}} \texttt{merge}\ x\ (k_1\texttt{->}e_1)...(k_n\texttt{->}e_n) \Downarrow \alpha}$$

Figure 8.7: Sémantique des expressions de LSNI

d'itération appliquée à cette mémoire M et à la valeur de x_i produit bien la mémoire associée à l'instance j dans l'instant d'après, et la valeur associée à x_o.

La vérification d'un système d'équations se fait en considérant toutes les équations en parallèle (règles SysO et SysS). Il n'y a aucun traitement séquentiel. En particulier, une équation ne modifie pas un état en cours qu'elle passe à l'équation suivante.

$$\textsc{EqExp}\ \frac{\mathcal{S}^{\mathcal{I}} \vdash_{\text{LSNI}} e \Downarrow \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x)}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} \texttt{(}x\texttt{)} = e}$$

$$\textsc{EqFbyAbs}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = abs \qquad \text{reg}_{\mathcal{M}_b}(x) = \text{reg}_{\mathcal{M}_a}(x)}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} x = k\ \texttt{fby}\ x'}$$

$$\textsc{EqFbyPres}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = \text{reg}_{\mathcal{M}_b}(x) \qquad \text{reg}_{\mathcal{M}_a}(x) = \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x')}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} x = k\ \texttt{fby}\ x'}$$

$$\textsc{EqAppAbs}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_r) = abs \qquad \text{mem}_{\mathcal{M}_b}(j) = \text{mem}_{\mathcal{M}_a}(j)}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} x_o = j\texttt{(}x_i\texttt{)}\ \texttt{every}\ k\texttt{(}x_r\texttt{)}}$$

$$\textsc{EqAppPres}\ \frac{\begin{array}{c}\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_r) = k' \qquad \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_i) = \alpha_i \\ \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_o) = \alpha_o \qquad ((\alpha_i, M), (\alpha_o, \text{mem}_{\mathcal{M}_a}(j))) \in N^{\mathcal{I}}_{iter}\left(\text{inst}_{\mathcal{E}^+}(j)\right) \\ \left(k' = k \wedge M = \text{mem}_{\mathcal{M}_b}(j)\right) \vee \left(k' \neq k \wedge M = N^{\mathcal{I}}_{init}\left(\text{inst}_{\mathcal{E}^+}(j)\right)\right)\end{array}}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} x_o = j\texttt{(}x_i\texttt{)}\ \texttt{every}\ k\texttt{(}x_r\texttt{)}}$$

$$\textsc{SysO}\ \frac{}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} \varepsilon_s} \qquad \textsc{SysS}\ \frac{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} eq \qquad \mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} sys}{\mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} eq\texttt{;}\ \ sys}$$

$$\textsc{Node}\ \frac{\begin{array}{c}l = l_1 : \tau_{l_1}; ...; l_r : \tau_{l_r} \qquad j = j_1 : f_1; ...; j_s : f_s \\ \mathcal{S}^{\mathcal{I}}, \mathcal{M}_b, \mathcal{M}_a \vdash_{\text{LSNI}} sys \qquad \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_i) = \alpha_i \qquad \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_o) = \alpha_o\end{array}}{\begin{array}{l}\texttt{node}\ f\ \texttt{(}x_i : \tau_i\texttt{)} \\ \texttt{returns (}x_o : \tau_o\texttt{);} \\ \texttt{var}\ l\texttt{;}\ \texttt{ins}\ m\texttt{;} \\ \texttt{let}\ sys\ \texttt{tel;}\end{array} \vdash_{\text{LSNI}} \Downarrow ((\alpha_i, \mathcal{M}_b), (\alpha_o, \mathcal{M}_a))}$$

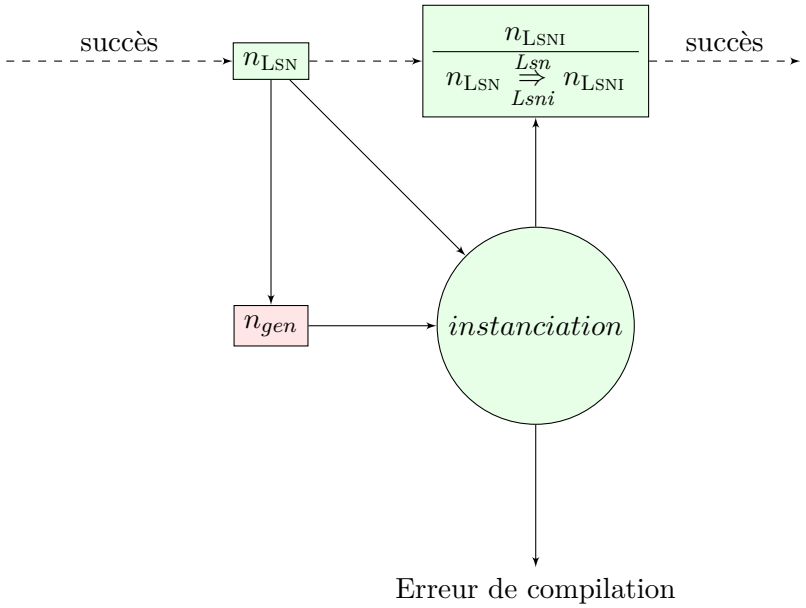
Figure 8.8: Sémantique des nœuds de LSNI

Chapitre 9

Instantiation et ordonnancement : de Lsn à Lsni

9.1 L'instanciation

L'algorithme consiste à repérer toutes les occurrences d'appels de nœud, de leur affecter un nom et de se souvenir de la classe de nœud instanciée. Cette passe n'a aucune subtilité hormis celle de s'assurer que les noms d'instances ne sont pas déjà pris. Elle pourrait donc être programmée directement dans l'assistant de preuve, cependant la création de nouveaux noms en Coq peut être un peu pénible, et il peut être préférable de demander à un outil externe de créer ces noms. Comme pour les passes précédentes, il suffira de mettre en relation le code source et le code généré pour s'assurer la préservation de la sémantique.



Le cadre rose représente le générateur de noms d'instances ; ce dernier a besoin du nœud que l'on veut compiler et passe les nouveaux noms à la passe d'instanciation. Cette

passe peut alors créer un nœud du langage Lsni qui est en relation d'instanciation avec le nœud source, ou produire une erreur.

9.1.1 Relation d'instanciation

$$\textsc{SysO}\ \frac{}{\varepsilon_s \underset{\emptyset}{\rightarrow} \varepsilon_s} \qquad \textsc{Eq}\ \frac{sys_1 \underset{E}{\rightarrow} sys_2}{x\texttt{=}e\texttt{;}\ sys_1 \underset{E}{\rightarrow} x\texttt{=}e\texttt{;}\ sys_2}$$

$$\textsc{Fby}\ \frac{sys_1 \underset{E}{\rightarrow} sys_2}{x_1\texttt{=}k\ \texttt{fby}\ x_2\texttt{;}\ sys_1 \underset{E}{\rightarrow} x_1\texttt{=}k\ \texttt{fby}\ x_2\texttt{;}\ sys_2}$$

$$\textsc{Every}\ \frac{sys_1 \underset{E}{\rightarrow} sys_2 \qquad j \notin E}{x_o\texttt{=}f\texttt{(}x_i\texttt{)}\ \texttt{every}\ x_r \underset{\{j:f\}\cup E}{\rightarrow} x_o\ \texttt{=}\ j\texttt{(}x_i\texttt{)}\ \texttt{every}\ x_r}$$

Figure 9.1: Relation d'instanciation de Lsn vers Lsni

Définition 9.1.1 (Relation d'instanciation entre nœuds)**.** *On définit la relation* $n_s \overset{Lsn}{\underset{Lsni}{\Rightarrow}} n_t$ *entre un code d'origine* n_s *et un code produit* n_t *par :*

$$n_s \overset{Lsn}{\underset{Lsni}{\Rightarrow}} n_t = \exists sys_1, sys_2, locs, i, o, insts, f.\ \bigwedge \begin{cases} sys_1 \underset{insts}{\rightarrow} sys_2 \\ n_s = \left[\begin{array}{l} \texttt{node}\ f\texttt{(}i\texttt{)}\ \texttt{returns}\ \texttt{(}o\texttt{)} \\ \texttt{var}\ locs\texttt{;} \\ \texttt{let}\ sys_1\ \texttt{tel;} \end{array}\right. \\ n_t = \left[\begin{array}{l} \texttt{node}\ f\texttt{(}i\texttt{)}\ \texttt{returns}\ \texttt{(}o\texttt{)} \\ \texttt{var}\ locs\texttt{;}\ \texttt{ins}\ insts\texttt{;} \\ \texttt{let}\ sys_2\ \texttt{tel;} \end{array}\right. \end{cases}$$

Cette relation exprime simplement le fait que toutes les applications de nœuds ont été instanciées, et que l'on a bien déclaré toutes les instances.

Recréer le nœud instancié à partir d'une déclaration d'instances accompagné d'une preuve de vérification de la relation précédemment définie ne pose aucune difficulté, cette passe peut donc se faire en fournissant uniquement une liste de déclarations d'instances, plutôt que de passer en argument le code d'origine complet et le code source produit.

Exemple 9.1.1 (Instanciation de `ms`)**.** *Les exemples des figures 7.8 et 8.2 vérifient bien cette relation. Les nœuds* `base_clock` *et* `get_mode` *restent inchangés car ne contenant pas d'application de nœud. Le code de* `ms` *(figure 7.5) peut être traduit en le code de la figure 9.2.*

```
node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    int32_1_0 : int32;
    bool_3_0 : bool;
ins
    chrono_base_2 : chrono_base;
let
    bool_3_0 = (mins==59) && (secs==59);
    int32_1_0 = chrono_base_2(tic) every reset;
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    reset = False fby bool_3_0;
tel;
```

Figure 9.2: L'instanciation dans `ms`

9.1.2 Preuve de l'algorithme

Théorème 9.1.1 (Préservation de relèvement sémantique)**.** *Étant donnés deux nœuds* n_s *et* n_t*, si* $n_s \overset{Lsn}{\underset{Lsni}{\Rightarrow}} n_t$*, alors la sémantique de* n_s *dans* Lsn *peut se décrire comme un relèvement de la sémantique instantanée de* n_t *dans* Lsni*. Plus formellement,*

$$\forall n_s, n_t.\ \left(n_s \overset{Lsn}{\underset{Lsni}{\Rightarrow}} n_t\right) \Rightarrow (n_s \lesssim n_t)$$

Démonstration. L'idée est de transformer l'environnement de flots dans la sémantique de Lsn en un flot d'environnements et de mémoires de Lsni, puis de démontrer que ce flot d'environnements et de mémoires valide le système dans sa version Lsni. On définit une fonction $\text{slice}_{\mathcal{E}^+}$ qui à tout environnement de flots E et mémoire initiale M_0 associe une erreur ou un flot de mémoires et d'environnements locaux instantanés de la manière suivante :

- si le flot associé au nom `base` est le flot vide, alors on retourne la mémoire initiale et un flot d'environnements vides
- si le flot associé au nom `base` n'est pas vide, on sépare le dernier instant R du reste E' de l'environnement, on construit récursivement un flot d'environnements H et de mémoires $M_h \circ M_b$, et on construit une mémoire M_a à partir de la dernière mémoire M_b en y mettant à jour tous les registres et toutes les instances associées à des valeurs présentes ; on retourne alors $H \cdot R$ et $M_h \cdot M_b \cdot M_a$.

Plus formellement :

$$\textsc{Init}\ \frac{\mathrm{sem}^{\mathcal{F}}_{E}\left(\texttt{base}\right) = \epsilon}{\mathrm{slice}_{\mathcal{E}^+}\left(E, M_0\right) = \left(\epsilon, [M_0]\right)}$$

$$\textsc{Next}\ \frac{\begin{array}{c}\forall x \in \mathit{dom}_{\mathit{Env}_{\mathcal{E}^+}}.\, \mathrm{sem}^{\mathcal{F}}_{E}\left(x\right) = \mathrm{sem}^{\mathcal{F}}_{E'}\left(x\right) \circ \mathcal{I}_R(x) \qquad \mathrm{slice}_{\mathcal{E}^+}\left(E', M_0\right) = \left(H, M_h \cdot M_b\right) \\ \forall x \in \mathit{regs}_{\mathcal{E}^+}. \bigwedge \left\{ \begin{array}{l} \mathrm{sem}^{\mathcal{I}}_{R}\left(x\right) = \mathit{abs} \Rightarrow \mathrm{reg}_{M_b}\left(x\right) = \mathrm{reg}_{M_a}\left(x\right) \\ \mathrm{sem}^{\mathcal{I}}_{R}\left(x\right) \neq \mathit{abs} \Rightarrow \mathrm{reg}_{M_a}\left(x\right) = \mathrm{sem}^{\mathcal{I}}_{R}\left(\mathrm{next}_{\mathcal{E}^+}\left(x\right)\right) \end{array} \right. \\ \forall x \in \mathit{inst}_{\mathcal{E}^+}. \bigwedge \left\{ \begin{array}{l} \mathrm{sem}^{\mathcal{I}}_{R}\left(\mathrm{Res}_{\mathcal{E}^+}\left(x\right)\right) = \mathit{abs} \Rightarrow \mathrm{mem}_{M_b}\left(x\right) = \mathrm{mem}_{M_a}\left(x\right) \\ \mathrm{sem}^{\mathcal{I}}_{R}\left(\mathrm{Res}_{\mathcal{E}^+}\left(x\right)\right) = k \Rightarrow \\ \qquad \exists\, o, M. \bigwedge \left\{ \begin{array}{l} \left(\mathrm{sem}^{\mathcal{I}}_{R}\left(\mathrm{In}_{\mathcal{E}^+}\left(x\right)\right), M, o, M_a\right) \in \mathcal{I}_{\mathrm{inst}_{\mathcal{E}^+}(x)} \\ k = \mathrm{Cst}_{\mathcal{E}^+}\left(x\right) \Rightarrow M = N_{mem}\left(\mathrm{inst}_{\mathcal{E}^+}\left(x\right)\right) \\ k \neq \mathrm{Cst}_{\mathcal{E}^+}\left(x\right) \Rightarrow M = \mathrm{mem}_{M_b}\left(x\right) \end{array} \right. \end{array} \right. \end{array}}{\mathrm{slice}_{\mathcal{E}^+}\left(E, M_0\right) = \left(H \cdot R, M_h \cdot M_b \cdot M_a\right)}$$

Lemme 9.1.1 (Préservation de sémantique par décomposition en instants)**.**

$$\begin{array}{l} \forall \mathcal{E}^+, E, E', M_0, H, R, M_h, M_b, M_a, e, \psi_1. \\ \quad \bigwedge \left\{ \begin{array}{l} \mathrm{slice}_{\mathcal{E}^+}\left(E, M_0\right) = \left(H \cdot R, M_h \cdot M_b \cdot M_a\right) \\ \mathrm{slice}_{\mathcal{E}^+}\left(E', M_0\right) = \left(H, M_h \cdot M_b\right) \\ E \vdash_{\textsc{Lsn}} e \Downarrow \psi_1 \end{array} \right. \\ \quad \Rightarrow \exists \beta, \psi_2.\ E' \vdash_{\textsc{Lsn}} e \Downarrow \psi_2 \wedge R \vdash_{\textsc{Lsni}} e \Downarrow \beta \wedge \psi_1 = \psi_2 \cdot \beta \end{array}$$

et

$$\begin{array}{l} \forall \mathcal{E}^+, E, E', M_0, H, R, M_h, M_b, M_a, eq_1, eq_2. \\ \quad \bigwedge \left\{ \begin{array}{l} M_0 \vdash_{\mathit{init}} eq_1 \\ \mathrm{slice}_{\mathcal{E}^+}\left(E, M_0\right) = \left(H \cdot R, M_h \cdot M_b \cdot M_a\right) \\ \mathrm{slice}_{\mathcal{E}^+}\left(E', M_0\right) = \left(H, M_h \cdot M_b\right) \\ E \vdash_{\textsc{Lsn}} eq_1 \\ eq_1 \overset{\mathit{Lsn}}{\underset{\mathit{Lsni}}{\Rightarrow}} eq_2 \end{array} \right. \\ \quad \Rightarrow E' \vdash_{\textsc{Lsn}} eq_1 \wedge R, M_b, M_a \vdash_{\textsc{Lsni}} eq_2 \end{array}$$

En termes moins formels, si on a un environnement E de flots contenant au moins un instant, on peut sous des hypothèses raisonnables couper E en un environnement E' des flots de E privés de leur dernier instant et en un environnement instantané R accompagné de deux mémoires, de sorte que toute équation valide sous E le reste sous E' et sous R (accompagné de ses mémoires).

La preuve en sera donnée plus loin.

Théorème 9.1.2 (Relévement de sémantique)**.**

$$\begin{array}{l} \forall n_1,\, n_2,\, M_0.\ n_1 \overset{\mathit{Lsn}}{\underset{\mathit{Lsni}}{\Rightarrow}} n_2 \ \wedge\ M_0 \vdash_{\mathit{init}} n_2 \\ \Rightarrow \forall i,\, o.\ (i, o) \in N^{\mathcal{F}}_{\mathit{sem}}\left(n_1\right) \Rightarrow (i, o) \in \mathrm{Lift}\left(M_0, N^{\mathcal{I}}_{\mathit{sem}}\left(n_2\right)\right) \end{array}$$

Démonstration. Soient sys_1 et sys_2 les systèmes associés aux nœuds n_1 et n_2 ; on a alors $sys_1 \underset{Lsni}{\overset{Lsn}{\Rightarrow}} sys_2$ (H_{inst}).

Par définition, il existe un environnement E tel que $E \vdash_{\text{LSN}} sys_1$ et dans lequel l'entrée donne le flot i, et la sortie le flot o.

On pose $\text{slice}_{\mathcal{E}^+}(E, M0) = (H, M_h \cdot M_a)$. On veut alors montrer que $((i,o), M_a) \in \text{Lift}_M\left(M_0, N^{\mathcal{I}}_{sem}(n_2)\right)$ pour pouvoir conclure.

On termine alors en raisonnant par induction sur $\text{sem}^{\mathcal{F}}_E(\texttt{base})$, c'est-à-dire en raisonnant sur le nombre d'instants.

- Le cas de base étant lorsqu'il n'y a aucun instant se démontre sans problèmes.
- Dans le cas récursif, on décompose E en E' ; H en H' et R ; M_h en M'_h et M_b. En généralisant le lemme 9.1.1 aux systèmes d'équations par induction sur H_{inst}, on en déduit :
  (H_{step}) $\quad E' \vdash_{\text{LSN}} sys_1 \wedge R, M_b, M_a \vdash_{\text{LSNI}} sys_2$.
  La première partie de (H_{step}) implique par récurrence que $((i',o'), M_b) \in \text{Lift}_M\left(M_0, N^{\mathcal{I}}_{sem}(n_2)\right)$ où i' et o' sont les flots i et o privés de leur dernier instant.
  De la seconde partie de (H_{step}), on en déduit que $((i'', M_b), (o'', M_a)) \in N^{\mathcal{I}}_{sem}(n_2)$, où i'' et o'' sont les dernières valeurs de i et o.
  En exploitant alors la définition de $\text{Lift}_M\left(M_0, N^{\mathcal{F}}_{sem}(n_2)\right)$, on en conclut que :
  $((i,o), M_a) \in \text{Lift}_M\left(M_0, N^{\mathcal{I}}_{sem}(n_2)\right)$.

□

Enfin, voici la preuve du lemme intermédiaire qui permet de clore le théorème :

preuve du lemme 9.1.1. Le version du lemme sur les expressions se démontre par induction sur l'expression, puis, dans chacun des cas, par induction sur H.

Pour les équations, trois cas se présentent :

- l'équation est une égalité entre un nom de flot et une expression simple.
  Dans ce cas, de l'hypothèse $x = e \vdash_{\text{LSN}}$ on peut déduire l'existence de ψ tel que $e \vdash_{\text{LSN}} \psi \Downarrow$ et $\text{sem}^{\mathcal{F}}_E(x) = \psi$.
  Par la version du lemme sur les expressions, on en déduit l'existence de ψ' et β tels que $e \vdash_{\text{LSN}} \psi' \Downarrow$, $e \vdash_{\text{LSNI}} \beta \Downarrow$ et $\psi = \psi' \cdot \beta$.
  De cette dernière égalité, on en déduit que $\text{sem}^{\mathcal{F}}_{E'}(x) = \psi'$ et $\mathcal{I}_R x = \beta$, ce qui permet de conclure en utilisant les définitions des sémantiques.
- l'équation est une égalité entre un nom de flot et un décalage $(x_1 \texttt{ = } k \texttt{ fby } x_2)$.
  Dans ce cas, on peut en déduire l'existence de ψ_1, ψ_2 et α tels que $\text{sem}^{\mathcal{F}}_E(x_1) = \psi_1$, $\text{sem}^{\mathcal{F}}_E(x_2) = \psi_2$ et $\mathbf{fby}^{\#}_{\boldsymbol{k}}(\psi_2) = (\psi_1, \alpha)$
  Par ailleurs, par induction sur H et en utilisant les propriétés de l'environnement

(voir la définition 8.3.1), on montre que

$$\begin{array}{l}\forall H, E, E', M_0, R, M_h, M_b, M_a.\\ \quad \bigwedge \left\{ \begin{array}{l} \mathrm{reg}_{M_0}(x_1) = k \\ \mathrm{slice}_{\mathcal{E}^+}(E, M_0) = (H \cdot R, M_h \cdot M_b \cdot M_a) \\ \mathrm{slice}_{\mathcal{E}^+}(E', M_0) = (H, M_h \cdot M_b) \end{array} \right. \\ \quad \Rightarrow \mathbf{fby}^{\#}_{\boldsymbol{k}} \left(\mathrm{sem}^{\mathcal{F}}_{E}(x_2)\right) = (\mathrm{sem}^{\mathcal{F}}_{E}(x_1), \mathrm{reg}_{M_a}(x_1)) \wedge \\ \qquad \mathbf{fby}^{\#}_{\boldsymbol{k}} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(x_2)\right) = (\mathrm{sem}^{\mathcal{F}}_{E'}(x_1), \mathrm{reg}_{M_b}(x_1))\end{array}$$

En utilisant alors la définition des sémantiques pour Lsn et Lsni, on peut conclure.

– l'équation est de la forme o `=` x`(`i`) every` k`(`r`)`.
Dans ce cas, on en démontre par induction sur H, et en utilisant les propriétés de l'environnement sémantique local que :

$$\begin{array}{l}\forall H, E, E', M_0, R, M_h, M_b, M_a.\\ \quad \bigwedge \left\{ \begin{array}{l} \mathrm{mem}_{M_0}(x) = N^{\mathcal{I}}_{init}(x) \\ \mathrm{slice}_{\mathcal{E}^+}(E, M_0) = (H \cdot R, M_h \cdot M_b \cdot M_a) \\ \mathrm{slice}_{\mathcal{E}^+}(E', M_0) = (H, M_h \cdot M_b) \end{array} \right. \\ \quad \Rightarrow \bigwedge \left\{ \begin{array}{l} \mathrm{sem}^{\mathcal{I}}_{R}(r) = abs \Rightarrow \\ \qquad \bigwedge \left\{ \begin{array}{l} \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E}(i), k\right) = \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E'}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(i), k\right) \\ \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E}(o), k\right) = \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E'}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(o), k\right) \\ \mathrm{mem}_{M_b}(x) = \mathrm{mem}_{M_a}(x) \end{array} \right. \\ \mathrm{sem}^{\mathcal{I}}_{R}(r) = k \Rightarrow \\ \qquad \exists \alpha_i, \alpha_o. \bigwedge \left\{ \begin{array}{l} \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E}(i), k\right) = \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E'}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(i), k\right) \cdot \alpha_i \\ \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E}(o), k\right) = \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E'}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(o), k\right) \cdot \alpha_o \\ ((\alpha_i, \mathrm{mem}_{M_0}(x)), (\alpha_o, \mathrm{mem}_{M_a}(x))) \in N^{\mathcal{I}}_{iter}(x) \end{array} \right. \\ \mathrm{sem}^{\mathcal{I}}_{R}(r) = k'(\neq k) \Rightarrow \\ \quad \exists \vec{\phi_i}, \phi_i, \alpha_i, \vec{\phi_o}, \phi_o, \alpha_o. \bigwedge \left\{ \begin{array}{l} \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E}(i), k\right) = \vec{\phi_i} \cdot \phi_i \circ \alpha_i \\ \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E'}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(i), k\right) = \vec{\phi_i} \cdot \phi_i \\ \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E}(o), k\right) = \vec{\phi_o} \cdot \phi_o \circ \alpha_o \\ \mathbf{seq}^{\#}_{\mathbf{sem}^{\mathcal{F}}_{\boldsymbol{E'}}(\boldsymbol{r})} \left(\mathrm{sem}^{\mathcal{F}}_{E'}(o), k\right) = \vec{\phi_o} \cdot \phi_o \\ ((\alpha_i, \mathrm{mem}_{M_b}(x)), (\alpha_o, \mathrm{mem}_{M_a}(x))) \in N^{\mathcal{I}}_{iter}(x) \end{array} \right. \end{array} \right.\end{array}$$

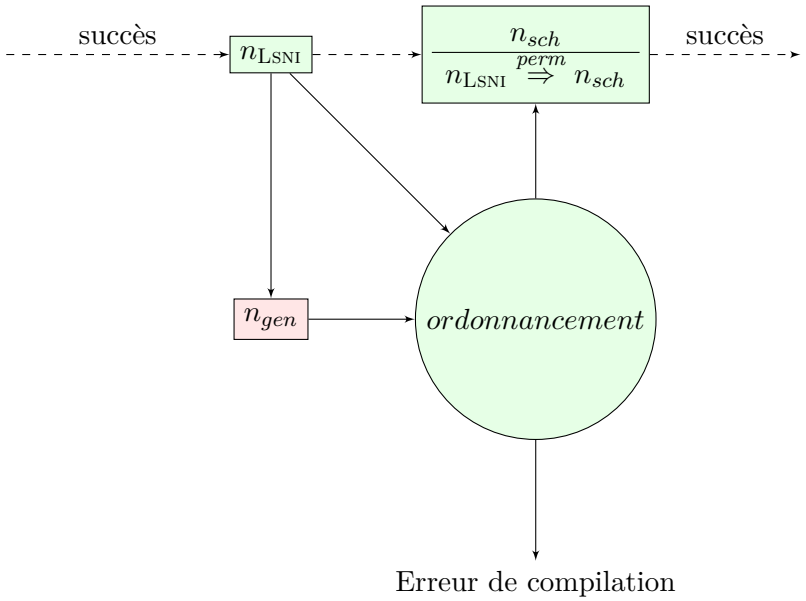
Ceci permet de conclure.

□

□

9.2 L'ordonnancement

L'ordonnancement consiste à changer l'ordre des équations afin de présenter au générateur de code impératif du code dans lequel on accède à des variables toujours initialisées et dans le « bon ordre ».



Comme pour l'introduction de nouveaux noms de flots, faire une passe qui met des équations dans le bon ordre qui soit écrite en Coq avec une preuve de cette spécification n'est pas aisé, et peut relever de choix d'optimisations. On va donc utiliser un programme externe non certifié (cadre rose) qui va choisir un ordre, dont on vérifiera plus tard, au moment du passage vers Obc qu'il vérifie les propriétés souhaitées. Ici l'ordonnanceur se contente de vérifier que la permutation proposée est bien définie, et l'applique pour produire du code que l'on espère bien ordonné.

9.2.1 Objectifs de l'ordonnancement

Après toutes les passes précédentes, on est passé d'un code déclaratif flot de données à un code déclaratif instantané. La dernière étape à franchir pour obtenir du code impératif est donc de passer du déclaratif à l'impératif. Il va donc falloir transformer les équations en affectations, et les noms de valeurs en variable.

Une équation ne modifie pas l'environnement en cours, elle indique l'égalité entre la valeur d'un nom de valeur et la valeur d'une expression. Une affectation, elle, est une modification de l'environnement ; elle indique l'égalité entre la valeur d'une variable après affectation et la valeur d'une expression avant affectation.

Une conséquence immédiate de la distinction entre équation et affectation est qu'alors que les équations n'ont pas besoin d'être ordonnées, les affectations, elles, le nécessitent, car on n'a pas le droit de référencer une variable de l'environnement non encore définie.

Exemple 9.2.1.

```
y=x;
x=3;
```

est un système d'équations correct, mais une suite d'affectations incorrectes car `y` *se voit affecter une valeur indéfinie.*

Il va donc falloir se poser la question de savoir s'il existe un moyen de bouger les affectations de sorte qu'une fois la séquence d'affectations appliquées à un environnement initial, on puisse les voir comme des équations, afin d'assurer la préservation de la sémantique lors du passage au code impératif.

Exemple 9.2.2.

```
x=3;
y=x;
```

Produit la suite d'environnements :

$$E_0 = \{x \mapsto ?, y \mapsto ?\}$$
$$E_1 = \{x \mapsto 3, y \mapsto ?\}$$
$$E_2 = \{x \mapsto 3, y \mapsto 3\}$$

Le dernier environnement est précisément un environnement qui vérifie le système d'équations qu'on s'est donné, et qui est le même à permutation près que celui de l'exemple précédent ; ce nouveau système est donc acceptable vis-à-vis de la préservation de la sémantique.

9.2.2 Quand lire, quand écrire ?

Dans la vision impérative, on part toujours d'un environnement initial sur lequel on ne peut faire aucune supposition. On dit d'un tel environnement n'est pas initialisé.

Supposons dans un premier temps qu'un nœud ne fasse jamais référence à un instant passé (c'est-à-dire ne contienne aucun `fby`). Les conditions de bon ordonnancement sont alors assez claires : l'affectation d'une variable doit se faire avant toute utilisation de cette variable, et c'est la seule règle à respecter. On va donc commencer par les affectations qui ne font entrer en jeu que des constantes et l'entrée, puis celles qui utilisent les nouvelles variables définies et ainsi de suite, jusqu'à ce que toutes les variables soient définies. Le cas où des variables dépendent mutuellement les unes des autres, est un cas qui doit être rejeté par le compilateur ; en effet dans la plupart des cas, cela se traduit soit par une équation sans solution (comme `x=x+1;`), soit par une équation admettant une multitude de solutions (comme `x=x;`) [1].

Supposons maintenant que l'on puisse faire référence au passé. Reprenons l'exemple de la suite de Fibonacci :

$$\begin{cases} u_n &= 0 \texttt{ fby } u_{n+1} \\ u_{n+1} &= 1 \texttt{ fby } u_{n+2} \\ u_{n+2} &= u_n + u_{n+1} \end{cases}$$

La compilation classique en Lustre (V4) donne pour code [2] :

```
typedef struct { int init; int u_n_1; int u_n_2 : int; } fibo_ctx;

int fibo_step(fibo_ctx* ctx) {
```

1. Les exceptions, telles que `x=x;` dans le cas où `x` est de type `unit` ne sont pas intéressantes en pratique, et rejeter de tels programmes est tout à fait raisonnable
2. Après adaptation pour le rendre plus lisible.

```
    int u_n;
    if (ctx->init) {
        ctx->init = 0;
        u_n = 0;
        ctx->u_n_1 = 1;
    }
    else {
        u_n = ctx->u_n_1;
        ctx->u_n_1 = ctx->u_n_2;
    }
    ctx->u_n_2 = u_n + ctx->u_n_1;
    return u_n;
}

void fibo_reset(fibo_ctx* ctx) {
    ctx->init = 1;
}
```

L'état contient alors une variable supplémentaire (`init`) qui indique si l'instance a déjà été exécutée depuis sa dernière réinitialisation, et un test exécuté à chaque itération pour savoir s'il faut exécuter le code de réinitialisation, ou le code d'itération. Pour éviter ces deux inconvénients, on a recours à une astuce : on pré-calcule le futur contenu des registres, ce qui amène au code suivant :

```
                          -- on suppose les valeurs actuelles
                          -- de 'u_n' et de 'u_n_1' deja calculees
    u_n_2 = u_n + u_n_1; -- on calcule donc 'u_n_2'
    u_n   = 0 fby u_n_1; -- on calcule la future valeur de 'u_n'
    u_n_1 = 1 fby u_n_2; -- enfin calcul de la valeur future de 'u_n_1'
```

Cette façon de procéder sera justifiée et détaillée dans le chapitre 11, et produit un code de meilleure qualité :

```
typedef struct { int u_n; int u_n_1 : int; } fibo_ctx;

int fibo_step(fibo_ctx* ctx) {
    int u_n_2;
    int fibo;
    fibo = ctx->u_n;
    u_n_2 = ctx->u_n + ctx->u_n_1;
    ctx->u_n = ctx->u_n_1;
    ctx->u_n_1 = ctx->u_n_2;
    return fibo;
}

void fibo_reset(fibo_ctx* ctx) {
    ctx->u_n = 0;
    ctx->u_n_1 = 1;
}
```

En résumé un ordonanceur correct et efficace doit commencer par ordonner naturellement les équations sans délais (équations simples et applications de nœuds). Une fois cet ordonnancement fait, il faut trier les équations contenant des délais (`fby`) de sorte qu'un nom de valeur défini ne soit pas utilisé par la suite.

$$\textsc{ExtrO}\ \frac{}{\text{insert}\,(sys, 0, eq) = eq\,;\ \ sys} \qquad \textsc{ExtrS}\ \frac{\text{insert}\,(sys_2, n, eq_1) = sys_1}{\text{insert}\,(eq_2\,;\ \ sys_2, n+1, eq_1) = eq_2\,;\ \ sys_1}$$

$$\textsc{Move}\ \frac{\text{insert}\,(sys, m, eq) = sys_1 \qquad \text{insert}\,(sys, n, eq) = sys_2}{sys_1 \overset{m \to n}{\longrightarrow} sys_2}$$

$$\textsc{PermRefl}\ \frac{}{sys \underset{perm}{\sim} sys} \qquad \textsc{PermTrans}\ \frac{sys_1 \underset{perm}{\sim} sys \qquad sys \overset{m \to n}{\longrightarrow} sys_2}{sys_1 \underset{perm}{\sim} sys_2}$$

Figure 9.3: Déplacement d'équations au sein d'un système

On a donc un traitement différent pour les variables locales et celles liées aux mémoires. À la fin de l'itération, toutes les variables locales doivent avoir pour valeur celles qui correspondent aux solutions mathématiques du système d'équations, et sont indéfinies au début de l'itération. Pour les variables liées aux mémoires, elles doivent être solution du système d'équation au début de l'itération, mais à la fin de l'itération elles doivent prendre pour valeur les solutions du système d'équation suivant. En particulier, il est nécessaire d'introduire une variable locale « `int fibo;` » dans le code généré pour garder en mémoire la valeur précédente de `ctx->u_n` et la retourner. Ceci explique le point (P_6) donné en section 7.2.2. On remarque aussi que le système équivalent suivant :

$$\begin{cases} u_n &= 0 \ \texttt{fby}\ u_{n+1} \\ u_{n+1} &= 1 \ \texttt{fby}\ (u_n + u_{n+1}) \end{cases}$$

n'est pas ordonnançable sans introduire de variable intermédiaire, c'est ce qui justifie le point (P_5) de cette même section.

9.2.3 Certificat, et validation de l'ordonnancement

Maintenant que l'on comprend ce qu'est l'ordonnancement, on peut définir quelles informations passer à un programme Coq afin qu'il puisse produire un nœud cible dont la sémantique a été préservée à partir d'un nœud source.

L'opération de base pour l'ordonnancement est le déplacement d'une équation du système ailleurs dans le système ; cette opération peut se décomposer en une étape d'extraction suivie d'une étape d'insertion.

Définition 9.2.1 (Relation d'ordonnancement entre deux nœuds)**.** *On définit la relation*

$n_s \overset{perm}{\Rightarrow} n_t$ *entre un code d'origine* n_s *et un code produit* n_t *par :*

$$n_s \overset{perm}{\Rightarrow} n_t = \begin{array}{l} \exists sys_1, sys_2, locs, i, o, insts, f. \\ \bigwedge \begin{cases} sys_1 \underset{perm}{\sim} sys_2 \\ n_s = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs\texttt{; ins } insts\texttt{;} \\ \texttt{let } sys_1 \texttt{ tel;} \end{array}\right. \\ n_t = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs\texttt{; ins } insts\texttt{;} \\ \texttt{let } sys_2 \texttt{ tel;} \end{array}\right. \end{cases} \end{array}$$

Théorème 9.2.1 (Préservation de la sémantique par permutation d'équations)**.** *La validation de la relation* $n_s \overset{perm}{\Rightarrow} n_t$ *implique la préservation de la sémantique de* n_s *après transformation en* n_t*. Plus formellement, en utilisant les notations de la partie précédente :*

$$\forall n_s, n_t.\ \left(n_s \overset{perm}{\Rightarrow} n_t\right) \Rightarrow (n_s \lesssim n_t)$$

Démonstration. Ce théorème est très simple à démontrer, et découle du lemme 9.2.2.

Lemme 9.2.1 (Préservation par extraction/insertion)**.**

$$\begin{array}{l} \forall \mathcal{E}^+, \mathcal{S}^{\mathcal{I}}, M_b, M_a, sys_1, eq, sys_2, m.\ \text{insert}\,(sys_1, m, eq) = sys_2 \Rightarrow \\ \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_1 \Leftrightarrow \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2 \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq\right)\right) \end{array}$$

Ce lemme signifie juste que si on a un environnement qui valide un système d'équations, alors cet environnement valide n'importe laquelle de ses équations ainsi que le système privé de cette équation.

Démonstration. La preuve se fait par induction sur la relation $\text{insert}\,(sys_1, m, eq) = sys_2$.

– Dans le cas de base ($m = 0$), notre but devient :

$$\begin{array}{l} \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq\texttt{;}\ \ sys_2 \Leftrightarrow \\ \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2 \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq\right) \end{array}$$

C'est en fait la règle SysS de la sémantique de LSNI.

– Dans le second cas ($m = n + 1$), on cherche à montrer :

$$\begin{array}{l} \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq'\texttt{;}\ \ sys_1 \Leftrightarrow \\ \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq'\texttt{;}\ \ sys_2 \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq\right) \end{array}$$

En utilisant la règle SysS, on peut décomposer ce but en :

$$\begin{array}{l} \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq' \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_1\right) \Leftrightarrow \\ \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq' \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2 \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq\right) \end{array}$$

Qui se simplifie alors en :

$$\begin{array}{l} \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_1 \Leftrightarrow \\ \left(\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2 \ \wedge\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq\right) \end{array}$$

Qui est précisément l'hypothèse d'induction.

□

Corollaire 9.2.1 (Préservation par déplacement)**.**

$$\forall \mathcal{E}^+, \mathcal{S}^{\mathcal{I}}, M_b, M_a, sys_1, sys_2, m, n.\ sys_1 \overset{m \to n}{\longrightarrow} sys_2 \Rightarrow$$
$$\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_1 \Rightarrow \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2$$

Ce lemme signifie juste que l'on peut déplacer à son gré une équation au sein d'un système sans en changer la sémantique.

Démonstration. Supposons $sys_1 \overset{m \to n}{\longrightarrow} sys_2$, alors il existe eq et sys tels que insert $(sys_2, sys_1, eq) = m$ (H_1) et insert $(sys, sys_2, eq) = n$ (H_2).

Supposons également $\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_1$ (H_3), nous devons alors montrer que $\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2$.

Par le lemme précédent, H_1 et H_3, on en déduit $\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys$ et $\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq$, hypothèses que l'on peut réappliquer au lemme précédent pour conclure. □

Corollaire 9.2.2 (Préservation par déplacements)**.**

$$\forall \mathcal{E}^+, \mathcal{S}^{\mathcal{I}}, M_b, M_a, sys_1, sys_2.\ sys_1 \underset{perm}{\sim} sys_2 \Rightarrow$$
$$\mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_1 \Rightarrow \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} sys_2$$

Ce lemme signifie juste que l'on peut changer à son gré l'ordre des équations d'un système sans en changer la sémantique.

Démonstration. La preuve se fait par induction sur $sys_1 \underset{perm}{\sim} sys_2$ (c'est à dire par transitivité) en utilisant le lemme précédent ; le cas réflexif étant trivial. □

□

Reste l'algorithme de création du nœud n_t avec garantie qu'il soit une permutation du nœud source n_s.

Donner à l'assistant de preuve les deux nœuds n_s et n_t et lui demander s'ils sont bien en relation de permutation reste envisageable, mais inutilement coûteux. La technique préconisée est de demander à un outil externe (non nécessairement qualifié) une suite de déplacements d'équations, par exemple sous la forme d'une paire de deux entiers.

Il est ensuite aisé d'écrire en Coq une fonction de déplacement d'une équation f : $\forall m, n, sys_1.\{sys_2 | sys_1 \overset{m \to n}{\longrightarrow} sys_2\} + E$ où E est un type qui puisse représenter toutes les erreurs possibles (essentiellement l'un des arguments est supérieur au nombre d'équations du système). La fonction f pourra alors être appelée sur la liste des déplacements tout en produisant une preuve selon laquelle à tout instant, le nœud produit est une permutation du nœud source.

Exemple 9.2.3 (La pendule aux échecs après ordonnancement)**.** *Comme l'introduction de nouveaux noms de flots a pu ajouter des équations sans tenir compte de l'ordonnancement, il est rare que la passe d'ordonnancement ne change rien. Voici donc les nœuds produits en figures 9.4 et 9.5. Pour chaque équation figure à sa droite en commentaire son numéro dans le système d'origine. Les choix de cet ordonnancement seront commentés au chapitre suivant, où sera bien formalisée la notion d'ordonnancement.*

```
node chrono_base (tic:unit) returns (time:int32);
var
    int32_1_0 : int32;
    int32_1_1 : int32;
let
    time = int32_1_1;              -- 3
    int32_1_0 = 1+time;            -- 2
    --
    int32_1_1 = 0 fby int32_1_0;   -- 1
tel;

node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when not real_flip;
    switch_ : mode when real_flip;
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
let
    real_flip = trigger(button_1_0, flip_button);                  -- 4
    current_mode = mode_4_1;                                       -- 7
    keep = current_mode when not real_flip;                        -- 5
    switch_ = flip(current_mode when real_flip);                   -- 6
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_); -- 2
    --
    mode_4_1 = Left fby mode_4_0;                                  -- 1
    button_1_0 = Pushed fby flip_button;                           -- 3
tel;
```

Figure 9.4: `chrono_base` et `get_mode` après ordonnancement

```
node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    int32_1_0 : int32;
    bool_3_0 : bool;
ins
    chrono_base_2 : chrono_base;
let
    int32_1_0 = chrono_base_2(tic) every reset;          -- 2
    mins = int32_1_0/60;                                 -- 3
    secs = int32_1_0%60;                                 -- 4
    bool_3_0 = (mins==59) && (secs==59);                 -- 1
    --
    reset = False fby bool_3_0;                          -- 5
tel;

node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
ins
    ms_2 : ms;
    ms_5 : ms;
    get_mode_8 : get_mode;
let
    bool_1_0 = True;                                                    -- 7
    current_mode = get_mode_8(flip_button) every not bool_1_0;          -- 8
    unit_2_7 = Only when Right(current_mode);                           -- 1
    bool_2_4 = True when Right(current_mode);                           -- 3
    (int32_2_5, int32_2_6) = ms_2(unit_2_7) every not bool_2_4;         -- 2
    unit_2_3 = Only when Left(current_mode);                            -- 4
    bool_2_0 = True when Left(current_mode);                            -- 6
    (int32_2_1, int32_2_2) = ms_5(unit_2_3) every not bool_2_0;         -- 5
    m = merge current_mode (Left->int32_2_1) (Right->int32_2_5);        -- 9
    s = merge current_mode (Left->int32_2_2) (Right->int32_2_6);        --10
    cc = make_chess_clock (current_mode, m, s);                         --11
tel;
```

Figure 9.5: `ms` et `chess_clock` après ordonnancement

9.3 Conclusion

Le langage Lsni est un langage dont l'introduction n'a rien de naturel, et peu de choses y sont faites en dehors de l'ordonnancement qui par ailleurs aurait pu être réalisé dans Lsn. L'ordonnancement a été fait dans Lsni car il va être exploité dans la partie suivante, et il vaut mieux faire cette passe au dernier moment pour ne pas risquer de casser les propriétés souhaitées par l'ordonnanceur en insérant des passes de compilation.

Comme plusieurs passes, celle ci est sujette à des optimisations discutées en annexes et qui ne devraient être étudiées qu'après avoir lu le chapitre suivant afin d'avoir une formalisation précise de ce que l'on attend de l'ordonnancement.

Enfin, il est à noter que cette passe d'ordonnancement ne garantit en aucun cas qu'à la fin de son exécution le système soit bien ordonné. Si ce n'est pas le cas, la passe suivante, c'est-à-dire la génération de code, signalera une erreur, et la compilation échouera. Il convient alors d'ajuster l'outil externe en charge de l'ordonnancement afin qu'il produise un code bien ordonné.

La traduction vers Lsni n'a été que partiellement démontrée en Coq. En fait seule la partie « dure » a été faite, la partie qui vérifie que le code cible et le code produit vérifient bien la relation d'instanciation n'a pas été réalisée par manque de temps.

Chapitre 10

Obc : syntaxe et sémantique

10.1 Syntaxe de Obc

Obc est le langage final de la chaîne de compilation. Il est conçu pour être aisément compilé vers d'autres langages impératifs.

Tout nœud doit pouvoir être initialisé et disposer d'une procédure d'itération ; il peut donc être naturel de les définir dans un langage objet assez simple où toute classe n'a que ces deux méthodes.

En conséquence, les horloges ont disparu du langage, remplacées par des structures de flot de contrôle. La mémoire de chaque nœud est clairement séparée du reste, et ses réinitialisations se font de manière explicite.

La définition de la mémoire est une liste de déclarations de variables avec leur type et leur valeur initiale, et d'instances.

Les variables pointées sont les mémoires associées aux décalages de flots. Elles ne figurent pas dans les variables locales associées à la méthode `step`.

Dans les définitions inductives, de même que ε_s représente la suite vide d'instructions, ε_m représente la mémoire vide.

Exemple 10.1.1 (`chrono_base` et `ms` après génération de code)**.** *Pour exemple, voici en*

n ::= `class` f `{`
`memory {`
md; ...; md
`}`
`step (`x`:` τ`)`
`returns (`x`:` τ`) {`
x`:` τ; ...; x`:` τ;
$stmts$
`}`
`}`
	md ::= x`:` τ `:=` k | j`:` f`.memory`
	
	$stmts$ ::= $stmt$; ...; $stmt$
	
	$stmt$ ::= x `=` e | `mem.`x `=` e | j`.reset` | x `=` j`.step(`e`)`
	| `switch(`e`)`
	`{`k`->` `{`$stmts$`}`... k`->` `{`$stmts$`}}`
	
	e ::= x | `mem.`x | k | o`(`e`,` ...`,` e`)`

Figure 10.1: Syntaxe des classes et expressions dans Obc

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    --
    time = mem.int32_1_1;
    mem.int32_1_1 = 1+time;
  }
}


class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    mem.reset = (mins==59) && (secs==59);
  }
}
```

Figure 10.2: `chrono_base` et `ms` en fin de compilation

figure 10.2 ce que donneront les nœuds `chrono_base` `ms` *à la fin de la chaîne de compilation. On peut y voir que l'appel de nœud se décompose en une possible réinitialisation (selon la valeur de* `mem.reset`*) suivi de l'appel à la méthode d'itération. Les initialisations de* `int32_1_1` *et* `mem.reset` *par* `0` *et* `False` *n'apparaissent plus dans le code d'itération, mais uniquement dans le code d'initialisation.*

10.2 Typage de Obc

Le typage d'Obc se trouve simplifié par rapport à celui de Lsni de par la disparition des horloges. Le typage n'impose plus que toute variable déclarée soit définie et ne force pas non plus que toute variable utilisée ait été initialisée. Obc n'est donc pas un langage qui offre des garanties de sûreté intrinsèquement. Cependant, les programmes issus de la chaîne de compilation garantissent l'accès uniquement à des variables déjà initialisées.

$$\textsc{Const}\ \frac{}{\mathcal{E}^+ \vdash_{\text{OBC}} k : \text{Ttype}\,(k)} \qquad \textsc{Reg}\ \frac{x \in \mathit{regs}_{\mathcal{E}^+}}{\mathcal{E}^+ \vdash_{\text{OBC}} \texttt{mem.}x : \text{type}_{\mathit{Env}_{\mathcal{E}^+}}(x)}$$

$$\textsc{Var}\ \frac{x \in \mathit{dom}_{\mathit{Env}_{\mathcal{E}^+}} \qquad x \notin \mathit{regs}_{\mathcal{E}^+}}{\mathcal{E}^+ \vdash_{\text{OBC}} x : \text{type}_{\mathit{Env}_{\mathcal{E}^+}}(x)}$$

$$\textsc{Op}\ \frac{O_{\mathit{sig}}\,(o) = \tau_1, ..., \tau_n \to \tau_o \qquad \forall 1 \leq i \leq n.\quad \mathcal{E}^+ \vdash_{\text{OBC}} e_i : \tau_i}{\mathcal{E}^+ \vdash_{\text{OBC}} o\texttt{(}e_1, ..., e_n\texttt{)} : \tau_o}$$

Figure 10.3: Règles de typage des expressions de OBC

$$\textsc{AssignVar}\ \frac{x \in \mathit{dom}_{\mathit{Env}_{\mathcal{E}^+}} \qquad x \notin \mathit{regs}_{\mathcal{E}^+} \qquad \mathcal{E}^+ \vdash_{\text{OBC}} e : \text{type}_{\mathit{Env}_{\mathcal{E}^+}}(x)}{\mathcal{E}^+ \vdash_{\text{OBC}} x\texttt{=}e}$$

$$\textsc{AssignReg}\ \frac{x \in \mathit{regs}_{\mathcal{E}^+} \qquad \mathcal{E}^+ \vdash_{\text{OBC}} e : \text{type}_{\mathit{Env}_{\mathcal{E}^+}}(x)}{\mathcal{E}^+ \vdash_{\text{OBC}} \texttt{mem.}x\texttt{=}e}$$

$$\textsc{Switch}\ \frac{\mathcal{E}^+ \vdash_{\text{OBC}} e : \tau \qquad \forall 1 \leq i \leq n.\ \mathcal{E}^+ \vdash_{\text{OBC}} \mathit{stmts}_i\ \wedge\ \text{Ttype}\,(k_i) = \tau}{\mathcal{E}^+ \vdash_{\text{OBC}} \texttt{switch(}e\texttt{)\{}k_1\texttt{->\{}\mathit{stmts}_1\texttt{\}...\{}k_n\texttt{->\{}\mathit{stmts}_n\texttt{\}\}}}$$

$$\textsc{NodeCall}\ \frac{\begin{array}{c} N_{\mathit{sig}}\left(\text{inst}_{\mathcal{E}^+}(j)\right) = \tau_i \to \tau_o \\ x_o \in \mathit{dom}_{\mathit{Env}_{\mathcal{E}^+}} \qquad x_o \notin \mathit{regs}_{\mathcal{E}^+} \qquad \text{type}_{\mathit{Env}_{\mathcal{E}^+}}(x_o) = \tau_o \end{array}}{\mathcal{E}^+ \vdash_{\text{OBC}} x_o = j\texttt{.step(}\,\text{In}_{\mathcal{E}^+}(j)\,\texttt{)}}$$

$$\textsc{Reinit}\ \frac{}{\mathcal{E}^+ \vdash_{\text{OBC}} j\texttt{.reset}}$$

Figure 10.4: Règles de typage des instructions de OBC

$$\textsc{SysO}\ \frac{}{\mathcal{E}^+ \vdash_{\text{OBC}} \varepsilon_s} \qquad \textsc{SysS}\ \frac{\mathcal{E}^+ \vdash_{\text{OBC}} eq \qquad \mathcal{E}^+ \vdash_{\text{OBC}} sys}{\mathcal{E}^+ \vdash_{\text{OBC}} eq;\ sys}$$

$$\textsc{Class}\ \frac{TEnv\,(i,o,l,m) \vdash_{\text{OBC}} stmts}{\vdash_{\text{OBC}} \begin{array}{l}\texttt{class } x \texttt{ \{} \\ \quad \texttt{memory \{}m\texttt{\}} \\ \quad \texttt{step } (i : \tau_i) \texttt{ returns } (o : \tau_o) \texttt{ \{}l\texttt{;}sys\texttt{\}} \\ \texttt{\}}\end{array} : (\tau_i, \tau_o)}$$

Figure 10.5: Règles de typage des blocs et nœuds de OBC

$$\textsc{Const}\ \frac{}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} k \Downarrow k} \qquad \textsc{Var}\ \frac{\text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}\,(x) = \alpha}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} x \Downarrow \alpha} \qquad \textsc{Reg}\ \frac{\text{reg}_M\,(x) = \alpha}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} \texttt{mem.}x \Downarrow \alpha}$$

$$\textsc{Op}\ \frac{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e_1 \Downarrow \alpha_1 \qquad ... \qquad \mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e_n \Downarrow \alpha_n}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} o(e_1, ..., e_n) \Downarrow O_{sem}\,(o)\,(\alpha_1, ..., \alpha_n)}$$

Figure 10.6: sémantique des expressions

10.3 Sémantique de Obc

La sémantique d'OBC repose sur une version plus simple des environnements, puisque ceux ci ne font plus référence aux horloges.

Étant donné un environnement E, on note $E[x \mapsto v]$ l'environnement dans lequel v est associé à x et toutes les autres associations sont celles de E.

On note $abs\mathcal{E}^+$ un environnement sémantique local où toutes les valeurs sont non initialisées (ou absentes). La sémantique de la méthode d'itération commence par un tel environnement et affecte aux variables d'entrées les valeurs passées en argument.

En pratique, comme la valeur *abs* n'apparaît nulle part dans la sémantique, cela implique qu'on peut partir de n'importe quel autre environnement et produire le même résultat, il n'est donc pas nécessaire d'initialiser l'environnement local.

10.3.1 Sémantique des expressions

On note $\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e \Downarrow \alpha$ le prédicat selon lequel l'expression e émet la valeur (non étendue) α sous l'environnement local $\mathcal{S}^{\mathcal{I}}$ et la mémoire M. Les expressions sont donc pures et ne peuvent modifier l'environnement ou la mémoire, contrairement au langage C. La définition de ce prédicat est donnée en figure 10.6.

La sémantique d'une constante k est de produire la constante k.

$$\textsc{AssignVar}\ \frac{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e \Downarrow \alpha}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} x\ \texttt{=}\ e \Downarrow \mathcal{S}^{\mathcal{I}}[x \mapsto \alpha], M}$$

$$\textsc{AssignReg}\ \frac{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e \Downarrow \alpha}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} \texttt{mem.}x\ \texttt{=}\ e \Downarrow \mathcal{S}^{\mathcal{I}}, M[x \mapsto \alpha]}$$

$$\textsc{Switch}\ \frac{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e \Downarrow k_i \qquad \mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} stmts \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M'}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} \texttt{switch(}e\texttt{)}\ \texttt{\{...}k_i\texttt{->\{}stmts\texttt{\}...\}} \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M'}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} e \Downarrow \alpha_1 \\ \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x_o) = \alpha_2 \qquad ((\alpha_1, \text{mem}_M(j)), (\alpha_2, m)) \in N^{\mathcal{I}}_{iter}\left(\text{inst}_{\mathcal{E}^+}(j)\right)\end{array}}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} x_o\texttt{=}j\texttt{.step(}e\texttt{)} \Downarrow \mathcal{S}^{\mathcal{I}}[x_o \mapsto \alpha_2], M[x \mapsto m]}$$

$$\textsc{Reinit}\ \frac{}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} j\texttt{.reset} \Downarrow \mathcal{S}^{\mathcal{I}}, M[x \mapsto N^{\mathcal{I}}_{init}(j)]}$$

Figure 10.7: sémantique des instructions OBC

La sémantique d'une variable simple est de produire la valeur (présente) qui lui est associée dans l'environnement sémantique local. Si la valeur associée est absente, la variable simple n'a pas de sémantique. La preuve de préservation de sémantique durant la génération de code assurre que ce cas de figure ne se présente pas pour les programmes compilés.

La sémantique d'un registre est de produire la valeur qui lui est associée.

10.3.2 Sémantique des instructions et des blocs

Les blocs, comme les instructions modifient l'environnement sémantique local et la mémoire. On note $\mathcal{S}^{\mathcal{I}}_b, M_b \vdash_{\text{OBC}} stmt \Downarrow \mathcal{S}^{\mathcal{I}}_a, M_a$ le prédicat qui dénote le fait que l'instruction (ou le bloc) *stmt* modifie l'environnement local $\mathcal{S}^{\mathcal{I}}_b$ en $\mathcal{S}^{\mathcal{I}}_a$ et la mémoire M_b en M_a. La définition de ces prédicats est donnée en figures 10.7 et 10.8.

La sémantique de l'affectation à une variable simple x d'une expression e est de remplacer le contenu de la variable simple x dans l'environnement sémantique local par la production de l'expression e. L'environnement sémantique local n'est modifié qu'après évaluation de l'expression, et la mémoire n'est pas affectée.

Inversement, l'affectation à un registre d'une expression modifie la mémoire et laisse intact l'environnement sémantique local.

Le branchement par cas (`switch`) évalue d'abord l'expression de contrôle en une constante k puis cherche la branche qui est indexée par cette constante. Le code contenu dans cette branche modifie alors l'environnement sémantique local et la mémoire conformément à sa sémantique.

$$\textsc{StmtO}\ \frac{}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} \varepsilon \Downarrow \mathcal{S}^{\mathcal{I}}, M}$$

$$\textsc{StmtS}\ \frac{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} stmt \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M' \qquad \mathcal{S}^{\mathcal{I}\prime}, M' \vdash_{\text{OBC}} stmts \Downarrow \mathcal{S}^{\mathcal{I}\prime\prime}, M''}{\mathcal{S}^{\mathcal{I}}, M \vdash_{\text{OBC}} stmt;\ \ stmts \Downarrow \mathcal{S}^{\mathcal{I}\prime\prime}, M''}$$

$$\textsc{Step}\ \frac{abs\mathcal{E}^{+}[x_i \mapsto \alpha_i], M_b \vdash_{\text{OBC}} stmts \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M_a \qquad \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}\prime}}(x_o) = \alpha_o}{\begin{array}{l}\texttt{class } x \texttt{ \{} \\ \quad \texttt{memory \{}m\texttt{\}} \\ \vdash_{\text{OBC}} \quad \texttt{step (}x_i : \tau_i\texttt{) returns (}x_o : \tau_o\texttt{)} \ \Downarrow ((\alpha, M_b), (\alpha_o, M_a)) \\ \quad\ \ \texttt{\{}l\texttt{;}\ sys\texttt{\}} \\ \texttt{\}}\end{array}}$$

Figure 10.8: sémantique des blocs et itérations dans OBC

L'appel à l'instance j commence par évaluer les entrées qui lui sont données en une valeur α_1, et consulte la mémoire $\text{mem}_M(j)$ associée à l'instance. Ensuite le couple $(\alpha_1, \text{mem}_M(j))$ est passé en argument à la méthode d'itération de l'instance pour produire le couple (α_2, m) de sortie et de nouvelle mémoire. L'environnement sémantique local est modifié en remplaçant l'ancienne valeur de la variable censée récupérer la sortie par α_2, et la mémoire associé à l'instance j par m.

La réinitialisation de l'instance j n'affecte pas l'environnement sémantique local, mais remplace la mémoire associée à j par la mémoire initiale de sa classe.

Pour ce qui est des blocs d'instructions, toutes les instructions sont exécutées séquentiellement. En absence d'instruction, l'environnement sémantique local et la mémoire ne sont pas affectés. Une instruction *stmt* suivie d'une suite d'instruction *stmts* commence par produire un environnement sémantique local $\mathcal{S}^{\mathcal{I}\prime}$ intermédiaire et une mémoire M' intermédiaire, conformément à la sémantique de *stmt*, puis à partir de cet environnement sémantique local et cette mémoire, on construit un nouvel environnement sémantique local $\mathcal{S}^{\mathcal{I}\prime\prime}$ et une nouvelle mémoire M'', qui sont la production du bloc *stmt*; *stmts*.

La sémantique de la fonction d'itération est alors, partant d'un environnement sémantique ne contenant que les entrées et une mémoire, d'exécuter séquentiellement toutes ses instructions pour obtenir un nouvel environnement sémantique contenant les sorties et une nouvelle mémoire.

Exemple 10.3.1. *Récapitulons cette sémantique sur l'exemple de* `ms`*. On part directement de l'état après 3598 secondes. L'exécution se lit de haut en bas, les deux premières colonnes indiquent la mémoire et l'environnement local en cours. La colonne de droite est l'instruction qui modifie la mémoire et l'environnement local.*

Voici donc comment le 3599e appel à `base_clock` *modifie sa mémoire interne et retourne le temps comptabilisé.*

Mémoire	*Local*	*Instruction*
`mem.int32_1_1`$\mapsto 3599$	`tic`$\mapsto$ `Only` `time`$\mapsto$ *abs*	
		`time = mem.int32_1_1`
`mem.int32_1_1`$\mapsto 3599$	`tic`$\mapsto$ `Only` `time`$\mapsto 3599$	
		`mem.int32_1_1 = 1+time;`
`mem.int32_1_1`$\mapsto 3600$	`tic`$\mapsto$ `Only` `time`$\mapsto 3599$	

$\vdash_{\text{OBC}}$ `chrono_base.step` $\Downarrow$ $(\texttt{Only}, \{\texttt{mem.int32\_1\_1} \mapsto 3599\}),$ $(3599, \{\texttt{mem.int32\_1\_1} \mapsto 3600\})$

À partir de cette exécution, on peut voir comment le 3599^e appel à `ms` *modifie sa mémoire interne et retourne le temps comptabilisé en minutes et secondes. On y voit en particulier que la condition de réinitialisation de* `chrono_base_2` *est calculée non pas dans l'instant même, mais dans l'instant précédent. Par économie de place, cb est un raccourci pour* `chrono_base_2`*.*

Mémoire	*Local*	*Instruction*
	`tic`$\mapsto$ `Only`	
`mem.reset`$\mapsto$ `False`	`int32_1_0`$\mapsto$ *abs*	
cb`.int32_1_1`$\mapsto 3599$	`mins`$\mapsto$ *abs*	
	`secs`$\mapsto$ *abs*	
		`switch(mem.reset)`
		`{True->{`cb`.reset;}}`
	`tic`$\mapsto$ `Only`	
`mem.reset`$\mapsto$ `False`	`int32_1_0`$\mapsto$ *abs*	
cb`.int32_1_1`$\mapsto 3599$	`mins`$\mapsto$ *abs*	
	`secs`$\mapsto$ *abs*	
		`(int32_1_0) =` cb`.step(tic);`
	`tic`$\mapsto$ `Only`	
`mem.reset`$\mapsto$ `False`	`int32_1_0`$\mapsto 3599$	
cb`.int32_1_1`$\mapsto 3600$	`mins`$\mapsto$ *abs*	
	`secs`$\mapsto$ *abs*	
		`mins = int32_1_0/60;`
	`tic`$\mapsto$ `Only`	
`mem.reset`$\mapsto$ `False`	`int32_1_0`$\mapsto 3599$	
cb`.int32_1_1`$\mapsto 3600$	`mins`$\mapsto 59$	
	`secs`$\mapsto$ *abs*	
		`secs = int32_1_0%60;`

$$\frac{\begin{array}{lll} & \texttt{tic} \mapsto \texttt{Only} & \\ \texttt{mem.reset} \mapsto \texttt{False} & \texttt{int32\_1\_0} \mapsto 3599 & \\ \mathrm{cb}.\texttt{int32\_1\_1} \mapsto 3600 & \texttt{mins} \mapsto 59 & \\ & \texttt{secs} \mapsto 59 & \\ & & \texttt{mem.reset = mins==59 \&\& secs==59;} \\ & \texttt{tic} \mapsto \texttt{Only} & \\ \texttt{mem.reset} \mapsto \texttt{True} & \texttt{int32\_1\_0} \mapsto 3599 & \\ \mathrm{cb}.\texttt{int32\_1\_1} \mapsto 3600 & \texttt{mins} \mapsto 59 & \\ & \texttt{secs} \mapsto 59 & \end{array}}{\vdash_{\mathrm{OBC}} \texttt{ms.step} \Downarrow \begin{array}{l} (\texttt{Only}, \{\texttt{mem.reset} \mapsto \texttt{False}; \mathrm{cb}.\texttt{int32\_1\_1} \mapsto 3599\}), \\ ((59, 59), \{\texttt{mem.reset} \mapsto \texttt{True}; \mathrm{cb}.\texttt{int32\_1\_1} \mapsto 3600\}) \end{array}}$$

Chapitre 11

Génération de code : de Lsni à Obc

11.1 Traduction de Lsni à Obc

La traduction vers du code impératif va faire deux choses simultanément, la première est la traduction proprement dite, cette traduction est suffisamment simple pour intégrer naturellement la seconde, à savoir la vérification du bon ordonnancement. Des passes supplémentaires se chargent ensuite de rendre cette traduction moins grossière.

Conformément à ce qui a été annoncé dans la présentation de l'ordonnancement dans la partie précédente, les variables ont un cycle de vie. On va avoir besoin d'un ensemble V dans les prédicats suivants qui contient l'ensemble des variables utilisables pour une équation considérée. De même on définit une paire d'ensembles des instances et registres déjà définis (I, R).

V est le point central de ce chapitre, il met en relation l'environnement issu de Lsni, ainsi que celui de Obc couplé à sa mémoire via une relation de compatibilité.

Définition 11.1.1 (Compatibilité entre environnements)**.** *Étant donnés deux environnements sémantiques locaux E_{Lsni} et E_{Obc} ainsi que trois mémoires M_b, M_a et M_{Obc} on dit que $(E_{\text{Lsni}}, M_b, M_a)$ et $(E_{\text{Obc}}, M_{\text{Obc}})$ sont compatibles sur V et (I, R) si les trois propriétés suivantes sont vérifiées :*

Localisation : *Toutes les variables de V ont la même valeur, lorsqu'elles sont présentes, dans E_{Lsni} et E_{Obc} ou M_{Obc} s'il s'agit d'une mémoire.*
Formellement :

$$\forall x \in V, \alpha.\ \text{sem}^{\mathcal{I}}_{E_{\text{Lsni}}}(x) = \alpha \Rightarrow \bigwedge \begin{cases} x \in \mathit{regs}_{\mathcal{E}^+} \Rightarrow \text{reg}_{M_{\text{Obc}}}(x) = \alpha \\ x \notin \mathit{regs}_{\mathcal{E}^+} \Rightarrow \text{sem}^{\mathcal{I}}_{E_{\text{Obc}}}(x) = \alpha \end{cases}$$

Stabilité : *Toutes les variables de V ont aussi leur horloge dans V, et aucune variable ne peut se trouver simultanément dans V et dans R.*
Formellement :

$$\forall x \in V, x_{ck}, k_{ck}.\ \text{ck}(x) = \texttt{when}\ k_{ck}(x_{ck}) \Rightarrow x_{ck} \in V\ \wedge\ x \notin R$$

$$\textsc{Const}\ \frac{}{\mathrm{Trad}_{loc}(V,k)=k} \qquad \textsc{Var}\ \frac{x\notin regs_{\mathcal{E}^+} \qquad x\in V}{\mathrm{Trad}_{loc}(V,x)=x} \qquad \textsc{Mem}\ \frac{x\in regs_{\mathcal{E}^+} \qquad x\in V}{\mathrm{Trad}_{loc}(V,x)=\texttt{mem.}x}$$

$$\textsc{When}\ \frac{\mathrm{Trad}_{loc}(V,e)=e' \qquad x\in V}{\mathrm{Trad}_{loc}(V,e\ \texttt{when}\ k\texttt{(}x\texttt{)})=e'}$$

$$\textsc{Op}\ \frac{\mathrm{Trad}_{loc}(V,e_1)=e'_1 \qquad ... \qquad \mathrm{Trad}_{loc}(V,e_n)=e'_n}{\mathrm{Trad}_{loc}(V,o\texttt{(}e_1,...,e_n\texttt{)})=o\texttt{(}e'_1,...,e'_n\texttt{)}}$$

Figure 11.1: traduction des expressions simples de Lsni vers Obc

Mémorisation : *Toutes les instances et tous les registres dans (I,R) ont les mêmes valeurs que dans M_a ; alors que celles qui ne sont pas dans (I,R) ont les mêmes valeurs que dans M_b.*
Formellement,

$$\forall x\in inst_{\mathcal{E}^+}.\ \bigwedge\begin{cases} x\in I \Rightarrow \mathrm{mem}_{\mathcal{E}^+}(M_{\mathrm{OBC}})\,x=\mathrm{mem}_{\mathcal{E}^+}(M_a)\,x\\ x\notin I \Rightarrow \mathrm{mem}_{\mathcal{E}^+}(M_{\mathrm{OBC}})\,x=\mathrm{mem}_{\mathcal{E}^+}(M_b)\,x\end{cases}$$

et

$$\forall x\in regs_{\mathcal{E}^+}.\ \bigwedge\begin{cases} x\in R \Rightarrow \mathrm{reg}_{M_{\mathrm{OBC}}}(x)=\mathrm{reg}_{M_a}(x)\\ x\notin I \Rightarrow \mathrm{reg}_{M_{\mathrm{OBC}}}(x)=\mathrm{reg}_{M_b}(x)\end{cases}$$

Cette relation est notée $(E_{\mathrm{LSNI}},M_b,M_a)\overset{(I,R)}{\underset{V}{\approx}}(E_{\mathrm{OBC}},M_{\mathrm{OBC}})$

11.1.1 Traduction des expressions simples

Étant donné un ensemble V de variables et deux expressions e_1 et e_2, on dit que e_1 est traduisible en e_2 au travers des variables vivantes V, et on note $\mathrm{Trad}_{loc}(V,e_1)=e_2$, si les règles en figure 11.1 sont vérifiées.

On se contente d'effacer les horloges (When), de reconnaitre les mémoires du nœud (Mem), de vérifier l'utilisation de variables vivantes uniquement (Var et Mem), et de s'assurer que toutes les variables de e_1 soient bien vivantes (c'est à dire font partie de l'ensemble V).

Théorème 11.1.1 (Préservation de la sémantique d'expressions simples)**.**

$$\begin{array}{l}\forall E_{\mathrm{LSNI}},M_b,M_a,E_{\mathrm{OBC}},M_{\mathrm{OBC}},V,I,R,e,e',\alpha.\\ \quad (E_{\mathrm{LSNI}},M_b,M_a)\overset{(I,R)}{\underset{V}{\approx}}(E_{\mathrm{OBC}},M_{\mathrm{OBC}})\ \wedge\ \mathrm{Trad}_{loc}(V,e)=e'\ \wedge\ E_{\mathrm{LSNI}}\vdash_{\mathrm{LSNI}} e\Downarrow\alpha\Rightarrow\\ \quad\quad E_{\mathrm{OBC}},M_{\mathrm{OBC}}\vdash_{\mathrm{OBC}} e\Downarrow\alpha\end{array}$$

Démonstration. Les preuves se font par induction sur la relation de traduction sans aucune difficulté. □

$$\textsc{BSelf}\ \frac{}{x \in \text{birth}\,(x)}$$

$$\textsc{BTrans}\ \frac{\text{ck}\,(x') = \texttt{when}\ \ k_{ck}\,(x_{ck}) \qquad x_{ck} \in \text{birth}\,(x) \qquad x' \in \mathit{regs}_{\mathcal{E}^+}}{x' \in \text{birth}\,(x)}$$

$$\textsc{DSelf}\ \frac{}{x \in \text{death}\,(x)} \qquad \textsc{DTrans}\ \frac{\text{ck}\,(x') = \texttt{when}\ \ k_{ck}\,(x_{ck}) \qquad x_{ck} \in \text{death}\,(x)}{x' \in \text{death}\,(x)}$$

Figure 11.2: Définition formelle inductive des ensembles $\text{birth}\,(x)$ et $\text{death}\,(x)$

11.1.2 Traduction d'une équation

La traduction d'une équation est assez complexe, il faut tenir compte à la fois de l'ordre des instructions et de savoir si une instruction donnée est ou non active.

La passe précédente, c'est à dire l'ordonnancement a changé l'ordre des équations afin de régler le premier point. Cependant, aucune contrainte sur l'ordre dans lequel les équations ont été ordonnées n'a été vérifiée. C'est donc la passe de traduction qui va faire la vérification du bon ordonnancement en même temps que la traduction. Pour régler le second point, c'est à dire s'assurrer de la bonne activation des instructions, il va falloir exploiter les informations sur les horloges.

La traduction va donc se dérouler en deux temps. Dans un premier temps, une instruction sera traduite en tenant compte des contraintes d'ordonnancement, et dans un second temps, on lui ajoutera les structures de contrôle pour s'assurrer que les instructions soient bien actives comme il le faut.

11.1.2.1 Vie et mort des variables

Les variables ont une durée de vie limitée par celle de la variable sur laquelle elles sont échantillonnées.

Toute variable mémoire m, ayant été définie à l'instant précédent naît en même temps que la variable v sur laquelle elle est échantillonnée, mais meurt avant elle, au moment de la définition de m. Toute variable locale n'ayant pas encore été définie naît après sa définition, et ne meurt qu'en même temps que la variable sur laquelle elle est échantillonnée.

La naissance d'une variable mémoire ou locale entraîne donc celle de toutes les mémoires qu'elle échantillonne, alors que la mort d'une variable mémoire entraîne la mort de toutes les variables mémoires ou locales qu'elle échantillonne.

On note $\text{birth}\,(x)$ l'ensemble des variables qui naissent en même temps que x et $\text{death}\,(x)$ celles qui meurent en même temps.

Les définitions formelles inductives de $\text{birth}\,(x)$ et de $\text{death}\,(x)$ sont données en figure 11.2.

$$\textsc{NoMerge}\ \frac{\mathrm{Trad}_{loc}\,(V, e) = e'}{\mathrm{Trad}_{loc}\,(V, x\texttt{=}e) = x\texttt{=}e'}$$

$$\textsc{Merge}\ \frac{\mathrm{Trad}_{loc}\,(V, x = e_1) = stmt_1 \quad \ldots \quad \mathrm{Trad}_{loc}\,(V, x = e_n) = stmt_n}{\mathrm{Trad}_{loc}\left(V,\ \begin{array}{lll} x\texttt{=merge } x' & (k_1\texttt{->}e_1) \\ & \ldots \\ & (k_n\texttt{->}e_n) \end{array}\right) = \begin{array}{ll} \texttt{switch(}x'\texttt{)\{} & k_1\texttt{->\{}stmt_1\texttt{\}} \\ & \ldots \\ & k_n\texttt{->\{}stmt_n\texttt{\}\}} \end{array}}$$

Figure 11.3: traduction d'une équation simple

$$\textsc{Eq}\ \frac{\mathrm{Trad}_{loc}\,(V, x = eq) = stmt \qquad \mathrm{ck}\,(x) = \texttt{when } k_{ck}\,(x_{ck}) \qquad x_{ck} \in V}{\mathrm{Trad}^{-}\,(V, IR, x = eq) = (\varepsilon_m, V \cup \mathrm{birth}\,(x), IR, (stmt, x_{ck}, k_{ck}))}$$

$$\textsc{Mem}\ \frac{\mathrm{Trad}_{loc}\,(V, x_2) = e \qquad \mathrm{ck}\,(x_1) = \texttt{when } k_{ck}\,(x_{ck}) \qquad x_{ck} \in V \qquad W = V/\,\mathrm{death}\,(x_1)}{\mathrm{Trad}^{-}\,(V, (I, R), x_1 \texttt{ = } k \texttt{ fby } x_2) = (x_1\texttt{=}k, W, (I, R \cup \{x_1\}), (\texttt{mem.}x_1 \texttt{ = } e, x_{ck}, k_{ck}))}$$

$$\textsc{App}\ \frac{\begin{array}{c} eq = (x_o \texttt{ = } j\texttt{.step(}x_i\texttt{) every } k\texttt{(}x_r\texttt{)}) \\ x_o \notin V \qquad \mathrm{ck}\,(x_r) = \texttt{when } k_{ck}\,(x_{ck}) \qquad x_{ck} \in V \qquad \mathrm{Trad}_{loc}\,(V, x_i) = e \\ W = \mathrm{birth}\,(x_o) \qquad stmt = (\texttt{switch(}x_r\texttt{) \{}k\texttt{-> \{}j\texttt{.reset\}\}; } x_o \texttt{ = } j\texttt{.step(}x_i\texttt{))} \end{array}}{\mathrm{Trad}^{-}\,(V, (I, R), eq) = (x\texttt{.memory}, V \cup W, (I \cup \{j\}, R), (stmt, x_{ck}, k_{ck}))}$$

Figure 11.4: traduction sans contrôle des équations

11.1.2.2 Traduction sans contrôle

Dans un premier temps, on définit la traduction d'une équation ne modifiant pas la mémoire. $\mathrm{Trad}_{loc}\,(V, eq) = stmt$ dénote le fait eq puisse être traduit en $stmt$ avec l'ensemble V des variables vivantes. La définition de cette relation est donnée en figure 11.3. Pour la traduction d'une équation simple, il suffit donc d'enfoncer les affectations à l'intérieur des branches du `merge`.

Maintenant, donnons celle plus générale qui permette la traduction de toute équation en instructions, sans s'occuper pour l'instant des structures de contrôle. Ca traduction doit modifier V, l'ensemble des variables vivantes. On note $\mathrm{Trad}^{-}\,(V, (I, R), eq) = (m, V', (I', R'), (stmt, x, k))$ le prédicat selon lequel la traduction de eq, échantillonnée sur x par k, produit $stmt$, transforme V en V' et I en I' et définit le fragment de mémoire m ; les règles sont données en figure 11.4.

Comme annoncé en sous-section 11.1.2.1, une équation avec une variable simple fait naître toutes les variables mémoires qui en dépendent (Eq). L'interprétation de cette règle est que précédement, on ne pouvait pas lire ces mémoires, car on ne savait pas encore si leur horloge était ou non active. Une équation avec une variable mémoire fait disparaître toutes les variables dont elle dépend (Mem), et ajoute cette variable mémoire

à R. L'interprétation en est qu'on ne peut plus savoir si ces variables sont ou non actives, on sait seulement si elles le seront à l'instant suivant. Enfin, l'application d'instance se traduit en ajoutant une structure de contrôle pour décider si oui ou non il faut réinitialiser la mémoire associée (App), et ajoute cette instance à I. Comme dans le cas des équations simples, aucune variable ne meurt, en revanche toutes les variables échantillonées sur la sortie de l'application naissent.

Lemme 11.1.1 (Préservation de la sémantique d'équation (sans contrôle))**.** *Soient un environnement* $(E_{\text{LSNI}}, M_b, M_a)$ *et une paire* (E, M) *compatible sous* V *et* (I, R) *; si une équation eq, échantillonée sur* x *par* k*, se traduit sans contrôle en une instruction stmt en produisant* V' *et* (I', R')*, et que de plus l'équation eq est vérifiée dans l'environnement, alors les deux propriétés suivantes sont vérifiées :*

cas passif *si l'équation met en relation des valeurs absentes, alors* $(E_{\text{LSNI}}, M_b, M_a)$ *et* (E, M) *sont compatibles sous* V' *et* (I', R')*.*

cas actif *si l'équation met en relation des valeurs présentes, alors stmt produit un environnement* E' *et une mémoire* M' *compatibles avec* E_{LSNI} *sous* V' *et* (I', R')

Preuve du lemme 11.1.1. L'énoncé formel de ce lemme est le suivant :

$$\forall E_{\text{LSNI}}, M_b, M_a, E, M, V, V', I, R, I', R', eq, stmt, x, k, m.$$
$$\bigwedge \begin{cases} (E_{\text{LSNI}}, M_b, M_a) \overset{(I,R)}{\underset{V}{\approx}} (E, M) \\ \text{Trad}^{-}(V, (I, R), eq) = (m, V', (I', R'), (stmt, x, k)) \\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq \end{cases}$$
$$\Rightarrow \bigwedge \begin{cases} m \vdash_{init} eq \\ \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) \neq k \Rightarrow (E_{\text{LSNI}}, M_b, M_a) \overset{(I',R')}{\underset{V'}{\approx}} (E, M) \\ \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x) = k \Rightarrow \exists E', M'. \bigwedge \begin{cases} E, M \vdash_{\text{OBC}} stmt \Downarrow E', M' \\ (E_{\text{LSNI}}, M_b, M_a) \overset{(I',R')}{\underset{V'}{\approx}} (E', M') \end{cases} \end{cases}$$

- $eq : x'\texttt{=}e$ et $\text{ck}(x') = \texttt{when}\ k(x)$
  Dans ce cas, m est vide, $I' = I$, $R' = R$ et $V' = V \cup \text{birth}(x')$.
  Si $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) \neq k$ alors on doit juste vérifier la compatibilité sous V'. Par les contraintes d'horloges sur les environnements, on a $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x') = abs$. Comme toutes les autres variables de $\text{birth}(x')$ sont des registres qui ne peuvent pas faire partie de R, celles qui sont présentes coïncident avec leur valeur dans M_b et donc aussi dans M. On garde donc la compatibilité.
  Si par contre $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) = k$ alors en posant $M' = M$ et $E' = E[x' \mapsto \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(x')]$, on a toujours $(E_{\text{LSNI}}, M_b, M_a) \overset{(I',R')}{\underset{V'}{\approx}} (E', M')$; de plus, par le lemme 11.1.1, et en raisonnant par induction sur les `merge` de l'expression e, on démontre que $E, M \vdash_{\text{OBC}} stmt \Downarrow E', M'$
- $eq : x_1\texttt{=}k\ \texttt{fby}\ x_2$ et $\text{ck}(x_1) = \texttt{when}\ k(x)$
  Dans ce cas, $m = x_1\texttt{:=}k$, et on a bien $m \vdash_{init} eq$, $I' = I$, $R' = R \cup \{x_1\}$ et $V' = V/\{\text{death}(x_1)\}$.
  Dans le cas passif, on a juste la compatibilité à vérifier, et là encore elle se démontre sans difficulté.

$$\textsc{Base}\ \frac{}{\mathrm{Ctrl}\left((stmt, \texttt{base}, \texttt{Only})\right) = stmt}$$

$$\textsc{NotBase}\ \frac{\mathrm{ck}\left(x\right) = \texttt{when}\ \ k_{ck}\left(x_{ck}\right) \qquad \mathrm{Ctrl}\left((\texttt{switch(}x\texttt{)\{}k\texttt{->\{}stmt\texttt{\}\}}, x_{ck}, k_{ck}\right)) = stmt'}{\mathrm{Ctrl}\left((stmt, x, k)\right) = stmt'}$$

$$\textsc{Eq}\ \frac{\mathrm{Trad}^{-}\left(V, IR, eq\right) = \left(m, V', IR', ckstmt\right) \qquad \mathrm{Ctrl}\left(ckstmt\right) = stmt'}{\mathrm{Trad}^{+}\left(V, IR, eq\right) = \left(m, V', IR', stmt'\right)}$$

Figure 11.5: ajout du contrôle dans les instructions

Dans le cas actif, il faut prendre $E' = E$ et $M' = M[x_1 \mapsto \mathrm{sem}^{\mathcal{I}}_{E_{\mathrm{LSNI}}}(x_1)]$. La compatibilité se démontre alors facilement, de même que la relation $E, M \vdash_{\mathrm{OBC}} stmt \Downarrow E', M'$.

– $eq : x_o = j(x_i)$ `every` $k(x_r)$ et $\mathrm{ck}\left(x_r\right) = \texttt{when}\ \ k\left(x\right)$
Dans ce cas $m = j$`.memory` et on a bien $m \vdash_{init} eq$, $I' = I \cup \{j\}$, $R' = R$ et $V' = V \cup \mathrm{birth}\left(x_o\right)$.
Dans le cas passif, la sortie est absente, et la mémoire de l'instance reste la même dans M_b et M_a, la compatibilité est donc vérifiée.
Dans le cas actif, il y a deux cas à considérer. Dans le premier cas, la variable de réinitialisation diffère de la constante de réinitialisation, et le code contenu dans le `switch` n'est pas exécuté. Les règles de sémantique et la compatibilité avant exécution du code permettent alors de conclure sur la préservation de la sémantique ainsi que de la compatibilité. Dans le second cas, le code de réinitialisation du `switch` est d'abord exécuté, ce qui permet de conclure en utilisant le même raisonnement que précédemment où la mémoire $\mathrm{mem}_{M_b}\left(j\right)$ est à remplacer par la mémoire initiale de j.

□

11.1.2.3 Traduction avec contrôle

Il faut maintenant rajouter du contrôle en traduisant les horloges des équations, afin d'unifier les deux propriétés du lemme précédent en une seule.

Pour cela on définit la relation d'ajout de contrôle à une instruction *stmt* selon l'horloge `when` $k\left(x\right)$ pour donner l'instruction $stmt'$ et notée $\mathrm{Ctrl}\left((stmt, x, k)\right) = stmt'$ par les règles Base et NotBase de la figure 11.5. Le but est d'enfoncer l'instruction *stmt* au fond d'une structure de contrôle qui correspond à l'activation de l'horloge `when` $k\left(x\right)$.

La traduction proprement dite correspond alors à extraire une horloge et une instruction d'une équation donnée et à rajouter à cette instruction le contrôle donné par l'horloge (règle Eq de la figure 11.5).

On peut alors démontrer le lemme suivant :

Lemme 11.1.2 (Contrôle)**.** *Soient* $(E_{\mathrm{LSNI}}, M_b, M_a)$, (E, M), V *et* (I, R), x, k, *stmt et*

stmt′ tels que $(E_{\text{LSNI}}, M_b, M_a) \overset{(I,R)}{\underset{V}{\approx}} (E, M)$, $\text{Ctrl}((stmt, k, x)) = stmt'$ *et* $x \in V$. *On a alors les deux propriétés suivantes qui sont vérifiées :*

absence

$$\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) \neq k \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E, M$$

présence

$$\begin{array}{l}\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) = k \Rightarrow \\ \quad \forall E', M'. E, M \vdash_{\text{OBC}} stmt \Downarrow E', M' \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E', M'\end{array}$$

preuve du lemme 11.1.2. Énoncé formel du lemme :

$$\begin{array}{l}\forall E_{\text{LSNI}}, M_b, M_a, E, M, V, I, R.\ (E_{\text{LSNI}}, M_b, M_a) \overset{(I,R)}{\underset{V}{\approx}} (E, M) \Rightarrow \\ \quad \forall x, k, stmt, stmt'.\ \text{Ctrl}((stmt, k, x)) = stmt' \ \wedge\ x \in V \Rightarrow \\ \qquad \bigwedge \left\{ \begin{array}{l} \text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) \neq k \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E, M \\ \text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) = k \Rightarrow \\ \quad \forall E', M'.\ E, M \vdash_{\text{OBC}} stmt \Downarrow E', M' \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E', M' \end{array} \right.\end{array}$$

La preuve se fait par induction sur l'hypothèse $\text{Ctrl}((stmt, k, x)) = stmt'$.

Dans le cas de base, $stmt = stmt'$, $x = \texttt{base}$ et $k = \texttt{Only}$. Comme alors, $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(\texttt{base})$ ne peut valoir que `Only`, la seule chose à vérifier est que $\forall E', M'.\ E, M \vdash_{\text{OBC}} stmt \Downarrow E', M' \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E', M'$, ce qui est trivial.

Dans le second cas, soient $\text{ck}(x) = \texttt{when}\ k_{ck}(x_{ck})$. Par compatibilité, $x_{ck} \in V$, et par induction, on en déduit que (H_1) :

$$\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x_{ck}) \neq k_{ck} \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E, M$$

et que (H_2) :

$$\begin{array}{l}\forall E', M'.\ \text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x_{ck}) = k_{ck} \Rightarrow \\ \quad E, M \vdash_{\text{OBC}} \texttt{switch}\ x\texttt{\{}k\ \texttt{->}\ stmt\texttt{\}} \Downarrow E', M' \Rightarrow E, M \vdash_{\text{OBC}} stmt' \Downarrow E', M'\end{array}$$

Il ne reste alors plus qu'à montrer chacune des deux branches.

- Supposons que $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) \neq k$.
  - soit $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x_{ck}) \neq k_{ck}$ et H_1 permet de conclure.
  - soit $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x_{ck}) = k_{ck}$, et alors en utilisant H_2, il suffit de prouver que $E, M \vdash_{\text{OBC}}$ $\texttt{switch}\ x\texttt{\{}k\ \texttt{->}\ stmt\texttt{\}} \Downarrow E, M$, ce qui est donné par la définition de la sémantique.
- Supposons que $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x) = k$. Par la structure de nos environnements, on en déduit que $\text{sem}^{\mathcal{I}}_{E_{\text{LSNI}}}(x_{ck}) = k_{ck}$. Soient maintenant E' et M' tels que $E, M \vdash_{\text{OBC}}$ $stmt' \Downarrow E', M'$. Par définition de la sémantique, on en déduit que $E, M \vdash_{\text{OBC}}$ $\texttt{switch}\ x\texttt{\{}k\ \texttt{->}\ stmt\texttt{\}} \Downarrow E', M'$, et donc par H_2, que $E, M \vdash_{\text{OBC}} stmt' \Downarrow E', M'$.

□

Corollaire 11.1.1 (Préservation de la sémantique d'une équation)**.** *Étant donnés un environnement* $(E_{\text{LSNI}}, M_b, M_a)$ *et une paire* (E, M) *compatible sous* V *et* IR*, si une équation*

$$\textsc{SysO}\ \frac{}{\mathrm{Trad}^+\,(V, IR, \varepsilon_s) = (\varepsilon_m, V, IR, \varepsilon_s)}$$

$$\textsc{SysS}\ \frac{\begin{array}{c}\mathrm{Trad}^+\,(V, IR, eq) = (m_1, V', IR', stmt) \\ \mathrm{Trad}^+\,(V', IR', sys) = (m_2, V'', IR'', stmts)\end{array}}{\mathrm{Trad}^+\,(V, IR, eq\,;\ sys) = (m_1\,;\ m_2, V'', IR'', stmt\,;\ stmts)}$$

Figure 11.6: traduction d'un système

eq se traduit avec contrôle en une instruction stmt en produisant V' et IR', et que l'équation eq est valide, alors stmt produit un environnement E' et une mémoire M' tels que $(E_{\text{LSNI}}, M_b, M_a)$ et (E', M') soient compatibles sous V' et IR'.

Formellement :

$$\forall E_{\text{LSNI}}, M_b, M_a, E, M, V, V', IR, IR', eq, stmt, m.$$
$$\bigwedge \begin{cases} (E_{\text{LSNI}}, M_b, M_a) \overset{IR}{\underset{V}{\approx}} (E, M) \\ \mathrm{Trad}^+\,(V, IR, eq) = (m, V', IR', stmt) \\ \mathcal{S}^{\mathcal{I}}, M_b, M_a \vdash_{\text{LSNI}} eq \end{cases}$$
$$\Rightarrow m \vdash_{init} eq\ \wedge\ \exists E', M'.\ E, M \vdash_{\text{OBC}} stmt \Downarrow E', M'\ \wedge\ (E_{\text{LSNI}}, M_b, M_a) \overset{IR'}{\underset{V'}{\approx}} (E', M')$$

11.1.3 Traduction d'un nœud

La traduction d'un système d'équations se fait en chaînant la traduction de chacune de ses équations, comme indiqué en figure 11.6.

Définition 11.1.2 (Relation de traduction d'un nœud vers une classe)**.** *On définit la relation $n_s \overset{Lsni}{\underset{Obc}{\Rightarrow}} n_t$ entre un code d'origine n_s et un code produit n_t par :*

$$n_s \overset{Lsni}{\underset{Obc}{\Rightarrow}} n_t = \exists sys, stmts, locs, i, o, insts, f, V', m.$$
$$\bigwedge \begin{cases} \mathrm{Trad}^+\,(\mathrm{birth}\,(i) \cup \mathrm{birth}\,(\texttt{base}), (\emptyset, \emptyset), sys) = \Big(m, V', (regs_{\mathcal{E}^+}, inst_{\mathcal{E}^+}), stmts\Big) \\ o \notin regs_{\mathcal{E}^+} \\ n_s = \left[\begin{array}{l} \texttt{node } f\texttt{(}i\texttt{) returns (}o\texttt{)} \\ \texttt{var } locs\texttt{; ins } insts\texttt{;} \\ \texttt{let } sys \texttt{ tel;} \end{array}\right. \\ n_t = \left[\begin{array}{l} \texttt{class } f \texttt{ \{} \\ \quad \texttt{memory \{}m\texttt{\}} \\ \quad \texttt{step (}i\texttt{) returns (}o\texttt{) \{}locs/regs_{\mathcal{E}^+}\texttt{;}stmts\texttt{\}} \\ \texttt{\}} \end{array}\right. \end{cases}$$

Théorème 11.1.2 (Préservation de la sémantique d'un nœud)**.** *La validation de la relation $n_s \overset{Lsni}{\underset{Obc}{\Rightarrow}} n_t$ implique la préservation de la sémantique après traduction vers* OBC *:*

$$\forall\ n_s, n_t.\ n_s \overset{Lsni}{\underset{Obc}{\Rightarrow}} n_t \Rightarrow n_s \lesssim n_t$$

Démonstration. La démonstration se fait en deux temps.

Dans un premier temps on doit démontrer que la mémoire m (dans OBC) construite est la mémoire initiale M_0 du nœud (LSNI) ; c'est trivial, puisque par notre lemme, $m \vdash_{init} sys$ où sys est le système du nœud traduit.

Dans un second temps, il faut montrer que sous l'hypothèse $M_b, M_a, sys \vdash_{\text{LSNI}}$, on a $abs_{\mathcal{E}^+}[i \mapsto \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(i)], M_b \vdash_{\text{OBC}} stmts \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M_a$ où $\mathcal{S}^{\mathcal{I}\prime}$ est un environnement qui coïncide avec $\mathcal{S}^{\mathcal{I}}$ sur la variable de sortie. Là encore, c'est trivial par notre lemme, puisque on se retrouve avec un environnement M tel que $\left(\mathcal{S}^{\mathcal{I}}, M_b, M_a\right) \overset{(regs_{\mathcal{E}^+}, inst_{\mathcal{E}^+})}{\underset{V'}{\approx}} \left(\mathcal{S}^{\mathcal{I}\prime}, M\right)$ et $abs_{\mathcal{E}^+}[i \mapsto \text{sem}^{\mathcal{I}}_{\mathcal{S}^{\mathcal{I}}}(i)], M_b \vdash_{\text{OBC}} stmts \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M$. En remarquant que V' contient toutes les variables locales sur l'horloge de base, et donc la sortie, par définition de la compatibilité, on en déduit que $M = M_a$ et que $\mathcal{S}^{\mathcal{I}}$ et $\mathcal{S}^{\mathcal{I}\prime}$ coïncident sur la variable de sortie. On en déduit alors que sur les mêmes entrées avec les mêmes mémoires, on a production des mêmes sorties et mêmes mémoires. □

Remarque 11.1.1 (Retour sur les propriétés P_5 et P_6 de la section 7.2.2)**.** *C'est lors de cette passe que de mauvais choix d'introduction de noms de flots peuvent entraîner le rejet du programme par le compilateur. En effet, après traduction la sortie doit être une variable locale, d'où la nécessité de respecter le point P_5, et si il y a une dépendance cyclique dans les décalages de flots, par exemple dans le cas où le système contient les équations $(E_1) x_1$ `= 0 fby` x_2 et $(E_2) x_2$ `= 1 fby` x_1, il est impossible d'ordonnancer le système pour que la traduction puisse se faire. En effet, par la règle* MEM *de la traduction sans contrôle, la traduction de (E_1) requiert que x_2 soit vivante à ce moment là, et tue par la même occasion x_1. Réciproquement la traduction de (E_2) requiert que x_1 soit vivante et tue x_2. On ne peut donc pas traduire (E_1) avant (E_2) sans ramener x_1 à la vie, ni traduire (E_2) avant (E_1) sans ramener x_2 à la vie. Or, on peut démontrer par induction sur les règles de traduction sans contrôle qu'on ne peut ramener à la vie aucune variable locale ni aucune variable mémoire. Une telle preuve est sans importance en ce qui concerne cette thèse car cet énoncé est indépendant de celui de la préservation de la sémantique. En réalité, ce problème est bien connu en programmation, c'est celui de l'échange de contenu de deux variables, il est nécessaire d'introduire une variable auxiliaire.*

Exemple 11.1.1.

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    int32_1_0 : int32;
    --
    time = mem.int32_1_1;
    int32_1_0 = 1+time;
    mem.int32_1_1 = int32_1_0;
  }
}


class get_mode {
  memory {
```

```
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    keep : mode;
    switch_ : mode;
    mode_4_0 : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { keep = current_mode; }
    }
    switch(real_flip) {
      True -> { switch_ = flip(current_mode); }
    }
    switch(real_flip) {
      False -> { mode_4_0 = keep; }
      True -> { mode_4_0 = switch_; }
    }
    mem.mode_4_1 = mode_4_0;
    mem.button_1_0 = flip_button;
  }
}


class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
    bool_3_0 : bool;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    bool_3_0 = (mins==59) && (secs==59);
    mem.reset := bool_3_0;
  }
}
```

La traduction brute de `chrono_base` *et de* `ms` *correspond bien à la manière dont on aurait naturellement écrit le code dans un langage impératif, en dehors du fait que l'affectation mémoire passe par une variable auxiliaire. Pour* `get_mode`*, la fusion de tous les* `switch` *reste à faire pour avoir moins de tests.*

```
class chess_clock {
```

```
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool;
    int32_2_1 : int32;
    int32_2_2 : int32;
    unit_2_3 : unit;
    bool_2_4 : bool;
    int32_2_5 : int32;
    int32_2_6 : int32;
    unit_2_7 : unit;
    --
    bool_1_0 = True;
    switch(bool_1_0) { False -> { get_mode.reset(get_mode_8); } }
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) { Right -> { unit_2_7 = Only; } }
    switch(current_mode) { Right -> { bool_2_4 = True; } }
    switch(current_mode) {
      Right -> { switch(bool_2_4) { False -> { ms_2.reset; } } }
    }
    switch(current_mode) {
      Right -> { (int32_2_5, int32_2_6) = ms_2.step(unit_2_7); }
    }
    switch(current_mode) { Left -> { unit_2_3 = Only; } }
    switch(current_mode) { Left -> { bool_2_0 = True; } }
    switch(current_mode) {
      Left -> { switch(bool_2_0) { False -> { ms_5.reset; } } }
    }
    switch(current_mode) {
      Left -> { (int32_2_1, int32_2_2) = ms_5.step(unit_2_3); }
    }
    switch(current_mode) {
      Left -> { m = int32_2_1; }
      Right -> { m = int32_2_5; }
    }
    switch(current_mode) {
      Left -> { s = int32_2_2; }
      Right -> { s = int32_2_6; }
    }
    cc = make_chess_clock (current_mode, m, s);
  }
}
```

La traduction de `chess_clock`*, en revanche a bien besoin d'optimisations pour regrouper les tests. De telles optimisations sont commentées en annexe.*

11.2 Optimisations

De très nombreuses optimisations sont possibles à ce point de la compilation. En fait, à ce niveau il n'y a plus rien en rapport avec les flots de donnés. Les optimisations relatives à la nature « flot de données » des nœuds auraient du être faites dans les passes précédentes. Il ne reste donc plus qu'à étudier les optimisations classiques que l'on peut prouver. Voici donc une liste non exhaustive d'optimisations qu'il semble raisonnable d'utiliser, ainsi que de possibles certificats de vérification. Aucune de ces preuves n'a été réalisée, cependant la plupart semblent plus faciles à réaliser que les preuves de cette thèse. En effet, les optimisations ici sont des passes d'endocompilation, la sémantique du langage source et celle de la cible sont donc les mêmes. Les relations à vérifier ne sont plus ici données de manière formelle, mais de manière informelle.

11.2.1 Fusion des blocs de contrôle

L'optimisation la plus évidente consiste à regrouper les blocs de traitements par cas. Par exemple dans le code suivant extrait de la fonction `get_mode` :

```
switch(real_flip) {
  False -> { keep = current_mode; }
}
switch(real_flip) {
  True -> { switch_ = flip(current_mode); }
}
switch(real_flip) {
  False -> { mode_4_0 = keep; }
  True -> { mode_4_0 = switch_; }
}
switch(real_flip) {
  False -> { mem.mode_4_1 = mode_4_0; }
  True -> { mem.mode_4_1 = mode_4_0; }
}
```

Des instructions sont effectuées sous les mêmes conditions, et il serait naturel de vouloir les regrouper en :

```
switch(real_flip) {
  False -> { keep = current_mode;
             mode_4_0 = keep;
             mem.mode_4_1 = mode_4_0;
           }
  True -> { switch_ = flip(current_mode);
            mode_4_0 = switch_;
            mem.mode_4_1 = mode_4_0;
          }
}
```

La relation que doit vérifier le code optimisé par rapport au code non optimisé est qu'une suite de deux instruction `switch (`e`)` dont la première ne contient d'affectation à aucune variable libre de e se trouve remplacée par une seule instruction `switch (`e`)` dans laquelle chaque branche est construite comme la branche correspondante dans la première instruction concaténée avec la branche correspondante dans la seconde instruction.

La condition de ne modifier aucune variable libre de e peut être enlevée si on arrive à maintenir les preuves de bon ordonnancement. Cependant comme il peut être coûteux de maintenir ces preuves à chaque optimisation, et que la condition elle même n'est pas très coûteuse à vérifier, il est plus simple, même si le temps de compilation s'en trouve rallongé, de ne pas maintenir cette preuve et de vérifier la condition systématiquement.

Comme *a priori* on veut fusionner au maximum les blocs de contrôles, il est inutile d'utiliser un oracle pour nous dire lesquels fusionner, et on peut décider de les fusionner tous.

Esquisse de preuve de la fusion de blocs. Quelque soit la valeur sémantique de e, le bloc associé ne modifiera pas les variables libres de e, donc après exécution du premier `switch`, l'évaluation de e retournera la même valeur et exécutera le bloc associé dans le second `switch`. C'est donc comme si on avait fusionné les deux blocs. □

11.2.2 Propagation des variables

De par les contraintes imposées au langage Lsn, de nombreuses variables ont pu être introduites pour simplifier la compilation du nœud. Maintenant que le nœud est compilé, on peut vouloir éliminer les variables introduites, qui soit ne correspondent à aucun calcul, soit correspondent à un calcul utilisé au plus une fois.

En soit c'est une « antioptimisation » dans le sens où une expression simple (une variable) est remplacée par une définition plus complexe (la définition de la variable) ; cependant, la variable propagée sera éliminée dans une phase ultérieure, amenant au final à un code pouvant être un peu plus efficace.

Par exemple :

```
switch(real_flip) {
  False -> { keep = current_mode;
             mode_4_0 = keep;
             mem.mode_4_1 = mode_4_0;
           }
  True -> { switch_ = flip(current_mode);
            mode_4_0 = switch_;
            mem.mode_4_1 = mode_4_0;
          }
}
```

Ce code pourrait être remplacé par :

```
switch(real_flip) {
  False -> { keep = current_mode;
             mode_4_0 = current_mode;
             mem.mode_4_1 = current_mode;
```

```
            }
    True -> { switch_ = flip(current_mode);
              mode_4_0 = switch_;
              mem.mode_4_1 = switch_;
            }
}
```

en vue d'une future élimination des instructions `mode_4_0 = switch_` et `mode_4_0 = keep` devenues inutiles.

La relation attendue entre le code antioptimisé et l'original ressemble beaucoup à la relation de substitution. Étant donnée une variable v à laquelle on affecte l'expression e, on peut remplacer toutes les occurrences suivantes de v par e dès lors que les variables libres de e ne sont pas affectées entre la définition de v et sa dernière occurrence. Tout comme pour la fusion de flots, si on maintient une preuve de bon ordonnancement, on peut se passer de cette condition, mais *a priori* ça n'en vaut pas la peine.

Esquisse de preuve de la propagation des variables. La preuve se fait par induction sur la sémantique du programme.

Toute instruction précédant ou incluant l'affectation de v dans le code antioptimisé est identique à celle du code original.

Juste après l'affectation on sait donc que l'évaluation de e et celle de v coïncident.

Dans toutes les instructions suivantes jusqu'à la dernière occurrence de v, les variables libres de e ni v ne sont affectées, et par conséquent les évaluations de v et e coïncident toujours, ce qui fait que chaque instruction antioptimisée produira le même environnement que sa version originale.

Après la dernière occurrence de v, le code est identique, et donc produira encore le même environnement. □

11.2.3 Réduction du contrôle

Parfois, il existe dans un nœud des applications que l'on ne veut pas réinitialiser, cela se traduit par un `f(x) every False(True)` qui produit lors de la compilation une instruction `switch(True) { False -> { f.reset; } }`. De même, certaines passes d'optimisation telles que celle précédemment décrite, peuvent produire des instructions de la forme `switch(A){ ... A -> {...} ...}`. De telles expressions peuvent être ι-réduites afin d'éliminer le branchement conditionnel inutile puisque le code associé sera nécessairement exécuté.

Ce genre d'optimisation n'a pas besoin d'outil externe pour se faire, puisque partout où de telles formes apparaissent, une réduction est souhaitable.

Esquisse de preuve de la réduction du contrôle. La preuve de préservation de la sémantique est une application directe de la sémantique. Une constante s'évalue toujours en la même valeur, quel que soit l'environnement dans lequel elle est évaluée. Le bloc associé sera donc nécessairement exécuté et produira donc les mêmes environnements que le code original. □

11.2.4 Nettoyage des variables locales inutilisées

Aucune des optimisations précédemment mentionnées n'efface les variables inutiles. C'est cependant une optimisation importante puisqu'elle peut diminuer de beaucoup la taille du code, permettre d'autres optimisations et alléger la tâche du compilateur puisqu'il a moins de variables à gérer.

Le prédicat sous-jacent est que la variable à éliminer n'apparaît nulle part à droite d'une affectation.

Un outil externe qui indique quelle variable éliminer est le bienvenu ici.

Esquisse de preuve du nettoyage de variables locales inutilisées. La preuve est une conséquence directe du lemme suivant :

$$\begin{array}{l}\forall \mathcal{S}^{\mathcal{I}}, M_b, stmt, \mathcal{S}^{\mathcal{I}\prime}, M_a, v.\\ \quad v \notin stmt \wedge \mathcal{S}^{\mathcal{I}}, M_b \vdash_{\text{OBC}} stmt \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M_a \Rightarrow\\ \quad\quad \mathcal{S}^{\mathcal{I}} - \{v\}, M_b \vdash_{\text{OBC}} stmt \Downarrow \mathcal{S}^{\mathcal{I}\prime} - \{v\}, M_a\end{array}$$

Ce lemme se démontre par induction sur la sémantique de *stmt*. □

11.2.5 Déplacement d'instructions

Comme présenté dans la partie sur l'ordonnancement, on peut réajuster l'ordre des instructions tant que l'on ne casse pas les dépendances de données. Le système étant déjà ordonnancé, une permutation de deux instructions qui n'interfèrent pas laissera le système ordonnancé.

Soient i_1 et i_2 deux instructions. On note r_1 (resp. r_2) les variables lues par i_1 (resp. i_2) et w_1 (resp. w_2) , celles écrites par i_1 (resp. i_2).

On dit que i_1 et i_2 commutent et on note $i_1 \bot i_2$si la proposition suivante est vérifiée : $r_1 \cap w_2 = \emptyset = r_2 \cap w_1$.

Si deux instructions consécutives commutent, alors on peut inverser leur ordre d'apparition.

On a même la propriété suivante : « Étant donnés deux ordonnancements d'un même système, il existe une suite de transpositions d'instructions consécutives qui commutent permettant de passer du premier ordonnancement au second. ».

Cet énoncé n'est pas fondamental et sa preuve, bien que facile n'est pas donnée. En revanche la conséquence en est que même si on a choisi le plus mauvais ordonnancement possible au moment du passage vers OBC, il y a encore moyen de rectifier le tir et de se retrouver avec un ordonnancement optimal par une suite de réagencements locaux sans avoir à tout réordonnancer.

Notons que l'opération de transposition d'instructions consécutives qui commutent n'est pas une optimisation en soi, mais qu'elle peut apporter de nouvelles possibilités d'optimisations en rapprochant des équations sur les mêmes horloges.

Esquisse de preuve du déplacement d'instructions. La preuve repose sur le lemme suivant :

$$\begin{aligned}&\forall \mathcal{S}^{\mathcal{I}}, M_b, i_1, \mathcal{S}^{\mathcal{I}\prime}, M_1, M_a, i_2, \mathcal{S}^{\mathcal{I}\prime\prime}.\\ &\quad i_1 \bot i_2 \wedge \left(\mathcal{S}^{\mathcal{I}}, M_b \vdash_{\text{OBC}} i_1 \Downarrow \mathcal{S}^{\mathcal{I}\prime}, M_1\right) \wedge \left(\mathcal{S}^{\mathcal{I}\prime}, M_1 \vdash_{\text{OBC}} i_2 \Downarrow \mathcal{S}^{\mathcal{I}\prime\prime}, M_a\right) \Rightarrow\\ &\qquad \exists \mathcal{S}^{\mathcal{I}\prime\prime\prime}, M_2. \left(\mathcal{S}^{\mathcal{I}}, M_b \vdash_{\text{OBC}} i_2 \Downarrow \mathcal{S}^{\mathcal{I}\prime\prime\prime}, M_2\right) \wedge \left(\mathcal{S}^{\mathcal{I}\prime\prime\prime}, M_2 \vdash_{\text{OBC}} i_1 \Downarrow \mathcal{S}^{\mathcal{I}\prime\prime}, M_a\right)\end{aligned}$$

Ce lemme se démontre par induction sur la sémantique de i_1 et i_2.

Les instructions précédentes étant identiques, elles produisent le même environnement, le lemme précédent nous assure alors que après exécution des deux instructions, les environnements sont à nouveau égaux, et les instructions suivantes étant égales, l'exécution de la totalité du code produit les mêmes environnements. □

11.2.6 Augmentation du contrôle

L'augmentation du contrôle consiste en le remplacement d'une instruction i par une instruction `switch(`e`){`k_1`->`i ... k_n`->`i`}` où e est une expression dont le type est un type énuméré quelconque, en particulier, on ne demande pas que ses variables libres aient été précédemment définies, bien qu'en pratique ce sera toujours le cas.

C'est l'inverse d'une optimisation (diminution du contrôle) qui n'a pas été évoquée car en pratique elle ne se rencontre que trop rarement. L'intérêt de cette anti-optimisation est simplement de pouvoir fusionner des blocs sur une même horloge mais séparés par une ou plusieurs instructions. En augmentant le contrôle, on peut placer ces instructions sur la même horloge et ainsi fusionner les blocs.

Pour un exemple d'augmentation du contrôle, on peut considérer

```
switch(real_flip) {
  False -> { keep = current_mode; }
}
switch(real_flip) {
  True -> { switch_ = flip(current_mode); }
}
switch(real_flip) {
  False -> { mode_4_0 = keep; }
  True -> { mode_4_0 = switch_; }
}
mem.mode_4_1 = mode_4_0;
```

qui a été transformé en

```
switch(real_flip) {
  False -> { keep = current_mode; }
}
switch(real_flip) {
  True -> { switch_ = flip(current_mode); }
}
switch(real_flip) {
  False -> { mode_4_0 = keep; }
  True -> { mode_4_0 = switch_; }
}
```

```
    switch(real_flip) {
      False -> { mem.mode_4_1 = mode_4_0; }
      True -> { mem.mode_4_1 = mode_4_0; }
    }
```

afin de pouvoir propager la constante `mode_4_0` et obtenir après nettoyage

```
      False -> { mem.mode_4_1 = current_mode;
                 }
      True -> { switch_ = flip(current_mode);
                mem.mode_4_1 = switch_;
              }
    }
    mem.button_1_0 = flip_button;
  }
```

Sans augmentation du contrôle, il n'était pas possible d'éliminer la variable `mode_4_0`.

L'augmentation du contrôle a besoin d'un outil externe pour indiquer quelles instructions doivent être concernées ainsi que l'expression sur laquelle on augmente le contrôle.

Esquisse de preuve de l'augmentation du contrôle. Il suffit de montrer que quelle que soit la valeur de l'expression sur laquelle on augmente le contrôle, après exécution l'état interne a été modifié de façon identique. Étant donné que le `switch` est exhaustif sur les valeurs de l'expression, et que chaque branche contient l'instruction dont le contrôle est augmenté, c'est le cas. □

11.2.7 Enchaînement des optimisations

Toutes ces optimisations peuvent en entraîner d'autres auparavant impossibles, il peut donc être nécessaire de faire plusieurs cycles de passes d'optimisation. C'est encore une fois à un outil externe qu'il revient de décider quand on doit arrêter les cycles d'optimisation. Un mauvais outil externe pourra produire du code moins efficace s'il est mal conçu. En effet, cette forme de compilation ne garantit en aucun cas que le code produit sera meilleur, mais seulement que la sémantique sera préservée.

11.3 Conclusion

La passe de génération de code est la dernière avant d'éventuelles optimisations source à source. Une fois ce point atteint, on peut dire que le code a été compilé, même s'il est assez inefficace de par le nombre de tests de comparaison effectués.

Quatrième partie

Conclusion

Chapitre 12

Vers du code exécutable

Ce chapitre ne contient plus d'énoncé formel ni de preuve. Le dernier langage de notre chaîne de compilation certifiée est Obc, et ce qui suit est simplement informel.

12.1 Génération de code C

Dans les applications temps réel critique, le langage C est très souvent utilisé. De nombreuses raisons justifient ce choix : l'absence de machine virtuelle sous-jacente qui tend à rendre le système imprévisible vis-à-vis des pires temps d'exécution ou de la consommation mémoire, le contrôle absolu de la machine qui permet d'effectuer certaines tâches à la vitesse maximale et avoir des pires temps d'exécution acceptables, ou encore l'existence d'un compilateur agréé pour le développement.

Obc ressemble beaucoup à un fragment de C, dans lequel on aurait retiré les allocations de mémoire dynamique ainsi que les structures de boucles.

Les objets définis dans Obc ne sont qu'un type de données (la mémoire), ainsi qu'une fonction d'initialisation de ce type et une fonction d'itération qui doit prendre explicitement en arguments un pointeur vers la mémoire allouée de l'objet ainsi que tous les paramètres d'entrée.

Typiquement, si on considère le code Obc suivant :

```
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    mem.reset = (mins==59) && (secs==59);
  }
}
```

```
class chess_clock {
```

le fichier d'interface C contiendra :

```
// MS
typedef struct {
  int reset;
  chrono_base_t chrono_base_2;
} ms_t;
void ms_reset (ms_t*);
```

alors que l'implémentation contiendra :

```
// MS
void ms_reset (ms_t* this) {
  this->reset = 0;
  chrono_base_reset(&(this->chrono_base_2));
}


// MS
void ms_step (unit tic, ms_t* this, int32_t* mins, int32_t* secs) {
  int32_t int32_1_0;
  if(this->reset) chrono_base_reset(&(this->chrono_base_2));
  chrono_base_step(Only, &(this->chrono_base_2), &int32_1_0);
  *mins = int32_1_0 / 60;
  *secs = int32_1_0 % 60;
  this->reset = ((*mins==59)&&(*secs==59));
```

Afin d'éviter les copies, tous les arguments se font par référence ; les arguments de type `unit` sont retirés ; et les fonctions manipulent leur mémoire au travers du pointeur `this`.

12.2 Connexion avec CompCert

CompCert [36] est un projet de compilateur d'un sous-ensemble large de C vers l'assembleur PowerPC [1], x86 et ARM. Le langage [2] C et l'assembleur PowerPC utilisés y ont été formellement définis, et une preuve de préservation de la sémantique a été réalisée en Coq.

On a donc deux langages formalisés en Coq : Obc et C. La problématique qui s'impose est alors de savoir comment mettre en relation leurs sémantiques pour pouvoir parler de préservation de sémantique de Ls vers l'assembleur PowerPC.

12.2.1 Compatibilité du code généré avec CompCert

CompCert se définit comme un sous-ensemble de C assez large pour que l'on puisse y traduire sans difficulté Obc. En effet, les fonctionnalités de C qui ne sont pas gérées par

1. Le PowerPC est une architecture fréquemment utilisée pour le temps réel, en témoigne son adoption par Airbus, notamment pour l'A380 [57]

2. Ici, et dans la suite, « langage » ne désigne pas simplement l'ensemble qu'il reconnaît, mais aussi la sémantique qui lui est associée.

CompCert se résument à [40] :
- l'utilisation `switch` non structuré (Duff's device) ;
- la déclaration de fonctions sans leur prototype ;
- la déclaration de fonctions d'arité arbitraire (fonctions dites variadiques) ;
- l'utilisation des fonctions `longjmp` et `setjmp` ;
- l'utilisation des types `long double` et `long long` ;
- l'utilisation de fonction retournant un type `struct` ou `union` ;
- l'utilisation de tableaux à taille variable ;
- l'utilisation de champs de bit dans les `struct` et les `union`

Or aucune de ces fonctionnalités ne nous intéresse dans la génération de code C.

En revanche, il y a un fait assez gênant dans la sémantique de CompCert, à savoir que tout programme doit avoir une fonction principale. Or notre chaîne de compilation ne produit pas de tel programme. En d'autres termes, notre chaîne ne produit pas vraiment un programme, mais plutôt une bibliothèque.

Il serait intéressant de pouvoir compléter la sémantique de CompCert sur les programmes avec une sémantique de CompCert sur les bibliothèques de programmes.

CompCert a deux sortes de sémantiques, les sémantiques « convergentes » pour les programmes qui terminent et les sémantiques « divergentes » pour ceux qui ne terminent pas. CompCert s'intéresse à la préservation des événements et l'ordre dans lequel ils arrivent ; un événement étant :
- soit un appel système (le nom de la fonction appelée avec les valeurs des arguments, et la valeur retournée)
- soit une lecture d'une variable volatile (nom de la variable associée, type de valeur, adresse du bloc mémoire et valeur du contenu)
- soit une écriture dans une variable volatile (nom de la variable associée, type de valeur, adresse du bloc mémoire et valeur écrite)
- soit une annotation (identifiant de l'annotation et liste de valeurs associées)

Les valeurs gérées dans ces événements sont les valeurs entières, flottantes et les pointeurs (adresse et nom du pointeur) ; quant aux types de valeurs, il s'agit de savoir si elle est signée ou non, et si elle occupe 8, 16, 32 ou 64 bits.

Dans le cas de sémantiques « convergentes », CompCert s'intéresse à la préservation des listes d'événements produits, dans le cas de sémantiques « divergentes », CompCert s'intéresse à la préservation des suites infinies d'événements produits.

Pour pouvoir parler de préservation de sémantique entre Obc et CompCert, il faudrait donc mettre en relation la sémantique instantanée des nœuds et la trace d'exécution d'un nœud.

12.2.2 Comment réinterpréter la sémantique d'Obc dans CompCert

Le fait de ne pas produire de programme exécutable dans Obc est un frein à la mise en relation des deux sémantiques. Il y a plusieurs façons de contourner cette difficulté.

Pour s'intégrer à CompCert, il faudrait définir un fragment du C de CompCert pour lequel toutes les fonctions termineraient, n'auraient jamais accès à des variables volatiles, ne feraient aucun appel système et ne produiraient aucune annotation et utiliser

ce fragment pour définir les opérateurs utilisés dans notre chaîne de compilation.

En effet, si un opérateur externe est susceptible de créer une trace d'exécution, les différentes possibilités d'optimisations, amèneraient à plusieurs compilations possibles qui n'auraient pas la même sémantique (côté CompCert) au final.

Sans exploiter les traces

Cependant, une inspection du code de CompCert donne un prédicat qui décrit la sémantique des fonctions qui terminent (sémantique « convergente ») :

```
eval_funcall : mem->fundef->(list val)->trace->mem->val->Prop.
```

`eval_funcall GMb f args tr GMa val` signifie que l'appel à la fonction f avec les arguments $args$ retourne la valeur val et modifie la mémoire globale GMb en GMa ; t est la trace produite par cette fonction.

Il se trouve justement que le code Obc est du code qui termine (sous hypothèse que toutes les opérateurs utilisés terminent). On pourrait donc envisager de définir la préservation de la sémantique non pas sur des programmes, mais seulement sur les fonctions. Cette préservation s'exprimerait alors sous la forme :

$$\forall\, M_b, M_a, args, val, n.\, N^{\mathcal{I}}_{iter}\,(n)\,((M_b, args), (M_a, val)) \Rightarrow$$
$$\forall\, GM_b, ref.\, \exists GM_a.GM_b[ref] = M_b \wedge GM_a[ref] = M_a \wedge (\forall x \neq ref.\, GM_a[x] = GM_b[x]) \wedge$$
$$\texttt{eval\_funcall}(G_b, trad(n), ref + args, \epsilon, G_a, val)$$

où $GM[x]$ correspond à la zone mémoire pointée par x et ϵ est la trace vide. Il faudrait aussi un prédicat similaire pour la fonction d'initialisation.

Cette relation donnerait en plus comme information que seule une zone mémoire très localisée (la mémoire de l'instance) est affectée par l'exécution de l'itération (en supposant qu'aucun opérateur externe ne modifie la mémoire globale).

En exploitant les traces

Une autre façon de mettre en relation les sémantiques serait de créer la boucle infinie d'itération du nœud principal, et de lire les entrées à chaque itération en mémoire volatile et d'écrire les sorties aussi en mémoire volatile.

La trace d'un tel programme correspondrait alors à la suite des entrées et la suite des sorties, ce qui correspondrait exactement à la sémantique de Ls ; l'état de la mémoire interne étant alors sans réelle importance, seul le résultat compterait.

Chapitre 13

Extensions du langage à considérer

Comme mentionné en introduction, Ls est une évolution d'un autre langage qui sert de noyau au langage Heptagon [23] [1]. Le langage Heptagon est un langage développé par Gwenaël Delaval, Léonard Gérard, Adrien Guato, Cédric Pasteur et Marc Pouzet, et dont le but est de refaire un langage de type Lustre avec de nouvelles fonctionnalités. Ce est un langage synchrone flots de données qui se distingue de Lustre entre autres par les aspects suivants qui seront abordés dans ce chapitre :

- la possibilité de définir hiérarchiquement des systèmes d'équations, des automates et des réinitialisations
- l'existence d'une analyse d'initialisation
- un système de typage semi-linéaire permettant l'écriture en place
- la présence d'itérateurs sur des tableaux

L'objectif initial de cette thèse était d'avoir un compilateur vérifié pour le noyau. On peut donc s'intéresser aux fonctionnalités de Lustre, de MiniLs et d'Heptagon qui manquent à Ls pour en faire un langage agréable à utiliser, tout en se gardant la possibilité de prouver formellement ces extensions. Aucune de ces extensions n'a été formellement prouvée, il est juste question ici de points qu'il pourrait être intéressant d'essayer de formaliser et de prouver.

13.1 Extensions pour exprimer tout programme Lustre

Revenons sur les fonctionnalités de Lustre absentes de Ls. Il y en a principalement deux. La première est celle des horloges pour les entrées/sorties. La seconde est celle de la présence d'opérateur de décalage non initialisé.

13.1.1 Multi-horloges pour les entrées/sorties

La sémantique de Ls est un peu plus simple que celle de Lustre quant à la gestion des entrées/sortie. En Lustre, les entrées comme les sorties peuvent être sur des horloges différentes.

1. Le noyau utilisé dans Heptagon s'appelle MiniLs, et contrairement à ce l'on pourrait penser, MiniLs est plus gros que Ls.

Typiquement, un nœud peut avoir pour signature :

`(a:bool, b:bool when a)→(c:bool when b, d:int when c)`

Un tel nœud doit attendre que sa première entrée soit `true` pour pouvoir lire sa seconde entrée. Si cette dernière est `true`, alors sa première sortie est produite. Si de plus cette première sortie est `true`, alors une seconde sortie est produite.

De telles signatures peuvent s'avérer utiles en LUSTRE, bien que l'on puisse s'en passer en remaniant le code, comme par exemple en ajoutant à chaque type une valeur supplémentaire pour indiquer une absence [2].

La raison pour laquelle toutes les entrées et la sortie sont sur la même horloge est le besoin d'introduire une forme de typage dépendant. Ceci complique les analyses, le typage et la sémantique. En effet, il existe essentiellement deux solutions.

- Une première solution consiste en l'utilisation de lieurs, c'est-à-dire en donnant un nom aux arguments pour les référencer dans les horloges des autres arguments. C'est le cas en LUSTRE, comme le montre la signature précédemment donnée. Le typage devient plus complexe puisqu'il faut unifier les horloges des arguments avec celle donnée par la signature lors de l'appel de nœud. Cette solution rend aussi la sémantique moins naturelle, puisque au lieu d'être une fonction des listes de suites de valeurs dans les suites de valeurs, elle devrait explicitement gérer les absences. La sémantique d'un nœud deviendrait une fonction des listes de suites de valeurs présentes ou absentes qui doivent vérifier des contraintes données par la signature. Il faudrait alors aussi justifier que l'insertion ou la suppression d'instants où toutes les entrées et sorties sont absentes ne change pas la sémantique.
- Une seconde solution consiste en l'utilisation de types algébriques. Tout type de donnée à n constructeurs pourrait être associé à un type de données algébrique à n variables de types et constructeurs. Ainsi, le type :

  ```
  type bool = True | False
  ```

  pourrait être associé au type :

  ```
  type ('a, 'b) ebool = Etrue of 'a | Efalse of 'b
  ```

  en utilisant la syntaxe OCAML. `(a:bool, b:int when a)` pourrait alors être encodé par par un flot de type `(int, unit) ebool`. Une valeur `(True, 3)` serait alors codée par `Etrue(3)` et une valeur `(False,` *abs*`)` serait codée par `Efalse(Only)`.
  Cela permettrait d'avoir des entrées échantillonnées sur d'autres entrées ou des sorties échantillonnées sur d'autres sorties, mais pas de sorties échantillonnées sur des entrées. Pour cela il faudrait exprimer que les sorties dépendent des entrées. Ce problème peut se résoudre en exploitant une idée similaire, cette fois-ci en associant à un type de données à n constructeurs, un unique constructeur de types à n variables. Ainsi, le type :

  ```
  type bool = True | False
  ```

  pourrait être associé au type :

2. On perd alors toutefois la notion d'horloge pour ces entrées/ sortie, et dans le code, il faut systématiquement tester la présence ou l'absence de valeur.

```
Definition fbool (a b : Type) :=
  forall (x : bool), match x with True => a | False -> b end
```

en syntaxe Coq (ce type n'est pas directement exprimable en Ocaml). La signature

```
(a : bool, b : bool when a)→(c:int when not a, d:int when b)
```

serait alors encodée en

```
fbool (fbool int  unit) int
```

C'est-à-dire une fonction d'argument booléen a qui, lorsque son argument a est `True`, retourne une fonction d'argument booléen b qui retourne un entier (d) quand b vaut `true` et rien (`unit`) sinon, et qui, lorsque son argument a est `false` retourne un entier (c).
De telles signatures sont élégantes, mais assez difficiles à lire. De plus il faut se donner de nouvelles primitives pour accéder aux différents champs de types aussi complexes.

13.1.2 Décalage de flots non initialisés

Dans de nombreux langages, il existe des constantes appartenant à tous les types. En terme de théorie des types ces langages peuvent être considérés comme incohérents, puisqu'on a un élément de type $\forall A.\ A$ qui est l'encodage du faux en logique minimale d'ordre supérieur.

Par exemple en Haskell, `undefined` est une telle constante, en Ocaml, `assert false` (ou plus généralement toute exception) en est également une ; en C, on peut projeter les types («cast» en terminologie anglo-saxonne) les uns sur les autres, et ainsi de suite.

Lustre n'échappe pas totalement à la règle, le premier instant présent de `pre x` n'est pas défini ; c'est pourquoi on a besoin de l'opérateur `->` et que

$$k \texttt{ fby } x$$

en Ls se traduit par

$$k \texttt{ -> pre } x$$

en Lustre.

Ajouter une constante `_nil_` à la syntaxe de nos langages permettrait de pouvoir traduire plus facilement un programme Lustre dans Ls :

$$\texttt{pre } x$$

se traduirait par

$$\texttt{\_nil\_ fby } x$$

Et

```
x = current(y)
```

se traduirait par

```
x = merge z True -> y False -> _nil_fby (x when not z)
```

quand `y` est d'horloge `when z`.

Voici quelques conséquences de l'ajout d'une telle constante :

typage Le plus raisonnable serait de demander à cette constante d'avoir le type de n'importe quel type simple ; donner pour cette constante un produit de type impliquerait de devoir indiquer quel produit de type cette constante représente à chacune de ses occurrences, de plus cela compliquerait la passe de distribution des tuples en autorisant `_nil_` à se séparer en plusieurs `_nil_` plus simples.

Les nœuds devraient aussi indiquer dans leur signature à quels instants les flots qu'ils lisent ou produisent peuvent avoir la valeur `_nil_`, afin de pouvoir déterminer statiquement si il y a accès à cette valeur (un accès à la valeur `_nil_` n'est généralement pas souhaitable car correspond à une erreur). Un tel système de typage a été proposé dans [15], mais n'est pas complet dans le sens où des programmes valides (c'est-à-dire n'accédant jamais à la valeur `_nil_`) peuvent être rejetés.

syntaxe Cette constante pourrait n'être autorisée que partout où des constructeurs sont autorisés, sauf aux endroits qui correspondent à une analyse du constructeur (à droite des `when`, dans les parties gauches des branches des `merge`, à droite des `every`).

sémantique Généralement, on ne veut pas pouvoir parler de `_nil_` dans la sémantique d'un nœud, mais on veut pouvoir l'accepter en interne. Il faudrait donc rajouter une propriété dans la définition de la sémantique d'un nœud qui demanderait aux entrées et sorties de ne pas contenir la valeur `_nil_` (comme on leur a demandé de ne pas contenir d'absence), et une propriété selon laquelle un appel de nœud ne peut se faire sur des entrées pouvant contenir `_nil_`.

Cette valeur resterait autorisée dans les environnements sémantiques locaux, il faudrait alors décider si les opérateurs importés devraient ou non être stricts (c'est-à-dire retourner `_nil_` dès lors que l'un de leurs paramètres est `_nil_`). Avoir des opérateurs stricts permet de ne pas avoir à étendre la sémantique de ces opérateurs pour leur faire gérer les cas exceptionnels, et permet de simplifier les analyses pour savoir si une expression s'évalue à `_nil_` (c'est le cas si et seulement si une de ses sous expressions s'évalue à `_nil_`), de plus la plupart des langages sont stricts, et ce serait donc la façon naturelle de définir la sémantique. Autoriser une sémantique non stricte serait aussi envisageable mais les analyses seraient plus complexes.

Dans ce cadre il serait bon d'avoir un outil capable de déterminer à l'avance si le nœud considéré a toujours un bon comportement (c'est-à-dire s'il est capable de tourner sans jamais s'arrêter ni retourner `_nil_` quelques soient ses entrées) ; c'est ce que fait l'analyse d'initialisation de Lucid Synchrone ou encore celui de ReLuC utilisé dans SCADE V6 [15]. Lustre ne dispose pas d'une telle analyse, il faut des outils externes pour s'assurer que le code n'utilisera pas de valeur non initialisée. Bien sûr un tel outil aurait besoin d'être certifié et devrait se reposer sur la sémantique ainsi définie.

compilation vers du code machine On ne veut généralement pas s'arrêter à la chaîne de compilation vers Obc, il faut donc alors pouvoir interpréter `_nil_` dans le langage final.

Il y a alors deux choix possibles. Ou bien on ajoute une valeur spéciale à chaque type qui permette de savoir si cette valeur provient d'un `_nil_`, en ajoutant une valeur spéciale à chaque type déclaré et rajouter partout des traitements spéciaux pour cette nouvelle valeur, et ce avec un coût en performances, mais on garde la possibilité de

retrouver rapidement la cause d'un problème. Ou bien on ne gère pas cette constante, et on lui affecte une valeur non initialisée (quand c'est possible, ce qui n'est pas le cas par exemple en OCAML) ou initialisée à une valeur par défaut. L'usage d'une valeur non initialisée fait perdre le déterminisme des programmes susceptibles de retourner `_nil_`.

13.2 Extensions pour exprimer tout programme MiniLs

Un des inconvénients des langages à assignation unique [18] est la surabondance des copies. Typiquement, si une fonction doit mettre à jour qui occupent une grande zone mémoire, pour éviter les problèmes d'effets de bords, cette grande zone mémoire va être dupliquée. Si au contraire le compilateur sait que cette zone mémoire ne sera plus jamais utilisée par la suite, il peut simplement la mettre à jour. Les types linéaires [26] sont une manière de donner au compilateur ce genre d'information. Cette idée a été reprise dans quelques langages comme par exemple CLEAN [60], et ont été introduits dans HEPTAGON [23]. L'introduction d'un tel typage n'est pas anodine, elle demande plus de contraintes sur le typage. Par contre, elle laisse envisager une simplification de la chaîne de compilation. En effet, dans OBC les états internes doivent être mis à jour « en place », il y a donc un effet de bord explicite au moment de l'appel à la fonction d'itération (`step`). Utiliser des types linéaires permettrait d'effacer ces effets de bords et de mentionner explicitement l'état interne. Typiquement,

```
y = f(x) every z
```

pourrait être transformé en

$$(y, st) = f'(x, ifzthens0else(s0fbyst))$$

`f'` étant la fonction (sans effet de bord) associée à `f` (qui modifie son état interne par effet de bord), et `s0` l'état interne initial de `f`. Cette transformation éliminerait le besoin de la notion d'instance, et donc l'utilité de LSNI.

Cette extension pourrait également permettre d'autoriser l'importation d'opérateurs à effets de bords. En effet, l'utilisation d'un opérateur à effet de bord est très délicate. Le code suivant,

```
x = f(y); z = f(y);
```

peut actuellement être optimisé en

```
x = f(y); z = x;
```

car `f` est sans effet de bord. En autorisant les effets de bords, ce n'est plus vrai, dans un cas `f` modifie deux fois son état interne, dans l'autre, elle ne le fait qu'une fois. Avec un typage linéaire, l'état interne de `f` peut être spécifié explicitement avec un type abstrait. Mieux encore, on peut forcer l'ordre d'exécution :

```
(x, stx) = f(y, st0); (z, st1) = f(y, stx);
```

ou

```
(x, st1) = f(y, stz); (z, stz) = f(y, st0);
```

Ce type de code n'a pas besoin de types linéaires, mais demande des précautions supplémentaires pour éviter la duplication des états internes qui ne sont plus nécessaires dans le cas des types linéaires.

Enfin, ces types linéaires ont permis l'implantation des types tableau dans MiniLs. Dans la version 4 de Lustre, on peut déclarer et utiliser des tableaux de taille fixe. Cependant, ces tableaux ne sont que des paquets de variables dans les faits, puisqu'on ne peut pas le parcourir dans une boucle et que les modifications en place ne sont pas possibles. MiniLs propose un type natif pour les tableaux avec des primitives pour les parcourir ou les modifier en place.

13.3 Extensions pour exprimer tout programme Heptagon

Le compilateur Heptagon transforme tout code Heptagon vers son noyau MiniLs pour produire du code Obc. La traduction de MiniLs vers Obc a déjà été discutée, mais il reste encore à considérer la traduction de Heptagon vers MiniLs.

13.3.1 Automates hiérarchiques

MiniLs est un langage d'assez bas niveau ; il encode exactement les machines à états finis, mais à la lecture de la définition d'un nœud, il n'est pas facile de visualiser quelles équations se rapportent à quel état.

Pour mieux visualiser ces transitions, il existe des langages graphiques tels StateCharts [28], Argos [38] ou encore SyncCharts [2].

Il existe plusieurs points de vue pour définir les automates hiérarchiques :
- une structuration du code qui permet d'éliminer la notion d'horloge, tout en regroupant les équations qui correspondent au même état
- permettre de définir des nœuds locaux « clos », c'est-à-dire sans entrées ni sortie et pouvant utiliser les noms de flots définis dans le nœud principal. Tout comme les applications de nœud, un état peut être réinitialisé, ce qui permet de réinitialiser un ensemble d'équation, fonctionnalité absente de Ls bien que simulable.

Une telle fonctionnalité implique des choix de conception.

Il existe deux grands formalismes dans les machines à états finis, à savoir :
- les machines de Mealy [41] qui réagissent pendant les transitions
- les machines de Moore [42] qui réagissent tant qu'elles restent dans le même état

Argos et SyncCharts utilisent des machines de Mealy.

L'avantage des machines de Mealy sur les machines de Moore est que la sémantique y est plus naturelle. En effet, il suffit de regarder quelle transition effectuer, et exécuter le code associé à la transition.

Pour les machines de Moore, le code est associé à l'état ; mais quand il y a une transition, deux états sont en jeux, donc deux codes. Plusieurs systèmes sont alors possibles, soit on exécute le code du premier état et on fait la transition ensuite (préemption forte), soit on fait la transition et on exécute le code du second état (préemption faible). On peut aussi essayer de mélanger les deux, on doit alors préciser dans chaque état si la préemption

est forte ou faible. Ce mélange tend à compliquer la sémantique, et la compilation. Si on veut pouvoir compiler les automates hiérarchiques vers Ls, alors on n'a pas le droit *a priori* d'exécuter dans le même instant le code de deux états différents (car ils sont sur des horloges disjointes), il faut donc mémoriser qu'au prochain passage dans l'état si oui ou non il faudra le réinitialiser. Ceci oblige à maintenir un drapeau de réinitialisation pour chaque état. Toute transition pouvant être amenée à modifier n'importe lequel de ces drapeaux, chaque transition implique une équation de la forme $(reset_{state_1}, ..., reset_{state_n}) = ...$ où $state_1$, ..., $state_n$ sont les états cibles possibles des transitions, ce qui tend à faire exploser la taille du code si tout état est accessible à partir de n'importe quel autre. Enfin certaines situations deviennent compliquées à gérer car il faut aussi tenir compte de savoir si on vient d'une transition fortement ou faiblement préemptive. Il est alors très difficile de produire un compilateur vers Ls qui soit correct et efficace. Les automates de Moore avec réinitialisation, faible et forte préemption sont donc trop complexes à gérer ; par contre si on retire l'une de ces trois fonctionnalités, les schémas de compilations redeviennent assez simples pour produire du code efficace.

13.4 Conclusion

Le langage Ls est vraiment minimaliste, et pourrait, s'il était entièrement qualifié servir de cible à de nombreux langages de programmation synchrone. Un peu comme l'assembleur, il lui manque des fonctionnalités de plus haut niveau. Lui ajouter un système de typage d'horloge au niveau des entrées sorties permettrait de le rendre plus agréable à utiliser que Lustre. En effet, ce langage ne gère pas la réinitialisation de nœuds et n'autorise que les branchements binaires contrairement au `merge` de Ls qui est un peu plus efficace. Lui ajouter des automates hiérarchiques permettrait de faciliter la compilation de plusieurs langages synchrones et de coller davantage à la représentation graphique utilisée par SCADE.

L'ajout de toutes ces extensions augmenterait également la lisibilité du code et permettrait de mieux comprendre les programmes écrits.

Chapitre 14

Conclusion

14.1 Travaux connexes

La preuve formelle de programmes informatiques est une vieille aventure. Un des plus anciens exemples remonte à A. Turing [59] en 1949. D'autres exemples sont dus à R.W. Floyd [21] en 1967, et à C.A.R. Hoare [29] en 1969.

Mais les programmes prouvés jusque là n'étaient pas vérifiés par une machine. Il a fallu attendre le développement d'outils pour exprimer les preuves formelles, tels qu' Automath en 1967 [19], de prouveurs automatiques et d'assistants de preuves pour voir l'apparition de programmes formellement prouvés et vérifiés par une machine.

Pour ce qui est de la vérification de compilateurs, l'histoire est beaucoup plus récente, un compilateur étant un programme souvent complexe. E.F. Moore [43] a ainsi écrit un compilateur d'un langage assembleur Piton vers du code machine pour un processeur FM8502 en utilisant le prouveur automatique ACL2. Et plus récemment, A. Chipala [12] a prouvé un compilateur (Lambda Tamer) d'un langage fonctionnel vers un assembleur. X. Leroy a prouvé un compilateur (CompCert) d'un gros sous-ensemble du langage C vers l'assembleur PowerPC [36] (ainsi que les architectures x86 et ARM). Concernant la vérification d'un compilateur de langages déclaratifs, il y a eu celle d'un compilateur de Prolog vers la WAM [54] en 1992. La WAM est cependant une machine d'assez haut niveau en comparaison des exemples précédemment cités.

Les langages synchrones, quant à eux datent du début des années 1980. Étant essentiellement conçus pour le temps réel critique, ils ont très vite eu une sémantique formelle, afin de pouvoir prouver des propriétés sur les programmes utilisés. Cependant, les outils développés pour prouver ces propriétés dépendent soit du code source, soit du code produit, mais ne font pas le lien entre le code source et le code produit. Pouvoir raisonner sur le code source est fondamental lors du développement, mais il est aussi nécessaire de pouvoir transférer ce raisonnement au code produit, c'est le but de la préservation de la sémantique au travers de la compilation. Certifier formellement un compilateur est très long, et tend à geler le compilateur certifié, toute amélioration impliquant de refaire tout ou partie des preuves. En 1998, dans le cadre du projet européen SACRES[4] (EP 20897), A. Pnueli propose non pas de prouver le processus de compilation, mais son résultat. C'est ce qu'il

appelle la « Translation Validation »[49, 48, 50]. Il utilise cette méthode pour produire un compilateur du langage SIGNAL vers C, dont il vérifie que le code produit bisimule le code source. Contrairement à ce qui a été fait dans cette thèse, il n'y a pas de vérification des étapes intermédiaires de production de code, et il n'y a pas de contraintes syntaxique sur les programmes. Cette méthode est donc plus souple d'utilisation, mais peut souffrir de temps de vérifications particulièrement longs, puisque le programme de vérification n'a pas connaissance du processus de compilation et ne peut donc pas en retirer d'information. Il faut alors utiliser des outils de model checking pour créer un script de preuve vérifiable par un module de vérification (un outil appelé TLV). Imposer des contraintes syntaxiques fortes permet de vérifier plus rapidement le code. Outre l'approche différente employée pour valider le code, cette thèse se distingue des travaux de A. PNUELI par l'utilisation d'un langage plus simple que celui de SIGNAL. SIGNAL est un langage de haut niveau disposant d'un calcul d'horloge plus complexe, permettant à un nœud de manipuler de flots qui ne sont pas tous sur la même échelle de temps (polychronie). Le calcul d'horloge présenté ici, est plus simple, tout est hiérarchique avec une unique horloge de base pour le système. L'utilisation de types énumérés (autres que le booléens) à des fins d'échantillonnages de flots n'existe pas en SIGNAL, ni en LUSTRE. Cette fonctionnalité n'a rien d'essentiel, mais permet de produire plus simplement du code efficace et améliore la lisibilité du code, en mettant en évidence les modes qui ne peuvent être actifs au même instant.

La formalisation de sémantiques synchrones en vue de leur instrumentalisation afin de prouver diverses propriétés dans un assistant de preuve a intéressé de nombreuses équipes de recherche. On doit par exemple l'implémentation des types coinductifs en COQ à Eduardo GIMÉNEZ[25], qui a par ailleurs travaillé avec Emmanuel LEDINOT à la certification de SCADE V3[24] dans l'assistant de preuve COQ. Cette entreprise n'a malheureusement pas été menée à terme. D. KAPLAN-TERRASSE[58] a certifié la compilation d'un fragment du compilateur d'ESTEREL (v.5), ce fragment est le fragment « pur » (le noyau du langage), privé des instructions `loop`, `pause` et `suspend`, afin de ne se focaliser que sur la partie combinatoire d'ESTEREL, c'est-à-dire juste ce qui se passe au premier instant.

De nombreux développements sur la coinduction en COQ sont apparus suite aux travaux de E. GIMÉNEZ. Ainsi :

- D. NOVAK et al.[46] ont défini en COQ une formalisation coinductive des flots utilisés dans SIGNAL, ainsi que la sémantique de primitives de ce langage (échantillonnage, décalage, extraction d'horloge).
- S. BOULMÉ et G. HAMON ont proposé un encodage de LUCID SYNCHRONE dans COQ[9]. Cet encodage permet de prouver des propriétés sur les programmes LUCID SYNCHRONE, mais ne repose pas sur la définition en COQ d'un arbre de syntaxe abstraite (un tel encodage est dit superficiel). L'avantage d'un tel encodage est qu'il permet d'utiliser directement le langage COQ pour faire des preuves, plutôt que de passer par une interprétation de la syntaxe abstraite. En contre partie, COQ ne permet pas l'introspection, et il est donc impossible d'écrire en COQ un compilateur efficace qui utilise cet encodage.
- S. COUPET-GRIMAL et L. JAKUBIEC ont travaillé sur la vérification de circuits en utilisant la coinduction en COQ[16].

D'un point de vue plus industriel, un effort vers une certification plus formelle de langages pour les application critiques a été entrepris en 2000 chez ESTEREL-TECHNOLOGIES

pour la conception de leur nouveau compilateur SCADE, reprenant les principes introduits par Lucid Synchrone[51, 15, 14], avec un noyau utilisant les mêmes primitives que Ls, et un encodage d'automates hiérarchiques su ces primitives. Ce compilateur (SCADE V6) est commercialisé depuis 2008, et qualifié DO-178B (Level A). Il n'y a pas de preuves sur papier de préservation de sémantique, mais le générateur de code même a été écrit en Ocaml, plutôt qu'en C comme c'est le cas habituellement, ce qui permet de raisonner plus aisément sur le compilateur, et a considérablement diminué la taille du code. Cette thèse s'inscrit donc dans la continuité de ces travaux en donnant un cadre formel pour pouvoir prouver formellement certaines propriétés attendues. Ce besoin de formalisme dans l'industrie du temps réel critique, devrait se faire plus fort avec l'apparition de nouvelles qualifications (DO-178C), autorisant l'utilisation de méthodes formelles pour remplacer une partie des tests. Le projet GeneAuto vise à produire une chaîne de développement open-source de Simulink/Stateflow/Scicos vers C pour les systèmes temps-réel. Ce projet cherche à intégrer des méthodes formelles dans leur chaîne de développement, ainsi des membres de ce projet ont formalisé et prouvé un algorithme d'ordonnancement d'exécution de nœuds[32], pour plus de détails, on peut consulter la thèse de N. Izerrouken[31]. Cette thèse contient la formalisation relative aux blocs Simulink, ainsi que les outils d'ordonnancement de ces blocs en Coq.

Pour ce qui est du langage source de la chaîne de compilation utilisés dans cette thèse, c'est une reprise du langage défini par Biernacki et al. [8] afin de pouvoir y définir plus simplement les automates de modes à états réinitialisables [1]. Quelques aménagements ont été faits, afin qu'une expression ait une horloge simple (et non un tuple d'horloges). La réinitialisation sur une constante donnée plutôt que la valeur « vrai » d'un flot booléen est juste un artifice pour ne pas avoir à définir nativement les booléens dans le langage.

14.2 Contributions

La compilation de langages synchrones n'est pas nouvelle, et comme pour le temps réel critique la sûreté est de mise, la plupart de ces langages sont très simples et ont une sémantique formalisée ou aisément formalisable.

Toujours pour des raisons de sûreté, les langages synchrones sont souvent conçus avec des outils qui permettent de vérifier certaines propriétés des programmes.

Par contre la sûreté du compilateur même n'est souvent due qu'à une étude détaillée du compilateur par des experts.

Cette thèse est allée un peu plus loin en proposant une compilation dont le résultat est vérifié par le compilateur lui même pour s'assurer que le code engendré est correct vis-à-vis du code source ; c'est-à-dire que soumis aux mêmes entrées, les sorties définies par la sémantique coïncident sous l'hypothèse qu'une telle sortie existe bien dans la sémantique du programme source.

Un autre aspect intéressant et la possibilité de réinitialiser un nœud sans avoir à le faire manuellement ce qui serait source d'erreurs ni au travers d'outils en lesquels il faudrait

1. Initialement un langage au dessus de Ls pour décrire les automates hiérarchiques était prévu, mais une sémantique trop complexe de ce langage a amené au retrait de celui-ci de la chaîne.

alors avoir confiance.

Enfin, la compilation de ce langage est bien plus modulaire que des langages tels que Lustre, ce qui peut présenter un intérêt dans l'industrie, car les différents éléments peuvent être compilés séparément, ce qui évite d'avoir à créer de gros automates qu'il faut ensuite réduire comme c'est le cas en Lustre.

Pour les parties non prouvées en Coq, les propriétés à valider ont cependant été formulées et prouvées à la main.

14.3 Perspectives

Il y a à l'heure actuelle très peu de compilateurs entièrement prouvés avec des assistants de preuve, que ce soit pour des langages synchrones ou non. Les perspectives sont donc encore très peu explorées, même si à l'heure actuelle de plus en plus de groupes de recherche s'orientent dans cette voie, principalement ouverte par le projet CompCert.

En ce qui concerne la compilation de Ls vers Obc, il y a encore du travail à accomplir pour terminer les preuves restantes dans un assistant de preuve. À l'heure actuelle la partie compilation certifiée dans l'assistant de preuve Coq s'arrête une fois le code traduit dans Lsn, la préservation de Lsn vers Lsni est bien prouvée, mais par manque de temps, la partie qui vérifie la bonne traduction n'a pas été écrite. Il y a également des questions à se poser vis-à-vis de quelles optimisations implémenter, et quelles relations doivent être vérifiées entre le code optimisé et celui qui ne l'est pas afin de garantir la préservation de la sémantique. Quelques suggestions sont proposées en annexes, bien que non prouvées dans l'assistant de preuve Coq. La question de l'optimisation est intéressante puisque à l'heure actuelle, la plupart des optimisations proposées par des compilateurs tels que `GCC` ne sont pas exploitées afin d'éviter l'introduction d'erreurs dues à des optimisations incorrectes. Apporter des optimisations devrait faire diminuer les pires temps d'exécutions et par la même rendre le système plus fiable en corrigeant plus tôt les anomalies rencontrées liées à l'environnement.

Les pistes à explorer portent aussi sur comment interfacer les sémantiques d'autres langages en amont (vers Ls) comme en aval (depuis Obc).

Ces autres langages posent un défi, car leurs sémantiques peuvent ne pas bien s'exprimer dans notre cadre, et réciproquement, la sémantique de Ls ou Obc peut mal s'exprimer dans celle d'un autre langage.

Parmi ces langages, il y a, en amont, toutes les extensions possibles de Ls qui ont été discutées pour ajouter par exemple des automates, ou gérer des horloges complexes en entrées et sorties de nœuds. Avoir cette dernière fonctionnalité, permettrait de pouvoir compiler du code Lustre directement (mis à part les `pre` non initialisés auxquels il faudrait donner une valeur par défaut), et par la même remplacer toutes les compilations vers Lustre vers des compilations vert Ls.

Il y a également, en aval, cette fois-ci, le projet CompCert bien sûr, mais aussi tout autre langage ayant une sémantique suffisamment simple, même si le compilateur C associé n'est pas prouvé. À défaut d'avoir toute la chaîne jusqu'au langage machine de prouvée, il faudra s'arrêter avant et certifier le compilateur C par des experts.

Dans cette dernière partie apparaît comme problématique qu'on se donne la sémantique d'un programme clos, c'est-à-dire qui correspond à l'exécution d'un `main` en C, alors qu'il n'y a pas de tel `main` dans Ls. La relation entre de telles sémantiques et celles utilisée dans notre chaîne n'est donc pas forcément évidente et plusieurs approches ont été avancées dans cette thèse.

Bibliographie

[1] *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987.

[2] C. André. Syncharts : a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Apr. 1996.

[3] L. Augustsson and K. Petersson. Silly type families * DRAFT, 1994.

[4] A. Benveniste. Safety critical embedded systems design : the sacres approach, 1998.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), Jan. 2003.

[6] G. Berry, G. Gonthier, A. B. G. Gonthier, and P. S. Laltte. The esterel synchronous programming language : Design, semantics, implementation, 1992.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[8] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Not.*, 43(7) :121–130, June 2008.

[9] S. Boulmé and G. Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`.

[10] P. Caspi and M. Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.

[11] A. Chlipala. *Certified Programming with Dependent Types*.

[12] A. J. Chlipala. *Implementing certified programming language tools in dependent type theory*. PhD thesis, Berkeley, CA, USA, 2007. AAI3311660.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.

[14] J.-L. Colaço and M. Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[15] J.-L. Colaço and M. Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, Aug. 2004.

[16] S. Coupet-Grimal and L. Jakubiec. Hardware verification using co-induction in coq. In *International Conference on Theorem Proving in Higher Order Logics, TPHOL'99*, pages 91–108, Nice, France, 1999. Springer Verlag.

[17] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, Oct. 1991.

[19] N. G. de Bruijn. AUTOMATH, a language for mathematics. T.H., Department of Mathematics, Eindhoven University of Technology, Nov. 1968.

[20] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 28(6) :237–247, June 1993.

[21] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[22] T. Gautier, P. L. Guernic, and J.-P. Talpin. Polychronous Design of Real-Time Applications with Signal, 2008. ARTIST Survey of Programming Languages, Alan Burns, Ed., http ://www.artist-embedded.org/artist/ARTIST-Survey-of-Programming,1489.html.

[23] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages : application to arrays in a lustre compiler. *SIGPLAN Not.*, 47(5) :51–60, June 2012.

[24] E. Gimenez and E. Ledinot. Certification de SCADE V3. Rapport final du projet GENIE II, Verilog SA, Janvier 2000.

[25] C. E. Giménez. *Un calcul de constructions infinies et son application à la vérification de programmes communicants.* These, École Normale Supérieure de Lyon, Dec. 1996. 186 pages.

[26] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50 :1–102, 1987.

[27] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[28] D. Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, June 1987.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1) :53–56, Jan. 1983.

[30] W. A. Howard. The Formulae-As-Types Notion Of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.

[31] N. Izerrouken. *Développement prouvé de composants formels pour un générateur de code embarqué critique pré-qualifié*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, juillet 2011.

[32] N. Izerrouken, X. Thirioux, M. Pantel, and M. Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. 2008.

[33] G. Kahn. *The semantics of a simple language for parallel programming*, volume 74, pages 471–475. North-Holland, 1974.

[34] R. Lalement. *Logique, réduction, résolution*. Dunod, 1990.

[35] L. Lamport. Specifying concurrent systems with TLA+, 1999.

[36] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[37] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[38] F. Maraninchi and Y. Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, (27) :61–92, 2001.

[39] The Coq development team. *The Coq proof assistant reference manual*. TypiCal Project, 2011. Version 8.3.

[40] Xavier Leroy. *The CompCert C Verified Compiler*. CompCert, 2012. Version 1.10.

[41] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[42] E. F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematical Studies*, 34 :129–153, 1956.

[43] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5 :461–492, 1989.

[44] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5) :333–344, May 1998.

[45] U. Norell. Dependently typed programming in agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[46] D. Nowak, J. R. Beauvais, and J. P. Talpin. Co-inductive axiomatisation of synchronous language. In *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'98)*. Springer Verlag, October 1998.

[47] C. Paulin-mohring and I. Futurs. A constructive denotational semantics for kahn networks in coq, 2007.

[48] A. Pnueli, O. Shtrichman, and M. Siegel. Translation Validation for Synchronous Languages. In *International Colloquium on Automata, Languages and Programming*, pages 235–246, 1998.

[49] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. pages 151–166. Springer, 1998.

[50] A. Pnueli, O. Strichman, and M. Siegel. Translation validation : From SIGNAL to C. In E.-R. Olderog and B. Steffen, editors, *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *Lecture Notes in Computer Science*, pages 231–255. Springer, 1999.

[51] M. Pouzet. *Lucid Synchrone : un langage synchrone d'ordre supérieur*. Paris, France, Nov. 2002. Habilitation à diriger les recherches.

[52] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3*. These, Institut National Polytechnique de Grenoble - INPG, Nov. 1991. 141 pages.

[53] P. Raymond. *LUSTRE V4 manual*, Feb. 2000.

[54] D. M. Russinoff. A verified prolog compiler for the warren abstract machine. *Journal of Logic Programming*, 13 :367–412, 1992.

[55] D. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry, and Logic*, number 274 in Lecture Notes in Mathematics, pages 97–136. Springer-Verlag, 1972.

[56] D. Scott and C. Strachey. Toward A Mathematical Semantics for Computer Languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., Apr. 1971. Polytechnic Press.

[57] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, and G. Borios. Computing the worst case execution time of an avionics program by abstract interpretation. In *In Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[58] D. Terrasse. Vers la certification du compilateur v5 de Esterel dans coq. Technical Report 4092, INRIA, December 2000.

[59] A. Turing. The early british computer conferences. chapter Checking a large routine, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[60] M. van Eekelen, S. Smetsers, and R. Plasmeijer. Graph rewriting semantics for functional programming languages. In *In : Proc. of the CSL '96, Fifth Annual conference of the European Association for Computer Science Logic*, pages 106–128. Springer-Verlag, 1996.

[61] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[62] S. Wehr and M. M. Chakravarty. ML modules and haskell type classes : A constructive comparison. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 188–204, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3) :1–53, Apr. 2008.

Annexes

Annexe A

Optimisations

Normalisation

Une optimisation est une opération qui fait diminuer une grandeur liée au code, sans pour autant en altérer la sémantique. En pratique il s'agit presque exclusivement du temps d'exécution, de l'occupation mémoire, ou d'un rapport entre les deux.

La factorisation de code qui peut être obtenue par une implémentation intelligente de l'outil externe d'introduction de nouveaux noms de flots se traduit à la fois par un gain en mémoire et en temps d'exécution sur le code final ; cette optimisation a été présentée dans cette thèse, car elle est gratuite du point de vue des preuves.

On peut alors se demander, quitte à devoir faire des preuves supplémentaires quelles optimisations pourraient être raisonnables.

Dans les optimisations proposées, aucune n'a été formalisée en Coq.

Élimination d'équations inutiles (EI)

Différentes passes d'optimisations peuvent éliminer des dépendances entre données, on peut alors se retrouver dans une situation où des équations définissent des noms de flots dont ne dépend plus le flot de sortie. C'est une forme d'élimination de code mort.

```
node exemple (b:bool) returns (c:bool);
var x,y:int32;
let x = 1;
    (c,y) = (b,0);
tel;
```

peut être simplifié en :

```
node exemple (b:bool) returns (c:bool);
var y:int32;
let (c,y) = (b,0);
tel;
```

Là encore, il est plus simple de faire appel à un programme externe pour savoir quels noms de flots peuvent être supprimés.

La vérification est assez aisée, il suffit de parcourir toutes les équations définissant des noms de flots à garder et de vérifier que les expressions associées ne contiennent aucune occurrence des noms à retirer, et que ces noms à retirer ne soient ni l'entrée ni la sortie. Il serait préférable de faire cette optimisation une fois la distribution des tuples accomplie, car on évite alors les situations où une équation définit un nom de flot à garder et un nom de flot à rejeter (comme c'est le cas dans l'exemple).

Minimisation d'automates (MA)

La factorisation d'expressions n'est pas toujours aussi puissante qu'on le souhaiterait. Par exemple dans le code :

```
y = 0 fby (1+y);
z = 0 fby (1+z);
```

on sent bien que `y` et `z` sont unifiables, et on souhaiterait pouvoir écrire :

```
y = 0 fby (1+y);
z = y;
```

Malheureusement, ce n'est pas possible avec seulement la factorisation d'expressions.

La cause de ce problème réside dans le fait que les équations sont définies récursivement dans le cas des `fby`. En essayant de calquer sur la passe d'introduction de nouvelles équations, on obtiendrait :

```
t = 0 fby (1+t);
y = t;
z = t;
```

La passe de substitution utilisée pour valider le nouveau système donnerait :

```
y = 0 fby (1+t);
z = 0 fby (1+t);
```

qui non seulement n'est pas le système d'origine, mais en plus est mal défini.

On peut définir une relation d'équivalence plus générale que celle décrite dans la phase d'introduction de nouvelles équations. On se donne un ensemble de nouveaux noms de flots N, et pour chaque $x_n \in N$, on se donne un ensemble U_{x_n} ainsi qu'une expression e_{x_n}. On définit alors E_{x_n} inductivement comme un ensemble d'expressions par :

$$\textsc{Refl}\ \frac{x_n \in N}{x_n \in E_{x_n}} \qquad \textsc{Trans1}\ \frac{e_1 \in E_{x_n} \qquad x_m \in N \qquad x_x \in U_{x_m} \qquad e_2 \overset{x_m \to x_x}{\sim} e_1}{e_2 \in E_{x_n}}$$

$$\textsc{Trans2}\ \frac{e_1 \in E_{x_n} \qquad x_m \in N \qquad e_2 \overset{x_m \to e_{x_m}}{\sim} e_1}{e_2 \in E_{x_n}}$$

La relation qui nous intéresse entre le code d'origine et le code produit est :

- pour tout flot nouvellement introduit x_n dans N, tous les flots $x_i \in U_{x_n}$ sont définis par x_i`=`x_n`;`

– pour toute occurrence o d'un nom de flot x_n, nouvellement introduit, il existe une expression e_o de E_{x_n}, telle qu'en remplaçant toutes les occurrences o par leur expression correspondante e_o dans le système produit et en éliminant toutes les équations définissant les flots de N, on retrouve le système d'origine

Dans cet exemple :

```
x = 0 fby y;
y = 1 fby x;
z = 0 fby t;
t = 1 fby z;
```

On construit $N = \{\mathtt{u}, \mathtt{v}\}$, $U_{\mathtt{u}} = \{\mathtt{x}, \mathtt{z}\}$, $U_{\mathtt{v}} = \{\mathtt{y}, \mathtt{t}\}$, $e_{\mathtt{u}} = \mathtt{0 = fby\ u}$, $e_{\mathtt{v}} = \mathtt{1 = fby\ v}$ et on produit le système :

```
u = 0 fby v;
v = 1 fby u;
x = u;
y = v;
z = u;
t = v;
```

On peut vérifier trivialement la première propriété demandée. Quant à la seconde, il suffit de remplacer la troisième occurrence de `u` par `0 fby y` et la quatrième par `0 fby t` (qui sont bien dans $E_{\mathtt{u}}$) ; de même, il suffit de remplacer la troisième occurrence de `v` par `1 fby x` et la quatrième par `1 fby z` (qui sont bien dans $E_{\mathtt{v}}$). Après retrait des définitions de `u` et `v`, on obtient bien le système d'origine.

Il est à noter que même dans des cas où il n'y a pas de définition récursive, ce genre d'optimisation peut aussi être bénéfique. Par exemple, dans le système :

```
a = x+y;
b = x+y;
c = 0 fby a;
d = 0 fby b;
```

il n'y a pas de factorisation possible pour les équations définissant `c` et `d`, contrairement à la situation où `a` et `b` sont déjà développés dans les définitions de `c` et `d`. Par contre l'optimisation de minimisation des automates permet d'unifier un peu mieux le système en :

```
t = x+y;
a = t;
b = t;
c = 0 fby t;
d = 0 fby t;
```

qui peut ensuite être convenablement factorisé.

Ce genre d'optimisation est bien connu et utilisé notamment dans le compilateur Lustre [52].

Propagation des non constantes (PNC)

L'idée des optimisations étant de trouver une meilleure forme pour la sémantique d'une expression, l'idéal serait que le compilateur puisse évaluer une expression pour la réduire. Cela implique de connaître la sémantique de chaque expression, or le compilateur ne la connaît pas (il ne manipule que la syntaxe). Il existe cependant des règles de réductions que le compilateur peut facilement implémenter, ce sont celles des fusions de flots lorsque l'on sait que la valeur du flot est une constante connue. Ce genre de réduction est cependant très limité, car il est rare d'avoir des noms de flots qui ne soient que de simples constantes.

En revanche, il est possible dans certains cas de démontrer qu'un flot ne peut jamais prendre un certain constructeur comme valeur. Dans ce cas, toute fusion de flot peut être simplifiée dès lors que l'on sait attribuer une valeur par défaut aux branches concernées. Si on veut ajouter une telle optimisation, il serait alors judicieux d'ajouter dans la syntaxe un mot clé `_mil_` pour indiquer que ce cas ne peut se produire. On pourrait alors considérer `_nil_` comme une constante habitant tous les types, comme $\bot$ en Haskell, et se rapprocher de Lustre en pouvant définir `pre` par `_nil_ fby` (voir la section 13.1.2).

```
x = merge i (A -> A when A(i))
            (B -> B when B(i))
            (C -> A when C(i)); -- x is never equal to C
y = merge x (A -> 0 when A(x))
            (B -> 1 when B(x))
            (C -> f(z) every P(p)); -- never happens
z = merge x (A -> A when A(x))
            (B -> A when B(x))
            (C -> B when C(x)); -- z is always equal to A
```

pourrait alors être simplifié en :

```
x = merge i (A -> A when A(i))
            (B -> B when B(i))
            (C -> A when C(i)); -- x is never equal to C
y = merge x (A -> 0 when A(x))
            (B -> 1 when B(x))
            (C -> _nil_ when C(x)); -- never happens
z = A; -- z is always equal to A
```

Pour ce genre d'optimisation, on aurait besoin d'un oracle (c'est à dire d'un outil externe) qui proposerait un nom de flot x, une constante k ainsi qu'un argument justifiant que x ne peut pas produire k. Savoir si un nom de flot peut retourner une constante donnée est trivialement CoNP-dur, il suffit de considérer le cas où tous les noms de flots considérés sont booléens et où on n'utilise que les primitives `when` et `merge` pour s'en persuader, cette optimisation peut donc s'avérer coûteuse dans le cas général. Mais des analyses moins fines et peu coûteuses sont aussi possibles.

Du point de vue des preuves pour ajouter cette optimisation, il faut définir un prédicat sur la syntaxe qui exprime que toutes les valeurs finales que peut prendre x sont toutes différentes de k. Dans le cadre général la définition même de ce prédicat peut être complexe à formuler et fera intervenir l'argument donné par l'oracle. La preuve de préservation devrait se découper en deux parties, la première énonce que si la relation est bien vérifiée,

alors x ne peut jamais prendre `_nil_` comme valeur, la seconde énonce que si x ne peut jamais prendre `_nil_` comme valeur alors on peut remplacer les expressions concernées par cette propriété par `nil_`. La seconde partie est facile à démontrer, mais la première peut être complexe.

Dépliage de code (DC)

Le dépliage de code («inlining» dans la terminologie anglo-saxonne) consiste à remplacer un appel de nœud par sa définition. En soi c'est une optimisation assez limitée du temps d'exécution, et n'est intéressant en termes de taille de code que si le code déplié n'est utilisé qu'une fois. Cependant après un dépliage de code, de nouvelles optimisations peuvent devenir possibles.

```
node cesaro (i:int32) returns (ces:int32);
var cpt : int32;
    cum : int32;
let cpt = 1 fby (cpt+1);
    cum = 0 fby (i+cum);
    ces = (i+cum)/cpt;
tel;

node cesaro2 (i,j:int32) returns (ces2:int32);
var x : int32;
    y : int32;
let x = cesaro(i) every False(True);
    y = cesaro(j) every False(True);
    ces2 = (x+y)/2;
tel;
```

Peut être transformé en :

```
node cesaro2 (i,j:int32) returns (ces2:int32);
var x_cpt, x_cum, y_cpt, y_cum, x, y : int32;
let x_cpt = 1 fby (x_cpt+1);
    x_cum = 0 fby (i+x_cum);
    y_cpt = 1 fby (y_cpt+1);
    y_cum = 0 fby (j+y_cum);
    x = (i+x_cum)/x_cpt;
    y = (j+y_cum)/y_cpt;
    ces2 = (x+y)/2;
tel;
```

et permettre l'optimisation (MA) :

```
node cesaro2 (i,j:int32) returns (ces2:int32);
var x_cpt, x_cum, y_cum, x, y : int32;
let x_cpt = 1 fby (x_cpt+1);
    x_cum = 0 fby (i+x_cum);
    y_cum = 0 fby (j+y_cum);
    x = (i+x_cum)/x_cpt;
    y = (j+y_cum)/x_cpt;
    ces2 = (x+y)/2;
```

```
tel;
```

Le choix de savoir quel nœud doit être déplié n'est pas trivial, et une fois de plus, on demandera à un oracle quel nœud déplier. Cependant, la vérification consistera seulement à vérifier que le nœud existe bien, et que son code est accessible.

Pour la preuve de cette optimisation, il y a cependant un petit problème, en effet, l'environnement sémantique ne garde pas la trace des codes de nœuds, cela fait en effet partie de l'abstraction sémantique des nœuds, l'environnement sémantique ne contient que des relations entre entrée et sortie. Si on veut pouvoir implémenter cette optimisation, il faut garder quelque part le code des nœuds à déplier ainsi qu'une preuve que leur sémantique est bien celle de l'environnement. Cette preuve existe puisque c'est la définition du nœud à déplier, il faut par contre en garder la trace du côté de l'assistant de preuve.

La preuve en soi serait assez lourde à réaliser, il faudrait introduire de nouveaux noms, s'assurer qu'ils ne sont pas utilisés, garder une correspondance entre ces nouveaux noms et les anciens. De plus, cette passe ne pourrait s'appliquer que lorsqu'il n'y a jamais de réinitialisation du nœud, car on ne dispose de bloc `every` que pour les applications de nœuds et pas pour les systèmes d'équations.

Dans le cas où il y a effectivement des réinitialisations de nœud, bien qu'on puisse rajouter toute une machinerie pour réimplémenter le fonctionnement de `every` pour les systèmes, il peut être préférable d'essayer de déplier le code une fois dans Obc, en effet dans ce langage les manipulations de mémoire sont explicites, et plus faciles à réaliser. Cependant, si on déplace cette optimisation dans Obc, alors il faudra refaire les preuves des autres optimisations dans Obc pour pouvoir les appliquer. Généralement lorsqu'il y a des réinitialisations, assez peu d'optimisations sur la mémoire deviennent possibles après des dépliages de nœuds, puisqu'elles ne peuvent se faire qu'entre nœuds réinitialisés sur la même expression, ce qui en limite (mais n'élimine pas pour autant) l'intérêt de ces optimisations.

A.0.1 Conclusion

De nombreuses optimisations assez simples pourraient être mises en places et démontrées correctes vis à vis de la préservation de la sémantique. Les preuves ne seraient cependant pas forcément aussi simples, et certaines optimisations pousseraient à une révision assez profonde du formalisme (comme l'ajout de `_nil_`). Enfin, certaines optimisations sont plus adaptées à d'autres parties de la compilation. Ainsi, si toutes ces optimisations étaient disponibles, il vaudrait mieux :

(DC) Commencer par déplier des nœuds sans réinitialisation proposés par un outil externe non certifié. Cette passe n'est pas une optimisation, mais peut favoriser l'apparition de possibilités d'optimisations (PNC), (MA) et de factorisation d'expressions.

(PNC) Puis propager les non constantes proposées par un outil externe. Ceci aurait besoin de la formalisation de ce qu'est un argument selon lequel un nom de flot ne peut pas prendre une constante donnée pour valeur. Cette passe peut éliminer des dépendances syntaxiques entre noms de flots et donc donner des possibilités d'optimisations (EI). Elle peut aussi permettre d'unifier d'autres expressions et donner des possibilités d'optimisation (MA) et de factorisation d'expressions. Des occasions

d'appliquer cette optimisation devraient être assez rares pour du code entré à la main. Cependant une génération de code, ou une passe de (DC), peut donner des occasions d'appliquer cette optimisation.

(MA) Ensuite, après avoir normalisé le code sans la passe de transition, pour éliminer les tuples et avoir des équations prêtes à être unifiées, on peut unifier le plus possible de noms de variables selon les conseils d'un outil externe non certifié.

(EI) Enlever les équations et noms de flots devenus inutiles.

avant de finir par la transition de Ls vers Lsn.

Il se trouve qu'ainsi, il suffit d'une seule grosse passe d'optimisations, chaque optimisation ne pouvant normalement pas créer d'opportunité de refaire une passe d'optimisation déjà faite. Une passe de (EI) pourrait cependant être insérée entre deux passes pour alléger le travail de la passe suivante. Il est aussi à noter que l'introduction de nouvelles équations avant l'optimisation (MA) n'a pas nécessairement besoin de vérifier les propriétés P_1 à P_6, car il est toujours possible de lancer tout le processus de normalisation après la dernière optimisation accomplie, processus qui, lui, devra bien s'assurer de la vérification de ces propriétés.

Ordonnancement

L'ordonnancement peut également être sujet à des optimisations.

En effet, l'ordonnancement décide de la structure de flot de contrôle. Typiquement, le fragment suivant :

```
x = a when A(i);
y = u when U(j);
z = v when A(i);
```

amènera au code Obc suivant :

```
switch (i) {
        A -> { x = a; }
}
switch (j) {
        U -> { y = u; }
}
switch (i) {
        A -> { z = v; }
}
```

Ce code est très inefficace puisqu'il effectue jusqu'à trois branchements, alors que deux seraient suffisants dans le code suivant, généré en échangeant les deux dernières équations :

```
switch (i) {
        A -> { x = a;
               z = v; }
}
switch (j) {
        U -> { y = u; }
}
```

Un ordonnanceur intelligent doit donc regrouper toutes les équations échantillonnées sur les mêmes valeurs afin de minimiser le nombre de branchements.

Si on ne veut pas faire d'optimisation qui change l'ordre des affectations après passage en code impératif, il est important de bien ordonnancer le code ici, et de se poser les questions quant à comment faire en sorte de maximiser les optimisations des passes suivantes.

Si au contraire, on s'autorise à déplacer les affectations dans les optimisations suivantes, ce qui est possible tant que les dépendances entre variables ne sont pas cassées, on peut être beaucoup moins exigeant quant à l'ordonnanceur choisi. On peut même se contenter d'un ordonnanceur qui génère du code complètement inefficace si on a des optimisations qui permettent de réordonnancer (efficacement) du code déjà ordonnancé (inefficacement) ; l'ordonnanceur inefficace n'est alors ici que pour assurer la préservation de la sémantique.

Optimisations dans Obc

Élimination de code mort par interprétation abstraite

L'interprétation abstraite est une analyse statique de programmes [17] qui étudie les domaines sur lesquels évoluent des variables du programme pour vérifier certaines propriétés. On peut utiliser certaines techniques d'interprétation abstraite afin de découvrir des blocs dont le code ne pourra pas être exécuté. Dans la sémantique que l'on s'est donnée, cette technique a ses limites puisque le compilateur ne connaît pas les opérateurs et les nœuds sont des symboles non interprétés.

L'élimination de code mort n'a qu'un très faible impact sur les performances puisque, par définition, le code concerné ne sera jamais exécuté. En revanche, il facilite les autres analyses et réduit la taille du code en mémoire.

Cette optimisation fait écho à la propagation de non-constantes vue dans la partie précédente.

Les remarques sont les mêmes (problème **CoNP** dur), à ceci près que cette fois, il n'y a plus la problématique du `_nil_` puisque le `switch` peut ne pas traiter tous les cas.

Dépliage de code

Le dépliage de code [1] est classique en compilation, il consiste à aller chercher le code d'une fonction et l'insérer là où elle est appelée. Le point positif est que cela évite un appel de fonction, chose assez coûteuse puisqu'il faut sauvegarder le contexte, sauter à l'adresse de la fonction, retourner où on était et restaurer le contexte. En contrepartie, si la fonction est appelée à plusieurs endroits et est assez complexe, déplier systématiquement la fonction allonge la taille du code, ce qui peut affecter les performances.

Quoiqu'il en soit, cette optimisation n'est pas proposée ici, car d'une part notre sémantique ne connaît pas le code des autres nœuds et d'autre part le prédicat de dépliage est assez compliqué à définir puisqu'il faut ajouter à l'environnement toutes les variables

1. *inlining* en anglais

locales du nœud déplié en les renommant pour éviter les conflits et remplacer les mémoires abstraites d'instances par leur mémoire concrète.

Cette optimisation n'est cependant pas impossible dans le cadre de cette chaîne de compilation, puisque le premier point peut être contré en maintenant du côté de l'assistant de preuve un environnement des nœuds avec leur code en plus de l'environnement des nœuds avec leur sémantique.

Annexe B

Codes intermédiaires

B.1 Interface

```
-- type declarations
type mode = Left | Right
type button = Pushed | Released
type chess_clock
;--end of type declarations, begin of operators declarations

-- returns True iff 1st argument is Released and 2nd is Pushed
fun trigger: button * button -> bool

-- returns the opposite mode
fun flip: mode -> mode

-- compute a chess clock to display it, only the active part is given
fun make_chess_clock: mode * int32 * int32 -> chess_clock

;--end of operators declarations, begin of nodes implementations
```

B.2 Flot de données

B.2.1 Code source (Ls)

```
-- some node to count time when active
node chrono_base (tic:unit) returns (time:int32);
let time = 0 fby (1+time);
tel;

-- to get the current mode
node get_mode (flip_button:button) returns (current_mode:mode);
var real_flip:bool;
    keep:mode when not real_flip; switch_:mode when real_flip;
let real_flip = trigger(Pushed fby flip_button, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
```

```
    current_mode =
      Left fby merge real_flip (False -> keep) (True -> switch_);
tel;

-- node to count time in minutes and seconds
node ms (tic:unit) returns (mins, secs: int32);
var reset:bool;
let mins = (chrono_base(tic) every reset)/60;
    secs = (chrono_base(tic) every reset)%60;
    reset = False fby ((mins==59) && (secs==59));
tel;

-- node to compute the chess clock in order to display it
node chess_clock (flip_button:button) returns (cc:chess_clock);
var current_mode:mode; m, s: int32;
let current_mode = get_mode(flip_button);
    (m,s) = merge current_mode
            (Left -> ms(Only when Left(current_mode)))
            (Right -> ms(Only when Right(current_mode)));
    cc = make_chess_clock (current_mode, m, s);
tel;
```

B.2.2 Après introduction de nouveaux flots (Ls)

```
node chrono_base (tic:unit) returns (time:int32);
var
    -- begin of new variables
    int32_1_0 : int32;
    int32_1_1 : int32;
    -- end of new variables
let
    -- begin of new equations
    int32_1_1 = 0 fby int32_1_0;
    int32_1_0 = 1+time;
    -- end of new equations
    time = int32_1_1;
tel;

node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when not real_flip;
    switch_ : mode on real_flip;
    -- begin of new variables
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
    -- end of new variables
let
    -- begin of new equations
    mode_4_1 = Left fby mode_4_0;
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_);
```

```
    button_1_0 = Pushed fby flip_button;
    -- end of new equations
    real_flip = trigger(button_1_0, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
    current_mode = mode_4_1;
tel;

node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    -- begin of new variables
    int32_1_0 : int32;
    bool_3_0 : bool;
    -- end of new variables
let
    -- begin of new equations
    bool_3_0 = (mins==59) && (secs==59);
    int32_1_0 = chrono_base(tic) every reset;
    -- end of new equations
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    reset = False fby bool_3_0;
tel;

node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    -- begin of new variables
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
    -- end of new variables
let
    -- begin of new equations
    unit_2_7 = Only when Right(current_mode);
    (int32_2_5, int32_2_6) = ms(unit_2_7) every not bool_2_4;
    bool_2_4 = True when Right(current_mode);
    unit_2_3 = Only when Left(current_mode);
    (int32_2_1, int32_2_2) = ms(unit_2_3) every not bool_2_0;
    bool_2_0 = True when Left(current_mode);
    bool_1_0 = True;
    -- end of new equations
```

```
    current_mode = get_mode(flip_button) every not bool_1_0;
    (m,s) = merge current_mode
            (Left -> (int32_2_1, int32_2_2))
            (Right -> (int32_2_5, int32_2_6));
    cc = make_chess_clock (current_mode, m, s);
tel;
```

B.2.3 Après distribution (Lsn)

```
node chrono_base (tic:unit) returns (time:int32);
var
    int32_1_0 : int32;
    int32_1_1 : int32;
let
    int32_1_1 = 0 fby int32_1_0;
    int32_1_0 = 1+time;
    time = int_32_1_1;
tel;

node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when False(real_flip);
    switch_ : mode when True(real_flip);
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
let
    mode_4_1 = Left fby mode_4_0;
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_);
    button_1_0 = Pushed fby flip_button;
    real_flip = trigger(button_1_0, flip_button);
    keep = current_mode when False(real_flip);
    switch_ = flip(current_mode when True(real_flip));
    current_mode = mode_4_1;
tel;

node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    int32_1_0 : int32;
    bool_3_0 : bool;
let
    bool_3_0 = (mins==59) && (secs==59);
    int32_1_0 = chrono_base(tic) every True(reset);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    reset = False fby bool_3_0;
tel;

node chess_clock (flip_button:button) returns (cc:chess_clock);
var
```

```
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
let
    unit_2_7 = Only when Right(current_mode);
    (int32_2_5, int32_2_6) = ms(unit_2_7) every False(bool_2_4);
    bool_2_4 = True when Right(current_mode);
    unit_2_3 = Only when Left(current_mode);
    (int32_2_1, int32_2_2) = ms(unit_2_3) every False(bool_2_0);
    bool_2_0 = True when Left(current_mode);
    bool_1_0 = True;
    current_mode = get_mode(flip_button) every False(bool_1_0);
    --
    m = merge current_mode (Left -> int32_2_1) (Right -> int32_2_5);
    s = merge current_mode (Left -> int32_2_2) (Right -> int32_2_6);
    --
    cc = make_chess_clock (current_mode, m, s);
tel;
```

B.3 Instantané

B.3.1 Après instanciation (Lsni)

```
node chrono_base (tic:unit) returns (time:int32);
var
    int32_1_0 : int32;
    int32_1_1 : int32;
let
    int32_1_1 = 0 fby int32_1_0;
    int32_1_0 = 1+time;
    time = int32_1_1;
tel;

node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when not real_flip;
    switch_ : mode when real_flip;
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
let
```

```
    mode_4_1 = Left fby mode_4_0;
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_);
    button_1_0 = Pushed fby flip_button;
    real_flip = trigger(button_1_0, flip_button);
    keep = current_mode when not real_flip;
    switch_ = flip(current_mode when real_flip);
    current_mode = mode_4_1;
tel;

node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    int32_1_0 : int32;
    bool_3_0 : bool;
ins
    chrono_base_2 : chrono_base;
let
    bool_3_0 = (mins==59) && (secs==59);
    int32_1_0 = chrono_base_2(tic) every reset;
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    reset = False fby bool_3_0;
tel;

node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
ins
    ms_2 : ms;
    ms_5 : ms;
    get_mode_8 : get_mode;
let
    unit_2_7 = Only when Right(current_mode);
    (int32_2_5, int32_2_6) = ms_2(unit_2_7) every not bool_2_4;
    bool_2_4 = True when Right(current_mode);
    unit_2_3 = Only when Left(current_mode);
    (int32_2_1, int32_2_2) = ms_5(unit_2_3) every not bool_2_0;
    bool_2_0 = True when Left(current_mode);
    bool_1_0 = True;
    current_mode = get_mode_8(flip_button) every not bool_1_0;
```

```
    m = merge current_mode (Left -> int32_2_1) (Right -> int32_2_5);
    s = merge current_mode (Left -> int32_2_2) (Right -> int32_2_6);
    cc = make_chess_clock (current_mode, m, s);
tel;
```

B.3.2 Après ordonnancement (Lsni)

```
node chrono_base (tic:unit) returns (time:int32);
var
    int32_1_0 : int32;
    int32_1_1 : int32;
let
    time = int32_1_1;              -- 3
    int32_1_0 = 1+time;            -- 2
    --
    int32_1_1 = 0 fby int32_1_0;   -- 1
tel;

node get_mode (flip_button:button) returns (current_mode:mode);
var
    real_flip : bool;
    keep : mode when not real_flip;
    switch_ : mode when real_flip;
    button_1_0 : button;
    mode_4_0 : mode;
    mode_4_1 : mode;
let
    real_flip = trigger(button_1_0, flip_button);                  -- 4
    current_mode = mode_4_1;                                       -- 7
    keep = current_mode when not real_flip;                        -- 5
    switch_ = flip(current_mode when real_flip);                   -- 6
    mode_4_0 = merge real_flip (False -> keep) (True -> switch_); -- 2
    --
    mode_4_1 = Left fby mode_4_0;                                  -- 1
    button_1_0 = Pushed fby flip_button;                           -- 3
tel;

node ms (tic:unit) returns (mins, secs: int32);
var
    reset : bool;
    int32_1_0 : int32;
    bool_3_0 : bool;
ins
    chrono_base_2 : chrono_base;
let
    int32_1_0 = chrono_base_2(tic) every reset;        -- 2
    mins = int32_1_0/60;                               -- 3
    secs = int32_1_0%60;                               -- 4
    bool_3_0 = (mins==59) && (secs==59);               -- 1
    --
    reset = False fby bool_3_0;                        -- 5
tel;
```

```
node chess_clock (flip_button:button) returns (cc:chess_clock);
var
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool when Left(current_mode);
    int32_2_1 : int32 when Left(current_mode);
    int32_2_2 : int32 when Left(current_mode);
    unit_2_3 : unit when Left(current_mode);
    bool_2_4 : bool when Right(current_mode);
    int32_2_5 : int32 when Right(current_mode);
    int32_2_6 : int32 when Right(current_mode);
    unit_2_7 : unit when Right(current_mode);
ins
    ms_2 : ms;
    ms_5 : ms;
    get_mode_8 : get_mode;
let
    bool_1_0 = True;                                                    -- 7
    current_mode = get_mode_8(flip_button) every not bool_1_0;          -- 8
    unit_2_7 = Only when Right(current_mode);                           -- 1
    bool_2_4 = True when Right(current_mode);                           -- 3
    (int32_2_5, int32_2_6) = ms_2(unit_2_7) every not bool_2_4;         -- 2
    unit_2_3 = Only when Left(current_mode);                            -- 4
    bool_2_0 = True when Left(current_mode);                            -- 6
    (int32_2_1, int32_2_2) = ms_5(unit_2_3) every not bool_2_0;         -- 5
    m = merge current_mode (Left->int32_2_1) (Right->int32_2_5);        -- 9
    s = merge current_mode (Left->int32_2_2) (Right->int32_2_6);        --10
    cc = make_chess_clock (current_mode, m, s);                         --11
tel;
```

B.3.3 Après génération de code (Obc)

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    int32_1_0 : int32;
    --
    time = mem.int32_1_1;
    int32_1_0 = 1+time;
    mem.int32_1_1 = int32_1_0;
  }
}

class get_mode {
  memory {
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
```

```
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    keep : mode;
    switch_ : mode;
    mode_4_0 : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { keep = current_mode; }
    }
    switch(real_flip) {
      True -> { switch_ = flip(current_mode); }
    }
    switch(real_flip) {
      False -> { mode_4_0 = keep; }
      True -> { mode_4_0 = switch_; }
    }
    mem.mode_4_1 = mode_4_0;
    mem.button_1_0 = flip_button;
  }
}

class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
    bool_3_0 : bool;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    bool_3_0 = (mins==59) && (secs==59);
    mem.reset := bool_3_0;
  }
}

class chess_clock {
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
```

```
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool;
    int32_2_1 : int32;
    int32_2_2 : int32;
    unit_2_3 : unit;
    bool_2_4 : bool;
    int32_2_5 : int32;
    int32_2_6 : int32;
    unit_2_7 : unit;
    --
    bool_1_0 = True;
    switch(bool_1_0) { False -> { get_mode.reset(get_mode_8); } }
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) { Right -> { unit_2_7 = Only; } }
    switch(current_mode) { Right -> { bool_2_4 = True; } }
    switch(current_mode) {
      Right -> { switch(bool_2_4) { False -> { ms_2.reset; } } }
    }
    switch(current_mode) {
      Right -> { (int32_2_5, int32_2_6) = ms_2.step(unit_2_7); }
    }
    switch(current_mode) { Left -> { unit_2_3 = Only; } }
    switch(current_mode) { Left -> { bool_2_0 = True; } }
    switch(current_mode) {
      Left -> { switch(bool_2_0) { False -> { ms_5.reset; } } }
    }
    switch(current_mode) {
      Left -> { (int32_2_1, int32_2_2) = ms_5.step(unit_2_3); }
    }
    switch(current_mode) {
      Left -> { m = int32_2_1; }
      Right -> { m = int32_2_5; }
    }
    switch(current_mode) {
      Left -> { s = int32_2_2; }
      Right -> { s = int32_2_6; }
    }
    cc = make_chess_clock (current_mode, m, s);
  }
}
```

B.3.4 Après développement (Obc)

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    int32_1_0 : int32;
    --
    time = mem.int32_1_1;
```

```
    int32_1_0 = 1+time;
    mem.int32_1_1 = int32_1_0;
  }
}

class get_mode {
  memory {
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    keep : mode;
    switch_ : mode;
    mode_4_0 : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { keep = current_mode; }
    }
    switch(real_flip) {
      True -> { switch_ = flip(current_mode); }
    }
    switch(real_flip) {
      False -> { mode_4_0 = keep; }
      True -> { mode_4_0 = switch_; }
    }
    switch(real_flip) {
      False -> { mem.mode_4_1 = mode_4_0; }
      True -> { mem.mode_4_1 = mode_4_0; }
    }
    mem.button_1_0 = flip_button;
  }
}

class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
    bool_3_0 : bool;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    bool_3_0 = (mins==59) && (secs==59);
    mem.reset := bool_3_0;
```

```
  }
}

class chess_clock {
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool;
    int32_2_1 : int32;
    int32_2_2 : int32;
    unit_2_3 : unit;
    bool_2_4 : bool;
    int32_2_5 : int32;
    int32_2_6 : int32;
    unit_2_7 : unit;
    --
    bool_1_0 = True;
    switch(bool_1_0) { False -> { get_mode.reset(get_mode_8); } }
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) { Right -> { unit_2_7 = Only; } }
    switch(current_mode) { Right -> { bool_2_4 = True; } }
    switch(current_mode) {
      Right -> { switch(bool_2_4) { False -> { ms_2.reset; } } }
    }
    switch(current_mode) {
      Right -> { (int32_2_5, int32_2_6) = ms_2.step(unit_2_7); }
    }
    switch(current_mode) { Left -> { unit_2_3 = Only; } }
    switch(current_mode) { Left -> { bool_2_0 = True; } }
    switch(current_mode) {
      Left -> { switch(bool_2_0) { False -> { ms_5.reset; } } }
    }
    switch(current_mode) {
      Left -> { (int32_2_1, int32_2_2) = ms_5.step(unit_2_3); }
    }
    switch(current_mode) {
      Left -> { m = int32_2_1; }
      Right -> { m = int32_2_5; }
    }
    switch(current_mode) {
      Left -> { s = int32_2_2; }
      Right -> { s = int32_2_6; }
    }
    cc = make_chess_clock (current_mode, m, s);
```

```
  }
}
```

B.3.5 Après fusion des branches (Obc)

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    int32_1_0 : int32;
    --
    time = mem.int32_1_1;
    int32_1_0 = 1+time;
    mem.int32_1_1 = int32_1_0;
  }
}

class get_mode {
  memory {
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    keep : mode;
    switch_ : mode;
    mode_4_0 : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { keep = current_mode;
                 mode_4_0 = keep;
                 mem.mode_4_1 = mode_4_0;
               }
      True -> { switch_ = flip(current_mode);
                mode_4_0 = switch_;
                mem.mode_4_1 = mode_4_0;
              }
    }
    mem.button_1_0 = flip_button;
  }
}

class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
```

```
    bool_3_0 : bool;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    bool_3_0 = (mins==59) && (secs==59);
    mem.reset = bool_3_0;
  }
}

class chess_clock {
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool;
    int32_2_1 : int32;
    int32_2_2 : int32;
    unit_2_3 : unit;
    bool_2_4 : bool;
    int32_2_5 : int32;
    int32_2_6 : int32;
    unit_2_7 : unit;
    --
    bool_1_0 = True;
    switch(bool_1_0) { False -> { get_mode.reset(get_mode_8); } }
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) {
      Right -> { unit_2_7 = Only;
                 bool_2_4 = True;
                 switch(bool_2_4) { False -> { ms_2.reset; } }
                 (int32_2_5, int32_2_6) = ms_2.step(unit_2_7);
                 m = int32_2_5;
                 s = int32_2_6;
               }
      Left -> { unit_2_3 = Only;
                bool_2_0 = True;
                switch(bool_2_0) { False -> { ms_5.reset; } }
                (int32_2_1, int32_2_2) = ms_5.step(unit_2_3);
                m = int32_2_1;
                s = int32_2_2;
              }
    }
    cc = make_chess_clock (current_mode, m, s);
```

```
  }
}
```

B.3.6 Après propagation des variables (Obc)

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    int32_1_0 : int32;
    --
    time = mem.int32_1_1;
    int32_1_0 = 1+time;
    mem.int32_1_1 = 1+time;
  }
}

class get_mode {
  memory {
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    keep : mode;
    switch_ : mode;
    mode_4_0 : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { keep = current_mode;
                 mode_4_0 = current_mode;
                 mem.mode_4_1 = current_mode;
               }
      True -> { switch_ = flip(current_mode);
                mode_4_0 = switch_;
                mem.mode_4_1 = switch_;
              }
    }
    mem.button_1_0 = flip_button;
  }
}

class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
```

```
    bool_3_0 : bool;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    bool_3_0 = (mins==59) && (secs==59);
    mem.reset := (mins==59) && (secs==59);
  }
}

class chess_clock {
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool;
    int32_2_1 : int32;
    int32_2_2 : int32;
    unit_2_3 : unit;
    bool_2_4 : bool;
    int32_2_5 : int32;
    int32_2_6 : int32;
    unit_2_7 : unit;
    --
    bool_1_0 = True;
    switch(True) { False -> { get_mode.reset(get_mode_8); } }
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) {
      Right -> { unit_2_7 = Only;
                 bool_2_4 = True;
                 switch(True) { False -> { ms_2.reset; } }
                 (int32_2_5, int32_2_6) = ms_2.step(Only);
                 m = int32_2_5;
                 s = int32_2_6;
               }
      Left -> { unit_2_3 = Only;
                bool_2_0 = True;
                switch(True) { False -> { ms_5.reset; } }
                (int32_2_1, int32_2_2) = ms_5.step(Only);
                m = int32_2_1;
                s = int32_2_2;
              }
    }
    cc = make_chess_clock (current_mode, m, s);
```

```
  }
}
```

B.3.7 Après élimination de code mort (Obc)

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    int32_1_0 : int32;
    --
    time = mem.int32_1_1;
    int32_1_0 = 1+time;
    mem.int32_1_1 = 1+time;
  }
}

class get_mode {
  memory {
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    keep : mode;
    switch_ : mode;
    mode_4_0 : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { keep = current_mode;
                 mode_4_0 = current_mode;
                 mem.mode_4_1 = current_mode;
               }
      True -> { switch_ = flip(current_mode);
                mode_4_0 = switch_;
                mem.mode_4_1 = switch_;
              }
    }
    mem.button_1_0 = flip_button;
  }
}

class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
```

```
    bool_3_0 : bool;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    bool_3_0 = (mins==59) && (secs==59);
    mem.reset := (mins==59) && (secs==59);
  }
}

class chess_clock {
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
    s : int32;
    bool_1_0 : bool;
    bool_2_0 : bool;
    int32_2_1 : int32;
    int32_2_2 : int32;
    unit_2_3 : unit;
    bool_2_4 : bool;
    int32_2_5 : int32;
    int32_2_6 : int32;
    unit_2_7 : unit;
    --
    bool_1_0 = True;
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) {
      Right -> { unit_2_7 = Only;
                 bool_2_4 = True;
                 (int32_2_5, int32_2_6) = ms_2.step(Only);
                 m = int32_2_5;
                 s = int32_2_6;
               }
      Left -> { unit_2_3 = Only;
                bool_2_0 = True;
                (int32_2_1, int32_2_2) = ms_5.step(Only);
                m = int32_2_1;
                s = int32_2_2;
              }
    }
    cc = make_chess_clock (current_mode, m, s);
  }
}
```

B.3.8 Après élimination de code inutile (Obc)

```
class chrono_base {
  memory {
    int32_1_1 : int32 = 0;
  }
  step (tic:unit) returns (time:int32) {
    --
    time = mem.int32_1_1;
    mem.int32_1_1 = 1+time;
  }
}

class get_mode {
  memory {
    button_1_0 : button = Pushed;
    mode_4_1 : mode = Left;
  }
  step (flip_button:button) returns (current_mode:mode) {
    real_flip : bool;
    switch_ : mode;
    --
    real_flip = trigger(mem.button_1_0, flip_button);
    current_mode = mem.mode_4_1;
    switch(real_flip) {
      False -> { mem.mode_4_1 = current_mode;
                 }
      True -> { switch_ = flip(current_mode);
                mem.mode_4_1 = switch_;
              }
    }
    mem.button_1_0 = flip_button;
  }
}

class ms {
  memory {
    reset : bool = False;
    chrono_base_2 : chrono_base.memory;
  }
  step (tic:unit) returns (mins, secs: int32) {
    int32_1_0 : int32;
    --
    switch(mem.reset) { True -> { chrono_base_2.reset; } }
    (int32_1_0) = chrono_base_2.step(tic);
    mins = int32_1_0/60;
    secs = int32_1_0%60;
    mem.reset = (mins==59) && (secs==59);
  }
}
```

```
class chess_clock {
  memory {
    ms_2 : ms.memory;
    ms_5 : ms.memory;
    get_mode_8 : get_mode.memory;
  }
  step (flip_button:button) returns (cc:chess_clock) {
    current_mode : mode;
    m : int32;
    s : int32;
    int32_2_1 : int32;
    int32_2_2 : int32;
    int32_2_5 : int32;
    int32_2_6 : int32;
    --
    (current_mode) = get_mode_8.step(flip_button);
    switch(current_mode) {
      Right -> { (int32_2_5, int32_2_6) = ms_2.step(Only);
                 m = int32_2_5;
                 s = int32_2_6;
               }
      Left -> { (int32_2_1, int32_2_2) = ms_5.step(Only);
                m = int32_2_1;
                s = int32_2_2;
              }
    }
    cc = make_chess_clock (current_mode, m, s);
  }
}
```

B.4 Code cible (C)

B.4.1 Interface

```
#ifndef __CHRONO_H__
#define __CHRONO_H__

#include <stdint.h>

typedef enum { Only } unit;
typedef enum { True, False} bool;
typedef enum { Left, Right } mode;
typedef enum { Pushed, Released } button;
typedef struct chess_clock* chess_clock;

int trigger (button, button);
mode flip (mode);
void make_chess_clock (mode, int32_t, int32_t, chess_clock);

// CHRONO_BASE
typedef struct {
  int32_t int32_1_1;
} chrono_base_t;
```

```
void chrono_base_reset (chrono_base_t*);
void chrono_base_step (unit, chrono_base_t*, int32_t*);

// GET_MODE
typedef struct {
  button button_1_0;
  mode mode_4_1;
} get_mode_t;
void get_mode_reset (get_mode_t*);
void get_mode_step (button, get_mode_t*, mode*);

// MS
typedef struct {
  int reset;
  chrono_base_t chrono_base_2;
} ms_t;
void ms_reset (ms_t*);
void ms_step (unit, ms_t*, int32_t*, int32_t*);

// CHESS_CLOCK
typedef struct {
  ms_t ms_2;
  ms_t ms_5;
  get_mode_t get_mode_8;
} chess_clock_t;
void chess_clock_reset (chess_clock_t*);
void chess_clock_step (button, chess_clock_t*, chess_clock);

#endif
```

B.4.2 Itération

```
#include "chrono.h"

// CHRONO_BASE
void chrono_base_step (unit tic, chrono_base_t* this, int32_t* time) {
  *time = this->int32_1_1;
  this->int32_1_1 = 1+*time;
}

// GET_MODE
void get_mode_step (button flip_button,
                    get_mode_t* this,
                    mode* current_mode) {
  int real_flip;
  mode switch_;
  real_flip = trigger(this->button_1_0, flip_button);
  *current_mode = this->mode_4_1;
  if(real_flip) {
    switch_ = flip(*current_mode);
    this->mode_4_1 = switch_;
  }
  else {
    this->mode_4_1 = *current_mode;
  }
  this->button_1_0 = flip_button;
```

```
}

// MS
void ms_step (unit tic, ms_t* this, int32_t* mins, int32_t* secs) {
  int32_t int32_1_0;
  if(this->reset) chrono_base_reset(&(this->chrono_base_2));
  chrono_base_step(Only, &(this->chrono_base_2), &int32_1_0);
  *mins = int32_1_0 / 60;
  *secs = int32_1_0 % 60;
  this->reset = ((*mins==59)&&(*secs==59));
}

// CHESS_CLOCK
void chess_clock_step (button flip_button,
                       chess_clock_t* this,
                       chess_clock cc) {
  mode current_mode;
  int32_t int32_2_1, int32_2_2, int32_2_5, int32_2_6, m, s;
  get_mode_step(flip_button, &(this->get_mode_8), &current_mode);
  switch(current_mode) {
    case Left: ms_step(Only, &(this->ms_5), &int32_2_1, &int32_2_2);
               m = int32_2_1;
               s = int32_2_2;
               break;
    case Right: ms_step(Only, &(this->ms_2), &int32_2_5, &int32_2_6);
               m = int32_2_5;
               s = int32_2_6;
               break;
  }
  make_chess_clock(current_mode, m, s, cc);
}
```

B.4.3 Initialisation

```
#include "chrono.h"

// CHRONO_BASE
void chrono_base_reset (chrono_base_t* this) {
  this->int32_1_1 = 0;
}

// GET_MODE
void get_mode_reset (get_mode_t* this) {
  this->button_1_0 = Pushed;
  this->mode_4_1 = Left;
}

// MS
void ms_reset (ms_t* this) {
  this->reset = 0;
  chrono_base_reset(&(this->chrono_base_2));
}

// CHESS_CLOCK
void chess_clock_reset (chess_clock_t* this) {
  ms_reset(&(this->ms_2));
```

```
  ms_reset(&(this->ms_5));
  get_mode_reset(&(this->get_mode_8));
}
```

B.5 Exemple d'utilisation (C)

```
#include <unistd.h>

#include "chrono.h"

/* TYPES ABSTRAITS */

struct chess_clock {
  int flag;
  char display[4];
};

int trigger (button a, button b) {
  return (a==Released && b==Pushed);
}

mode flip (mode m) {
  if(m==Left) return Right;
  return Left;
}

void two_digits (int i, char* buffer, int j) {
  buffer[j] = (i/10)+'0';
  buffer[j+1] = (i%10)+'0';
  return;
}

void make_chess_clock (mode m, int32_t min, int32_t sec, chess_clock c) {
  int m2 = 19 - min;
  int s2 = 59 - sec;
  c->flag=m;
  if (m2<=0 && s2<=0) {
    c->display[0] = 'L';
    c->display[1] = 'O';
    c->display[2] = 'S';
    c->display[3] = 'T';
    return;
  }
  two_digits(m2, c->display, 0);
  two_digits(s2, c->display, 2);
  return;
}

/* Affichage de l'horloge sur le terminal */
char texte[13] = "XX:XX␣XX:XX\n";

void display (chess_clock c) {
  int shift;
  if(c->flag==Left) shift=0; else shift=6;
  texte[0+shift]=c->display[0];
```

```
  texte[1+shift]=c->display[1];
  texte[3+shift]=c->display[2];
  texte[4+shift]=c->display[3];
  write(1, texte, 12);
  return;
}

int main (int argc, char** argv) {
  char c;
  int end = 0;
  button b = Pushed;
  chess_clock_t instance;
  struct chess_clock cc;
  char msg[5*40] =
  "Taille␣de␣chess_clock_t␣=␣XX␣octets␣␣␣\n"
  "Commandes␣:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
  "P/p␣:␣appuie␣sur␣le␣bouton␣␣␣␣␣␣␣␣␣␣␣␣\n"
  "R/r␣:␣relache␣le␣bouton␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
  "q/Q␣:␣quitte␣la␣simulation␣␣␣␣␣␣␣␣␣␣␣␣\n"
  ;
  two_digits(sizeof(chess_clock_t), msg, 26);
  write(1, msg, 5*40);

  /* INITIALISATION */

  chess_clock_reset(&instance);

  /* BOUCLE */

  while(!end) {
    read(0, &c, 1);
    while(c=='\n') read(0, &c, 1);
    switch(c) {
      case 'P' : b = Pushed; break;
      case 'Q' : end = 1; break;
      case 'R' : b = Released; break;
      case 'p' : b = Pushed; break;
      case 'q' : end = 1; break;
      case 'r' : b = Released; break;
    }
    chess_clock_step(b, &instance, &cc);
    display(&cc);
  }

  return 0;
}
```

Annexe C

Références pour le développement en Coq

C.1 `common` : axiomes et fonctions générales (env. 2000 lignes)

Le dossier common du développement contient les fonctions utilitaires et tactiques utilisées dans le développement Coq.

C.1.1 `Axioms.v` (14 lignes)

Trois axiomes ont été utilisés dans le développement. Il s'agit des axiomes :

- d'extensionnalité fonctionnelle, c'est-à-dire affirmer que si deux fonctions f et g ont la même valuation ($\forall x. f(x) = g(x)$), alors elles sont égales.
  Par exemple, si on définit un environnement sémantique comme étant une application `Identificateur->option Valeur`, les environnements, et que l'on suppose deux identificateurs `x` et `y` distincts de valeurs associées `vx` et `vy` et une fonction d'égalité booléenne sur les identificateurs, les environnements
  ```
  fun e => if e=x then Some vx else if e=y then Some vy else None
  ```
  et
  ```
  fun e => if e=y then Some vy else if e=x then Some vx else None
  ```
  sont considérés différents en Coq. Ajouter l'extensionnalité fonctionnelle permet de les considérer comme étant égaux, ce qui simplifie la manipulation d'hypothèses d'équivalences.
- d'indifférence aux preuves, c'est-à-dire que deux preuves d'un même énoncé sont toujours égales.
  Cet axiome est essentiellement utile pour l'utilisation de sous types. Typiquement, si on considère l'ensemble des entiers strictement supérieurs à 1, `{ x : nat | x > 1 }`. Les habitants de ce type sont des paires (n, π) où n est l'entier considéré et π une preuve de $n > 1$. Soient maintenant deux preuves π_1 et π_2 de $n > 1$. Habituellement en mathématiques, π_1 et π_1 ne sont pas considérés, et on n'écrit jamais $f(n, \pi_1)$ ou $f(n, \pi_2)$, mais simplement $f(n)$, et bien sûr on s'attend à ce que $f(n, \pi_1) = f(n, \pi_2)$. La logique de Coq n'autorise cependant pas ce raisonnement. Si f est une fonction

arbitraire, $f(n, \pi_1) = f(n, \pi_2)$ n'est démontrable que si on sait démontrer $\pi_1 = \pi_2$. Dans certains cas une telle preuve est constructible, mais pas dans d'autres. Par ailleurs le système de typage de Coq implique que le résultat d'une fonction définie en Coq ne dépend pas du contenu de ses preuves [1]. L'axiome d'indifférence aux preuves est une façon de prendre en compte cette garantie de typage, ici en énonçant que π_1 et π_2 sont égaux, de même que n'importe quelle autre preuve de $n > 1$. En admettant cet axiome, on montre alors très facilement que $f(n, \pi_1) = f(n, \pi_2)$.

– d'extensionnalité pour les prédicats, c'est-à-dire deux prédicats P et Q équivalents $(\forall x.P(x) \Leftrightarrow Q(x))$ sont toujours égaux, ce qui simplifie la manipulation d'hypothèses d'équivalences.
  Une façon naturelle de représenter un ensemble en Coq et de donner une proposition qui caractérise ses éléments. Typiquement, une représentation des entiers strictement supérieurs à 1 est donnée par `fun(n:nat)=>n>1`. Là encore, il y a de multiples façons de représenter cet ensemble, une alternative est `fun(n:nat)=>n=0->False`. Si on veut traduire l'un en l'autre, on a besoin d'une preuve d'équivalence et de l' axiome d'extensionnalité des prédicats.

Ces axiomes ne sont pas connus pour être incohérents avec la logique de Coq. Il faut cependant bien veiller à n'utiliser ces axiomes que dans les preuves, afin que le mécanisme d'extraction n'essaie pas de les extraire. Par défaut [2], l'extracteur remplace le code correspondant à l'axiome extrait par `assert false`, c'est-à-dire que le programme est interrompu par une erreur au moment de l'exécution.

C.1.2 `CMapPositive.v` (1367 lignes)

Dans un compilateur, les environnements sont très sollicités. Il faut avoir une représentation qui soit à la fois efficace, et dont on puisse prouver facilement des propriétés. Le module `Map` de la bibliothèque standard est une option intéressante pour cette tâche, ce module provient d'ailleurs du projet CompCert. Cependant, son utilisation est assez lourde. C'est pourquoi ce module n'est pas utilisé dans la sémantique. Dans la sémantique, une simple fonction a été préférée pour simplifier les preuves et également parce qu'il s'agit là d'une modélisation où l'efficacité n'est pas de mise (la sémantique n'est pas soumise à l'extraction).

Pour ce qui est de la gestion du compilateur, en revanche, une légère variante du module `Map` de CompCert a été utilisée. Cette variante permet une représentation canonique des tables d'association, c'est-à-dire qu'étant données deux tables d'association T_1 et T_2, si ces deux tables contiennent les mêmes associations, alors elles sont égales (et pas seulement équivalentes).

Ce module introduit la notation `M[<k>]==v` pour signifier que dans la table `M`, la valeur `v` est associée à la clé `k`.

1. Mais rien n'est garanti pour des fonctions déclarées mais non définies dans Coq, comme les axiomes.
2. On peut toujours demander à l'extracteur de remplacer n'importe quelle définition donnée par du code Ocaml.

C.1.3 Coqlib.v (612 lignes)

Dans le développement, il y a des preuves et des programmes. Coq ne dispose pas de fonctionnalités courantes des langages de programmation traditionnels, tels que les exceptions ou les tests d'égalité. L'absence d'effets de bords amène à utiliser les mêmes techniques qu'en Haskell.

Dans le fichier `Coqlib.v`, une sorte de monade a été écrite afin de gérer les cas d'erreurs. Cette monade est un peu plus générale que celle utilisée dans le projet CompCert (fichier `Errors.v`), puisque les erreurs retournées sont paramétrées par un type. Étant donné un type `Ok` des valeurs que l'on veut retourner, et un type `Ko` des erreurs que l'on peut retourner, le type `result Ok Ko` est le coproduit de ces deux types. La notation `[Ok!Ko]` a été introduite pour désigner ces valeurs. On trouve comme dans Haskell et CompCert les notations :

- `operation >>= suite`
  exécute `operation`, si le résultat est une erreur, cette erreur est retournée ; sinon retourne `suite x` où `x` est le résultat précédemment calculé.
- `do x ↓ operation ; suite`
  effectue l'opération `operation`, si cette opération retourne une erreur, la même erreur est retournée, sinon `x` prend la valeur calculée dans l'exécution de `suite`. C'est un raccourci pour `operation >>= (fun x => suite x)`.
- `ret x`
  retourne la valeur `x`.
- `fail x`
  retourne l'erreur `x`.

On y trouve également le type des spécifications. `{ x st P x }`, c'est une redéfinition du type `{ x | P x }` qui utilise les notations suivantes :

- `{: x :}`
  désigne un élément `x` avec une preuve cachée. L'intérêt de cette notation est de pouvoir écrire du code bien typé en omettant les preuves, le système les demandant plus loin. Typiquement, si on veut écrire une fonction prédécesseur avec sa spécification, on pourra écrire :

```
Definition pred (n : nat) : [{ m st S m = n}!n=0].
refine (
  match n with
  | 0 => fail _ (*n=0*)
  | S m => ret {:m (*(S m)=n*):}
  end
).
Proof. (*n=0*)
  reflexivity.
Proof. (*(S m)=n*)
  reflexivity.
Defined.
```

- `{> h <}`
  où `h` est une preuve de `P x` pour un certain `x`, désigne le coupe (x, h) dans `{x st P x}`.
- `ν p`

qui retourne la valeur d'un couple valeur-spécification. En particulier, on a les égalités suivantes :
- ν {: x :} = x ;
- ν {>h<} = x, si h est une preuve de P x ;
- {:ν p:} = p, en admettant l'indifférence aux preuves.

– π p
est la preuve h de la paire (x, h) dénotée par p. En particulier, on a les égalités suivantes :
- π {: x :} = h, si h est une preuve de P x et en admettant l'indifférence aux preuves ;
- π {>h<} = h ;
- {>π p<} = p.

Enfin, ce fichier contient quelques opérations utiles sur les tables d'associations, afin de gérer les collisions.

C.1.4 Autres fichiers

`Equality.v` Ce fichier décrit l'égalité vérifiable (`forall x y, {x=y} + {x=y->False}`). Contrairement à de nombreux langages de programmation, il n'y a pas de fonction d'égalité définie pour tous les types en Coq. Comme en Haskell, on définit une classe de type pour pouvoir appeler la bonne fonction d'égalité chaque fois que l'on souhaite en utiliser une. Dans le code, on note `x =d y` pour l'application de la bonne fonction d'égalité sur `x` et `y`. Une tactique de simplification de ces égalités est également présente de ce fichier (`eq_simpl`).

`tactics.v` Ce fichier répertorie la quasi-totalité des tactiques utilisées dans le développement.

`MoreList.v` Ce fichier contient quelques fonctions sur les listes qui manquaient à la bibliothèque standard au moment où le code était écrit.

C.2 `minils` : les structures syntaxiques (env. 1200 lignes)

C.2.1 `Utils.v`, `Clocks.v`, `Consts.v`, `Types.v`, `Operators.v` (322 lignes)

`Utils.v` Ce fichier définit les identificateurs (en réalité de simples entiers binaires) et exporte la bibliothèque `Coqlib`.

`Types.v` Ce fichier définit les types utilisés. Un type a deux composantes : un élément de $\omega + 1$ (c'est-à-dire un entier fini ou l'infini) appelé son cardinal, et un entier qui permet de distinguer les types de même cardinal. Le type unité (`unit_type`) y est défini, et c'est le seul type considéré natif (il est nécessaire pour donner un type au flot de base). Les types abstraits sont ceux de cardinal infini.

`Consts.v` Ce fichier définit les constantes utilisées. Une constante est définie par un type et un index inférieur au cardinal du type. En OCaml, il faut passer par la fonction `mk_const` pour créer une constante. En effet, la version extraite du type perd l'information de l'index inférieur au cardinal, ce qui a pour conséquence qu'il est possible de créer (côté OCaml) des constantes inconsistantes. La fonction `mk_const` n'a pas ce défaut, car elle a été définie en Coq et les types de ses arguments sont extraits à l'identique. La constante `Only` y est définie (`iTT`).

`Clocks.v` Ce fichier définit les horloges (un identificateur et une constante). L'horloge de base y est définie (`base_clock`).

`Operator.v` Ce fichier définit les opérateurs formels. Un opérateur formel est la donnée d'un identificateur, d'un type de sortie, et d'une liste non vide de types pour les entrées.

C.2.2 `Vardec.v`, `Static.v` (163 lignes)

`Vardec.v` Ce fichier définit les déclarations de flots/variables. Une déclaration est la donnée d'un identificateur, d'un type et d'une horloge.

`Static.v` Ce fichier définit les environnements de typage utilisés pour les sémantiques. Ces environnements sont assez complexes et ne sont pas utilisés par le compilateur lui même. Le compilateur possède sa propre représentation de l'environnement qui est moins contrainte et dont l'implémentation est plus efficace (mais moins adaptée aux preuves).

- Un environnement de nœuds `NEnvironment` est un ensemble `is_a_node` d'identificateurs (une fonction de `ident` dans `Prop`) et une fonction `sig_of_node` de ces identificateurs vers des paires de listes de types (les signatures).
- Un environnement local `TEnvironment` est un ensemble `dom` d'identificateurs, une application `typ` qui à chaque identificateur dans `dom` associe un type et une application `vck` qui à chaque identificateur dans `dom` associe une horloge. En plus de ces fonctions, cet environnement véhicule une preuve d'existence d'un flot/variable (la `base`), et pour chaque flot/variable v, son horloge est dans `dom` et de même type que sa constante associée.
- Un environnement instantané qui complète un environnement local en définissant un ensemble `mem` comme sous domaine de `dom`, une fonction `mem_next` qui à tout élément de `mem` associe un élément de `dom` (sa valeur future), ainsi qu'un ensemble des instances `inst` et une application `class` de ces instances dans `is_a_node`. Une application `inst_reset` de `inst` dans `dom`, une application `inst_reset_const` de `inst` dans les constantes de même type que `inst_reset`, et d'une application de `inst` dans les listes d'éléments de `dom` qui sont compatibles avec le type des entrées de `class` et sur la même horloge que `inst_reset`.

C.2.3 Autres fichiers

`LS.v`, `LSN.v`, `LSNI.v`, `OBC.v` **(514 lignes)** Chacun de ces fichiers définit la syntaxe du langage correspondant.

`LSPervarsives.v` **(212 lignes)** Ce fichier déclare des types standards (booléens, entier de 8, 16, 32 et 64 bits, flottants de 32 et 64 bits) ainsi que des primitives (addition, soustraction, ...) sur ces types. Ce fichier (ou plutôt sa version extraite) est utilisé pour l'analyse syntaxique des programmes. La sémantique de ces primitives est inconnue du compilateur, et ne peut donc faire aucune optimisation les exploitant.

C.3 `ssemantics` : sémantique synchrone (env. 3100 lignes)

C.3.1 `SynchronousStreams.v`, `Primitives.v`, `DFPrimitives.v`, `DFPrimitivesFunctionnal.v` (1119 lignes)

`SynchronousStreams.v` Ce fichier décrit les flots utilisés comme des listes (construites « par la droite »). Il introduit les notations :

- `flot >> valeur` pour le flot `flot` auquel on a ajouté la valeur `valeur` en dernière position ;
- [> pour le flot vide ;
- `flot1 <= flot2` pour signifier que `flot1` est préfixe de `flot2`.

`Primitives.v` Ce fichier définit le type des valeurs (une constante pour un type concret, ou une référence pour un type abstrait), la notion d'absence, le typage des valeurs, et la notion d'opérateur. Il introduit les notations :

- `abs` pour les valeurs absentes ;
- `x` $\downarrow$ pour la valeur présente `x`.

`DFPrimitives.v` Ce fichier définit les primitives flot de données. Il introduit toutes les notations en haut de casse utilisées par la suite.

`DFPrimitivesFunctionnal.v` Ce fichier contient les preuves de l'aspect fonctionnel des primitives flot de données. En effet ces primitives ont été définies comme des relations, afin de pouvoir exprimer des fonctions partielles. De plus les preuves sont simplifiées puisqu'une analyse sur la relation permet d'analyser simultanément toutes les entrées et les sorties d'une primitive. En contrepartie, le déterminisme des primitives n'est plus automatique, il faut le démontrer, et c'est le rôle de fichier.

C.3.2 `SSEnv.v`, `SSEnv_ext.v`, `SIEnv.v`, `Dynamic.v` (966 lignes)

`SSEnv.v` Ce fichier décrit l'environnement sémantique local pour les langages Ls et Lsn. Si `H` est un environnement sémantique local dans un environnement de typage local `T` (ie. un `TEnvironment`, voir `Static.v`) et `p` un élément de {`x st dom T x`}, `H`⟦`p`⟧ est le flot

associé à ν `p`. Cet environnement contient aussi une preuve selon laquelle toutes les valeurs présentes dans `H⟦p⟧` sont de même type que `typ T p`.

`SIEnv.v` Ce fichier décrit l'environnement sémantique local pour les langages Lsni et Obc. Si `R` est un environnement sémantique local dans un environnement de typage local `T` (ie. un `TEnvironment`, voir `Static.v`) et `p` un élément de `{x st dom T x}`, `H{|p|}` est la valeur associé à ν `p`. Cet environnement contient aussi une preuve selon laquelle `H{|p|}` est de même type que `typ T p` (quand elle est présente).

`Dynamic.v` Ce fichier met en relation les deux types d'environnements sémantique locaux. Il permet notamment de définir en particulier de découper un environnement sémantique local flots de données en un autre environnement sémantique local flot de données et un environnement sémantique local instantané (`cutting\_history`) quand on sait que le flot de base de ce premier est différent du flot vide. Il permet également de décrire comment une mémoire peut abstraire un environnement sémantique local flot de données (`last_memory`).

`SSEnv_ext.v` Ce fichier contient les opérations nécessaire à l'extension d'un environnement sémantique local flot de données, utile à la passe d'introduction de nouveaux noms de flots.

C.3.3 `LSSemantics.v`, `LSNSemantics.v`, `LSNISemantics.v`, `ObcSemantics.v`, `Memory.v` (1219 lignes)

Chacun de ces fichier décrit la sémantique synchrone du langage correspondant.

C.4 `ls` : Propriétés et preuves concernant Ls (env. 4100 lignes)

C.4.1 `LSTyped.v`, `LSTypingValidator.v` (1177 lignes)

`LSTyped.v` décrit le bon typage de Ls, alors que `LSTypingValidator.v` contient toutes les procédures opérationnelles visant à vérifier ce bon typage.

C.4.2 `prop/LSTyped_ext.v`, `prop/LSTypingCoherency.v`, `prop/LSTypingUnicity.v`, `prop/LSDeterminism.v` (916 lignes)

`prop/LSTyped_ext.v` Ce fichier montre la préservation du typage par extension de l'environnement de typage local.

`prop/LSTypingCoherency.v` Ce fichier contient la preuve selon laquelle dans tout environnement sémantique local flot de données `H` sur un environnement de typage `T`, si une expression `e` est de type t et d'horloge ck dans `T`, alors tout flot dénoté par cette expression a toutes ses valeurs présentes de type t, et de plus ses absences coïncident avec les instants donnés par l'horloge ck.

`prop/LSTypingUnicity` Ce fichier contient la preuve que toutes les structures considérées ne peuvent admettre qu'un unique type.

`prop/LSDeterminism.v` Ce fichier contient la preuve que dans un environnement sémantique local, une expression ne peut produire qu'au plus une liste de flots possible. Ceci ne prouve cependant pas qu'il n'existe qu'au plus un environnement sémantique local dans lequel un nœud ait une sémantique.

C.4.3 `LSSubst.v`, `LSDist.v`, `LSNorm.v` (308 lignes)

Ces fichiers décrivent les certificats utilisés pour valider les passes de compilation, ainsi que les relations attendues entre ces certificats, le code source et le code produit, pour chacune des passes ayant Ls comme langage source.

C.4.4 `LSDistPreservation.v`, `LSNormPreservation.v` `LSSubstPreservation.v` (1479 lignes)

Ces fichiers sont les preuves de préservation de la sémantique vis-à-vis de la relation correspondante définie dans la sous-section précédente.

C.5 `ls_to_lsn` : compilation de Ls vers Lsn (env. 1000 lignes)

C.5.1 `LSSubst_Validator.v`, `LSDistValidator.v`, `LSNormValidator.v` (788 lignes)

Ces fichiers contiennent les preuves de vérifiabilité des relations, c'est-à-dire les algorithmes qui acceptent les entrées avec une preuve qu'elles vérifient la propriété souhaitée, ou les rejettent sans justification.

C.5.2 `LS2LSN.v` (178 lignes)

C'est le fichier qui résume la partie prouvée avec assistant de preuve de cette thèse. `ls_to_lsn` est la fonction qui étant donnés un nœud et des certificats pour chacune des passes de la compilation vers Lsn, produit un nœud bien typé dans Lsn, dont il est prouvé que sa sémantique est préservée (ou retourne une erreur).

C.6 `lsn` : propriétés et preuves concernant Lsn

C.6.1 `LSNTyped.v` (168 lignes)

Ce fichier décrit les contraintes de typage sur Lsn.

C.6.2 `LSNInst.v` (96 lignes)

Ce fichier décrit la propriété d'instantiation entre un nœud de Lsn et un nœud de Lsni.

C.6.3 `LSNInstPreservation.v` (1185 lignes)

Ce fichier apporte la preuve de préservation de sémantique entre un nœud de Lsn et sa version instanciée.

C.7 Fichiers en Ocaml

L'état actuel du compilateur ne contient pas d'autres passes prouvées. Le validateur de Lsn vers Lsni manque pour pouvoir exploiter la preuve de préservation de la sémantique de Lsn vers Lsni, mais devrait être raisonnablement facile à implémenter.

Aucun des autres certificats dont il est question dans la thèse n'a été formalisé en Coq. La partie certifiée en Coq s'arrête donc à Lsn.

C.7.1 Analyse syntaxique de Ls

`common/mlUtils.ml` **(92 lignes)** Ce fichier permet de traduire les entiers et chaînes de caractères de Ocaml vers Coq.

`parsing/global.ml` **(239 lignes)** Ce fichier ajoute toutes les primitives de `LSPervasives.v` à l'analyseur syntaxique.

`location.ml` **(146 lignes)** Ce fichier permet de localiser les erreurs rapportées par l'analyseur syntaxique. Le compilateur lui même ne garde pas de trace des numéros de ligne, et rapporte les erreurs en donnant le nom du nœud et le numéro de l'équation concernée.

`lexer.mll`, `parser.mly` **(438 lignes)** L'analyseur syntaxique pour le langage Ls.

C.7.2 Affichage des codes produits

À terme, seul du code Obc devrait être produit. Cependant, pour tracer la compilation, des codes intermédiaires sont produits dans un fichier hypertexte (html).

`html_emiters.ml` **(154 lignes)** Quelques fonctions de base pour produire du code Html.

`lsPrinter.ml`, `lsnPrinter.ml`, `lsniPrinter.ml`, `obcPrinter.ml` **(répertoire** `minils`**, 1326 lignes)** Fichiers permettant d'afficher du code dans chacun des langages, afin de pouvoir suivre la compilation.

C.7.3 Générateurs de certificats

Le fichier `ls_to_lsn/normalize.ml` contient deux générateurs de certificats pour l'introduction de nouveaux noms de flots, `Normalizer` et `AggressiveNormalizer`. Le premier introduit le moins de nouveaux noms possibles, alors que le second découpe toutes les expressions en expressions élémentaires. `Normalizer` ne remplit cependant pas très bien sa fonction, et produit parfois du code rejeté par le compilateur. Dans la version actuelle, c'est donc `AggressiveNormalizer` qui est utilisé (voir le fichier `main/compiler.ml`) et qui semble moins souvent faire échouer la compilation.

C.7.4 Le compilateur

`main/misc.ml` **(81 lignes)** Ce fichier donne les fonctions pour rapporter les erreurs du compilateur.

`compiler.ml` **(115 lignes)** Ce fichier décrit la chaîne de compilation, en agençant les différentes passes et en appelant le code Coq extrait.

`main.ml` **(51 lignes)** Ce fichier appelle le compilateur avec les paramètres passés en ligne de commande.

Annexe D

Justifications de quelques choix d'implémentation

Plusieurs choix ont été faits dans les développements en Coq de cette thèse. Coq est un langage très riche, et pour un problème donné, de nombreuses approches sont possibles ; dans cette partie, je présente les raisons qui m'ont amené à faire certains choix critiquables.

D.0.5 La relation de substitution

La première passe de compilation consiste à vérifier qu'en substituant de noms de flots frais par des expressions, on obtient le code d'origine. Dans ce cas pourquoi ne pas définir simplement la substitution de nom de flot ?

La réponse est que dans le code Coq, les applications de nœuds retournent un tuple. Il faut donc être plus général et être capable de substituer un tuple de noms de flots par une expression. Tant qu'à exprimer cette généralisation, autant être le plus général et utiliser la substitution d'une expression quelconque par une autre.

De plus, cette version plus générale pourrait être utilisée pour introduire des optimisations. Par exemple, si on montre que `(x+y) when t` et `(x when t) + (y when t)` ont même sémantique, on peut essayer de remplacer toutes les occurrences de la première expression par la seconde qui peut être plus efficace, selon la façon dont le code est compilé.

D.0.6 Ajout de valeurs dans un flot

Traditionnellement, les flots se construisent en ajoutant des valeurs en première position, ce qui correspond à la définition suivante :

```
Inductive flot1 : Set :=
| Fin1 : flot1
| Cons1 : forall (v : valeur) (f : flot1), flot1
.
```

Cependant plusieurs définitions inductives sont plus naturelles en se donnant la possibilité d'ajouter des éléments à la fin du flot.

Par exemple, il est plus naturel de dire que x est préfixe de y en disant qu'on ajoute des valeurs à la fin de x pour obtenir y, qu'en disant qu'en prenant une liste l et en lui ajoutant au début toutes les valeurs de x, on obtient y.

De la même façon, quand on relève une sémantique instantanée en une sémantique flot de données, on veut avoir un accès direct à la dernière mémoire calculée depuis la mémoire initiale, ce que ne permet pas la représentation traditionnelle de l'historique, où il faut chercher cette mémoire tout au fond de cet historique.

C'est pourquoi, la définition des flots utilisés correspond à la suivante :

```
Inductive flot2 : Set :=
| Fin2 : flot2
| Cons2 : forall (f : flot2) (v : valeur), flot2
.
```

Les deux types sont parfaitement isomorphes, mais dans le second type, la dernière valeur du flot est en accès direct.

Par exemple, les deux flots suivants codent la même suite de valeurs, à savoir, la suite finie $0, 1, 2, 3, 4$:

```
Let f1 : flot1 := Cons1 0 (Cons1 1 (Cons1 2 (Cons1 3 (Cons1 4 Fin1)))).
Let f2 : flot2 := Cons2 (Cons2 (Cons2 (Cons2 (Cons2 Fin2 0) 1) 2) 3) 4.
```

Afin de ne pas avoir à définir une opération d'ajout d'une valeur par la droite, et comme l'opérateur de concaténation ($\circ$) était déjà défini, j'ai préféré utiliser $flot \circ [valeur]$, ce qui définit bien le même objet, quelque soit la convention sur les flots utilisée, mais qui s'implémente facilement avec la seconde définition des flots.

D.0.7 Fonctions et prédicats

Un prédicat est une relation entre plusieurs objets. Toute fonction peut s'exprimer en coq comme une relation entre des entrées et des sorties.

Les avantages des fonctions sur les prédicats relèvent essentiellement de la possibilité de les utiliser pour calculer.

Les avantages des prédicats sur les fonctions relèvent des considérations suivantes :

- les prédicats sont souvent plus simples à écrire et à lire, surtout lorsque certaines relations sont attendues entre les entrées
- les prédicats peuvent être avantageusement utilisés pour décrire des fonctions partielles, c'est-à-dire non définies sur la totalité de leur domaine. Les fonctions définies dans Coq de U dans T doivent, comme au sens mathématique, être définies totalement. Ceci oblige à :
  - soit retourner une valeur par défaut dans T dans les cas non définis
  - soit gérer des erreurs dans les sorties en retournant une valeur de type $T + erreur$ au lieu d'une valeur de type T
  - soit spécifier le type d'entrée pour éliminer les cas mal définis et donc utiliser une entrée de type $u|Pu$ au lieu de U

  Souvent, aucune de ces trois solutions n'est vraiment satisfaisante. La première solution a pour défaut d'être silencieuse, localiser une erreur peut devenir ardu. La

seconde solution a pour défaut de perdre la composabilité ; on est alors obligé d'user d'un style monadique souvent un peu lourd. La troisième solution a pour défaut d'obliger à systématiquement prouver que l'entrée est bien dans le bon domaine.
- il est souvent plus aisé d'inverser un prédicat qu'une fonction. De même qu'en PROLOG, un prédicat peut être tout aussi bien utiliser pour une fonction que son inverse, en COQ, on peut parfois obtenir directement des informations sur l'entrée à partir de la sortie

illustration du premier point

```
Inductive trois := Un | Deux | Trois.
Definition trois_eq (a b : trois) :=
  match a, b with
  | Un, Un
  | Deux, Deux
  | Trois, Trois => true
  | _, _ => false
  end.
Inductive trois_rel (a : trois) : trois -> bool -> Prop :=
  | teq :                        trois_rel a a true
  | tneq : forall b, b <> a -> trois_rel a b false.
```

Alors que la relation d'égalité est triviale à exprimer comme prédicat, écrire une fonction qui la décide est souvent long et fastidieux ; au point qu'il existe une tactique en COQ pour essayer de le faire automatiquement [1].

illustration du second point

```
Definition pred1 (m : nat) : nat :=
  match m with
  | O => O (* valeur par defaut *)
  | S m => m
  end.

Definition pred2 (m : nat) : option nat :=
  match m with
  | O => None
  | S m => Some m
  end.

Definition pred3 (m : {n | n > 0}) : nat.
refine(
  match projT1 m as n return n > 0 -> nat with
  | O => _ (* cas absurde *)
  | S m => fun _ => m
  end (projT2 m)
).
Proof. (* cas absurde *)
  intros H.
```

1. Cette tactique est `decide equality`.

```
  assert (absurd : False); [inversion H|].
  elim absurd.
Defined.

Inductive pred_rel : nat -> nat -> Prop :=
  Succ_rel_intro : forall x, pred_rel (S x) x.
```

Dans cet exemple, le prédicat ne définit pas ce qui se passe lorsque $x = 0$; ce cas est donc simplement exclu, sans traitement supplémentaire.

illustration du troisième point

```
Definition map2 {A B C : Type} (f : A -> B -> C)
: list A -> list B -> list C :=
  fix map2 l m :=
    match l, m with
    | nil, nil => nil
    | cons hl tl, cons hm tm => cons (f hl hm) (map2 tl tm)
    | _, _ => nil
    end.

Inductive map2_rel {A B C : Type} (f : A -> B -> C)
: list A -> list B -> list C -> Prop :=
| map2nil : map2_rel f nil nil nil
| map2cons : forall hda hdb tla tlb l,
             map2_rel f tla tlb l ->
             map2_rel f (cons hda tla)
                        (cons hdb tlb)
                        (cons (f hda hdb) l).
```

Dans cet exemple, on s'intéresse à des combinaisons de listes pour obtenir d'autres listes ; et on suppose toutes les listes de même longueurs.

Pour se simplifier la vie, on a choisi de retourner une valeur par défaut dans le cas de fonctions. Dans la version avec prédicat, on voit tout de suite que si le résultat n'est pas la liste vide, alors aucune des listes en entrée n'est vide. Pour connaître les valeurs des têtes de listes en entrée, il suffit d'inverser [2] le prédicat ; dans la version fonctionnelle, il faut raisonner par cas sur chacune des listes et éliminer les trois cas qui ne correspondent pas à la situation voulue.

Choix réalisé

La quasi totalité des fonctions ont été écrites sous forme de prédicats, notamment pour les primitives sur les flots, afin de garantir que tous les flots utilisés soient de même longueur, et aussi faciliter le raisonnement par induction. En effet, avec des fonctions, il faut raisonner par induction sur les entrées, alors qu'avec des prédicats, on peut directement raisonner sur la fonction elle même.

2. avec la tactique `inversion` par exemple

D.0.8 Sémantique avec absences et sémantique de Kahn

Un autre point intéressant dans les choix de la sémantique est l'interprétation des flots.

Dans la thèse, les flots ont été interprétés comme des listes de valeurs présentes ou absentes.

Cette vision s'éloigne du modèle mathématique décrit par Gilles Kahn [33], où les flots sont des suites mathématiques (éventuellement finies), et où il n'y a pas de notion d'absence.

Le modèle de Kahn n'a pas été envisagé ; il est fort probable que certaines parties auraient été simplifiées. Cependant la description de certaines primitives aurait été alourdie ; et la vision avec les présences et les absences rapproche du modèle flot de contrôle, ce qui simplifie les preuves.

Ainsi, la primitive de fusion de deux flots aurait eu besoin d'un flot « maître » pour expliciter comment entrelacer les deux flots ; ce n'est pas nécessaire avec les présences et les absences puisque celles ci indiquent à quels endroits les valeurs doivent être insérées.

En revanche, la sémantique des nœuds s'en trouve compliquée, puisqu'il faut commencer par enlever toutes les absences ; et la sémantique de l'opérateur `fby` aurait été plus simple dans une sémantique de Kahn.

Choix réalisé

Comme dit précédement, le choix ici n'a pas été réellement fait, car j'étais parti sur une sémantique avec présence et absence sans y avoir vraiment réfléchi. Cependant, rétrospectivement, ce choix n'a pas été un mauvais choix. Il y a somme toute assez peu d'endroits où les absences se sont trouvées être pénalisantes. La sémantique avec absence est peut être même d'une certaine façon plus naturelle que celle de Kahn quand on veut comprendre la relation entre le code source et le code compilé ; et il suffit d'effacer les absences pour se retrouver dans une sémantique de Kahn, alors que l'injection d'absences pour passer d'une sémantique de Kahn à une sémantique avec absences est plus délicate.

D.0.9 Inductifs et coinductifs

Coq n'autorise que des fonctions qui terminent. Fabriquer un type de données à valeurs « infinies » peut donc paraître impossible à première vue. Ce n'est toutefois pas le cas, toute fonction de `nat` dans un type donné peut être vue comme une liste infinie de valeurs de ce type.

Il existe même une sorte de types de données appelée types de données coinductifs que Coq peut gérer. Ce type de données ressemble à celui des inductifs, mais alors que la production d'une valeur à partir d'un inductif est soumise à des règles sur la consommation de l'inductif, la consommation d'une valeur pour produire un coinductif est soumise à des règles de production du coinductif.

Ces règles sont décrites dans le manuel de référence de Coq [39]. Les coinductifs sont pénibles à utiliser quand on débute avec Coq du fait de l'impossibilité de réduire les co-

points fixes sans consommer le résultat produit [3]. En pratique, il est courant de démontrer des égalités de réduction afin de pouvoir avancer dans les preuves.

De plus, les preuves par coinduction ne peuvent produire d'inductifs. Démontrer que deux listes infinies sont égales est donc impossible en général, même avec des conditions fortes. Il faut alors raisonner avec des relations d'équivalence, et pour chaque propriété définie, montrer qu'elle est compatible avec la relation d'équivalence définie.

```
(* L'identite *)
Definition id (A : Type) : A -> A := fun a => a.

(* deux foncteurs *)
Inductive list (A : Type) : Type :=
  | Nil : list A
  | Cons : A -> list A -> list A.
CoInductive colist (A : Type) : Type :=
  | CoNil : colist A
  | CoCons : A -> colist A -> colist A.

(* et leur methode *)
Definition fmap {A B : Type} (f : A -> B) : (list A -> list B) :=
  fix fmap l :=
  match l with
  | Nil => Nil B
  | Cons x l => Cons B (f x) (fmap l)
  end.

Definition cofmap {A B : Type} (f : A -> B) : (colist A -> colist B) :=
  cofix cofmap l :=
  match l with
  | CoNil => CoNil B
  | CoCons x l => CoCons B (f x) (cofmap l)
  end.

(* premiere loi sur les foncteurs pour les listes *)
Lemma list_unit : forall A (l : list A), fmap (id A) l = l.
Proof.
  intros A; fix l; intros [|head tail].
    (* |- fmap (id A) (Nil A) = Nil A *)
    simpl.
    (* Nil A = Nil A *)
    reflexivity.
  (* list_unit : forall l <substructure (Cons A head tail)>,
                 fmap (id A) l = l
     |- fmap (id A) (Cons A head tail) = Cons A head tail *)
  generalize (list_unit tail); simpl.
  (* |- fmap (id A) tail = tail ->
        Cons A (id A head) (fmap (id A) tail) = Cons A head tail *)
  intros rw; rewrite rw.
  (* |- Cons A (id A head) tail = Cons A head tail *)
```

3. On dit que la 'subject reduction' n'est pas garantie par les coinductifs

```
  reflexivity.
Qed.

(* premiere loi sur les foncteurs pour les colistes

Lemma colist_unit : forall A (l : colist A), cofmap (id A) l = l.
est indemontrable, de meme que sa negation *)
CoInductive coeq {A : Type} : colist A -> colist A -> Prop :=
| EqNil : coeq (CoNil A) (CoNil A)
| EqCons : forall a l m, coeq l m -> coeq (CoCons A a l) (CoCons A a m)
.

Lemma red {A : Type} : forall l, l = match l with
                                     | CoNil => CoNil A
                                     | CoCons a l => CoCons A a l
                                     end.
Proof.
  intros []; split.
Qed.

Lemma colist_unit : forall A (l : colist A), coeq (cofmap (id A) l) l.
Proof.
  intros A; cofix; intros [|head tail].
    (* |- coeq (cofmap (id A) (CoNil A)) (CoNil A) *)
    simpl. (* ne change rien car [cofmap (id A) (CoNil A)] n'est
              pas consomme. *)
    rewrite (red (cofmap (id A) (CoNil A))); simpl.
    (* |- coeq (CoNil A) (CoNil A) *)
    left.
  (* colist : ...
     |- coeq (cofmap (id A) (CoCons A head tail)) (CoCons A head tail) *)
  rewrite (red (cofmap (id A) (CoCons A head tail)));
  simpl; fold (cofmap (id A)).
  (* colist : ...
     | - coeq (CoCons A (id A head) (cofmap (id A) tail))
              (CoCons A head tail) *)
  right.
  (* colist : ... |- coeq (cofmap (id A) tail) tail *)
  apply colist_unit.
Qed.
```

L'exemple sur les listes illustre les lourdeurs liées aux coinductifs.

Comme l'énoncé `forall x y, coeq x y -> x = y` n'est pas démontrable, chaque fois qu'on a `P x` et `coeq x y`, on ne peut en déduire `P y` ; il faut d'abord prouver que `P` est un morphisme compatible avec la relation `coeq` (ce qui en pratique est toujours le cas).

Enfin remarquons que plusieurs définitions des flots par coinductifs sont possibles et que chacune a un impact sur la sémantique que l'on se donne.

– on peut se donner comme définition de flot celle donnée plus haut (`colist`), alors la sémantique des primitives doit s'exprimer par des relations coinductives. En effet, si les flots utilisent des valeurs présentes ou absentes, on ne peut pas définir de

fonction qui efface les absences, car cela impliquerait la capacité de savoir à l'avance si un flot ne contient que des valeurs absentes. Dans une vision sans absences, c'est l'échantillonnage de flots qui pose problème, car alors il faut être capable de prédire si un flot donné prendra une valeur donnée.
Se donner en axiome un oracle capable de prédire si un flot prend une valeur donnée est envisageable, mais un tel oracle ne peut être implémenté en Coq[4]. En conséquence cette fonction ne pourrait être utilisée que comme une relation, le système Coq ne pouvant pas l'évaluer.

– on également se donner comme définition de flots un définition utilisée par Christine Paulin-mohring [47] et reproduite en figure D.1 où l'accès à la valeur suivante du flot (quand elle existe) n'est pas immédiate.
Cette définition peut paraître très artificielle, mais colle parfaitement avec la vision mathématique si on utilise une version sans les absences.
De plus cette définition permet de définir très naturellement la topologie sous-jacente au système décrit par Gilles Kahn [33].
Cette définition permet aussi de donner une sémantique des primitives par des relations, mais celles-ci deviennent alors inutilement compliquées du fait de l'ajout des retards au sein des flots.

Le passage à des flots coinductifs fait donc perdre beaucoup en flexibilité.

4. On serait en effet capable de décider de l'arrêt d'un programme.

```
CoInductive Flot (T : Type) : Type :=
| Queue : Flot T -> Flot T
| Tete  : T -> Flot T -> Flot T.
Implicit Arguments Queue [T].
Implicit Arguments Tete [T].

(* Subject reduction *)
Definition red {T} (f : Flot T) :=
  match f with Tete t f => Tete t f | Queue f => Queue f end.
Lemma red_eq {T} (f : Flot T) : f = red f.
Proof. destruct f; split. Qed.

(* Exemple: flot vide *)
Definition vide (T : Type) : Flot T := cofix vide := Queue vide.
Lemma vide_eq (T : Type) : vide T = Queue (vide T).
Proof.
  set (x := Queue (vide T)); rewrite (red_eq (vide T)); simpl; split.
Qed.

(* Relation prefixe entre flots *)
  Inductive EnQueue {T} (t : Flot T) : Flot T -> Prop :=
  | EnQueue_intro : EnQueue t (Queue t).
  Inductive Suivant {T} (f : Flot T) : Prop:=
  | Ensuite : (forall {u}, EnQueue u f -> Suivant u) -> Suivant f.
  Definition ensuivant {T U} (f : T -> Flot T -> U)
  : forall {t : Flot T}, Suivant t -> U
  := fix ensuivant {t} H {struct H} :=
     match t as f return (forall u, EnQueue _ f -> _) -> _ with
     | Queue q => fun H => ensuivant (H q (EnQueue_intro _))
     | Tete t q => fun _ => f t q
     end (let (H) := H in H).
  Definition queue {T t} := @ensuivant T (Flot T) (fun _ q => q) t.
  Definition tete {T t} := @ensuivant T T (fun t _ => t) t.
CoInductive Prefixe {T} : Flot T -> Flot T -> Prop :=
| PQ : forall {f1 f2}, Prefixe f1 f2 -> Prefixe (Queue f1) f2
| PT : forall {f1 f2} (s : Suivant f2),
       Prefixe f1 (queue s) -> Prefixe (Tete (tete s) f1) f2.

(* Approximation finie (Approx u v => Prefixe u v) *)
Inductive App {T} : Flot T -> Flot T -> Prop :=
| App_bottom : forall t, App (vide T) t
| App_cons : forall u v (s : Suivant u),
             App v (queue s) -> App (Tete (tete s) v) u.

(* Continuite *)
Record ouvert {T} (P : Flot T -> Prop) :=
{ monotony : forall u v, Prefixe u v -> P u -> P v
; scott : forall u, P u -> exists a, App a u /\ P a }.
Definition continue {A B} (f : Flot A -> Flot B) :=
  forall (P : Flot B -> Prop), ouvert P -> ouvert (fun a => P (f a)).
```

Figure D.1: Définition coinductive de flots en vue de définitions constructive d'opérateurs

Index

Liste des symboles