# Modèles de programmation de haut niveau pour microcontrôleurs à faibles ressources

Steven Varoumas

Sorbonne Université

EDITE de Paris (ED 130)

---

# Modèles de programmation de haut niveau pour microcontrôleurs à faibles ressources

---

Par Steven Varoumas

Thèse de doctorat en Informatique

Dirigée par Tristan Crolard

Présentée et soutenue publiquement à Paris le 5 novembre 2019

Devant un jury composé de

| | | | |
|---|---|---|---|
| Claire Pagetti | rapporteure | Maître de Recherche | ONERA |
| Alan Schmitt | rapporteur | Directeur de Recherche | Inria Rennes |
| Carlos Agon | président | Professeur | Sorbonne Université |
| Timothy Bourke | examinateur | Chargé de Recherche | Inria Paris |
| Mihaela Sighireanu | examinatrice | Maître de Conférences | Université Paris Diderot |
| Tristan Crolard | directeur | Professeur | Cnam |
| Emmanuel Chailloux | encadrant | Professeur | Sorbonne Université |
| Philippe Trébuchet | encadrant | Maître de Conférences | Sorbonne Université / ANSSI |

**Remerciements**

Je remercie en premier lieu les rapporteurs de cette thèse, Claire Pagetti et Alan Schmitt, pour avoir accepté de sacrifier un peu de leur temps (et peut-être même une partie de leurs vacances d'été) à la relecture de mon manuscrit, ainsi que pour leurs remarques et retours constructifs. Je tiens de la même façon à remercier Mihaela Sighireanu, Carlos Agon et Timothy Bourke, pour avoir accepté de faire partie de mon jury, et d'évaluer ainsi mon travail.

Je remercie chaleureusement Tristan, mon directeur de thèse, pour son extrême patience, sa grande disponibilité, et sa vaste expertise scientifique. Je réalise ma chance d'avoir pu travailler avec quelqu'un d'humainement et professionnellement irréprochable. Je veux aussi remercier mes encadrants, Emmanuel et Philippe, pour la confiance qu'ils m'ont accordée, leurs conseils avisés, aisi que la bienveillance et l'humour dont ils ont toujours fait preuve. Sans le soutien de mon directeur et mes encadrants, j'aurais sans doute « explosé en plein vol », selon une formule chère à l'un d'entre eux.

Un grand merci aussi à l'équipe APR, chez qui je me sens pratiquement chez moi depuis tant d'années, mais qu'il va maintenant falloir quitter. Merci en particulier à ceux qui m'ont fait confiance et/ou m'ont conseillé pour mener à bien toutes ces heures d'enseignements. Parmi ceux non préalablement cités : Annick, Pascal, Frédéric, Romain, Antoine, Tong. J'ai une petite pensée d'ailleurs pour tous les étudiants qui n'ont jamais cessé de me surprendre (en bien comme en mal !) et généralement de beaucoup m'amuser. Merci aux collègues doctorants et post-doctorants, anciens et actuels, qui ont pimenté la vie de laboratoire (et parfois de bar), et sans qui cette aventure aurait été bien triste. Par ordre pseudo-aléatoire, je pense à Matthieu (D), Ghiles, Clément, Martin, Raphaël, Matthieu (J), Jules, Boubacar, Yi-Ting, Rémy, Vincent, Abdelraouf, Christelle, Adilla, et j'en oublie forcément. Merci aussi à notre super gestionnaire Thuy pour sa réactivité et sa bonne humeur sans faille.

Merci aux amis de l'extérieur, qui m'ont souvent permis de faire retomber la pression, sans parfois même s'en rendre compte. Merci pour leur écoute, leurs conseils, et leurs blagues ! Sans pouvoir tous les nommer, je remercie Anna, Benoît, Marwan, Pierre, et Alice.

Bien sûr, je tiens aussi à remercier ma sœur et mes parents, qui ont toujours bien plus cru en moi que je n'ai été en mesure de le faire moi-même. Merci pour leur soutien permanent et leur aide, dans les épreuves passées comme futures.

Enfin, merci à Adrien, dont la confiance et le soutien indéfectibles ont été le carburant de toutes ces années. Sa gentillesse, sa présence et son courage sont mes principales sources d'inspiration, qui me poussent chaque jour à devenir meilleur.

Qu'on se le dise !
S.

# Table des matières

# Introduction

Le lecteur de ce document, comme tout un chacun, interagit avec une quantité impressionnante de microcontrôleurs au cours d'une journée typique de son quotidien. Ces appareils électroniques, de petite taille (quelques millimètres à quelques centimètres), se nichent en effet dans de nombreux objets que nous utilisons chaque jour, et régissent discrètement de nombreux aspects de nos vies. Que le lecteur ait pris les transports en commun pour se rendre sur son lieu de travail, ou plutôt sa voiture personnelle, des centaines de microcontrôleurs l'ont alors accompagné dans son trajet, afin de réaliser de nombreuses tâches automatiques, comme le contrôle du système de freinage du véhicule, ou l'actionnement des portes automatiques des quais du métro. Leur présence ne se limite pas aux moyens de transport : par exemple, au travail, la machine distribuant le café, le système de climatisation, ou même les périphériques des ordinateurs (écrans, souris, clavier, . . . ) en contiennent potentiellement des dizaines, chargés du bon fonctionnement de ces différents systèmes. Les appareils de la maison ne sont pas en reste, puisque le petit et gros électroménager, comme les machines à laver, les réfrigérateurs, ou les fours à micro-ondes, en abrite une multitude. En réalité, presque toutes les pièces d'une maison sont concernées : les radio-réveils, radiateurs, thermostats, systèmes d'éclairage, chauffe-eaux, et même certains jouets peuvent en contenir. Certains microcontrôleurs peuvent même être retrouvés sur soi, dans le téléphone mobile [1] rangé dans sa poche, ou à l'intérieur de certains appareils médicaux. De par leur omniprésence, les microcontrôleurs contribuent ainsi quotidiennement aux diverses tâches de nos vies modernes.

Vu de l'intérieur, un microcontrôleur (parfois abrégé en $\mu C$) est un circuit imprimé programmable dont les divers éléments sont équivalents aux composants d'un ordinateur très simplifié, mais néanmoins complet. En effet, un microcontrôleur possède une unité de calcul arithmétique et logique, un ensemble de mémoires, généralement composé d'une mémoire vive qui permet de traiter les données dynamiques d'un programme et de mémoires non volatiles destinées à stocker le programme ou certaines données, ainsi que plusieurs dispositifs d'entrées/sorties, appelés *broches* ou *pins* : des petites pattes métalliques capables de porter un courant électrique afin de communiquer avec son environnement (représentées dans la figure 1). L'environnement d'un microcontrôleur est typiquement un montage électronique constitué de multiples composants qui permettent d'intéragir avec le monde réel : des capteurs qui communiquent au microcontrôleur des propriétés au sujet de l'environnement physique du montage (valeurs de température, luminosité, état d'un bouton-poussoir, . . . ), et des actionneurs, qui influencent l'état du système physique par leur déclenchement (allumage d'une diode électroluminescente, rotation d'un moteur, émission de son ou de chaleur, . . . ). Un microcontrôleur constitue ainsi souvent l'unité centrale de calcul d'un montage (ou d'une partie d'un montage lorsqu'il est question de réseaux de circuits plus complexes) électronique : il se charge de coordonner les réactions du matériel à l'état de l'environnement physique qui l'entoure. De ce fait, de tels appareils sont fréquemment utilisés afin de

1. Bien que les téléphones portables récents contiennent des systèmes sur puce plus puissants et plus complexes, il n'est pas rare que des microcontrôleurs soient chargés de tâches auxiliaires comme le contrôle des écrans tactiles ou de certains capteurs.

programmer des systèmes embarqués, et régissent l'automatisation des tâches propres à ces systèmes (par exemple, la régulation thermique d'une pièce).

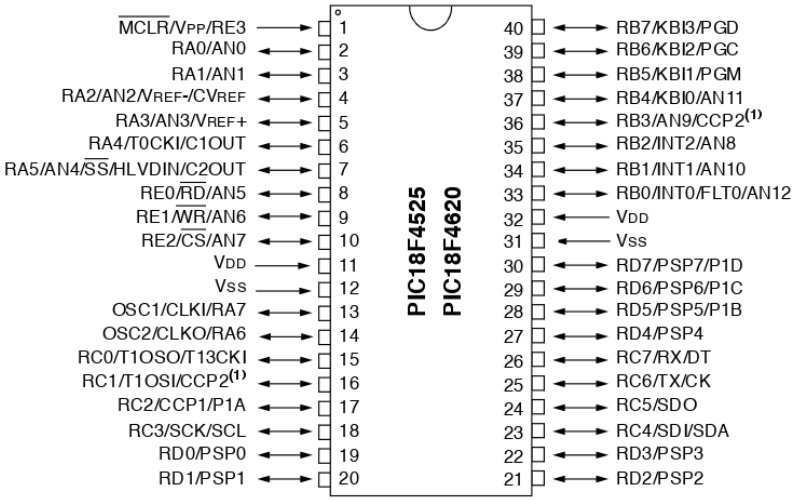



FIGURE 1 – Un microcontrôleur PIC et ses broches d'entrée/sortie (numérotées de 1 à 40)

Les ressources physiques de microcontrôleurs sont souvent limitées, comparables à celles d'un ordinateur personnel des années 1980. Ainsi, la fréquence d'horloge d'un microcontrôleur dépasse rarement quelques dizaines de mégahertz, sa mémoire vive se limite à quelques kilooctets, et sa mémoire flash, destinée au stockage du programme, n'atteint au maximum que quelques mégaoctets. Ces ressources peuvent sembler fortement anachroniques, dans un monde où le moindre système sur puce récent possède une vitesse de calcul cent fois plus rapide, et des ressources mémoire cent mille fois plus importantes, néanmoins les microcontrôleurs continuent à être très fréquemment utilisés dans le monde industriel : leur faible consommation énergétique et leur faible coût d'achat (quelques dixièmes d'euros à quelques dizaines d'euros) contribuent à leur omniprésence dans plusieurs objets du quotidiens. De plus, un nombre croissant d'utilisateurs hobbyistes et bricoleurs s'adonnent à la programmation de microcontrôleurs, dans le but par exemple de développer de petits montages destinés à une utilisation domotique (arrosage automatique de plantes, contrôle de l'allumage d'une pièce . . . ) ou ludique (petites consoles de jeux, drones, murs de LED, etc.). Ces utilisations, stimulées par l'apparition de nouvelles catégories d'objets connectés, font des microcontrôleurs des composants toujours fréquemment utilisés pour le développement de solutions embarquées, professionnelles ou amateures. Ainsi, sous l'impulsion de l'internet des objets (*Internet of Things – IoT*), dont certaines prédictions évaluent le nombre de nouveaux appareils produits entre 2017 et 2025 à un *billion* [2] [Spa17], il a été estimé que le marché global des microcontrôleurs posséderait un taux de croissance annuel d'environ 12% entre les années 2016 et 2023 [⚓5] [3].

De manière analogue au fait que les ressources de microcontrôleurs s'apparentent à du matériel informatique classique relativement daté, les procédés de développement pour microcontrôleurs ont également peu évolué avec le temps : la programmation de microcontrôleurs est traditionnellement réalisée dans des langages qui peuvent être considérés comme étant de bas, voire de très bas niveau. Il

2. Un billion = $10^{12}$.
3. Les références web présentes dans ce document seront toutes précédées d'un symbole représentant une ancre ⚓.

n'est en effet pas exceptionnel que des programmes destinés à être exécutés sur des microcontrôleurs continuent de nos jours à être écrits directement en langage assembleur. De ce fait, la programmation de microcontrôleur est une activité difficile, puisqu'elle contraint le développeur à être familier avec le jeu d'instructions, souvent peu expressif, du langage assembleur du microcontrôleur cible. L'utilisateur se doit également de connaître avec précision le matériel qui exécutera son programme, puisque le niveau d'abstraction matériel offert par de tels langages est très limité, voire inexistant. Cette spécialisation extrême des programmes destinés aux microcontrôleurs, en plus d'être difficile à aborder pour un programmeur néophyte plus familier de langages de plus haut niveau comme Python ou Java, restreint la capacité pour un programme d'être porté facilement vers une architecture différente, de ce fait le moindre changement de modèle de microcontrôleur pour l'application développée peut entraîner une réécriture complète du code d'un programme.

Tester et déboguer un programme de microcontrôleur est un procédé complexe et tout aussi contraignant. Alors que le débogage d'un programme destiné à un ordinateur personnel est simplifié par l'utilisation de débogueurs logiciels permettant de simuler pas à pas l'exécution du programme tout en surveillant le contenu de la mémoire utilisée, et que lancer un jeu de tests sur un PC peut se faire d'un simple clic de souris, les environnements de développement destinés à la programmation de microcontrôleurs n'offrent pas systématiquement la possibilité de simuler l'exécution d'un programme, et cette simulation peut être difficile, compte tenu du fait que le programme est fortement lié à son environnement physique d'exécution. De ce fait, le test d'un programme destiné à un microcontrôleur revient couramment à exécuter ce dernier sur le montage réel auquel il est destiné. Déboguer un programme de microcontrôleur consiste alors très souvent, pour un développeur amateur, à le compiler, transférer l'exécutable généré sur le microcontrôleur, et à simplement tester si le programme a le comportement attendu au sein du montage électronique réel. Ce processus peut alors se répéter à de nombreuses reprises jusqu'à ce que le programme *semble* fonctionner correctement. Cette pratique fastidieuse et peu sûre peut, en plus d'être extrêmement chronophage, endommager le matériel électronique utilisé, par exemple par usure des ressources matérielles du microcontrôleur (dont la mémoire non volatile est limitée en nombre d'écritures totales), ou même par la destruction des composants périphériques au microcontrôleur (ou même ses composants internes), si celui-ci intéragit avec eux de façon erronée.

Au sein de certaines communautés de développeurs familiers de la programmation de microcontrôleurs, la simple utilisation d'un sous-ensemble du langage C pour programmer des systèmes embarqués peut être considérée comme relevant de la programmation de haut niveau. En effet, comparé aux langages assembleurs, le langage C offre une couche d'abstraction matérielle notable permettant un développement et un débogage simplifiés, et apporte diverses garanties sur les programmes écrits (parmi lesquelles, quelques vérifications statiques simples sur le typage des programmes) qui rendent ces programmes plus sûrs et moins enclins aux bugs. L'utilisation de langages de relativement bas niveau comme le C représente ainsi souvent une limite rarement dépassée par les développeurs de systèmes embarqués, attachés à la volonté de contrôler avec précision la consommation en ressources de leurs programmes et leur interaction avec le matériel. Pourtant, de nombreux langages de plus haut niveau possèdent des avantages bienvenus dans le contexte de la programmation de microcontrôleurs : en effet, en plus de l'expressivité accrue fournie par l'utilisation de langages implantant des paradigmes de programmation de haut niveau, comme la programmation orientée objet ou la programmation fonctionnelle, les garanties offertes par l'utilisation de tels langages (comme des vérifications de typage statiques ou dynamiques) forment un avantage certain pour la programmation de systèmes embarqués dont le mauvais fonctionnement peut

avoir des conséquences désastreuses. De surcroît, les abstractions matérielles offertes par ces langages ne contrecarent pas forcément l'utilisation raisonnée de ces ressources : en effet, les capacités mémoire des microcontrôleurs étant faibles, il est justement intéressant dans certains cas de pouvoir bénéficier d'un environnement d'exécution capable de réutiliser ces ressources de manière dynamique, dépendant des besoins du programme. Pour ces raisons, plusieurs travaux ont été réalisés afin de permettre le développement de programmes embarqués dans des langages de programmation de plus haut niveau comme Java, Python ou bien Scheme. D'autres solutions proposent des langages spécialisés pour la programmation de microcontrôleurs inspirés eux mêmes de modèles de programmation de plus haut niveau, comme la programmation fonctionnelle réactive [HG16], ou la programmation graphique [Kat10, MS17].

En outre, les langages généralistes classiquement utilisés ne sont pas forcément adaptés aux caractères inhérents à la programmation d'applications pour microcontrôleurs : en effet, puisqu'un système embarqué se doit généralement de réagir rapidement à divers stimuli (comme l'appui d'un bouton ou le changement de la valeur calculée par un capteur), la programmation de microcontrôleurs expose souvent un comportement concurrent. De ce fait, offrir un modèle de programmation concurrente léger et adapté à de telles applications constitue un avantage notable pour la simplification du développement de tels programmes. En conséquence, le modèle de concurrence à adopter pour programmer un microcontrôleur ne peut pas systématiquement, de par les limites en ressources d'un microcontrôleur ainsi que ses particularités intrinsèques (telle que l'absence de système d'exploitation), être calqué sur les modèles de concurrence habituellement utilisés pour programmer des applications pour ordinateurs ou téléphones mobiles (comme des systèmes à bases de *threads*). Toutefois, des modèles de programmation concurrente moins courants sont adaptés à la programmation de microcontrôleurs : en particulier, le modèle de programmation synchrone nous semble particulièrement adéquat, en raison de sa capacité à ne nécessiter que très peu de ressources matérielles, ainsi que de la simplicité de son approche reposant sur *l'hypothèse synchrone*, qui permet de considérer que les divers composants d'un programme s'exécutent tous *instantanément*, de manière concurrente. Ce paradigme a l'avantage de libérer le développeur des considérations liées à la synchronisation entre les divers composants d'un programme, ce qui contribue à la montée en abstraction du processus de développement logiciel.

Par ailleurs, les programmes destinés aux systèmes embarqués, parfois critiques, nécessitent souvent de satisfaire des contraintes physiques ou comportementales strictes. En effet, certaines applications de microcontrôleurs (comme les drones, les moyens de transport ou certains robots) correspondent à des *systèmes temps réel* pour lesquels la réaction aux stimuli extérieurs doit se faire en dessous d'un certain intervalle de temps. Ce délai imposé correspond généralement à des contraintes de sûreté, et permettre d'assurer la vitesse de réaction du système est une condition essentielle pour que le programme ait le fonctionnement attendu. D'autres contraintes peuvent concerner le fonctionnement logique des programmes, pour lesquels certains invariants, traduisant la correction du programme, peuvent être vérifiés.

Nous nous inspirons en partie des méthodes de développement de l'avionique civile, pour laquelle des outils industriels dérivés de langages synchrone, comme SCADE [CPP17], permettent la certification DO-178C des aéronefs (et de leurs logiciels embarqués). L'utilisation de méthodes formelles (interprétation abstraite, *model checking*, . . . ) permettant de vérifier certaines propriétés des programmes, est par ailleurs encouragée. Par exemple, il est désormais possible de remplacer les tests par des preuves dans

la norme DO-178 [4]. Ces analyses robustes offrent un intérêt certain pour la programmation de systèmes embarqués critiques. Notre approche, mêlant des modèles de programmation de haut niveau au modèle de programmation synchrone à flots de données, reprend des analyses semblables pour vérifier la correction de leur fonctionnement ainsi que certaines propriétés liées par exemple au temps de calcul d'un instant synchrone.

L'ambition de cette thèse est de proposer une solution contribuant au rapprochement de deux mondes qui peuvent sembler éloignés. D'un côté, le monde de la programmation de microcontrôleurs, riche en applications relativement simples et en utilisateurs hobbyistes et industriels, mais limitée par les ressources du matériel, et dont les procédés de développement sont généralement assez pauvres en sûreté et en garanties logicielles. De l'autre, le monde des modèles de programmation de plus haut niveau, qui offre des abstractions permettant une plus grande richesse d'expressivité et une plus grande sûreté, mais produit des programmes potentiellement plus gourmands en ressources. Toute l'ambition de notre approche repose sur la capacité à offrir au développeur de microcontrôleurs des techniques de développement modernes et puissantes, qui permettent d'apporter des garanties supplémentaires aux programmes réalisés, tout en prenant en compte des capacités limitées du matériel considéré. Nos solutions auront ainsi pour objectif d'être exécutables sur du matériel à très faibles ressources, de l'ordre de quelques kilo-octets de mémoire vive. Cette volonté de compatibilité avec des microcontrôleurs aux ressources limitées dérive du fait qu'aujourd'hui encore, les microcontrôleurs 8 bits, dotés de quelques kilo-octets de RAM, dominent toujours le marché [SSD+17]. De plus, le fait d'offrir des solutions qui fonctionnent sur du matériel aussi limité nous permet d'assurer que notre proposition est adaptée au développement d'une large palette de modèles de microcontrôleurs, sans se limiter aux modèles de gammes supérieures, mieux pourvus en ressources, mais moins couramment utilisés.

Ce manuscrit décrit le détail de notre solution, qui repose sur une suite d'abstractions ayant pour objectif de simplifier les procédés de développement et de garantir la correction des programmes pour systèmes embarqués. Nous nous attacherons en particulier à formaliser plusieurs aspects de notre discours, ainsi qu'à prouver certaines caractéristiques métathéoriques de notre solution. Cette formalisation a été en grande partie mécanisée par des moyens logiciels, et plusieurs preuve ayant trait à ce formalisme ont été réalisées à l'aide de l'assistant de preuve Coq [Tea19]. Les sources des programmes, exemples, et preuves de lemmes et théorèmes énoncés tout au long de ce manuscrit sont accessibles en ligne [⚓1].

4. des informations additionnelles à ce propos sont disponibles dans un document nommé DO-333

**Plan du manuscrit**

— Le chapitre 1 est une présentation des méthodes classiques de programmation de microcontrôleurs, ainsi que de travaux préexistants visant à augmenter l'expressivité et la sûreté de la programmation de microcontrôleurs. Nous y abordons l'état de l'art du modèle de programmation synchrone, ainsi que les techniques classiques visant à apporter des garanties sur les programmes synchrones dans le domaine de l'embarqué critique.

— Le chapitre 2 est la présentation de notre implantation de la machine virtuelle OCaml, nommée OMicroB. Cette machine virtuelle, destinée à être exécutée sur microcontrôleurs dont les limitations en mémoire sont importantes, constitue, de par sa portabilité ainsi que les spécificités du langage OCaml et de sa bibliothèque d'exécution, un premier niveau d'abstraction pour une programmation plus expressive et plus sûre de microcontrôleurs.

— Dans le chapitre 3, nous proposons OCaLustre, une extension synchrone à flots de données du langage OCaml. Destinée à la programmation de microcontrôleurs et inspirée du langage Lustre, cette extension de langage permet d'offrir un modèle de programmation simple et peu gourmand pour le développement de comportements concurrents d'un système embarqué. Après un tour d'horizon des fonctionnalités du langage, nous décrivons dans ce chapitre les aspects de spécification formelle relatifs à ce dernier.

— Au chapitre 4, nous décrivons les principales étapes de compilation des programmes OCaLustre. Ce processus de compilation permet de transformer un code synchrone en un programme OCaml séquentiel, pleinement compatible avec tout compilateur du langage, tout en vérifiant plusieurs garanties statiques relatives au typage et à l'ordonnancement des composants logiciels des programmes OCaLustres.

— Au chapitre 5, nous présentons des méthodes de formalisation et vérification de diverses propriétés issues de la spécification du langage. Nous décrirons en particulier des méthodes permettant de vérifier la cohérence d'un programme OCaLustre avec deux systèmes de types dont les règles permettent de définir la bonne sémantique d'un programme.

— Le chapitre 6 décrit une méthode permettant le calcul du temps d'exécution pire cas d'un programme OCaLustre. Cette méthode profite de la portabilité de notre approche en reposant sur l'analyse du fichier *bytecode* généré après compilation d'un tel programme. Nous présentons la preuve de correction de ce procédé, avant de décrire le prototype logiciel qui implante ce dernier.

— Le chapitre 7 constitue une analyse des performances des différentes solutions logicielles présentées dans ce manuscrit. À partir de plusieurs exemples de programmes qui implantent chacun des fonctionnalités du langage OCaml, nous y décrivons l'empreinte mémoire et la vitesse de la machine virtuelle OMicroB. Nous détaillons par la suite la faible empreinte mémoire de l'extension synchrone OCaLustre, ainsi que ses performances de vitesse.

— Le chapitre 8 est destiné à présenter plusieurs applications concrètes, mettant en exergue les avantages liés à nos divers niveaux d'abstraction et à la vérification de garanties sur les programmes réalisés. Ils témoignent en particulier de la facilité de décrire les interactions entre un microcontrôleur et son environnement, et confirment l'adéquation entre le modèle choisi et les ressources limitées des appareils que nous ciblons.

— Nous concluons finalement cette thèse en rappelant les diverses approches proposées dans nos travaux, avant d'aborder diverses extensions à nos solutions, destinées à poursuivre la montée en abstraction et en sûreté des modèles considérés.

# 1 Préliminaires

Les différentes montées en abstractions proposées dans cette thèse reposent sur des choix techniques et technologiques motivés par des considérations pratiques. Ce chapitre constitue un panorama de différentes solutions issues de l'état de l'art qui concernent chacun des niveaux d'abstraction considérés. Au fil de la description de ces divers travaux, nous justifions les choix pris dans cette thèse pour s'orienter dans une direction plutôt qu'une autre. Ainsi, nous abordons tout d'abord la composition physique d'un microcontrôleur, ainsi que les méthodes de programmation classiques utilisées dans le cadre de la programmation d'applications embarquées. Nous motivons par la suite notre volonté d'abstraire le matériel utilisé par l'intermédiaire d'une machine virtuelle d'un langage de haut niveau, riche en expressivité et pour lequel l'empreinte mémoire de l'environnement d'exécution est faible. Notre volonté de fournir un modèle de programmation concurrente qui soit simple et léger est ensuite appuyée par le choix d'utilisation d'un modèle de programmation synchrone à flots de données, adapté à la nature des programmes pour microcontrôleurs. Enfin, nous abordons différents moyens de vérifier et garantir certaines propriétés sur les programmes réalisés à partir de ces abstractions.

## 1.1 Caractéristiques physiques et logicielles des microcontrôleurs

Dans le but de rendre compte au lecteur de la spécificité des enjeux et techniques liés à la programmation pour microcontrôleurs, nous présentons dans cette section une vision générale des aspects matériels principaux de tels circuits intégrés génériques. Nous décrivons rapidement la structure d'un microcontrôleur générique, avant d'aborder leurs caractéristiques techniques, et en particulier leurs limitations mémoires. Nous discutons également des méthodes de programmation classiques de microcontrôleurs, les environnements de développement qui leur sont liés, ainsi que les limitations de ces procédés de développement.

### 1.1.1 Composition et ressources d'un microcontrôleur

Un microcontrôleur est constitué de plusieurs éléments lui permettant d'effectuer les calculs nécessaires à l'exécution de programmes qui lui sont dédiés, et à l'interaction avec le circuit électronique qui constitue son environnement [BV97]. Les principaux composants d'un microcontrôleur sont les suivants :

— Une unité centrale de calcul, ou *CPU* (*Central Processing Unit*), dont le rôle est de réaliser les calculs arithmétiques et logiques nécessaires à l'exécution d'un programme. La vitesse de calcul d'un microcontrôleur doté de ressources limitées se compte en général en dizaines de MIPS [1].

— Une mémoire vive, ou *RAM* (*Random-Access Memory*) qui contient les données dynamiques, volatiles, générées lors de l'exécution du programme. Dans le contexte de cette thèse, la RAM constitue

1. Millions d'instructions par seconde.

certainement la ressource la plus limitée : de l'ordre de quelques kilo-octets (ko) sur les microcontrôleurs que nous considérons, elle contraint le développeur à contrôler finement la consommation mémoire de son programme.

— Une mémoire flash, non-volatile, qui est destinée à stocker le code du programme. Bien que la mémoire flash soit techniquement accessible en écriture au cours de l'exécution d'un programme, il est habituel de la considérer comme une mémoire morte (ou *ROM* : *Read-Only Memory*), en raison d'une part de la volonté de séparer les données et les programmes, et d'autre part de ses limitations physiques. En effet, la mémoire flash est limitée en nombre maximal d'écriture. Par exemple, la mémoire flash d'un microcontrôleur ATmega peut supporter (selon la spécification du fabricant) environ 10 000 cycles d'écriture au maximum tandis que sa RAM est virtuellement inusable. La taille de la mémoire flash est généralement dix à cent fois plus vaste que celle de la RAM. Celle des microcontrôleurs que nous considérons est de quelques dizaines voire quelques centaines de kilo-octets : le microcontrôleur ATmega328P possède par exemple 32ko de mémoire flash.
— Des compteurs (ou *timers*) qui permettent de mesurer des durées, afin de synchroniser les divers composants électroniques d'un montage entre eux. Ces compteurs sont incrémentés à intervalles de temps réguliers. Un signal d'horloge interne, apparaissant à chaque exécution d'une instruction du programme, cadence la vitesse d'incrémentation des compteurs.
— Un mécanisme d'interruptions permettant de déclencher, dès l'apparition d'un stimulus interne ou externe particulier, l'exécution immédiate d'une routine spécialisée pour le traiter. Les interruptions externes peuvent provenir de l'apparition (ou la variation) d'un signal électrique sur une broche (par exemple via l'appui d'un bouton-poussoir), tandis que les interruptions internes sont par exemple causées par le débordement (*overflow*) d'un compteur ou les pulsations d'un oscillateur électronique interne.
— Des ports d'entrées/sorties permettent de communiquer avec les composants électroniques connectés au microcontrôleur. Cette communication est réalisée en échangeant des signaux électriques de tension variable (par exemple 0 ou 5 Volts), permettant de représenter la transmission de signaux binaires entre le microcontrôleur et son environnement. Ces ports correspondent à un ensemble de broches métalliques visibles depuis l'extérieur du microcontrôleur, sur lesquelles sont soudés des fils ou des circuits imprimés qui les relient aux composants électroniques externes. Chaque broche d'un port doit généralement être configurée (typiquement par la modification d'une valeur d'un bit du registre correspondant au port concerné) au début du programme afin de la déclarer comme une interface d'émission, ou de réception.
— Certains ports d'entrée/sortie supportent par ailleurs la lecture de valeurs analogiques par l'intermédiaire de *convertisseurs analogique-numérique* (ou *ADC* : *Analog-to-Digital Converters*). Ces derniers sont capables de traduire, par échantillonnage temporel, la variation de valeurs de tensions aux bornes d'une broche en une valeur analogique. De la même façon, la traduction d'un signal numérique interne vers un signal analogique externe est rendue possible par la présence d'un *convertisseur numérique-analogique*, ou *DAC* (*Digital-to-Analog Converter*).

La figure 1.1 est une représentation schématique simplifiée des relations entre les éléments principaux d'un microcontrôleur. Les interactions entre ces derniers sont essentiellement réalisées via des transferts de valeurs sur un (ou plusieurs) bus bi-directionnel, permettant par exemple de communiquer les instructions du programme depuis la mémoire flash jusqu'au CPU, ou d'écrire des données issues d'un

calcul vers la RAM.

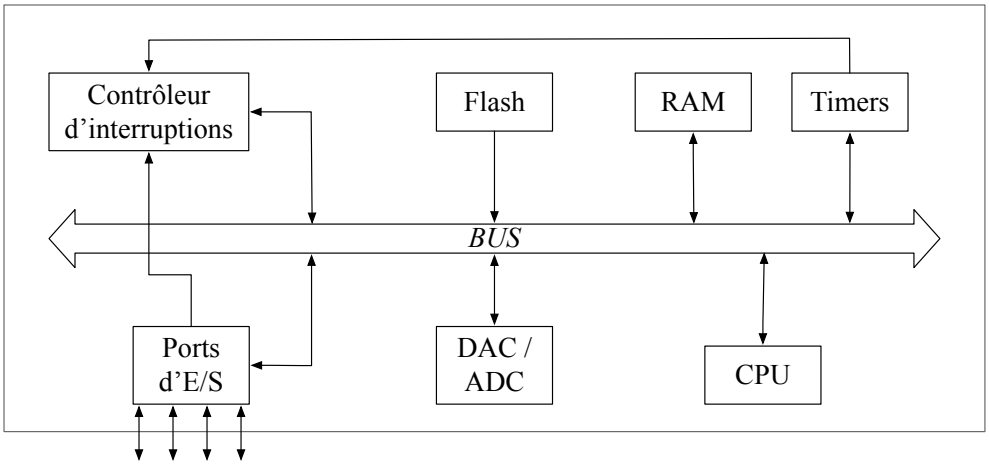



FIGURE 1.1 – Structure interne (simplifiée) d'un microcontrôleur

Plusieurs familles de microcontrôleurs existent actuellement sur le marché. Parmi elles, les microcontrôleurs AVR constituent des appareils très utilisés par les programmeurs amateurs, de par leur présence dans les cartes Arduino/Genuino [⚓15]. Ces cartes, qui contiennent un microcontrôleur et des composants électroniques pré-connectés au microcontrôleur (un port USB, un bouton de réinitialisation, un fiche d'alimentation, etc.) permettent de simplifier le processus de développement de programmes embarqués. La carte Arduino Uno [Bad14], contient par exemple un microcontrôleur AVR ATmega328, doté de 2 kilo-octets de RAM et de 32 kilo-octets de mémoire flash. D'autres familles de microcontrôleurs, aux architectures différentes, sont également disponibles, comme les STM32 basés sur une architecture ARM, et souvent pourvus de ressources plus importantes. Les microcontrôleurs PIC, moins connus des publics hobbyistes, sont très communément utilisés par les industriels en raison de leur faible coût et de leur efficacité.

À titre d'exemple, le tableau de la figure 1.2 représente les caractéristiques physiques d'un ensemble de microcontrôleurs issus de plusieurs familles, certains très pauvres en ressources (moins d'un kilo-octet de RAM) et d'autres se rapprochant des capacités physiques de PC des années 1990. Dans notre contexte d'utilisation, nous nous intéressons à des microcontrôleurs plus proches de la gamme inférieure, dotés d'assez peu de mémoire flash (moins de 100 kilo-octets) et d'une mémoire vive fortement limitée (moins de 8 kilo-octets), comme le PIC 18F4620.

| Modèle | Architecture | Mémoire flash (ko) | RAM (o) | Vitesse CPU (MIPS) | Tensions de fonctionnement |
|---|---|---|---|---|---|
| AT89C51 | Intel 8051 - 8 bits | 4 | 128 | 12 | 4 à 6 V |
| ATmega328P | AVR - 8 bits | 32 | 2048 | 20 | 1,8 à 5,5 V |
| ATmega2560 | AVR - 8 bits | 256 | 8192 | 16 | 1,8 à 5,5 V |
| PIC 18F4620 | PIC - 8 bits | 64 | 4096 | 10 | 2 à 5,5 V |
| PIC 24FJ128GA006 | PIC - 16 bits | 128 | 8192 | 16 | 2 à 3,6 V |
| STM32 L051C8 | ARM - 32 bits | 64 | 8192 | 32 | 1.65 à 3.6 V |
| STM32 F091VCT6 | ARM - 32 bits | 256 | 32768 | 48 | 2 à 3,6 V |

FIGURE 1.2 – Quelques microcontrôleurs et leurs caractéristiques

Un microcontrôleur est donc très proche structurellement d'une représentation simplifiée d'un ordinateur personnel, mais possède des ressources bien inférieures à ce dernier. Ses applications sont pour leur part très différentes de celles destinées aux PC classiques, ou même à des ordinateurs monocartes (comme les *Raspberry Pi*) : il ne possède en effet généralement ni système d'exploitation, ni système de fichier, ni périphériques classiques comme un clavier ou une souris. Un microcontrôleur est typiquement dédié au contrôle d'une tâche automatisée, bien souvent dans un contexte *embarqué* où il n'est relié qu'à un circuit électronique spécifique à cette tâche, et où une application est exécutée « sur du métal nu » (*bare-metal*) : sans aucun intermédiaire logiciel entre le matériel et le programme.

### 1.1.2 Modèles de programmation classiques de microcontrôleurs

#### Langages de programmation de bas niveau

Réaliser une application pour microcontrôleur est une tâche assez complexe, qui nécessite pour le développeur de bien connaître le matériel utilisé. En effet, la proximité, dans un système embarqué, entre les programmes et le matériel entraîne traditionnellement l'utilisation de langages de programmation de bas niveau, qui n'offrent pas ou peu d'abstractions vis-à-vis du matériel pour le développement d'applications. Les applications pour systèmes embarqués sont souvent réalisées en langage assembleur, dans une volonté de contrôler finement la configuration du matériel et sa consommation de ressources. Dès lors, le choix du microcontrôleur a une influence importante sur le processus de développement, car les différentes familles de microcontrôleurs disponibles ne partagent pas le même jeu d'instruction assembleur. Un programme réalisé pour microcontrôleur PIC n'est ainsi pas adapté pour être exécuté sur un microcontrôleur AVR, et vice-versa. Les différences physiques entre microcontrôleurs de la même famille peuvent également limiter la portabilité d'un programme.

Relativement plus portables, des sous-ensembles du langage C sont également couramment utilisés pour la programmation d'applications embarquées. En effet, l'expressivité plus importante du C par rapport à l'assembleur est appreciée par les développeurs d'applications embarquées, tandis que sa finesse de contrôle des ressources mémoire est considérée comme essentielle au développement d'applications destinées à être exécutées sur des appareils à faibles ressources. Malgré tout, les programmes traditionnellement réalisés pour microcontrôleurs sont assez limités en abstraction matérielle, en portabilité, et n'offrent que peu de vérifications statiques vis-à-vis de certaines propriétés sur les programmes, comme par exemple la correction du typage d'un programme.

Par exemple, la figure 1.3 représente le code source d'un programme C destiné à un microcontrôleur AVR ATmega328P. Ce programme émet à intervalles réguliers une impulsion électrique sur la broche PB5 (c.-à-d. la broche numéro 5 du port B du microcontrôleur), afin de faire clignoter une diode électroluminescente (aussi appelée *LED – Light-Emitting Diode*) à laquelle elle est connectée [2].

Il est ici important de noter la lisibilité assez faible d'un tel programme : les interactions avec les ports d'entrées/sorties sont réalisées par des opérations binaires qui modifient, par l'intermédiaire de masques, les bits de registres représentant les ports du programme (`DDRB` permet la configuration des broches du port B en entrée ou en sortie, et `PORTB` la modification de leurs valeurs). Dans un programme plus complexe, la moindre petite erreur dans ce type d'opérations peut être assez difficile à retrouver par le développeur, et être source de dysfonctionnements qui peuvent être dommageables pour le montage électronique.

2. L'opérateur ^ correspond en C au « ou-exclusif » entre deux bits

```
#ifndef F_CPU
#define F_CPU 20000000UL // vitesse d'horloge 20 MHz
#endif

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  DDRB = DDRB | 0b00001000; // Configure PB5 en sortie
  while(1)
  {
    PORTB = PORTB | 0b00001000; // Allumer PB5
    _delay_ms(500); // attente d'une demi-seconde
    PORTB = PORTB ^ 0b00001000; // Eteindre PB5
    _delay_ms(500); // attente d'une demi-seconde
  }
}
```

FIGURE 1.3 – Un programme C pour microcontrôleur ATmega328P

Certaines bibliothèques logicielles, comme la bibliothèque Arduino, offrent au programmeur des primitives C qui permettent d'abstraire légèrement ce type d'opérations (par exemple, via l'utilisation d'une fonction `digitalWrite()` prenant en paramètre le nom d'une broche et le signal à émettre), mais restent assez limitées, et sont comparables à de simples macros rendant les programmes un peu plus faciles à lire et écrire, mais n'apportant pas de sûreté supplémentaire aux programmes réalisés.

La portabilité d'un tel programme est également fortement limitée : par exemple, certains microcontrôleurs AVR ne disposent pas des mêmes noms de ports, et le portage de ce programme vers un microcontrôleur d'une autre famille entraîne des changements profonds, liés au microcontrôleur choisi (par exemple, la configuration des registres des ports est différente entre les microcontrôleurs AVR et les microcontrôleurs PIC : un bit à 1 dans le registre de configuration d'un port représente une broche réglée en sortie sur un microcontrôleur AVR, alors qu'il représente une broche configurée en entrée sur un PIC).

#### Environnements de développement

Les environnements de développement permettant la réalisation de programmes pour microcontrôleurs et leur débogage sont eux aussi complexes, et assez peu modulaires. Quelques solutions propriétaires, généralement mono-plateformes, sont fournies par les fabricants. Par exemple, l'environnement intégré (*Integrated Development Environment – IDE*) MPLAB [⚓10] permet de programmer, simuler, et transférer des programmes sur microcontrôleurs PIC. *AtmelStudio* [⚓8] est, quant à lui, un IDE pour microcontrôleurs AVR offrant peu ou prou les mêmes fonctionnalités que MPLAB. Citons enfin l'*IDE Arduino*[⚓7], une application multi-plateformes qui permet d'écrire des programmes pour cartes de développement Arduino, les compiler, et transmettre les exécutables générés sur la carte. La figure 1.4 représente l'interface utilisateur très simple de ce logiciel.

Ces outils grand public sont pour la plupart assez modestes en fonctionnalités : ils sont capables de générer un exécutable pour le microcontrôleur, mais ne permettent pas forcément de tester correctement le programme réalisé, puisque le comportement d'un tel programme est intrinsèquement lié au montage électronique qui l'entoure. De ce fait, des solutions logicielles plus puissantes, comme la suite


FIGURE 1.4 – L'environnement de développement intégré (IDE) Arduino

logicielle Proteus [⚓17], permettent la composition assistée par ordinateur de montages électroniques et la simulation des programmes de microcontrôleurs sous forme exécutable. Malheureusement, ces solutions logicielles sont très onéreuses et assez complexes à aborder, et donc difficilement utilisables par des développeurs non spécialistes du domaine. Il n'est ainsi pas rare que les programmeurs de petites applications embarquées préfèrent une méthode de débogage *in-situ* consistant en l'utilisation directe du microcontrôleur physique pour tester leur programme : avec une connexion série reliée à un ordinateur, ils analysent les réponses du programme à plusieurs stimuli. Cette méthode de débogage est alors une tâche longue et pénible, et plusieurs erreurs dans le code du programme, qui auraient pu être détectées en amont du transfert sur le montage physique, peuvent ralentir le cycle de développement d'un programme.

Le programmeur désirant s'orienter vers des solutions *open source* peut recourir à une poignée de compilateurs C libres, basés la plupart sur GCC, qui sont disponibles sur plusieurs plateformes. Par exemple le compilateur avr-gcc [⚓25], associé à la bibliothèque standard avr-libc et les outils binaires avr-binutils constituent une chaîne de compilation complète pour microcontrôleurs AVR. L'envoi du programme exécutable généré peut être alors réalisé avec l'outil avrdude (*AVR Downloader UploaDEr*), fourni avec le compilateur. Côté PIC, un travail toujours en cours au sein du compilateur *sdcc* [⚓27] permet la traduction de code source C pour des appareils des familles PIC16 et PIC18. L'outil *usbpicprog* [⚓23] constitue quant à lui un *programmateur de puce* (c.-à-d. un appareil permettant de transférer un programme depuis un ordinateur vers la mémoire flash d'un microcontrôleur) *open source* pour ces microcontrôleurs.

Les microcontrôleurs sont des circuits intégrés programmables munis de ressources et environnements de développement assez limités. Les méthodes de programmation de microcontrôleurs suivent généralement cette faiblesse de ressources, et sont ainsi traditionnellement basées sur des utilisations de langages de bas niveau. De ce fait, l'utilisation de ces langages peut dissuader des développeurs non-aguerris, et peu familiers de telles technologies. Le développement dans des langages assembleur ou en langage C induisent la réalisation de programmes plutôt longs à écrire, et qui peuvent, qui plus est, être sources d'erreurs difficilement détectables.

Il nous semble alors pertinent d'explorer des modèles de programmation différents, s'inspirant des abstractions utilisées pour le développement d'applications pour ordinateurs personnels, dans le but d'offrir à la programmation de microcontrôleurs des outils plus faciles à appréhender. Nous nous intéressons en particulier à l'utilisation de langages dits de *haut-niveau*, pour la programmation d'applications embarquées.

## 1.2 Abstraction matérielle et langages de haut niveau

Il est désormais très rare, dans le contexte plus courant de la programmation pour ordinateurs personnels (ou pour des appareils modernes équivalents comme les smartphones ou autres tablettes tactiles), que des programmes soient réalisés directement en langages assembleur. De la même façon, l'utilisation du langage C se limite de plus en plus à des domaines particuliers, comme le développement d'applications systèmes censées par leur nature manipuler directement la mémoire. Il est aujourd'hui monnaie courante de programmer les applications avec lesquelles l'utilisateur est en interaction directe (programmes pour téléphones mobiles, applications web, ...) dans des langages de haut niveau, qui facilitent la programmation d'applications complexes.

Les langages de programmation de haut niveau, sont des langages dont la richesse expressive s'abstrait du modèle de calcul de la machine sur laquelle les programmes sont exécutés. Ils offrent un contrôle riche, implantant des paradigmes de programmation variés qui peuvent manipuler par exemple des valeurs fonctionnelles, des objets, des exceptions, ou des continuations. Ces langages offrent nativement des structures de données riches, permettant de réaliser plus facilement des applications complexes. Plusieurs langages de programmation de haut niveau utilisent une notion de typage statique afin d'améliorer la sûreté des programmes, et reposent sur des environnements d'exécution permettant la gestion automatique de la mémoire d'un programme. Le processus de débogage d'applications écrites dans ces langages est simplifié par l'utilisation de débogueurs symboliques, qui manipulent les structures de données complexes du langage. Enfin, ces langages de haut niveau sont pourvus de mécanismes permettant l'interopérabilité avec des langages de bas niveau afin de permettre l'interfaçage avec le matériel.

Il semble alors naturel d'envisager, afin d'apporter à la programmation de microcontrôleur ces mêmes avantages, l'utilisation de tels langages pour la programmation de systèmes embarqués, en lieu et place du classique couple de langages assembleur/C. Nous abordons de plus, dans cette section, la compilation de ces langages de haut niveau vers du code non-natif, interprétable dans une *machine virtuelle* (ou *machine abstraite*) positionnée entre le programme et le matériel. Nous décrivons dans la suite les différentes expérimentations et systèmes, issus de l'état de l'art, qui permettent la programmation de haut niveau sur microcontrôleurs à l'aide de cette *approche machine virtuelle*.

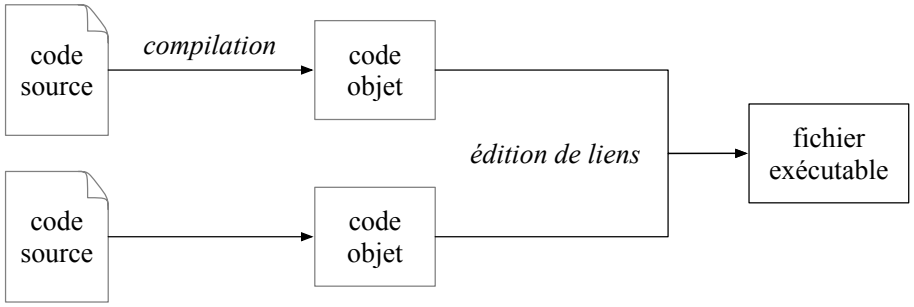



FIGURE 1.5 – Compilation vers un programme natif

### 1.2.1 L'approche machine virtuelle

Dans une vision très simplifiée, le processus de compilation classique d'un programme consiste à en traduire le code source du programme vers un code natif, capable d'être « compris » par le matériel sur lequel il sera exécuté. La figure 1.5 décrit un tel mécanisme : des fichiers sources sont traduits vers le langage natif de la machine à laquelle le programme est destiné. Dans le cas de langages de programmation de bas niveau, cette traduction est plutôt aisée, étant donnée la proximité des instructions du langage et de celles du matériel. L'assembleur n'est qu'une version lisible par un humain des instructions binaires du processeur, et la compilation d'un programme C est assez directe (dès lors qu'on ne cherche pas à optimiser le code produit), puisque les opérateurs de contrôle impératifs du langage se rapprochent fortement de ceux du matériel.

La compilation de programmes implantant des traits de langages de programmation de haut niveau (programmation fonctionnelle, programmation objet, . . . ) est un procédé plus complexe. En effet, puisque les processeurs physiques ne sont capables de traiter nativement que des opérations impératives simples, plusieurs transformations sont appliquées lors de la compilation à un tel programme dans le but de convertir les traits de haut niveau vers du code natif, purement impératif. Les compilateurs des langages de haut niveau manipulent alors de nombreuses représentations intermédiaires des programmes, et ces représentations sont les mêmes que le programme final soit destiné à être exécuté sur une machine dans une architecture $A$, ou sur du matériel possédant une architecture $B$ complètement différente.

L'approche machine virtuelle consiste alors à suivre le processus de compilation d'un programme jusqu'à la production d'une représentation commune et non-native des programmes : le *bytecode* (ou code-octet). Il est ensuite laissé à une *machine virtuelle*, c'est-à-dire un *interprète* de bytecode associé à un *environnement d'exécution*, le soin d'exécuter le programme dans cette forme intermédiaire [DS00]. Cette approche permet la *portabilité* des applications réalisées, puisque tout programme traduit vers le bytecode du langage est exécutable par un quelconque appareil, dès lors que ce dernier bénéficie d'un interprète adapté à ce bytecode. L'utilisation de machines virtuelles pour l'exécution de programmes riches a été popularisée par le langage Java, et sa JVM (*Java Virtual Machine*), dont le crédo « *Write once, run anywhere* » [3] illustrait la capacité d'un même bytecode Java de pouvoir être exécuté sur de nombreuses plateformes différentes.

La figure 1.6 représente la façon dont le code source d'un programme est compilé pour être interprété par une machine virtuelle.

3. « *Écrire une fois, exécuter partout* »

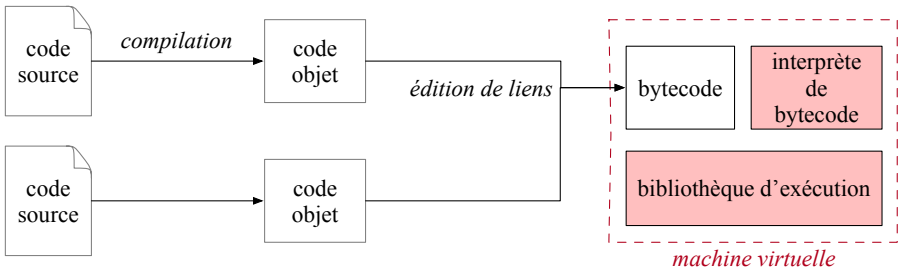



FIGURE 1.6 – Approche machine virtuelle : compilation en *bytecode* et interprétation

Le *bytecode* d'un langage de haut niveau est composé d'instructions plus « riches » que le code machine natif. De ce fait, une instructions *bytecode* correspond typiquement à une séquence d'instructions dans le langage machine. Un programme traduit vers un bytecode peut donc, grâce à cette *factorisation*, être plus compact que son équivalent en langage machine. Bien sûr, le bytecode n'est pas exécutable sans son environnement d'exécution, et la taille de celui-ci doit donc également être prise en compte afin de proposer une comparaison honnête. Néanmoins, puisque cette taille est fixe, l'empreinte mémoire totale d'un programme au bytecode et de sa machine virtuelle peut, dans le cas de programmes conséquents, être inférieure à celle d'un programme compilé vers le langage de la machine. Une telle compaction de la taille totale du programme est un avantage évident dans notre contexte d'utilisation, où les ressources mémoires sont à utiliser avec parcimonie.

En outre, l'utilisation d'un bytecode commun à toutes les plateformes d'exécution permet aussi la factorisation de plusieurs analyses sur les programmes, comme par exemple l'estimation de ressources mémoires [AGG07], du temps de calcul [SP06], la détection de vulnérabilités [LL05], ou même celle de plagiat [JWC08]. Même si certaines analyses peuvent perdre en précision du fait de cette factorisation [LF08], cette dernière reste appréciable pour notre cas d'utilisation, en raison de la variété d'architectures et modèles de microcontrôleurs existants.

### 1.2.2 Programmation de microcontrôleurs dans des langages de haut niveau

Un grand nombre de projets visant à exécuter des langages de haut niveau sur microcontrôleurs ont été réalisés au cours des dernières années. Nous étudions dans cette section un échantillon des divers projets, qui permettent le développement de programmes qui implantent des paradigmes de programmation variés, comme la programmation orientée objet, ou la programmation fonctionnelle. Certaines de ces solutions se basent sur des modèles de compilation natifs, qui consistent à générer du code machine à partir du code source d'un programme, tandis que d'autres profitent des avantages l'approche machine virtuelle et de la représentation commune sous forme de bytecode.

#### Langages de haut niveau et modèles de compilation natifs pour microcontrôleurs

L'utilisation d'un langage de haut niveau n'implique pas systématiquement celle d'une machine virtuelle et d'un interprète de bytecode. En effet, plusieurs solutions qui visent à exécuter des programmes réalisés dans des langages de haut niveau reposent sur un modèle de compilation vers du code natif, qui peut être directement exécuté sur le matériel concerné, sans intermédiaire logiciel. Parmi ces solutions, il est ainsi possible de réaliser des programmes C++, un langage multiparadigmes dont la proximité avec le langage C permet de conserver de bonnes performances, grâce par exemple au compilateur *avr-g++*

pour microcontrôleurs AVR, ou au compilateur *MPLAB XC32++* qui permet la compilation du langage C++, mais seulement pour des microcontrôleurs PIC d'architecture 32 bits. Le compilateur *IAR Embedded Workbench* permet, quant à lui, de compiler des programmes C++ pour des microcontrôleurs de familles variées, comme les microcontrôleurs MSP, les microcontrôleurs AVR, ou encore les microcontrôleurs STM32. Pour autant, le langage C++ ne bénéficie pas de toutes les abstractions apportées par des langages de plus haut niveau, comme par exemple la gestion automatique de la mémoire des programmes, ou un système de type plus sûr.

Plusieurs travaux visant à l'exécution sur microcontrôleurs de programmes écrits dans le langage ADA, un langage de programmation orientée objet souvent utilisé dans les sytèmes embarqués critiques (automobiles, industrie aéronautique, etc), ont également été réalisés [Reg12]. Cependant, certains d'entre eux ne peuvent proposer tous les avantages du langage en raison de leur importante empreinte mémoire [RP19]. Ainsi, le projet de compilateur AVR-Ada [⚓24] est compatible avec des microcontrôleurs AVR 8 bits à faibles ressources, mais ne permet pas, par exemple, l'utilisation du profil *Ravenscar* destiné à la programmation de systèmes temps réel sûrs. Le compilateur *GNAT GPL* développé par la société Adacore bénéficie quant à lui de l'ensemble des *profils* disponibles dans le langage, mais ne permet de viser que les microcontrôleurs sur puce ARM dont les ressources sont bien supérieures.

D'autres langages de haut niveau, comme par exemple le langage Rust, sont compilés vers une représentation intermédiaire commune, avant d'être traduit vers le langage spécifique à la machine. Cette représentation, le *bitcode* de l'infrastructure de compilateur *LLVM* (signifiant historiquement *Low-Level Virtual Machine - Machine Virtuelle de Bas Niveau*), est équivalent à un bytecode bas niveau générique, qui peut être traduit vers du code machine spécifique à chaque cible [Lop09]. Ce *bitcode* pourrait également être interprété, même si cette approche est assez rare. Un projet universitaire visant à réaliser un interprète de *bitcode* pour des microcontrôleurs MSP430 a ainsi constitué une approche originale [Cam16], malgré les lenteurs remarquées sur le prototype. D'autres travaux sont en cours de réalisation pour compiler des programmes Rust vers des microcontrôleurs AVR à l'aide du *backend* LLVM pour AVR [⚓26]. Un autre projet vise l'exécution de programmes écrits dans un sous-ensemble du langage *Go* sur toute cible supportée par LLVM [⚓28]. Malheureusement, le support de microcontrôleurs dans le projet LLVM est très limité : en dehors des travaux sur AVR, seuls certains microcontrôleurs basés sur une architecture ARM sont supportés, et le support de microcontrôleurs PIC a par exemple été abandonné en 2011.

#### Langages de haut niveau et leur machine virtuelle sur microcontrôleurs

Les solutions basées sur une approche machine virtuelle nous semblent être les plus à même d'apporter, de façon aisée, un modèle de programmation de plus haut niveau pour des microcontrôleurs aux ressources limitées. En effet, celles-ci permettent d'augmenter la portabilité des programmes réalisés, grâce à la généricité du bytecode généré lors de leur compilation, qui s'abstrait du matériel sur lequel est exécuté la machine virtuelle. Pour illustrer les avantages de cette approche, nous décrivons dans cette section un échantillon qui nous semble représentatif de différents projets de mise en œuvre de machines virtuelles pour des microcontrôleurs de familles variées.

**Java sur AVR et MSP :** De par sa popularité, la richesse de ses applications, et sa philosophie orientée vers la portabilité des programmes, le langage Java est un candidat évident pour l'utilisation d'un langage de haut niveau sur microcontrôleur. De ce fait, plusieurs projets et expériences de portage de la machine

virtuelle Java, appelée *JVM* (*Java Virtual Machine* [LYBB14]), ont été réalisés. Ces projets permettent aux développeurs d'applications embarquées de profiter de la richesse du langage Java qui propose un modèle de programmation orientée objet, impérative et plus récemment fonctionnelle, ainsi que de son typage statique avec sous-typage nominal, mais aussi de la très riche bibliothèque de classes fournie par le langage (l'*API* Java), qui permet de représenter aisément des structures de données avancées.

Par exemple, la solution industrielle microEJ [⚓16] permet l'exécution de programmes Java sur des cartes de développement contenant des microcontrôleurs basés pour la plupart sur une architecture ARM. Le système Darjeeling [BCL08] se compose, quant à lui, d'une machine virtuelle Java multi-threads qui permet l'exécution de bytecode Java sur des microcontrôleurs AVR128 et MSP430, qui contiennent entre 1 et 8 kilo-octets de RAM et entre 16 et 128 kilo-octets de mémoire flash. D'autres solutions, comme HaikuVM [⚓13] ou NanoVM [⚓14] permettent l'exécution de bytecode Java sur des microcontrôleurs utilisés dans les cartes de développement Arduino.

La richesse du langage Java peut cependant être parfois incompatible avec ces solutions pensées pour être exécutées sur des machines à faibles ressources. Java manipule en effet des données de taille conséquente, qui dépendent elles-mêmes de bon nombre d'objets. Un programme écrit dans un style Java standard, qui fait donc usage d'objets et de types de données complexes, peut induire l'utilisation d'une grande quantité de mémoire, qui n'est pas forcément disponible sur de tels appareils. L'exemple de *Java Card* [Gut97], un système permettant l'exécution de programmes Java sur cartes à puce, témoigne de cette difficulté à utiliser l'intégralité du langage Java sur du matériel aux ressources limitées : la version 2 de Java Card peut être exécutée sur du matériel ayant seulement 2 ko de RAM et 64 ko de ROM, mais ne contient alors pas garbage collector, ne gère pas le multitâche, et se limite à des types de données assez simples (il n'y a par exemple pas de support des tableaux multidimensionnels ou de classes pour manipuler des collections). La version 3 de Java Card, pallie ces manquements, mais au prix d'une consommation en ressources bien supérieure : elle nécessite 24 ko de RAM et au moins 128ko de ROM pour fonctionner. De la même façon, le projet TinyVM [HPK+09], destiné à l'élaboration d'une machine virtuelle Java à faible empreinte mémoire pour des microcontrôleurs RCX (inclus dans les briques programmables *Lego Mindstorm*) qui visait à l'origine le développement de réseaux de capteurs a fini par être intégré au projet LeJOS [LHS10] (dédié lui aussi à l'exécution de programmes Java sur briques programmables), plus riche en fonctionnalités, mais plus gourmand en ressources.

**Python sur STM32 :** MicroPython [Bel17] est une implémentation du langage Python 3 destinée principalement à être exécutée sur une carte nommée *pyboard*, qui contient un microcontrôleur STM32F405RG doté d'un processeur Cortex-M4 cadencé à 168 MHz, d'une mémoire flash de 1024 kilo-octets, et d'une mémoire RAM de 192 kilo-octets. MicroPython permet d'exécuter des programmes Python 3, bénéficiant des avantages et particularités du langage, comme sa syntaxe claire et facile à aborder pour un débutant, son implantation de la programmation orientée objet, et certains traits de haut niveau (comme les compréhensions de listes) qui permettent de développer avec concision des programmes complexes. MicroPython se base sur *CPython*, l'implantation de référence du langage, qui fournit un interprète de bytecode issu de la traduction d'un programme Python. Une capacité intéressante de MicroPython est celle de pouvoir communiquer, depuis un ordinateur, avec une boucle d'interaction (*REPL - Read-Eval-Print-Loop*) qui est exécutée sur le microcontrôleur [VVF18]. Cette communication permet de réaliser rapidement des petits tests, et de déboger facilement les programmes.

MicroPython est destiné à des microcontrôleurs aux ressources plutôt riches, au delà des capacités des microcontrôleurs que nous considérons dans cette thèse. La limite basse des capacités techniques des microcontrôleurs pouvant exécuter l'interprète MicroPython est en effet de 16ko de RAM et 256ko de mémoire pour le programme. D'autres travaux consistent aussi en la réalisation d'un interprète Python pour exécution sur microcontrôleurs [BSR12], mais ceux-ci visent également des appareils aux ressources assez importantes : des Cortex-M3 cadencés au moins à 50 MHz, munis d'au moins 64 ko de RAM, et de 512 kilo-octets de mémoire flash. Enfin, les projets *python-on-a-chip* [⚓12], suivi de *PyMite* [Hal03], aujourd'hui abandonnés, permettaient l'exécution de programmes Python sur des microcontrôleurs à plus faibles ressources (une cinquantaine de kilo-octets de mémoire flash, et moins de 8 kilo-octets de RAM étaient conseillés pour l'exécution d'un programme), mais au prix d'une limitation des fonctionnalités : seul un sous-ensemble du langage Python 2.5 était supporté, et aucune bibliothèque standard n'était fournie. L'ensemble des travaux qui concernent le langage Python traduit ici aussi la difficulté de porter toutes les fonctionnalités d'un langage multiparadigmes sur du matériel dont les ressources sont très limitées.

**Scheme sur PIC :** Scheme est un langage de programmation fonctionnelle dérivé du langage Lisp. Scheme a un fort pouvoir expressif grâce, par exemple, au système de *macros* dont il est muni, ainsi qu'à la manipulation explicite de continuations dans les programmes, via l'instruction *call-with-current-continuation* (*call/cc*). Il existe de multiples implémentations du langage Scheme visant des ordinateurs personnels. Celles-ci se basent sur des spécifications (les « *Revised Reports on the Algorithmic Language Scheme* » ou *RnRS*, où *n* correspond au numéro de la révision du rapport) qui définissent les traits du langage. Toutes ces implémentations (par exemple *Bigloo* [⚓18]) ne reposent pas sur l'utilisation d'une machine virtuelle, mais certaines d'entre elles, par exemple *Guile* [⚓22] (une implémentation de Scheme respectant le standard R6RS) repose sur la traduction du code Scheme vers du bytecode constitué de 175 instructions différentes. Un interprète, écrit en C, permet l'exécution de ce dernier. Quelques machines virtuelles capables d'exécuter des sous-ensembles du langage Scheme sur microcontrôleurs à faibles ressources ont été réalisées. Parmi elles, BIT [DF05] et PICBIT [FD03] permettent l'exécution du standard R4RS sur des microcontrôleurs de moins de 8 kilo-octets de RAM et 64 kilo-octets de mémoire programme, tandis que le système PICOBIT [SF09] permet à des programmes Scheme écrits dans des sous-ensembles du R5RS d'être exécutés sur des PIC18 pouvant être limités à 1 kilo-octet de RAM et 6 kilo-octets de ROM. De part sa légèreté, le langage Scheme semble ainsi plus adapté à des appareils à faibles ressources.

**OCaml sur PIC :** Le langage OCaml est un langage mêlant de nombreux paradigmes de programmation. Il implante, depuis sa création, des traits de programmation par objets, fonctionnelle, modulaire, et impérative. Cette variété des modèles permet un développement facilité et expressif de programmes complexes. Son typage statique fort, associé à un mécanisme d'inférence de types, lui permet d'assurer, au moment de la compilation, l'absence d'incohérences dans l'utilisation des valeurs typées d'un programme. La machine virtuelle standard du langage OCaml, appelée *ZAM* et dérivée de la machine virtuelle *ZINC* [Ler90], est une machine à pile, qui repose sur une représentation uniforme des données. Le bytecode associé à la VM OCaml est constitué de 148 instructions, qui proposent des opérateurs classiques de calcul et de branchement, ainsi que des instructions dédiées à la manipulation de valeurs fonctionnelles et leur application. Le projet OCaPIC [VWC15] est une implantation de la machine virtuelle standard du langage OCaml pour la programmation de microcontrôleurs de la famille PIC18.

OCaPIC permet ainsi l'exécution de l'intégralité du langage OCaml sur des microcontrôleurs aux ressources limitées (4 kilo-octets de RAM et 64 kilo-octets de mémoire flash). Cette capacité d'exécuter l'intégralité du langage OCaml sur des microcontrôleurs avec des ressources aussi limitées est notable, et témoigne de la légèreté de la machine virtuelle OCaml et de la puissance des optimisations apportées par cette implantation. OCaPIC fournit de plus plusieurs outils destinés à améliorer le développement de programmes OCaml pour microcontrôleurs, dont en particulier *ocamlclean*, un outil permettant d'éliminer statiquement les allocations de fermetures inutilisées dans un programme, ainsi qu'un ensemble de simulateurs qui permettent le débogage simplifié des programmes réalisés.

La figure 1.7 est un diagramme de Venn qui représente les diverses technologies abordées dans cette section.

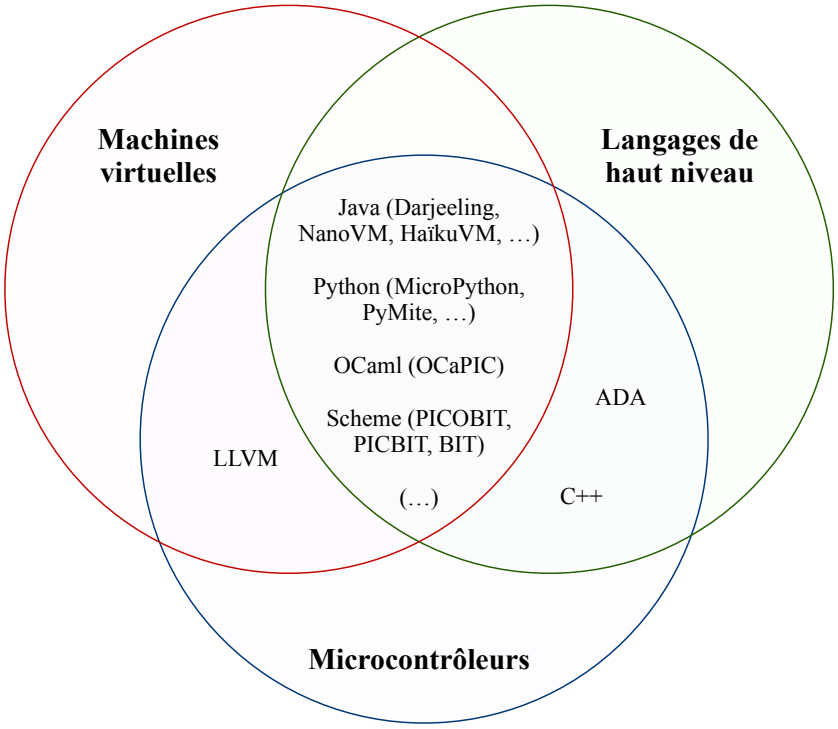



FIGURE 1.7 – Microcontrôleurs, machines virtuelles, et langages de haut niveau

Le tableau de la figure 1.8 condense les principales caractéristiques des approches présentées. Les approches à base de machines virtuelles possèdent plusieurs avantages intéressants pour la programmation de microcontrôleurs. Elles offrent des traits issus de langages de haut niveau, avec lesquels sont familliers les développeurs d'applications modernes, tout en permettant de potentiellement réduire l'empreinte mémoire des programmes grâce à la factorisation inhérente à la représentation des programmes sous forme de bytecode. Néanmoins, certaines de ces machines virtuelles sont parfois assez peu adaptées à une utilisation sur des microcontrôleurs à faibles ressources, qui n'offrent que quelques kilo-octets de RAM et moins de cent kilo-octets de mémoire flash. Les limitations mémoire peuvent restreindre l'utilisation de tous les aspects d'un programme. De surcroît, la plupart de ces solutions sont parfois assez peu portables : elles ciblent souvent une certaine gamme de microcontrôleurs, et il peut être difficile de les adapter à une architecture différente.

| Langage | Implémentation | VM / Interprète ? | Microcontrôleur (famille) |
|---|---|---|---|
| Java | JVM | oui | - |
| | Darjeeling | oui | AVR, MSP |
| | NanoVM | oui | AVR |
| | HaikuVM | oui | AVR |
| | MicroEJ | oui | ARM |
| Python | CPython | oui | - |
| | MicroPython | oui | STM32 |
| Scheme | Guile | oui | - |
| | BIT | oui | Motorola 68HC11 |
| | PICBIT | oui | PIC18 |
| | PICOBIT | oui | PIC18 |
| OCaml | ZAM | oui | - |
| | OCaPIC | oui | PIC18 |
| ADA | GNAT | non | STM32 |
| | AVR-Ada | non | AVR |
| C++ | avr-g++ | non | AVR |
| | IAR | non | AVR, MSP, STM32 |
| | MPLAB XC32++ | non | PIC32, SAM |
| C, Go, Rust, ... | LLVM | généralement non (représentation intermédiaire) | AVR, MSP |

FIGURE 1.8 – Implémentation de langages de programmations sur microcontrôleurs

Dans le cadre de nos travaux, nous considérons que l'approche adoptée par OCaPIC est la plus adaptée à la programmation de microcontrôleurs. En effet, la machine virtuelle du langage OCaml est assez légère pour supporter l'exécution de tous les traits du langage sur des microcontrôleurs assez limités en ressources mémoires, et le langage lui-même possède des aspects avantageux, comme une meilleure sûreté des programmes grâce au typage statique, ainsi qu'un système de gestion automatique de la mémoire. Néanmoins, à l'image de la plupart des solutions abordées dans cette section, OCaPIC est assez limité dans sa portabilité puisqu'il n'est accessible qu'à des microcontrôleurs de la gamme PIC18. Nous proposerons alors, dans le chapitre suivant, une machine virtuelle OCaml *générique*, destinée à être exécutée sur de nombreuses cibles différentes, tout en conservant une empreinte mémoire faible, et une vitesse de calcul satisfaisante pour un langage riche.

## 1.3 Programmation synchrone

Le rôle d'un microcontrôleur est d'être le *chef d'orchestre* d'un montage électronique : il se doit de réagir aux stimuli provenant des composants électroniques auxquels il est branché (capteurs, boutons ...) afin d'émettre des signaux à certains composants du circuit (écran LCD, effecteurs ...). Par exemple, à l'intérieur d'un clavier d'ordinateur, un microcontrôleur réagit dans un délai imperceptible aux divers appuis de touches réalisés par l'utilisateur. Les programmes pour microcontrôleurs nécessitent souvent que le matériel réagisse rapidement à des signaux d'entrée, et ce quel que soit l'ordre d'apparition de ces derniers : l'appui de certaines touches du clavier ne doit, par exemple, pas masquer l'appui simultané d'une autre. Ainsi, la programmation de microcontrôleur est inhéremment concurrente : les

composants logiciels d'un programme embarqué qui gèrent chaque signal issu de l'environnement du système doivent généralement donner l'impression de réagir *en même temps*.

Les systèmes embarqués, parfois critiques, exhibent ainsi des comportements concurrents, soumis à des contraintes temporelles plus ou moins strictes. De ce fait, de tels systèmes sont souvent qualifiés de *systèmes temps réel*.

### 1.3.1 Systèmes temps réel

Un système temps réel est un système informatique dans lequel le temps de réaction et de calcul des différents composants d'un programme (nommés *tâches*) sont soumis à des contraintes qui doivent être respectées. Ces tâches traitent de manière concurrente l'apparition des stimuli auxquels le programme est censé réagir. Ces stimuli peuvent apparaître, pendant l'exécution du programme, de façon périodique, ou sporadique.

Le réalisation de systèmes temps réel induit la notion d'*ordonnancement*, qui consiste à définir l'ordre dans lequel les différentes tâches du programmes vont être exécutées (généralement de manière cyclique), en tenant compte des contraintes qui leur sont associées (comme leur fréquence, leur priorité, ou leur échéance).

Dans le cas de tâches périodiques, l'ordonnancement d'un programme peut être réalisé statiquement, en amont de l'exécution du programme : il est alors possible d'utiliser des solutions logicielles, comme l'outil Cheddar [SLNM04], pour établir automatiquement (s'il en existe) un ordre d'exécution des différents tâches d'un programme compatible avec leurs contraintes temporelles. La figure 1.9 représente un exemple d'ordonnancement réalisé par Cheddar pour trois tâches périodiques différentes. Dans un système contenant uniquement des tâches périodiques, et sans préemption, l'exécution d'un programme peut simplement correspondre à des appels tour-à-tour aux fonctions représentant chaque tâche, en parcourant par exemple une table représentant la séquence d'ordonnancement.

Pour des tâches sporadiques, traitant l'apparition d'évènement ponctuels, l'ordonnancement ne peut être réalisé « *offline* » (lors de la conception du programme), puisque les instants d'apparition de ces évènements ne sont pas prédictibles. L'ordonnancement des différentes tâches d'un programmes est alors réalisé dynamiquement, au fur et à mesure de l'apparition des évènements auxquels le programme doit réagir. L'ordonnançabilité du système peut toutefois être garantie statiquement dès lors que le délai minimum entre deux apparitions d'évènements sporadiques de même type est connu.

Un programme temps réel correspond généralement à un système mêlant stimuli périodiques et évènements sporadiques. De ce fait, les systèmes temps réels font usage d'ordonnanceurs (*schedulers*) logiciels, inclus dans des *systèmes d'exploitation temps réel* (*Real-Time Operating System – RTOS*), comme FreeRTOS [GPPT16]), qui s'exécutent avec le programme, et donnent dynamiquement la main aux tâches par exemple selon leur niveau de priorité et l'apparition d'évènements.

Les procédés d'ordonnancement statiques ou dynamiques, ainsi que l'utilisation de systèmes d'exploitation temps réels, sont plutôt contraignants et parfois difficiles à mettre en place sur certains appareils : les ressources mémoires nécessaires à l'exécution des composants logiciels destinés à l'ordonnancement et à la création des tâches concurrentes ne sont pas négligables, et elles sont parfois indisponibles sur les microcontrôleurs que nous considérons (qui ne possèdent que quelques kilo-octets de mémoire). De plus, même lorsque l'ordonnançabilité d'un programme a été établie, des erreurs parfois difficilement prévisibles peuvent toutefois avoir lieu en raison des accès concurrents à des ressources partagées par plusieurs tâches d'un programme, via l'utilisation de primitives de synchronisation (exclusion mutuelle,

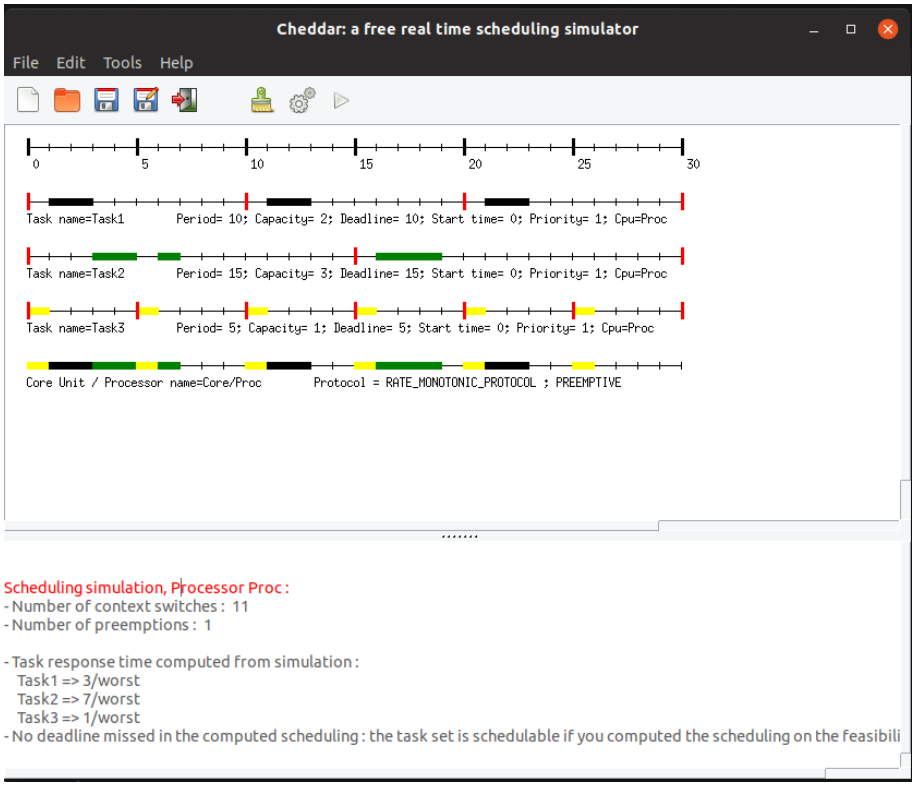



FIGURE 1.9 – Ordonnancement statique de trois tâches périodiques avec Cheddar

barrières de sychronisation, etc). C'est le cas, par exemple, des phénomènes d'inversion de priorité, dans lesquels une tâche de moindre priorité peut monopoliser une ressource partagée et ne jamais rendre la main à une tâche plus prioritaire. De tels phénomènes peuvent avoir des conséquences fâcheuses : le programme de *PathFinder*, la sonde spatiale de la NASA envoyée sur la planète Mars en 1997, a rencontré un problème d'inversion de priorité quelques jours après son atterrissage, ce qui entraîna plusieurs redémarrages intempestifs du logiciel [Jon97] [4].

Dans le contexte de cette thèse, nous adoptons un modèle de concurrence plus simple à aborder et capable de garantir que ce type de comportements imprévus ne peut pas avoir lieu. Ce modèle, la *programmation synchrone*, a fait ses preuves pour la programmation de systèmes embarqués critiques (avions, centrales nucléaires . . . ), et représente selon nous une solution adaptée pour la programmation concurrente de microcontrôleurs [VVC16]. Le modèle de programmation synchrone permet de représenter les aspects concurrents d'un programme sans nécessiter l'utilisation d'une machinerie logicielle responsable de la gestion des tâches du programme. En effet, la programmation synchrone peut être considérée comme un modèle de programmation de systèmes temps réel avec ordonnancement *statique* et *déterministe*, qui n'impose pas l'utilisation d'un ordonnanceur logiciel embarqué. Cela n'empêche cependant pas son utilisation dans des contextes où des ordonnanceurs sont utilisés, par exemple après compilation de composants synchrones vers des tâches périodiques exécutées sur des plateformes temps réel [PFB+11]. La légèreté induite par l'approche consistant à réaliser des programmes pouvant être exécutés sur du matériel exempt de tout système d'exploitation en fait toutefois un paradigme particulièrement compatible avec la programmation de microcontrôleurs à faibles ressources.

4. L'émission d'un correctif depuis la Terre permit néanmoins de régler le problème.

### 1.3.2 Hypothèse synchrone

La simplicité et la puissance d'expressivité du paradigme de programmation synchrone repose sur un principe d'abstraction, nommé *hypothèse synchrone*, qui stipule que le temps pris par les différents composants d'un programme pour calculer des valeurs de sortie à partir de valeurs en entrée est considéré comme nul. Cette hypothèse est comparable aux abstractions utilisées pour la réalisation de circuits (cette comparaison est reprise de [BB91]) : le temps pris par le signal électrique pour parcourir un ensemble de portes logiques est généralement ignoré lors de la confection de circuits électriques. De la même façon, dans la mécanique newtonienne, la vitesse de propagation d'un champ gravitationnel (c.-à-d. celle de la lumière) n'est pas considérée, et les interactions entre les corps sont vues comme instantanées. Dans un programme synchrone, les entrées du programme sont alors supposées instantanées avec ses sorties, et il est considéré que toutes les instructions du programme sont réalisées au sein d'un même instant logique, appelé *instant synchrone*. Bien sûr, pour que cette hypothèse soit valide, il faut soit être en mesure de garantir que le délai entre deux entrées est suffisant, soit envisager l'utilisation d'une mémoire tampon pour n'en perdre aucune. L'abstraction introduite par l'hypothèse synchrone est représentée dans la figure 1.10. Elle illustre le fait que la durée d'exécution réelle du code nécessaire au traitement des entrées et à la création des sorties (en haut sur la figure) n'est plus considéré par l'abstraction : les sorties $o_n$ sont produites « *en même temps* » que l'arrivée des entrées $i_n$ (en bas sur la figure).

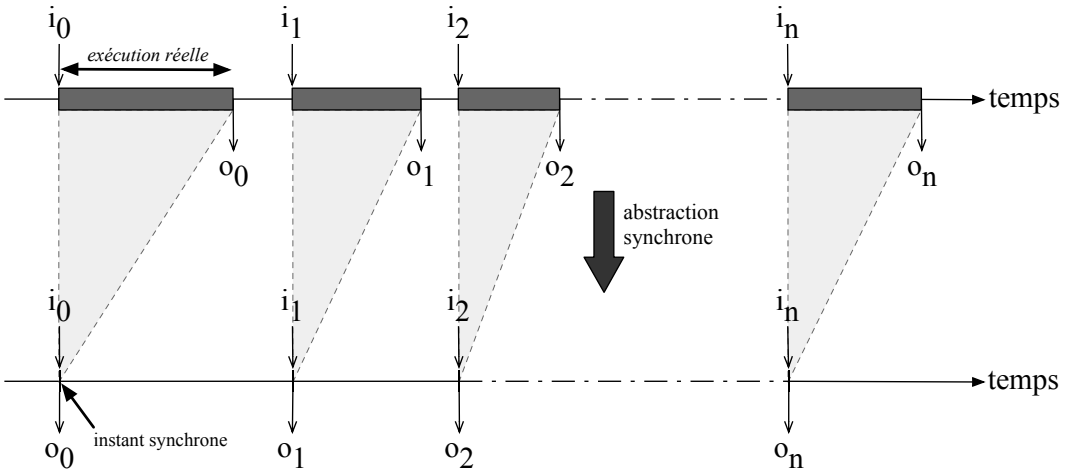



FIGURE 1.10 – L'hypothèse synchrone :
*Le temps de calcul des valeurs de sortie ($o_n$) d'un programme à partir de la valeur de ses entrées ($i_n$) est considéré comme nul.*

L'exécution d'un programme synchrone est une tâche cyclique qui consiste, à chaque instant du programme, à lire les entrées du programme (typiquement, des valeurs issues de l'environnement du système) et à calculer des valeurs de sortie qui seront émises à la fin de l'instant. L'hypothèse synchrone est vérifiée dès lors qu'à chaque instant l'intervalle d'apparition entre deux entrées du programme est supérieur au temps des calcul des valeurs en sortie. Le programme est ainsi synchronisé avec ses entrées, et il est possible de considérer que son temps de réaction est instantané.

L'hypothèse synchrone permet alors de simplifier le raisonnement sur les aspects concurrents d'un programme en supprimant, du point de vue du développeur, les considérations temporelles de synchronisation des différents composants logiciels d'une application.

### 1.3.3 Esterel : un langage de programmation synchrone impératif

Le modèle de programmation synchrone voit le jour au début des années 1980, et l'un des premiers langages implantant ce modèle est alors créé par une équipe composée de chercheurs de l'École des Mines et de l'INRIA Sophia-Antipolis. Ce langage, baptisé *Esterel* [BC84], repose sur un style impératif dans lequel les divers composants d'un programme communiquent par la diffusion de *signaux*, potentiellement valués, qui permettent l'échange d'informations entre les composants concurrents d'un programme. Le langage Esterel, dont la sémantique à *flot de contrôle* est dirigée par l'apparition d'évènements au cours de l'exécution d'un programme, fut initialement réalisé dans l'optique de programmer des robots industriels avec un haut niveau d'abstraction.

Un programme Esterel est constitué de plusieurs *modules*, chacun responsable d'une tâche à accomplir. Le corps d'un module est un code aux aspects impératifs contenant des opérateurs classiques comme l'opérateur de séquence (`;`) ou l'opérateur conditionnel (`if`), auxquels s'ajoutent des opérateurs synchrones comme un opérateur de parallélisation (`||`) ainsi que des opérateurs permettant d'émettre (`emit`) ou d'attendre (`await`) des signaux.

Un exemple courant de la programmation en Esterel est représenté dans la figure 1.11. Cet exemple correspond à la définition d'un module nommé `ABRO` dont le rôle est d'attendre, en parallèle, la présence d'un signal `A` et celle d'un signal `B` avant d'émettre un signal `O`. La présence du signal `R` réinitialise le comportement du programme.

```
module ABRO:

% Interface
input A, B, R;
output O;

% Corps
loop
   [ await A || await B ];
   emit O
each R
end module
```

Figure 1.11 – Exemple de programme Esterel : le module ABRO

Au delà de la programmation de systèmes embarqués industriels, le modèle événementiel d'Esterel est aujourd'hui utilisé afin de programmer plusieurs types d'applications. Par exemple, le langage de programmation *ReactiveML* [MP05] est une extension réactive synchrone du langage OCaml reprenant de nombreux aspects du langage Esterel. Ce langage, basé lui aussi sur un modèle d'émissions/réceptions de signaux, est destiné à la programmation d'applications riches tirant profit des traits concurrents offerts par le modèle de programmation synchrone. Les programmes ReactiveML couvrent une large gamme d'utilisations, parmi lesquelles des applications musicales, des jeux, des simulations de systèmes physiques, ou des applications algorithmiques plus classiques.

Par exemple, le programme ReactiveML de la figure 1.12, extrait du site officiel de ReactiveML [⚓21], réalise le parcours en largeur d'un arbre binaire en exécutant de façon concurrente le parcours du sous-arbre gauche et celui du sous-arbre droit de l'arbre considéré.

```
(* Definition of binary trees. *)
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a tree

(* Breadth first traveral. *)
let rec process iter_breadth f t =
  match t with
  | Empty -> ()
  | Node (x, l, r) ->
      f x;
      pause;
      run (iter_breadth f l) || run (iter_breadth f r)
```

FIGURE 1.12 – Exemple de programme ReactiveML

La programmation réactive synchrone se prête également bien à des applications destinées à des domaines en expansion, comme la programmation d'applications web. Par exemple, le langage *Pendulum* [SC16b] mêle les aspects algorithmiques du langage OCaml et un modèle synchrone réactif inspiré d'Esterel pour le développement d'applications multimédia riches pour le web, reposant sur le moteur *Js_of_OCaml* [VB14] qui permet la traduction du bytecode d'un programme OCaml vers une application Javascript.

### 1.3.4 Lustre et Signal : des langages de programmation synchrone déclaratifs

À une époque très proche de celle du développement d'Esterel, d'autres langages de programmation synchrone ont été développés, s'inspirant de modèles toutefois différents. En particulier, le langage *Lustre* [CPHP87] vit le jour lui aussi au début des années 1980, développé par des chercheurs du laboratoire grenoblois VERIMAG. Ce langage de programmation synchrone, à la différence d'Esterel, ne repose pas sur la réaction à des évènements ponctuels, mais sur un modèle de programmation à *flots de données*, permettant de représenter l'évolution de valeurs au fil du temps. Lustre fut initialement développé dans l'optique d'offrir un langage de programmation manipulable par des ingénieurs de contrôle habitués au formalisme déclaratif d'un modèle à flots de données [Hal05].

Sur le modèle du langage Lucid [AW77] [5], toutes les valeurs manipulées par un programme Lustre sont des flots de données : des séquences de valeurs qui peuvent varier au fil de l'exécution d'un programme. Chaque flot possède ainsi, par défaut, une valeur à chaque instant du programme, et cette valeur est susceptible de changer d'un instant synchrone à l'autre. De ce fait, une variable $x$ dans le langage Lustre correspond au flot de toutes les valeurs que prend $x$ d'instant en instant :

$$x \equiv (x_0, x_1, x_2, x_3, \ldots, x_i \ldots)$$

Une constante correspond alors à un flot invariant de valeurs :

$$2 \equiv (2, 2, 2, 2, \ldots, 2, \ldots)$$

Les opérateurs arithmétiques et logiques du langage s'appliquent point à point aux valeurs que prennent les flots lors de l'exécution d'un programme :

5. LUSTRE est à l'origine un acronyme de « LUcid Synchrone Temps RéEl »

$$x + y \equiv (x_0 + y_0, x_1 + y_1, x_2 + y_2, x_3 + y_3, \ldots, x_i + y_i, \ldots)$$

De manière semblable aux modules d'Esterel, le composant logiciel de base d'un programme Lustre est le *nœud*. Un nœud peut être considéré comme une fonction qui associe un flot de sorties à un flot d'entrées. Le corps d'un nœud est un système d'équations, semblables à des fonctions temporelles [CH86], qui déclarent des variables dont la valeur est susceptible de changer à chaque instant d'exécution. Chacun de ces flots est calculé dans le même instant synchrone, et le modèle synchrone flot de données de Lustre est alors une classe restreinte des réseaux de processus de Kahn (*Kahn Process Networks - KPN* [Kah74]) dans laquelle les communications peuvent être réalisées sans *buffers* [MPP10].

Le style déclaratif d'un programme Lustre est assez semblable à la structure d'un programme fonctionnel : chaque équation définit une variable dont la valeur dépend de l'instant, et le système d'équation du corps d'un nœud est un ensemble de déclarations de variables. Les considérations « impératives » d'exécution du programmes sont absentes de la sémantique du langage : le sens de lecture des équations ne traduit pas leur « ordre » de calcul.

Par exemple, la figure 1.13 correspond à la définition d'un nœud nommé `BOOLOPS`, qui reçoit en entrée deux flots booléens `A` et `B`, et calcule en sortie les flots `ANDB`, `ORB` et `XORB`, qui correspondent, respectivement, au calcul de $\mathtt{A} \wedge \mathtt{B}$, $\mathtt{A} \vee \mathtt{B}$ et $\mathtt{A} \oplus \mathtt{B}$.

```
node BOOLOPS (A:bool ; B:bool) returns (ANDB:bool; ORB:bool; XORB:bool);
let
  XORB = ORB and (not ANDB);
  ANDB = if A then B else false;
  ORB = if A then true else B;
tel;
```

FIGURE 1.13 – Un nœud Lustre qui calcule le « et », le « ou » , et le « ou exclusif » de ses entrées

Dans la version originelle du langage Lustre, les opérateurs disponibles sont les classiques opérateurs arithmétiques et logiques, ainsi que plusieurs opérateurs *temporels* :

— L'opérateur de mémoire `pre` qui permet d'avoir accès à la valeur d'un flot à l'instant précédent :

$$\mathtt{pre}\ \ a \equiv (nil, a_0, a_1, a_2, \ldots, a_{i-1}, \ldots)$$

— L'opérateur d'initialisation `->` qui permet de définir un flot à partir de la valeur d'un flots pour le premier instant et d'un autre flot de valeurs pour les instants suivants :

$$a\ \ \mathtt{->}\ \ b \equiv (a_0, b_1, b_2, ..., b_i, ...)$$

Il est alors possible de définir en Lustre la suite des entiers positifs de la sorte :

```
n = 0 -> pre n + 1
```

– Enfin, l'opérateur d'échantillonnage `when`, qui permet de ralentir l'exécution de composants du programme en conditionnant la présence de flots en fonction de valeurs booléennes (les *horloges*) :

$$(a \texttt{ when } x)_n \equiv \begin{cases} a_n \text{ si } x_n = \texttt{true} \\ \bot \text{ sinon} \end{cases}$$

Le symbole $\bot$ représente l'absence de valeur.
Chaque flot de données Lustre est implicitement couplé à une horloge qui est par défaut l'horloge globale, la plus rapide. On peut explicitement le restreindre à une horloge plus lente grâce à l'opérateur `when`, par exemple l'extrait de programme suivant force le flot `x` à n'être défini que quand `b` est vrai :

```
x = 4 when b;
```

Le tableau de la figure 1.14 représente une simulation de l'exécution de ces trois opérateurs :

| instant | 0 | 1 | 2 | 3 | 4 | 5 | ... |
|---|---|---|---|---|---|---|---|
| $c$ | true | false | true | true | false | true | ... |
| $x$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | ... |
| $y$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | ... |
| `pre` $y$ | *nil* | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | ... |
| $x$ `->` `pre` $y$ | $x_0$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | ... |
| $x$ `when` $c$ | $x_0$ | $\bot$ | $x_2$ | $x_3$ | $\bot$ | $x_5$ | ... |

FIGURE 1.14 – Les opérateurs temporels de Lustre

La compilation d'un programme Lustre nommée *compilation en boucle simple* consiste à transformer le programme en un programme séquentiel composé d'une seule boucle qui réalise la scrutation de la valeur de ses flots d'entrée en début d'instant synchrone, calcule la valeur de ses flots de sortie, et émet la valeur de ces derniers en fin d'instant.

### Signal

Basé sur un modèle déclaratif proche de Lustre, le langage Signal [GG87] est un langage de programmation synchrone à flots de données, destiné à la programmation de systèmes temps réel. Développé dès les années 1980 par une équipe du laboratoire IRISA de Rennes, il constitue le troisième grand langage synchrone développé en France à cette époque.

Signal est un langage relationnel qui combine en quelque sorte la vision flot de données de Lustre, et la vision évènementielle des signaux d'Esterel. En effet, un programme dans le langage Signal définit (sous une forme équationnelle comparable à celle de Lustre) des signaux, qui sont vus comme des séquences de données dont les valeurs ne sont pas systématiquement présentes à tous les instants d'exécution du programme. Pour représenter cette absence, Signal partage la notion d'horloge avec le langage Lustre : l'ensemble des instants pendant lesquels un signal a une valeur correspond à son horloge. Signal intègre

une notion de synchronicité permettant d'établir le fait que deux signaux sont sur la même horloge, et donc qu'ils sont *synchrones*, avec l'opérateur `^=`.

Signal définit des opérateurs temporels assez comparables à ceux de Lustre. Par exemple, l'opérateur `when` est semblable à celui de Lustre, et l'opérateur `$`, placé après le nom d'un signal, correspond à l'opérateur `pre` de Lustre. On peut ainsi par exemple définir un compteur `CPT` qui s'incrémente à chaque instant, et qui est remis à zéro dès l'arrivée d'un évènement `RESET` de la façon suivante :

```
(| CPT := 0 when RESET default CPT$ init 0 + 1 |)
```

L'opérateur `default` permet de définir une valeur pour CPT quand `RESET` est absent (il fait partie des opérateurs dits *polychrones* de par le fait qu'il s'applique à des signaux dont les horloges sont différentes), et le mot-clé `init` permet d'indiquer la valeur d'initialisation du signal pour le premier instant d'exécution.

La figure 1.15 correspond à un processus Signal (équivalent à un nœud Lustre) nommé `COUNTERS`, qui reçoit un signal en entrée (dénoté par le symbole « ? ») nommé `RESET`, et retourne deux signaux en sortie (dénotés par « ! ») appelés `CPT1` et `CPT2`. Le premier est un compteur croissant, initialisé à 0, le second un compteur décroissant, initialisé à 100, et tous deux peuvent être réinitialisés dès que le signal `RESET` est présent. L'expression `CPT1 ^= CPT2` permet de contraindre ces deux signaux à avoir la même horloge.

```
process COUNTERS =
      ( ? event RESET;
        ! integer CPT1;
          integer CPT2;
      )
      (| CPT1 := 0 when RESET default CPT1$ init 0 + 1
       | CPT2 := 100 when RESET default CPT2$ init 100 - 1
       | CPT1 ^=CPT2
      |);
```

FIGURE 1.15 – Un processus Signal

**Langages synchrones dérivés**

Quelques années après la réalisation de Lustre, le cœur du langage fut repris pour la définition d'un ensemble d'outils industriels, nommé *SCADE* [CPP17], permettant la programmation graphique de systèmes synchrones, via une représentation des éléments d'un programme sous forme de « planches ». Cette suite logicielle est aujourd'hui utilisée dans de nombreuses applications critiques, par exemple dans le système de contrôle des avions Airbus de la série A300. Cette utilisation repose sur la particularité de SCADE de posséder un générateur de code, nommé KCG, qui a reçu la certification DO-178C niveau A [⚓2], nécessaire à la production de code embarqué critique destiné au vol d'un avion. Cette certification libère le processus de développement de l'obligation de réaliser de nombreux tests afin de vérifier la correction du code généré, et représente ainsi un avantage de poids pour l'utilisation de SCADE [PAM+09]. Cette *factorisation* de la certification permet à des développeurs d'une application de n'avoir alors qu'à démontrer la traçabilité entre depuis le cahier des charges jusqu'au programme SCADE, et non pas jusqu'au code exécuté. De manière similaire, d'autres certifications de KCG permettent d'utiliser SCADE pour la programmation de systèmes à fortes exigences de sûreté, comme la certification européenne EN 50128 ayant trait aux transports ferroviaires, la norme internationale CEI 61508 portant

sur les composants programmables d'un système électronique, ou la norme CEI 60880 qui concerne les systèmes de contrôle des centrales nucléaires.

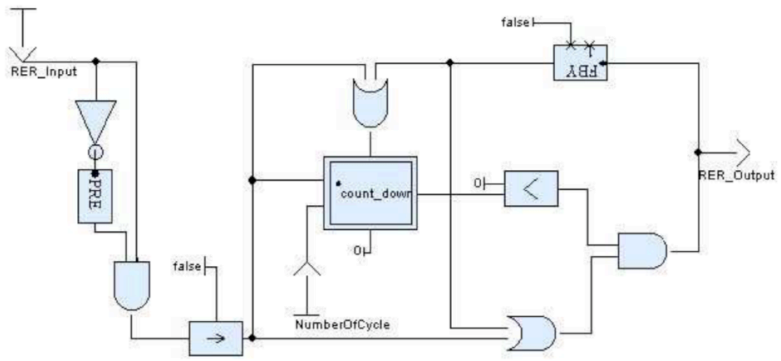



FIGURE 1.16 – Une planche SCADE
(image tirée de [CHP06])

De nombreux langages synchrones académiques, inspirés du modèle de langages à flots de données, et en particulier de Lustre (qui continue lui-même de s'enrichir dans sa version 6), ont été développés depuis son apparition. Parmi ces différents langages, le langage Lucid Synchrone [CP99, Pou06] est une extension du langage Lustre à l'ordre supérieur : en effet, les paramètres des nœuds dans Lucid Synchrone peuvent eux-mêmes être des nœuds synchrones. Lucid Synchrone représente une combinaison puissante de langages synchrones « à la Lustre » et de langages fonctionnels « à la ML » : les flots peuvent par exemple correspondre à des valeurs de types produits, utilisables avec un opérateur de filtrage. Plusieurs constructions de ce langage (signaux, analyse d'initialisation des flots, machines à états, système d'horloges . . . ) ont par ailleurs été reprises dans SCADE.

```
let node iter init f x = y where
   rec y = f x (init -> pre y)
```

FIGURE 1.17 – Un nœud Lucid Synchrone qui itère une fonction `f` sur un flot de valeurs `x`

D'autres langages synchrones visent à étendre le modèle de Lustre. Par exemple, le langage Heptagon, développé dans l'équipe PARKAS de l'École Normale Supérieure inclut une représentation optimisée des tableaux de valeurs [GGPP12], et inclut une extension (nommée BZR) qui permet d'intégrer la synthèse de contrôleurs discrets (SCD) à la compilation d'un programme synchrone [DRM11]. Le langage hybride Zélus [BP13], développé par la même équipe, est un langage qui dérive de Lustre et Lucid Synchrone mêlant des notions de temps discret et de temps continu grâce à l'utilisation d'équations différentielles ordinaires.

Dans le cadre de nos travaux, nous estimons que l'approche flots de données adoptée par ces langages synchrones est particulièrement adaptée au matériel et aux applications que nous visons. En effet, le fonctionnement physique du matériel est aisément représenté par des flots : chaque broche d'un microcontrôleur possède à tout moment une valeur (indiquant si elle est parcourue ou non par un courant

électrique), et chaque composant de l'application est alors représentable par un nœud synchrone qui permet de calculer, à chaque instant, des valeurs de courant en sortie à partir des stimuli électriques reçus en entrée par le microcontrôleur. La représentation schématique adoptée par le langage SCADE, proche de la représentation d'un circuit électronique, témoigne par ailleurs bien de cette proximité entre le modèle adopté par un langage comme Lustre et le modèle physique réel. De plus, le modèle de compilation classique de Lustre génère un code peu gourmand en ressources, spécialement adapté aux limitations du matériel que nous considérons.

## 1.4 Sûreté des programmes

La *sûreté* est la garantie qu'une chose indésirable particulière ne peut pas se produire [Lam77]. Plus précisément, la sûreté d'un programme désigne la garantie qu'un comportement « anormal », voire imprévu, ne puisse pas subvenir lors de son exécution [AS87]. Cette garantie peut concerner plusieurs aspects des comportements d'un programme, vérifier par exemple que certaines zones mémoires censées être protégées ne soient pas consultées lors de l'exécution d'une application [ACR+08], ou bien que des erreurs liées au déréférencement illicite de pointeurs nuls ne puissent se produire pendant qu'un programme s'exécute. À ce titre, les garanties de sûreté permettent à un développeur d'assurer que son programme ne peut atteindre des états qui ne partagent pas une même caractéristique désirée (telle que l'absence de *runtime errors*).

De telles garanties sont précieuses dans le cadre de la programmation de systèmes embarqués, où un comportement indésirable du programme peut mener à des conséquences désastreuses [WDS+10]. Un système embarqué critique dont le programme réagit de manière imprévue peut provoquer des accidents graves de personnes, qu'il est très important d'éviter. Pourtant, les modèles de programmation de bas niveau généralement utilisés pour la programmation de microcontrôleurs ne proposent que peu de garanties permettant d'assurer la cohérence du comportement du programme. Des erreurs qui pourraient être détectées statiquement peuvent passer inaperçues lors de la compilation dans des langages « traditionnels ». Par exemple, la faiblesses des vérifications concernant le typage de données d'un programme C permet de réaliser des opérations incongrues, et souvent involontaires, comme la multiplication d'un nombre entier par la valeur d'un caractère.

L'utilisation de langages de programmation de plus haut niveau permet souvent de détecter statiquement de nombreuses erreurs qui peuvent apparaître dans un programme, et ainsi de refuser ces programmes incorrects dès la compilation (quitte, bien sûr, à refuser certains programmes néanmoins corrects). Notre intérêt se porte en particulier, dans ce manuscrit, sur les garanties liées à la sûreté du typage, au calcul du temps d'exécution pire cas des programmes, ainsi qu'aux méthodes permettant de spécifier formellement un langage et de développer la métathéorie associée.

### 1.4.1 Typage statique

De nombreux langages de programmation de haut niveau, comme Haskell, Java, ou OCaml, implantent un mécanisme de *typage statique*. Ce dernier consiste en l'association d'un type (c'est-à-dire une information concernant la nature des valeurs portées par une variable ou un objet) à chaque identifieur dès la compilation d'un programme [Pie02]. Ce mécanisme permet alors de vérifier, avant l'exécution du programme, les erreurs liées à l'utilisation d'opérateurs ou fonctions incompatibles avec les types des données qui leur sont appliquées. Ainsi, le typage statique permet d'augmenter la sûreté du typage

en excluant des programmes incorrects dès la compilation. De ce fait, l'utilisation d'un langage de programmation statiquement typé comme OCaml est un avantage important pour la programmation de systèmes embarqués qui peuvent parfois être critiques : la détection supplémentaire d'erreurs de typage lors de la compilation permet d'éviter certaines réactions physiques inadaptées des programmes comme la mise en alimentation de composants électroniques dangereux (comme des résistances chauffantes, qui peuvent dépasser 200 degrés Celsius lors de leur mise sous tension). Nous verrons dans le chapitre suivant, en section 2.3.3, un exemple qui permet d'utiliser des traits avancés de typage (à l'aide de GADT - *Generalized Algebraic Data Types*) afin de représenter les interactions de bas niveau grâce à l'utilisation du système de typage statique expressif d'OCaml.

De la même façon, les systèmes de types spécifiques à certains langages de programmation non généralistes, comme Lustre, permettent d'augmenter d'autant plus la sûreté des programmes. En effet, le système de type des horloges synchrones de Lustre et ses dérivés permet d'expliciter les conditions qui régissent la présence de certaines valeurs, et ainsi d'empêcher statiquement la consultation d'une valeur d'un flot *absent*. De surcroît, la montée en abstraction apportée par les langages de programmation synchrone permet de vérifier des garanties supplémentaires lors de leurs diverses phases de compilation, comme le contrôle de la *causalité* d'un programme, qui consiste en la possibilité de séquentialiser statiquement les différents composants concurrents exécutés par le programme [BCE+03].

### 1.4.2 Temps d'exécution pire cas

L'hypothèse synchrone, sur laquelle reposent le langage Lustre et ses dérivés, n'est valide qu'à condition que le temps de calcul des sorties d'un programme soit plus rapide, à tout instant d'exécution du programme, que l'intervalle de temps entre l'apparition (ou la scrutation) de ses entrées. Dans le cadre de la programmation de systèmes temps réel stricts, il est alors essentiel, afin de respecter les échéances, de vérifier que le temps nécessaire au programme pour calculer des valeurs de sortie à partir de quelconques valeurs en entrée ne dépasse pas, dans le pire des cas (c'est-à-dire le cas où il est le plus lent), une borne de temps maximale. Ce temps d'exécution dans le pire des cas (*Worst-Case Execution Time – WCET*) se doit alors, afin qu'aucune entrée du programme ne soit ignorée pendant son exécution, d'être inférieur à la plus petite période qui sépare l'apparition des dites entrées, ou au délai minimal entre l'arrivée de deux entrées dans le cas d'évènements sporadiques.

Longtemps, la pratique plus courante pour déduire le temps d'exécution maximal d'un programme reposait sur des mesures empiriques : le programme était exécuté un certain nombre de fois sur le matériel désiré, et la mesure de temps la plus grande était considérée comme une borne réaliste au temps d'exécution du programme [WEE+08]. Cette méthode a toutefois des limitations évidentes : l'exhaustivité des tests étant souvent impossible, il est probable qu'en situation réelle d'exécution, un certain ensemble de valeurs en entrées particulier entraîne une durée de réaction du programme supérieure à celle estimée au moment des tests effectués en amont. Des techniques plus sûres permettent désormais de déduire le WCET d'un programme en calculant le plus grand nombre de cycles machines nécessaires à son exécution. Elles appliquent pour la plupart des analyses statiques permettant de générer un *graphe de flot de contrôle* représentant l'exécution d'un programme. Ce graphe est ensuite analysé, afin de déduire l'ensemble des chemins d'exécution *valides* de ce programme, et d'estimer le coût du chemin d'exécution valide le plus long. Différentes méthodes d'analyse statique existent pour réaliser cette estimation, comme par exemple la méthode d'énumération implicite des chemins d'exécution (*Implicit Path Enumeration Technique* ou *IPET*) [LM97]. Cette technique consiste à représenter les composantes du programme sous

forme de contraintes linéaires entières, et de les utiliser pour calculer le WCET, vu comme une expression linéaire à maximiser. Cette maximisation est réalisée à l'aide de solveurs de programmes linéaires entiers (*Integer Linear Programming*, ou *ILP*) ou à l'aide de techniques de programmation par contrainte. Plusieurs outils implantent cette technnique comme par exemple l'outil OTAWA [BCRS10], développé à l'institut de recherche en informatique de Toulouse (IRIT), qui permet d'estimer le temps d'exécution pire cas de programmes sur des architectures variées.

D'autres méthodes d'estimation du WCET reposent par exemple sur l'analyse de la structure du code source du programme : un arbre représentant la syntaxe du programme est parcouru dans le but de grouper, dans le graphe de flot de contrôle, plusieurs nœuds qui correspondent aux composantes syntaxiques du programme (boucle, conditionnelle, etc.) en un même nœud, et à répéter ce procédé pour à terme en déduire le coût du programme. Par exemple, l'outil Heptane [CP01] réalise une telle analyse et permet d'estimer le WCET de programmes exécutés sur architectures ARM et MIPS.

La figure 1.18 représente la différence entre le pire temps déduit par une succession de mesures sur un programme (qui peut être une sous-estimation du réel pire temps d'exécution), celui garanti par des outils d'estimation du WCET comme Heptane ou OTAWA (qui peuvent au contraire en sur-estimer la durée), et le pire temps d'exécution réel.

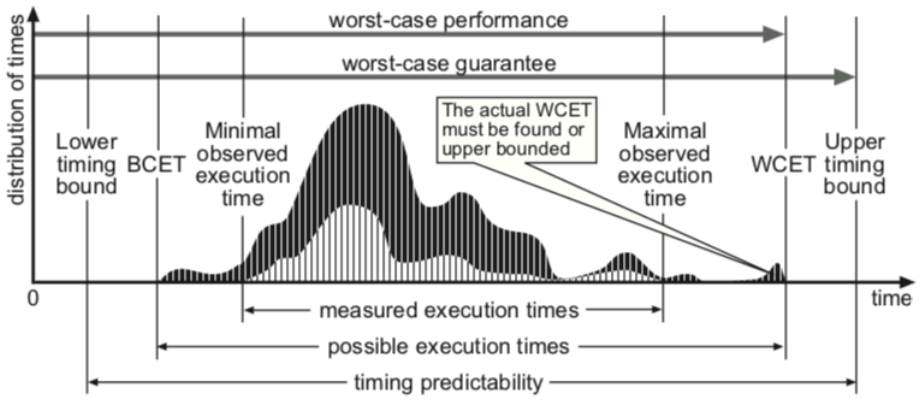



FIGURE 1.18 – Pire temps d'exécution mesuré, garanti, et réel
(image tirée de [WEE+08])

Le calcul du WCET d'un programme peut être grandement complexifié par l'architecture du processeur avec lequel un programme est exécuté. En effet, les différents niveaux de caches dont sont dotés les processeurs récents, ainsi que les optimisations modernes qu'ils implantent (pipeline, prédiction de branchements, etc.), rendent le temps de calcul d'une instruction variable, et dépendant de l'historique d'exécution des autres instructions du programme. Il n'est donc pas possible, sur de telles architectures modernes, de calculer le WCET de programme en considérant indépendamment les coûts de ses instructions, et un modèle abstrait du comportement du processeur doit être utilisé pour réaliser l'estimation du WCET des programmes. Cependant, les microcontrôleurs que nous visons ici n'implantent pas ces systèmes sophistiqués qui peuvent rendre le calcul du WCET si complexe : un microcontrôleur de la gamme ATmega328, par exemple, ne possède pas de système de caches, son processeur n'implante pas de prédiction de branchement, et est limité à un pipeline scalaire à deux niveaux permettant simplement de charger la prochaine instruction lors du traitement de l'instruction actuelle. Cette limitation a pour effet de simplifier grandement le calcul du temps d'exécution pire cas car le temps d'exécution de chaque instruction peut alors être considéré séparément. Nous aborderons en particulier, dans le chapitre 6,

une méthode de calcul du WCET d'un programme pour microcontrôleur par l'analyse des instructions bytecode qui le composent.

L'estimation de WCET des programmes synchrones est habituellement réalisée sur le programme séquentiel généré suite à la compilation du programme. Dans le cas de Lustre, ce programme prend la forme d'une boucle unique, sans récursion ni allocation dynamique. Cette simplicité du code généré permet alors de borner simplement le temps de calcul d'un tour de boucle via des solutions logicielles automatisées.

### 1.4.3 Spécifications formelles et métathéorie

Une spécification formelle définit, dans un langage non-ambigü, les diverses règles du comportement d'un système [Spi89]. Dans le cadre de la description d'un langage de programmation, ces règles décrivent par exemples les systèmes de types appropriés, ou la sémantique du langage.

Des outils de *spécification profonde* [ABC+17], comme les assistants de preuve Coq [Tea19] et Isabelle/HOL [NPW02], ainsi que des méthodes formelles comme la méthode B [Abr96], utilisée dans de nombreuses applications industrielles critiques (comme dans le cadre de l'automatisation des lignes 1 et 14 du métro parisien), permettent ainsi de réaliser la spécification formelle de systèmes dans le but d'en certifier la correction. À partir de ces spécifications formelles, les outils sus-cités peuvent générer du code exécutable dans un langage de programmation généraliste : par exemple, Coq permet l'extraction de fonctions OCaml (entre autres langages), Isabelle/HOL permet la génération de code Haskell et d'autres langages de haut niveau. Par une procédure de raffinement successif de la spécification il est alors possible de réaliser, grâce aux outils de spécification, des programmes exécutables certifiés corrects. Parmi ces derniers, on peut citer les exemples du micro-noyau seL4 [KEH+09] dont la correction de l'implantation a été prouvée à l'aide de Isabelle/HOL, le compilateur certifié CompCert [KLW14] (prouvé avec Coq) qui permet la compilation de programmes C pour de nombreuses architectures de processeurs (ARM, RISC-V, x86 . . . ), le projet CakeML [KMNO14] qui inclut un compilateur spécifié et vérifié (à l'aide de l'assistant de preuve HOL4 [SN08]) pour un langage de programmation fonctionnelle, ou encore une formalisation en Coq du langage JavaScript nommée JSCert [BCF+14] et son interprète JSRef dont il a été prouvé qu'il respecte la spécification formelle.

Par ailleurs, le langage Lustre a été conçu dès son origine avec une sémantique définie formellement, illustrant la volonté de ses créateurs d'appliquer ce langage à des systèmes qui nécessitent de garantir un certain niveau de sûreté. De nombreux travaux ont ainsi porté sur la formalisation et le développement de compilateurs certifiés corrects pour la réalisation de programmes respectant la sémantique de Lustre [BBD+17, BBP18, Aug13]. Ces travaux consistent en la formalisation d'un langage « à la Lustre » et en la certification à l'aide de Coq d'un générateur de code qui soit compatible avec CompCert afin de proposer une chaîne de compilation certifiée correcte depuis le programme Lustre jusqu'au code machine.

## Conclusion du chapitre

La vocation principale de cette thèse est de réaliser la communion de tous les aspects présentés dans ce chapitre. En effet, en raison des avantages offerts par des langages de haut niveau, nous désirons exécuter des programmes écrits dans un tel langage sur des microcontrôleurs dont les ressources matérielles sont faibles, voire très faibles. L'utilisation du langage OCaml nous semble ainsi être une approche pertinente, tant par la richesse du langage que la légèreté de son modèle et la sûreté qu'il apporte grâce à son typage statique strict, qui permet de vérifier qu'aucune erreur liée au typage n'adviendra lors de l'exécution des programmes. Ainsi, nous décrirons dès le chapitre suivant une implémentation portable de la machine virtuelle OCaml, nommée OMicroB. Cette machine virtuelle est adaptée à une exécution sur des microcontrôleurs variés, dotés parfois de moins de 4 kilo-octets de RAM. Les applications pour microcontrôleurs étant inhéremment concurrentes, de par leurs interactions multiples avec leur environnement, nous proposerons alors OCaLustre, une extension synchrone du langage OCaml, compatible avec la machine virtuelle sus-citée. Cette extension permettra de faire le pont entre les avantages liés à l'utilisation d'un langage généraliste de haut niveau, et un modèle de concurrence léger et adapté aux ressources des microcontrôleurs. Nos solutions profiteront des garanties offertes par ces deux mondes, et des analyses statiques pourront par ailleurs être réalisées directement sur le bytecode d'un programme OCaml, offrant ainsi un niveau d'abstraction accru pour le développement d'applications profitant de la portabilité de cette approche. Nous décrirons à ce sujet une analyse permettant de calculer le temps d'exécution pire cas d'un instant synchrone d'un programme OCaLustre, permettant d'assurer le non-chevauchement des activations et ainsi de valider l'hypothèse synchrone. De surcroît, bien que la preuve de correction de la chaîne complète de compilation d'un langage de programmation dépasse le cadre de cette thèse, nous aborderons au cours de ce manuscrit de nombreux aspects formels pour la description d'OCaLustre, dont nous établirons une spécification formelle. Certaines preuves de propriétés métathéoriques réalisées à l'aide de Coq seront aussi proposées.

# 2 OMicroB : Une machine virtuelle OCaml générique

Les travaux initiés par la machine virtuelle OCaPIC semblent constituer une solution prometteuse et adaptée pour la programmation de microcontrôleurs dont les ressources sont limitées. En effet, l'utilisation du langage OCaml permet d'offrir aux techniques de programmation de microcontrôleurs un langage de développement moderne, qui implante divers paradigmes de programmation, tels que la programmation fonctionnelle ou la programmation orientée objet, sans omettre le plus classique modèle de programmation impérative. Plus sûr que le traditionnel couple assembleur/C, ce langage de programmation de haut niveau permet la détection anticipée de certaines erreurs dans les programmes grâce à son système de typage statique. De plus, OCaml offre des outils permettant de simplifier et garantir le développement de programmes, par exemple par l'intermédiaire de son système de gestion automatique de la mémoire (*garbage collection*) ou son système d'inférence de type permettant au développeur de s'abstraire d'un grand nombre d'annotations de typage superflues, tout en conservant la sûreté du typage statique. Néanmoins, OCaPIC n'est pas adapté à la variété des modèles de microcontrôleurs présents sur le marché : la décision de développer cet outil en grande partie dans le langage d'assemblage des microcontrôleurs PIC18 permet d'obtenir des performances (en vitesse d'exécution des programmes) importantes, mais limite grandement la portabilité de cette machine virtuelle, puisqu'elle n'est exécutable que par du matériel de cette famille. Nous considérons qu'en raison de la multiplicité des familles et architectures de microcontrôleurs utilisées par les industriels et les amateurs (microcontrôleurs AVR, PIC16, PIC32, ARM, ...), une solution *générique et portable*, pouvant être exécutée sur de nombreuses cibles différentes, constituerait un avantage important pour l'adoption de méthodes de programmation de haut niveau dans le domaine de la programmation de microcontrôleurs. Nous proposons ainsi OMicroB, une machine virtuelle *générique*, destinée à l'exécution de programmes OCaml sur du matériel varié et disposant de peu de ressources. Cette machine virtuelle, réalisée en collaboration avec Benoît Vaugon (concepteur d'OCaPIC), tire justement avantage de l'omniprésence du langage C pour la programmation traditionnelle de microcontrôleurs, qui assure l'existence presque systématique de compilateurs C pour chaque matériel visé. Ainsi, une grande partie de la machine virtuelle OMicroB est écrite en langage C, que nous utilisons en tant qu'*assembleur portable* qui permet un portage simplifié d'OMicroB sur des machines parfois très différentes. En raison des contraintes fortes concernant les tailles mémoires des appareils considérés, nous nous attachons à fournir une implémentation finement contrôlée pour être exécutable sur du matériel à peu de ressources. Un effort particulier est ainsi appliqué pour la réduction de l'empreinte mémoire des programmes, par l'intermédiaire d'optimisations variées. Ce chapitre rappelle en premier lieu les principaux aspects à la fois syntaxiques et sémantiques du langage de programmation OCaml, avant de décrire en détail l'implémentation de la machine virtuelle OCaml standard, pour ensuite décrire notre implantation d'une machine virtuelle destinée à être exécutée sur du matériel varié bénéficiant de peu de ressources.

## 2.1 Le langage OCaml

Cette section est destinée à familiariser le lecteur non coutumier avec le langage OCaml avec la syntaxe et les fonctionnalités essentielles du langage. Nous y présentons les aspects principaux du langage, qui seront utilisés dans les exemples de ce manuscrit. Tout lecteur déjà familier du langage OCaml peut passer directement à la section suivante. Nous réalisons ici un tour d'horizon non exhaustif des principaux traits de programmation de haut niveau qui permettent de réaliser dans un style expressif des applications riches et offrant une sûreté accrue, grâce au système de typage d'OCaml, ainsi qu'à son mécanisme de gestion automatique de la mémoire.

**Valeurs fonctionnelles :** Le noyau du langage OCaml est fonctionnel et impératif, il permet la manipulation en tant que valeurs des fonctions définies dans le langage. Par exemple, la fonction `compose` calcule l'application à une variable $x$ de la composition $(f \circ g)$ :

```
let compose f g x = f (g x)
```

Cette notation est une alternative à une notation plus explicite, qui définit une fonction via le mot-clé **`fun`**. La définition de la fonction `compose` peut donc aussi être réalisée de la sorte :

```
let compose = (fun f g x -> f (g x))
```

Cette même fonction peut également être représentée sous une forme *curryfiée*. La *curryfication* consiste à transformer une fonction à $n$ arguments en une fonction qui reçoit un seul argument et retourne une chaîne de $n-1$ fonctions qui ne reçoivent à leur tour qu'un seul argument, et ainsi de suite jusqu'à retourner la valeur calculée par le corps de la fonction d'origine [Rey98]. Ainsi, la définition suivante de la fonction `compose` est sémantiquement identique aux deux définitions précédentes :

```
let compose = (fun f -> (fun g -> fun x -> f (g x)))
```

OCaml permet l'utilisation de *lambda-expressions*, des fonctions anonymes utilisables dans le corps d'autres fonctions. Par exemple, à l'aide de la fonction `compose`, la fonction suivante utilise une fonction anonyme (définie avec le même mot-clé **`fun`**) pour multiplier par deux son paramètre $x$, puis affiche le résultat (grâce à la primitive `print_int`) :

```
let double_print x = compose print_int (fun x -> x * 2)
```

Il est à noter que les valeurs fonctionnelles manipulées dans les programmes, aussi appelées *fermetures*, peuvent contenir un environnement qui leur est propre. Ainsi, la fonction `retourne_fermeture` suivante calcule, à partir de $x$, une fermeture qui contient la valeur de $x$ dans son environnement et est en attente d'un paramètre $y$ pour calculer la valeur $x + y$.

```
let retourne_fermeture x = (fun y -> x + y)
```

Le type d'une fonction est un *type flèche* de la forme *type du paramètre* → *type de la valeur en sortie*. Puisque `retourne_fermeture` est une fonction qui prend en paramètre un entier et retourne une fonction qui prend en paramètre un autre entier et retourne un entier (car l'opérateur + ne s'applique qu'à des entiers et produit un entier [1]), son type est :

$$int \rightarrow (int \rightarrow int)$$

Le « parenthésage » des types de fonctions est associatif à droite, ce qui permet d'alléger cette notation de la sorte :

$$int \rightarrow int \rightarrow int$$

**N-uplets :** Une fonction peut manipuler des n-uplets de valeurs. Par exemple, la fonction `add` calcule la somme des trois éléments contenus dans le triplet qui lui est passé en paramètre :

```
let somme_triplet (x,y,z) = x+y+z
```

La fonction `plus_moins`, quant à elle, retourne un couple correspondant à la somme et à la différence des valeurs qui lui sont passées en paramètre :

```
let plus_moins x y = (x+y,x-y)
```

**Typage statique, polymorphisme, et inférence des types à la compilation :** La définition de variables et fonctions, dans le langage OCaml, n'impose pas en général au développeur d'annoter chacune de ces définitions avec son type. Les types des valeurs sont en effet inférés par le compilateur du langage, et ce dernier vérifie statiquement que les programmes réalisés sont cohérents avec le type ayant été déduit à la compilation lors d'une phase de *typage*.

Par exemple, le type de la fonction `somme_triplet`, inféré par le compilateur, est alors $(int * int * int) \rightarrow int$ [2] en raison du fait que les trois éléments du triplet d'entrée sont additionnés avec l'opérateur d'addition entière « + ». De la même façon, le compilateur infère pour la fonction `plus_moins` le type $int \rightarrow int \rightarrow (int * int)$. Toute utilisation de ces fonctions avec des valeurs incompatibles avec leurs types (par exemple en passant en paramètre à `somme_triplet` un triplet de flottants, ou en utilisant le résultat de `plus_moins` comme un simple entier) sera détectée et refusée lors de la compilation du programme.

Le typage statique strict du langage permet ainsi d'accroître la sûreté des programmes, en assurant qu'aucune erreur liée à une incohérence dans le typage des valeurs d'un programme ne puisse apparaître lors de son exécution, et l'inférence des types permet d'épargner au développeur le fait de devoir annoter des valeurs avec leur type le plus général. La notion de *type le plus général* provient du fait qu'OCaml

1. C'est l'opérateur « `+.` » qui permet d'additionner deux flottants.
2. Le symbole $*$ dans le type d'un n-uplet permet de séparer les différents types des éléments de celui-ci.

implante un système de types avec polymorphisme paramétrique, qui permet par exemple de représenter des fonctions applicables à des arguments de types divers. Par exemple, la fonction `identity` suivante retourne simplement son paramètre $x$ :

```
let identity x = x
```

Le type de cette fonction, tel qu'il est inféré par OCaml, est alors $\alpha \to \alpha$ (aussi noté `'a -> 'a`), signifiant que cette fonction reçoit un argument d'un type quelconque ($\alpha$) et qu'elle retourne un argument du même type.

De la même façon, le compilateur infère, pour la fonction `compose` présentée en début de section, le type suivant :

$$(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$$

**Types sommes et filtrage par motif :** Un programme OCaml peut définir de nouveaux types de données sous la forme de *types sommes* (qui généralisent les types énumérés). La définition d'un tel type revient à lui attribuer un nom, ainsi que divers constructeurs.

Par exemple, le type énuméré `couleur` permet de représenter la couleur d'une carte à jouer :

```
type couleur = Pique | Coeur | Carreau | Trefle
```

Il est également possible de définir des constructeurs paramétrés, qui permettent en particulier de représenter des types récursifs. Par exemple, le type `couleur_list` permet de représenter une liste de couleurs comme étant la liste vide (`Nil`), ou une liste construite (avec le constructeur `Cons`) à partir d'une couleur (la tête de la liste), et d'une liste de couleurs (sa queue).

```
type couleur_list = Nil | Cons of couleur * couleur_list

(* la liste [Coeur; Trefle; Trefle] : *)
let l_ex = Cons(Coeur,Cons(Trefle,Cons(Trefle,Nil)))
```

Pour manipuler les valeurs d'un type somme, il peut alors être réalisé un *filtrage par motif* sur les constructeurs du type concerné à l'aide de l'instruction *match / with*. Par exemple, la fonction récursive `longueur` calcule la longueur d'une liste de couleurs :

```
let rec longueur l =
  match l with
  | Nil -> 0
  | Cons (_,l') -> 1 + longueur l'

let l = Cons(Coeur,Cons(Trefle,Cons(Trefle,Nil))) in
  print_int (longueur l) (* affiche ''3'' *)
```

Il est à noter que dans un motif, le caractère _ sert à exprimer « n'importe quelle valeur » : il est ici par exemple utilisé avec `Cons (_,l')`, dans le « `match` » de la fonction `longueur` pour réaliser le filtrage d'une liste non vide dont la valeur du premier élément n'a pas d'importance sur la sémantique de la fonction en question, et n'a pas besoin d'être nommée.

Un type peut par ailleurs être défini en exploitant le mécanisme de polymorphisme paramétrique. Par exemple, le type qui représente une liste dont les éléments sont d'un type quelconque peut être ainsi défini :

```
type 'a list = Nil | Cons of 'a * 'a list

(* la liste [Coeur; Carreau; Carreau] de type "couleur list"  *)
let liste_de_couleurs = Cons(Coeur,Cons(Carreau,Cons(Carreau,Nil)))
(* la liste [1;2;3;4] de type "int list"  *)
let liste_d_entiers = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
(* la liste [2.33; 4.55; 6.77] de type "float list"  *)
let liste_de_flottants = Cons(2.33,Cons(4.55,Cons(6.77,Nil)))
```

Il est donc possible grâce à une telle définition de créer (entre autres) des listes d'entiers, de flottants, de valeurs d'un type somme, de fonctions (toutes du même type), ou même des listes de listes. Il est à noter toutefois que les listes générées doivent être homogènes : le type `'a list` permet de définir la structure d'une liste qui contient des valeurs d'un certain type, mais tous les éléments d'une *même* liste doivent être du *même* type.

Par ailleurs, ce type `'a list` des listes polymorphes est prédéfini par la bibliothèque standard du langage OCaml. Le constructeur `Nil` y est noté `[]` et le constructeur `Cons` correspond à un opérateur infixe `::`. La liste `Cons(1,Cons(2,Cons(3,Nil)))` peut ainsi être écrite `1::2::3::[]`, ou même sous la forme `[1;2;3]` grâce à du *sucre syntaxique*, fourni par le compilateur du langage.

**Types enregistrements :** Des valeurs composées de plusieurs éléments distincts peuvent être représentées à l'aide de types *enregistrements* (ou de *types produits*). Un type enregistrement est défini en donnant le nom et le type de chaque élément contenu dans la valeur créée. L'exemple suivant définit un type enregistrement pour représenter un point sur un plan (avec deux coordonnées : $x$ et $y$), ainsi qu'une fonction de calcul de la distance entre deux points `p1` et `p2` :

```
type point = { x : float ; y : float }

let distance p1 p2 =
  let x_diff = p1.x -. p2.x in
  let y_diff = p1.y -. p2.y in
  sqrt (x_diff *. x_diff +. y_diff *. y_diff)
```

**Mutabilité et programmation impérative :** Le contenu des champs d'un type enregistrement peut être déclaré comme étant modifiable au cours de l'exécution d'un programme grâce au mot-clé **`mutable`**. La modification de la valeur d'un champ mutable s'effectue avec l'opérateur <-.

Par exemple, le code suivant déclare un point dont les champs sont mutables, ainsi qu'une fonction qui réalise la translation d'un point $p$ selon un vecteur de coordonnées $(u, v)$ en modifiant *en place* les valeurs contenues dans les champs $x$ et $y$ de $p$ (l'opérateur infixe `;` permet de réaliser une séquence d'actions) :

```
type point = { mutable x : float ; mutable y : float }

let translate p (u,v) =
  p.x <- p.x +. u;
  p.y <- p.y +. v
```

Généralement, à l'instar de plusieurs langages fonctionnels (comme Haskell ou Scheme), les variables sont immutables en OCaml. En effet, une fois qu'une variable est déclarée, sa valeur ne peut pas être modifiée au cours de l'exécution du programme. Néanmoins, un type de données particulier, la *référence*, permet de représenter des variables dont la valeur peut être modifiée pendant l'exécution du programme, et ainsi de réaliser des programmes aux traits impératifs. Une référence est définie avec le mot-clé **`ref`**, son contenu est accessible avec l'opérateur `!`, et peut être modifié avec l'opérateur `:=`.

Le programme suivant définit un compteur `cpt`, utilisé pour réaliser 10 tours d'une boucle dans laquelle la valeur contenue dans `cpt` est affichée, puis incrémentée.

```
let loop =
   let cpt = ref 0 in
   while (!cpt < 10) do
     print_int !cpt;
     cpt := !cpt + 1
   done
```

En réalité, le type d'une référence correspond à celui d'un enregistrement qui contient un unique champ `content`, déclaré comme étant *mutable* :

```
type 'a ref = { mutable content : 'a }
```

Les autres structures de données élémentaires du langage étant mutables sont les tableaux[3]. Par exemple, le programme suivant utilise une boucle « *for* » (telle qu'il en existe dans de nombreux langages de programmation) afin d'ajouter 2 à chaque élément du tableau $t$ (défini avec la syntaxe `[| ... |]`) en accédant à la case $i$ du tableau $t$ via la notation `t.(i)` :

3. Historiquement les chaînes de caractères étaient également mutables, mais ce comportement est désormais déprécié. Des séquences d'octets représentés par un module *Bytes* peuvent être utilisés en remplacement.

```
let t = [| 1 ; 2 ; 3 ; 4 ; 5 |] in
for i = 0 to 4 do
  t.(i) <- t.(i) + 2
done
```

**Exceptions :** Comme dans de nombreux langages de programmation, OCaml implante un mécanisme d'exceptions, permettant d'agir sur le flot de contrôle d'un programme. Par exemple, le programme suivant définit une exception `Division_par_zero` qui est levée dans la fonction `divise` si son deuxième paramètre est égal à `0.0`. Cette exception est alors rattrapée, à l'aide de l'instruction **`try _ with`**, dans la fonction appelante `appelle_divise` qui retourne alors la valeur prédéfinie `max_float` :

```
exception Division_par_zero

let divise x y =
  if y = 0.0 then raise Division_par_zero;
  x /. y

let appelle_divise x y =
  try divide x y
  with Division_par_zero -> max_float
```

**Modules :** OCaml est un langage *modulaire*, qui permet de regrouper un ensemble de déclarations de types et de valeurs (variables, fonctions) dans des modules distincts. Par exemple, le module *Carte* suivant définit un ensemble de types et fonctions permettant de manipuler des cartes à jouer.

```
module Carte = struct

type couleur = Pique | Coeur | Carreau | Trefle
type valeur = Numero of int | Valet | Dame | Roi | As

(* Une carte = une valeur et une couleur : *)
type carte = { v : valeur ; c : couleur }

(* valeur des cartes pour la belote : *)
let points carte atout =
    match carte.v with
    | Numero 10 -> 10
    | Numero 9 -> if carte.c = atout then 14 else 0
    | Numero _ -> 0
    | Valet -> if carte.c = atout then 20 else 2
    | Dame -> 3
    | Roi -> 4
    | As -> 11
```

```
(* fonction d'affichage : *)
let affiche_carte carte =
   let string_couleur =
      match carte.c with
        | Coeur -> "Coeur"
        | Pique -> "Pique"
        | Trefle -> "Trefle"
        | Carreau -> "Carreau"
   in
   let string_valeur =
     match carte.v with
     | Numero i -> string_of_int i
     | Valet -> "Valet"
     | Dame -> "Dame"
     | Roi -> "Roi"
     | As -> "As"
   in
   print_string ((string_valeur)^"␣de␣"^(string_couleur))
end
```

Tout fichier `foo.ml` correspond implicitement à la déclaration d'un module `Foo`, dont l'interface peut être définie dans le fichier `foo.mli`. Ce modèle de programmation modulaire permet de réaliser la compilation séparée des composants distincts d'un programme.

**Objets :** Enfin, OCaml est aussi un langage de programmation orienté objet. Le système de typage des objets implanté dans le langage permet l'héritage multiple, ainsi que le polymorphisme de classes. Par exemple, le code OCaml suivant (extrait du site officiel OCaml [⚓29]) définit une classe représentant une structure de pile (*stack*) polymorphe.

```
class ['a] stack =
    object (self)
      val mutable list = ( [] : 'a list )  (* variable d'instance *)
      method push x =
        list <- x :: list
      method pop =
        let result = List.hd list in
        list <- List.tl list;
        result
      method peek =
        List.hd list
      method size =
        List.length list
    end
```

En raison de l'utilisation du polymorphisme paramétrique (dénoté par le paramètre `'a` de la classe), la façon dont est utilisée une instance de cette classe permet de restreindre le type des valeurs contenues dans la pile qu'elle représente. Par exemple, le programme suivant manipule une pile de cartes à jouer :

```
open Carte

let () =
  let s = new stack in
  s#push {v = Valet ; s = Pique}; (* s est donc une ''carte stack'' *)
  s#push {v = Dame ; s = Coeur};
  let c = s#pop in
  affiche_carte c; (* affiche ''Dame de Coeur'' *)
  print_int s#size (* affiche 1 *)
```

**Gestion automatique de la mémoire :** Comme pour le langage Java, OCaml implante un algorithme de *ramasse-miettes* (*garbage collector*). Ce mécanisme permet, à l'exécution, de libérer automatiquement la mémoire associée aux valeurs n'étant plus utilisées par le programme. Ainsi, l'utilisation du langage OCaml permet d'épargner au développeur des considérations liées à la désallocation dynamique des différentes valeurs manipulées. De surcroît, cette automatisation rend les programmes moins enclins aux bugs liés justement à des erreurs faites par le programmeur dans la manipulation de la mémoire des programmes, qui sont souvent assez difficiles à détecter.

**Constructions avancées :** Enfin, OCaml implante divers autres traits de programmation de haut niveau, comme les GADTs (*Generalized Algebraic Data Types* – « Types algébriques de données généralisés »), les *variants polymorphes* ou les *modules paramétrés* (aussi nommés *foncteurs*). Ces constructions avancées sont pleinement compatibles avec la machine virtuelle générique que nous décrivons dans ce chapitre. Nous aborderons certains exemples tirant profit de telles constructions au fil de notre discours (par exemple, nous présenterons dans la section 2.3.3 un exemple d'utilisation des GADTs destiné à augmenter la sûreté du typage des primitives réalisant la communication entre un microcontrôleur et son environnement).

## 2.2 La ZAM : Machine virtuelle OCaml de référence

OMicroB est une machine virtuelle dérivée de la machine virtuelle standard OCaml. À ce titre, elle possède de nombreux points communs avec cette dernière. Nous présentons alors, dans cette section, les principaux aspects techniques de la machine virtuelle OCaml de référence, depuis le format du *bytecode* qu'elle est capable d'interpréter, jusqu'à une courte description de sa bibliothèque d'exécution, en passant par le détail de la représentation des valeurs OCaml qu'elle manipule.

### 2.2.1 Présentation de la ZAM

Développée depuis 1996 à l'Inria dans plusieurs projets de recherche (*Cristal*, puis *Gallium*) cette machine virtuelle est une machine à pile, à noyau fonctionnel et impératif, qui peut être vue comme une version de la machine abstraite de Krivine [Kri07] qui implante un modèle d'application stricte plutôt que

l'appel par nom. Également nommée ZAM (*Zinc Abstract Machine*) en référence au projet ZINC [Ler90] dont elle tire ses origines [4], la machine virtuelle OCaml de référence constitue une des deux cibles pour la compilation du langage OCaml : en effet, un programme OCaml peut être compilé vers un fichier *bytecode* (grâce au compilateur *ocamlc*), interprétable par la ZAM (et plus précisémment par son implantation de référence, nommée *ocamlrun*), ou bien vers un exécutable natif grâce au compilateur *ocamlopt*. Dans cette thèse, nous concentrons nos travaux sur l'approche machine virtuelle de compilation d'OCaml, en raison des opportunités en matière de portabilité qu'offre cette dernière, ainsi que de la relative légèreté et simplicité de ce modèle de compilation pour le langage OCaml.

### 2.2.2 Bytecode OCaml

Dans sa version 4.06, la machine virtuelle OCaml standard contient un ensemble de 148 instructions bytecode différentes, vers lequel tout programme OCaml peut être compilé. Ces instructions bytecode correspondent à des instructions permettant d'agir sur le flot de contrôle des programmes (comme l'instruction *BRANCHIF* de branchement conditionnel), des instructions réalisant des calculs arithmétiques ou booléens (comme l'instruction *MULINT* qui réalise la multiplication de valeurs entières), des instructions qui permettent d'allouer des valeurs dynamiquement (comme l'instruction *CLOSURE* qui permet de stocker une valeur fonctionnelle sous la forme d'une fermeture dans le tas de la machine virtuelle), ou des instructions qui manipulent la pile de la machine virtuelle (comme l'instruction *PUSH* ou *POP* qui empilent et dépilent des valeurs). La majorité des instructions bytecode sont des raccourcis correspondant à des combinaisons de plusieurs instructions atomiques (par exemple, l'instruction *PUSHACC1* a le même comportement que l'instruction PUSH suivie de l'instruction *ACC1*, qui récupère le deuxième élément de la pile). Par souci de brièveté, l'ensemble des instructions bytecode OCaml et leurs descriptions ne sont pas donnés dans ce manuscrit, mais les principales instructions seront tout de même présentées [5].

La génération d'un bytecode OCaml à partir d'un fichier source OCaml est réalisée par le compilateur *ocamlc*. Le fichier généré par *ocamlc* est un programme exécutable qui contient plusieurs sections distinctes, nécessaires à l'initialisation et l'interprétation du programme OCaml par la machine virtuelle :

- Une section CODE contenant les instructions bytecode qui correspondent au code du programme compilé.
- Une section DATA contenant une représentation sérialisée des différentes variables globales du programme (constantes, exceptions, etc).
- Une section PRIM qui est une table contenant la correspondance entre les primitives externes utilisées par un programme et les entiers qui servent à y faire référence dans le bytecode.
- Une section (facultative) DLLS qui contient les noms des bibliothèques externes nécessaires à l'exécution d'un programme (par exemple, la bibliothèque `Unix` qui permet d'effectuer des appels systèmes).
- Une section (facultative) DLPT qui contient les chemins des bibliothèques utilisées par le programme.
- Une section (facultative) DBUG, utilisée pour le débogage du programme.

Par exemple, la figure 2.1 représente la définition en OCaml d'une fonction `facto`, qui calcule la factorielle d'un nombre entier, ainsi que son application à la valeur 4. Une représentation du bytecode

4. Ce projet consistait en une implantation légère du langage ML.
5. Le lecteur intéressé par la sémantique de chaque instruction bytecode peut se reporter à la documentation du projet Cadmium [⚓6].

contenu dans le fichier généré par *ocamlc* est reproduite à ses côtés. Ce bytecode n'est qu'un extrait de la section CODE de ce fichier, qui contient, en plus des instructions correspondant à la fonction facto, un grand nombre d'instructions bytecode destinées à l'initialisation du programme. De plus, le bytecode généré contient également le code des fonctions d'un module importé par défaut par tout programme OCaml, nommé `Pervasives`, qui contient la définition de fonctions usuelles (par exemple, des fonctions d'affichage) et d'opérateurs de base (comme l'addition entre deux entiers, ou le « ou » logique).

```
let rec facto x =
  match x with
  | 0 -> 1
  | _ -> x * facto (x-1)
in
facto 4
```

```
(...)
1673   BRANCH 1685

1674   ACC 0
1675   BRANCHIFNOT 1683
1676   ACC 0
1677   OFFSETINT -1
1678   PUSHOFFSETCLOSURE 0
1679   APPLY 1
1680   PUSHACC 1
1681   MULINT
1682   RETURN 1
1683   CONST 1
1684   RETURN 1

1685   CLOSUREREC 1 0 1674 []
1686   CONST 4
1687   PUSHACC 1
1688   APPLY 1
1689   POP 1
```

FIGURE 2.1 – Définition et application de la fonction factorielle, et bytecode correspondant

### 2.2.3 Structure de la machine virtuelle

La machine virtuelle OCaml manipule, lors de l'interprétation d'un programme, plusieurs registres nécessaires à son exécution. Ces registres sont les suivants :

- Un accumulateur (`acc`) permettant de stocker une valeur pour éviter une trop grande fréquence d'empilements et de dépilements dans la pile de la machine virtuelle.
- Un pointeur (`pc`) vers l'instruction bytecode suivante à interpréter.
- Un pointeur (`sp`) vers la dernière case de la pile.
- Un pointeur (`trapSp`) vers le récupérateur d'exception courant.
- Un compteur (`extra_args`) qui représente le nombre de paramètres restants à l'application d'une fonction.
- Un pointeur vers l'environnement (`env`) de la fermeture courante.
- Un pointeur vers le tableau des variables globales (`global_data`) du programme.

Le contenu de ces registres est modifié lors de l'interprétation du programme en fonction de la sémantique de chaque instruction bytecode contenue dans le fichier généré par *ocamlc*. L'interprète de bytecode OCaml de la machine virtuelle est réalisé dans le langage C.

La figure 2.2 décrit l'influence de chacune des instructions du programme bytecode présenté dans la figure 2.1 sur l'état des registres de la machine virtuelle.

| Position | Instruction | Description |
|---|---|---|
| 1673 | `BRANCH 1685` | Aller à l'instruction 1685 (`pc = 1685`) |
| 1674 | `ACC 0` | Mettre la valeur en tête de pile dans l'accumulateur (`acc = sp[0]`) |
| 1675 | `BRANCHIFNOT 1683` | Si `acc` = 0, alors aller à l'instruction 1683 (`pc = 1683`) |
| 1676 | `ACC 0` | Mettre la valeur en tête de pile dans l'accumulateur (`acc = sp[0]`) |
| 1677 | `OFFSETINT -1` | Décrémenter `acc` |
| 1678 | `PUSHOFFSETCLOSURE 0` | Empiler `acc`, et mettre dans `acc` la fermeture située dans `env` |
| 1679 | `APPLY 1` | Empiler le contexte courant (`extra_args`, `pc`, `env`), mettre `acc` dans `env`, et aller au code de la fermeture dans `acc` (i.e. `facto`) |
| 1680 | `PUSHACC 1` | Empiler `acc`, et mettre `sp[1]` dans `acc` |
| 1681 | `MULINT` | Multiplier `acc` et sp[0], mettre le résultat dans `acc`, et dépiler sp[0] |
| 1682 | `RETURN 1` | Dépiler un élément et sortir de la fonction (`pc = pop(), env = pop(), extra_args = pop()`) |
| 1683 | `CONST 1` | Mettre 1 dans `acc` |
| 1684 | `RETURN 1` | Sortir de la fonction (`pc = pop(), env = pop(), extra_args = pop()`) |
| 1685 | `CLOSUREREC 1 0 1674 []` | Créer la fermeture dont le code est à l'adresse 1674 (i.e. `facto`) dans l'accumulateur et l'ajouter au sommet de la pile |
| 1686 | `CONST 4` | Mettre la constante 4 dans `acc` |
| 1687 | `PUSHACC 1` | Empiler `acc` et mettre `sp[1]` dans `acc` |
| 1688 | `APPLY 1` | Empiler le contexte courant (`extra_args`, `pc`, `env`), mettre la valeur de `acc` dans `env`, et aller au code de la fermeture dans `acc` (i.e. `facto`) |
| 1689 | `POP 1` | Dépiler une valeur (`sp--`) |

FIGURE 2.2 – Inteprétation du bytecode du calcul du programme de la figure 2.1

### 2.2.4 Représentation des valeurs

En raison du polymorphisme paramétrique du langage OCaml, toutes les valeurs manipulées par la ZAM sont basées sur une représentation uniforme : les valeurs OCaml possèdent toutes une longueur identique, définie par le compilateur en fonction de l'architecture de l'ordinateur sur lequel le programme est exécuté (typiquement, 32 ou 64 bits). Néanmoins, le système de gestion automatique de la mémoire d'OCaml a en particulier besoin, lors de l'exécution du programme, de distinguer les valeurs immédiates, qu'il doit ignorer, des valeurs allouées dynamiquement sur le tas de la machine virtuelle, qu'il doit visiter et traiter. Cette dichotomie, dans la ZAM, est très simple : les valeurs immédiates correspondent à toutes les valeurs encodées par des entiers (comme par exemple les constructeurs de types énumérés, ou bien sûr les entiers eux-mêmes), tandis que toute autre valeur est *encapsulée* (*boxed*) sur le tas. La distinction entre ces deux catégories est faite en réservant le bit de poids faible de chaque valeur pour représenter sa nature : si le bit de poids faible d'une valeur OCaml est à 1, alors cette dernière est un entier. S'il est à 0, alors la valeur considérée correspond à celle d'un pointeur puisque toutes les adresses sont paires étant donné qu'en représentation sur 32 bits (resp. sur 64 bits) toutes les valeurs OCaml ont une taille de 4 octets (resp. de 8 octets), et donc que toutes les adresses de pointeurs sont des multiples de 4 (resp. de 8). Cette représentation permet aux éléments de la machine virtuelle de rapidement distinguer les valeurs immédiates des valeurs allouées.

La figure 2.3 illustre la représentation binaire des valeurs OCaml dans la ZAM, sur un ordinateur disposant d'un processeur avec une architecture 32 bits.

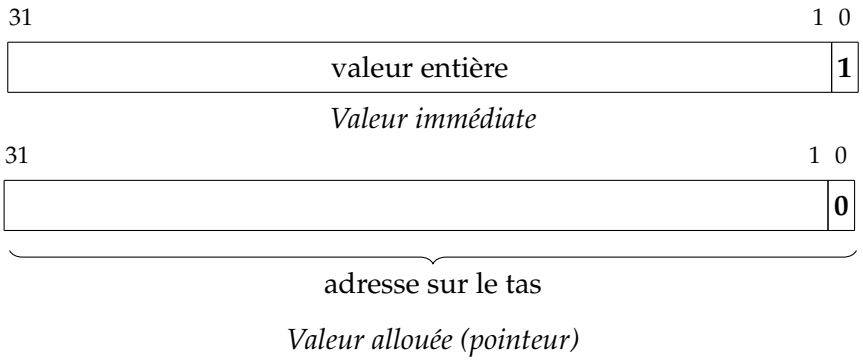



FIGURE 2.3 – Représentation dans la ZAM des valeurs OCaml sur 32 bits

Les valeurs entières peuvent alors être comprises entre $-2^{30}$ et $2^{30} - 1$, tandis que $2^{32}$ bits sont accessibles pour représenter les pointeurs vers le tas. La représentation est équivalente sur une version 64 bits de la ZAM dans laquelle les entiers sont représentés par 63 bits tandis que les pointeurs finissent aussi par zéro [6].

**En-tête des blocs alloués :** Les valeurs OCaml allouées sur le tas, telles que les fermetures, nombres à virgule flottante, n-uplets, enregistements, ou valeurs de types récursifs (listes, arbres, etc), sont encapsulées dans des blocs dont la première valeur correspond à un en-tête qui contient plusieurs informations destinées à l'interprète de la machine virtuelle ainsi qu'au garbage collector. Cet en-tête est constitué, dans une configuration 32 bits, de 22 bits représentant la taille du bloc alloué (en-tête exclu), de 2 bits de couleur nécessaires au bon fonctionnement du garbage collector, ainsi que de 8 bits de *tag*, représentant la nature de la valeur (fermeture, exception, objet, etc), et qui permet en particulier d'indiquer au garbage collector s'il doit analyser les valeurs à l'intérieur du bloc ou non :

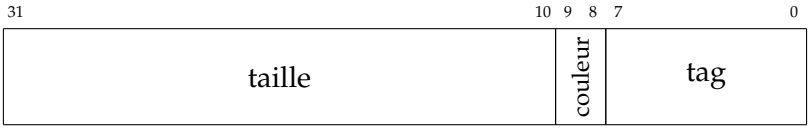



FIGURE 2.4 – En-tête d'un bloc alloué

La structure du contenu d'un bloc est dépendante de la nature de la valeur représentée : par exemple, dans le cas d'une fermeture (de tag 247), la première valeur est un pointeur vers un segment de bytecode qui correspond au code de la fermeture, et les valeurs suivantes correspondent à l'environnement de la fermeture.

6. Plus précisément, ces pointeurs finissent par `000` puisque toutes les adresses sont multiples de 8, tout comme les adresses sur 32 bits finissent par `00` car elles sont multiples de 4.

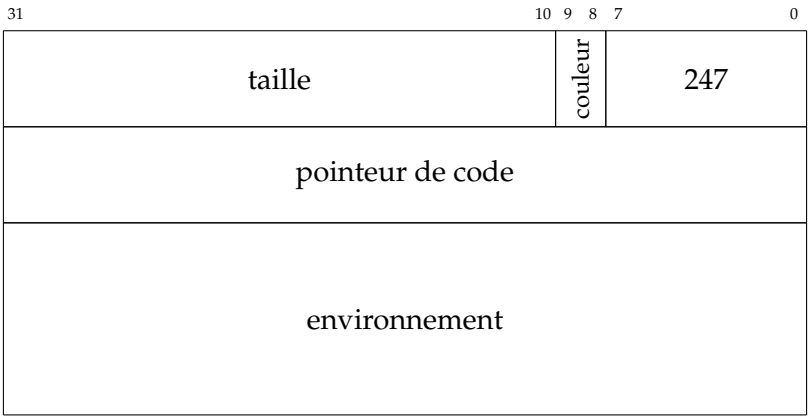
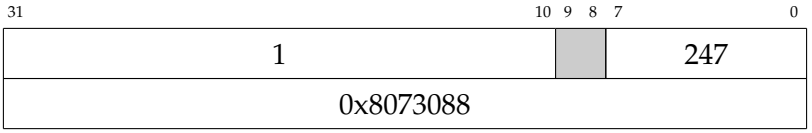
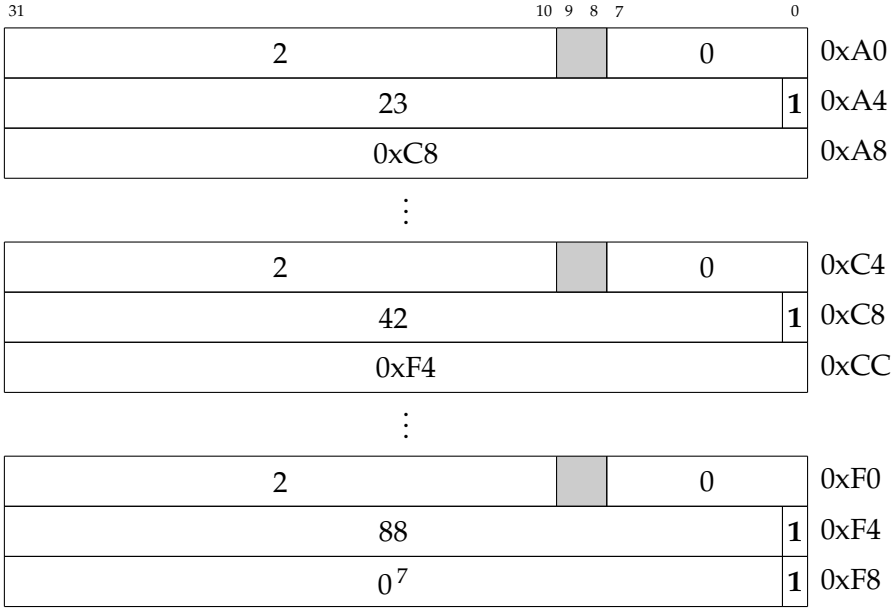



FIGURE 2.5 – Représentation d'une fermeture

Par exemple, la fermeture générée par l'instruction CLOSUREREC dans l'exemple de la figure 2.1, qui correspond à la fonction `facto`, est réduite à un bloc contenant un pointeur de code vers l'adresse 0x8073088, et un environnement vide :

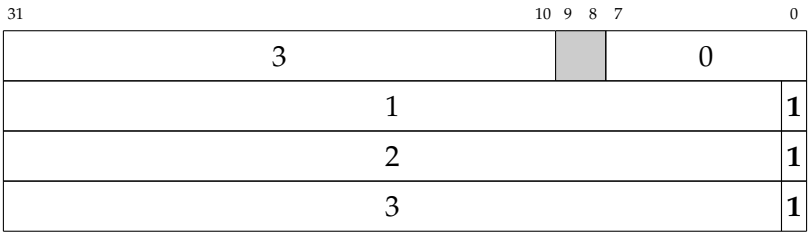



Toute structure de données est allouée dans un bloc qui contient chaque élément de cette structure. Par exemple, un élément d'une liste est représenté par la valeur de l'élément suivie d'un pointeur vers le prochain élément (non nécessairement contigu) de la liste. Il est à noter qu'un pointeur vers un bloc pointe en réalité vers la première valeur contenue dans ce bloc, et non pas vers son en-tête. La liste `[23;42;88]` est alors représentée de la sorte :




Les structures de données de taille fixe, telles que les tableaux ou bien les n-uplets, sont représentées par des blocs contenant des valeurs contiguës en mémoire. Par exemple, le triplet (1,2,3) est représenté par le bloc suivant :

7. La valeur 0 représente ici le constructeur constant `[]` de liste vide.




Cette représentation est par ailleurs identique pour le tableau [| 1 ; 2 ; 3 |] ou bien pour une référence qui contient trois champs dont les valeurs sont 1, 2, et 3. Certaines informations relatives au typage sont ainsi impossibles à retrouver après compilation du programme vers son bytecode (qui est lui-même non-typé), ainsi que pendant son exécution. Le typage statique permet toutefois d'assurer qu'un tableau n'est pas par exemple utilisé en tant que n-uplet lors de l'exécution du programme.

### 2.2.5 Bibliothèque d'exécution

La ZAM dispose d'une bibliothèque standard riche, qui permet de manipuler de nombreux types prédéfinis, tels que les listes, tableaux mutables, références, ou bien les chaînes de caractères. La plupart des modules de cette bibliothèque sont définis directement dans le langage OCaml, à l'exception de certaines fonctions ne pouvant pas être représentées par le langage, qui sont écrites en langage C, comme la fonction générique `compare`, qui permet de comparer deux éléments d'un type de donnée OCaml quelconque, que ce dernier soit un type de base (int, float, etc) ou même une structure récursive, prédéfinie ou non (liste, arbre, etc).

De surcroît, les zones mémoire des valeurs qui au cours de l'exécution du programme ne sont plus utilisées finissent par être libérées par un algorithme de ramasse-miette (*garbage collector* ou *GC*). La ZAM implante un GC hybride incrémental et générationnel qui utilise deux tas : un tas *mineur* de taille fixe dans lequel les blocs de valeurs sont d'abord allouées, et un tas *majeur* qui contient des blocs ayant « survecu » (i.e. qui sont encore utilisées par le programme) après le nettoyage du tas mineur. La phase de nettoyage du tas mineur implante un algorithme *Stop and Copy* (qui consiste à copier l'ensemble des valeurs vivantes du tas mineur dans le tas majeur), tandis que le nettoyage du tas majeur se base sur une méthode *Mark and Sweep* incrémentale qui parcourt par morceaux le tas en marquant tous les blocs vivants, pour ensuite libérer les autres. Le GC d'OCaml contient également un algorithme *Mark and Compact* lancé régulièrement, afin de compacter le tas majeur et éviter sa fragmentation.

## 2.3 Compilation et exécution d'un programme OCaml avec OMicroB

Nous présentons dans cette section le fonctionnement de la machine virtuelle OMicroB à l'aune de la description détaillée des différentes étapes permettant l'exécution d'un programme sur un microcontrôleur. Cette description permet en particulier de renseigner les différences essentielles entre la ZAM et cette machine virtuelle générique, destinée à la programmation d'appareils aux ressources limitées.

Ainsi, nous décrivons dans la suite les diverses transformations appliquées à un programme source OCaml par OMicroB, afin d'engendrer un programme exécutable pouvant être inscrit dans la mémoire d'un microcontrôleur. La figure 2.6 est une représentation schématique de cette chaîne de compilation : le fichier OCaml source est compilé vers un fichier bytecode (a), nettoyé (b), puis intégré dans un fichier C

(c) qui est alors compilé avec son interprète et sa bibliothèque d'exécution (d) vers un fichier exécutable pour le matériel considéré (e).

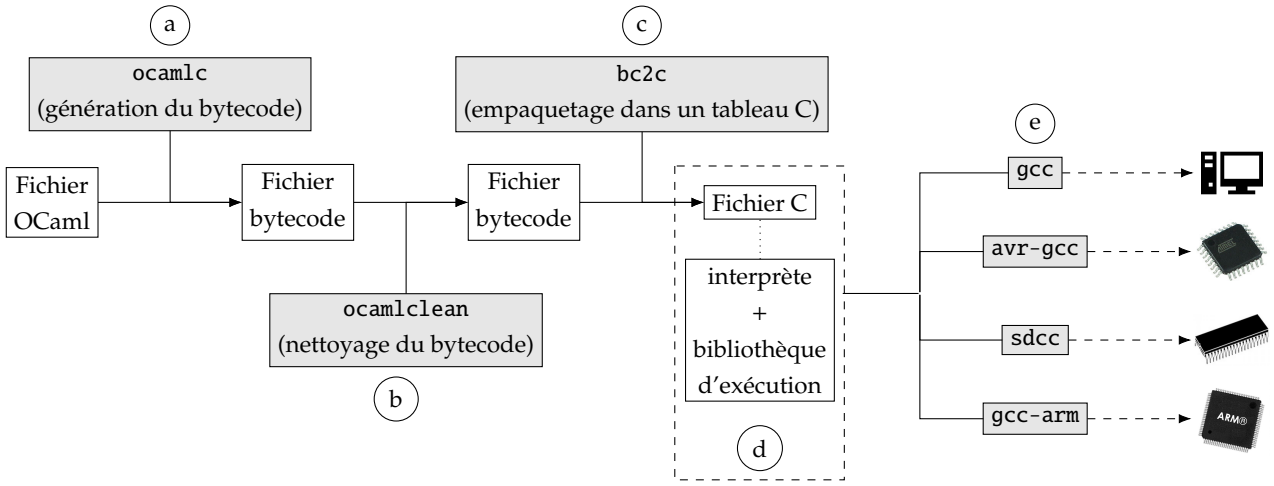



FIGURE 2.6 – Compilation d'un fichier source OCaml vers un exécutable pour microcontrôleur

### 2.3.1 Représentation d'un programme OCaml dans un fichier C

Les étapes liminaires de réalisation d'un programme exécutable par un microcontrôleur à partir d'un fichier source OCaml réalisent la compilation du programme source vers une représentation intermédiaire, le bytecode OCaml, capable d'être interprétée par la machine virtuelle. Dans OMicroB, ce bytecode est généré, nettoyé, puis intégré à un fichier C contenant une représentation des instructions, ainsi que la déclaration de tableaux C qui représentent, entre autres, la pile et le tas de la machine virtuelle.

#### Génération et nettoyage du bytecode d'un programme

Les programmes OCaml destinés à la machine virtuelle OMicroB sont écrits dans la syntaxe standard du langage, et sont ainsi totalement compatibles avec le compilateur bytecode OCaml classique, nommé `ocamlc`. Un programme OCaml destiné à être utilisé avec OMicroB est alors tout d'abord compilé vers un fichier bytecode standard à l'aide de ce dernier. L'utilisation du compilateur standard permet la réutilisation d'outils standard de l'écosystème OCaml, comme l'utilisation du logiciel *ocamldebug* afin de déboger des programmes OCaml par l'analyse de leur bytecode.

Le fichier bytecode généré par `ocamlc` contient potentiellement du code inutilisé par le programme final, en particulier le code d'initialisation de fermetures non-utilisées dans le programme, induit par l'ouverture de bibliothèques OCaml dans le code source du programme. Par exemple, un programme faisant usage du module `List` dans son code source entraînera dans le bytecode généré l'initialisation de toutes les fermetures présentes dans ce module. Par conséquent, l'empreinte mémoire d'un programme peut être sensiblement alourdie par de telles initialisations, et des programmes *a priori* légers peuvent être alors incompatibles avec les contraintes mémoires limitées incombant à l'utilisation de microcontrôleurs. Dans le but de réduire la consommation mémoire des programmes OCaml, nous faisons usage de l'outil *ocamlclean*[⚓9], issu du projet OCaPIC antérieur à nos travaux. Cet outil est capable de détecter par analyse statique du bytecode les blocs inutilisés d'un programme, et de supprimer le code d'initialisation

des fermetures inutiles qui leur sont associées [8]. Le programme résultant de cette étape de nettoyage est alors plus petit, ce qui permet son utilisation dans un cadre limité en ressources.

Le gain en mémoire apporté par l'utilisation d'*ocamlclean* peut être extrêmement variable d'un programme OCaml à l'autre, en fonction de la quantité de modules externes dont dépend un programme, et de sa structure. À titre d'exemple, considérons le programme OCaml au début de la figure 2.7. Après compilation de ce programme en un fichier bytecode avec le compilateur bytecode `ocamlc` (pour la version 4.06.0 du langage OCaml), l'utilisation de l'option `-verbose` d'ocamlclean produit l'affichage sur la sortie standard des informations positionnées en dessous du code source du programme sur la figure 2.7. Il en résulte que le nettoyage de ce programme, qui ne fait usage que des fonctions `map` et `iter` du module `List`, permet de diviser par 14.8 la taille du fichier (qui passe de 32.34 à 2.18 kilo-octets).

Il est par ailleurs à noter que du fait que le module `Pervasives` (qui définit certaines opérations et primitives de base) est implicitement chargé par tout programme, alors même un programme qui semble presque vide contient un nombre conséquent d'instructions bytecode inutilisées. L'outil *ocamlclean* apporte de ce fait une réduction quasi-systématique de la taille des programmes, y compris pour des programmes a priori très peu gourmands. Par exemple, sur un ordinateur personnel, le programme OCaml réduit à la simple valeur « `()` » (*unit*) est compilé par *ocamlc* vers un bytecode contenant 2279 instructions, et l'utilisation d'*ocamlclean* permet de réduire ce nombre à 81 instructions au total, et à diviser la taille du programme par 12 (de 18.5 ko à 1.53 ko).

```
let () =
  let l = List.map (fun x -> x + 1) [1;2;3;4;5] in
  List.iter print_int l
```

```
Statistics:
* Instruction number:     5454  ->     188   (/29.01)
* CODE segment length:   23844  ->     964   (/24.73)
* Global data number:       66  ->      25   (/2.64)
* DATA segment length:     769  ->     376   (/2.05)
* Primitive number:        352  ->       9   (/39.11)
* PRIM segment length:    7065  ->     191   (/36.99)
* File length:           32340  ->    2177   (/14.86)
```

FIGURE 2.7 – Statistiques d'élimination du code mort d'un programme avec *ocamlclean*

### Génération d'un fichier C

L'étape suivante de la chaîne de compilation d'*OMicroB* repose sur la transformation du fichier bytecode OCaml généré par *ocamlc* vers un fichier C. Ce fichier, généré par un outil nommé *bc2c*, contient une représentation du bytecode du programme OCaml, ainsi que des différentes zones mémoires nécessaires à son exécution. D'autres structures définies dans le fichier généré permettent de représenter certains ensembles de données utilisées par le programme OCaml, comme le tableau des variables globales du programme, ou une table de primitives C utilisées par le programme OCaml pour réaliser essentiellement des interactions de bas niveau avec son environnement.

8. On peut noter que *ocamlclean* « casse » le chargement dynamique des programmes étant donné que le code non-utilisé par le programme principal est supprimé. Cette limitation n'est cependant pas gênante dans notre cadre d'utilisation sur microcontrôleurs car nous ne faisons pas usage du chargement dynamique.

**Structure globale du programme C :** Le fichier C généré par *bc2c* définit plusieurs éléments spécifiques au programme considéré, qui sont manipulés par l'interprète de la machine virtuelle lors de l'exécution de ce programme. Ces éléments correspondent aux différentes sections (CODE, PRIM, DATA) du fichier bytecode originel, ainsi qu'à la mémoire utilisée par le programme OCaml, représentée par des tableaux C. La figure 2.8 est une représentation schématique du fonctionnement de *bc2c*. Les divers éléments présents dans le fichier C généré sont les suivants :

— Le bytecode du programme OCaml, sous la forme d'un tableau de constantes.
— Le tas de la machine virtuelle, qui contient les valeurs allouées dynamiquement, dans un tableau de valeurs OCaml.
— La pile de la machine virtuelle, représentée également par un tableau de valeurs.
— Le tableau des variables globales utilisées dans le programme considéré.
— L'ensemble des fonctions primitives utilisées par le programme. C'est un tableau de pointeur vers des fonctions C issues de la bibliothèque standard, ou de fonctions externes définies par l'utilisateur.
— Les tailles des tableaux générés, ainsi qu'un ensemble de valeurs constantes correspondant au numéros des instructions bytecode, sont représentées par des macros en début de fichier. En particulier, la taille du tas et celle de la pile sont choisies par le développeur au moment de la compilation du programme.

**Représentation du bytecode :** Le bytecode standard généré par le compilateur OCaml ne contient que 148 instructions différentes. Les arguments des diverses instructions bytecode sont aussi statistiquement assez petits (par exemple, l'argument $p$ de l'instruction de branchement conditionnel `BRANCHIF` $p$, qui représente une position relative dans le bytecode, correspond souvent à une position qui se trouve à proximité dans le bytecode du programme). Aligner les instructions bytecode sur des valeurs de 32 bits, à l'instar de la machine virtuelle standard OCaml, conduirait dans notre cas d'utilisation à une consommation assez excessive et plutôt inutile de ressources : la mémoire serait utilisée pour représenter des valeurs « creuses », qui contiendraient essentiellement des 0 destinés uniquement à leur alignement sur 4 octets. La lecture des instructions bytecode par l'interprète serait également lente, compte-tenu du fait que les microcontrôleurs que nous considérons ont généralement des architectures de processeurs 8 bits, et nécessitent donc plusieurs cycles pour lire et manipuler des valeurs plus grandes. L'outil *bc2c* représente alors le bytecode OCaml par un tableau constant d'octets dont chaque élément correspond soit au code d'une instruction (ou *opcode*), soit à un argument d'une instruction bytecode. Un argument qui ne tient pas sur 8 bits est représenté sur plusieurs cases consécutives du tableau généré. Cette représentation du bytecode avec des valeurs de 8 bits améliore la vitesse d'exécution du programme, et diminue surtout la taille des programmes : à partir de mesures réalisées sur un ensemble de programmes standards, nous estimons qu'un bytecode représenté par un tableau d'octets est environ 3.5 fois plus petit que le bytecode d'origine.

Afin de permettre à l'interprète de distinguer les instructions dont les arguments sont capables de tenir sur un seul octet de celles dont les arguments nécessitent l'utilisation de plusieurs octets consécutifs, il est réalisé des versions *spécialisées* des instructions bytecode concernées. Par exemple l'instruction `PUSHCONSTINT` $k$, qui empile la valeur de l'accumulateur et ajoute dans ce dernier une valeur entière constante $k$, est spécialisée dans OMicroB en trois versions capables de gérer une valeur entière pouvant

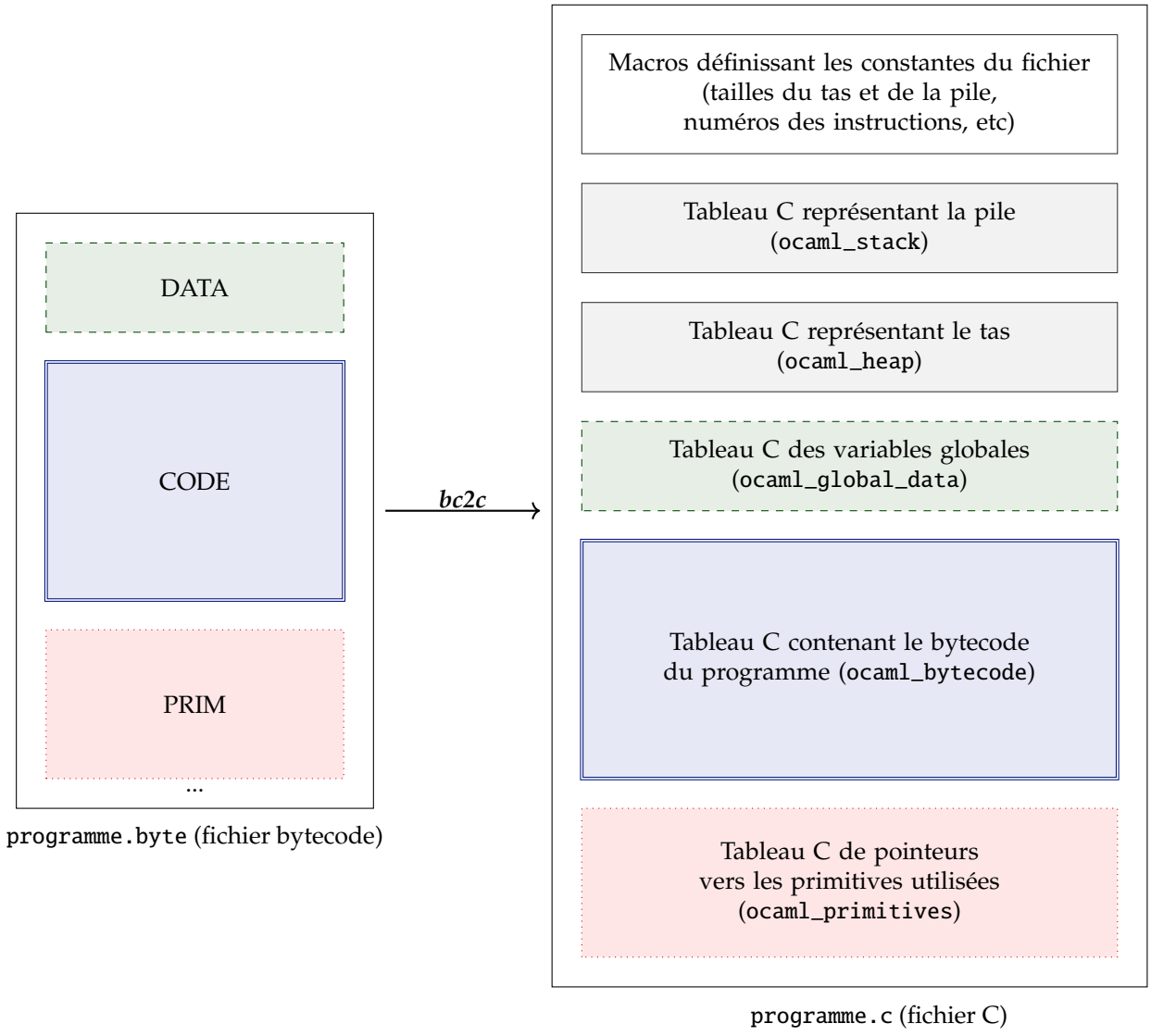



FIGURE 2.8 – bc2c : inclusion d'un bytecode OCaml dans un fichier C

être représentée sur un, deux, ou quatre octets. Cette spécialisation des instructions est symbolisée par l'adjonction des annotations `_1B`, `_2B` et `_4B` à la fin des noms classiques des instructions bytecode (par exemple : `PUSHCONSTINT_1B`, `PUSHCONSTINT_2B`, et `PUSHCONSTINT_4B`).

La figure 2.9 est une représentation du fichier généré par *bc2c* à partir du fichier issu de la compilation du programme calculant la factorielle de 4 présenté dans la figure 2.1. Afin d'économiser la RAM du microcontrôleur, l'annotation `PROGMEM` permet d'indiquer au compilateur C [9] d'allouer le tableau représentant le bytecode dans la mémoire flash, étant donné qu'il est uniquement lu pendant l'exécution du programme, et jamais modifié. Le type `opcode_t` est un alias de `int8_t`. La macro `OCAML_BYTECODE_BSIZE` est calculée par *bc2c* et représente la taille totale (en nombre d'octets) du bytecode du programme. Il est à noter que les adresses des diverses instructions bytecode sont mises à jour pour tenir compte des décalages induits par l'alignement sur 8 bits du bytecode.

```
#define OCAML_STACK_WOSIZE       42
#define OCAML_HEAP_WOSIZE        200
/* (...) */
#define OCAML_BYTECODE_BSIZE    28
#define OCAML_PRIMITIVE_NUMBER   0

value ocaml_stack[OCAML_STACK_WOSIZE];
value ocaml_heap[OCAML_HEAP_WOSIZE];

value ocaml_global_data[OCAML_RAM_GLOBDATA_NUMBER] = { /* ... */ };

PROGMEM opcode_t const ocaml_bytecode[OCAML_BYTECODE_BSIZE] = {
  /*  0 */  OCAML_BRANCH_1B, 17,
  /*  2 */  OCAML_ACC0,
  /*  3 */  OCAML_BRANCHIFNOT_1B, 11,
  /*  5 */  OCAML_ACC0,
  /*  6 */  OCAML_OFFSETINT_1B, (opcode_t) -1,
  /*  8 */  OCAML_PUSHOFFSETCLOSURE0,
  /*  9 */  OCAML_APPLY1,
  /* 10 */  OCAML_PUSHACC1,
  /* 11 */  OCAML_MULINT,
  /* 12 */  OCAML_RETURN, 1,
  /* 14 */  OCAML_CONST1,
  /* 15 */  OCAML_RETURN, 1,
  /* 17 */  OCAML_CLOSUREREC_1B, 1, 0, (opcode_t) -15,
  /* 21 */  OCAML_CONSTINT_1B, 4,
  /* 23 */  OCAML_PUSHACC1,
  /* 24 */  OCAML_APPLY1,
  /* 25 */  OCAML_POP, 1,
  /* 27 */  OCAML_STOP
};

PROGMEM void * const ocaml_primitives[OCAML_PRIMITIVE_NUMBER] = {};
```

FIGURE 2.9 – Exemple de fichier généré par bc2c

9. Cette annotation est utilisée par le compilateur *avr-gcc*.

### 2.3.2 Interprète de bytecode

À l'exécution du programme, l'interprète de la machine virtuelle OMicroB parcourt le code contenu dans le tableau représentant le bytecode du programme, et manipule les données stockées dans les tableaux qui représentent la pile et le tas. Cet interprète, dont la sémantique est identique à celle de la ZAM, a pour rôle de lire les instructions bytecode du programme et de modifier en fonction de ces dernières la mémoire du programme. L'interprète d'OMicroB contient les mêmes sept registres que la ZAM (`acc`, `pc`, `trapSp`, `extra_args`, `env`, et `globaldata`), avec la différence notable que le registre `pc` pointe vers la mémoire flash du microcontrôleur, puisque le bytecode du programme y est stocké, afin de réduire la consommation en RAM du programme. Les éléments dynamiques (pile, tas, et valeurs des registres) sont contenus dans la RAM. La figure 2.10 est une représentation graphique de l'interaction entre les divers éléments de la machine virtuelle OMicroB.

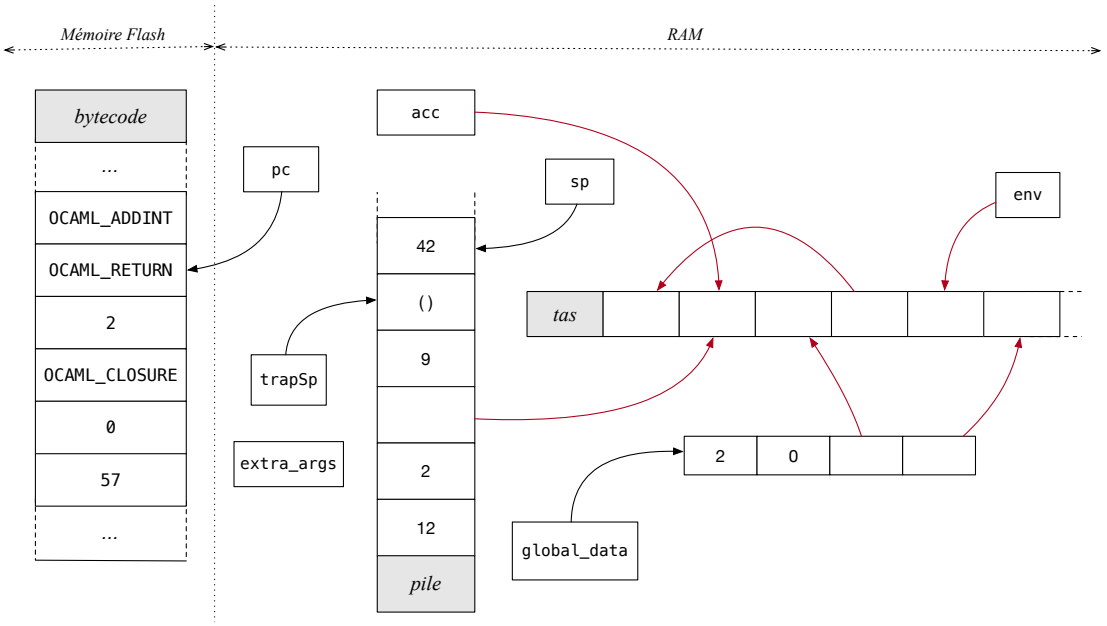



FIGURE 2.10 – La machine virtuelle OMicroB

Chacune des 148 différentes instructions (ainsi que leurs versions spécialisées) du bytecode OCaml est gérée par l'interprète d'OMicroB, écrit en C, qui manipule ces divers registres et la mémoire du programme. Comme celui de la ZAM, l'interprète exécute le programme OCaml en modifiant au fur et à mesure de son évaluation le contenu des registres de la machine virtuelle en fonction des instructions rencontrées. Nous illustrons l'interprétation du bytecode d'un programme exemple, représenté dans la figure 2.11, qui génère la valeur fonctionnelle $\lambda y.y+4$ puis l'applique à la valeur 8. La figure 2.12 représente le bytecode généré à partir du programme de la figure 2.11, associé à une description informelle de la sémantique de chacune des instructions du programme, et au détail de l'évolution des valeurs des registres au cours de l'interprétation de ce bytecode.

Dans cette figure, le flot de contrôle est représenté par des flèches annotées par des lettres, correspondant au différents états de la machine virtuelle au fil de l'exécution du programme. Pour chaque instruction bytecode, nous représentons l'état des registres `pc`, `acc`, `extra_args`, `env`, ainsi que le contenu de la pile (`sp` pointe vers la première valeur de cette dernière) à l'issu de son interprétation. Les registres `trapSp` et `global_data` ne sont pas manipulés lors de l'exécution de ce programme, et ne sont par

```
let add_x x =                (* add_x reçoit un entier x et    *)
  (fun y -> y + x)           (* crée une valeur fonctionnelle  *)
in                           (* qui reçoit y et calcule y + x  *)
  let add_4 = add_x 4 in     (* add_4 = (fun y -> y + 4)       *)
  add_4 8                    (* = 12                           *)
```

FIGURE 2.11 – Un programme OCaml qui manipule une valeur fonctionnelle

conséquent pas représentés. Les fermetures, allouées sur le tas, sont représentées entre accolades par un pointeur de code (précédé du signe @) suivi potentiellement d'une suite d'éléments qui constitue leur environnement. Par exemple, la fermeture qui correspond à `add_4`, dont le code commence à l'instruction 2 et l'environnement contient lui-même une fermeture (qui pointe vers l'instruction 3) ainsi que la valeur entière 4, est représentée de la sorte : `{@2 ; {@3} ; 4}`.

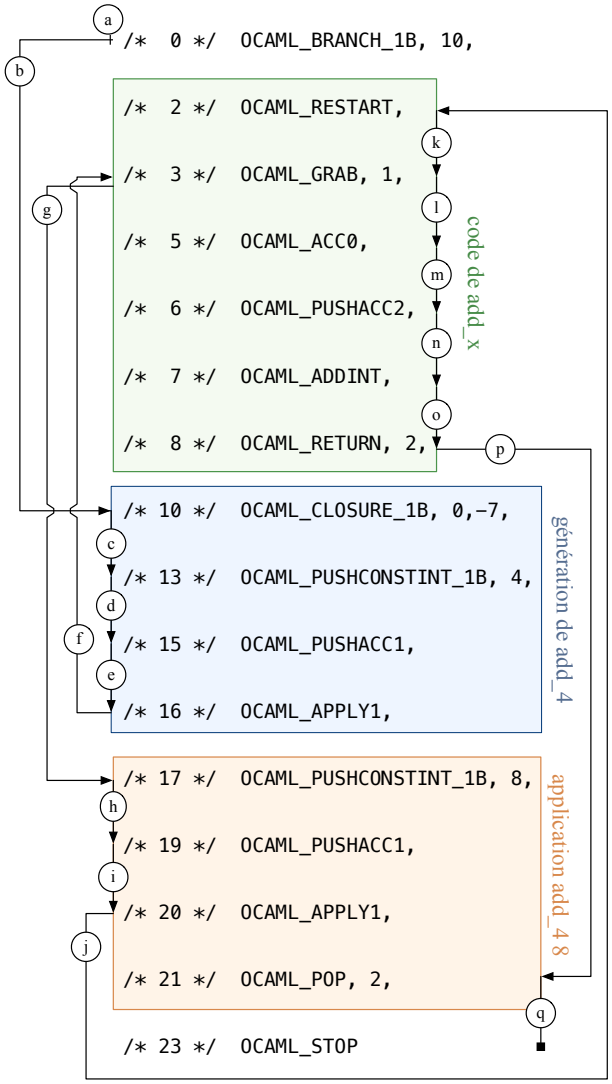
**Limitations de la version 16 bits :** Les instructions bytecode relatives au traitement des variants polymorphes et des objets manipulent les variants polymorphes et les méthodes des objets sous forme de codes de hachage (*hashs*) de leurs noms. Ces *hashs* sont représentés sur 31 bits par le compilateur standard de bytecode OCaml [10], et sont alors incompatibles avec une représentation des valeurs seulement sur 16 bits. Pour ne pas rendre l'utilisation de variants polymorphes ou d'objets impossible dans la version 16 bits d'OMicroB, la VM est fournie avec un *plug-in* pour le compilateur (sous forme d'extension de syntaxe PPX [⚓3]) qui traduit automatiquement les noms de variants polymorphes et méthodes en de nouveaux noms dont la valeur hachée tient sur 15 bits. En effet, pour les noms générés par cette extension la fonction de hachage du compilateur produit des hachés dont les 16 bits de poids fort sont tous à zéro. La génération de ces noms ne dépend que du nom d'origine, ce qui permet de conserver un mécanisme de compilation séparée des différents modules OCaml.

### Représentation des valeurs

Comme dans la machine virtuelle OCaml standard, la représentation des valeurs manipulées est uniforme. La taille de ces valeurs (16, 32, ou 64 bits) est configurable lors de la compilation, selon le choix du développeur. Nous considérons que le mécanisme natif de la ZAM consistant en l'allocation systématique de toute valeur non entière, et en particulier des valeurs flottantes, n'est pas forcément approprié dans le cadre de la programmation de microcontrôleurs. Il peut en effet être utile, afin d'éviter tout déclenchement du système de récupération de mémoire au cours de l'exécution, d'allouer au démarrage du programme l'espace mémoire nécessaire à la représentation des variables manipulées par ce dernier, et de se dispenser alors de toute allocation dynamique pendant l'exécution de celui-ci. Nous reviendrons plus précisément sur les avantages d'éviter le déclenchement du garbage collector en cours d'exécution du programme lorsque nous aborderons le calcul de WCET d'un programme au chapitre 5.

Dans OMicroB, les valeurs immédiates (non allouées) peuvent être ainsi de deux types : valeurs représentées par des entiers, ou flottants. La représentation mémoire des valeurs dans OMicroB est alors quelque peu différente de celle de la ZAM. Nous détaillons dans la suite de cette section la représentation des valeurs d'OMicroB pour une configuration 32 bits de la machine virtuelle.

10. Quelle que soit l'architecture du processeur – 32 ou 64 bits – du PC qui compile le programme.

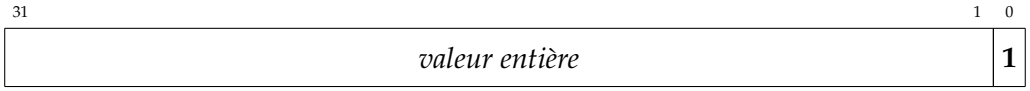



| état | description |
|---|---|
| a | début du programme |
| b | aller à l'adresse (0 + 10) |
| c | créer une fermeture (dont le code commence à l'adresse 10 − 7) dans `acc` |
| d | empiler le contenu de l'accumulateur (`acc`) et mettre la constante 4 dans `acc` |
| e | empiler le contenu de `acc` et mettre `sp[1]` dans `acc` |
| f | enregistrer le contexte courant et aller au code de la fermeture contenue dans `acc` |
| g | puisque `extra_args` < 1 alors créer une fermeture (dont le code commence à l'adresse 2) dans `acc` et retourner à l'appelant |
| h | empiler le contenu de `acc` et mettre la constante 8 dans `acc` |
| i | empiler `acc` puis mettre `sp[1]` dans `acc` |
| j | enregistrer le contexte courant et aller au code de la fermeture contenue dans `acc` |
| k | empiler les valeurs présentes dans l'environnement (`env`) et mettre la taille de `env` dans `extra_args` |
| l | puisque `extra_args=1` alors décrémenter `extra_args` |
| m | mettre la valeur de `sp[0]` dans `acc` |
| n | empiler le contenu de l'accumulateur et mettre la valeur de sp[2] dans `acc` |
| o | dépiler `sp[0]`, l'additionner avec `acc`, et mettre le résultat dans `acc` |
| p | fin de la fonction : dépiler deux élements et retourner à l'appelant |
| q | dépiler deux éléments |
| | fin du programme |

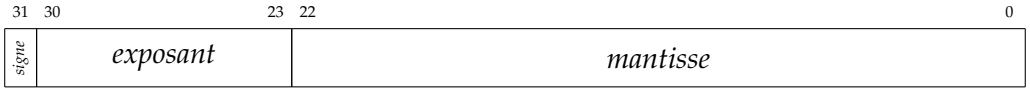
| état | pc | acc | stack | env | extra_args |
|---|---|---|---|---|---|
| a | 0 | () | [] | [] | 0 |
| b | 10 | () | [] | [] | 0 |
| c | 13 | {@3} | [] | [] | 0 |
| d | 15 | 4 | [{@3}] | [] | 0 |
| e | 16 | {@3} | [4; {@3}] | [] | 0 |
| f | 3 | {@3} | [4; @17; []; 0; {@3}] | [{@3}] | 0 |
| g | 17 | {@2; {@3}; 4} | [{@3}] | [] | 0 |
| h | 19 | 8 | [{@2; {@3}; 4}; {@2}] | [] | 0 |
| i | 20 | {@2; {@3}; 4} | [8; {@2; {@3}; 4}; {@3}] | [] | 0 |
| j | 2 | {@2; {@3}; 4} | [8; @21; []; 0; {@2; {@3}; 4}; {@3}] | [{@2; {@3}; 4}] | 0 |
| k | 3 | {@2; {@3}; 4} | [4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}] | [{@2}] | 1 |
| l | 5 | {@2; {@3}; 4} | [4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}] | [{@2}] | 0 |
| m | 6 | 4 | [4; 8; @14; []; 0; {@2; {@3}; 4}; {@3}] | [{@2}] | 0 |
| n | 7 | 8 | [4; 4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}] | [{@2}] | 0 |
| o | 8 | 12 | [4; 8; @21; []; 0; {@2; {@3}; 4}; {@3}] | [{@2}] | 0 |
| p | 21 | 12 | [{@2; {@3}; 4}; {@3}] | [] | 0 |
| q | 23 | 12 | [] | [] | 0 |

FIGURE 2.12 – Évolution des registres de la machine virtuelle pendant l'exécution du programme de la figure 2.11

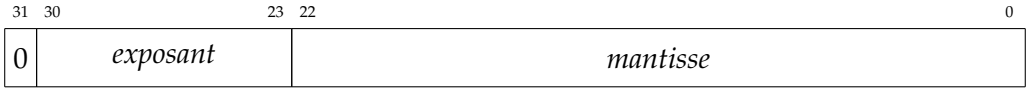
**Valeurs entières** : Les nombres entiers sont représentés, dans OMicroB, de la même façon que dans la ZAM originale : ils sont encodés sur 31 bits, et le bit de poids faible de chaque valeur OCaml correspondante est fixé à **1** :



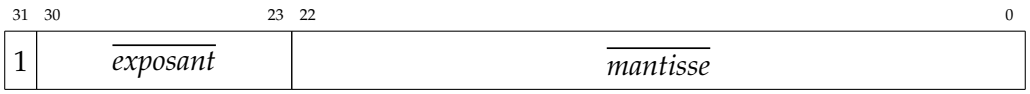

**Valeurs flottantes** : La représentation des valeurs flottantes dans OMicroB est basée sur le standard IEEE 754 [STD85]. Ce standard, sépare un nombre à virgule flottante (32 ou 64 bits) en trois éléments : son signe $s$, son exposant $e$, et sa mantisse $m$. Sur 32 bits, une valeur flottante est ainsi représentée par un bit de signe, 8 bits pour son exposant, et 23 bits pour sa mantisse :




Dans OMicroB, une nombre à virgule flottante *positif* est représenté naturellement, dans ce format, sur 32 bits :




Il est à noter que la représentation d'un entier et celle d'un flottant peuvent entrer en collision. En effet, pour peu qu'un flottant ait son bit de poids faible à 1, il n'est plus distinguable d'un entier. Ce cas de figure n'est toutefois pas gênant dans notre situation, grâce à la sûreté introduite par le typage statique strict des programmes OCaml : en aucun point du programme un flottant ne peut être confondu avec un entier, et vice-versa. Toutes les opérations du programme sont réalisées sur des valeurs de types compatibles : il n'y a par exemple aucun risque d'additionner une valeur représentant un entier et une valeur représentant un flottant. Néanmoins, la bibliothèque standard fournit une fonction de comparaison polymorphe, nommée `compare`, capable de réaliser la comparaison entre deux variables de n'importe quel type, et il est donc important, du fait qu'un flottant et un entier sont indistinguables lors de l'exécution du programme, qu'il n'existe qu'une seule manière de comparer deux valeurs immédiates (deux entiers ou deux flottants). Cette contrainte, qui permet en quelque sorte de comparer deux flottants en les voyant comme deux entiers, impose, pour que la relation d'ordre entre la représentation des entiers et celle des flottants soit identique, que la représentation des valeurs flottantes *négatives* soit modifiée. De ce fait, les flottants négatifs sont représentés, dans OMicroB, avec les bits de leur exposant et de leur mantisse inversés :

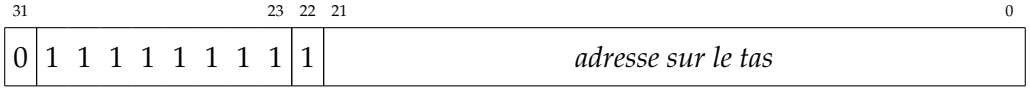



Cette modification de la représentation des flottants par rapport au standard induit un léger surcoût pour l'exécution des opérateurs arithmétiques flottants : avant l'application de ces derniers, un « xor » est appliqué sur les flottants négatifs d'OMicroB pour inverser les bits de la mantisse et de l'exposant, et les rendre alors compatibles avec les implantations standards des opérateurs flottants.

**Pointeurs sur le tas** : La distinction entre les différente catégories de valeurs OCaml n'est réellement utile que lorsqu'il est question de séparer les valeurs immédiates (dans OMicroB : les entiers et les flottants) des valeurs allouées, représentées par des adresses sur le tas. En effet, une dichotomie non-ambigüe est nécessaire au garbage collector de la machine virtuelle afin qu'il puisse distinguer un pointeur vers une valeur sur le tas (qu'il doit visiter) d'une valeur immédiate (qu'il doit ignorer). Dans OMicroB, il n'est plus possible, en raison de la représentation des flottantes comme des valeurs immédiates, d'utiliser le bit de poids faible d'une valeur pour en déduire sa nature (un flottant peut en effet terminer par 1 ou 0). Pour représenter des pointeurs sur le tas, nous utilisons alors, dans OMicroB, une astuce induite par une particularité du standard IEEE 754. Dans ce standard, les valeurs pour lesquelles les bits de l'exposant sont tous égaux à 1 (i.e. pour lesquelles l'exposant est égal à 128 sur une représentation 32 bits) correspondent à des valeurs *spéciales*, à distinguer en deux catégories :

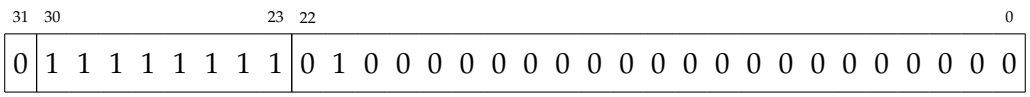
1. Si la mantisse est égale à 0, alors cette représentation correspond, en fonction du bit de signe, à la valeur $\pm\infty$.
2. Si la mantisse est différente de 0, ces valeurs correspondent à des *NaN* (*Not a Number*) : des valeurs permettant de représenter le résultat de certaines opérations invalides (comme par exemple la division 0/0 ou le calcul de la racine carrée d'un nombre négatif). Ces valeurs sont très nombreuses, puisque tout nombre flottant avec une mantisse non nulle et dont l'exposant vaut 128 est un *NaN* : il existe alors, dans une représentation 32 bits, $2^{23} - 1$ valeurs *NaN* différentes. Nous tirons profit, dans OMicroB, de cet espace conséquent de valeurs flottantes « inutilisées » afin d'y représenter les pointeurs vers des valeurs OCaml allouées sur le tas.

Ainsi, nous représentons les pointeurs vers le tas par une valeur *NaN* qui commence par la suite de bits **0111 1111 11** :



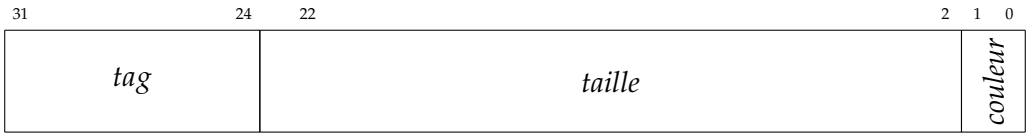

Sur une représentation 32 bits de la machine virtuelle, ce système de *NaN-boxing* [Gud93] avec marquage des bits de poids faibles nous offre ainsi un espace de $2^{21}$ bits pour représenter des adresses, et permet ainsi d'allouer $2^{19}$ valeurs OCaml (car toutes les adresses sont des multiples de 4 étant donné qu'une valeur OCaml occupe 4 octets). Cet espace d'adressage théorique de 2 mégaoctets est alors largement suffisant pour le matériel que nous visons, limité au maximum à quelques dizaines de kilo-octets de mémoire vive, et pour lequel les adresses vers la RAM sont souvent physiquement limitées à 16 bits. Pour du matériel plus riche en ressources, l'utilisation d'une configuration 64 bits de la machine virtuelle permet de s'offrir un espace de $2^{50}$ bits pour représenter une adresse, soient plus de $2^{47}$ valeurs OCaml distinctes, puisque chacune d'elle occupe alors 8 octets.

Toute valeur *NaN* réellement calculée par le programme est représentée dans OMicroB par l'unique valeur suivante :




La bibliothèque standard de OMicroB est adaptée pour que les opérations sur les *NaN* soient cohérentes avec le standard IEEE 754 (qui stipule par exemple qu'un test d'égalité entre de deux *NaN* est toujours faux).

**En-têtes de blocs** : Enfin, les en-têtes des blocs OCaml alloués sur le tas sont représentés sur 32 bits, avec 8 bits de *tag*, 22 bits de taille, et 2 bits de couleur :




**Représentation 16 et 64 bits** : Le détail de la représentation des valeurs dans OMicroB en 16, 32 et 64 bits est donné en annexe A. La représentation 64 bits est semblable à celle sur 32 bits, en étendant les valeurs entières à 63 bits et en représentant les flottants sous le standard IEEE 754 au format double précision. La représentation 16 bits est proche de la représentation *binary16* du standard IEEE 754, mais ne permet (pour distinguer un flottant d'un pointeur) que l'utilisation de 15 bits effectifs (nous réduisons donc la mantisse d'un flottant de 10 bits à 9 bits).

### 2.3.3 Bibliothèque d'exécution

La bibliothèque standard disponible avec OMicroB reprend de nombreux modules usuels de la machine virtuelle standard, tel que le module `List` qui définit de nombreuses fonctions manipulant des listes (comme la fonction `map` qui permet d'appliquer une fonction à tous les éléments d'une liste), le module `Queue` qui permet de représenter des files d'attentes (*FIFO*) mutables, ou le module `Hashtbl` qui permet de représenter des tables de hachage. À ces modules communs aux deux implantations, s'ajoutent des modules particuliers à la programmation de microcontrôleurs, comme un module `Avr` qui définit des primitives d'interaction de bas niveau avec un microcontrôleur *ATmega*, ou bien un module `LiquidCrystal` [11], permettant de communiquer avec un écran à cristaux liquides. À l'inverse, certains modules n'ayant pas ou peu de sens pour la programmation de microcontrôleurs, comme le module `Unix` permettant d'effectuer des appels système, ne sont pas disponibles.

L'utilisation d'OCaml pour définir les primitives de base de la configuration du microcontrôleur et de ses interactions avec son environnement apporte une augmentation de la sûreté des programmes réalisés. En effet, le typage strict d'OCaml, associé à son support de types de données algébriques généralisés (*Generalized Algebraic Data Types* ou *GADT*), permet de définir des primitives bas niveau qui vérifient certains critères de typage.

Par exemple, les bits du registre `SPCR` (*SPI Control Register*) permettant la configuration du port série d'un microcontrôleur AVR sont représentés par le type `spcr_bit` :

```
type spcr_bit = SPR0 | SPR1 | CPHA | CPOL | MSTR | DORD | SPE | SPIE
```

Tandis que ceux du registre `SPSR` (*SPI Status Resister*), qui permet de scruter l'état de la communication série (par exemple pour vérifier si une transmission est terminée) sont représentés par le type `spsr_bit` :

```
type spsr_bit = SPI2x | SPSR1 | SPSR2 | SPSR3 | SPSR4 | SPSR5 | SPSR6 | SPIF
```

11. Réalisé par un étudiant en master lors d'un stage destiné à la programmation par contraintes avec OMicroB [Pes18].

Les registres du microcontrôleur sont alors représentés par un type `'a register` paramétré par le type des bits qu'il contient :

```
type 'a register =
  | (* ... *)
  | SPCR : spcr_bit register
  | SPSR : spsr_bit register
  | (* ... *)
```

La primitive `set_bit`, de type `'a register -> 'a -> unit`, permet de donner la valeur 1 à un des bits d'un registre du microcontrôleur. À la compilation, toute utilisation de cette fonction dans un programme entraîne alors, par typage, la vérification que le bit passé en deuxième paramètre à la fonction est bien un bit du registre passé en premier paramètre. Par exemple, l'appel incorrect à `set_bit SPCR SPIF` entraîne une erreur explicite lors de la compilation du programme :

```
Error: This variant expression is expected to have type Avr.spcr_bit
       The constructor SPIF does not belong to type Avr.spcr_bit
Hint: Did you mean SPIE?
```

Cette incohérence entre le nom du bit et le nom du registre n'aurait pu être détectée par la simple utilisation de macros pour représenter le nom des bits dans un langage de bas niveau, comme C. La vérification de la correction du typage d'un programme illustre bien l'un des avantages de l'utilisation d'un langage de programmation de haut niveau, et ce même en ce qui concerne de telles interactions de bas niveau.

Dans OMicroB, la mémoire du programme est automatiquement gérée par un *garbage collector* qui implante un algorithme *stop-and-copy* dont le fonctionnement est standard. Il manipule en effet deux bassins pour les valeurs allouées sur le tas : un bassin *vivant* et un bassin *de copie*. À chaque exécution du GC, les valeurs toujours utilisées par le programme sont copiées du bassin vivant vers le bassin de copie, puis les rôles de chaque bassin sont inversés (le bassin vivant devient le bassin de copie, et vice-versa). Pour ce faire, l'algorithme de gestion mémoire parcourt un ensemble de *racines* qui correspondent aux registres et autres blocs mémoire qui peuvent contenir des pointeurs vers des valeurs allouées sur le tas (les flèches qui pointent sur le tas dans la figure 2.10 représentent de tels pointeurs), copie les valeurs vers lesquelles elles pointent dans le nouveau bassin et met à jour chaque pointeur avec le nouvel emplacement des valeurs copiées. Le GC fait usage de la distinction introduite par la représentation des valeurs dans OMicroB : il visite toutes les valeurs de pointeurs encodées par NaN-Boxing. Un tel algorithme a l'avantage d'être rapide, mais possède l'inconvénient notable de réserver la moitié de la mémoire vive disponible (le bassin de copie) pour son fonctionnement. Un *garbage collector* de type *Mark and Compact*, qui évite de « gâcher » la moitié du tas (au prix tout de même d'une vitesse d'exécution moindre) est également disponible.

### 2.3.4 Réalisation d'un exécutable

#### Compilation pour l'architecture cible

La compilation du programme exécutable consiste en la dernière étape avant le transfert du programme sur le matériel visé. Elle fait usage de compilateurs C pré-existants, comme *avr-gcc* ou *sdcc*. Le compilateur C réalise alors l'édition de liens entre le fichier C réalisé par *bc2c*, l'interprète de la machine virtuelle, et sa bibliothèque d'exécution (bibliothèque standard et garbage collector). Le fichier généré est alors transferé sur microcontrôleur (avec un outil adapté comme *avrdude*) et exécuté sur ce dernier.

L'utilisation du langage C comme une sorte d'*assembleur portable* permet d'utiliser OMicroB à moindre peine sur de nombreuses architectures différentes. Pour supporter une nouvelle architecture de microcontrôleurs, la machine virtuelle ne nécessite que la réécriture des primitives C de bas niveau, essentielles à l'interaction avec le matériel (comme les fonctions permettant de lire la mémoire flash du microcontrôleur), et n'impose aucune modification profonde du code de l'interprète ou des autres composants d'OMicroB.

#### Débogage et simulation des programmes pour microcontrôleurs

La génération d'un code C générique permet également d'exécuter la machine virtuelle sur du matériel plus conventionnel. En effet, le compilateur *gcc* peut être utilisé sur l'ordinateur sur lequel a été écrit le programme afin de le compiler vers un exécutable compatible avec l'architecture de ce dernier (un exécutable x86 ou x86-64). Ce mode de compilation permet de simuler sur un ordinateur l'exécution de programmes avant même leur transfert sur le microcontrôleur cible, dans l'objectif d'en simplifier le processus de débogage.

À ce titre, OMicroB contient des outils permettant de simuler les effets d'un programme avec une représentation graphique de l'état des broches d'entrée/sortie d'un microcontrôleur (figure 2.13). Cette simulation permet de vérifier la cohérence de l'exécution du programme avec le comportement attendu sans contraindre le développeur à transférer systématiquement le programme sur un microcontrôleur réel.

Le simulateur d'OMicroB, à l'image de celui d'OCaPIC, permet également de représenter les interactions entre le microcontrôleur et le montage qui lui est associé : il offre en effet la possibilité au programmeur de décrire les composants branchés au microcontrôleur (boutons, écrans, capteurs, ...) pour simuler également l'effet du programme sur ces derniers. Par exemple, la figure 2.14 représente la simulation du montage d'un petit programme qui affiche un bonhomme souriant sur un écran OLED (*Organic Light-Emitting Diode* – diode électroluminescente organique) si l'utilisateur appuie sur le bouton « SMILE », et un bonhomme triste si l'utilisateur appuie sur le bouton « FROWN ». Les différents composants du montage sont décrits dans un fichier `circuit.txt` utilisé par le simulateur pour réaliser les affichages adaptés.

## 2.4 Optimisations d'OMicroB

Considérant les limitations importantes du matériel auquel est destinée la machine virtuelle OMicroB, nous avons concentré nos travaux sur la réduction de l'empreinte mémoire des programmes exécutés sur microcontrôleurs. Pour ce faire, OMicroB implante plusieurs optimisations destinées à limiter la consommation en ressources de la machine virtuelle et des programmes OCaml réalisés. Par exemple,

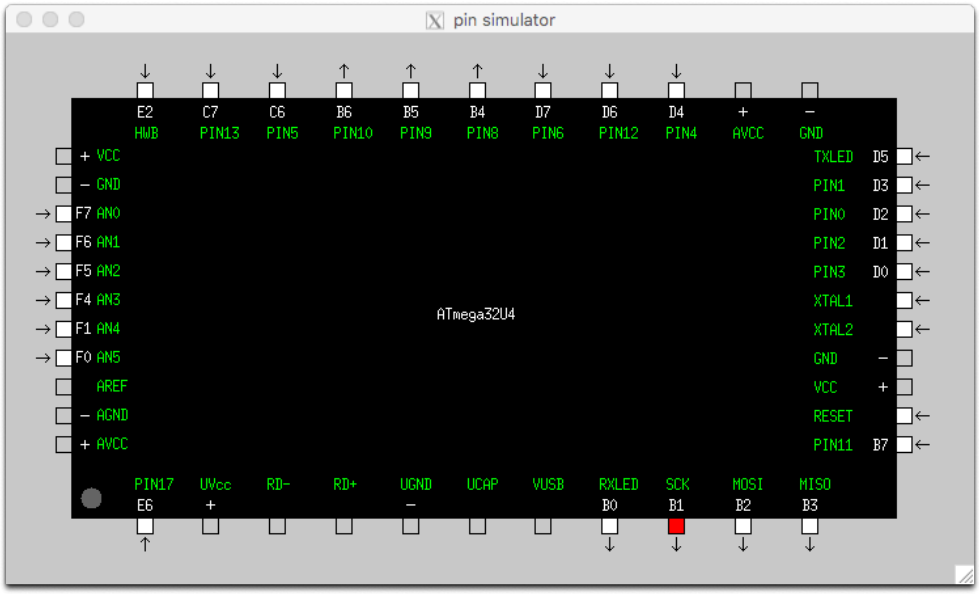



FIGURE 2.13 – Simulation de l'état des broches d'un microcontrôleur ATmega32u4


circuit.txt

```
window width=350 height=150 bgcolor=lightgray title="Simulator"
oled x=30 y=10 column_nb=128 line_nb=8 cs=PIN12 dc=PIN4 rst=PIN6
button x=310 y=100 width=40 height=40 label="SMILE" pin=PIN7 color=green
button x=310 y=50 width=40 height=40 label="FROWN" pin=PIN8 color=red
```

FIGURE 2.14 – Simulation d'un petit montage

la réduction de la taille des *opcodes* des instructions bytecode de 32 bits vers 8 bits constitue une telle optimisations. Nous présentons dans cette section deux autres optimisations appliquées à la machine virtuelle et aux programmes OCaml.

### 2.4.1 Évaluation anticipée des programmes

Le bytecode interprété au démarrage d'un programme OCaml correspond à une séquence d'initialisations de valeurs utiles à son exécution. Tout d'abord, le programme se charge de déserialiser certaines constantes (listes, chaînes de caractères, . . . ) et de les allouer sur le tas. L'initialisation du programme se poursuit ensuite par l'allocation des fermetures utilisées dans le programme, par la création de modules, et le calcul de variables globales. Cette phase d'initialisation peut être lente et consommer une quantité non négligeable de mémoire. En particulier, la profondeur de pile nécessaire au chargement des modules utilisés par le programme peut être conséquente. Par exemple, le programme de la figure 2.15 qui définit une classe `pt` nécessite au moins 106 niveaux de pile pour pouvoir charger les modules responsables de l'utilisation des objets en OCaml.

```
class pt (x:int) (y:int) =
 object
   method get_x = x
   method get_y = y
 end

let () =
  Avr.delay 1000;
  let p = new pt 1 2 in
  p#get_x
```

FIGURE 2.15 – Exemple de programme OCaml manipulant des objets

Toutefois, l'exécution d'un programme OCaml est totalement déterministe pendant ces phases de précalculs, quel que soit le matériel sur lequel il est exécuté. Ce déterminisme perdure jusqu'à la première entrée/sortie du programme, qui amorce la communication entre le programme et son environnement. Il est alors possible, dans le but d'accélérer la vitesse de démarrage des programmes, et surtout d'en diminuer la consommation mémoire, de réaliser les différentes étapes d'initialisation du programme en amont de son exécution sur microcontrôleur. L'interprétation du bytecode est ainsi simulée par *bc2c* sur l'ordinateur sur lequel est compilé le programme jusqu'à la première entrée/sortie, et l'état mémoire résultant (i.e. les valeurs contenues dans le tas, la pile, et le tableau des variables globales du programme) est alors directement inscrit dans le code C généré par *bc2c*. Ainsi, la pile et le tas de la machine virtuelle, lors de la production du code C qui embarque le bytecode OCaml, sont préalablement peuplées de valeurs correspondant aux fermetures et valeurs calculées avant la première instruction qui débute l'interaction entre le programme et son environnement. Ce processus d'évaluation anticipée peut être vu comme une forme d'évaluation partielle [JGS93] réalisée sur le bytecode du programme OCaml.

Dans l'exemple de la figure 2.15, l'activation de l'évaluation anticipée (qui a pour conséquence de pré-exécuter le programme jusqu'à l'appel à la primitive `Avr.delay` [12]) permet alors de réduire à seulement 16 niveaux la taille de la pile nécessaire pour exécuter le programme.

12. Cette primitive permet de mettre en pause le programme pendant la durée indiquée par son paramètre (en millisecondes).

### 2.4.2 Machine virtuelle sur mesure

La plupart des programmes compilés ne font pas usage de l'intégralité du jeu d'instructions bytecode associé au langage OCaml, mais seulement d'une sous-partie de ce dernier, qui correspond aux traits du langage réellement utilisés dans le programme. Par exemple, un programme n'utilisant pas la couche objet du langage OCaml ne contiendra pas, après compilation vers le bytecode, l'instruction `GETPUBMET` permettant l'appel à une méthode d'un objet. De même, puisqu'un grand nombre d'arguments est capable de tenir sur uniquement un octet, il est rare qu'il soit réellement fait usage d'instructions spécialisées pour les arguments sur quatre octets, comme l'instruction `CLOSURE_4B`. De ce fait, embarquer dans l'exécutable final un interprète capable de gérer de telles instructions absentes du bytecode du programme représente une perte d'espace, alors même que la mémoire d'un microcontrôleur est une ressource peu abondante.

OMicroB, permet d'éviter ce gaspillage inutile de mémoire, en incluant lors de la compilation du programme un interprète de bytecode qui ne peut traîter que les instructions réellement présentes dans le bytecode du programme. Pour ce faire, des directives destinées au préprocesseur du compilateur C sont ajoutées au code de l'interprète afin d'empêcher la compilation du code lié à l'interprétation des instructions non utilisées dans le programme. Chaque opcode présent dans le programme correspond alors à une *macro*, définie dans le programme C généré par *bc2c*, qui représente une valeur constante (choisie de façon à ce que toutes les valeurs soient contiguës). La taille totale de l'exécutable transféré sur le microcontrôleur est ainsi réduite.

La figure 2.16 illustre la structure de l'interprète sur mesure sur un extrait de ce dernier : le code responsable du traitement de l'instruction `BRANCHIF_4B` n'est compilé qu'à condition que la macro correspondant à cette instruction soit définie dans le code C généré par *bc2c*.

```
#ifdef BRANCHIF_4B
    case BRANCHIF_4B :
       /* code d'interpretation de l'instruction */
    break;
#endif
```

FIGURE 2.16 – Interprète sur mesure

D'autres optimisations qui reposeraient sur la même idée de généricité d'un programme C pourraient être envisageables. Nous pourrions en effet aller encore plus loin dans notre approche consistant à réaliser une machine virtuelle spécifique à chaque programme, par exemple en remplaçant statiquement chaque instruction bytecode d'un programme par l'ensemble des instructions C qui lui correspondent, et ainsi se passer de l'utilisation d'un interprète de bytecode. Cette approche a été suivie par l'outil *OCamlCC* [MV13], dont la première implantation transformait un bytecode OCaml en un programme C en remplaçant chaque instruction du bytecode par le code de bas niveau associé par un mécanisme de macro-expansion. Une telle approche donne de bonnes performances de vitesse, mais la taille des programmes croît rapidement compte tenu du fait que plusieurs références à une instruction bytecode particulière entraînent à chaque fois la duplication du code de bas niveau nécessaire pour l'exécuter. D'autres approches, comme CeML [Cha92] ou Camlot [Cri92] consistent à transformer du code source d'un programme écrit dans un langage ML en un code C, qui peut alors profiter des optimisations de compilateurs C. Le code généré par ces solutions est tout de même assez volumineux, et leurs bibliothèques d'exécution ont généralement aussi une taille conséquente, car elles doivent par exemple intégrer

un mécanisme d'application générique pour construire des fermetures en cas d'application partielle de fonctions. Ces approches sont donc plutôt réservées à du matériel pour lequel les considérations liées à la consommation mémoires sont moins importantes que la vitesse d'exécution d'un programme. OMicroB nous semble être alors un bon compromis entre la portabilité apportée par l'implantation de la VM en C, et la réduction de taille des programmes issue de la représentation des programmes sous forme d'un bytecode qui factorise des séquences d'instructions de plus bas niveau. Nous suivons donc cette approche pragmatique, pour laquelle les vitesses d'exécution peuvent être un peu plus faibles, sans être particulièrement pénalisant pour les applications que nous visons.

### Conclusion du chapitre

L'utilisation de la machine virtuelle OMicroB constitue une approche générique et configurable pour permettre l'exécution de programmes OCaml sur des appareils très variés. En particulier, les optimisations réalisées dans OMicroB permettent l'interprétation de bytecode OCaml sur des microcontrôleurs issus de gammes bénéficiant de peu de ressources matérielles. Cette machine virtuelle est capable d'exécuter des programmes OCaml sur des microcontrôleurs AVR dont les ressources en mémoire RAM sont fortement restreintes, comme le ATMega325p (2 ko de RAM), ATmega32u4 (2.5 ko de RAM), ou le ATmega2560 (8ko de RAM). Plusieurs projets académiques visant à porter OMicroB sur d'autres architectures, comme les PIC32 ou les ARM Cortex-M0 (utilisés par les cartes de développement *micro :bit*, conçues par la BBC dans le but d'accompagner l'enseignement de la programmation à de jeunes enfants [⚓20]) sont en cours de réalisation, et les premiers résultats sont fort prometteurs : les premiers portages d'OMicroB ont en effet été réalisés sans encombre [PB19].

Le modèle de compilation des programmes OCaml considéré, qui consiste en l'utilisation du bytecode d'un langage de haut niveau, associé à un interprète générique, augmente fortement le portabilité des programmes : un même programme OCaml peut ainsi être aisément porté d'un appareil à un autre. De surcroît, cette portabilité permet de simuler facilement les programmes réalisés, en tenant compte des contraintes mémoires issues de la configuration de la machine virtuelle (comme la taille de la pile, ou celle du tas), et offre ainsi un procédé de débogage accéléré et simplifié.

Dans le but de poursuivre les efforts ayant pour cible de réduire l'empreinte des programmes OCaml sur la mémoire RAM, plusieurs techniques sont actuellement en cours d'étude. Notamment, une analyse fine permettant de détecter les valeurs constantes immutables dans un programme OCaml (comme par exemple les chaînes de caractères, ou les *sprites* d'un programme de jeu vidéo) permettrait de déplacer ces dernières dans la mémoire flash du programme lors de sa compilation, et ainsi de libérer la mémoire vive, plus étroite.

La table 2.1 recense les principales différences entre OMicroB et la machine virtuelle OCaml standard. Dans le chapitre 7, nous traiterons plus en détail des performances d'OMicroB, à la fois sur le plan de la vitesse d'exécution des programmes OCaml, mais également de leur empreinte mémoire.

En raison de sa richesse et son haut niveau d'expressivité, OCaml est un langage puissant pour décrire les comportements algorithmiques des programmes, et la sûreté de son typage apporte des garanties importantes pour la réalisation de programmes embarqués. Toutefois, OCaml n'est pas vraiment adapté à l'heure actuelle à la description des aspects concurrents d'un programme, ni au développement de systèmes temps réel. Pourtant, les programmes embarqués que nous considérons possèdent de nombreux traits concurrents, et les systèmes *critiques* qu'ils contrôlent correspondent très souvent à des systèmes temps réel, pour lesquels les temps d'exécution doivent être finement contrôlés. De ce fait, nous présentons dans le chapitre suivant le langage *OCaLustre*, une extension du langage OCaml à la programmation synchrone, qui offre un modèle de concurrence léger et adapté à la nature des applications pour microcontrôleurs. Cette extension est pleinement compatible avec OMicroB, et bénéficie ainsi des avantages d'OCaml et des optimisations de la machine virtuelle décrites dans cette section.

| | ZAM (ocamlrun) | OMicroB |
|---|---|---|
| Taille des opcodes | 32 bits | 8 bits |
| Nombre d'instructions | 148 | 148 + 46 inst. spécialisées = 194 |
| Taille des valeurs OCaml | non configurable (dépendante de l'architecture) | configurable (16, 32, 64 bits) |
| Taille des adresses (espace d'adressage) dans une configuration 32 bits | 32 bits (4 Go) | 21 bits (2 Mo) |
| Représentation des flottants | Avec encapsulation (allocation sur le tas) | Immédiate |
| Algorithme de GC | Hybride (générationnel / incrémental) | Stop and Copy ou Mark and Compact |

TABLE 2.1 – Principales différences entre la machine virtuelle OCaml standard et OMicroB

# 3 OCaLustre : Programmation synchrone en OCaml

OCaLustre est une extension synchrone à flot de données du langage OCaml, inspirée du langage Lustre [CPHP87, Ber86, HCRP91], permettant d'utiliser la couche d'abstraction synchrone pour programmer la concurrence des programmes, et d'utiliser les traits de haut niveau du langage support, OCaml, pour développer les aspects logiques des applications. Cette extension, à l'empreinte mémoire très légère, est destinée à être exécutée en association avec la machine virtuelle OMicroB afin d'exécuter des programmes concurrents légers et sûrs sur des microcontrôleurs à faibles ressources.

Dans ce chapitre, nous décrivons en détail cette extension de langage. Nous présentons dans une première section les principaux traits et principes du langage permettant de réaliser des programmes synchrones en OCaml. Nous abordons en particulier dans cette section le *système d'horloges synchrones* implanté dans OCaLustre, qui permet de conditionner la présence de certaines valeurs lors de l'exécution d'un programme. Nous présentons ensuite la spécification des systèmes de types des programmes OCaLustre, qu'ils concernent à la fois les types « standards » représentant les valeurs portées par les éléments d'un programme, ainsi que les types d'horloges liés à la notion de *cadencement* d'un programme synchrone. Enfin, nous décrivons formellement la sémantique opérationnelle du langage, tirée de celle du langage Lustre.

## 3.1 Programmer en OCaLustre

Dans cette section, nous présentons les principaux concepts et les fonctionnalités générales du langage OCaLustre. Cette présentation a pour objectif d'enseigner au potentiel développeur OCaLustre les différents aspects du langage permettant de réaliser un programme correct. En dehors de certaines variations syntaxiques particulières, ces aspects ne devraient pas surprendre le lecteur familier du langage Lustre (ou de ses dérivés), tant le langage OCaLustre suit les mêmes fondements sémantiques que ce dernier.

### 3.1.1 Syntaxe du langage

À la manière d'un programme Lustre, un programme OCaLustre est composé de nœuds, qui calculent des flots de valeurs, mais également de fonctions OCaml standards, qui permettent de gérer les aspects algorithmiques des programmes en faisant appel à la puissance expressive du langage OCaml. Un nœud OCaLustre est défini à l'aide du mot-clé **`let%node`**, suivi de son nom, du nom de ses paramètres en entrée, et d'un n-uplet représentant ses différents flots de sortie (labellisé par le mot-clé **`return`**). Le corps d'un programme est un système d'équations résolu à chaque instant synchrone d'exécution du programme, attribuant ainsi des valeurs aux flots de sortie de chaque nœud du programme. Les équations présentes

dans le corps d'un nœud sont des définitions de flots de la forme $y = e$, avec $y$ le nom (ou un n-uplet de noms) du flot défini, et $e$ une expression qui permet d'évaluer la valeur du, ou des flots concernés.

Nous présentons en premier lieu une version partielle du langage, que nous enrichirons au fil du discours. Dans cette version, une expression peut correspondre à une constante (y compris la valeur *unit*), à une variable, à un constructeur d'un type énuméré défini en OCaml, à l'application de l'opérateur conditionnel (*if-then-else*), à l'application d'un opérateur arithmétique ou logique, ou à un n-uplet. La figure 3.1 décrit la syntaxe de cette version partielle du langage, en forme de Backus-Naur (*Backus-Naur Form – BNF*) [1].

| | | |
|---|---|---|
| ⟨int⟩ | ::= | $[0-9]+$ |
| ⟨float⟩ | ::= | $[0-9]+$ `.` $[0-9]*$ |
| ⟨bool⟩ | ::= | **`true`** \| **`false`** |
| ⟨constant⟩ | ::= | `()` \| ⟨int⟩ \| ⟨float⟩ \| ⟨bool⟩ |
| ⟨ident⟩ | ::= | $[a-z]+[a-zA-Z0-9]*$ |
| ⟨lidents⟩ | ::= | `()` \| `(`⟨ident⟩$[$`,`⟨ident⟩$]*$ `)` \| ⟨ident⟩ |
| ⟨enum⟩ | ::= | $[A-Z]+[a-zA-Z0-9]*$ |
| ⟨int_op⟩ | ::= | `+` \| `-` \| `*` \| `/` \| **`mod`** |
| ⟨float_op⟩ | ::= | `+.` \| `-.` \| `*.` \| `/.` |
| ⟨bool_op⟩ | ::= | **`&&`** \| **`\|\|`** |
| ⟨comp_op⟩ | ::= | `<` \| `>` \| `<=` \| `>=` \| `=` \| `<>` |
| ⟨binop⟩ | ::= | ⟨int_opt⟩ \| ⟨float_op⟩ \| ⟨bool_op⟩ \| ⟨comp_op⟩ |
| ⟨unop⟩ | ::= | `-` \| `-.` \| **`not`** |
| ⟨eqn⟩ | ::= | ⟨lidents⟩ `=` ⟨expr⟩ |
| ⟨leqns⟩ | ::= | ⟨eqn⟩ \| ⟨eqn⟩`;`⟨leqns⟩ |
| ⟨expr⟩ | ::= | ⟨constant⟩ |
| | | \| ⟨ident⟩ |
| | | \| ⟨enum⟩ |
| | | \| ⟨expr⟩⟨binop⟩⟨expr⟩ |
| | | \| ⟨unop⟩⟨expr⟩ |
| | | \| **`if`** ⟨expr⟩ **`then`** ⟨expr⟩ **`else`** ⟨expr⟩ |
| | | \| ⟨expr⟩`,`⟨expr⟩ |
| | | \| `(`⟨expr⟩`)` |
| ⟨node⟩ | ::= | **`let%node`** ⟨ident⟩ ⟨lidents⟩ **`~return:`**⟨lidents⟩ `=` ⟨leqns⟩ |

FIGURE 3.1 – Syntaxe partielle du langage OCaLustre

Ainsi, l'exemple de la figure 3.2 correspond à la définition du nœud nommé `plus_moins` qui calcule un couple qui correspond à la somme et la différence de ses deux paramètres en entrée, et le tableau associé représente l'évolution des valeurs de chaque flot de ce nœud, au cours de son exécution.

1. La représentation des littéraux et des identificateurs est ici simplifiée. Dans l'implantation réelle du compilateur OCaLustre, tout littéral et identificateur valide en OCaml l'est aussi en OCaLustre.

```
let%node plus_moins (x,y) ~return:(p,m) =
  p = x + y;
  m = x - y
```

| *instant* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | *i* | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| x | 3 | 2 | 6 | 32 | 8 | 26 | 67 | ... | 7 | ... |
| y | 4 | 12 | 42 | 9 | 22 | 7 | 53 | ... | 3 | ... |
| p | 7 | 14 | 48 | 41 | 30 | 33 | 120 | ... | 10 | ... |
| m | -1 | -10 | -36 | 23 | -14 | 19 | 14 | ... | 4 | ... |

FIGURE 3.2 – Exemple de nœud OCaLustre

**Retard initialisé**

Dans un langage impératif, il est courant d'attribuer une valeur à une variable à partir de la valeur qu'elle contient : par exemple, l'incrémentation d'une variable (`x := x + 1`) a implicitement accès à la valeur « précédente » de la variable, afin d'en modifier la valeur « courante ». OCaLustre est basé sur un modèle de programmation plus proche de la programmation fonctionnelle, dans laquelle une variable ne peut être modifiée pendant l'exécution du programme. En réalité, la sémantique de flots de données redéfinit implicitement des *nouvelles* variables à chaque instant, et chacune de ces définitions est elle-même immutable. Il est cependant parfois nécessaire de pouvoir accéder à la valeur antérieure d'une variable afin de lui attribuer une valeur courante (typiquement, incrémenter un compteur). Dans un montage électronique, il peut être utile d'accéder à la valeur antérieur émise par un capteur, afin d'en calculer la différence avec une valeur courante, par exemple pour déterminer la différence de température d'un capteur entre un instant et le suivant, dans le but d'en déduire la vitesse de croissance.

Par conséquent, afin de permettre à un flot d'avoir accès, pendant l'instant $i$, à la valeur d'un flot à l'instant $i-1$, le langage OCaLustre offre alors un opérateur de *retard initialisé*, noté $\ggg$ et semblable à l'opérateur `fby`, pour "*followed-by*" (litt. « *suivi-de* » ) utilisé par le langage à flot de données Lucid [AW77] et des langages synchrones divers, comme Heptagon [Gér13], Lucid Synchrone [CP99], ou Lustre V6 [JRH19]. Cet opérateur, utilisé dans l'expression `0` $\ggg$ `x`, signifie que l'expression a pour valeur la constante à gauche de $\ggg$ (i.e. `0`) au premier instant d'exécution du programme, puis la valeur *précédente* de l'expression à droite de l'opérateur (i.e. `x`) pour tous les instants futurs :

$$k \ggg x \equiv (k, x_0, x_1, x_2, \ldots, x_{i-1}, \ldots)$$

Grâce à cet opérateur, il est aisé de représenter des suites de valeurs. Par exemple, le nœud suivant calcule un flot `n` qui correspond à la suite des entiers naturels :

```
let%node cpt () ~return:(n) =
  n = (0 ⋙ (n+1))
```

En effet, le flot `n` a pour valeur la constante 0 au premier instant, suivie de la valeur précédente de l'expression `n+1` pour chaque instant successif.

De la même façon, l'exemple suivant permet de calculer la suite de Fibonacci :

```
let%node fibonacci () ~return:f =
  f = (0 ⋙ ((1 ⋙ f) + f))
```

Le tableau de la figure 3.3 détaille l'évolution des différentes valeurs des expressions du nœud `fibonacci` au fil du temps.

| *instant* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| `f` | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | ... |
| `(1 ⋙ f)` | 1 | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | ... |
| `(1 ⋙ f) + f` | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | ... |

FIGURE 3.3 – Calcul de la suite de Fibonacci `f` grâce à l'opérateur de retard initialisé

Illustrons l'utilisation de cet opérateur dans le contexte du développement de programmes pour microcontrôleurs. Nous reprenons l'exemple de programme de la section 1.1.2, dans lequel le microcontrôleur fait clignoter une LED à intervalles réguliers.

Un programme OCaLustre très simple permet de reproduire ce comportement. Définissons en effet le type OCaml énuméré et le nœud OCaLustre suivants :

```
type light_state = ON | OFF

let%node blinker () ~return:led =
   led = (ON ⋙ if led = ON then OFF else ON)
```

Le flot `led` est donc initialisé avec la valeur « `ON` » pour allumer la *LED*, et à chaque instant synchrone suivant la lampe prend l'état inverse de son précédent.

L'opérateur ⋙ permet, en outre, de représenter facilement des flots dont la séquence de valeurs se répète de manière cyclique. Par exemple, si on définit un type énuméré `tictactoc` de la façon suivante :

```
type tictactoc = Tic | Tac | Toc
```

Le flot `x` correspond alors à la suite de valeurs (`Tic` , `Tac` , `Toc` , `Tic` , `Tac` , `Toc` ,..., `Tic` , `Tac` , `Toc` ,...) :

```
x = (Tic ⋙ (Tac ⋙ (Toc ⋙ x)))
```

Le nœud `blinker` peut donc également être écrit de la façon suivante :

```
let%node blinker () ~return:led =
   led = (ON ⋙ (OFF ⋙ led))
```

Il est par ailleurs possible de redéfinir des opérateurs classiques du langages Lustre à l'aide de l'opérateur de retard initialisé :

— L'opérateur d'initialisation `->` de Lustre peut être réécrit à l'aide de l'opérateur ⋙ de cette façon :

$$\texttt{x -> y} \equiv \texttt{if (true} \ggg \texttt{false) then x else y}$$

— L'opérateur de retard **pre** pourrait également être défini avec $\ggg$ à condition de fournir une valeur $k$ initiale (du bon type) pour le flot en question :

$$\texttt{pre x} \equiv k \ggg \texttt{x}$$

Notre choix d'utiliser, dans OCaLustre, l'opérateur $\ggg$ au lieu d'une combinaison de ces opérateurs nous permet de libérer la compilation d'un programme OCaLustre de certaines considérations relatives à l'initialisation des flots. En effet, puisque pour tout `x` la valeur du flot **pre** `x` à l'instant 0 est indéfinie (*nil*), il est habituellement important de vérifier, au moment de la compilation d'un programme, que l'opérateur **pre** n'apparaît, dans la définition d'un flot, qu'à droite d'un `->`. Si tel n'était pas le cas, la valeur du flot à l'instant 0 serait inconnue, et le programme aurait un comportement indéterminé. Cette question de l'initialisation des flots, qui ne se pose donc pas en OCaLustre, peut être toutefois résolue statiquement via l'utilisation d'un système de types permettant de représenter la position des **pre** dans les expressions [CP04].

### Flots à portée locale

Il est à noter que, dans la définition d'un nœud OCaLustre, les équations ne doivent pas forcément correspondre à des flots d'entrée ou de sortie. Il peut être utile, pour améliorer la lecture d'un programme, représenter un registre interne, ou bien factoriser certains calculs, de définir des flots dont on considère qu'ils ont une portée locale au nœud. En OCaLustre, un flot local est défini sans mot-clé particulier, sa simple absence de la signature du nœud suffit à en limiter la portée.

Le nœud `fibonacci` de la section précédente peut ainsi être réécrit, pour en simplifier la lecture, en un nœud `fibonacci2` définissant les flots locaux `sum_last` et `previous_f` :

```
let%node fibonacci2 () ~return:(f) =
  f = (0 ⋙ sum_last);
  previous_f = (1 ⋙ f);
  sum_last = previous_f + f
```

### Application de nœud

Afin d'organiser en blocs de code distincts les différents comportements d'un programme, nous introduisons dans le langage un mécanisme d'*application* de nœuds, tel qu'il en existe dans le langage Lustre. L'appel (ou application) de nœud dans un programme OCaLustre permet de factoriser des morceaux de code redondants, et ainsi de réduire l'empreinte mémoire d'un programme. Il suit la syntaxe classique d'application de fonctions du langage OCaml : par exemple, le nœud suivant fait appel au nœud `plus_moins` défini plus haut, avec comme paramètre en entrée le couple `x,y`, et comme sorties les flots `a` et `b` :

```
let%node call_pm (x,y) ~return:(a,b) =
  (a,b) = plus_moins (x,y)
```

Il est à noter que dans la sémantique du langage, chaque appel à un nœud correspond en réalité à l'appel à une *instance* de ce nœud. Chaque point d'appel à un nœud entraîne en effet l'initialisation

implicite d'une instance de ce nœud et, de ce fait, aucun flot n'est partagé entre ces différents points d'appel.

Par exemple, le nœud suivant définit *deux* compteurs s'exécutant de façon concurrente :

```
let%node two_cpt () ~return:(c1,c2) =
  c1 = cpt ();
  c2 = cpt ()
```

Les compteurs internes dans chacun des appels à `cpt` sont distincts, et les flots `c1` et `c2` s'incrémentent alors à la même vitesse, car ils possèdent chacun leur propre registre interne mis à jour à chaque instant synchrone.

### Appels externes à des fonctions OCaml

Les opérateurs intégrés dans OCaLustre sont par nature très simples : ils se limitent aux opérateurs arithmétiques et logiques de base, complétés par quelques opérateurs faisant référence au temps. Pourtant le langage support de cette extension, OCaml, possède une richesse expressive importante, via son implantation de divers paradigmes de programmation (impérative, fonctionnelle, objet) et de constructions puissantes (foncteurs, types de données algébriques généralisés, polymorphisme, . . . ), qui offrent au développeur des outils de haut niveau pour réaliser des programmes complexes. Afin de combiner les aspects synchrones d'un programme, régissant l'interaction entre les divers composants logiciels d'une application, et ses aspects purement algorithmiques, exploitant la richesse du langage support, OCaLustre est enrichi d'un opérateur **`call`** permettant l'appel à une fonction OCaml depuis un nœud OCaLustre.

Par exemple, dans le programme suivant, le nœud synchrone `sqrt_cpt` fait appel à la fonction OCaml `f` qui calcule, *dans un instant synchrone*, la racine carrée du flot `a`.

```
let f x = if x > 0.0 then sqrt x else 0.0

let%node sqrt_cpt () ~return:b =
  a = (0.0 ⋙ (a +. 1.0));
  b = call f a
```

Il est à noter que l'utilisation de **`call`** peut être « dangereuse » en raison du fait qu'elle ouvre OCaLustre à toutes les constructions du langage OCaml, qui sont susceptibles de réaliser des opérations incompatibles avec la sémantique du modèle synchrone [2], ou qui peuvent ne jamais retourner (à cause d'une boucle infinie ou du déclenchement d'une exception). Il est donc de la responsabilité du programmeur de se prémunir de ces comportements erronés. Par sécurité, nous considérerons dans la suite du manuscrit que **`call`** n'est utilisé que pour exécuter du code purement fonctionnel, qui termine.

La syntaxe des expressions d'OCaLustre présentée dans la figure 3.1 est alors étendue avec l'opérateur de retard initialisé, l'application de nœud, et l'opérateur d'application d'une fonction OCaml n-aire :

2. Par exemple, le comportement d'un programme OCaLustre dans lequel deux utilisations de `call` concurrentes modifient une même variable globale mutable est indéterminé.

```
⟨expr⟩ ::= (...)
         | ⟨constant⟩ ⋙ ⟨expr⟩
         | ⟨ident⟩ (⟨expr⟩)
         | call ⟨ident⟩ ⟨expr⟩[ ⟨expr⟩]*
```

### 3.1.2 Horloges synchrones

Chaque composant d'un programme OCaLustre s'exécute de manière concurrente. De ce fait, chaque équation définissant la valeur d'un flot dans un nœud est calculée à chaque instant synchrone, et chacune des expressions contenues dans ces équations est aussi évaluée dans l'instant. Ainsi, l'équation suivante entraîne l'exécution, dans le même instant, à la fois de `e1` et de `e2` — la valeur attribuée à $x$ étant par la suite choisie en fonction de la valeur du booléen `b` :

```
x = if b then e1 else e2
```

Cette sémantique diffère de la sémantique courante des langages de programmation généralistes, pour lesquels l'évaluation de l'opérateur conditionnel *if-then-else* est communément paresseuse. Par exemple, en OCaml, l'évaluation de l'expression suivante entraîne uniquement l'évaluation de la branche *positive* (resp. *négative*) de l'opérateur conditionnel quand `b` est vrai (resp. faux), et le message affiché est `"vrai"` (resp. `"faux"`) :

```
if b then print_string "vrai" else print_string "faux"
```

À l'inverse, en OCaLustre, chaque branche d'une conditionnelle est exécutée à chaque instant : par exemple, si on définit le nœud `count` qui calcule un compteur :

```
let%node count () ~return:cpt =
  cpt = (0 ⋙ (cpt + 1))
```

Alors, dans l'équation suivante, l'incrémentation du compteur dans `count ()` est réalisée *à chaque instant* :

```
x = if b then count () else 0
```

Il est pourtant nécessaire, afin de pouvoir écrire des programmes non triviaux, d'offrir un moyen de conditionner l'exécution de certaines parties d'un programme, et ainsi de récupérer la notion de *flot de contrôle*. Du point de vue de la programmation synchrone, cette idée revient à *ralentir* l'exécution de certaines branches du programme : en effet, on cherche à n'exécuter des expressions qu'à certains instants, et donc à *sous-échantillonner* dans le temps la valeur de certains flots d'un programme OCaLustre.

#### Sous-échantillonnage

Pour conditionner, en OCaLustre, une expression `e` à ne posséder une valeur que lorsqu'un certain flot booléen $b$ est vrai, on utilise l'annotation `[@when` $b$ `]`. Par exemple, l'équation suivante est la déclaration

d'un flot `x` ayant pour valeur celle de l'expression `(y+1)` seulement quand le flot `clk` s'évalue à *true*, et n'ayant pas de valeur sinon :

```
x = (y+1) [@when clk]
```

On dit alors que le flot `x` est *cadencé* par l'horloge `clk`. Une horloge est un flot booléen qui représente la *condition de présence* des flots qu'elle cadence : lorsque la valeur d'une horloge est fausse, tous les flots qu'elle cadence sont considérés comme absents (ils n'ont pas de valeur). Il est alors interdit d'accéder à la valeur de ces flots. Par exemple, l'extrait de programme OCaLustre suivant est incorrect :

```
let%node bad_clocks () ~return:(x,y,w) =
  w = 2;
  y = 3;
  x = w [@when b] + y
```

En effet, le flot `y`, qui n'est cadencé par aucune horloge (on dira qu'il est cadencé par l'*horloge de base* — i.e. l'horloge la plus rapide du nœud, qui est un paramètre implicite de chaque nœud), voit sa valeur additionnée à celle du flot `w`, qui est lui sur l'horloge `b`. Le calcul de la valeur de `x` quand `b` est faux n'a alors pas de sens, car il n'y a alors « rien » à gauche de l'addition[3] noté $\perp$). La totalité des opérateurs binaires d'OCaLustre ne peut être appliquée qu'à des flots étant tous deux cadencés par la même horloge.

Par ailleurs, les flots cadencés par une horloge qui est elle-même absente (car sa propre horloge est fausse) sont égalements absents. Dans l'exemple suivant, le flot `z` n'a donc pas de valeur car son horloge `ck2` est absente :

```
let%node sampled_clock () ~return:(ck1,ck2,z) =
  ck1 = false;
  ck2 = true [@when ck1];
  z = 42 [@when ck2];
```

À l'inverse du sous-échantillonnage *positif*, OCaLustre dispose d'une annotation de sous- échantillonnage *négatif* `[@whennot _ ]`. Il est alors possible de contraindre la présence d'une valeur à la condition qu'un booléen soit *faux*. Ainsi, dans l'extrait suivant les flots `a`, `b` et `c` ont une valeur uniquement si le flot `clk` s'évalue à *false*. On dira qu'ils sont cadencés par l'horloge *not* `clk` (ou $\overline{\texttt{clk}}$) :

```
a = 2 [@whennot clk];
b = 3 [@whennot clk];
c = a + b
```

L'unique opérateur d'OCaLustre autorisant à manipuler des flots qui ne sont pas cadencés par la même horloge est l'opérateur **`merge`** : celui ci permet de « fusionner » des flots cadencés par des horloges complémentaires. Cet opérateur reçoit en premier paramètre une horloge, puis un flot cadencé positivement par cette dernière, puis un flot cadencé négativement par la même horloge. Dans l'exemple suivant le flot k a pour valeur celle du flot *i* quand *d* est vrai, et celle du flot *j* quand *d* est faux :

3. Il est à noter que dans la sémantique du langage, *l'absence* de valeur n'est pas équivalent à la présence de la constante *nil*, ni d'une quelconque valeur par défaut.

```
c = (cpt () < 5);
d = (true ⋙ false) [@when c];
i = 23 [@when d];
j = 45 [@whennot d];
k = merge d i j
```

Les valeurs à l'exécution des différents flots définis dans cet exemple sont données dans la figure 3.4. Il est par ailleurs à noter ici que l'utilisation de l'opérateur **@when** réalise un *échantillonage* des valeurs, et n'induit pas de *retard* dans la calcul de ces valeurs : l'expression (**true** ⋙ **false**) est évaluée à *chaque* instant d'exécution (même quand c est faux)[4], mais la valeur correspondante n'est associée à d que lorsque c est vrai. Nous reviendrons sur la distinction entre échantillonnage et retard lorsque nous aborderons le cas des applications conditionnelles en section 3.1.2.

| *instant* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c | true | true | true | true | true | false | false | false | false | false | false | ... |
| d | true | false | false | false | false | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ... |
| i | 23 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ... |
| j | ⊥ | 45 | 45 | 45 | 45 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ... |
| k | 23 | 45 | 45 | 45 | 45 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ... |

FIGURE 3.4 – Évolution des valeurs de flots cadencés

L'opérateur **merge** peut donc être vu comme le traditionnel opérateur conditionnel *if-then-else* à évaluation paresseuse des langages généralistes. Il importe de préciser que le résultat (le flot k) est présent à la même cadence que l'horloge d'échantillonnage des flots i et j (le flot d — que ce dernier ait pour valeur *true* ou *false*), et n'est pas plus rapide : puisque d n'est présent que quand c est vrai, k ne l'est qu'à cette même condition.

La syntaxe des expressions OCaLustre, étendue avec les annotations de sous-échantillonnage et l'opérateur **merge**, est alors complète :

⟨expr⟩ ::= (...)
| **merge** ⟨ident⟩ ⟨expr⟩ ⟨expr⟩
| ⟨expr⟩ [**@when** ⟨ident⟩]
| ⟨expr⟩ [**@whennot** ⟨ident⟩]

### Cas particulier : les constantes

Dans OCaLustre, les constantes ont la particularité d'être cadencées par une horloge quelconque : on peut considérer que leur type d'horloge est polymorphe, et qu'elles sont donc compatibles avec n'importe quelle horloge. Cette distinction résulte de la volonté de rendre les programmes OCaLustre plus simples à écrire et à relire. En effet, dans l'exemple suivant

4. Ceci en raison du *principe de substitution* de Lustre qui stipule que si un flot $x$ vaut *expr*, alors toute occurence de *expr* dans le programme peut être remplacée par $x$, et vice-versa, sans en changer la sémantique. L'expression (**true** ⋙ **false**) pourrait donc être extraite vers une nouvelle variable, évaluée à chaque instant.

```
let%node ex_const (a,b) ~return:x =
  c = a;
  d = b [@when c];
  v = 12 [@whennot c];
  x = merge c d (4 [@whennot c] + v)
```

une expression comme `((4 [@whennot c])` est relativement « lourde », et n'est pas utile compte tenu du fait que dans ce contexte d'utilisation (en troisième paramètre d'un **merge** dont le premier paramètre est $c$), l'horloge de `4` ne peut qu'être $\bar{c}$. De la même façon, puisque la variable `v` est additionnée à une valeur cadencée par $\bar{c}$, son horloge est également $\bar{c}$.

Le même nœud est ainsi plus lisible lorsque les constantes sont privées de leurs annotations d'horloge :

```
let%node ex_const (a,b) ~return:x =
  c = a;
  d = b [@when c];
  v = 12;
  x = merge c d (4+v)
```

### Application conditionnelle

Lorsqu'un flot est défini comme le résultat de l'appel à un nœud, il est important de distinguer le sous-échantillonnage des paramètres de l'appel du sous-échantillonnage de la valeur de retour. En effet, sous-échantillonner une valeur en entrée sert à *conditionner l'exécution* de l'appel à un nœud, et ainsi à ralentir la fréquence d'exécution du nœud appelé ; tandis que sous-échantillonner la valeur de retour d'un appel à un nœud permet seulement de *limiter la présence* de cette valeur.

Par exemple, nous donnons ci-après la définition d'un nœud `cpt` qui calcule la suite des entiers naturels, modulo `n` :

```
let%node cpt n ~return:c =
  c = (0 ⋙ (c + 1)) mod n
```

Nous définissons ensuite un nœud `call_cpt` qui calcule deux flots : un flot `a`, correspondant à l'appel à `cpt` avec sous-échantillonnage de ses paramètres, et un flot `b` qui correspond à l'appel à `cpt` avec sous-échantillonnage de la valeur de retour :

```
let%node call_cpt ck ~return:(a,b) =
  a = cpt (10 [@when ck]);
  b = (cpt 10) [@when ck]
```

Du point de vue de leurs horloges de cadencement, les flots `a` et `b` sont indiscernables, chacun de ces flots n'étant présent que lorsque le flot `ck` est vrai. Pourtant, leur sémantique est différente : pour le flot `b`, l'appel `(cpt 10)` est réalisé à chaque instant (quelle que soit la valeur de `ck`), et c'est alors la valeur de retour de cet appel qui est sous-échantillonnée par `ck`. À l'inverse, pour le flot `a`, l'exécution de `(cpt 10)` n'est réalisée qu'à condition que `ck` soit vrai. Le compteur n'est alors incrémenté que lorsque `ck` est vrai,

et la fréquence d'exécution de `cpt` est potentiellement plus lente que celle de `call_cpt`. Le tableau de la figure 3.5 illustre la différence entre ces deux sémantiques.

| *instant* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| ck | true | false | true | true | false | false | true | ... |
| `cpt (10 [@when ck])` | 0 | ⊥ | 1 | 2 | ⊥ | ⊥ | 3 | ... |
| `(cpt 10) [@when ck]` | 0 | ⊥ | 2 | 3 | ⊥ | ⊥ | 6 | ... |

FIGURE 3.5 – Différence entre le sous-échantillonnage des paramètres et le sous-échantillonnage de la valeur de retour.

Le sous-échantillonnage des paramètres d'un nœud permet donc de *ralentir* la fréquence d'appel de ce nœud, et ainsi de faire évoluer « moins vite » son état interne. Un tel comportement sera nommé dans ce manuscrit *application conditionnelle*.

Un exemple classique d'utilisation de l'application conditionnelle est l'exemple de la montre [Pla89]. Dans cet exemple, les différents appels à `cpt` sont conditionnés par des horloges à des fréquences de plus en plus lentes, permettant de simuler le calcul des heures, minutes, et secondes [5] :

```
let%node watch (sec) ~return:(hour,min,h,m,s) =
  s = cpt (60 [@when sec]);
  min = (s = 60);
  m = cpt (60 [@when min]);
  hour = (m = 60);
  h = cpt (24 [@when hour])
```

Chaque appel à `cpt` est exécuté à une fréquence distincte :
— Le flot `s` est incrémenté chaque fois que son horloge *sec* est vraie.
— Le flot `m` est incrémenté chaque fois que *min* est vrai (60 fois plus lentement que `s`).
— Le flot `h` est incrémenté chaque fois que *hour* est vrai (60 fois plus lentement que `m`).

La notion d'*application conditionnelle* est à différencier de celle d'*activation conditionnelle*, utilisée par exemple dans le langage Scade (où elle est implantée par l'opérateur `condact` dans les versions antérieures à Scade 6, puis par la construction `activate/every` [Dor08]). L'activation conditionnelle correspond à une application conditionnelle d'un nœud associée à une projection : si l'horloge de l'application conditionnelle est fausse, alors le flot associé conserve la valeur calculée la dernière fois que son horloge était vraie (et si elle n'a encore jamais été vraie, le flot possède une valeur par défaut explicitement renseignée par le programmeur).

Ce mécanisme d'activation conditionnelle peut être implanté dans OCaLustre en mêlant l'application conditionnelle, l'opérateur de fusion, et l'opérateur de décalage. Par exemple, le nœud suivant réalise l'activation conditionnelle d'un appel au nœud cpt :

```
let%node condact_cpt (c) ~return:(x) =
  x = merge c (cpt (10 [@when c])) ((0 ⋙ x) [@whennot c])
```

5. La raison de la présence des horloges de `h` et `m` dans le n-uplet en sortie du nœud sera explicitée lorsque nous aborderons le typage d'horloge en section 3.1.3.

### Détermination de la condition d'application

La condition régissant l'exécution d'un appel de nœud dépend de l'horloge de ses paramètres. Ainsi, l'équation suivante n'entraîne l'exécution de `cpt 60` qu'à condition que `sec` soit vrai :

```
s = cpt (60 [@when sec])
```

Cependant, lorsqu'un nœud possède plusieurs paramètres, la question de savoir sous quelle condition un appel doit être exécuté se pose. En effet, cette condition ne se limite pas au fait que tous les paramètres de l'appel doivent être présents. Par exemple, commençons par définir un nœud `merger_swap` qui « fusionne » deux flots (`x` et `y`) sur des horloges complémentaires (`c` et *not* `c`)), en utilisant l'opérateur **`merge`** avec ses paramètres dans l'ordre inverse :

```
let%node merger_swap (x,y,c) ~return:z =
  z = merge c x y
```

Et considérons désormais l'équation suivante :

```
m = merger_swap (x [@when c], y [@whennot c], c)
```

Les paramètres de l'appel à `merger_swap` ne peuvent jamais tous être présents dans le même instant, puisque `x` et `y` sont cadencés par des horloges complémentaires (quand `x` est présent, `y` est absent, et vice-versa). Imposer la présence de tous les paramètres pour conditionner l'appel constituerait une contrainte trop limitante. En réalité, l'activation d'un nœud ne se fait qu'à condition que certains paramètres soient présents : ceux qui sont, dans le contexte du nœud appelé, sur l'horloge de base (la plus rapide). Puisque l'horloge de base d'un nœud est l'horloge la plus rapide d'un nœud, la présence des paramètres sur cette horloge constitue alors la garde de l'exécution de ce nœud. Par exemple, l'unique paramètre `n` du nœud `cpt` est sur l'horloge de base. C'est alors la présence de ce paramètre qui conditionne l'appel à `cpt`.

Dans le nœud `merger_swap`, c'est le dernier paramètre qui est sur l'horloge de base du nœud. Le flot `c` devra ainsi être présent pour que l'appel `merger_swap (x,y,c)` soit effectué.

Cette sémantique, similaire à celle de Heptagon, est légèrement plus souple que celle de Lustre : en Lustre, le premier paramètre doit être sur l'horloge de base, et les autres paramètres sont potentiellement sur des horloges plus lentes [6]. Lors de l'appel d'un nœud, il est alors vérifié la présence ou l'absence du premier paramètre pour en conditionner l'exécution.

Nous présentons, dans la section suivante, une discipline de typage permettant de modéliser le système des horloges synchrones. Ce système d'horloges synchrones, dont la cohérence est vérifiable au moment de la compilation d'un programme OCaLustre, régit le *bon cadencement* des programmes dans ce langage.

6. Le nœud `merger_swap` devrait donc, en Lustre, avoir *c* comme premier paramètre.

### 3.1.3 Typage d'horloges des programmes

Dans un programme OCaLustre, chaque expression possède une horloge définie explicitement (via l'utilisation des annotations **`@when`** et **`@whennot`**), ou implicitement (telle l'horloge de base du nœud, ou une horloge issue de la composition de deux nœuds sous-échantillonnés). À l'instar des types de données des expressions d'OCaLustre, les informations concernant les horloges de chaque flot sont toutes déductibles à la compilation, par conséquent les horloges synchrones d'OCaLustre peuvent être représentées par un système de types, attribuant statiquement à chaque expression du langage un *type d'horloge*. Ce système est alors associé à un ensemble de règles représentant la cohérence du *cadencement* d'un programme OCaLustre. Ainsi, un programme *bien cadencé* correspond à un programme dont les types d'horloges respectent les règles de ce système, et dire qu'un programme est bien cadencé revient à dire qu'aucun accès à des valeurs absentes n'est possible lors d'un instant synchrone, quels que soient les valeurs des flots manipulés par le programme. Cette section présente la notion de typage d'horloges synchrones par l'intermédiaire de plusieurs exemples qui permettent d'en évaluer la spécificité, tandis que la description formelle de ce système de types sera décrite dans la section suivante.

Un type d'horloge est associé à une grammaire, présentée dans la figure 3.6. Cette grammaire permet de représenter l'horloge de base d'un nœud, ou des horloges correspondant à des sous-échantillonnages de valeurs :

| | | | |
|---|---|---|---|
| $ck$ | ::= | | horloge |
| | \| | $\bullet$ | horloge de base |
| | \| | $ck$ **on** $x$ | sous-échantillonnage positif |
| | \| | $ck$ **onnot** $x$ | sous-échantillonnage négatif |
| | \| | $ck \times ck'$ | n-uplet |
| | \| | $ck \to ck'$ | fonction |

FIGURE 3.6 – Types d'horloges en OCaLustre

— Un flot qui n'est échantillonné par aucune horloge au sein d'un nœud est considéré comme étant sur l'horloge de base du nœud. Sur le modèle du formalisme adopté pour le langage Heptagon [Gér13], on note le type d'horloge de base avec le symbole suivant : $\bullet$ .
— Un flot cadencé positivement par un flot $x$ (lui-même de type $ck$) a pour type d'horloge $ck$ **on** $x$.
— Un flot cadencé négativement par un flot $x'$ (lui-même de type $ck'$) a pour type d'horloge $ck'$ **onnot** $x'$.
— À un n-uplet de flots correspond un n-uplet d'horloges.
— Une instance d'un nœud a un type *fonctionnel* de la forme *horloges des entrées* $\to$ *horloges des sorties*.

La signature d'un nœud est associée à un *schéma de type* (noté $\omega$), c'est-à-dire un type pour lequel des variables peuvent être globalement quantifiées (tel que défini par Damas et Milner dans [DM82]), de la même façon que dans les travaux de Colaço et Pouzet [CP03]. À l'image du typage de données classique décrit à la section précédente, le type d'horloge d'un nœud est un type flèche dont l'élément de gauche correspond au type d'horloge de ses flots d'entrée, et celui de droite au type d'horloge de ses flots de sortie. La signature est annotée avec les noms des entrées $\vec{x}$ et des sorties $\vec{y}$ du nœud correspondant. Le schéma de type associé est quantifié par l'horloge de base du nœud :

$$\omega ::= \forall \bullet . (\vec{x} : ck) \to (\vec{y} : ck')$$

Par exemple, le nœud `sampler` sous-échantillonne son premier paramètre avec son second paramètre :

```
let%node sampler (x,c) ~return:y =
  y = x [@when c]
```

Puisque dans le corps de ce nœud il n'y a pas d'information qui puisse indiquer que les paramètres `x` ou `c` soient sous-échantillonés par une horloge plus lente que celle de base, alors ces paramètres sont considérés comme étant cadencés par l'horloge de base du nœud. La valeur en sortie est elle explicitement conditionnée par la présence de `c`.

La signature de `sampler` est alors :

$$\forall \bullet . ((x : \bullet) \times (c : \bullet)) \rightarrow (y : \bullet \textbf{ on } c).$$

Il est à noter que le nom $c$ dans le type d'horloge des sorties ne signifie pas que le paramètre `c` est lui-même un type, mais que le flot `c` est une valeur booléenne qui sous-échantillonne le flot `y`. Dans la suite, nous appellerons « *supports* » de tels noms de variables qui apparaissent dans le type d'un nœud.

L'opérateur **`merge`** combine des flots dont les types d'horloge sont complémentaires : ($ck$ **on** $x$) et ($ck$ **onnot** $x$). La valeur résultante est cadencée *par l'horloge de* **`x`** puisque l'opérateur de fusion permet en quelque sorte de « remonter » d'un niveau d'horloge. Par exemple, considérons l'exemple suivant qui définit un nœud qui réalise l'application de l'opérateur merge sur ses paramètres d'entrée :

```
let%node merger (c,a,b) ~return:d =
  d = merge c a b
```

Le flot `c` est sur l'horloge de base du nœud ($\bullet$). Les flots `a` et `b` doivent être sur deux horloges complémentaires, cadencées (la première positivement et la seconde négativement) par `c`. Le nœud `merger` a donc la signature suivante :

$$\forall \bullet . ((c : \bullet) \times (a : \bullet \textbf{ on } c) \times (b : \bullet \textbf{ onnot } c)) \rightarrow (d : \bullet)$$

Dans OCaLustre, si un flot en sortie est localement sous-échantillonné par une horloge définie dans le corps d'un nœud, alors il est impératif que son horloge soit également retournée par le nœud concerné. Par exemple, le nœud suivant appelle le nœud `sampler`, et retourne le flot `v` sous-échantillonné avec son horloge `c` :

```
let%node sampler2 (u) ~return:(v,c) =
  v = sampler (u,c);
  c = (true ⋙ (false ⋙ c))
```

La signature de ce nœud est alors $\forall \bullet . (u : \bullet) \rightarrow ((v : \bullet \textbf{ on } c) \times (c : \bullet))$. Cette contrainte permet d'assurer la cohérence du cadencement d'un programme : en effet, l'utilisation d'un flot cadencé par une horloge dont le propre type d'horloge serait inconnu (puisque définit localement à un nœud et donc inaccessible de l'extérieur) entraîne l'incomplétude des informations liées aux conditions de présence de ce flot.

Par exemple, supposons que la définition du nœud suivant soit autorisée :

```
let%node sampler_wrong (u) ~return:(v) =
  v = sampler (u,c);
  c = (true ⋙ (false ⋙ c))
```

La signature de `sampler_wrong` serait donc $\forall \bullet . (u : \bullet) \rightarrow (v : \bullet \textbf{ on } c)$. Mais, comme la valeur de $c$ n'est connue que localement à ce nœud, les informations de cadencement de $v$ sont incomplètes du point de vue de tout nœud qui fait appel à `sampler_wrong` : il y a ici un phénomène d'échappement de portée de la variable locale $c$. Par exemple, dans l'extrait de code suivant, la variable conditionnant la présence de `a_sampled`, censée être passée en premier paramètre de l'opérateur **`merge`** dans la définition du flot `b`, est inaccessible puisqu'elle est locale à `sampler_wrong` :

```
let%node call_sampler_wrong (a) ~return:(b) =
  a_sampled = sampler_wrong (a);
  b = merge XX a_sampled 32 (* problème : XX est inconnue *)
```

### Appels de nœuds et substitution de l'horloge de base

Le typage d'horloge des appels à des nœuds a une sémantique proche du typage de l'application d'une fonction dans un langage de programmation implantant un système de types polymorphe : l'horloge de base ($\bullet$) joue alors le rôle de *variable de type*, et elle est donc instanciée pour chaque appel.

Dans le système de types classique d'un tel langage de programmation (par exemple celui d'OCaml), si une fonction `f` a pour schéma de type $\forall \alpha . \alpha \rightarrow \alpha$, alors dans l'application `f 2` la variable de type $\alpha$ est instanciée avec le type `int`. Le type de `f` dans ce contexte est donc `int` $\rightarrow$ `int` (et le résultat de l'application est alors de type `int`).

De façon analogue, si on définit en OCalustre le nœud suivant :

```
let%node plus_un x ~return:y =
  y = (x + 1)
```

Alors, la signature de ce nœud est $\forall \bullet . (x : \bullet) \rightarrow (y : \bullet)$ et l'horloge de base ($\bullet$) peut être substituée par une horloge plus lente lors de son instantiation.

Par exemple, considérons l'équation suivante :

```
y = plus_un (42 [@when c])
```

Puisque l'expression `42 [@when c]` a pour type d'horloge ($\bullet$ **on** $c$), alors, après instantiation de l'horloge de base, le type d'horloge de `plus_un` dans ce contexte sera le suivant :

$$(\bullet \rightarrow \bullet)[\bullet := \bullet \textbf{ on } c] = (\bullet \textbf{ on } c) \rightarrow (\bullet \textbf{ on } c)$$

Le flot `y` aura alors le type d'horloge du résultat de l'appel : ($\bullet$ **on** $c$).

### Appels de nœuds et substitution des noms de variables

Une particularité du système de types des horloges synchrones est que certains types d'horloges contiennent des noms de variables (par exemple la variable $c$ dans le type ($\bullet$ *on* $c$)). Ces noms, que

nous appellerons *supports*, correspondent aux noms *formels* des flots utilisés comme horloges dans les équations.

Bien sûr, les supports présents dans les signatures de nœuds peuvent différer des noms *effectifs* des arguments d'un appel. Par exemple, considérons la signature du nœud `sampler` :

$$\forall \bullet . ((x : \bullet) \times (c : \bullet)) \rightarrow (y : \bullet \textbf{ on } c)$$

Le flot en sortie du nœud `sampler` a pour type d'horloge ($\bullet$ **on** $c$), avec $c$ le support de son horloge (correspondant au deuxième paramètre de `sampler`), mais dans l'exemple suivant le nom du deuxième argument de l'appel à `sampler` n'est pas `c`, mais `d` :

```
let%node call_sampler (d) ~return:w =
  w = sampler(8,d)
```

Il serait alors faux d'attribuer à `w` le type d'horloge ($\bullet$ **on** $c$), puisque le flot `c` est indéfini dans le corps de ce nœud. Cette différence entre les noms de paramètres *formels* d'un nœud et les noms de ses arguments *effectifs* entraîne la nécessité de réaliser la substitution des supports par les noms réels des variables effectives correspondantes.

Ainsi, dans l'exemple précédent, le type de cette instance de `sampler` devient :

$$((\bullet \times \bullet) \rightarrow (\bullet \textbf{ on } c))[c := d] = (\bullet \times \bullet) \rightarrow (\bullet \textbf{ on } d)$$

et le type d'horloge de `w` est alors ($\bullet$ **on** $d$). Par conséquent, la signature du nœud `call_sampler` est :

$$\forall \bullet \ . (d : \bullet) \rightarrow (w : \bullet \textbf{ on } d)$$

Enfin, considérons l'exemple suivant qui associe la substitution de l'horloge de base et celle des supports :

```
let%node call_sampler_slower (c,e) ~return:w =
  d = c [@when e];
  w = sampler(8 [@when e],d)
```

Les paramètres de l'appel à `sampler` sont chacun sous-échantillonnés par `e`, l'horloge de base de cet appel est donc de type ($\bullet$ **on** $e$). De plus, le nom du second argument de cet appel est `d`, par conséquent le type attribué à `sampler` dans l'équation `w` est alors :

$$((\bullet \times \bullet) \rightarrow (\bullet \textbf{ on } c))\,[c := d]\,[\bullet := \bullet \textbf{ on } e] = ((\bullet \textbf{ on } e) \times (\bullet \textbf{ on } e)) \rightarrow ((\bullet \textbf{ on } e) \textbf{ on } d)$$

Le nœud `call_sampler_slower` possède alors la signature suivante :

$$\forall \bullet . (d : \bullet) \times (e : \bullet) \rightarrow (w : (\bullet \textbf{ on } e) \textbf{ on } d)$$

## 3.2 Spécification du langage OCaLustre formalisée avec Ott et Coq

Cette section est destinée à la description d'une spécification formelle pour le langage OCaLustre. Nous y détaillons en particulier des règles ayant trait à la sémantique et au typage d'un programme

OCaLustre. Ces différentes règles seront exprimées à partir d'une représentation intermédiaire d'un programme OCaLustre, dite *forme normale*. Par ailleurs, la structure d'un programme OCaLustre s'organise sous la forme d'un en-tête comportant des définitions en OCaml (de types ou de fonctions), suivi d'une liste de définitions de nœuds. L'en-tête OCaml (qui est non restreint) ne sera pas formalisé dans ce manuscrit, et la forme normale ne concernera que les nœuds synchrones et les équations qu'ils contiennent.

La grammaire adoptée, ainsi que chacune des règles d'inférence décrites dans cette section ont été formalisées avec l'outil Ott [SNO+10]. Cet outil est destiné à la définition de grammaires et règles d'évaluation de langages de programmations, et est capable d'extraire à partir de cette définition des fichiers lisibles par plusieurs langages et assistants de preuve. Ainsi, la spécification formelle des systèmes de types et de la sémantique décrites dans cette section s'appuie sur l'assistant de preuve Coq [Tea19]. L'affichage des diverses règles inductives dans cette section est pour sa part issue de l'extraction de Ott vers LaTeX.

### 3.2.1 Représentation d'un programme synchrone en forme normale

Les diverses opérations et analyses que nous décrirons formellement dans la suite de ce manuscrit reposent sur une représentation des programmes structurée. Cette représentation, la *forme normale*, résulte d'une procédure consistant en l'application, par le compilateur d'OCaLustre, de plusieurs transformations statiques visant à homogéniser la structure des programmes afin d'en simplifier les analyses et transformations ultérieures. En particulier, la mise en forme normale permet d'extraire les sous-expressions induisant l'utilisation de registres lors de la compilation d'un programme. Nous détaillerons avec précision le processus de mise en forme normale (ou *normalisation*) d'un programme OCaLustre lorsque nous présenterons les diverses étapes de compilation d'un programme OCaLustre, dans le chapitre 4.

#### Grammaire

Dans un souci de cohérence avec des travaux transverses aux nôtres, la grammaire associée à la représentation en forme normale des programmes OCaLustre est inspirée du formalisme introduit dans [BDPR17] et la définition de la syntaxe du langage CoreDF[7]. Un programme OCaLustre est donc, en premier lieu, converti depuis un arbre de syntaxe abstrait (AST) résultant de l'analyse syntaxique du programme source vers un AST *normalisé*, dont les composantes sont issues de la grammaire suivante :

— Un programme synchrone en OCaLustre correspond à une liste de définitions de nœuds :

| | | | |
|---|---|---|---|
| $program, \vec{nodes}$ | ::= | | programme |
| | \| | $\varnothing$ | programme vide |
| | \| | $node;;\vec{nodes}$ | liste de nœuds |

7. À la différence de CoreDF, notre représentation supporte la définition de nœuds ayant plusieurs valeurs en sortie.

— Un nœud est défini à partir de son nom $f$, d'une liste de noms de variables $\vec{x}$[8] qui représente le ou les flots en entrée, d'une liste de noms $\vec{y}$ qui correspond aux flots de sortie, et d'une liste (non vide) d'équations $\vec{eqn}$ qui constitue son corps :

| | | | |
|---|---|---|---|
| *node* | ::= | | définition de nœud |
| | \| | **node** $f(\vec{x})$ **return** $(\vec{y}) = \vec{eqn}$ | |
| $\vec{x}$, $\vec{y}$ | ::= | | noms de variables |
| | \| | () | liste vide |
| | \| | $y$ | unique variable |
| | \| | $y, \vec{y}$ | multiples variables |
| $\vec{eqn}$ | ::= | | liste d'équations |
| | \| | $[eqn]$ | unique |
| | \| | $eqn;\ \vec{eqn}$ | multiples |

— Une équation, de la forme *nom(s) = expression*, correspond à la définition d'une ou plusieurs variables de flots. Ces flots peuvent correspondre à une expression de contrôle *ce*, à l'utilisation de l'opérateur ⋙ (dans un souci d'harmonisation avec d'autres langages synchrones, cet opérateur sera représenté **fby** en forme normale), à l'application d'un nœud $f$ avec pour paramètre la liste (non vide) d'expressions $\vec{e}$, ou à l'utilisation de l'opérateur `call` pour appliquer une fonction OCaml :

| | | | |
|---|---|---|---|
| *eqn* | ::= | | équation |
| | \| | $y = ce$ | expression |
| | \| | $y = k$ **fby** $e$ | fby |
| | \| | $\vec{y} = f(\vec{e})$ | application |
| | \| | $y =$ **call** f $e_0\, e_1\, ..\, e_{n-1}$ | application d'une fonction OCaml |
| $\vec{e}$ | ::= | | liste d'expressions |
| | \| | $[e]$ | unique |
| | \| | $e, \vec{e}$ | multiples |

— Les expressions de contrôle correspondent à l'utilisation de l'opérateur conditionnel **if** − **then** − **else**, de l'opérateur de fusion **merge**, ou à des expressions « simples » $e$ :

| | | | |
|---|---|---|---|
| *ce* | ::= | | expression de contrôle |
| | \| | $e$ | expression |
| | \| | **merge** $x\ ce\ ce'$ | fusion |
| | \| | **if** $e$ **then** $ce$ **else** $ce'$ | alternative |

8. Des flèches situées au dessus d'une certaine catégorie syntaxique représenteront tout au long de ce manuscrit une liste d'éléments de cette catégorie.

— Enfin, les expressions « simples » correspondent à la valeur *unit* (représentée par la notation standard « () » ), aux constantes ($k$), variables ($x$), constructeurs de types énumérés ($X_i$), application des opérateurs de sous-échantillonnage positif ou négatif (**when** et **whennot**), application d'un opérateur arithmétique préfixe unaire ($\square$), ainsi qu'à l'application d'un opérateur arithmétique ou logique binaire infixe ($\diamond$) :

| | | | |
|---|---|---|---|
| $e$ | ::= | | expression |
| | \| | () | unit |
| | \| | $k$ | constante |
| | \| | $x$ | variable |
| | \| | $X_i$ | constructeur de type |
| | \| | $e \diamond e'$ | opération binaire |
| | \| | $\square\, e$ | opération unaire |
| | \| | $e$ **when** $x$ | échantillonnage positif |
| | \| | $e$ **whennot** $x$ | échantillonnage négatif |
| $k$ | ::= | | constante |
| | \| | *int_literal* | entier |
| | \| | *float_literal* | flottant |
| | \| | *bool_literal* | booléen |
| $\diamond$ | ::= | | opérateur binaire |
| | \| | $\diamond_{int}$ | opérateur entier |
| | \| | $\diamond_{float}$ | opérateur flottant |
| | \| | $\diamond_{comp}$ | opérateur de comparaison |
| | \| | $\diamond_{bool}$ | opérateur booléen |
| $\square$ | ::= | | opérateur unaire |
| | \| | $-$ | opposée entière |
| | \| | $-.$ | opposée flottante |
| | \| | **not** | négation booléenne |

### 3.2.2 Typage et cadencement

Le système de types d'un langage de programmation est un ensemble de règles qui attribuent un *type* aux diverses constructions d'un programme de ce langage. Les types les plus généralement considérés, dans le domaine de la programmation informatique, concernent ceux qui représentent la nature des données manipulées : entiers, booléens, nombres flottants, listes d'entiers, fonctions, instances d'objets, etc.

Toutefois, un système de types peut également définir des règles qui ont trait à d'autres aspects d'un programme que la nature des valeurs calculées : il est possible, par exemple, de construire un système de types qui soit dédié non pas à la représentation de la nature des valeurs que « portent » les différentes variables et expressions, mais à celle des propriétés mises en avant dans certains types d'applications, comme par exemple le niveau de sécurité des différentes constructions d'un programme [Wal00].

L'ensemble de règles de typage, qui régissent la cohérence d'un système de type, permet de distinguer les programmes bien typés (qui respectent ces règles) des programmes dont le typage est incorrect (par exemple, dans lequel des conversions de types implicites interdites sont réalisées).

Nous présentons dans cette section le typage d'un programme OCaLustre à l'aune de deux systèmes de type qui offrent des informations bien différentes. Le premier système de types que nous décrivons est un système classique, qui attribue à des valeurs ou des expression du langage, des types de données représentant la nature de l'information qui est transmise : flots d'entiers, flots de booléens, flots de flottants, etc. Ce système de types repose sur le système du langage support et en partage ainsi toutes les caractéristiques : ainsi, dans OCaLustre, les données sont, à l'instar d'OCaml, fortement typées, et leurs types sont statiquement inférés à la compilation des programmes.

Le second système de types que nous abordons dans cette section est le système de types des horloges synchrones. Ce système, à la différence du système de types précédent, ne représente pas la nature des valeurs manipulées par les diverses composantes d'un programme, mais la présence (ou l'absence) de valeurs. Le typage des horloges synchrones est, à l'instar du typage des données d'OCaLustre, un système de typage statique fort, avec inférence des horloges des flots lors de la compilation des programmes.

L'avantage de l'implantation de tels systèmes, avec cette contrainte de typage statique fort, est qu'un très grand nombre de bugs potentiels, issus d'incohérences lors de l'écriture, par le programmeur, de certaines expressions du langage, peuvent ainsi être detectés au moment de la compilation, avant même que le programme ait pu être transféré au microcontrôleur. Ces systèmes de types, qui régissent chacun deux aspects distincts de la cohérence des programmes OCaLustre, contribuent ainsi à la sûreté des programmes OCaLustre.

### 3.2.3 Règles de typage des programmes

À l'instar de son langage cible OCaml, l'extension de langage OCaLustre correspond à un langage de programmation statiquement typé, implantant un mécanisme d'inférence de types à la compilation. Chaque variable d'un programme OCaLustre possède un type dont la nature peut être déterminée pendant la compilation. Il est à noter cependant que le système de types d'OCaLustre est monomorphe, contrairement au système polymorphe d'OCaml.

Dire qu'un flot $x \equiv (x_1, x_2, ..., x_i, ...)$ est de type « *flot de t* » (noté simplement « *t* ») revient à dire que tous les éléments de la séquence de valeurs à laquelle il correspond sont de type $t$. À l'image des listes homogènes du langage OCaml, un flot ne peut « porter » des valeurs dont le type change au fil du temps. Par conséquent, la création d'un flot dont la valeur peut changer de type d'un instant à l'autre , comme le flot $(0, 2, 4, \mathit{true}, 10, 4.5, 14, \dots)$, est donc impossible dans OCaLustre.

Les types manipulés dans OCaLustre peuvent être des types de base (int, bool, float), des types « flèches » (les types des nœuds), des types énumérés (représentés par l'ensemble de leurs constructeurs $X_i$), ou bien des n-uplets de types (correspondant au type d'une liste d'expressions) :

$$t ::= \mathit{unit} \mid \mathit{int} \mid \mathit{bool} \mid \mathit{float} \mid t \rightarrow t' \mid \{X_1, \dots, X_n\} \mid t_1 \times ... \times t_n$$

À partir de la représentation en forme normale des programmes OCaLustre, nous définissons alors formellement les conditions qui régissent le bon typage d'un programme.

Soit $\Gamma$ un environnement de typage local, c'est-à-dire une fonction qui à chaque variable d'un nœud OCaLustre associe son type : par exemple, si le flot $x$ est de type $\mathit{int}$, alors $\Gamma(x) = \mathit{int}$. On note $\Gamma \vdash e : t$ le

prédicat (appelé *jugement de typage*) indiquant que dans l'environnement de typage $\Gamma$ l'expression $e$ est de type $t$.

Nous définissons à partir de tels jugements des règles d'inférence représentant la cohérence du typage des expressions d'OCaLustre. De telles règles, de la forme

$$\frac{P_1 \qquad \dots \qquad P_n}{Q}$$

permettent d'établir qu'à partir des conditions en *prémisse* $P_1$ à $P_n$, on peut déduire le *jugement* $Q$ .

**Typage des expressions :** La figure 3.7 représente les règles de typage des expressions OCaLustre. Par exemple, la règle de typage de l'application d'un opérateur arithmétique entier $\diamond_{int}$ (par exemple l'opérateur +) décrit le fait que cette dernière ne peut être réalisée qu'entre deux entiers, et le résultat est lui-même un entier :

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \diamond_{int} e_2 : int}$$

Il est à noter qu'il est ici supposé, pour la règle de typage des opérateurs de comparaison, qu'aucune comparaison de valeurs de type fonctionnels n'est réalisée dans un programme OCaLustre. Ceci est laissé à la responsabilité du programmeur, qui doit s'assurer de ne pas réaliser une telle opération [9]. Comme nous reposons sur une exécution du code OCaml généré par compilation d'OCaLustre, la fonction de comparaison est polymorphe et son usage sur des valeurs de types fonctionnels lève une exception. Toute exception dans un programme OCaLustre entraîne l'arrêt de son exécution.

Par ailleurs, le jugement de la forme $t = \{X_1, \dots X_n\}$ signifie que le type $t$ a été préalablement défini (dans l'en-tête OCaml) comme un type énuméré composé des constructeurs $X_1$ à $X_n$.

Enfin, nous distinguons les règles de typage d'une expression « simple » de celles d'une expression conditionnelle en annotant pour les règles portant sur les expressions conditionnelles le symbole « *thèse* » ($\vdash$) de la façon suivante : $\vdash_{ce}$.

Les règles relatives au typage des équations et des nœuds OCaLustre sont données dans la figure 3.8. Dans la suite, nous les passons en revue afin de les décrire avec précision.

**Typage des équations :** Soit $\mathcal{G}$ un environnement de typage *global* qui associe à chaque fonction OCaml et à chaque nœud OCaLustre son type (un type « flèche » de la forme *type des entrées* $\rightarrow$ *type des sorties*). Une équation dans le corps d'un nœud OCaLustre correspondant à l'application d'un autre nœud $\vec{y} = f(\vec{e})$ est bien typée si et seulement si $f$ a pour signature un type $t_1 \rightarrow t_2$, le paramètre $\vec{e}$ de l'application est de type $t_1$ et le résultat $\vec{y}$ est de type $t_2$ :

$$\frac{\Gamma \vdash \vec{y} : t_2 \qquad \mathcal{G}(f) = t_1 \rightarrow t_2 \qquad \Gamma \vdash \vec{e} : t_1}{\mathcal{G}, \Gamma \vdash \vec{y} = f(\vec{e})}$$

9. Tout comme l'application de fonctions partielles par l'intermédiaire de `call` ou de division par zéro sont proscrites.

$\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash () : unit} \qquad \frac{}{\Gamma \vdash int\_literal : int} \qquad \frac{}{\Gamma \vdash bool\_literal : bool} \qquad \frac{}{\Gamma \vdash float\_literal : float}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{t = \{X_1, ..., X_n\}}{\Gamma \vdash X_i : t}$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \diamond_{int} e_2 : int} \qquad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \diamond_{bool} e_2 : bool}$$

$$\frac{\Gamma \vdash e_1 : float \quad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \diamond_{float} e_2 : float} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \diamond_{comp} e_2 : bool}$$

$$\frac{\Gamma \vdash e : int}{\Gamma \vdash -e : int} \qquad \frac{\Gamma \vdash e : float}{\Gamma \vdash -. e : float} \qquad \frac{\Gamma \vdash e : bool}{\Gamma \vdash \textbf{not}\, e : bool}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma(x) = bool}{\Gamma \vdash e\, \textbf{when}\, x : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma(x) = bool}{\Gamma \vdash e\, \textbf{whennot}\, x : t}$$

$\boxed{\Gamma \vdash \vec{e} : t}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash [e] : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash \vec{e} : t'}{\Gamma \vdash e, \vec{e} : t \times t'}$$

$\boxed{\Gamma \vdash_{ce} ce : t}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash_{ce} e : t} \qquad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash_{ce} ce_2 : t \quad \Gamma \vdash_{ce} ce_3 : t}{\Gamma \vdash_{ce} \textbf{if}\, e_1\, \textbf{then}\, ce_2\, \textbf{else}\, ce_3 : t}$$

$$\frac{\Gamma(x) = bool \quad \Gamma \vdash_{ce} ce_1 : t \quad \Gamma \vdash_{ce} ce_2 : t}{\Gamma \vdash_{ce} \textbf{merge}\, x\, ce_1\, ce_2 : t}$$

FIGURE 3.7 – Règles de typage des expressions d'OCaLustre

$\boxed{\Gamma \vdash \vec{y} : t}$

$$\frac{}{\Gamma \vdash () : unit} \qquad \frac{\Gamma(y) = t}{\Gamma \vdash y : t} \qquad \frac{\Gamma \vdash y : t \quad \Gamma \vdash \vec{y} : t'}{\Gamma \vdash y, \vec{y} : t \times t'}$$

$\boxed{\mathcal{G}, \Gamma \vdash eqn}$

$$\frac{\Gamma \vdash y : t \quad \Gamma \vdash_{ce} ce : t}{\mathcal{G}, \Gamma \vdash y = ce} \qquad \frac{\Gamma \vdash y : t \quad \Gamma \vdash k : t \quad \Gamma \vdash e : t}{\mathcal{G}, \Gamma \vdash y = k \,\mathbf{fby}\, e}$$

$$\frac{\Gamma \vdash \vec{y} : t_2 \quad \mathcal{G}(f) = t_1 \to t_2 \quad \Gamma \vdash \vec{e} : t_1}{\mathcal{G}, \Gamma \vdash \vec{y} = f(\vec{e})}$$

$$\frac{\Gamma \vdash y : t_n \quad \mathcal{G}(\mathtt{f}) = t_0 \to t_1 \to \ldots \to t_{n-1} \to t_n \quad \Gamma \vdash e_0 : t_0 \quad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_{n-1} : t_{n-1}}{\mathcal{G}, \Gamma \vdash y = \mathbf{call}\, \mathtt{f}\, e_0\, e_1 \ldots e_{n-1}}$$

$\boxed{\mathcal{G}, \Gamma \vdash \vec{eqn}}$

$$\frac{\mathcal{G}, \Gamma \vdash eqn}{\mathcal{G}, \Gamma \vdash [eqn]} \qquad \frac{\mathcal{G}, \Gamma \vdash eqn \quad \mathcal{G}, \Gamma \vdash \vec{eqn}}{\mathcal{G}, \Gamma \vdash eqn;\ \vec{eqn}}$$

$\boxed{\mathcal{G} \vdash node : t}$

$$\frac{\mathcal{G}, \Gamma \vdash \vec{eqn} \quad \Gamma \vdash \vec{x} : t_1 \quad \Gamma \vdash \vec{y} : t_2}{\mathcal{G} \vdash \mathbf{node}\, f(\vec{x})\, \mathbf{return}\, (\vec{y}) = \vec{eqn} : t_1 \to t_2}$$

$\boxed{\mathcal{G} \vdash program}$

$$\frac{}{\mathcal{G} \vdash \varnothing} \qquad \frac{\mathcal{G} \vdash \mathbf{node}\, f(\vec{x})\, \mathbf{return}\, (\vec{y}) = \vec{eqn} : t_1 \to t_2 \quad (f : t_1 \to t_2) \uplus \mathcal{G} \vdash \vec{nodes}}{\mathcal{G} \vdash \mathbf{node}\, f(\vec{x})\, \mathbf{return}\, (\vec{y}) = \vec{eqn};; \vec{nodes}}$$

FIGURE 3.8 – Règles de typage des équations et des nœuds

Une équation qui correspond à l'application de l'opérateur « followed-by » est bien typée à condition que la constante à gauche de l'opérateur $\ggg$ et l'expression à droite de ce dernier soient de même type, et que la variable à gauche de l'opérateur d'égalité soit aussi du même type :

$$\frac{\Gamma \vdash y : t \quad \Gamma \vdash k : t \quad \Gamma \vdash e : t}{\mathcal{G}, \Gamma \vdash y = k \,\mathbf{fby}\, e}$$

L'application d'une fonction OCaml est de type $t_n$ si et seulement si la fonction OCaml est de type $t_0 \to \cdots \to t_{n-1} \to t_n$ et ses paramètres consécutifs sont de type $t_0$ à $t_{n-1}$ :

$$\frac{\Gamma \vdash y : t_n \quad \mathcal{G}(\mathtt{f}) = t_0 \to t_1 \to \ldots \to t_{n-1} \to t_n \quad \Gamma \vdash e_0 : t_0 \quad \Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_{n-1} : t_{n-1}}{\mathcal{G}, \Gamma \vdash y = \mathbf{call}\, \mathtt{f}\, e_0\, e_1 \ldots e_{n-1}}$$

Il est à noter toutefois que cette règle induit un léger glissement sémantique : nous passons des *flots* à

des fonctions OCaml qui sont censées être uniquement capables de manipuler des valeurs « standards ». Par exemple, une valeur de type « flot de `int` » dans OCaLustre est considérée par la fonction OCaml appelée comme une valeur de type `int`. Cette distinction ne pose cependant pas de problème compte tenu du fait qu'à chaque instant du programme un flot ne possède qu'une seule valeur (un « flot de `int` » est donc réduit à un simple `int`) [10]. Formellement, l'opérateur **`call`** peut être vu comme une fonction *map* qui « *lifte* » une fonction OCaml pour qu'elle puisse s'appliquer à des flots :

$$\texttt{call}\ f\ e_0\ e_1\ \ldots\ e_{n-1} \equiv (map\ f)\ e_0\ e_1\ \ldots\ e_{n-1}$$

Toute autre équation $y = ce$ est bien typée si l'expression $ce$ qui lui est associée est bien typée et que la variable $y$ est du même type dans $\Gamma$ :

$$\frac{\Gamma \vdash y : t \qquad \Gamma \vdash_{ce} ce : t}{\mathcal{G}, \Gamma \vdash y = ce}$$

Une liste d'équations (le corps d'un nœud) est bien typée dès lors que toutes les équations qu'elle contient sont bien typées :

$$\frac{\mathcal{G}, \Gamma \vdash eqn}{\mathcal{G}, \Gamma \vdash [eqn]} \qquad \frac{\mathcal{G}, \Gamma \vdash eqn \qquad \mathcal{G}, \Gamma \vdash \vec{eqn}}{\mathcal{G}, \Gamma \vdash eqn;\ \vec{eqn}}$$

**Typage d'un nœud et d'un programme synchrone :** Le type d'un nœud OCaLustre est similaire au type d'une fonction dans OCaml. C'est un type flèche $t_1 \rightarrow t_2$ avec $t_1$ le type des flots qui peuvent lui être passés en entrée, et $t_2$ le type des valeurs qu'il calcule en sortie :

$$\frac{\mathcal{G}, \Gamma \vdash \vec{eqn} \qquad \Gamma \vdash \vec{x} : t_1 \qquad \Gamma \vdash \vec{y} : t_2}{\mathcal{G} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn} : t_1 \rightarrow t_2}$$

Enfin, un programme synchrone (i.e. une liste de nœuds) est bien typé à condition que chaque nœud qu'il contient soit bien typé :

$$\frac{}{\mathcal{G} \vdash \varnothing} \qquad \frac{\mathcal{G} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn} : t_1 \rightarrow t_2 \qquad (f : t_1 \rightarrow t_2) \uplus \mathcal{G} \vdash \vec{nodes}}{\mathcal{G} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn};; \vec{nodes}}$$

Le jugement de typage d'un programme synchrone est initialement invoqué avec l'environnement de typage qui correspond aux fonctions OCaml précédemment définies dans l'en-tête du programme. Les environnements de typage entre les fonctions OCaml et les nœuds sont donc partagés, et nous supposerons donc que les espaces de noms entre les fonctions et les nœuds sont différents et qu'il n'y a donc pas de conflits de noms (qui entraîneraient des phénomènes de masquage).

Les règles de typage de l'extension OCaLustre sont suffisamment proches des règles de typage d'OCaml pour qu'un programme OCaLustre bien typé soit bien typé après sa traduction vers OCaml et

10. Ce phénomène se retrouve aussi lorsqu'il est question de « brancher » un programme OCaLustre à des fonctions qui calculent les entrées et les sorties d'un instant synchrone.

pour que la génération d'un programme OCaml mal typé dénote le fait que le programme OCaLustre d'origine est mal typé. De ce fait, grâce à la *correction du typage vis-à-vis de la traduction vers OCaml* (dont nous présenterons la preuve dans la section 5.1), les règles énoncées dans cette section ne nécessitent pas, pour maintenir la sûreté des programmes, d'être explicitement vérifiées par le compilateur d'OCaLustre. En effet, nous tirons profit du langage cible dont le compilateur réalise statiquement la vérification et l'inférence de types. Cette analyse se fait donc, dans notre situation, après traduction du code du langage OCaLustre vers un code OCaml classique. Le compilateur OCaml est alors capable de faire remarquer au développeur les erreurs de typage des programmes, y compris lorsqu'ils proviennent du code OCaLustre.

Par ailleurs, puisque OCaLustre est réalisé avec des outils faisant partie de l'écosystème de compilation d'un programme OCaml, il est tout à fait compatible avec des d'autres outils puissants d'analyse statique tel que l'outil Merlin [BRS18] qui offre de nombreuses fonctionnalités pour simplifier le développement de programmes OCaml (tel qu'un mécanisme d'auto-completion sensible au contexte, ou l'accès « en direct » à des informations relatives au typage) en s'intégrant dans des éditeurs de texte variés (Vim, Emacs, Atom ...). Une telle compatibilité permet d'offrir à l'utilisateur d'OCaLustre un retour immédiat et précis quant à la position des erreurs relatives (entre autres) à la cohérence de typage et du cadencement de ses programmes, et ce au sein même de son éditeur de texte favori.

### 3.2.4 Règles de bon cadencement des programmes

La cohérence du système de types des horloges synchrones régit le *bon cadencement* d'un programme OCaLustre. Ce bon cadencement est défini à partir d'un ensemble de règles de typages dont le respect induit la sûreté de cadencement d'un programme.

À l'instar de la formalisation du typage des programme OCaLustre, nous définissons un *jugement de cadencement* comme un prédicat de la forme $\mathcal{H}, \mathcal{C} \vdash eqn$ signifiant que dans un environnement de cadencement global $\mathcal{H}$ (qui associe à chaque nom de nœud son horloge) et un environnement de cadencement local $\mathcal{C}$ (qui associe une horloge à chaque nom de variable d'un nœud), une équation *eqn* est bien cadencée. Pareillement, un jugement $\mathcal{C} \vdash e : ck$ stipule que dans l'environnement de typage $\mathcal{C}$ l'expression $e$ a pour type d'horloge $ck$.

À partir de tels jugements de typages, nous décrivons alors des règles d'inférence qui établissent formellement les conditions nécessaires au bon cadencement des programmes, nœuds, équations, et expressions. Ces règles sont dérivées d'un système de types considérant les horloges comme des types abstraits de données [CP03], implanté dans le langage *Lucid Synchrone*. Notre solution, moins expressive car elle n'autorise pas l'existence de *plusieurs* variables de types d'horloge au sein d'un nœud mais juste une seule (l'horloge de base $\bullet$), simplifie néanmoins grandement le modèle de compilation séparée des nœuds, sans limiter particulièrement le pouvoir expressif du langage. Le système de types d'OCaLustre est ainsi plus proche du système de types du langage synchrone *Heptagon*, à mi-chemin entre le système d'horloges de Lustre et celui de Lucid Synchrone.

Les règles de cadencement des expressions du langage OCaLustre sont présentées dans la figure 3.9, et celles qui sont relatives aux équations et aux nœuds sont données dans la figure 3.10. Nous décrivons dans la suite les règles principales qui régissent le bon cadencement d'un programme OCaLustre.

$\boxed{\mathcal{C} \vdash e : ck}$

$$\frac{}{\mathcal{C} \vdash () : ck} \quad \frac{}{\mathcal{C} \vdash k : ck} \quad \frac{}{\mathcal{C} \vdash X_i : ck} \quad \frac{\mathcal{C}(x) = ck}{\mathcal{C} \vdash x : ck}$$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash \square\, e : ck} \quad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e' : ck}{\mathcal{C} \vdash e \diamond e' : ck}$$

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\ \mathbf{when}\ x : ck\ \mathbf{on}\ x} \quad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\ \mathbf{whennot}\ x : ck\ \mathbf{onnot}\ x}$$

$\boxed{\mathcal{C} \vdash \vec{e} : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash [e] : ck} \quad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash \vec{e} : ck'}{\mathcal{C} \vdash e, \vec{e} : ck \times ck'}$$

$\boxed{\mathcal{C} \vdash_{ce} ce : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash_{ce} e : ck}$$

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash_{ce} ce : ck \quad \mathcal{C} \vdash_{ce} ce' : ck}{\mathcal{C} \vdash_{ce} \mathbf{if}\ e\ \mathbf{then}\ ce\ \mathbf{else}\ ce' : ck} \quad \frac{\mathcal{C} \vdash x : ck \quad \mathcal{C} \vdash_{ce} ce : ck\ \mathbf{on}\ x \quad \mathcal{C} \vdash_{ce} ce' : ck\ \mathbf{onnot}\ x}{\mathcal{C} \vdash_{ce} \mathbf{merge}\ x\ ce\ ce' : ck}$$

FIGURE 3.9 – Règles de cadencement des expressions

En OCaLustre, une constante est systématiquement bien cadencée, elle est considérée comme bien typée quel que soit le type d'horloge :

$$\frac{}{\mathcal{C} \vdash k : ck}$$

Le type d'horloge d'une variable est issu de l'environnement de typage :

$$\frac{\mathcal{C}(x) = ck}{\mathcal{C} \vdash x : ck}$$

L'utilisation d'un quelconque opérateur impose à ses opérandes de posséder le même type d'horloge. Par exemple, la règle suivante définissant le bon cadencement d'un opérateur arithmétique ou logique quelconque ($\diamond$) stipule que, dans l'environnement de cadencement $\mathcal{C}$, si chaque opérande de l'opérateur $\diamond$ est cadencée par l'horloge $ck$, alors son résultat est lui même cadencé par $ck$ :

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e' : ck}{\mathcal{C} \vdash e \diamond e' : ck}$$

$\boxed{\mathcal{C} \vdash \vec{y} : ck}$

$$\frac{}{\mathcal{C} \vdash () : \bullet} \qquad \frac{\mathcal{C}(y) = ck}{\mathcal{C} \vdash y : ck} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{C} \vdash y, \vec{y} : ck \times ck'}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash eqn}$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash_{ce} ce : ck}{\mathcal{H}, \mathcal{C} \vdash y = ce} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e_0 : ck \quad \mathcal{C} \vdash e_1 : ck \quad \ldots \quad \mathcal{C} \vdash e_{n-1} : ck}{\mathcal{H}, \mathcal{C} \vdash y = \textbf{call}\ \mathtt{f}\ e_0\, e_1 \ldots e_{n-1}}$$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e : ck}{\mathcal{H}, \mathcal{C} \vdash y = k\ \textbf{fby}\ e} \qquad \frac{\mathcal{H}(f) = \omega \quad ck'_1 \to ck'_2 = inst(\omega) \quad \mathcal{C} \vdash \vec{e} : ck'_1 \quad \mathcal{C} \vdash \vec{y} : ck'_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y} = f(\vec{e})}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash \vec{eqn}}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash eqn}{\mathcal{H}, \mathcal{C} \vdash [eqn]} \qquad \frac{\mathcal{H}, \mathcal{C} \vdash eqn \quad \mathcal{H}, \mathcal{C} \vdash \vec{eqn}}{\mathcal{H}, \mathcal{C} \vdash eqn;\ \vec{eqn}}$$

$\boxed{\mathcal{H} \vdash node : \omega}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash \vec{eqn} \quad \mathcal{C} \vdash \vec{x} : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{H} \vdash \textbf{node}\, f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn} : \forall \bullet .\ (\vec{x} : ck) \to (\vec{y'} : ck')}$$

$\boxed{\mathcal{H} \vdash program}$

$$\frac{}{\varnothing \vdash \varnothing} \qquad \frac{\mathcal{H} \vdash \textbf{node}\, f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn} : \omega \quad (f : \omega) \uplus \mathcal{H} \vdash \vec{nodes}}{\mathcal{H} \vdash \textbf{node}\, f(\vec{x})\ \textbf{return}\ (\vec{y}) = \vec{eqn};; \vec{nodes}}$$

FIGURE 3.10 – Règles de cadencement des équations et des nœuds

L'utilisation de l'opérateur **merge** contraint son deuxième et son troisième paramètre à être cadencés positivement et négativement par son premier paramètre. Le résultat est sur l'horloge du premier paramètre :

$$\frac{\mathcal{C} \vdash x : ck \quad \mathcal{C} \vdash_{ce} ce : ck\ \textbf{on}\ x \quad \mathcal{C} \vdash_{ce} ce' : ck\ \textbf{onnot}\ x}{\mathcal{C} \vdash_{ce} \textbf{merge}\ x\ ce\ ce' : ck}$$

L'application d'un nœud $\vec{y} = f(\vec{e})$ est bien cadencée à condition que le type d'horloge $ck_1 \to ck_2$ soit une *instance* de la signature $\omega$ de $f$ (i.e. un type d'horloge dans lequel les substitutions nécessaires à l'application conditionnelle et à la cohérence des noms de supports ont été effectuées), que les paramètres $\vec{e}$ de l'application soient de type d'horloge $ck_1$ et que le résultat ait pour type d'horloge $\vec{ck_2}$ :

$$\frac{\mathcal{H}(f) = \omega \quad ck'_1 \to ck'_2 = inst(\omega) \quad \mathcal{C} \vdash \vec{e} : ck'_1 \quad \mathcal{C} \vdash \vec{y} : ck'_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y} = f(\vec{e})}$$

L'application d'une fonction OCaml est quant à elle bien cadencée si *tous* ses paramètres ont le même type d'horloge. L'horloge du résultat est également identique à l'horloge des paramètres :

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e_0 : ck \quad \mathcal{C} \vdash e_1 : ck \quad \ldots \quad \mathcal{C} \vdash e_{n-1} : ck}{\mathcal{H}, \mathcal{C} \vdash y = \textbf{call}\ \mathtt{f}\, e_0\, e_1 \ldots e_{n-1}}$$

Un nœud est bien cadencé à condition que les équations qu'il contient, ainsi que ses paramètres en entrée et en sortie soient tous bien cadencés. Sa signature correspond alors à un schéma de type quantifié par l'horloge de base du nœud. C'est un type « flèche » des paramètres en entrée du nœud vers ses paramètres en sortie, tous annotés par leurs types d'horloge :

$$\frac{\mathcal{H}, \mathcal{C} \vdash \vec{eqn} \quad \mathcal{C} \vdash \vec{x} : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{H} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn} : \forall \bullet . (\vec{x} : ck) \rightarrow (\vec{y'} : ck')}$$

Enfin, un programme est bien cadencé si tous ses nœuds sont bien cadencés :

$$\frac{}{\varnothing \vdash \varnothing} \qquad \frac{\mathcal{H} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn} : \omega \quad (f : \omega) \uplus \mathcal{H} \vdash \vec{nodes}}{\mathcal{H} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn};; \vec{nodes}}$$

Un programme respectant ces diverses règles de typage est assuré de ne pas accéder, lors de son exécution, à des valeurs de flots absents ($\bot$), et ainsi de ne pas avoir un comportement imprévisible et erroné. Nous détaillerons dans la section 5.2 le procédé consistant en la vérification automatique de ces diverses règles au moment de la compilation d'un programme OCaLustre.

### 3.2.5 Sémantique opérationnelle

La sémantique opérationnelle du langage OCaLustre est semblable à celle du langage Lustre définie dans [CPHP87]. Néanmoins, cette sémantique de référence considère l'*inlining* des nœuds, tandis que nous verrons dans le chapitre 4 que le compilateur d'OCaLustre suit un modèle de compilation séparée. Bien que ce modèle de compilation n'accepte pas certains programmes synchrones valides en Lustre (nous aborderons cette limitation en 4.2.2), ceux qui sont acceptés respectent bel et bien la sémantique décrite dans cette section. Cette sémantique est présentée succintement, sous forme de règles d'inférence appliquées à une représentation simplifiée de la forme normale des programmes OCaLustre. Dans cette représentation simplifiée, un programme est une liste de toutes les équations calculées pendant un instant d'exécution. Ceci revient à dire que tous les appels à des nœuds sont remplacés par le corps de ces derniers, et que l'intégration (ou *inlining*) des appels de nœuds est ainsi réalisée, comme dans le modèle de compilation de Lustre. Par ailleurs, les équations du programme sont représentées annotées par leurs horloges respectives, placées en dessous du symbole d'égalité. De surcroît, toute constante est elle aussi annotée par son horloge, placée en exposant.

Soit $\sigma$ la *mémoire* associée à un programme OCaLustre, c'est-à-dire une fonction qui associe à chaque nom de variable une valeur à chaque instant d'exécution. Une mémoire $\sigma$ est dite *compatible* avec le système d'équations d'un programme si pour toute équation $x = e$ du système, la valeur du membre droit de l'équation est identique à $\sigma(x)$ si l'horloge de $x$ est vraie (sinon, $\sigma(x) = \bot$ peu importe la valeur de $e$). Autrement dit, une mémoire est compatible avec le système d'équations du programme si elle est solution de ce dernier.

Nous définissons alors les jugements suivants :

— $\sigma \vdash exp : v$ signifie que, dans un instant synchrone, l'expression $exp$ est réduite à la valeur $v$ lors de son évaluation dans le contexte de la mémoire $\sigma$.

— $eqn \xrightarrow{\sigma} eqn'$ signifie que l'équation $eqn$ est compatible avec la mémoire $\sigma$, et qu'elle sera remplacée par l'expression $eqn'$ pour l'exécution de l'instant suivant.

— $\vec{eqn} \xrightarrow{\sigma} \vec{eqn}'$ signifie que la liste d'équations $\vec{eqn}$ est compatible avec la mémoire $\sigma$, et qu'elle sera remplacée par la liste d'équations $\vec{eqn}'$ pour l'exécution de l'instant suivant.

— Enfin, $h \vdash \vec{eqn} : h'$ signifie qu'à partir de l'historique des entrées $h$ du programme, le système des équations $\vec{eqn}$ du programme produit l'historique des sorties $h'$.

L'intégralité des règles de la sémantique opérationnelle du langage est décrite dans la figure 3.11.

Nous décrivons ici les principales règles de la sémantique opérationnelle :

— La valeur d'une variable est extraite de la mémoire $\sigma$ :

$$\frac{\sigma(x) = v}{\sigma \vdash x : v}$$

— Si l'expression $e_1$ est réduite à la valeur $v_1$, et si l'expression $e_2$ est réduite à la valeur $v_2$, alors l'application d'un opérateur infixe binaire à ces deux expressions $e_1 \diamond e_2$ est réduite à la valeur $v_1 \diamond v_2$ :

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1 \diamond e_2 : v_1 \diamond v_2}$$

— Le résultat de la fusion (**merge**) de deux flots correspond à la valeur de son second (resp. troisième) paramètre dans le cas où son premier paramètre est vrai (resp. faux).

$$\frac{\sigma \vdash x : true \quad \sigma \vdash ce_1 : v \quad \sigma \vdash ce_2 : \bot}{\sigma \vdash \mathbf{merge}\, x\, ce_1\, ce_2 : v} \qquad \frac{\sigma \vdash x : false \quad \sigma \vdash ce_1 : \bot \quad \sigma \vdash ce_2 : w}{\sigma \vdash \mathbf{merge}\, x\, ce_1\, ce_2 : w}$$

— Un flot défini par une équation n'a pas de valeur ($\bot$) si son horloge est fausse (ou si elle est elle-même absente) :

$$\frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} ce \xrightarrow{\sigma} y \underset{ck}{=} ce}$$

— L'évaluation d'une équation contenant l'opérateur **fby** implique un changement de la forme de l'équation pour l'instant suivant. La valeur à gauche de l'opérateur correspond alors à la valeur calculée à l'instant précédent :

$$\frac{\sigma(ck) = true \quad \sigma(y) = k \quad \sigma \vdash e : k'}{y \underset{ck}{=} k\ \mathbf{fby}\ e \xrightarrow{\sigma} y \underset{ck}{=} k'\ \mathbf{fby}\ e}$$

$\boxed{\sigma \vdash e : v}$

$$\frac{\sigma(ck) = true}{\sigma \vdash ()^{ck} : ()} \qquad \frac{\sigma(ck) \neq true}{\sigma \vdash ()^{ck} : \bot} \qquad \frac{\sigma(ck) = true}{\sigma \vdash k^{ck} : k} \qquad \frac{\sigma(ck) \neq true}{\sigma \vdash k^{ck} : \bot}$$

$$\frac{\sigma(ck) = true}{\sigma \vdash X_i^{ck} : X_i} \qquad \frac{\sigma(ck) \neq true}{\sigma \vdash X_i^{ck} : \bot} \qquad \frac{\sigma(x) = v}{\sigma \vdash x : v} \qquad \frac{\sigma \vdash e : v}{\sigma \vdash \square\, e : \square\, v}$$

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1 \diamond e_2 : v_1 \diamond v_2} \qquad \frac{\sigma(x) = true \quad \sigma \vdash e : v}{\sigma \vdash e\ \mathbf{when}\ x : v} \qquad \frac{\sigma(x) \neq true}{\sigma \vdash e\ \mathbf{when}\ x : \bot}$$

$$\frac{\sigma(x) = false \quad \sigma \vdash e : v}{\sigma \vdash e\ \mathbf{whennot}\ x : v} \qquad \frac{\sigma(x) \neq false}{\sigma \vdash e\ \mathbf{whennot}\ x : \bot}$$

$\boxed{\sigma \vdash ce : v}$

$$\frac{\sigma \vdash e_1 : true \quad \sigma \vdash ce_2 : v \quad \sigma \vdash ce_3 : w}{\sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ ce_2\ \mathbf{else}\ ce_3 : v} \qquad \frac{\sigma \vdash e_1 : false \quad \sigma \vdash ce_2 : v \quad \sigma \vdash ce_3 : w}{\sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ ce_2\ \mathbf{else}\ ce_3 : w}$$

$$\frac{\sigma \vdash x : true \quad \sigma \vdash ce_1 : v \quad \sigma \vdash ce_2 : \bot}{\sigma \vdash \mathbf{merge}\ x\ ce_1\ ce_2 : v} \qquad \frac{\sigma \vdash x : false \quad \sigma \vdash ce_1 : \bot \quad \sigma \vdash ce_2 : w}{\sigma \vdash \mathbf{merge}\ x\ ce_1\ ce_2 : w}$$

$\boxed{eqn \xrightarrow{\sigma} eqn'}$

$$\frac{\sigma(ck) = true \quad \sigma(y) = k \quad \sigma \vdash e : k'}{y \underset{ck}{=} k\ \mathbf{fby}\ e \xrightarrow{\sigma} y \underset{ck}{=} k'\ \mathbf{fby}\ e} \qquad \frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} k\ \mathbf{fby}\ e \xrightarrow{\sigma} y \underset{ck}{=} k\ \mathbf{fby}\ e}$$

$$\frac{\sigma(ck) = true \quad \sigma(y) = w \quad \sigma \vdash e_0 : v_0 \quad \sigma \vdash e_1 : v_1 \quad \ldots \quad \sigma \vdash e_{n-1} : v_{n-1} \quad (\mathtt{f}\, v_0\, v_1 \ldots v_{n-1}) \Downarrow w}{y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1} \xrightarrow{\sigma} y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1}}$$

$$\frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1} \xrightarrow{\sigma} y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1}}$$

$$\frac{\sigma(ck) = true \quad \sigma(y) = v \quad \sigma \vdash ce : v}{y \underset{ck}{=} ce \xrightarrow{\sigma} y \underset{ck}{=} ce} \qquad \frac{\sigma(ck) \neq true \quad \sigma(y) = \bot}{y \underset{ck}{=} ce \xrightarrow{\sigma} y \underset{ck}{=} ce}$$

$\boxed{\vec{eqn} \xrightarrow{\sigma} \vec{eqn}'}$

$$\frac{eqn \xrightarrow{\sigma} eqn' \quad \vec{eqn} \xrightarrow{\sigma} \vec{eqn}'}{eqn;\ \vec{eqn} \xrightarrow{\sigma} eqn';\ \vec{eqn}'}$$

$\boxed{h \vdash \vec{eqn} : h'}$

$$\frac{\vec{eqn} \xrightarrow{\sigma} \vec{eqn}' \quad h \vdash \vec{eqn}' : h'}{\sigma[input].h \vdash \vec{eqn} : \sigma[output].h'}$$

Figure 3.11 – Sémantique opérationnelle d'OCaLustre

— Évaluer un appel de fonction OCaml (désigné par un **call**) revient à évaluer tous les paramètres de l'appel ($e_0$ à $e_n$) dans OCaLustre puis à appliquer la fonction aux valeurs calculées. Le résultat $w$ de cette application, calculé dans la sémantique d'OCaml [11], correspond à la valeur du flot résultant $y$.

$$\frac{\sigma(ck) = true \quad \sigma(y) = w \quad \sigma \vdash e_0 : v_0 \quad \sigma \vdash e_1 : v_1 \quad \ldots \quad \sigma \vdash e_{n-1} : v_{n-1} \quad (\mathtt{f}\, v_0\, v_1 \ldots v_{n-1}) \Downarrow w}{y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1} \xrightarrow{\sigma} y \underset{ck}{=} \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1}}$$

— Enfin, évaluer un programme revient à évaluer les équations qu'il contient en calculant une solution du système d'équations pour des valeurs en entrée données. Il produit alors un ensemble de valeurs qui correspondent à celles des variables en sortie du programme, et induit récursivement l'évaluation de nouvelles sorties à partir des entrées de l'instant suivant.

$$\frac{\vec{eqn} \xrightarrow{\sigma} \vec{eqn}' \quad h \vdash \vec{eqn}' : h'}{\sigma[input].h \vdash \vec{eqn} : \sigma[output].h'}$$

Pour illustrer ces diverses règles sémantiques, nous donnons dans la figure 3.12 une dérivation de l'évaluation de l'équation $y \underset{\bullet}{=} 2\ \mathbf{fby}\ (y + 1)$ : à l'instant suivant d'exécution, cette équation devient $y \underset{\bullet}{=} 3\ \mathbf{fby}\ (y + 1)$.

$$\frac{\sigma(\bullet) = true \qquad \sigma(y) = 2 \qquad \dfrac{\dfrac{\sigma(y) = 2}{\sigma \vdash y : 2} \quad \dfrac{}{\sigma \vdash 1 : 1}}{\sigma \vdash (y + 1) : 3}}{y \underset{\bullet}{=} 2\ \mathbf{fby}\ (y + 1) \xrightarrow{\sigma} y \underset{\bullet}{=} 3\ \mathbf{fby}\ (y + 1)}$$

FIGURE 3.12 – Dérivation de l'évaluation d'un flot OCaLustre

Une mémoire est *consistante* du point de vue du cadencement si et seulement si elle associe $\perp$ à chaque variable dont la valeur n'est pas calculée (i.e. si son horloge est fausse ou si cette horloge n'est elle-même pas calculée). Une propriété attendue de cette sémantique est que si le programme est bien cadencé, alors une mémoire compatible avec ce programme est nécessairement consistante (par conséquent $\perp$ n'est alors jamais consulté lors de l'évaluation des expressions d'un programme bien cadencé). La preuve de cette propriété qui fait le lien entre typage et sémantique est néanmoins reportée à des travaux futurs, qui pourraient s'inspirer de techniques similaires à celles utilisées dans des travaux connexes de certification de la compilation de Lustre [Aug13, BBD+17]. D'autre part, avec la sémantique décrite dans cette section il peut exister plusieurs solutions au système d'équations d'un instant synchrone. Des contraintes statiques supplémentaires sont nécessaires pour garantir l'existence d'une unique solution, et de permettre le calcul de cette solution de manière séquentielle. Nous détaillerons dans le chapitre suivant la nature de ces propriétés de causalité et d'ordonnançabilité, à la base de la méthode de compilation en boucle simple.

11. La notation $e \Downarrow w$ indique que dans la sémantique d'OCaml l'expression $e$ est évaluée en la valeur $v$

## Conclusion du chapitre

Ce chapitre a constitué un tour d'horizon complet des constructions et des principes offerts par notre extension synchrone du langage OCaml. Nous avons abordé de nombreux aspects formels, en particulier des propriétés de typage inhérentes à l'utilisation de ce paradigme de programmation. OCaLustre est destiné à être utilisé conjointement avec la machine virtuelle OMicroB présentée dans le chapitre précédent, et il est donc essentiel qu'elle soit pleinement compatible avec l'interprète de bytecode d'OMicroB. C'est aussi le cas de Lucid Synchrone, mais sa grande expressivité (ordre supérieur, multiples horloges de base pour un même nœud, automates . . . ) produit des programmes utilisant plus de ressources, qui ne sont pas compatibles avec les limitation matérielles que nous considérons. Nous aborderons à ce sujet quelques éléments de comparaison dans le chapitre 7 qui illustrent que notre solution permet de limiter l'empreinte mémoire des programmes synchrones car le modèle de compilation d'OCaLustre est simplifié par une expressivité plus limitée.

Dans le chapitre suivant, nous présentons alors les étapes qui permettent la compilation d'un programme OCaLustre vers un programme OCaml séquentiel standard, totalement compatible avec OMicroB et les appareils visés, mais aussi avec tout compilateur du langage OCaml.

# 4 Compilation des programmes OCaLustre

OCaLustre est une extension du langage OCaml, et nous utilisons à ce titre un ensemble d'outils logiciels permettant la définition de telles extensions. Nous faisons en particulier usage de *PPX* [⚓3], un outil du compilateur standard OCaml permettant la définition d'extensions syntaxiques, via l'utilisation d'annotations particulières dans le code source du programme. Dans notre cas d'utilisation, nous annotons ce qui est initialement considéré par le compilateur OCaml comme une définition de fonction avec le mot-clé « `node` » afin de construire un nœud OCaLustre (d'où la syntaxe des annotations de la forme `let%node`).

L'intérêt de l'utilisation de PPX est de confier au compilateur standard du langage OCaml [1] le rôle de réaliser les étapes nécessaires à l'analyse syntaxique du programme source, et d'en construire une représentation interne au compilateur. Le compilateur OCaLustre peut ensuite se brancher directement sur cette représentation interne (un arbre de syntaxe abstraite – « *AST* » – OCaml), y appliquer plusieurs transformations, avant de le remettre au compilateur standard qui se charge alors de produire un code exécutable. Le compilateur d'OCaLustre peut ainsi être considéré comme un *préprocesseur*, chargé de modifier la structure de l'AST du programme, avant la poursuite de sa prise en charge par le compilateur standard.

Dans ce chapitre, nous décrivons les étapes nécessaires à la compilation d'un programme OCaLustre, depuis sa mise en forme normale jusqu'à la génération d'une représentation standard d'un programme OCaml. Nous suivons un processus bien documenté dans la littérature [Pla88, BCHP08], ayant par le passé servi à réaliser des compilateurs pour langages synchrones comparables à OCaLustre.

La figure 4.1 représente schématiquement la chaîne de compilation d'un programme OCaLustre. Celle-ci est constituée de quatre étapes principales :

— Une première étape de *normalisation* transforme le programme en restructurant ces composantes afin de simplifier les analyses statiques appliquées lors de sa compilation.
— Une seconde étape d' *ordonnancement* consiste à trier les diverses équations qui constituent le corps d'un nœud OCaLustre afin que leur ordre de lecture corresponde à leur ordre de déclaration lors de la génération de code OCaml.
— La troisième étape de compilation consiste en l'*inférence des types d'horloges* des équations OCaLustre. Cette étape annote automatiquement chaque équation d'un programme avec son horloge, en suivant les règles de typage des horloges définies en section 3.2.4.
— L'ultime étape de compilation d'un programme OCaLustre consiste en la *traduction* du programme OCaLustre annoté vers un programme OCaml standard, sous la forme d'un AST compréhensible par le compilateur du langage. Cet AST est ensuite pris en charge par ce dernier, et compilé (dans le cas d'une compilation vers un fichier bytecode) vers un fichier pleinement compatible avec une quelconque machine virtuelle OCaml, dont OMicroB.

1. Qu'il s'agisse de *ocamlc*, le compilateur générant du bytecode OCaml ; ou bien *ocamlopt*, le compilateur générant du code natif.

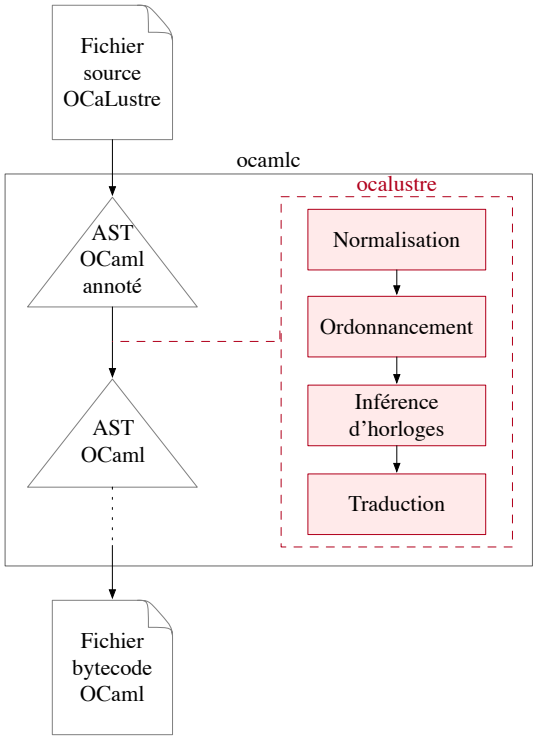



FIGURE 4.1 – Chaîne de compilation d'un programme OCaLustre

Il est à noter que notre décision de suivre un modèle de *compilation séparée* des nœuds OCaLustre (sur un modèle comparable à SCADE, contrairement à celui de Lustre V4 [HR01] qui réalise l'*inlining* des appels), repose sur la volonté d'éviter la duplication de code lors de la compilation de plusieurs appels au même nœud. En raison des faibles ressources mémoire d'un microcontrôleur, il est en effet important pour notre cas d'utilisation de limiter l'empreinte mémoire des programmes. Nous verrons lors de la description du procédé d'ordonnancement que cette décision a des conséquences sur la validation statique de certains programmes, mais ce désavantage nous semble acceptable compte tenu de l'importance dans nos travaux de limiter la consommation en ressources des programmes.

## 4.1 Mise en forme normale

La première étape de compilation d'un programme OCaLustre consiste à transformer les nœuds issus de l'analyse syntaxique et lexicale du programme source vers leur représentation dans la *forme normale* définie dans la section 3.2.1.

Le rôle principal de la mise en forme normale, ou *normalisation* est d'extraire, dans les équations d'un nœud, les expressions que nous appelons « *expressions à effets de bord* » afin de simuler, dans le langage cible (OCaml), l'utilisation de l'opérateur conditionnel à évaluation stricte d'OCaLustre. Cette transformation permet ainsi de conserver la sémantique d'OCaLustre sans avoir à définir un opérateur « `if` » spécialement destiné à l'exécution de nœuds OCaLustre. Les expressions à effets de bord sont celles qui, lors de leur évaluation, ont une influence sur l'état interne d'un nœud. Dans OCaLustre, elles sont au nombre de trois :

1. L'opérateur de décalage `⋙` entraîne implicitement la manipulation d'un registre de l'état interne du nœud dans lequel il est utilisé : l'expression `0 ⋙ e` est donc une expression à effet de bord qui

induit le calcul de la valeur de l'expression `e` à l'instant $t$ ainsi que l'enregistrement de cette valeur dans l'état interne du nœud afin que celle-ci soit accessible au programme à l'instant $t + 1$.

Afin que cette évaluation ne soit pas « gelée » dans le cas où cette expression serait nichée dans la conséquence ou l'alternant d'une conditionnelle, toute sous-expression apparaissant dans la définition d'un flot `x` qui contient un `⋙` est extraite de l'expression dans laquelle elle était nichée, et déplacée dans la définition d'un nouveau flot, nommé `x_aux`. De cette façon, la valeur de `x_aux` est réellement calculée à chaque instant d'exécution du programme.

Par exemple, le nœud non normalisé suivant :

```
let%node ex_norm (b) ~return:(x) =
  x = if b then (0 ⋙ (x + 1)) else (0 ⋙ x)
```

est traduit en forme normale vers cette représentation sémantiquement équivalente :

```
node ex_norm b return x =
  x_aux1 = 0 fby (x + 1);
  x_aux2 = 0 fby x;
  x = if b then x_aux1 else x_aux2
```

Ainsi, le calcul de `0 ⋙ (x + 1)` et de `0 ⋙ x` est bien réalisé à chaque instant d'exécution (quelle que soit la valeur de `b`), sans avoir à distinguer le **`if`** à évaluation paresseuse du langage OCaml, et le **`if`** à évaluation stricte d'OCaLustre.

2. De la même façon, puisque le corps d'un nœud peut contenir des équations utilisant l'opérateur de décalage, tout appel à ce dernier peut produire ce même type d'effets de bord. Le mécanisme d'extraction est donc étendu également aux applications. Par exemple, le nœud suivant :

   ```
   let%node call_cpt_cond (b) ~return:(x) =
     x = if b then cpt () else 0
   ```

   devient, après normalisation :

   ```
   node ex_norm b return x =
     x_aux = cpt ();
     x = if b then x_aux else 0
   ```

3. Enfin, la même opération que pour les appels de nœuds est appliquée pour chaque utilisation de l'opérateur **`call`**, car la fonction appelée peut elle-même réaliser des effets de bord.

Plus généralement, la mise en forme normale consiste à transformer un programme OCaLustre depuis une représentation correspondant à la syntaxe concrète du langage vers un AST, défini à partir de la grammaire normalisée décrite dans la section 3.2.1, manipulable par le compilateur du langage. Ce processus uniformise la représentation interne des nœuds, ainsi que leur analyse, en structurant les divers types d'expressions (expressions de contrôle, expressions normales, ...) et en simplifiant

certaines constructions. Par exemple, les n-uplets présents dans la syntaxe concrète du langage sont éclatés en plusieurs équations distinctes après normalisation. Le programme suivant :

```
let%node ex_tuples (a,b) ~return:(c,d) =
  (c,d) = (a,b)
```

est représenté, en forme normale, de la façon suivante :

```
node ex_tuples (a,b) return (c,d) =
  c = a;
  d = b
```

La mise en forme normale d'un programme OCaLustre correspond à l'application d'une fonction *normalize* qui transforme l'arbre de syntaxe abstraite issu de l'analyse syntaxique et lexicale d'un programme source OCaLustre en un AST correspondant à la représentation interne, normalisée, du programme ($ast_{norm}$). La fonction de normalisation peut être décomposée en deux fonctions : une première fonction *split* qui « éclate » les n-uplets (donnée dans la figure 4.2), suivie de la fonction *norm* qui réalise effectivement la normalisation en distribuant les expressions vers leur équivalent dans la grammaire de la forme normale. La fonction *norm* de normalisation d'un nœud est illustrée dans la figure 4.3, elle fait appel à des fonctions auxiliaires mutuellement récursives $norm_x$ dont l'indice $x$ correspond au nom de la catégorie syntaxique vers laquelle chaque fonction convertit une expression ou une équation donnée. Le rôle de ces fonctions est de parcourir l'AST du programme issu de son analyse syntaxique (dont nous représentons le code en caractères d'imprimerie) afin de produire un AST en forme normalisée, et ce en collectionnant de nouvelles équations issues de la réorganisation de l'AST selon la forme normale. Lorsqu'une nouvelle équation doit être créée, elles font appel à une fonction $fresh()$ qui génère un nom de variable frais à chaque appel.

$$
\begin{aligned}
split(\vec{y} = f\ e) &= [\vec{y} = f\ e] \\
split(y = e) &= [y = e] \\
split(y, \vec{y} = e, \vec{e}) &= (y = e) :: split(\vec{y} = \vec{e}) \\
split(eqn_1 ; eq\vec{n}_2) &= let\ eq\vec{n}'_1 = split(eqn_1)\ in \\
&\qquad let\ eq\vec{n}_2{}' = split(eq\vec{n}_2)\ in \\
&\qquad eq\vec{n}'_1 +\!\!+ eq\vec{n}'_2 \\
split(\texttt{let\%node}\ f\ \vec{x}\ \texttt{\sim return:}\ \vec{y} = eq\vec{n}) &= let\ eq\vec{n}' = split(eq\vec{n})\ in \\
&\qquad \texttt{let\%node}\ f\ \vec{x}\ \texttt{\sim return:}\ \vec{y} = eq\vec{n}'
\end{aligned}
$$

FIGURE 4.2 – Fonction d'éclatement des n-uplets

## 4.2 Ordonnancement

Dans un programme OCaLustre, les diverses équations constituant le corps d'un nœud ne sont contraintes par aucune condition régissant leur ordre d'apparition. En effet, puisque le corps d'un nœud

$$
\begin{aligned}
norm_e(e_1 \diamond e_2) &= let\ (l_1, e_1') = norm_e(e_1)\ in \\
&\quad let\ (l_2, e_2') = norm_e(e_2)\ in\ (l_1 +\!\!+ l_2, e_1' \diamond e_2') \\
norm_e(\square\ e_0) &= let\ (l, e_0') = norm_e(e_0)\ in\ (l, \square\ e_0') \\
norm_e(e_0\ \texttt{[@when x]}) &= let\ (l, e_0') = norm_e(e_0)\ in\ (l, e_0'\ \textbf{when}\ x) \\
norm_e(e_0\ \texttt{[@whennot x]}) &= let\ (l, e_0') = norm_e(e)\ in\ (l, e_0'\ \textbf{whennot}\ x) \\
norm_e(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= let\ x = fresh()\ in \\
&\quad let\ (l, eqn) = norm_{eqn}(x = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)\ in\ (eqn :: l, x) \\
norm_e(\texttt{merge}\ x\ e_1\ e_2) &= let\ x = fresh()\ in \\
&\quad let\ (l, eqn) = norm_{eqn}(x = \texttt{merge}\ x\ e_1\ e_2)\ in\ (eqn :: l, x) \\
norm_e(k\ \texttt{-»}\ e) &= let\ x = fresh()\ in \\
&\quad let\ (l, eqn) = norm_{eqn}(x = k\ \texttt{-»}\ e)\ in\ (eqn :: l, x) \\
norm_e(\texttt{call f}\ e_0\ e_1\ \ldots\ e_{n-1}) &= let\ x = fresh()\ in \\
&\quad let\ (l, eqn) = norm_{eqn}(x = \texttt{call f}\ e_0\ e_1\ \ldots\ e_{n-1})\ in\ (eqn :: l, x) \\
norm_e(f\ \texttt{(}e_0\texttt{)}) &= let\ x = fresh()\ in \\
&\quad let\ (l, eqn) = norm_{eqn}(x = f\texttt{(}e_0\texttt{)})\ in\ (eqn :: l, x) \\
norm_e(e_0) &= ([], e_0) \text{ pour toutes les autres formes de } e_0
\end{aligned}
$$

$$
\begin{aligned}
norm_{ce}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= let\ (l_1, e_1') = norm_e(e_1)\ in \\
&\quad let\ (l_2, ce_2) = norm_{ce}(e_2)\ in \\
&\quad let\ (l_3, ce_3) = norm_{ce}(e_3)\ in \\
&\quad (l_1 +\!\!+ l_2 +\!\!+ l_3, \textbf{if}\ e_1'\ \textbf{then}\ ce_2\ \textbf{else}\ ce_3) \\
norm_{ce}(\texttt{merge}\ x\ e_1\ e_2) &= let\ (l_1, e_1') = norm_e(e_1)\ in \\
&\quad let\ (l_2, e_2') = norm_e(e_2)\ in \\
&\quad (l_1 +\!\!+ l_2, \textbf{merge}\ x\ e_1'\ e_2') \\
norm_{ce}(e_0) &= norm_e(e_0) \text{ pour toutes les autres formes de } e_0
\end{aligned}
$$

$$
\begin{aligned}
norm_{\vec{e}}(e_0) &= norm_e(e_0) \\
norm_{\vec{e}}(e_1\texttt{,}e_2) &= let\ (l_1, e_1') = norm_e(e_1)\ in \\
&\quad let\ (l_2, \vec{e_2}) = norm_{\vec{e}}(e_2)\ in\ (l_1 +\!\!+ l_2, e_1' :: \vec{e_2})
\end{aligned}
$$

$$
\begin{aligned}
norm_{eqn}(y = k\ \texttt{-»}\ e) &= let\ (l, e') = norm_e(e)\ in\ (l, y = k\ \textbf{fby}\ e') \\
norm_{eqn}(\vec{y} = f\texttt{(}e\texttt{)}) &= let\ (l, \vec{e}) = norm_{\vec{e}}(e)\ in\ (l, \vec{y} = f\ (\vec{e})) \\
norm_{eqn}(y = \texttt{call f}\ e_0\ e_1\ \ldots\ e_{n-1}) &= let\ (l_0, e_0') = norm_e(e_0)\ in \\
&\quad let\ (l_1, e_1') = norm_e(e_1)\ in \ldots \\
&\quad let\ (l_{n-1}, e_{n-1}') = norm_e(e_{n-1})\ in\ (l_0 +\!\!+ l_1 +\!\!+ \ldots +\!\!+ l_{n-1}, y = \textbf{call f}\ e_0'\ e_1'\ \ldots\ e_{n-1}') \\
norm_{eqn}(y = e) &= let\ (l, ce) = norm_{ce}(e)\ in\ (l, y = ce)
\end{aligned}
$$

$$
\begin{aligned}
norm_{e\vec{q}n}(eqn) &= let\ (l, eqn) = norm_{eqn}(eqn)\ in\ (l, [eqn]) \\
norm_{e\vec{q}n}(eqn\texttt{;}e\vec{q}n) &= let\ (l, eqn') = norm_{eqn}(eqn)\ in \\
&\quad let\ (l', e\vec{q}n') = norm_{e\vec{q}n}(e\vec{q}n)\ in\ (l +\!\!+ l', eqn' :: e\vec{q}n')
\end{aligned}
$$

$$
norm(\texttt{let\%node}\ f\ \vec{x}\ \texttt{~return:}\ \vec{y} = e\vec{q}n) = let\ (e\vec{q}n', e\vec{q}n'') = norm_{e\vec{q}n}(e\vec{q}n)\ in\ \textbf{node}\ f\ \vec{x}\ \textbf{return}\ \vec{y} = (e\vec{q}n' +\!\!+ e\vec{q}n'')
$$

FIGURE 4.3 – Fonctions de normalisation

est un système d'équations dont les valeurs sont calculées à chaque instant, rien n'interdit (comme dans un système d'équations mathématiques classique) à une de ces équations de faire référence à une variable étant définie plusieurs lignes après sa propre définition. De ce fait, le nœud suivant constitue un exemple parfaitement valide, bien que l'équation définissant le flot `z` fasse référence au flot `k` n'étant pas encore défini dans le sens de lecture naturel (de haut en bas), et que l'équation définissant le flot `k` fasse référence à `w` avant que lui aussi ait été défini :

```
node ordo (x,y) return (z,k) =
  z = 3 * k;
  k = if x then w else 4;
  w = y + 42
```

Le compilateur d'OCaLustre réalise alors, dans le but final de transformer les équations OCaLustre en déclarations de variables OCaml, un *ordonnancement* du corps d'un nœud. Ce procédé consiste à réorganiser le système d'équation afin de faire apparaître chaque définition de flot avant son utilisation dans d'autres équations. Ainsi, l'ordonnancement de l'exemple précédent consiste à traiter une version de `ordo` où chaque définition de flot apparaîtrait avant son utilisation :

```
node ordo_correct (x,y) return (z,k) =
  w = y + 42;
  k = if x then w else 4;
  z = 3 * k
```

Bien sûr, l'ordonnancement du corps d'un nœud ne modifie en rien la sémantique du langage, puisque chaque équation est considérée comme étant calculée *parallèlement* aux autres équations : le sens de lecture des équations ne traduit pas leur réelle sémantique d'exécution (qui ne définit aucun ordre séquentiel d'évaluation).

La réalisation de l'ordonnancement des équations repose sur la création d'un graphe acyclique[2] orienté représentant les dépendances entre les divers flots *au sein du même instant*. Il est considéré qu'un flot $y$ est une dépendance d'un flot $x$ dès lors qu'il est fait référence (au même instant) à $y$ dans la définition de $x$. Dans le graphe de dépendances, une arête d'un sommet $x$ vers un sommet $y$ représente le fait que le flot $x$ dépend de $y$. Par exemple, la figure 4.4 représente le graphe de dépendances du nœud `ordo`.

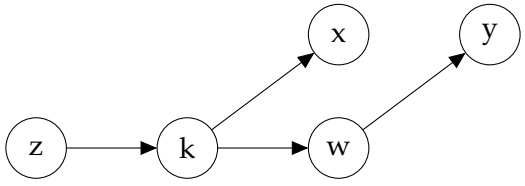



Figure 4.4 – Graphe de dépendances du nœud `ordo`

Une fois ce graphe réalisé, l'ordonnancement d'un nœud consiste simplement à suivre le tri topologique inverse de celui-ci (sans prendre en compte les flots en entrée du nœud) pour réaliser la séquence

2. Le cas erroné où ce graphe serait cyclique sera décrit dans la sous-section suivante.

d'équations qui constitue le corps du nœud considéré. Dans notre exemple, l'ordonnancement de `ordo` consiste ainsi à définir d'abord `w`, puis `k`, et enfin `z`.

Une équation est bien ordonnancée à condition que toutes les variables qu'elle contient, et dont elle dépend *instantanément*, aient été préalablement définies. Formellement, nous définissons la fonction *idents* qui retourne la liste de toutes les variables présentes dans une expression (ou une liste d'expressions) :

$\boxed{idents(e)}$

$$
\begin{aligned}
&idents(k) \equiv \varnothing \\
&idents(x) \equiv [x] \\
&idents(X_i) \equiv \varnothing \\
&idents(e \diamond e') \equiv idents(e) +\!\!+ idents(e') \\
&idents(\square\, e) \equiv idents(e) \\
&idents(e \,\mathbf{when}\, x) \equiv x :: idents(e) \\
&idents(e \,\mathbf{whennot}\, x) \equiv x :: idents(e)
\end{aligned}
$$

$\boxed{idents(\vec{e})}$

$$
\begin{aligned}
&idents([e]) \equiv idents(e) \\
&idents(e, \vec{e}) \equiv idents(e) +\!\!+ idents(\vec{e})
\end{aligned}
$$

$\boxed{idents(ce)}$

$$
\begin{aligned}
&idents(e) \equiv idents(e) \\
&idents(\mathbf{merge}\, x\, ce\, ce') \equiv x :: idents(ce) +\!\!+ idents(ce') \\
&idents(\mathbf{if}\, e \,\mathbf{then}\, ce' \,\mathbf{else}\, ce'') \equiv idents(e) +\!\!+ idents(ce') +\!\!+ idents(ce'')
\end{aligned}
$$

Pour toute liste de variables $\mathcal{V}$, les règles suivantes définissent la notion d'équations *bien ordonnancées* dans le contexte où les variables contenues dans $\mathcal{V}$ ont toutes été définies au préalable. On notera alors $\mathcal{V} \vdash_{ws} eqn$ pour signifier que l'équation *eqn* est bien ordonnancée (« *ws* » pour « *well scheduled* »).

$\boxed{\mathcal{V} \vdash_{ws} eqn}$

$$
\frac{y \notin \mathcal{V} \quad idents(ce) \in \mathcal{V}}{\mathcal{V} \vdash_{ws} y = ce} \qquad \frac{y \notin \mathcal{V}}{\mathcal{V} \vdash_{ws} y = k \,\mathbf{fby}\, e}
$$

$$
\frac{\vec{y} \notin \mathcal{V} \quad idents(\vec{e}) \in \mathcal{V}}{\mathcal{V} \vdash_{ws} \vec{y} = f(\vec{e})} \qquad \frac{y \notin \mathcal{V} \quad idents(e_0) \in \mathcal{V} \quad idents(e_1) \in \mathcal{V} \quad \ldots \quad idents(e_{n-1}) \in \mathcal{V}}{\mathcal{V} \vdash_{ws} y = \mathbf{call}\, \mathtt{f}\, e_0\, e_1 \ldots e_{n-1}}
$$

$\boxed{\mathcal{V} \vdash_{ws} \vec{eqn}}$

$$\frac{\mathcal{V} \vdash_{ws} eqn}{\mathcal{V} \vdash_{ws} [eqn]} \qquad \frac{\mathcal{V} \vdash_{ws} y = ce \quad y :: \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} y = ce;\ \vec{eqn}}$$

$$\frac{\mathcal{V} \vdash_{ws} y = k\ \mathbf{fby}\ e \quad y :: \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} y = k\ \mathbf{fby}\ e;\ \vec{eqn}} \qquad \frac{\mathcal{V} \vdash_{ws} \vec{y} = f(\vec{e}) \quad \vec{y} +\!\!+ \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} \vec{y} = f(\vec{e});\ \vec{eqn}}$$

$$\frac{\mathcal{V} \vdash_{ws} y = \ \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1} \quad y :: \mathcal{V} \vdash_{ws} \vec{eqn}}{\mathcal{V} \vdash_{ws} y = \ \mathbf{call}\ \mathtt{f}\ e_0\ e_1 \ldots e_{n-1};\ \vec{eqn}}$$

Un nœud est alors bien ordonnancé à condition que toutes les équations qui constituent son corps soient elles-mêmes bien ordonnancées, dans le contexte où sont définies les variables $\vec{x}$ qui représentent ses paramètres d'entrée :

$\boxed{\vdash_{ws} node}$

$$\frac{\vec{x} \vdash_{ws} \vec{eqn}}{\vdash_{ws} \mathbf{node}\, f(\vec{x})\, \mathbf{return}\, (\vec{y}) = \vec{eqn}}$$

Le procédé d'ordonnancement des nœuds OCaLustre est réalisé après mise en forme normale du programme. L'ordonnancement se fait donc sur la représentation intermédiaire du programme et peut être vu comme une fonction *schedule* de type $ast_{norm} \rightarrow ast_{norm}$.

Par exemple, soit le programme $\mathcal{P}$ (normalisé) suivant :

```
node f a return b =
  b = c + 1;
  c = 0 fby a

node g x return (y,z) =
  z = y + x;
  y = x
```

Alors, le résultat de l'application $schedule(\mathcal{P})$ est :

```
node f a return b =
  c = 0 fby a;
  b = c + 1

node g x return (y,z) =
  y = x;
  z = y + x
```

### 4.2.1 Détection des boucles de causalité

L'étape d'ordonnancement des nœuds OCaLustre permet de détecter statiquement des erreurs ou incohérences dans la définition des flots d'un nœud. Par exemple, le nœud suivant possède une sémantique indéterminée :

```
node causal_loop () return (c,d) =
  c = 5 * d;
  d = 2 - c
```

En effet, dans cet exemple le flot `c` dépend du flot du flot `d`, et le flot `d` dépend, lui, de `c`. Les flots `c` et `d` dépendent ainsi, *dans le même instant*, l'un de l'autre. Il est alors impossible de calculer, par exemple, la valeur de `c`, puisqu'elle dépend de celle de `d`, qui elle-même dépend de `c` : il y a alors dépendances cycliques entre ces deux flots.

Ce cas de figure, nommé *boucle de causalité*, peut apparaître de façon moins flagrante, dans une longue chaîne de dépendances entre flots, et même à travers divers appels à des nœuds auxiliaires. Pour cette raison, le compilateur OCaLustre détecte automatiquement de telles dépendances cycliques, et refuse les programmes qui en contiennent. Cette détection est simplement réalisée en vérifiant qu'aucune boucle n'existe dans le graphe représentant les dépendances entre flots : la présence d'un cycle dans ce graphe entraîne l'arrêt de la compilation, et l'affichage d'un message d'erreur :

```
Error: Causality loop in node "causal_loop" with variables (d, c)
```

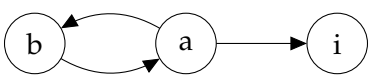
Il est par ailleurs à noter qu'une équation comme `y = 0 ⋙ (y+1)` ne dénote pas le fait que le flot `y` dépend de lui même, puisque l'utilisation de l'opérateur ⋙ implique que c'est la valeur du flot `y` à l'instant *précédent* qui est considérée dans l'expression définissant `y`, et les relations de dépendance ne sont calculées que pour le même instant. Ainsi, une telle équation est tout à fait compatible avec la sémantique du langage, et ne traduit pas l'existence d'une boucle de causalité.

### 4.2.2 Limitation dûe à la compilation séparée

Ce processus d'ordonnancement du corps des nœuds, associé à notre modèle de compilation séparée de chaque nœud OCaLustre peut entraîner le rejet de certains programmes pourtant corrects. En effet, prenons pour exemple l'extrait de programme suivant :

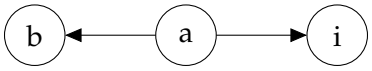
```
node shift_one x return y =
  y = 0 fby x

node call_shift_one i return b =
  a = b + i;
  b = shift_one a
```

Lors de l'ordonnancement du nœud `call_shift_one`, le graphe de dépendances suivant est généré :



Puisque ce graphe contient une boucle, le nœud est alors refusé à la compilation. Pourtant, si on considère une version du programme dans laquelle est pratiqué l'*inlining* de `shift_one`, c'est-à-dire le remplacement de l'appel à `shift_one` par l'équation contenue dans son corps, alors un graphe de dépendances différent est généré :

```
node call_shift_one i return b =
  a = b + i;
  b = 0 fby a
```




En effet, en réalité le flot *b* ne dépend pas *instantanément* de a, puisque a apparaît à droite de l'opérateur ⋙ une fois l'inlining de `shift_one` réalisé : le programme ne contenait pas de boucle de causalité. Les informations de décalage temporel entre les entrées et les sorties d'un nœud ne sont pas connues par le compilateur OCaLustre, qui considère les nœuds extérieurs comme des boîtes noires, et suppose ainsi que tout paramètre d'un appel à un nœud est une dépendance instantanée pour ses flots de sortie.

Dans notre cadre d'utilisation, cette limitation nous paraît néanmoins acceptable dans la mesure où le modèle de compilation séparée évite la recopie du corps d'un nœud pour chaque équation faisant appel à ce dernier, et permet ainsi de réduire l'empreinte mémoire finale des programmes OCaLustre.

## 4.3 Inférence d'horloges

Une fois sa normalisation et son ordonnancement effectués, l'étape suivante de compilation d'un programme OCaLustre consiste en l'inférence des horloges de chacune des expressions contenues dans les corps de ses divers nœuds synchrones.

Ce mécanisme d'inférence est dérivé du système d'inférence de types pour un système à la Hindley-Milner [Mil78], sur le modèle des travaux de Colaço et Pouzet [CP03] concernant le typage d'horloge, en limitant le polymorphisme à une unique variable de type : l'horloge de base (•).

Une fois qu'un type d'horloge a été attribué à chaque expression d'un programme, et donc que le programme respecte bien la sémantique de typage d'horloge (3.2.4), l'AST correspondant est annoté en associant chaque équation et chaque constante à son horloge. Ces annotations sont en particulier nécessaires à la vérification statique du typage des horloges, que nous aborderons dans la section 5.2.

L'arbre de syntaxe abstraite généré a une structure ainsi légèrement différente de celle d'un programme normalisé. Elle correspond à la grammaire de la forme normale dans laquelle certains éléments sont annotés par leur type d'horloge. Nous noterons $ast_{clocked}$ un arbre de syntaxe abstraite construit à partir de cette grammaire, et le procédé d'inférence d'horloge correspond alors à une fonction *clocks* de type $ast_{norm} \rightarrow ast_{clocked}$. Nous décrivons dans la suite uniquement les constructions qui diffèrent de la grammaire de la forme normale.

— Les constantes sont annotées avec leur type d'horloge, inféré par le compilateur :

$$\begin{array}{lll} e & ::= & \\ & | & k^{ck} \\ & | & (\ldots) \end{array}$$

— Et les équations sont annotées par leur horloge, en particulier l'application est annotée par son horloge d'application conditionnelle :

| $eqn$ | ::= | | équation |
|---|---|---|---|
| | \| | $y \underset{ck}{=} ce$ | expression |
| | \| | $y \underset{ck}{=} k$ **fby** $e$ | fby |
| | \| | $\vec{y} \underset{ck}{=} f(\vec{e})$ | application |
| | \| | $y \underset{ck}{=}$ **call** f $e_0\ e_1\ ..\ e_{n-1}$ | application de fonction OCaml |

Par exemple, soit le programme $\mathcal{Q}$ suivant :

```
node g (a,b,c) return z =
  x = a when c;
  y = b + 2;
  z = merge c x y
```

Alors, le résultat de l'application $clocks(\mathcal{Q})$ est :

```
node g (a,b,c) return z =
  x  =  a when c;
  (• on c)
  y  =  b + 2 (• onnot c);
  (• onnot c)
  z = merge c x y
  •
```

Le compilateur d'OCaLustre contient, par ailleurs, un *vérificateur* des types d'horloges inférés. Ce vérificateur, dont la preuve de correction sera donnée dans le chapitre 5, permet d'assurer que, pour un programme donné, les types inférés par le compilateur d'OCaLustre sont cohérents avec la description formelle du système de types donnée précédemment. Ce vérificateur est activé en utilisant l'option `-check_clocks` du compilateur.

## 4.4 Traduction vers OCaml

La dernière étape de compilation d'un programme OCaLustre consiste à traduire le programme, dans sa représentation intermédiaire annotée par les horloges, vers un AST OCaml compatible avec un quelconque compilateur OCaml.

Cette étape de *traduction* est réalisée en convertissant séparément chaque nœud d'un programme OCaLustre vers une fonction OCaml, en suivant l'approche décrite par Biernacki *et al.* [BCHP08] et utilisée pour la compilation de SCADE 6 [CPP17] ainsi que d'autres compilateurs de langages « à la Lustre » [GHKT14, MDLM18]. Ce modèle de compilation séparée, distinct du modèle de compilation de

Lustre où tous les composants d'un programme sont regroupés au sein d'une boucle principale, a été choisi pour réduire la taille des programmes dans lesquels plusieurs appels seraient faits aux mêmes nœuds, ainsi que pour permettre une meilleure modularisation des divers composants d'un programme.

Notre solution suit le modèle de compilation *en boucle simple* de Lustre, mais avec compilation séparée : chaque nœud OCaLustre est compilé vers une fonction distincte, mais le nœud principal du programme, qui fait appel aux fonctions générées, est exécuté dans une boucle infinie, qui lit les entrées du programme, exécute le nœud principal, et écrit les sorties calculées.

Nous décrivons dans cette section des *schémas de compilation* représentant la traduction de l'AST d'un programme OCaLustre vers un programme OCaml. La traduction de l'AST OCaLustre vers le code OCaml repose sur la fonction de traduction $[\![\cdot]\!]$ de type $ast_{clocked} \rightarrow ast_{ocaml}$ qui parcourt le programme OCaLustre et retourne un arbre de syntaxe abstraite correspondant à un programme OCaml, manipulable par le compilateur OCaml standard.

#### Traduction des expressions

Les expressions d'OCaLustre sont traduites de manière directe, en effaçant les opérateurs de sous-échantillonnage destinés au typage des horloges :

$$
\begin{aligned}
[\![()]\!] &= () \\
[\![k]\!] &= \texttt{k} \\
[\![x]\!] &= \texttt{x} \\
[\![X_i]\!] &= \texttt{X}_i \\
[\![e_1 \diamond e_2]\!] &= [\![e_1]\!] \diamond [\![e_2]\!] \\
[\![\square\ e]\!] &= \square\ [\![e]\!] \\
[\![e\ \textbf{when}\ x]\!] &= [\![e]\!] \\
[\![e\ \textbf{whennot}\ x]\!] &= [\![e]\!]
\end{aligned}
$$

Les listes d'expressions OCaLustre deviennent des n-uplets d'expressions OCaml :

$$
\begin{aligned}
[\![[e]]\!] &= [\![e]\!] \\
[\![e, es]\!] &= [\![e]\!], [\![es]\!]
\end{aligned}
$$

L'opérateur **merge** est traduit de la même façon que l'opérateur conditionnel :

$$
\begin{aligned}
[\![\textbf{if}\ e\ \textbf{then}\ ce_1\ \textbf{else}\ ce_2]\!] &= \texttt{if}\ [\![e]\!]\ \texttt{then}\ [\![ce_1]\!]\ \texttt{else}\ [\![ce_2]\!] \\
[\![\textbf{merge}\ x\ ce_1\ ce_2]\!] &= \texttt{if}\ [\![x]\!]\ \texttt{then}\ [\![ce_1]\!]\ \texttt{else}\ [\![ce_2]\!]
\end{aligned}
$$

#### Traduction d'un nœud et de ses équations

Un nœud OCaLustre est traduit vers une fonction OCaml qui génère une fermeture dont l'environnement contient les registres nécessaires à l'utilisation de l'opérateur *followed-by*, ainsi que les diverses

initialisations des instances de nœuds auxiliaires utilisées dans le nœud d'origine :

$$[\![\textbf{node } f\ (\vec{x})\ \textbf{returns}\ (\vec{y}) = \vec{eqs}]\!] =$$
$$\texttt{let } [\![f]\!]\ \texttt{()} = \mathit{inits}(\vec{eqs}, \texttt{fun } [\![\vec{x}]\!] \texttt{ -> } \mathit{decls}(\vec{eqs}, \mathit{updates}(\vec{eqs}, [\![\vec{y}]\!]))_0)_0$$

**Initialisations :** La fonction *inits* génère les déclarations de variables qui constituent l'environnement de la fermeture créée. Elle est paramétrée par une liste d'équations $\vec{eqs}$, un entier $n$ (nécessaire pour distinguer deux instances d'un même nœud), ainsi qu'une continuation $K$ qui représente les étapes suivantes de compilation.

Chaque utilisation de l'opérateur $\ggg$ entraîne la définition d'un registre (une référence OCaml) initialisé par la valeur à gauche de cet opérateur :

$$\mathit{inits}(y \underset{ck}{=} k\ \textbf{fby}\ e; \vec{eqs}, K)_n = \texttt{let } \_[\![y]\!]\_\texttt{fby} = \texttt{ref } [\![k]\!] \texttt{ in } \mathit{inits}(\vec{eqs}, K)_n$$

Chaque appel à un nœud auxiliaire entraîne la génération d'une instance de ce nœud, en faisant appel à la fonction du même nom. Ces instances sont numérotées pour supporter la situation où il serait fait appel à plusieurs instances d'un même nœud :

$$\mathit{inits}(\vec{y} \underset{ck}{=} g\ (\vec{e}); \vec{eqs}, K)_n = \texttt{let } [\![g]\!]\_[\![n]\!] = [\![g]\!]\ \texttt{()} \texttt{ in } \mathit{inits}(\vec{eqs}, K)_{n+1}$$

Les autres formes d'équations ne produisent aucune déclaration dans l'environnement de la fermeture créée :

$$\mathit{inits}(eq; \vec{eqs}, K)_n = \mathit{inits}(\vec{eqs}, K)_n$$
$$\mathit{inits}(\varnothing, K)_n = K$$

Une liste de variables $\vec{y}$ est traduite vers un n-uplet de variables, ou vers la valeur *unit* si la liste est vide :

$$[\![()]\!] = \texttt{()}$$
$$[\![y]\!] = \texttt{y}$$
$$[\![y, \vec{y}]\!] = [\![y]\!], [\![\vec{y}]\!]$$

**Déclarations :** Le corps du nœud est parcouru par la fonction *decls* afin de convertir chaque équation en déclation de variable dans la fermeture retournée par la fonction générée. Une équation est « gardée » par la condition émanant de son horloge (la fonction *guard* est définie dans la suite de cette section) :

$$\mathit{decls}(y \underset{ck}{=} ce; \vec{eqs}, K)_n = \texttt{let } [\![y]\!] \ = \ \mathit{guard}(ck, ce)_y \texttt{ in } \mathit{decls}(\vec{eqs}, K)_n$$
$$\mathit{decls}(y \underset{ck}{=} k\ \textbf{fby}\ e; \vec{eqs}, K)_n = \texttt{let } [\![y]\!] \ = \ \mathit{guard}(ck, \texttt{!}\_[\![y]\!]\_\texttt{fby})_y \texttt{ in } \mathit{decls}(\vec{eqs}, K)_n$$
$$\mathit{decls}(y \underset{ck}{=} \textbf{call } \texttt{f}\ e_0\ e_1\ \ldots\ e_{m-1}); \vec{eqs}, K)_n = \texttt{let } [\![y]\!] \ = \ \mathit{guard}(ck, \texttt{f } [\![e_0]\!]\ [\![e_1]\!] \ldots [\![e_{m-1}]\!])_y \texttt{ in } \mathit{decls}(\vec{eqs}, K)_n$$
$$\mathit{decls}(\vec{y} \underset{ck}{=} g\ (\vec{e}); \vec{eqs}, K)_n = \texttt{let } [\![\vec{y}]\!] \ = \ \mathit{guard}(ck, [\![g]\!]\_[\![n]\!]\ [\![\vec{e}]\!])_{\vec{y}} \texttt{ in } \mathit{decls}(\vec{eqs}, K)_{n+1}$$
$$\mathit{decls}(\varnothing, K)_n = K$$

Lorsque l'horloge d'un appel de nœud est fausse, une valeur de remplissage représentant l'absence (i.e. le $\perp$ de la sémantique du langage) doit être attribuée à la variable déclarée. Plusieurs solutions sont envisageables pour représenter une telle valeur générique. Il pourrait par exemple être considéré de traduire tous les flots du langage OCaLustre vers des valeurs de type `option`[3], et d'attribuer ainsi à un flot absent la valeur OCaml `None`, mais cette solution provoque une expansion notable de la taille du programme généré, et entraîne des allocations mémoire systématiques. Une autre alternative serait d'utiliser un compilateur OCaml capable de gérer les types *nullables* [MV14], avec le désavantage de perdre la portabilité du code produit. Plus simplement, le compilateur actuel d'OCaLustre remplace toute valeur absente ($\perp$) par la valeur `Obj.magic ()`. Cette valeur spéciale a la particularité d'être considérée comme bien typée quelle que soit son contexte d'utilisation (par exemple, l'expression `Obj.magic () + 42` ne pose pas de problème au compilateur), et elle viole de ce fait les hypothèses qui garantissent la sûreté du typage d'un programme. Il est donc important de pouvoir assurer statiquement qu'à aucun moment de l'exécution du programme cette valeur n'est réellement utilisée, et qu'elle n'est présente que pour satisfaire la construction d'un programme OCaml où une expression doit toujours avoir une valeur (même si elle n'est ici qu'une valeur par défaut jamais utilisée).

La fonction *guard* conditionne donc certaines évaluations d'expressions et les remplace par une valeur d'absence pour le cas où leur horloge est fausse (ou absente). Elle fait appel à une fonction *bottom* qui génère ainsi un représentant de l'absence de valeur qui possède le bon nombre d'éléments :

$$\begin{aligned} guard(\bullet, e)_{\vec{y}} &= e \\ guard(ck, e)_{\vec{y}} &= \texttt{if } [\![ck]\!] \texttt{ then } e \texttt{ else } bottom(\vec{y}) \end{aligned}$$

avec

$$\begin{aligned} [\![\bullet]\!] &= \texttt{true} \\ [\![ck \textbf{ on } x]\!] &= \texttt{(}[\![ck]\!] \texttt{ \&\& } [\![x]\!]\texttt{)} \\ [\![ck \textbf{ onnot } x]\!] &= \texttt{(}[\![ck]\!] \texttt{ \&\& not } [\![x]\!]\texttt{)} \end{aligned}$$

et

$$\begin{aligned} bottom(()) &= \texttt{()} \\ bottom(y) &= \texttt{Obj.magic ()} \\ bottom(y, \vec{y}) &= \texttt{Obj.magic (),}bottom(\vec{y}) \end{aligned}$$

**Mises à jour :** Avant de retourner le n-uplet de variables correspondant aux flots de sortie du nœud, la fermeture générée met à jour chaque registre issu de l'utilisation de $\ggg$ à l'aide de la fonction *updates* :

$$\begin{aligned} updates(\underset{ck}{x = k} \textbf{ fby } e :: e\vec{q}s, K) &= guard(ck, [\![x]\!]\texttt{\_fby := } [\![e]\!])\texttt{ ; } updates(e\vec{q}s, K) \\ updates(eq :: e\vec{q}s, K) &= updates(e\vec{q}s) \text{ (pour les autres formes de } eq) \\ updates(\varnothing, K) &= K \end{aligned}$$

3. La définition de ce type est la suivante : `type 'a option = None | Some of 'a`.

**Traduction d'un programme**

Enfin, générer le code OCaml d'un programme OCaLustre revient à générer séquentiellement le code de chaque nœud qu'il contient :

$$[\![node :: \vec{node}]\!] = [\![node]\!]\,;\,; \\ [\![\vec{node}]\!]$$

$$[\![node :: \varnothing]\!] = [\![node]\!]$$

## 4.5 Exemple

Considérons un exemple de programme OCaLustre simple qui permet d'illustrer toutes les étapes de compilation décrites ci-dessus. Ce programme est constitué de deux nœuds : un premier nœud `count` réalise l'incrémentation d'un compteur à chaque instant, qui est réinitialisé si son paramètre est *vrai*. Le second nœud, `count_and`, compte le nombre de fois que son deuxième et son troisième paramètre sont vrais en même temps tant que son premier paramètre est faux :

```
let%node count (reset) ~return:(cpt) =
  cpt = if reset then 0 else 0 ⋙ (cpt + 1);

let%node count_and (reset,b1,b2) ~return:(clk,cpt_sampled) =
  cpt_sampled = count (reset [@when clk]);
  clk = (b1 && b2)
```

Après normalisation, l'expression à effets de bord `(cpt + 1)` dans `count` est extraite de l'équation `cpt` vers une équation distincte, `cpt_aux` :

```
node count (reset) return (cpt) =
  cpt_aux = 0 fby (cpt + 1);
  cpt = if reset then 0 else cpt_aux

node count_and (reset,b1,b2) return (clk,cpt_sampled) =
  cpt_sampled = count (reset when clk);
  clk = (b1 && b2)
```

L'étape d'ordonnancement réorganise le corps de `count_and` afin que le flot `clk` soit défini avant son utilisation dans `cpt_sampled` :

```
node count (reset) return (cpt) =
  cpt_aux = 0 fby (cpt + 1);
  cpt = if reset then 0 else cpt_aux

node count_and (reset,b1,b2) return (clk,cpt_sampled) =
  clk = (b1 && b2);
  cpt_sampled = count (reset when clk)
```

Après inférence des horloges, les équations et constantes sont annotées par leurs types d'horloges :

```
node count (reset) return (cpt) =
  cpt_aux =• 0 fby (cpt + 1•);
  cpt =• if reset then 0• else cpt_aux

node count_and (reset, b1, b2) return (clk, cpt_sampled) =
  clk =• (b1 && b2);
  cpt_sampled =•on clk count (reset when clk)
```

Enfin, la traduction vers un programme OCaml standard entraîne la définition de deux fonctions, dont chacune correspond à un nœud du programme d'origine :

```
let count () =
  let cpt_aux_fby = ref 0 in
  fun reset ->
    let cpt_aux = !cpt_aux_fby in
    let cpt = if reset then 0 else cpt_aux in
    cpt_aux_fby := (cpt + 1);
    cpt

let count_and () =
  let count1 = count () in
  fun (reset,b1,b2) ->
    let clk = b1 && b2 in
    let cpt_sampled = if clk then count1 reset else Obj.magic () in
    (clk,cpt_sampled)
```

Pour chaque nœud de type $t_{in} \rightarrow t_{out}$ le type de la fonction OCaml générée au terme de ce processus de compilation est alors $unit \rightarrow (t_{in} \rightarrow t_{out})$, ce qui correspond au type d'une fonction qui retourne une *instance* du nœud sous la forme d'une fonction OCaml. Dans l'exemple précédent, le type de `count` est alors $unit \rightarrow (bool \rightarrow int)$, et celui de `count_and` est $unit \rightarrow ((bool * bool * bool) \rightarrow (bool * int))$.

**Génération de la machine d'exécution :**

Le code responsable de la lecture des entrées depuis l'environnement, de l'exécution d'un instant synchrone, et de l'émission des sorties calculées pendant l'instant est nommé *machine d'exécution* [Bou98]. Le code de la fonction principale du programme OCaml, qui implante une machine d'exécution minimale permettant d'exécuter le programme synchrone précédent, a alors la forme suivante :

```
let () =
  (* initialisation du noeud principal *)
  let main = count_and () in
  while true do
   (* lecture des entrées *)
   let (reset,b1,b2) = input_count_and () in
   (* exécution d'un pas du noeud principal *)
   let (clk,cpt_sampled) = main (reset,b1,b2) in
   (* écriture des sorties *)
   output_count_and (clk,cpt_sampled)
  done
```

Ce code peut être généré automatiquement par l'utilisation de l'option `-m` (suivie du nom du nœud principal) du compilateur d'OCaLustre. Cette option entraîne également la génération d'un squelette de code à compléter, qui contient les fonctions d'entrée et de sortie du programme synchrone.

### Conclusion du chapitre

Nous avons décrit dans ce chapitre les différentes étapes permettant de transformer un programme OCaLustre en un programme OCaml séquentiel. Ces transformations apportent plusieurs garanties de typage et d'ordonnançabilité qui permettent de réaliser des programmes dont la sûreté est renforcée. Plusieurs propriétés relatives aux garanties de typage peuvent alors être vérifiées afin de valider cette montée en sûreté. Nous décrirons alors dans le chapitre suivant la formalisation ainsi que la preuve de telles propriétés.

# 5 Propriétés d'OCaLustre formalisées et prouvées avec Coq

Dans ce chapitre, nous décrivons plusieurs propriétés qui dérivent de la spécification formelle d'OCaLustre, et nous les formalisons et en apportons la preuve en Coq. Ces propriétés ont trait aux systèmes de types décrits dans la section 3.2 (qui régissent le typage de données « standard », et le typage des horloges), et permettent de vérifier la cohérence entre ces systèmes formels et l'implémentation dans le prototype de compilateur d'OCaLustre. La vérification de ces propriétés permet ainsi de garantir la sûreté de certains aspects du compilateur d'OCaLustre, comme la correction du typage des valeurs OCaLustre après leur traduction en OCaml, ainsi que l'assurance de la cohérence entre les types d'horloges inférés par OCaLustre, et le système formel décrit précédemment. En particulier, le code d'un *vérificateur d'horloges*, issu de l'extraction avec Coq de la spécification formelle des règles de cadencement d'un programme, est intégré au compilateur d'OCaLustre, qui suit les étapes décrites au chapitre précédent.

## 5.1 Traduction et correction du typage

Dans le but de laisser au typeur inclus dans le compilateur standard d'OCaml la responsabilité de vérifier que les nœuds OCaLustre sont bien typés, nous présentons dans cette section une preuve de la correction du typage d'OCaLustre vis-à-vis de sa traduction.

Cette preuve permet de vérifier deux propriétés qui assurent que les programmes OCaLustre sont bien typés si et seulement si le code OCaml généré est lui-même bien typé :

— Une propriété de *préservation*, qui stipule que si le code OCaLustre est bien typé, alors sa traduction l'est aussi [1].

— Une propriété de *sûreté*, selon laquelle si le code OCaml généré est bien typé, alors le nœud OCaLustre correspondant l'est aussi : on ne perd pas d'informations de typage qui pourraient transformer un programme OCaLustre mal typé en un code OCaml bien typé.

La preuve métathéorique de ces propriétés permet d'éviter l'ajout d'un typeur *ad hoc* à OCaLustre, et ainsi de se reposer sur la puissance du mécanisme de typage du compilateur OCaml natif. C'est en effet le typeur d'OCaml qui, s'il détecte une erreur dans le typage dans le code généré, permet de déduire que le programme OCaLustre d'origine est mal typé.

Dans cette section, nous décrivons progressivement les étapes principales de cette preuve de correction. Notre raisonnement repose par ailleurs uniquement sur des nœuds considérés comme « bien formés ». Nous considérons qu'un nœud *node* est bien formé (noté $\vdash_{wf} node$) si tous les noms des flots qu'il définit sont distincts, si tous ses paramètres sont distincts, et si aucun nom de paramètre en entrée n'est réutilisé pour définir un nouveau nom de flot dans le corps du nœud :

1. On peut voir cette propriété comme une forme de *complétude* dans le sens où on ne rejette pas de programme correct : des erreurs de typage ne sont pas introduites par la traduction.

$\boxed{\vdash_{wf} node}$

$$\frac{distinct(\vec{x}) \quad distinct(\vec{y}) \quad \vec{x} \cap \vec{y} = \varnothing \quad distinct(names(e\vec{q}n)) \quad \vec{x} \cap names(e\vec{q}n) = \varnothing}{\vdash_{wf} \textbf{node}\, f(\vec{x}) \textbf{ returns } (\vec{y}) = e\vec{q}n}$$

avec

$\boxed{distinct(\vec{x})}$

$$\frac{}{distinct(\varnothing)} \qquad \frac{y \notin \vec{y} \quad distinct(\vec{y})}{distinct(y :: \vec{y})}$$

et

$\boxed{names(e\vec{q}n)}$

$$\begin{aligned} &names(\varnothing) \equiv \varnothing \\ &names(x \underset{ck}{=} ce; e\vec{q}n) \equiv x :: names(e\vec{q}n) \\ &names(x \underset{ck}{=} k \textbf{ fby } e; e\vec{q}n) \equiv x :: names(e\vec{q}n) \end{aligned}$$

La preuve de correction vis-à-vis du typage de la traduction présentée dans cette section repose principalement sur un théorème d'équivalence, qui stipule qu'un nœud OCaLustre bien formé est bien typé si et seulement si sa traduction est également bien typée :

**Théorème 5.1.1** (Correction du typage d'un nœud)**.**

$$\forall\, node\ t,\ \vdash_{wf} node \Rightarrow (\vdash node : t \Leftrightarrow \vdash [\![node]\!] : unit\ \rightarrow t)$$

La preuve de cette équivalence permet de vérifier les deux propriétés de préservation et de sûreté. L'implication de gauche à droite permet en effet d'assurer, par contraposée, que si la traduction d'un nœud est mal typée dans OCaml, alors ce dernier est mal typé dans OCaLustre (*préservation*). L'implication de droite à gauche stipule que si la traduction d'un nœud est bien typée, cela signifie que celui-ci est bien typé. Par contraposée, si un nœud est mal typé alors sa traduction l'est aussi (*sûreté*).

L'intégralité des théorèmes et lemmes annexes nécessaires à cette preuve est disponible en ligne, sous forme d'une documentation et de fichiers Coq [⚓1]. Il est à noter toutefois que cette preuve est réalisée à partir d'une version simplifiée d'OCaLustre, ainsi que du langage OCaml (que nous nommerons « *pseudo-ML* ») : en particulier, nous ne traitons pas le polymorphisme paramétrique adopté par OCaml, les programmes OCaLustre seront représentés sous la forme d'un nœud unique (sans possibilité d'appels de nœuds [2]), et les flots ne pourront être que de type *int* ou *bool*. De surcroît, les appels de fonctions OCaml par l'intermédiaire du mot-clé **`call`** n'ont pas été formalisés. Ces derniers ne devraient pas contrevenir à la correction de la preuve car leur compilation est directe : un **`call`** génère une application de fonction (et les types coïncident). L'ajout des appels de nœuds est plus complexe, car il entraîne l'introduction de nouveaux noms dans le code généré après compilation (puisque chaque appel à un nœud conduit à la création d'une instance d'une fermeture dans le code généré). L'intégration de ces constructions pour la preuve de correction du typage de la traduction est un travail actuellement en cours. La version de la preuve présentée dans ce manuscrit et dans les fichiers Coq associés est donc simplifiée.

2. En ce sens, les programmes OCaLustre ici traités s'approchent des programmes Lustre, pour lesquels est réalisée, lors de la compilation, l'intégration du code des différents nœuds intermédiaires dans le nœud principal.

### 5.1.1 Traduction partielle utilisant des déclarations simultanées

Afin de décomposer la preuve de correction du typage, nous prouvons tout d'abord l'équivalence du typage entre le programme OCaLustre et ce même programme traduit dans le langage *pseudo-ML* étendu avec une construction non standard. Cette construction, de la forme « *let ... with ... in ...* », permet la définition simultanée de variables.

Par exemple, l'extrait de code suivant est correct dans *pseudo-ML*, alors qu'il ne le serait pas dans le langage OCaml ne dépendent pas les unes des autres qui exige que les définitions de variables multiples (construites à partir du mot-clé « *and* ») [3] :

```
let x = y with y = 2 in x
```

Cette construction intermédiaire n'a pas vocation à être exécutée, elle permet simplement de factoriser les preuves de correction du typage (en reportant l'introduction de la notion de bon ordonnancement à une étape de preuve décrite plus tard), pour lesquelles le théorème final ne fera pas apparaître de telles déclarations simultanées.

#### Règles de typage de *pseudo-ML*

La grammaire du langage pseudo-ML, très proche de celle d'OCaml, est donnée dans la figure 5.1. La principales différence entre cette dernière et celle d'OCaml correspond à l'ajout de la construction « *let ... with ...* » de déclaration simultanée décrite ci-dessus.

Il est à noter par ailleurs que, afin de simplifier le processus de preuve (en ignorant les phénomènes de masquages de noms), l'espace de noms réservé aux variables nouvellement introduites par la traduction et celui réservé aux variables déjà présentes dans la déclaration du nœud sont distincts par construction : l'environnement de typage $\mathcal{R}$ contient l'ensemble des couples $(nom, type)$ des variables introduites par la compilation, tandis que l'environnement $\Gamma$ concerne les couples $(nom, type)$ des autres variables. En l'occurence, les noms contenus dans $\mathcal{R}$ (issus de la traduction des $\ggg$ ) correspondent tous à des références.

3. Les définitions mutuellement récursives n'étant possibles qu'avec des valeurs de types fonctionnels en OCaml.

| | | | |
|---|---|---|---|
| $e$ | ::= | | expressions |
| | \| | $\bot$ | magic |
| | \| | $()$ | unit |
| | \| | $k$ | constante |
| | \| | $x$ | variable |
| | \| | $x_r$ | registre |
| | \| | **ref** $e$ | référence |
| | \| | $!e$ | déréférencement |
| | \| | $e := e'$ | assignation |
| | \| | $e \diamond e'$ | opérateur binaire |
| | \| | $\square\, e$ | opérateur unaire |
| | \| | **if** $e$ **then** $e'$ **else** $e''$ | conditionnelle |
| | \| | $e$ **;** $e'$ | séquence |
| | \| | $(\vec{e})$ | n-uplet |
| | \| | **fun** $\vec{x} \rightarrow\ e$ | fonction n-aire |
| | \| | **let** $\delta$ **in** $e'$ | déclarations de variables |
| | \| | **let** $\delta_r$ **in** $e'$ | déclarations de variables (nouveaux noms) |
| | \| | **let** $y = e$ **in** $e'$ | déclaration de variable |
| | \| | **let** $y_r = e$ **in** $e'$ | déclaration de variable (nouveau nom) |
| $\delta$ | ::= | | déclarations de variables simultanées |
| | \| | $\varnothing$ | |
| | \| | $(y = e)$ **with** $\delta$ | |
| $\delta_r$ | ::= | | déclarations de variables simultanées (nouveaux noms) |
| | \| | $\varnothing$ | |
| | \| | $(y_r = e)$ **with** $\delta_r$ | |

FIGURE 5.1 – Grammaire de *pseudo-ML*

Les règles de typage des déclarations simultanées sont alors les suivantes :

$\boxed{\mathcal{R}, \Gamma \vdash \delta : \vec{t}}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \varnothing : \varnothing} \qquad \frac{\Gamma(x) = t \quad \mathcal{R}, \Gamma \vdash e : t \quad \mathcal{R}, \Gamma \vdash \delta : \vec{t}}{\mathcal{R}, \Gamma \vdash (x = e) \textbf{ with } \delta : t * \vec{t}}$$

$\boxed{\mathcal{R}, \Gamma \vdash \delta_r : \vec{t}}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \varnothing : \varnothing} \qquad \frac{\mathcal{R}(x) = t \quad \mathcal{R}, \Gamma \vdash e : t \quad \mathcal{R}, \Gamma \vdash \delta_r : \vec{t}}{\mathcal{R}, \Gamma \vdash (x_r = e) \textbf{ with } \delta_r : t * \vec{t}}$$

Elles décrivent que pour qu'un ensemble de déclarations simultanées de variables soit bien typé, chacun de ses éléments doit être bien typé dans le *même* environnement de typage.

L'ensemble des autres règles de typage de *pseudo-ML*, qui ne diffèrent pas particulièrement de celles d'un langage ML classique, sont retranscrites dans la figure 5.2.

$\boxed{\Gamma \vdash \vec{x} : \vec{t}}$

$$\frac{}{\Gamma \vdash \varnothing : \varnothing} \qquad \frac{\Gamma(x) = t \qquad \Gamma \vdash \vec{x} : \vec{t}}{\Gamma \vdash x :: \vec{x} : t * \vec{t}}$$

$\boxed{\mathcal{R}, \Gamma \vdash \vec{e} : \vec{t}}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \varnothing : \varnothing} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t \qquad \mathcal{R}, \Gamma \vdash \vec{e} : \vec{t}}{\mathcal{R}, \Gamma \vdash e, \vec{e} : t * \vec{t}}$$

$\boxed{\mathcal{R}, \Gamma \vdash e : t}$

$$\frac{}{\mathcal{R}, \Gamma \vdash \bot : t} \qquad \frac{}{\mathcal{R}, \Gamma \vdash () : \mathbf{unit}} \qquad \frac{}{\mathcal{R}, \Gamma \vdash \mathit{int\_literal} : \mathbf{int}} \qquad \frac{}{\mathcal{R}, \Gamma \vdash \mathit{bool\_literal} : \mathbf{bool}}$$

$$\frac{\Gamma(x) = t}{\mathcal{R}, \Gamma \vdash x : t} \qquad \frac{\mathcal{R}(x) = t}{\mathcal{R}, \Gamma \vdash x_r : t} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t}{\mathcal{R}, \Gamma \vdash \Box e : t}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : \mathbf{int} \qquad \mathcal{R}, \Gamma \vdash e' : \mathbf{int}}{\mathcal{R}, \Gamma \vdash e \diamond_{int} e' : \mathbf{int}} \qquad \frac{\mathcal{R}, \Gamma \vdash e : \mathbf{bool} \qquad \mathcal{R}, \Gamma \vdash e' : \mathbf{bool}}{\mathcal{R}, \Gamma \vdash e \diamond_{bool} e' : \mathbf{bool}}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : t \qquad \mathcal{R}, \Gamma \vdash e' : t}{\mathcal{R}, \Gamma \vdash e \diamond_{comp} e' : \mathbf{bool}} \qquad \frac{\mathcal{R}, \Gamma \vdash e : \mathbf{bool} \qquad \mathcal{R}, \Gamma \vdash e' : t \qquad \mathcal{R}, \Gamma \vdash e'' : t}{\mathcal{R}, \Gamma \vdash \mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' : t}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : t}{\mathcal{R}, \Gamma \vdash \mathbf{ref}\, e : t\, \mathbf{ref}} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t\, \mathbf{ref}}{\mathcal{R}, \Gamma \vdash\, !e : t} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t\, \mathbf{ref} \qquad \mathcal{R}, \Gamma \vdash e' : t}{\mathcal{R}, \Gamma \vdash e := e' : \mathbf{unit}}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : \mathbf{unit} \qquad \mathcal{R}, \Gamma \vdash e' : t}{\mathcal{R}, \Gamma \vdash e\ ;\ e' : t} \qquad \frac{\mathcal{R}, \Gamma \vdash \vec{e} : \vec{t}}{\mathcal{R}, \Gamma \vdash (\vec{e}) : \vec{t}} \qquad \frac{\Gamma' = \left(\vec{x} : \vec{t}\right) \qquad \mathcal{R}, \Gamma' \cup \Gamma \vdash e : t'}{\mathcal{R}, \Gamma \vdash \mathbf{fun}\, \vec{x} \to\ e : \vec{t} \to t'}$$

$$\frac{\mathcal{R}, \Gamma \vdash e : t \qquad \mathcal{R}, \{x : t\} \cup \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, x = e\, \mathbf{in}\, e' : t'} \qquad \frac{\mathcal{R}, \Gamma \vdash e : t \qquad \{x : t\} \cup \mathcal{R}, \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, x_r = e\, \mathbf{in}\, e' : t'}$$

$$\frac{\mathit{names}(\delta) = \vec{y} \qquad \Gamma' = \left(\vec{y} : \vec{t}\right) \qquad \mathcal{R}, \Gamma' \cup \Gamma \vdash \delta : \vec{t} \qquad \mathcal{R}, \Gamma' \cup \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, \delta\, \mathbf{in}\, e' : t'}$$

$$\frac{\mathit{names}(\delta_r) = \vec{y} \qquad \mathcal{R}' = \left(\vec{y} : \vec{t}\right) \qquad \mathcal{R}' \cup \mathcal{R}, \Gamma \vdash \delta_r : \vec{t} \qquad \mathcal{R}' \cup \mathcal{R}, \Gamma \vdash e' : t'}{\mathcal{R}, \Gamma \vdash \mathbf{let}\, \delta_r\, \mathbf{in}\, e' : t'}$$

FIGURE 5.2 – Règles de typage de *pseudo-ML*

**Règles de typage algorithmiques d'OCaLustre**

Pour la réalisation de la preuve de l'équivalence attendue, nous raisonnons à partir de règles de typage d'OCaLustre *algorithmiques*, semblables aux règles de typage *déclaratives* abordées dans la section 3.2.2. Ces règles sont données dans la figure 5.3. La règle de typage d'un nœud y fait référence à la fonction *names* présentée précédemment, et dont le rôle est de retourner les noms des différents flots définis par une liste d'équations. Il est également à noter que dans le but d'exposer un ensemble de règles aisément compréhensibles, la règle de typage des nœuds implique ici que les nœuds d'un programme OCaLustre ne contiennent aucun flot à portée locale : tous les flots définis dans le corps d'un nœud se retrouvent en sortie. Cette limitation temporaire, qui nous permet de suivre ici la progression de la preuve en Coq, sera levée à la fin de cette section, où nous étendrons notre preuve aux nœuds qui retournent un sous-ensemble des flots définis par les équations présentes dans le corps d'un nœud.

$\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash () : \mathbf{unit}} \qquad \frac{}{\Gamma \vdash \mathit{int\_literal} : \mathbf{int}} \qquad \frac{}{\Gamma \vdash \mathit{bool\_literal} : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \square\, e : t}$$

$$\frac{\Gamma \vdash e : \mathbf{int} \quad \Gamma \vdash e' : \mathbf{int}}{\Gamma \vdash e \diamond_{int} e' : \mathbf{int}} \qquad \frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash e' : \mathbf{bool}}{\Gamma \vdash e \diamond_{bool} e' : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t}{\Gamma \vdash e \diamond_{comp} e' : \mathbf{bool}}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : \mathbf{bool}}{\Gamma \vdash e\ \mathbf{when}\ x : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : \mathbf{bool}}{\Gamma \vdash e\ \mathbf{whennot}\ x : t}$$

$\boxed{\Gamma \vdash_{ce} ce : t}$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash_{ce} e : t} \qquad \frac{\Gamma \vdash x : \mathbf{bool} \quad \Gamma \vdash_{ce} ce : t \quad \Gamma \vdash_{ce} ce' : t}{\Gamma \vdash_{ce} \mathbf{merge}\ x\ ce\ ce' : t} \qquad \frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash_{ce} ce : t \quad \Gamma \vdash_{ce} ce' : t}{\Gamma \vdash_{ce} \mathbf{if}\ e\ \mathbf{then}\ ce\ \mathbf{else}\ ce' : t}$$

$\boxed{\Gamma \vdash ck}$

$$\frac{}{\Gamma \vdash \bullet} \qquad \frac{\Gamma \vdash ck \quad \Gamma(x) = \mathbf{bool}}{\Gamma \vdash ck\ \mathbf{on}\ x} \qquad \frac{\Gamma \vdash ck \quad \Gamma(x) = \mathbf{bool}}{\Gamma \vdash ck\ \mathbf{onnot}\ x}$$

$\boxed{\Gamma \vdash eqn : t}$

$$\frac{\Gamma \vdash ck \quad \Gamma(y) = t \quad \Gamma \vdash_{ce} ce : t}{\Gamma \vdash y \underset{ck}{=} ce : t} \qquad \frac{\Gamma \vdash ck \quad \Gamma(y) = t \quad \Gamma \vdash e : t \quad \Gamma \vdash k : t}{\Gamma \vdash y \underset{ck}{=} k\ \mathbf{fby}\ e : t}$$

$\boxed{\Gamma \vdash \vec{eqn} : \vec{t}}$

$$\frac{}{\Gamma \vdash \varnothing : \varnothing} \qquad \frac{\Gamma \vdash eqn : t \quad \Gamma \vdash \vec{eqn} : \vec{t}}{\Gamma \vdash eqn; \vec{eqn} : t * \vec{t}}$$

$\boxed{\vdash node : t}$

$$\frac{\Gamma = \left(\vec{y} : \vec{t'}\right) \quad \Gamma' = \left(\vec{x} : \vec{t}\right) \quad \Gamma \cup \Gamma' \vdash \vec{eqn} : \vec{t'} \quad names(\vec{eqn}) = \vec{y}}{\vdash \mathbf{node}\, f(\vec{x})\ \mathbf{returns}\ (\vec{y}) = \vec{eqn} : \vec{t} \to \vec{t'}}$$

Figure 5.3 – Règles de typage *algorithmiques* d'OCaLustre

**Traduction vers *pseudo-ML***

La fonction de traduction $[\![\cdot]\!]$ d'un nœud OCaLustre vers du code *pseudo-ML* est donnée dans la figure 5.4. Cette traduction est semblable à celle d'OCaLustre vers OCaml, à ceci près qu'une liste d'équations n'est pas traduite vers une succession de déclarations « *let ... in ...* » imbriquées, mais vers des déclarations simultanées « *let ... with ...* » spécifiques à *pseudo-ML*.

$[\![e]\!]$

$$
\begin{aligned}
[\![()]\!] &\equiv () \\
[\![k]\!] &\equiv k \\
[\![x]\!] &\equiv x \\
[\![e \diamond e']\!] &\equiv [\![e]\!] \diamond [\![e']\!] \\
[\![e \ \mathbf{when}\ x]\!] &\equiv \mathbf{if}\ x\ \mathbf{then}\ [\![e]\!]\ \mathbf{else}\ \bot \\
[\![e \ \mathbf{whennot}\ x]\!] &\equiv \mathbf{if}\ x\ \mathbf{then}\ \bot\ \mathbf{else}\ [\![e]\!] \\
[\![\Box\, e]\!] &\equiv \Box\, [\![e]\!]
\end{aligned}
$$

$[\![ce]\!]$

$$
\begin{aligned}
[\![\mathbf{if}\ e\ \mathbf{then}\ ce'\ \mathbf{else}\ ce'']\!] &\equiv \mathbf{if}\ [\![e]\!]\ \mathbf{then}\ [\![ce']\!]\ \mathbf{else}\ [\![ce'']\!] \\
[\![\mathbf{merge}\ x\ ce\ ce']\!] &\equiv \mathbf{if}\ [\![x]\!]\ \mathbf{then}\ [\![ce]\!]\ \mathbf{else}\ [\![ce']\!]
\end{aligned}
$$

$[\![ck]\!]$

$$
\begin{aligned}
[\![\bullet]\!] &\equiv \mathbf{true} \\
[\![ck\ \mathbf{on}\ x]\!] &\equiv [\![ck]\!]\ \&\&\ x \\
[\![ck\ \mathbf{onnot}\ x]\!] &\equiv [\![ck]\!]\ \&\&\ (\mathbf{not}\ x)
\end{aligned}
$$

$[\![e\vec{q}n]\!]_{inits}$

$$
\begin{aligned}
[\![\varnothing]\!]_{inits} &\equiv \varnothing \\
[\![y \underset{ck}{=} k\ \mathbf{fby}\ e; e\vec{q}n]\!]_{inits} &\equiv (y_r = \mathbf{ref}\ k)\ \mathbf{with}\ [\![e\vec{q}n]\!]_{inits} \\
[\![eqn; e\vec{q}n]\!]_{inits} &\equiv [\![e\vec{q}n]\!]_{inits}
\end{aligned}
$$

$[\![e\vec{q}n]\!]$

$$
\begin{aligned}
[\![\varnothing]\!] &\equiv \varnothing \\
[\![y \underset{ck}{=} ce; e\vec{q}n]\!] &\equiv (y = \mathbf{if}\ [\![ck]\!]\ \mathbf{then}\ [\![ce]\!]\ \mathbf{else}\ \bot)\ \mathbf{with}\ [\![e\vec{q}n]\!] \\
[\![y \underset{ck}{=} k\ \mathbf{fby}\ e; e\vec{q}n]\!] &\equiv (y = \mathbf{if}\ [\![ck]\!]\ \mathbf{then}\ !\, y_r\ \mathbf{else}\ \bot)\ \mathbf{with}\ [\![e\vec{q}n]\!]
\end{aligned}
$$

$[\![e\vec{q}n]\!]_{updates}$

$$
\begin{aligned}
[\![\varnothing]\!]_{updates} &\equiv () \\
[\![y \underset{ck}{=} k\ \mathbf{fby}\ e; e\vec{q}n]\!]_{updates} &\equiv (\mathbf{if}\ [\![ck]\!]\ \mathbf{then}\ (y_r := [\![e]\!])\ \mathbf{else}\ ())\ ;\ [\![e\vec{q}n]\!]_{updates} \\
[\![eqn; e\vec{q}n]\!]_{updates} &\equiv [\![e\vec{q}n]\!]_{updates}
\end{aligned}
$$

$[\![node]\!]$

$$[\![\mathbf{node}\, f(\vec{x})\ \mathbf{returns}\ (\vec{y}) = e\vec{q}n]\!] \equiv \mathbf{fun}\ () \rightarrow\ \mathbf{let}\ [\![e\vec{q}n]\!]_{inits}\ \mathbf{in}\ (\mathbf{fun}\ \vec{x} \rightarrow\ \mathbf{let}\ [\![e\vec{q}n]\!]\ \mathbf{in}\ ([\![e\vec{q}n]\!]_{updates}\ ;\ ([\![\vec{y}]\!])))$$

FIGURE 5.4 – Fonction de traduction d'OCaLustre vers *pseudo-ML*

**Correction du typage en *pseudo-ML***

À partir des fonctions et règles définies dans les sections précédentes, nous présentons alors les diverses étapes permettant de prouver la correction du typage vis-à-vis de la traduction d'un nœud OCaLustre vers un programme pseudo-ML.

D'abord, il est prouvé que toute expression simple ($e$) conserve, à partir du même environnement $\Gamma$, le même type après sa traduction :

**Lemme 5.1.1** (Correction du typage des expressions simples)**.**

$$\forall\ \mathcal{R}\ \Gamma\ e\ t, (\Gamma \vdash e : t \Leftrightarrow \mathcal{R}, \Gamma \vdash [\![e]\!] : t)$$

De ce lemme, on en déduit que toute expression conditionnelle ($ce$) conserve également le même type après traduction :

**Lemme 5.1.2** (Correction du typage des expressions conditionnelles)**.**

$$\forall\ \mathcal{R}\ \Gamma\ ce\ t, (\Gamma \vdash ce : t \Leftrightarrow \mathcal{R}, \Gamma \vdash [\![ce]\!] : t)$$

À partir de ces lemmes, nous prouvons qu'une liste d'équations conserve le même type après traduction, à la condition que les étapes d'initialisation et de mise à jour des registres dans la fonction générée (qui peuvent apparaître dans les équations traduites) soient bien typées dans l'environnement $\mathcal{R}$. De plus, les noms des flots définis par ces équations doivent être tous distincts : par exemple, un même flot x ne peut pas être défini deux fois, par deux équations différentes, dans le même nœud.

**Lemme 5.1.3** (Correction du typage des équations)**.** $\forall\ \mathcal{R}\ \Gamma\ \vec{eqn}\ t$,

$$distinct(names(\vec{eqn})) \Rightarrow \mathcal{R}, \varnothing \vdash [\![\vec{eqn}]\!]_{inits} : t \Rightarrow \mathcal{R}, \Gamma \vdash [\![\vec{eqn}]\!]_{updates} : unit \Rightarrow \forall t', (\Gamma \vdash \vec{eqn} : t' \Leftrightarrow \mathcal{R}, \Gamma \vdash [\![\vec{eqn}]\!] : t')$$

Nous raisonnons tout d'abord sur des nœuds dont toutes les équations qui sont définies dans leur corps sont des flots de sortie. Autrement dit, nous traitons initialement des nœuds qui ne possèdent pas de flots dont la portée est uniquement locale. Un nœud *node* qui ne définit pas de flot à portée locale respecte alors le prédicat $nolocals(node)$ :

$\boxed{nolocals(node)}$

$$\frac{\vec{y} = names(\vec{eqn})}{nolocals(\mathbf{node}\, f(\vec{x})\ \mathbf{returns}\ (\vec{y}) = \vec{eqn})}$$

La combinaison des lemmes précédents permet de prouver le lemme de correction du typage d'un nœud, qui stipule que, pour tout type $t$, si un nœud *node* est bien formé, alors il est de type $t$ si et seulement si sa traduction est de type $unit \to t$ (dans un environnement initialement vide) :

**Lemme 5.1.4** (Correction du typage d'un nœud (sans flot à portée locale))**.**

$$\forall\ node\ t, \vdash_{wf} node \Rightarrow nolocals(node) \Rightarrow (\vdash node : t \Leftrightarrow \varnothing, \varnothing \vdash [\![node]\!] : unit \to t)$$

Le lemme précédent n'est applicable qu'aux nœuds pour lesquels aucun flot n'est à portée locale. Autrement dit, la fonction *pseudo-ML* générée pour de tels nœuds retourne un n-uplet qui contient

exactement la liste de toutes les variables définies dans la fonction. Dans le but d'étendre cette propriété à toute définition de nœud (y compris celles qui font usage de flots définis localement – et donc non contenus dans la liste des variables en sortie), nous dérivons de la règle de typage d'un nœud présentée dans la figure 5.3 une nouvelle règle qui permet en particulier de faire référence au fait que les flots en sortie sont un sous-ensemble des flots calculés dans un nœud :

$\boxed{\vdash node : t}$

$$\frac{\vdash \mathbf{node}\, f(\vec{x})\, \mathbf{returns}\, (\vec{y}) = e\vec{q}n : \vec{t} \rightarrow \vec{t'} \qquad \vdash_{wf} \mathbf{node}\, f(\vec{x})\, \mathbf{returns}\, (\vec{y}) = e\vec{q}n \qquad (\vec{z} : \vec{t'}) \subseteq (\vec{y} : \vec{t''})}{\vdash \mathbf{node}\, f(\vec{x})\, \mathbf{returns}\, (\vec{z}) = e\vec{q}n : \vec{t} \rightarrow \vec{t'}}$$

Nous dérivons alors enfin le lemme de correction du typage d'un nœud OCaLustre bien formé, qui peut potentiellement contenir des flots à portée locale :

**Lemme 5.1.5** (Correction du typage d'un nœud)**.**

$$\forall\, node\ t, \vdash_{wf} node \Rightarrow (\vdash node : t \Leftrightarrow \varnothing, \varnothing \vdash [\![node]\!] : unit \rightarrow t)$$

Nous pouvons ainsi en déduire que tout programme OCaLustre cohérent du point de vue des règles de typage de ce langage entraîne la réalisation d'un programme *pseudo-ML* qui est lui-même correctement typé.

### 5.1.2 Traduction utilisant des déclarations imbriquées

Le langage *pseudo-ML* possède, en plus de la construction « *let ... with ... in ...* » abordée dans la section précédente, une construction « *let ... in ...* » ne permettant de déclarer qu'une seule variable, à la manière du langage OCaml. Ainsi, convertir un code *pseudo-ML* faisant usage de déclarations simultanées en un code *pseudo-ML* faisant usage d'une succession de déclarations imbriquées reviendrait à produire un programme dont les règles de typage sont semblables à celui d'un programme OCaml. De ce fait, prouver que la traduction des programmes OCaLustre vers pseudo-ML conserve sa correction de typage dès lors que les « *let ... with ... in ...* » sont remplacés par des « *let ... in ...* » permet de vérifier que le programme OCaml séquentiel généré à partir d'un nœud OCaLustre (détaillé précédement dans la section 4.4) est lui même bien typé, puisque les règles de traduction vers *pseudo-ML* et vers OCaml sont dans ce cas très proches [4].

La fonction $\ll \cdot \gg$ de conversion d'un programme contenant des déclarations simultanées vers un programme qui contient des déclarations imbriquées est décrite dans la figure 5.5.

Néanmoins, l'équivalence de typage entre un code contenant des déclarations simultanées et un code contenant des déclarations imbriquées ne peut être vérifiée qu'à condition qu'il existe un ordre d'imbrication des « *let* » dans lequel aucune variable non encore déclarée n'est utilisée par les déclarations des variables antérieures. Par exemple, le simple fait de remplacer « *let ... with ...* » par des « *let ... in ...* » dans l'extrait de code présenté dans la section précédente serait incorrect (en OCaml comme en pseudo-ML) puisqu'il serait alors fait référence à la variable $y$ pour associer une valeur à la variable $x$ avant même que $y$ soit définie :

4. Même si quelques différences subsistent, comme le fait que certaines variables soient typées dans un environnement distinct.

$\boxed{\ll \delta_r \gg_{e'}}$

$$\ll \varnothing \gg_e \equiv e$$
$$\ll (y_r = e') \textbf{ with } \delta_r \gg_e \equiv \textbf{let } y_r = e' \textbf{ in } \ll \delta_r \gg_e$$

$\boxed{\ll \delta \gg_{e'}}$

$$\ll \varnothing \gg_e \equiv e$$
$$\ll (y = e') \textbf{ with } \delta \gg_e \equiv \textbf{let } y = e' \textbf{ in } \ll \delta \gg_e$$

$\boxed{\ll e' \gg}$

$$\ll \textbf{let } \delta \textbf{ in } e \gg \equiv \ll \delta \gg_{\ll e \gg}$$
$$\ll \textbf{let } \delta_r \textbf{ in } e \gg \equiv \ll \delta_r \gg_{\ll e \gg}$$
$$\ll \textbf{fun } \vec{x} \rightarrow\ e \gg \equiv \textbf{fun } \vec{x} \rightarrow \ll e \gg$$
$$\ll e \gg \equiv e$$

FIGURE 5.5 – Fonction de conversion de déclarations simultanées vers déclarations imbriquées

```
let x = y in let y = 2 in x
```

tandis que le changement de l'ordre des déclarations correspond à un code correct, à la sémantique attendue :

```
let y = 2 in let x = y in x
```

La conversion des déclarations simultanées vers des déclarations imbriquées n'est donc valide qu'à condition qu'il existe un ordre d'apparition des variables dans des déclarations simultanées qui permette de ne pas faire référence à une variable avant sa définition. Cette propriété est en réalité exactement équivalente à la notion de « bon ordonnancement » d'un programme OCaLustre. En effet, l'étape de réordonnancement des programmes OCaLustre est justement destinée à trouver un ordre de déclaration de variables qui permette la définition de variables dans un code séquentiel.

Des déclarations simultanées *pseudo-ML* issues de la traduction d'équations d'un nœud OCaLustre ne sont ainsi convertibles en déclarations imbriquées qu'à la condition que ces équations soient bien ordonnancées :

**Théorème 5.1.2** (Équivalence *let ... with* et *let* (déclarations))**.**

$$\forall\ \vec{x}\ \vec{t_x}\ \vec{eqn}, \delta = [\![\vec{eqn}]\!] \Rightarrow \vec{x} \cap\ vars(\delta) = \varnothing \Rightarrow distincts(vars(\delta)) \Rightarrow \vec{x} \vdash_{ws} eqns \Rightarrow$$
$$\forall\ \Gamma\ t,\ \Gamma = (\vec{x} : \vec{t_x}) \Rightarrow (\mathcal{R}, \Gamma \vdash lets\ \delta\ in\ e : t \Leftrightarrow \mathcal{R}, \Gamma \vdash \ll \delta \gg_e : t)$$

Par conséquent, tout programme *pseudo-ML* (issu de la traduction d'un nœud OCaLustre bien ordonnancé) avec déclarations simultanées est bien typé si et seulement si un programme identique dans lequel les déclarations simultanées sont converties en déclarations imbriquées est lui aussi bien typé, et du même type :

**Lemme 5.1.6** (Équivalence let with et let (nœud))**.**

$$\forall\ node\ t,\ \vdash_{ws} node \Rightarrow (\vdash [\![node]\!] : t \Leftrightarrow \vdash \ll [\![node]\!] \gg : t)$$

En conclusion, nous pouvons en déduire des lemmes et théorèmes énoncés dans cette section que tout nœud OCaLustre bien ordonnancé est bien typé si et seulement si sa traduction dans un langage ML avec déclarations imbriquées (comme OCaml) est elle aussi bien typée :

**Théorème 5.1.3** (Correction du typage d'un nœud traduit en OCaml)**.**

$$\forall\ node\ t,\ \vdash_{ws} node \Rightarrow (\vdash node : t \Leftrightarrow \vdash \ll [\![node]\!] \gg : t)$$

Ce théorème nous permet de valider le fait qu'il n'est pas réellement utile de réaliser un typeur spécifique à OCaLustre, puisque le code OCaml généré par compilation d'un nœud possède l'intégralité des informations de typage de ce dernier. Ainsi, tout nœud mal typé en OCaLustre sera détecté par le typeur du compilateur OCaml. Pour étendre cette propriété de correction à l'intégralité des programmes OCaLustre, il serait toutefois nécessaire de considérer en particulier le mécanisme d'appels de nœuds, qui est converti vers des applications de fonctions en OCaml.

## 5.2 Vérification du typage d'horloges

Le système d'horloges synchrones, intégré au langage OCaLustre, offre la possibilité au développeur d'associer des *conditions de présence* à chacun des flots manipulés par un programme synchrone. La sémantique de typage des horloges, définie dans la section 3.2.4, permet de contraindre les opérateurs du langage à n'opérer (pour la plupart) que sur des flots étant cadencés par la même horloge. De ce fait, le respect de cette sémantique entraîne l'assurance qu'aucune valeur de flot absente n'est lue pendant l'exécution du programme, évitant ainsi tout comportement erroné et imprévisible.

Afin d'attribuer à chaque flot du langage une horloge, le compilateur du langage OCaLustre implémente un algorithme d'inférence, sur le modèle de ceux implantés par Heptagon et Scade et dérivés des travaux de Colaço et Pouzet [CP03] (sur la base de l'algorithme $\mathcal{W}$ décrit par Milner [Mil78]) qui associe statiquement chaque expression du langage à une horloge synchrone en respectant la sémantique de typage d'horloges.

Dans le but d'assurer formellement que les horloges inférées par le compilateur respectent bien la sémantique de cadencement du langage, nous proposons dans cette section un *vérificateur d'horloge* pour OCaLustre. Le rôle de ce vérificateur, intégré au compilateur d'OCaLustre, est de lire un arbre de syntaxe abstraite dans lequel chaque expression est annotée avec son horloge synchrone (inférée par l'algorithme de typage d'horloges) et d'indiquer si cet AST est correct vis-à-vis de la sémantique de typage des horloges synchrones. La validation par le vérificateur d'horloges contribue ainsi à la sûreté du langage, en assurant que les horloges inférées sont conformes à la spécification formelle donnée dans la section 3.2.4.

Le vérificateur d'horloge réalisé prend alors place dans le schéma de compilation d'un programme OCaLustre juste après le processus d'inférence d'horloges (fig. 5.6) : si les horloges inférées sont cohérentes vis à vis des règles de typage connues du vérificateur, alors la compilation d'un programme OCaLustre

peut se poursuivre. Dans le cas contraire, le processus de compilation est stoppé, et le compilateur retournera alors une erreur.

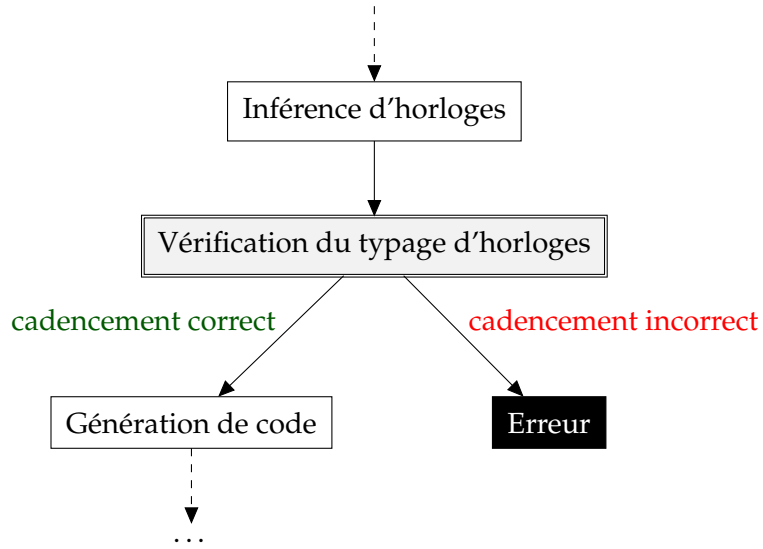



FIGURE 5.6 – Insertion du vérificateur d'horloge à la chaîne de compilation d'un programme OCaLustre

La solution détaillée dans cette section consiste de ce fait en la vérification *a posteriori* de la cohérence des horloges attribuées aux expressions, et représente alors un compilateur *certifiant* [PSS98] la sûreté des types d'horloge inférés. Une telle méthode constitue une alternative à un compilateur *certifié* pour lequel la preuve de correction de l'algorithme d'inférence d'horloge implanté dans OCaLustre aurait été réalisée.

Il est à noter que la vérification du bon cadencement d'un nœud est compatible avec la possibilité pour ce nœud d'avoir des entrées ou des sorties qui soient absentes. En ce sens, nos travaux se rapprochent de travaux récents qui poursuivent les efforts de conception d'un compilateur Lustre certifié en y ajoutant des *arguments cadencés* [BP19].

### 5.2.1 Règles algorithmiques de bon cadencement

Le vérificateur d'horloges d'OCaLustre s'assure que le programme, représenté sous forme d'AST annoté par les horloges décrit dans la section 4, respecte une sémantique de typage représentant le bon cadencement des flots. Cette sémantique, dérivant de la sémantique de typage présentée dans la section 3.2.4, tient compte des annotations ajoutées à l'AST du programme OCaLustre lors de sa compilation afin d'en vérifier le bon cadencement.

Les règles *algorithmiques* de cadencement des expressions, dans cette version de l'AST annoté par les horloges, sont retranscrites dans la figure 5.7. Elles sont quasiment identiques à celles des expressions non-annotées, à l'exception des règles de cadencement de la valeur *unit*, d'une constante, ou d'un constructeur de type, qui tiennent compte de ces annotations. Par exemple, alors que dans sa version non-annotée une constante était compatible avec un quelconque type d'horloge, une constante $k$ ici annotée avec une horloge $ck$ a pour type d'horloge $ck$ :

$$\frac{}{\mathcal{C} \vdash k^{ck} : ck}\ \textsc{Const}$$

$\boxed{\mathcal{C} \vdash e : ck}$

$$\frac{}{\mathcal{C} \vdash ()^{ck} : ck}\ \textsc{Unit} \qquad \frac{}{\mathcal{C} \vdash k^{ck} : ck}\ \textsc{Const} \qquad \frac{}{\mathcal{C} \vdash X_i^{ck} : ck}\ \textsc{Constr}$$

$$\frac{\mathcal{C}(x) = ck}{\mathcal{C} \vdash x : ck}\ \textsc{Var} \qquad \frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash \square\, e : ck}\ \textsc{Unop} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e' : ck}{\mathcal{C} \vdash e \diamond e' : ck}\ \textsc{Binop}$$

$$\frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\ \mathbf{when}\ x : ck\ \mathbf{on}\ x}\ \textsc{When} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash x : ck}{\mathcal{C} \vdash e\ \mathbf{whennot}\ x : ck\ \mathbf{onnot}\ x}\ \textsc{Whennot}$$

$\boxed{\mathcal{C} \vdash \vec{e} : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash [e] : ck}\ \textsc{One\_E} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash \vec{e} : ck'}{\mathcal{C} \vdash e, \vec{e} : ck \times ck'}\ \textsc{Cons\_E}$$

$\boxed{\mathcal{C} \vdash_{ce} ce : ck}$

$$\frac{\mathcal{C} \vdash e : ck}{\mathcal{C} \vdash_{ce} e : ck}\ \textsc{Exp} \qquad \frac{\mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash_{ce} ce : ck \quad \mathcal{C} \vdash_{ce} ce' : ck}{\mathcal{C} \vdash_{ce} \mathbf{if}\ e\ \mathbf{then}\ ce\ \mathbf{else}\ ce' : ck}\ \textsc{If}$$

$$\frac{\mathcal{C} \vdash x : ck \quad \mathcal{C} \vdash_{ce} ce : ck\ \mathbf{on}\ x \quad \mathcal{C} \vdash_{ce} ce' : ck\ \mathbf{onnot}\ x}{\mathcal{C} \vdash_{ce} \mathbf{merge}\ x\ ce\ ce' : ck}\ \textsc{Merge}$$

FIGURE 5.7 – Règles de cadencement, en forme normale annotée par les horloges, des expressions

L'ensemble des règles de cadencement qui concernent les autres catégories syntaxiques est donné dans la figure 5.8. Ces règles de cadencement sont pour la plupart directement dérivées des règles présentées en section 3.2.4, néanmoins la règle qui concerne l'application d'un nœud est plus complexe. Elle doit tenir compte, comme nous l'avons décrit lors de la présentation du système des horloges d'OCaLustre, du mécanisme d'application conditionnelle, ainsi que des diverses substitutions apportées aux noms des variables dans les types d'horloges.

$\boxed{\mathcal{C} \vdash \vec{y} : ck}$

$$\frac{}{\mathcal{C} \vdash () : \bullet}\ \textsc{NNil} \qquad \frac{\mathcal{C}(y) = ck}{\mathcal{C} \vdash y : ck}\ \textsc{NVariable} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{C} \vdash y, \vec{y} : ck \times ck'}\ \textsc{NList}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash eqn}$

$$\frac{\vec{x} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma_1 \quad ck_1 = \sigma_1(ck'_1[ck]) \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \rhd \sigma_2 \quad ck_2 = \sigma_2 \oplus \sigma_1(ck'_2[ck])}{ck_1 \to ck_2 = \underset{(\vec{e},\vec{y'},ck)}{inst}\left(\forall \bullet.(\vec{x}{:}ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}{:}ck'_2)\right)}\ \textsc{Inst}$$

$$\frac{\mathcal{H}(f) = \forall \bullet .(\vec{x} : ck) \to (\vec{y} : ck') \quad \mathcal{S} = carriers(ck) +\!\!+ carriers(ck')}{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck) \xrightarrow{\mathcal{S}} (\vec{y} : ck')}\ \textsc{Sign}$$

$$\frac{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck'_1) \xrightarrow{\mathcal{S}} (\vec{y'} : ck'_2) \quad ck_1 \to ck_2 = \underset{(\vec{e},\vec{y'},ck)}{inst}\left(\forall \bullet.(\vec{x}{:}ck'_1) \xrightarrow{\mathcal{S}} (\vec{y}{:}ck'_2)\right) \quad \mathcal{C} \vdash \vec{e} : ck_1 \quad \mathcal{C} \vdash \vec{y'} : ck_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y'} \underset{ck}{=} f(\vec{e})}\ \textsc{App}$$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash_{ce} ce : ck}{\mathcal{H}, \mathcal{C} \vdash y \underset{ck}{=} ce}\ \textsc{Expr} \qquad \frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e : ck}{\mathcal{H}, \mathcal{C} \vdash y \underset{ck}{=} k\ \textbf{fby}\ e}\ \textsc{Fby}$$

$$\frac{\mathcal{C} \vdash y : ck \quad \mathcal{C} \vdash e : ck \quad \mathcal{C} \vdash e_1 : ck \quad .. \quad \mathcal{C} \vdash e_{n-1} : ck}{\mathcal{H}, \mathcal{C} \vdash y \underset{ck}{=} \textbf{call}\ \mathtt{f}\ e\ e_1\ ..\ e_{n-1}}\ \textsc{Call}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash \vec{eqn}}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash eqn}{\mathcal{H}, \mathcal{C} \vdash [eqn]}\ \textsc{OneEqn} \qquad \frac{\mathcal{H}, \mathcal{C} \vdash eqn \quad \mathcal{H}, \mathcal{C} \vdash \vec{eqn}}{\mathcal{H}, \mathcal{C} \vdash eqn;\ \vec{eqn}}\ \textsc{ConsEqns}$$

$\boxed{\mathcal{H}, \mathcal{C} \vdash node : \omega}$

$$\frac{\mathcal{H}, \mathcal{C} \vdash \vec{eqn} \quad \mathcal{C} \vdash \vec{x} : ck \quad \mathcal{C} \vdash \vec{y} : ck'}{\mathcal{H}, \mathcal{C} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn} : \forall \bullet .(\vec{x} : ck) \to (\vec{y} : ck')}\ \textsc{Node}$$

$\boxed{\mathcal{H} \vdash program}$

$$\frac{}{\mathcal{H} \vdash \varnothing}\ \textsc{EmptyProg} \qquad \frac{\mathcal{H}, \mathcal{C} \vdash \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn} : \omega \quad (f : \omega) \cup \mathcal{H} \vdash \vec{nodes}}{\mathcal{H} \vdash (\mathcal{C}, \textbf{node}\, f(\vec{x})\, \textbf{return}\, (\vec{y}) = \vec{eqn});; \vec{nodes}}\ \textsc{ConsProg}$$

Figure 5.8 – Règles de cadencement dans la forme normale annotée par les horloges (suite)

Pour représenter ces substitutions, nous introduisons alors le jugement $\vec{x} \underset{\mathcal{S}}{\sim} \vec{e} \triangleright \sigma$ qui indique que, pour un ensemble de supports $\mathcal{S}$, une liste de paramètres formels $\vec{x}$, et une liste d'arguments effectifs $\vec{e}$, $\sigma$ est la substitution des noms de variables présents dans la liste des paramètres formels de l'application vers les noms de ses arguments.

Les règles inductives liées à ce jugement sont les suivantes :

$\boxed{\vec{y} \underset{\mathcal{S}}{\sim} \vec{e} \triangleright \sigma}$

$$\frac{x \in \mathcal{S}}{x \underset{\mathcal{S}}{\sim} [y] \triangleright \{x \mapsto y\}}\ \textsc{Sub\_E\_in} \qquad \frac{x \notin \mathcal{S}}{x \underset{\mathcal{S}}{\sim} [e] \triangleright \varnothing}\ \textsc{Sub\_E\_notin}$$

$$\frac{}{() \underset{\mathcal{S}}{\sim} [()^{ck}] \triangleright \varnothing}\ \textsc{Sub\_E\_unit} \qquad \frac{y \underset{\mathcal{S}}{\sim} [e] \triangleright \sigma \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{e} \triangleright \sigma'}{y, \vec{y} \underset{\mathcal{S}}{\sim} e, \vec{e} \triangleright \sigma \oplus \sigma'}\ \textsc{Sub\_E\_list}$$

Par exemple, si un nœud $f$ a pour signature $(x : \bullet) \to (y : \bullet \text{ on } x)$, alors l'application $f\ z$ entraîne la substitution de $x$ par $z$ pour l'instantiation du type de $f$ dans ce contexte : par conséquent, dans $f\ z$, $f$ est de type $\bullet \to \bullet$ on $z$. On peut donc formellement en déduire le jugement $x \underset{\{x\}}{\sim} z \triangleright \{x \mapsto z\}$ .

En raison du fait que les flots de sortie d'un nœud peuvent être cadencés par d'autres flots de sortie, il est également nécessaire de substituer le nom des paramètres de sortie d'un nœud par les noms réels à gauche du symbole d'égalité dans l'équation correspondant à l'application d'un nœud. Le jugement $\vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \triangleright \sigma$ signifie ainsi que, pour un ensemble de supports $\mathcal{S}$, $\sigma$ est la substitution des noms des paramètres de sortie formels $\vec{y}$ par les noms effectifs des flots en sortie $\vec{y'}$ (correspondant aux noms des variables à gauche du signe d'égalité dans l'équation concernée). Les règles inductives régissant ce jugement portent sur une catégorie syntaxique différente des règles précédentes : elles traitent des listes de noms de variables $\vec{y}$, et non plus des expressions $e$. Ces règles sont les suivantes :

$\boxed{\vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \triangleright \sigma}$

$$\frac{y \in \mathcal{S}}{y \underset{\mathcal{S}}{\sim} y' \triangleright \{y \mapsto y'\}}\ \textsc{Sub\_V\_in} \qquad \frac{y \notin \mathcal{S}}{y \underset{\mathcal{S}}{\sim} y' \triangleright \varnothing}\ \textsc{Sub\_V\_notin}$$

$$\frac{}{() \underset{\mathcal{S}}{\sim} () \triangleright \varnothing}\ \textsc{Sub\_V\_nil} \qquad \frac{y \underset{\mathcal{S}}{\sim} y' \triangleright \sigma \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \triangleright \sigma'}{y, \vec{y} \underset{\mathcal{S}}{\sim} y', \vec{y'} \triangleright \sigma \oplus \sigma'}\ \textsc{Sub\_V\_list}$$

Par exemple, si le nœud $g$ a pour signature $(x : \bullet) \to ((y : \bullet) \times (z : \bullet \text{ on } y))$, alors dans l'équation $(a, b) = g(32)$ le type de l'instance de $g$ est $\bullet \to (\bullet \times (\bullet \text{ on } a))$ car $a$ est le nom effectif utilisé pour faire référence à la première valeur de sortie de $g$. Cet exemple induit le jugement $(y, z) \underset{\{y\}}{\sim} (a, b) \triangleright \{y \mapsto a\}$.

Nous rappelons également que le mécanisme d'application conditionnelle d'un nœud consiste à remplacer l'horloge de base dans la signature d'un nœud par une horloge plus lente afin d'en ralentir l'exécution. Nous noterons $\vec{ck}[c]$ la substitution de tous les occurrences de l'horloge de base ($\bullet$) par l'horloge $c$ dans $\vec{ck}$.

$\boxed{ck[ck']}$

$$
\begin{aligned}
&\bullet[ck'] \equiv ck' \\
&(ck\ \mathbf{on}\ x)[ck'] \equiv (ck[ck'])\ \mathbf{on}\ x \\
&(ck\ \mathbf{onnot}\ x)[ck'] \equiv (ck[ck'])\ \mathbf{onnot}\ x \\
&(ck_1 \to ck_2)[ck'] \equiv (ck_1[ck']) \to (ck_2[ck']) \\
&(ck_1 \times ck_2)[ck'] \equiv (ck_1[ck']) \times (ck_2[ck'])
\end{aligned}
$$

De surcroît, nous donnons la définition d'une fonction *carriers*, qui calcule la liste des supports contenus dans un type d'horloge :

$\boxed{carriers(ck)}$

$$
\begin{aligned}
&carriers(\bullet) \equiv \varnothing \\
&carriers(ck\ \mathbf{on}\ x) \equiv x :: carriers(ck) \\
&carriers(ck\ \mathbf{onnot}\ x) \equiv x :: carriers(ck) \\
&carriers(ck_1 \to ck_2) \equiv carriers(ck_1) +\!\!+ carriers(ck_2) \\
&carriers(ck_1 \times ck_2) \equiv carriers(ck_1) +\!\!+ carriers(ck_2)
\end{aligned}
$$

Afin de rendre la compréhension de la règle de l'application d'un nœud plus aisée, nous la séparons en trois règles distinctes :

1. Une première règle, Sign

$$
\frac{\mathcal{H}(f) = \forall \bullet .(\vec{x} : ck) \to (\vec{y} : ck') \quad \mathcal{S} = carriers(ck) +\!\!+ carriers(ck')}{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck) \xrightarrow{\mathcal{S}} (\vec{y} : ck')}\ \textsc{Sign}
$$

   permet de récupérer les informations nécessaires au typage de l'application. Elle stipule que, dans un environnement de typage global $\mathcal{H}$, si la fonction $f$ a pour signature $(\vec{x} : ck) \to (\vec{y} : ck')$, et si l'ensemble $\mathcal{S}$ correspond aux supports dans $ck$ et $ck'$, alors on peut en déduire le jugement $f : (\vec{x} : ck) \xrightarrow{\mathcal{S}} (\vec{y} : ck')$, qui associe la signature de $f$ et sa liste de supports.

2. Une seconde règle, la règle d'instantiation Inst :

$$
\frac{\vec{x} \underset{\mathcal{S}}{\sim} \vec{e} \rhd \sigma_1 \quad ck_1 = \sigma_1(ck'_1[ck]) \quad \vec{y} \underset{\mathcal{S}}{\sim} \vec{y'} \rhd \sigma_2 \quad ck_2 = \sigma_2 \oplus \sigma_1(ck'_2[ck])}{ck_1 \to ck_2 = \underset{(\vec{e},\vec{y'},ck)}{inst}\left(\forall \bullet .(\vec{x} : ck'_1) \xrightarrow{\mathcal{S}} (\vec{y} : ck'_2)\right)}\ \textsc{Inst}
$$

   indique que, pour un nœud de signature $(\vec{x} : ck'_1) \to (\vec{y} : ck'_2)$ possédant un ensemble $\mathcal{S}$ de supports, si les conditions suivantes sont respectées :
   - $\sigma_1$ est la substitution des noms des paramètres formels $\vec{x}$ du nœud vers les noms de ses arguments $\vec{e}$.
   - L'horloge $ck_1$ correspond au résultat de l'application à $ck'_1$ de la substitution $\sigma_1$ ainsi que de la substitution de l'horloge de base par l'horloge $ck$ d'application conditionnelle du nœud.
   - $\sigma_2$ est la substitution des noms formels $\vec{y}$ des flots de sortie du nœud vers les noms des variables $\vec{y'}$ effectivement présentes à gauche du signe d'égalité dans l'équation qui réalise l'application.

— L'horloge $ck_2$ correspond au résultat de l'application à $ck'_2$ des substitutions $\sigma_1$ et $\sigma_2$, ainsi que de la substitution de l'horloge de base par l'horloge $ck$ d'application conditionnelle de l'appel de nœud.

alors le type $ck_1 \to ck_2$ est une *instance* valide du type du nœud considéré.

3. Enfin, la troisième règle, App :

$$\frac{\mathcal{H} \vdash f : \forall \bullet .(\vec{x} : ck'_1) \xrightarrow{S} (\vec{y'} : ck'_2) \quad ck_1 \to ck_2 = \underset{(\vec{e},\vec{y'},ck)}{inst} \left(\forall \bullet .(\vec{x}:ck'_1) \xrightarrow{S} (\vec{y}:ck'_2)\right) \quad \mathcal{C} \vdash \vec{e} : ck_1 \quad \mathcal{C} \vdash \vec{y'} : ck_2}{\mathcal{H}, \mathcal{C} \vdash \vec{y'} \underset{ck}{=} f(\vec{e})} \text{ App}$$

combine les deux règles précédentes : elle stipule, pour un nœud $f$, que si le type $ck_1 \to ck_2$ est une instance de la signature de $f$, et que si les paramètres de l'application sont de type $ck_1$ et les valeurs retournées sont de type $ck_2$, alors l'application $\vec{y'} = f(\vec{e})$, dont l'horloge d'application conditionnelle est $ck$, est bien cadencée.

À titre d'exemple, la figure 5.9 représente l'arbre de dérivation de l'application conditionnelle du nœud `cpt` avec pour paramètre l'expression `1 when c`. Dans cet exemple, aucune substitution entre le nom des paramètres et le nom des arguments n'est nécessaire, compte tenu du fait que la signature du nœud `cpt` ne contient aucun support. Néanmoins, afin que le mécanisme d'application conditionnelle soit cohérent avec le typage des horloges, l'horloge de base • est substituée par l'horloge de l'expression `1 when c` (c'est-à-dire (• on $c$)) .

### 5.2.2 Extraction vers OCaml du vérificateur d'horloges

Une fois les règles de typage d'horloge de la forme normale annotée définies, l'étape préliminaire de la réalisation d'un vérificateur de types repose sur l'extraction des règles définies ci-dessus vers Coq. L'utilisation de l'extraction en Coq des règles décrites avec Ott permet alors de traduire ces dernières vers des *inductifs* Coq. Par exemple, l'extrait suivant est l'inductif correspondant à l'extraction vers Coq de la règle de cadencement associée à l'application d'un opérateur arithmétique binaire.

```
Inductive clk_exp : C → exp → clock → Prop :=
  (...)
  | Binop : forall (C:C) (e:exp) (op:operator) (e':exp) (ck:clock),
    clk_exp C e ck →
    clk_exp C e' ck →
    clk_exp C (Ebinop e op e') ck.
```

À partir des différentes règles qui régissent le bon cadencement des programmes, décrites à la section précédente, Ott produit les inductifs Coq suivants :

— *clk_exp* tel que *clk_exp C e ck* correspond au jugement $\mathcal{C} \vdash e : ck$ .

— *clk_cexp* tel que *clk_cexp C ce ck* correspond au jugement $\mathcal{C} \vdash ce : ck$ .

— *Well_clocked_eq* tel que *Well_clocked_eq H C eqn* stipule qu'une équation est bien cadencée dans l'environnement global $\mathcal{H}$ et l'environnement local $\mathcal{C}$ (i.e. $\mathcal{H}, \mathcal{C} \vdash eqn$ ).

$$\dfrac{\dfrac{\dfrac{incr \notin \varnothing}{incr \underset{\varnothing}{\sim} (1^\bullet \textbf{ when } c) \rhd \varnothing} \quad \bullet \textbf{ on } c \; = \bullet [\bullet \textbf{ on } c] \quad \dfrac{n \notin \varnothing}{n \underset{\varnothing}{\sim} x \rhd \varnothing} \quad \bullet \textbf{ on } c \; = \bullet [\bullet \textbf{ on } c]}{(\bullet \textbf{ on } c) \to (\bullet \textbf{ on } c) = \underset{(1^\bullet \textbf{ when } c, x, \bullet \textbf{ on } c)}{inst} ((incr : \bullet) \xrightarrow{\varnothing} (n : \bullet))}\,\textsc{Inst}}{\dfrac{\dfrac{\mathcal{H} \vdash cpt : (incr : \bullet) \to (n : \bullet) \quad \varnothing = carriers(\bullet) +\!\!+ carriers(\bullet)}{\mathcal{H} \vdash cpt : (incr : \bullet) \xrightarrow{\varnothing} (n : \bullet)}\,\textsc{Sign} \quad (\,\textsc{Inst}\,) \quad \dfrac{\dfrac{\dfrac{}{\mathcal{C} \vdash 1^\bullet : \bullet}\,\textsc{Const} \quad \dfrac{(x, \bullet \textbf{ on } c) \in \mathcal{C}}{\mathcal{C} \vdash c : \bullet}\,\textsc{Var}}{\mathcal{C} \vdash (1^\bullet \textbf{ when } c) : \bullet \textbf{ on } c}\,\textsc{When} \quad \dfrac{(x, \bullet \textbf{ on } c) \in \mathcal{C}}{\mathcal{C} \vdash x : \bullet \textbf{ on } c}\,\textsc{Var}}{\mathcal{H}, \mathcal{C} \vdash x \underset{\bullet \textbf{ on } c}{=} cpt(1^\bullet \textbf{ when } c)}\,\textsc{App}}$$

Figure 5.9 – Exemple de dérivation des règles Inst et App pour l'application conditionnelle

— *Well_clocked_node* tel que *Well_clocked_node* $\mathcal{H}$ $\mathcal{C}$ *node* stipule qu'un nœud *node* est bien cadencé dans l'environnement global $\mathcal{H}$ et l'environnement local $\mathcal{C}$ (i.e. $\mathcal{H}, \mathcal{C} \vdash$ *node* ).

La génération de relations inductives est cependant insuffisante pour notre cas d'utilisation : nous désirons en effet réaliser un vérificateur de type, qui doit donc être *exécutable*, afin de constituer une étape logicielle de la séquence de compilation d'un programme OCaLustre. Afin de bénéficier d'une version version calculatoire et exécutable des diverses règles de typage d'horloge, des versions fonctionnelles de ces dernières ont également été réalisées en Coq.

Par exemple, l'extrait de la fonction récursive `clockof_exp` suivant est le code Coq correspondant au calcul de l'horloge de l'application d'un opérateur binaire :

```
Fixpoint clockof_exp (C : clockenv) (e:exp) :=
  match e with
    (...)
    | Ebinop e1 op e2 ⇒
      let c1 := clockof_exp C e1 in
      let c2 := clockof_exp C e2 in
      match c1,c2 with
      | Some a, Some b ⇒ if clock_eqb a b then Some a else None
      end
    end.
```

Il est alors fondamental, pour garantir la correction de notre approche, que ces fonctions exécutables respectent la sémantique de typage définie formellement par les règles inductives définies dans cette section. La certification du respect de ces règles repose sur la preuve, en Coq, de l'équivalence entre les versions inductives et les versions fonctionnelles de ces règles. Nous avons alors prouvé, dans l'assistant de preuve Coq, les équivalence suivantes :

— À l'échelle des expressions, si, dans un environnement local $\mathcal{C}$, une expression $e$ est associée à l'horloge *ck* dans la relation inductive *clk_exp*, alors la fonction *clockof_exp*, appliquée à $e$, retourne également l'horloge *ck* :

$$\forall\ \mathcal{C}\ e\ ck,\ \text{clockof\_exp}\ \mathcal{C}\ e = \mathit{Some}\ ck \Leftrightarrow \mathit{clk\_exp}\ \mathcal{C}\ e\ ck$$

— Cette équivalence permet de déduire que, au niveau des expressions de contrôle, si, dans un environnement local $\mathcal{C}$, une expression *ce* est associée à l'horloge *ck* dans la relation inductive *clk_exp*, alors la fonction *clockof_cexp*, appliquée à *ce*, retourne également l'horloge *ck* :

$$\forall\ \mathcal{C}\ e\ ck,\ \text{clockof\_cexp}\ \mathcal{C}\ e = \mathit{Some}\ ck \Leftrightarrow \mathit{clk\_cexp}\ \mathcal{C}\ e\ ck$$

— De ces relations, nous pouvons déduire que si, pour un environnement de typage d'horloges global $\mathcal{H}$ et un environnement local $\mathcal{C}$, une équation *eqn* est bien cadencée dans la version inductive des règles, alors elle l'est également dans la version fonctionnelle :

$$\forall\ \mathcal{H}\ \mathcal{C}\ eqn,\ \text{well\_clocked\_eq}\ \mathcal{H}\ \mathcal{C}\ eqn \Leftrightarrow \mathit{Well\_clocked\_eq}\ \mathcal{H}\ \mathcal{C}\ eqn$$

— Nous pouvons finalement en déduire que si un nœud *node* est bien cadencé dans la version inductive des règles, alors il est bien cadencé dans leur version fonctionnelle :

$$\forall\ \mathcal{H}\ \mathcal{C}\ node,\ \text{well\_clocked\_node}\ \mathcal{H}\ \mathcal{C}\ node \Leftrightarrow Well\_clocked\_node\ \mathcal{H}\ \mathcal{C}\ node$$

Les sources Coq représentant l'intégralité des définitions et preuves ayant mené à ce résultat sont disponibles en ligne [⚓1].

Ces propriétés, une fois formellement prouvées, nous apportent l'assurance que nos fonctions exécutables manuellement définies respectent la sémantique de cadencement représentée par les règles inductives transmises originellement à Ott. Ainsi, toute extraction de ces fonctions vers du code exécutable respectera cette même sémantique.

Les fonctions Coq sont alors extraites vers des fonctions OCaml classiques, par exemple l'extrait précédent qui gérait le cadencement d'un opérateur binaire est converti vers cet extrait de code OCaml :

```
let rec clockof_exp c = function
(...)
| Ebinop (e1, _, e2) ->
  let c1 = clockof_exp c e1 in
  let c2 = clockof_exp c e2 in
  (match c1 with
   | Some a ->
     (match c2 with
      | Some b -> if clock_eqb a b then Some a else None
      | None -> None)
   | None -> None)
```

Le code extrait depuis Coq est par la suite branché par un code « glu » capable de convertir certains types incompatibles issus de ce mécanisme d'extraction (par exemple, Coq extrait les chaînes de caractères vers des listes de caractères, ce qui ne correspond pas au type `string` de base du langage OCaml). Finalement, le code du vérificateur d'horloge est inséré dans la chaîne de compilation, et il est vérifié, pour chaque définition de nœud, qu'elle respecte la sémantique de cadencement des programmes OCaLustre en l'appliquant à la fonction well_clocked_node. Dans le cas où cette dernière retourne la valeur **`false`**, le compilateur génère alors une erreur et la compilation du programme est arrêtée.

Le vérificateur d'horloge est activé par l'option `-check_clocks` du compilateur OCaLustre. Par exemple, la vérification du nœud `merger` (qui applique simplement ses trois paramètres *c*, *a*, et *b* à l'opérateur **`merge`**) produit l'affichage suivant :

```
$ ocamlc -ppx "ocalustre -check_clocks" tests/merger.ml
merger :: (c:base * a:(base on c) * b:(base onnot c)) -> (d:base)
Checking of merger : OK
```

Le vérificateur permet ainsi de s'assurer que le typeur d'horloges implanté dans OCaLustre, qui infère les horloges de chaque équation, déduit pour chacune d'entre elle une horloge cohérente avec le système de typage des horloges synchrones. Cette vérification est une alternative à la réalisation d'un typeur vérifié : si le typeur d'horloges d'OCaLustre fonctionne correctement, alors le vérificateur

calcule systématiquement la valeur *true*. Dans le cas de figure où le vérificateur retournerait la valeur *false*, cette propriété ne peut être attestée. Pour l'intégralité des exemples présents dans ce manuscrit, ainsi que des tests réalisés au cours de nos travaux, le vérificateur d'horloges atteste que l'inférence est correcte. À terme, cette vérification pourrait permettre d'assurer qu'aucune valeur absente n'est par erreur manipulée par le programme lors de son exécution. Cette propriété nécessiterait néanmoins de relier la sémantique opérationnelle du langage à sa sémantique statique de typage d'horloges comme nous l'avons abordé dans la section 3.2.5.

### Conclusion du chapitre

Les propriétés vérifiées dans cette section apportent des garanties sur les programmes réalisés. Ces garanties ont surtout trait au typage des données manipulées par un programme OCaLustre, qu'il soit question du typage standard des valeurs, ou bien du cadencement des différents composants synchrones du programme. Grâce à la preuve en Coq de ces propriétés, nous pouvons assurer que le typage d'un programme OCaLustre apporte la sûreté attendue par la spécification du langage. De telles garanties sont bienvenues dans un cadre embarqué critique où la sécurité des utilisateurs est en jeu. Par ailleurs, des analyses statiques supplémentaires, qui profitent de l'approche portable d'OMicroB, peuvent permettre d'assurer le bon comportement d'un programme embarqué. Nous présentons dans le chapitre suivant une analyse permettant de borner le temps d'exécution d'un programme OCaLustre. À l'instar de la démarche adoptée dans ce chapitre, nous vérifierons la méthode adoptée pour cette analyse à l'aide de Coq.

# 6 Calcul du temps d'exécution pire cas d'un programme OCaLustre

L'approche machine virtuelle adoptée dans nos travaux nous permet de *factoriser* plusieurs analyses statiques qui peuvent êtres réalisées sur les programmes indépendamment des architectures matérielles visées. En effet, en raison de la représentation des programmes exécutables sous forme de bytecode, commune à toutes les implémentations de la machine virtuelle, ces analyses peuvent être réalisées directement sur les fichiers bytecode générés, et ainsi s'abstraire du matériel sur lequel le programme s'exécute. Nous illustrons dans ce chapitre une telle analyse, qui permet d'estimer le temps d'exécution pire cas (*WCET – Worst Case Execution Time*) d'un programme OCaLustre. La méthode utilisée profite du modèle mémoire très simple des microcontrôleurs et repose alors sur la *compositionabilité* du temps d'exécution du bytecode du programme : l'analyse des coûts distincts de chaque instruction du programme permet de déduire son coût total. Cette analyse, qui peut aisément s'adapter à l'exécution d'un programme sur des microcontrôleurs de modèles ou d'architectures différents, présente ainsi l'intérêt d'être compatible avec la portabilité de l'approche machine virtuelle.

Dans la suite, nous présentons et formalisons sur un bytecode idéalisé la méthode utilisée pour mesurer le temps d'exécution pire cas d'un programme OCaLustre, et en prouvons la correction à l'aide de Coq. Cette méthode est alors appliquée au réel langage des instructions bytecode OCaml à l'aide d'un outil logiciel nommé *Bytecrawler*, dont nous décrivons le fonctionnement. Enfin, nous discutons des limitations de la compatibilité de l'analyse de WCET avec le modèle de compilation décrit précédemment, et présentons un *mode dédié* de génération de code OCaml qui rend possible, sans en changer la sémantique, l'estimation du temps d'exécution maximal de tout programme OCaLustre.

## 6.1 Validation formelle de la méthode en Coq

Dans cette section, nous décrivons et formalisons la méthode que nous adoptons pour réaliser le calcul du temps d'exécution pire cas d'un programme bytecode issu de la compilation d'un programme OCaLustre. À des fins de simplification, la formalisation ainsi que la preuve de correction de la méthode sont réalisées à partir d'un langage bytecode *idéalisé*, réduit à quelques instructions impératives. Nous conjecturons que les résultats considérés sur ce langage bytecode idéalisé sont transposables au sous-ensemble concret des instructions bytecode OCaml générées par la compilation d'un programme OCaLustre.

### 6.1.1 Définition d'un langage de bytecode idéalisé

Le langage bytecode idéalisé contient les sept instructions suivantes :

$$instr ::= \mathsf{Init}\ x\ v \mid \mathsf{Assign}\ x\ y \mid \mathsf{Add}\ x\ y \mid \mathsf{Sub}\ x\ y \mid \mathsf{Branch}\ v \mid \mathsf{Branchif}\ x\ v \mid \mathsf{Stop}$$

— L'instruction Init permet d'initialiser une variable avec une valeur, par exemple $x = 3$.
— L'instruction Assign modifie la valeur d'une variable par la valeur d'une autre variable ($x = y$) [1].
— L'instruction Add correspond à l'incrémentation de la valeur d'une variable $x$ par celle d'une variable $y$ (i.e. $x$ += $y$).
— L'instruction Sub correspond à la décrémentation de la valeur d'une variable $x$ par celle d'une variable $y$ (i.e. $x$ -= $y$).
— L'instruction Branch réalise un saut dans le code.
— L'instruction Branchif réalise un saut à condition qu'une variable donnée soit différente de 0.
— L'instruction Stop entraîne l'arrêt de l'exécution du programme.

Les valeurs manipulées par le langage sont uniquement des valeurs entières :

$$values, v, w ::= int\_literal \mid v + w \mid v - w$$

Un état $\sigma$ d'un programme correspond à un triplet contenant le code $P$ du programme (une structure qui associe à chaque adresse l'instruction du langage correspondante), un pointeur de code $pc$, et une mémoire $M$ (une structure d'association variable-valeur entières) :

$$\sigma \quad ::= \quad (P, pc, M)$$

Les règles d'évaluation en sémantique opérationnelle à petit pas de ce langage sont définies dans la figure 6.1.

**Traces d'exécution**

Dans la suite, nous considérerons la séquence de tous les états rencontrés au cours de l'exécution d'un programme écrit dans ce langage. Nous nommons cette séquence *trace d'exécution*.

**Définition 6.1.1** (Trace d'exécution)**.** *Une trace d'exécution $\mathcal{T}$ d'un programme est une suite d'états dont l'un est l'image de l'autre par une transition dans la sémantique opérationelle du langage idéalisé.*

Notre méthode porte uniquement sur les traces d'exécution *finies* d'un programme. Il ne serait en effet pas possible d'analyser des programmes dont l'exécution est infinie puisque par définition leur temps d'exécution ne serait pas bornable.

**Définition 6.1.2** (Trace d'exécution finie)**.** *Une trace d'exécution $\mathcal{T}$ est finie (noté $finite(\mathcal{T})$) si elle termine par l'instruction Stop.*

À l'exécution du programme, toutes les valeurs des variables sont connues, et la trace d'exécution est *unique*. De plus, quand l'exécution termine correctement, la dernière instruction de la trace associée est l'instruction Stop. Une telle trace est appelée *trace d'exécution déterministe*.

1. L'accès à une variable se fait en temps constant.

$\boxed{\sigma \longrightarrow \sigma'}$

$$\frac{P[pc] = \mathsf{Init}\ x\ v}{(P, pc, M) \longrightarrow (P, pc + 1, M[x := v])}$$

$$\frac{P[pc] = \mathsf{Assign}\ x\ y \quad M[y] = v}{(P, pc, M) \longrightarrow (P, pc + 1, M[x := v])}$$

$$\frac{P[pc] = \mathsf{Add}\ x\ y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \longrightarrow (P, pc + 1, M[x := v{+}w])}$$

$$\frac{P[pc] = \mathsf{Sub}\ x\ y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \longrightarrow (P, pc + 1, M[x := v{-}w])}$$

$$\frac{P[pc] = \mathsf{Branch}\ v}{(P, pc, M) \longrightarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = 0}{(P, pc, M) \longrightarrow (P, pc + 1, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] \neq 0}{(P, pc, M) \longrightarrow (P, v, M)}$$

FIGURE 6.1 – Sémantique opérationnelle du langage idéalisé

**Définition 6.1.3** (Trace d'exécution déterministe)**.** *La fonction run, appliquée à un état $\sigma$, retourne la trace d'exécution déterministe dont l'état initial est $\sigma$ :*

$\boxed{run(\sigma) = \mathcal{T}}$

$$\frac{P[pc] = \mathsf{Stop}}{run((P, pc, M)) = [(P, pc, M)]} \qquad \frac{\sigma \longrightarrow \sigma' \quad run(\sigma') = \mathcal{T}}{run(\sigma) = \sigma \ ::\ \mathcal{T}}$$

Par exemple, considérons le programme $P$ suivant :

$$[0 : \mathsf{Init}\ \text{"x"}\ 4\ ;\ 1 : \mathsf{Branchif}\ \text{"x"}\ 3\ ;\ 2 : \mathsf{Branch}\ 0\ ;\ 3 : \mathsf{Add}\ \text{"x"}\ \text{"x"};\ 4 : \mathsf{Stop}]$$

La suite d'états $\mathcal{T}$ constitue ainsi une trace d'exécution du programme $P$, avec pour état initial l'état composé de $P$, d'un pointeur de code initialisé à 0, et d'une mémoire vide (i.e. $(P, 0, \varnothing)$) :

$$\mathcal{T} = run((P, 0, \varnothing)) = [(P, 0, \varnothing)\ ;\ (P, 1, [x = 4])\ ;\ (P, 3, [x = 4])\ ;\ (P, 4, [x = 8])]$$

**Coûts**

Nous associons un *coût* à chaque instruction du langage. Ce coût correspond au temps d'exécution (en nombre de cycles) de chacune des instructions. Notre analyse dépend ainsi d'une fonction de coût $cost_{instr}$, qui associe à chaque instruction une valeur entière :

$$cost_{instr} : instr \rightarrow nat$$

Le coût $cost_{step}$ d'une transition est égal au coût de l'instruction correspondante :

$$cost_{step}(P, pc, M) = cost_{instr}(P[pc])$$

Et le coût *cost* d'une trace d'exécution correspond à la somme des valeurs retournées par la fonction de coût pour chaque instruction de cette trace :

$$cost(\mathcal{T}) = \sum_{\sigma \in \mathcal{T}} cost_{step}(\sigma)$$

Par exemple, nous représentons la fonction de coût des instructions du langage idéalisé par table suivante, qui associe à chaque instruction un coût arbitraire :

| Instruction | Coût (cycles) |
|---|---|
| Init | 4 |
| Assign | 2 |
| Add | 5 |
| Sub | 7 |
| Branch | 2 |
| Branchif | 3 |
| Stop | 1 |

Le coût de la trace d'exécution définie dans l'exemple précédent, qui correspond à la somme des coûts des instructions de cette dernière, est alors :

$$\begin{aligned} cost(\mathcal{T}) &= cost_{step}(P, 0, \varnothing) + cost_{step}(P, 1, [x = 4]) + cost_{step}(P, 3, [x = 4]) + cost_{step}(P, 4, [x = 8]); \\ &= cost_{instr}(P[0]) + cost_{instr}(P[1]) + cost_{instr}(P[3]) + cost_{instr}(P[4]) \\ &= cost_{instr}(\text{Init "x" 4}) + cost_{instr}(\text{Branchif "x" 3}) + cost_{instr}(\text{Add "x" "x"}) + cost_{instr}(\text{Stop}) \\ &= 13\ cycles \end{aligned}$$

### 6.1.2 Effacement de variables

Dans un programme réel et non trivial, il est courant que les valeurs de certaines variables ne soient pas connues avant l'exécution du programme. Ces valeurs peuvent par exemple être issues de l'utilisation d'opérateurs ayant une sémantique non-déterministe [CL14] ou, plus couramment, provenir de l'environnement d'exécution du programme (entrées utilisateur, signaux, etc.). Dans notre contexte d'application, de nombreuses valeurs manipulées par les programmes proviennent en effet de l'environnement électronique des montages pysiques qui contiennent un microcontrôleur : lorsque celui-ci réagit par exemple à la valeur d'un capteur de température, cette dernière est inconnue au moment

de la compilation du programme. Il n'est donc pas toujours possible d'évaluer statiquement et de façon déterministe le chemin d'exécution complet d'un programme donné, puisque des valeurs issues de son environnement peuvent par exemple conditionner un branchement dans le flot de contrôle du programme : on ne peut ainsi pas « prédire » quel chemin prendra le programme lors de son exécution réelle quand la valeur d'une condition est inconnue statiquement. Ce non-déterminisme, introduit par le sondage de l'environnement du programme, rend alors difficile le calcul de la durée totale d'exécution de ce dernier : selon qu'un chemin ou un autre soit emprunté lors de l'exécution réelle du programme, sa durée d'exécution peut ainsi fortement varier.

Pour être en capacité de borner statiquement le temps d'exécution d'un programme, il est alors nécessaire de raisonner sur une représentation « abstraite » de ce programme, en représentant statiquement tous les chemins d'exécutions possibles du programme. À partir de l'ensemble produit, il est possible de déduire une valeur de durée d'exécution qui majore le temps d'exécution de tous les éléments de cet ensemble.

La représentation abstraite d'un programme manipule alors des variables dont les valeurs ne sont pas connues lors de la compilation du programme. Pour ce faire, nous étendons dans la suite la grammaire des valeurs du langage idéalisé afin de supporter les valeurs de variables *inconnues*, que nous représentons avec le symbole $\top$ (« *top* ») pour signifier qu'elles peuvent correspondre à *n'importe quelle* valeur :

$$values, v, w ::= int\_literal \mid v + w \mid v - w \mid \top$$

La mémoire du programme peut ainsi associer à certaines variables des valeurs inconnues. Nous appellerons *effacement* une mémoire dont tout ou partie des variables sont associées à la valeur $\top$.

**Définition 6.1.4** (Effacement)**.** *Les règles inductives suivantes définissent l'effacement d'une mémoire :*

$\boxed{M \backslash\!\searrow M'}$

$$\frac{}{\varnothing \backslash\!\searrow \varnothing} \qquad \frac{M \backslash\!\searrow M'}{M[x := v] \backslash\!\searrow M'[x := v]} \qquad \frac{M \backslash\!\searrow M'}{M[x := v] \backslash\!\searrow M'[x := \top]}$$

Une mémoire $M'$ est ainsi un effacement d'une mémoire $M$ (que l'on note $M \backslash\!\searrow M'$) si et seulement si $M$ et $M'$ possèdent toutes deux le même ensemble de variables, mais certaines valeurs des variables qui sont connues dans $M$ sont inconnues dans $M'$.

Dans la suite, nous utiliserons cette même notation pour représenter un état $\sigma'$ dont la mémoire correspond à un effacement de la mémoire contenue dans un état $\sigma$. Si la mémoire de $\sigma'$ est un effacement de la mémoire dans $\sigma'$, on note alors $\sigma \backslash\!\searrow \sigma'$.

Puisque le coût d'une transition ne dépend que du programme et du pointeur de code, il ne change pas après effacement de la mémoire de l'état associé :

**Lemme 6.1.1.** $\forall\ \sigma\ \sigma', \sigma \backslash\!\searrow \sigma' \Rightarrow cost_{step}(\sigma) = cost_{step}(\sigma')$

Par extension, le coût d'une trace d'exécution effacée est identique au coût d'une trace non-effacée :

**Lemme 6.1.2.** $\forall\ \mathcal{T}\ \mathcal{T}',\ \mathcal{T} \backslash\!\searrow \mathcal{T}' \Rightarrow cost(\mathcal{T}) = cost(\mathcal{T}')$

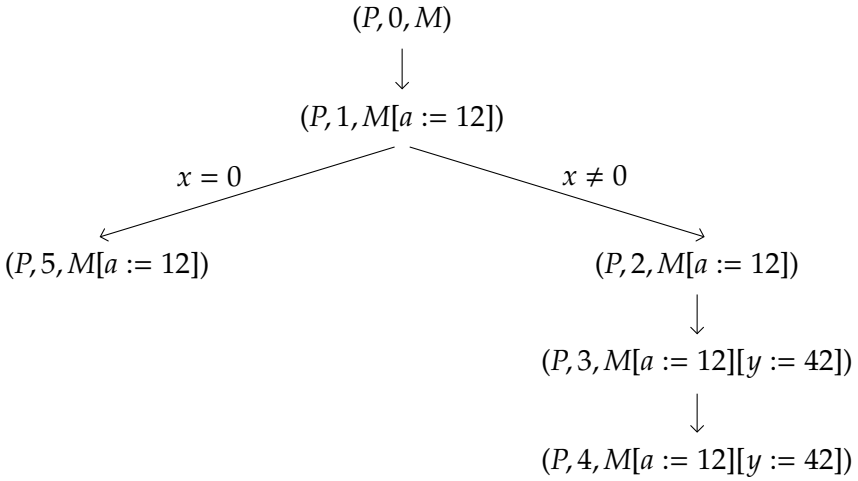
### 6.1.3 Évaluation non-déterministe

L'apparition de valeurs inconnues ($\top$) introduit un non-déterminisme à la sémantique du langage. En effet, si, pour une instruction de branchement conditionnel (Branchif x v), la valeur de la condition x est effacée, alors il n'est plus possible de savoir statiquement si le programme doit prendre le chemin correspondant au cas où x est vraie (i.e. $x \geq 1$), ou celui dans le cas où x est fausse (i.e. $x = 0$). Le coût total d'exécution d'un programme peut alors, dès lors que plusieurs chemins d'exécution sont possibles, considérablement diverger selon le chemin emprunté.

Par exemple, considérons le programme suivant :

$P$ = [Init "a" 12 ; 1 : Branchif "x" 5 ; 2 : Init "y" 42 ; 3 : Branch 4 ; 4 : Stop]

Si la valeur de la variable x n'est pas connue à la compilation du programme (i.e. $M[x] = \top$), alors il n'est pas possible de présumer de son chemin d'exécution exact lors de son exécution réelle. À partir du branchement causé par la valeur de x, deux chemins d'exécution différents sont possibles :




Le coût de $P$ est donc potentiellement différent selon le chemin réellement emprunté : si l'on conserve la même fonction de coût que dans l'exemple précédent, alors le branchement positif (celui qui "saute" directement à $pc = 5$) a un coût de 8, tandis que le branchement négatif a un coût de 14.

Pour permettre de calculer un majorant du coût d'exécution d'un programme, nous introduisons alors la notion de *coût maximum* d'une exécution. Ce coût est calculé en cumulant les coûts des divers états rencontrés en suivant les transitions de la sémantique opérationnelle du langage idéalisé. En présence d'un branchement dont la condition est inconnue, le coût maximum est calculé en suivant la transition vers la branche dont le coût est le plus grand.

**Définition 6.1.5** (Coût maximum). *La fonction $cost_{max}$ calcule le coût maximum d'un programme, à partir d'un état $\sigma$ :*

$$\boxed{cost_{max}(\sigma) = c}$$

$$\frac{P[pc] = \mathsf{Stop}}{cost_{max}((P, pc, M)) = cost_{instr}(\mathsf{Stop})} \qquad \frac{\sigma \longrightarrow \sigma' \quad cost_{step}(\sigma) = c \quad cost_{max}(\sigma') = k}{cost_{max}(\sigma) = c + k}$$

$$\frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \top \\ cost_{step}((P, pc, M)) = c \quad cost_{max}((P, pc + 1, M)) = k \quad cost_{max}((P, v, M)) = k' \end{array}}{cost_{max}((P, pc, M)) = c + max(k, k')}$$

Par abus de notation, pour des raisons de lisibilité, nous utilisons le symbole d'égalité dans les règles précédentes. Néanmoins il convient de préciser que la fonction $cost_{max}$ est partielle, et définie dans Coq sous forme d'une relation inductive. La définition de cette fonction nous permet d'énoncer le théorème suivant, qui stipule que le coût maximum d'exécution, calculé quel que soit l'effacement appliqué à la mémoire initiale d'un programme, constitue un majorant du coût d'exécution du programme réel.

**Théorème 6.1.1** (Correction). $\forall\ \sigma\ \mathcal{T}, run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') \geq cost(\mathcal{T})$

Ce théorème constitue la base du calcul du WCET de nos programmes : si la fonction de coût associe à chaque instruction son temps d'exécution, et si la mémoire initiale associe à chaque variable la valeur $\top$, alors le coût maximum correspondant est un majorant du temps d'exécution pire cas du programme (puisqu'un état dont la mémoire contient uniquement des valeurs inconnues est un effacement de tous les états initiaux possibles).

La suite de cette section est dédiée à la preuve de ce théorème, qui établit la correction de la méthode de calcul de WCET.

### 6.1.4 Preuve de correction

Les sources Coq contenant les diverses représentations formelles des notions énoncées dans cette section, ainsi que les preuves des lemmes et théorèmes associés, sont disponibles en ligne [⚓1].

#### Sémantique non-déterministe du langage idéalisé

Comme nous l'avons illustré précédemment, l'introduction de valeurs inconnues ($\top$) dans le langage idéalisé induit le non-déterminisme de son exécution : certains effacements peuvent mener à des traces d'exécutions différentes qui, bien que toutes valides, peuvent avoir des coûts qui varient fortement. Afin de représenter ce non-déterministe nous présentons une *sémantique non-déterministe* pour le langage idéalisé, capable de manipuler des valeurs *inconnues*. Celle-ci, dont les règles d'évaluation sont décrites dans la figure 6.2, correspond à la fois à la transposition directe des règles déterministes définies dans la figure 6.1[2], ainsi qu'à l'ajout de deux règles permettant de traiter le cas, pour une instruction de

2. À ceci près que les opérations d'addition et de soustraction sont étendues pour retourner la valeur $\top$ dès qu'une des opérandes est inconnue : par exemple $x + \top = \top$. Pour dénoter ce comportement, les opérateurs arithmétiques seront représentés en gras.

branchement conditionnel (Branchif) où la valeur de la condition est inconnue. Les prémisses de la règle de branchement conditionnel traitant le cas où la condition est *fausse* et de celle qui gère le cas où elle est *vraie* ne sont pas distinguables dès lors que la valeur de la condition de branchement est inconnue. L'adjonction de la notion de valeurs non-connues à la compilation, associée à l'ajout de ces deux règles apportent ainsi le caractère non-déterministe attendu à la sémantique des instructions du langage idéalisé.

$\boxed{\sigma \rightsquigarrow \sigma'}$

$$\frac{P[pc] = \mathsf{Init}\ x\ v}{(P, pc, M) \rightsquigarrow (P, pc + 1, M[x := v])}$$

$$\frac{P[pc] = \mathsf{Assign}\ x\ y \quad M[y] = v}{(P, pc, M) \rightsquigarrow (P, pc + 1, M[x := v])}$$

$$\frac{P[pc] = \mathsf{Add}\ x\ y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \rightsquigarrow (P, pc + 1, M[x := v + w])}$$

$$\frac{P[pc] = \mathsf{Sub}\ x\ y \quad M[x] = v \quad M[y] = w}{(P, pc, M) \rightsquigarrow (P, pc + 1, M[x := v - w])}$$

$$\frac{P[pc] = \mathsf{Branch}\ v}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = 0}{(P, pc, M) \rightsquigarrow (P, pc + 1, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] \neq 0}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \top}{(P, pc, M) \rightsquigarrow (P, pc + 1, M)}$$

$$\frac{P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \top}{(P, pc, M) \rightsquigarrow (P, v, M)}$$

FIGURE 6.2 – Sémantique non-déterministe du langage idéalisé

**Conservation de transitions et de traces**

Notre raisonnement porte sur le coût maximal des traces d'exécution d'un programme possédant une mémoire qui contient des variables effacées : les valeurs issues de l'environnement, inconnues en amont de l'exécution du programme. Pourtant, lors de l'exécution réelle du programme, la mémoire ne contient aucune valeur inconnue, et la trace d'exécution correspondante est la suite unique des

transitions dans la sémantique opérationnelle déterministe du langage idéalisé. Il est alors important, afin que le raisonnement appliqué aux traces « non-déterministes » du programme puisse être pertinent, de s'assurer que la trace d'exécution réelle du programme est bien contenue dans l'ensemble des traces non-déterministes considérées. En d'autres termes, raisonner sur des versions « effacées » de la mémoire ne peut être utile qu'à condition que, parmi toutes les traces effacées valides calculées, l'une de ces dernières corresponde à la trace réelle du programme (sinon, nous ne pourrions rien conclure concernant l'exécution réelle du programme).

Nous commençons d'abord par raisonner à l'échelle des transitions de la sémantique du langage. À cet effet, nous définissons d'abord un premier lemme déclarant qu'une transition dans la sémantique déterministe perdure après « plongement » dans la sémantique non-déterministe du langage idéalisé. Cette propriété est évidente, puisque la sémantique non-déterministe correspond à une extension de la sémantique déterministe du langage idéalisé :

**Lemme 6.1.3.** $\forall\ \sigma\ \sigma',\ (\sigma \longrightarrow \sigma') \Rightarrow (\sigma \rightsquigarrow \sigma')$

De plus, toute transition est conservée après effacement de la mémoire du programme : s'il existe une transition d'un état $\sigma_1$ vers un état $\sigma_2$ dans la sémantique non-déterministe, alors pour tout effacement $\sigma'_1$ de l'état d'origine il existe une transition menant à un état $\sigma'_2$, et ce dernier est un effacement de $\sigma_2$ :

**Lemme 6.1.4.** $\forall\ \sigma_1\ \sigma_2\ \sigma'_1, (\sigma_1 \rightsquigarrow \sigma_2) \wedge (\sigma_1 \backslash\!\searrow \sigma'_1) \Rightarrow (\exists\ \sigma'_2,\ (\sigma'_1 \rightsquigarrow \sigma'_2) \wedge (\sigma_2 \backslash\!\searrow \sigma'_2))$

En combinant les deux lemmes précédents, nous sommes alors en mesure de déduire que toute transition dans la sémantique déterministe est conservée après plongement dans la sémantique non-déterministe, même après effacement (partiel ou complet) de la mémoire de l'état d'origine :

**Lemme 6.1.5** (Conservation)**.** $\forall\ \sigma_1\ \sigma_2\ \sigma'_1, (\sigma_1 \longrightarrow \sigma_2) \wedge (\sigma_1 \backslash\!\searrow \sigma'_1) \Rightarrow (\exists\ \sigma'_2,\ (\sigma'_1 \rightsquigarrow \sigma'_2) \wedge (\sigma_2 \backslash\!\searrow \sigma'_2))$

Cette propriété est représentée schématiquement dans la figure 6.3.

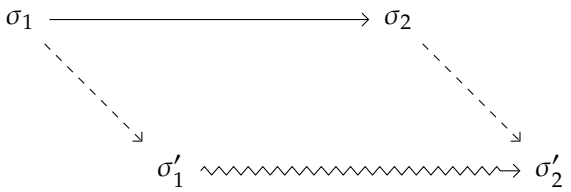



Figure 6.3 – Conservation de transition

Enfin, comme illustré par la figure 6.4, l'extension des propriétés précédentes aux traces d'exécution est directe.

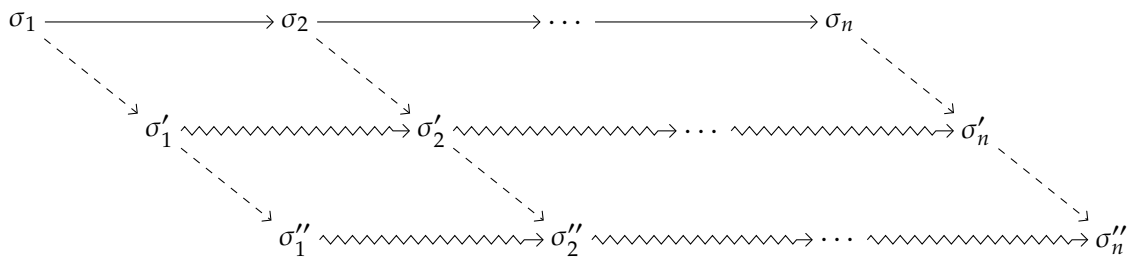



Figure 6.4 – Conservation de trace

**Majoration du coût de l'exécution réelle du programme**

Afin de vérifier que la fonction $cost_{max}$ concerne bien la plus « coûteuse » trace d'exécution du programme, nous définissons formellement une telle trace, nommée *trace d'exécution maximale*.

**Définition 6.1.6** (Trace maximale)**.** *La fonction $run_{max}$ qui calcule la trace d'exécution maximale est définie par induction de la façon suivante :*

$\boxed{run_{max}\,(\sigma) = \mathcal{T}}$

$$\frac{\sigma \longrightarrow \sigma' \quad run_{max}\,(\sigma') = \mathcal{T}}{run_{max}\,(\sigma) = (\sigma \;::\; \mathcal{T})} \qquad \frac{P[pc] = \mathsf{Stop}}{run_{max}\,((P, pc, M)) = [(P, pc, M)]}$$

$$\frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \top \\ run_{max}\,((P, pc+1, M)) = \mathcal{T} \quad run_{max}\,((P, v, M)) = \mathcal{T}' \quad cost(\mathcal{T}) > cost(\mathcal{T}') \end{array}}{run_{max}\,((P, pc, M)) = ((P, pc, M) \;::\; \mathcal{T})}$$

$$\frac{\begin{array}{l} P[pc] = \mathsf{Branchif}\ x\ v \quad M[x] = \top \\ run_{max}\,((P, pc+1, M)) = \mathcal{T} \quad run_{max}\,((P, v, M)) = \mathcal{T}' \quad cost(\mathcal{T}) \leq cost(\mathcal{T}') \end{array}}{run_{max}\,((P, pc, M)) = ((P, pc, M) \;::\; \mathcal{T}')}$$

De la même façon que $cost_{max}$, la fonction $run_{max}$ est une fonction partielle définie par un inductif dans Coq.

La trace maximale contient les mêmes états que ceux considérés par la fonction $cost_{max}$ : elle parcourt les chemins pour lesquels la somme totale des coûts est la plus importante. Son coût est donc bien identique au coût maximum d'exécution :

**Lemme 6.1.6.** $\forall\ \sigma,\ cost_{max}(\sigma) = cost(run_{max}(\sigma))$

Par définition, la trace maximale est une trace finie. De plus, le coût de celle-ci est supérieur à celui de toute autre trace finie qui commence par le même état :

**Lemme 6.1.7.** $\forall\ \sigma\ \mathcal{T}\ k,\ finite(\sigma :: \mathcal{T}) \Rightarrow cost_{max}(\sigma) = k \Rightarrow k \geq cost(\sigma :: \mathcal{T}))$

En appliquant le lemme de conservation aux traces d'exécution, nous dérivons alors la propriété principale de la preuve de correction de notre méthode : le coût de la trace maximale d'exécution est plus grand que le coût de toute trace finie qui commence par le même état, même lorsque l'on considère un effacement de sa mémoire.

**Lemme 6.1.8.** $\forall\ \sigma\ \mathcal{T}\ k, finite(\sigma :: \mathcal{T}) \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') = k \Rightarrow k \geq cost(\sigma :: \mathcal{T})$

Puisque la trace déterministe est finie, la combinaison des précédents lemmes nous permet enfin de conclure avec le théorème de correction qui stipule que le coût maximal d'un programme, estimé à partir d'un effacement de la mémoire initiale, est un majorant du coût réel d'exécution du programme :

**Théorème 6.1.2** (Correction)**.** $\forall\ \sigma\ \mathcal{T}\ k, run(\sigma) = \mathcal{T} \Rightarrow \forall\ \sigma',\ \sigma \searrow \sigma' \Rightarrow cost_{max}(\sigma') = k \Rightarrow k \geq cost(\mathcal{T})$

## 6.2 Application au calcul du WCET d'un programme OCaLustre : l'outil *Bytecrawler*

La méthode de calcul du temps d'exécution pire cas décrite dans la section précédente s'applique à un langage idéalisé limité à quelques instructions, mais qui permet tout de même, de par les instructions de branchement dont il dispose, de faire boucler un programme lors de son exécution. L'absence de boucle non statiquement bornée dans le programme est pourtant une condition essentielle pour assurer la finitude des traces considérées : si tel n'était pas le cas, certains effacements de la mémoire [3] pourraient conduire à l'exécution infinie de l'analyseur statique (qui serait alors occupé à calculer indéfiniment la trace d'exécution de coût maximal du programme). L'analyse décrite dans la section précédente n'était ainsi adaptée qu'aux traces *finies*, avec donc la condition que le programme ne boucle pas indéfiniment. Notre analyse n'est alors adaptée qu'à condition que nous puissions assurer qu'aucune boucle ou appel récursif non-bornés statiquement ne soient possibles dans les programmes considérés.

Lors de la compilation du langage OCaLustre, chaque nœud est converti vers une fonction OCaml non-récursive et, bien que le programme principal soit exécuté dans une boucle principale dont un tour est réalisé à chaque instant synchrone, aucune boucle « imbriquée » n'est réalisée pendant un instant : le programme calcule simplement, *instantanément*, des valeurs en sortie à partir de valeurs en entrée. Les traces d'exécutions d'un instant synchrone sont alors *finies* et, dans ces conditions, il est alors possible d'adapter au langage OCaLustre les analyses formalisées dans la section précédente, en les appliquant non plus au langage idéalisé, mais au langage des instructions de la machine virtuelle OCaml, puisque c'est celui vers lequel est compilé (après une première conversion vers un code OCaml standard) le programme OCaLustre.

Bien que plus nombreuses et potentiellement plus puissantes, les instructions de la machine virtuelle OCaml générées par le langage OCaLustre sont comparables à celles du langage idéalisé : de façon analogue, certaines d'entre elles réalisent des opérations de changement de valeurs de références (`SETFIELD`), des calcul arithmétiques (`ADDINT`), ou encore des branchements conditionnels (`BRANCHIF`). Les analyses réalisées dans le langage idéalisé sont donc transposables aux instructions de la machine virtuelle OCaml, et le calcul du temps d'exécution pire cas d'un instant synchrone d'un programme OCaLustre consiste alors, à l'image de l'analyse du langage idéalisé, à calculer le coût maximal d'exécution de l'instant synchrone. Ce coût constitue alors le temps d'exécution pire cas d'un instant du programme.

Il est à noter que l'analyse ici présentée est valide grâce au fait que les microcontrôleurs que nous considérons possèdent des modèles mémoires très simples. En effet, l'absence de systèmes de caches rend predictible la durée de chaque instruction [BN94], sans avoir à considérer des scénarios où les accès mémoire seraient plus ou moins coûteux en temps, et assure les temps d'exécution, en nombre de cycles, des différents instructions du langage. En ce sens, les cibles considérées sont dites exemptes d'anomalies temporelles (« *timing anomalies* » [RWT+06]) et induisent la compositionabilité des analyses de temps d'exécution.

### 6.2.1 Calcul du coût des instructions

Lors de la formalisation de l'analyse du temps d'exécution pire cas, nous avions abordé le fait qu'il existait une fonction de coût spécialisée, associant à chaque instruction du langage idéalisé une valeur

3. Par exemple, l'effacement de la variable conditionnant l'exécution d'un tour de boucle.

entière. Dans le contexte d'exécution d'OCaLustre, il est donc nécessaire de se munir d'une telle fonction permettant d'associer à chaque instruction de la machine virtuelle un temps d'exécution maximal, afin de pouvoir déduire une valeur majorant le temps d'exécution du programme complet. À cet effet, nous tirons profit d'analyseurs statiques pré-existants, permettant d'attribuer un nombre de cycles maximum à un programme compilé. À des fins d'illustration, nous avons utilisé l'outil logiciel *Bound-T* [HS02], un analyseur d'occupation mémoire et de temps d'exécution, capable de calculer le WCET d'un programme compilé pour un microcontrôleur AVR.

Notre démarche a consisté à faire calculer par Bound-T le temps d'exécution pire cas de chaque bloc de code responsable de l'interprétation d'une instruction de la machine virtuelle de OMicroB. De cette façon Bound-T nous permet d'associer à chaque instruction de la machine virtuelle une valeur correspondant au nombre de cycles machines maximum pour l'exécution de cette instruction. Il est à noter que le temps d'exécution de certaines instructions de la machine virtuelle OCaml dépend en partie de la valeur d'un de ses paramètres (par exemple, l'instruction `APPTERM n v` entraîne la réalisation de `n` tours d'une boucle dans le code associé de l'interprète). Une façon de traiter ces instructions est de calculer leur coût quand leur paramètre est égal à la valeur la plus grande possible. Cependant cette méthode entraîne l'estimation d'un majorant potentiellement loin de la réalité. Dans le but d'associer plus finement un coût à chaque instruction, nous recalculons plusieurs fois les coûts de telles instructions en fournissant une nouvelle valeur « en dur » en tant que paramètre de ces instructions. Cette méthode est plus lente à réaliser, mais elle n'est à faire qu'une fois : dès que Bound-T a été exécuté pour chaque instruction de la machine virtuelle, l'outil n'est plus à réutiliser, car chaque bytecode d'un programme OCaml contiendra un sous-ensemble de ce jeu d'instructions dont nous avons pré-calculé les coûts.

### 6.2.2 *Bytecrawler* : un interprète « abstrait » de bytecode

Une fois l'analyse de Bound-T terminée, nous avons à notre disposition une table complète représentant les instructions de la machine virtuelle et leur coût (en nombre de cycles). Un extrait de cette table, générée pour le microcontrôleur AVR ATmega2560 pour une version 16 bits d'OMicroB, est reproduit dans la figure 6.5. Cette table constitue la fonction de coût décrite dans la section précédente. Il est alors possible de répliquer la méthode de calcul de trace de coût maximum, en considérant que les valeurs absentes ($\top$) correspondent aux résultats des appels aux primitives C (via les instructions `CCALL`) qui permettent de communiquer avec l'environnement.

Nous proposons alors un outil d'analyse statique, nommé *Bytecrawler*, qui exécute le programme en suivant le même fonctionnement que la fonction $cost_{max}$ décrite plus haut, par exemple en parcourant les deux branches d'une conditionnelle afin d'en déduire celle de coût maximum. Bytecrawler permet ainsi de déduire, en accumulant les valeurs issues de la fonction de coût calculée par Bound-T, le coût (en nombre de cycles) de la trace d'exécution maximale d'un instant OCaLustre. En étendant le résultat du théorème de correction au langage des instructions de la machine virtuelle OCaml, nous pouvons en déduire que le coût ainsi trouvé correspond à une estimation du temps d'exécution pire cas de l'instant synchrone du programme OCaLustre. La fonction `wcet` dans la figure 6.6 est un extrait simplifié de la fonction au cœur du fonctionnement de Bytecrawler : elle évalue un programme en suivant la sémantique standard des instructions bytecode OCaml, tout en étant capable de gérer les valeurs *inconnues* de variables, issues d'appels à des primitives C d'entrée/sortie.

| Instruction bytecode | Coût (en nombre de cycles) |
|---|---|
| ACC | 46 |
| PUSH | 43 |
| PUSHACC | 74 |
| POP | 42 |
| ASSIGN | 52 |
| ENVACC | 50 |
| PUSHENVACC | 78 |
| APPLY | 50 |
| RETURN | 85 |
| GETFIELD | 50 |
| SETFIELD | 67 |
| GETVECTITEM | 54 |
| SETVECTITEM | 69 |
| BRANCHIF_1B | 36 |
| BRANCHIF_2B | 46 |
| BRANCHIF_4B | 67 |
| CONST0 | 25 |
| ADDINT | 51 |
| SUBINT | 51 |
| MULINT | 59 |

FIGURE 6.5 – Table de coûts des instructions bytecode calculée par Bound-T (extrait)

```
let rec wcet state =
  let state' = {state with pc = state.pc + 1} in
  let instr = state.instrs.(pc) in
  cost instr + match instr with
  | CONST i -> wcet {state' with accu = Int i}
  | BRANCH ptr -> wcet {state with pc = ptr}
  | BRANCHIF ptr ->
    (match state.accu with
    | Int 0 -> wcet state'
    | Int _ -> wcet {state with pc = ptr}
    | Unknown -> max (wcet {state with pc = ptr}) (wcet state'))
  | ADDINT ->
    (match state.accu, state.stack with
    | Int x, (Int y)::s -> wcet {state' with accu = Int (x+y); stack = s}
    | _, _::s -> wcet {state' with accu = Unknown ; stack = s})
  | C_CALL1 _ -> wcet {state' with accu = Unknown}
  | STOP -> 0
```

FIGURE 6.6 – Fonction de calcul du WCET (extrait)

Par ailleurs, en plus d'associer un nombre de cycles à chaque instruction de la machine virtuelle OCaml, il est aussi nécessaire de calculer le coût de toutes les primitives d'entrées/sorties intégrées à la bibliothèque standard de la machine virtuelle. Ces dernières, écrites en C, sont généralement compatibles avec des analyseurs comme Bound-T. De la même façon que le jeu d'instructions ne change pas au fil du temps, les primitives d'entrées/sorties, étant de très bas niveau (elles consistent souvent à simplement écrire ou lire les valeurs des broches du microcontrôleurs) ne sont pas censées changer de programme en programme. L'outil Bytecrawler doit alors parcourir, dès qu'il rencontre une instruction `CCALL` d'appel à une primitive, une seconde table qui associe à chaque nom de primitive C son temps d'exécution estimé. Le développeur désirant définir ses propres primitives C devra alors fournir, dans cette table, leurs coûts associés.

L'approche utilisée par Bytecrawler offre un avantage important par rapport aux outils classiques de calcul de WCET destinés à des programmes compilés vers du code natif : puisque les coûts associés à chaque instruction de la machine virtuelle sont fixes, ils peuvent ainsi être réutilisés pour analyser le coût d'autres programmes, sans contraindre le programmeur à relancer la machinerie parfois complexe d'un analyseur comme Bound-T au moindre changement de son programme. De surcroît, le calcul du WCET d'un programme peut être réalisé pour plusieurs modèles de microcontrôleurs différents dès lors qu'il est fourni à Bytecrawler une table de coûts des instructions bytecode adapté à chacun de ces modèles. Une telle factorisation des analyses est la bienvenue dans ce contexte de développement destiné à du matériel embarqué, dont les cibles (montage, type du microcontrôleur, . . . ) peuvent changer avec l'évolution des besoins.

### 6.2.3 Mode dédié du compilateur OCaLustre

Puisque la machine virtuelle utilisée fait usage d'un *garbage collector*, le calcul de ces coûts pourrait être erroné. En effet, lors de l'allocation d'une nouvelle valeur dans le tas, l'algorithme de récupération de la mémoire peut potentiellement être déclenché, et plusieurs cycles machine peuvent ainsi être occupés pour qu'il complète son exécution. Ce déclenchement difficilement prévisible du garbage-collector peut entraîner, s'il n'est pas pris en compte, des erreurs dans le calcul du WCET d'un programme. Plusieurs méthodes peuvent être réalisées afin de prendre en compte l'existence d'un tel mécanisme de récupération automatique de la mémoire : l'une d'entre elles consiste à simplement considérer qu'un nettoyage complet de la mémoire est exécuté à chaque allocation d'une valeur dans le tas. Cette sur-estimation permet de calculer correctement un majorant du temps d'exécution d'un instant synchrone mais ce dernier manque de « finesse », de par le fait qu'il n'est pas réaliste de considérer que le *garbage-collector* se déclenche aussi souvent.

Il est alors intéressant de remarquer que, dans le contexte de la compilation du langage OCaLustre, les valeurs nécessitant une allocation dans le tas (typiquement, les registres issus de l'utilisation de l'opérateur ⋙ ) sont déclarées lors de la phase d'initialisation du nœud : ces valeurs représentent l'environnement de la fermeture générée par la compilation d'un nœud. Par exemple, dans le nœud suivant, seule la valeur du registre permettant d'enregistrer la valeur précédente de l'expression `cpt + 1` nécessite d'être conservée en mémoire :

```
let%node count () ~return:cpt =
  cpt = (0 ⋙ (cpt + 1))
```

Et cette particularité transparait après compilation : la variable `cpt_fby` est la seule à être présente dans l'environnement de la fermeture retournée par la fonction `count` :

```
let count () =
  let cpt_fby = ref 0 in
    fun () ->
      let cpt = !cpt_fby in
      cpt_fby := cpt + 1;
      cpt
```

De ce fait, il est possible de considérer que, dans la condition où ne sont utilisées que des valeurs de type de base (tels que les entiers, les flottants [4], ou les booléens), la quantité de valeurs à allouer en mémoire pour l'exécution d'un nœud est connue en amont de l'exécution du programme, et ainsi que si les registres nécessaires sont alloués avant l'exécution du programme OCaLustre, aucune nouvelle allocation ne sera réalisée pendant l'exécution de l'instant. Cela permettrait d'affirmer qu'il n'est pas utile de prendre en compte le temps pris par le garbage-collector dans le calcul du temps d'exécution pire cas de l'instant synchrone, puisqu'aucune allocation dans le tas n'est réalisée dans cet instant.

Néanmoins, la méthode de compilation décrite dans la section 4.4 n'est pas entièrement compatible avec cette affirmation : le code compilé d'un nœud peut en effet contenir des n-uplets correspondant à ses paramètres ainsi qu'aux flots de sortie. Ces n-uplets sont par conséquent alloués dans le tas à chaque instant, et la question du déclenchement de l'algorithme de garbage-collection se pose à nouveau, dès lors qu'un nœud retourne et/ou reçoit plus d'un flot.

Par exemple, considérons le nœud suivant :

```
let%node triple (x) ~return:(a,b,c) =
  a = x;
  b = x + 1;
  c = 42
```

Celui-ci entraîne, lors de sa compilation, la génération d'une fonction OCaml qui produit une fermeture, et cette dernière induit la création d'un triplet `(a,b,c)` dont la représentation mémoire conduit à une allocation dans le tas à chaque instant synchrone :

```
let triple () =
  fun x ->
    let a = x in
    let b = x + 1 in
    let c = 42 in
    (a,b,c)
```

4. Grâce à notre représentation des valeurs dans OMicroB, les flottants ne nécessitent pas en effet d'être alloués sur le tas

Ainsi, l'analyse présentée dans ce chapitre n'est compatible avec le modèle de compilation présenté dans la section 4.4 qu'à condition que les nœuds considérés n'aient pas plusieurs paramètres en entrée ou en sortie. Cette limitation forte nous paraît trop importante pour rendre Bytecrawler réellement utilisable sur des programmes non triviaux. Par conséquent, afin d'éviter de telles allocations se produisant « en cours d'instant », nous présentons un schéma de compilation alternatif pour les nœuds OCaLustre. Cette nouvelle méthode de compilation conserve la sémantique du langage OCaLustre, mais nous permet d'assurer cette fois qu'aucune allocation mémoire n'est réalisée lors d'un instant synchrone, quelle que soit la forme des nœuds.

Dans ce schéma alternatif de compilation, tout nœud *n* entraîne la définition d'un type OCaml *n*`_state` correspondant à l'état d'une instance de ce nœud. Cet état contient des registres mutables correspondant aux sorties du nœud et aux registres internes nécessaires à son exécution (tels que ceux issus de l'utilisation de l'opérateur *followed-by*) :

```
type ('a, 'b, 'c) triple_state = {
 mutable triple_out_a: 'a ;
 mutable triple_out_b: 'b ;
 mutable triple_out_c: 'c }
```

Le code du nœud est alors distribué en deux fonctions distinctes. Une première fonction d'initialisation correspond à l'allocation de l'état du nœud. Comme à cette étape de compilation du programme les types des valeurs n'ont pas été encore inférés, il est à nouveau fait usage de la valeur `Obj.magic ()`, sauf pour le flots constants (initialisés avec leur valeur), ou les équations de la forme `k` ⋙ `e` (qui sont initialisées avec la constante `k`). Le rôle de `Obj.magic ()` n'est ici que de permettre de réserver sur le tas un espace nécessaire pour stocker l'état mutable du nœud : il ne sera jamais évalué pendant l'exécution du programme synchrone.

```
let triple_alloc x = {
    triple_out_a = Obj.magic ();
    triple_out_b = Obj.magic ();
    triple_out_c = 42 }
```

Cette fonction génère l'état initial des registres du nœud. Suivant un modèle d'exécution par passage d'état, ou *state-passing style*, l'état interne du programme est alors un paramètre de la fonction de *pas* du nœud, qui calcule à chaque instant les valeurs des flots définis par le nœud :

```
let triple_step state x =
  let a = x in
  let b = x + 1 in
  let c = 42 in
  state.triple_out_a ← a;
  state.triple_out_b ← b;
  state.triple_out_c ← c
```

Cette fonction se charge, à chaque instant, de mettre à jour (par effets de bord) les valeurs des différentes variables présentes dans l'état du nœud. Puisque l'état du nœud est un type enregistrement dont les champs sont mutables, chacun de ces champs est alloué lors de la génération de l'état, et aucune nouvelle allocation mémoire ne se produit pour les instants suivants. Grâce à ce modèle d'exécution qui génère du code aux aspects plus « impératifs », le calcul du WCET de l'instant peut ainsi s'appliquer sur la fonction *n*_step générée.

Le code de la fonction principale du programme OCaml généré, compatible avec ce mode compilation, aura la forme suivante :

```
let () =
  (* génération de l'état du noeud principal *)
  let _st = triple_alloc () in
  while true do
   (* lecture des entrées *)
   let x = input_triple_x () in
   (* exécution du noeud *)
   triple_step _st x;
   (* écriture des sorties *)
   output_triple _st.triple_out_a _st.triple_out_b _st.triple_out_c
  done
```

L'option -na du compilateur OCaLustre permet d'activer le mode de compilation « non-allouant » décrit dans cette section.

### 6.2.4 Exemple d'application

Dans cette section, nous illustrons le fonctionnement de *bytecrawler* sur un court exemple. Le programme OCaLustre de cet exemple est réduit à un seul nœud nommé count qui représente un compteur : il calcule à chaque instant synchrone le successeur de l'entier calculé à l'instant précédent. De plus, ce compteur peut être remis à zéro dès que le paramètre reset du nœud est vrai.

Le code de ce nœud (à gauche), ainsi que l'ensemble des instructions bytecode vers lesquelles ce nœud est traduit suite à la compilation d'OCaLustre vers un programme OCaml puis la compilation standard de ce dernier (à droite [5]) sont présentés ci-après :

5. L'option *-dinstr* du compilateur standard permet l'affichage de ce bytecode *labellisé*.

```
let%node count reset ~return:cpt =
  cpt = (0 ⋙ if reset then 0 else (cpt + 1))
```

```
L1:  GRAB 1
     ACC 0
     GETFIELD 0
     PUSH
     ACC 2
     BRANCHIFNOT L7
     CONST 0
     BRANCH L6
L7:  ACC 0
     OFFSETINT 1
L6:  PUSH
     ACC 1
     PUSH
     ACC 3
     SETFIELD 1
     ACC 0
     PUSH
     ACC 3
     SETFIELD 0
     CONST 0
     RETURN 4
```

Nous déclarons désormais deux fonctions OCaml responsables de la captation des entrées et du traitement des sorties de ce programme synchrone. La première définit la valeur en entrée du nœud (`reset`) comme le résultat du test permettant de vérifier si la broche numéro 0 du port *B* est alimentée, par exemple si un bouton-poussoir branché à cette broche a été pressé :

```
(* fonction d'entrée *)
let input_count_reset () =
    read_bit PORTB PB0
```

```
L4:  CONST0
     PUSH
     CONST1
     CCALL caml_avr_read_bit, 2
     RETURN1
```

Puisque la valeur retournée par la lecture de cette broche (via l'appel à la primitive `caml_avr_read_bit`) ne peut pas être connue au moment de la compilation, elle sera représentée dans Bytecrawler sous la forme d'une valeur inconnue ($\top$).

Par souci de simplicité, nous déclarons que la deuxième fonction censée traiter la valeur `cpt` calculée par l'instant synchrone ne fait « rien », c'est-à-dire qu'elle calcule simplement la valeur « unit » :

```
(* fonction de sortie *)
let output_count cpt = ()
```

```
L3:  CONST 0
     RETURN 1
```

Après initialisation de l'état du programme synchrone, selon le modèle de compilation décrit dans la section précédente, le programme OCaml se charge (en boucle) de lire l'entrée du programme, d'appeler la fonction de pas de l'instant synchrone, et d'appeler la fonction de traitement de ses sorties. Afin que Bytecrawler puisse reconnaître dans le bytecode où commence et où finit chaque instant synchrone, la boucle du programme est remplacée pour l'analyse de WCET par des appels aux primitives `begin_loop` et `end_loop` :

```
let () =
  let _st = count_alloc () in
  begin_loop ();
    let reset = input_count_reset () in
    count_step _st reset;
    output_count _st.count_out_cpt;
  end_loop ()
```

Le bytecode associé au code encadré par les appels à `begin_loop` et `end_loop` est alors le suivant :

```
CCALL begin_loop, 1
CONST 0
PUSH
ACC 5
APPLY 1
PUSH
ACC 0
PUSH
ACC 2
PUSH
ACC 4
APPLY 2
ACC 1
GETFIELD 1
PUSH
ACC 5
APPLY 1
CONST 0
CCALL end_loop, 1
```

Bytecrawler exécute le programme depuis la toute première instruction bytecode, mais ne commence à comptabiliser les coûts des instructions rencontrées que dès l'appel à la primitive `begin_loop`, et jusqu'à l'appel à `end_loop`, permettant ainsi de calculer le coût (en cycles) de l'instant synchrone. Ainsi, avec une table de coûts d'instructions calculée par Bound-T pour un ATmega2560, le nombre de cycles maximal estimé par Bytecrawler est de 3080. Puisqu'un tel microcontrôleur a une fréquence d'horloge de 16 MHz, alors la durée d'exécution pire cas d'*un* instant synchrone de ce programme est de $1,92 \times 10^{-4}$ secondes (192 $\mu s$). Ainsi, si l'intervalle entre deux changements de l'état de la broche *PB0* est supérieur à 192 microsecondes, l'hypothèse synchrone est valide.

## Conclusion du chapitre

Dans ce chapitre, nous avons tiré profit de la factorisation des analyses apportée par la représentation des programmes sous forme de bytecode afin de calculer le temps d'exécution pire cas d'un programme. La preuve de la correction de la méthode utilisée permet d'assurer sa viabilité, et un outil logiciel suivant cette méthode a été implémenté. Bien sûr, un des processus de certification du fonctionnement de cet outil imposerait de formaliser et prouver la méthode non plus sur le bytecode idéalisé présenté dans cette section, mais sur le vrai sous-ensemble du langage d'instructions bytecode OCaml issu de la compilation d'OCaLustre. Bien que constituant un travail conséquent en raison du nombre d'instructions du bytecode OCaml, cette transposition semble néanmoins plutôt directe.

Par ailleurs, l'intérêt principal de la méthode concerne, à nouveau, sa portabilité : les analyses portant sur le bytecode et celles inhérentes à la plateforme utilisée sont distinctes [HK07]. Le développeur d'applications n'a ainsi pas besoin de faire usage d'annotations particulières pour réaliser le calcul du WCET d'un programme OCaLustre. Ces annotations (ayant par exemple trait au nombre de tours d'une boucle) peuvent toutefois être utilisées par le développeur de la plateforme, afin de fournir une table de coûts des instructions et des primitives. Une fois créée, cette table suffit alors à l'analyse du bytecode de tout programme OCaLustre. La portabilité de la méthode a pour conséquence la portabilité de l'outil Bytecrawler : pour un même programme, le simple fait de fournir une table de coûts mesurée pour un autre microcontrôleur permet de déduire le WCET du programme sur cette plateforme, sans même avoir à le recompiler.

La méthode présentée dans ce chapitre pourrait certainement bénéficier d'analyses statiques supplémentaires, grâce par exemple à l'exécution symbolique [BFL18], ou concolique [Sen07] (une approche hybride mêlant, d'une façon plutôt proche de notre méthode, des valeurs concrètes et des valeurs symboliques), qui permettrait de ne pas considérer les chemins d'exécution inatteignables. Toutefois, cette méthode constitue avant tout une illustration d'une analyse tirant profit d'une représentation commune (le bytecode), plutôt qu'une technique d'estimation la plus fine possible du temps d'exécution maximal d'un programme.

Enfin, il est intéressant de remarquer que la méthode présentée dans ce chapitre semble pouvoir être facilement étendue pour la réalisation d'autres mesures que celle de temps d'exécution : en changeant seulement la définition de la fonction de coût (et donc la table qui la représente), il serait possible d'évaluer d'autres critères, et d'estimer par exemple la taille maximale de la pile d'exécution d'un programme synchrone, ou même la quantité maximale de valeurs allouées dans le tas pendant l'exécution d'un programme. Ces considérations relatives à l'empreinte mémoire des programmes seront déterminantes dans le chapitre suivant, au sein duquel nous détaillerons différentes mesures permettant de juger des performances des principales solutions présentées dans cette thèse.

# 7 Performances d'OMicroB et OCaLustre

Ce chapitre est destiné à la présentation et l'analyse de plusieurs mesures qui visent à constater la puissance des solutions logicielles décrites dans ce document. Nous y présentons en premier lieu un ensemble de mesures réalisées sur la machine virtuelle OMicroB à partir de programmes destinés à tester les capacités de cette dernière à l'aune des divers aspects du langage OCaml. Puis, nous traiterons des performances d'OCaLustre en particulier via l'analyse de la consommation en ressources mémoire d'un programme OCaLustre complet, que nous comparerons avec un programme équivalent réalisé dans le langage Lucid Synchrone. Tout au long du chapitre, nous discuterons en détail des résultats de ces mesures, et nous nous attacherons à les mettre en perspective relativement aux performances de solutions préexistantes.

## 7.1 Mesures de performances d'OMicroB

Dans cette section, nous réalisons plusieurs mesures permettant d'évaluer les performances de la machine virtuelle OMicroB. Ces mesures, réalisées à partir de programmes de test simples, concernent la vitesse et la consommation mémoire de la machine virtuelle.

### 7.1.1 Méthodologie

OMicroB est une machine virtuelle hautement configurable. En effet, l'utilisateur dispose de la possibilité de spécifier plusieurs aspects concrets de la machinerie permettant d'exécuter un bytecode OCaml. Cette configurabilité permet d'adapter précisément les capacités de la machine virtuelle au matériel sur lequel elle est exécutée. Le réglage des divers aspects de la machine virtuelle est réalisé à l'aide d'options de compilation particulières, utilisées en supplément de la commande permettant de compiler un programme OCaml avec OMicroB :

```
omicrob <options> <fichier.ml>
```

Dans ce chapitre, nous ferons ainsi usage des options de compilations suivantes :

— L'option `-arch <n>` permet de choisir la longueur des mots OCaml dans la représentation utilisée par OMicroB. Cette longueur peut être de 16, 32, ou 64 bits. L'utilisation d'une représentation sur 16 bits, suffisante dans une grande majorité des programmes, permet de réduire l'utilisation de la mémoire vive.

— L'option `-gc <algorithme>` permet à l'utilisateur de choisir l'algorithme de ramasse-miettes utilisé. Ce dernier peut être l'algorithme *Stop and Copy* (`SC`) abordé au chapitre 2, ou bien un algorithme de type *Mark and Compact* (`MC`).

— L'option `-no-shortcut-initialization` sert à désactiver l'optimisation qui réalise l'évaluation partielle du programme OCaml jusqu'à sa première primitive d'entrée/sortie.

— L'option `-stack-size <n>` permet de régler la taille (en mots) de la pile utilisée par la machine virtuelle pour exécuter le bytecode du programme OCaml. Par défaut, cette taille est réglée à 64 mots, mais certains programmes peuvent nécessiter une pile plus conséquente. La taille de la pile est un facteur de la consommation en mémoire vive de la machine virtuelle.
— L'option `-heap-size <n>` permet de régler le nombre (en mots) de valeurs adressables dans le tas. Par défaut, ce nombre est de 256 mots. Tout comme celle de la pile, la taille du tas a des conséquences directes sur la consommation en mémoire vive du programme OCaml. L'algorithme de *garbage collection* choisi influence l'utilisation réelle de la RAM pour le tas : en effet, de par son utilisation de deux « bassins », l'algorithme *Stop and Copy* nécessite la réservation d'une section de mémoire dont la taille est doublée par rapport à celle d'un *garbage collector* implantant un algorithme *Mark and Compact*.
— Enfin, l'option `-no-clean-interpreter` permet de désactiver l'optimisation qui produit une machine virtuelle qui embarque uniquement le code de traitement des instructions bytecode utilisées par le programme.

En raison de leur utilisation courante dans des projets amateurs ou professionnels, nous utiliserons tout au long de cette section une version d'OMicroB destinée à être exécutée sur des microcontrôleurs de la famille AVR. Ces microcontrôleurs, souvent embarqués dans les cartes *Arduino*, ont des capacités mémoire plutôt faibles (notamment quelques kilo-octets de RAM), qui permettent de rendre compte de l'importance des optimisations d'OMicroB, et démontrent la possibilité d'exécuter des programmes OCaml sur du matériel à (très) faibles ressources. Nos diverses mesures seront en particulier réalisées sur un microcontrôleur ATmega2560 doté d'une puissance de calcul de 16 MIPS, de 256 kilo-octets de mémoire flash, et 8 kilo-octets de RAM. L'exécution de la commande `omicrob <fichier>.ml` génère alors deux fichiers notables : un premier fichier `<fichier>.avr` correspondant au programme exécutable destiné à être transmis au microcontrôleur, et un second fichier `<fichier>.elf`, résultat de la compilation avec le compilateur *gcc* natif. Ce dernier permet de simuler sur PC l'exécution des programmes réalisés.

Les deux principaux aspects mesurés dans ce chapitre sont les suivants :

— Des critères de taille : en particulier, la taille des programmes exécutables produits ainsi que leur consommation de mémoire vive sont des informations importantes dans ce cadre applicatif où les ressources mémoires du matériel sont fortement limitées. Nous mesurerons à ce propos la consommation mémoire des programmes AVR produits grâce à la commande `avr-size`, variante de la commande UNIX `size` qui permet d'afficher la taille des différentes sections mémoire utilisées par le programme. En particulier, la commande `avr-size` fournit l'option supplémentaire de formattage `-C` qui regroupe les sections mémoire en deux sections : la quantité de mémoire flash utilisée pour le programme, et la quantité de mémoire RAM utilisée par le programme.
Afin d'illustrer notre propos, la figure 7.1 représente la sortie de la commande avr-size appliquée à un programme `prog.avr` [1]. Celle-ci informe l'utilisateur que ce programme utilise 6968 octets de mémoire flash (2.7% de la mémoire flash disponible sur un ATmega2560) et 7104 octets de RAM (86.7% de la RAM disponible sur ce microcontrôleur).
— Des critères de vitesse qui permettent d'illustrer les performances temporelles de la machine virtuelle. Les calculs de vitesse seront ici réalisés à la fois sur les programmes *.elf* correspondant à la simulation d'OMicroB sur PC (dans le but d'évaluer les capacités d'OMicroB par rapport à l'implantation de la machine virtuelle standard), mais aussi par l'intermédiaire des mesures

1. L'option `--mcu` permet de renseigner le modèle de microcontrôleur.

```
$ avr-size -C prog.avr --mcu=atmega2560
AVR Memory Usage
----------------
Device: atmega2560

Program:    6968 bytes (2.7% Full)
(.text + .data + .bootloader)

Data:       7104 bytes (86.7% Full)
(.data + .bss + .noinit)
```

FIGURE 7.1 – Exemple d'utilisation de la commande `avr-size`

physiques de temps d'exécution de programmes sur un microcontrôleur. Les exécutions simulées des programmes seront mesurées grâce à la commande UNIX `time`, qui permet de mesurer la durée d'exécution d'un programme. À partir de ces différentes mesures, et grâce à l'information – indiquée par la simulation du programme – du nombre total d'instructions exécutées, il est alors possible de déduire une vitesse moyenne d'OMicroB en nombre d'instructions par seconde. Il est à noter que cette vitesse peut varier selon la complexité des instructions bytecode exécutées par le programme considéré : le traitement d'une instruction riche (comme l'instruction `CLOSURE` de création de fermeture) est potentiellement bien plus long que celui d'une autre instruction bytecode plus basique (comme l'instruction `PUSH` qui ajoute un élément au sommet de la pile). En ce sens, la vitesse calculée ne peut être qu'une indication imprécise des capacités d'OMicroB, qui peuvent varier fortement d'un programme à l'autre.

Ces deux critères d'évaluation sont fortement dépendants des options du compilateurs citées précédemment. En effet, la taille des programmes en RAM est variable selon les tailles de la pile et du tas choisies, tandis que la vitesse d'exécution d'un programme peut dépendre de l'architecture choisie : par exemple, un microcontrôleur pourra prendre plus de temps à charger et traiter des données représentées sur 32 bits que si ces dernières étaient représentées sur 16 bits. De surcroît, l'activation plus fréquente de l'algorithme de *garbage collection* lors de l'utilisation d'un faible espace mémoire induit l'allongement des temps d'exécution des programmes OCaml, dont une partie de cette exécution est dédiée au nettoyage de la mémoire inutilisée dans le tas.

### 7.1.2 Programmes de test

Les différentes mesures présentées dans cette section sont réalisées sur des programmes OCaml simples, qui permettent chacun de tester les traits fondamentaux du langage et de sa machine virtuelle. Le code source de ces différents programmes est disponible en ligne [⚓1], et nous détaillons succintement le fonctionnement de chacun d'entre eux ci-après :

1. Le programme `apply.ml` réalise mille fois l'application successive d'une fonction ($\lambda f\ x.f\ (f\ x)$) représentant l'encodage des entiers de Church. Ce programme permet de tester le mécanisme d'application de fonctions d'ordre supérieur.
2. Le programme `fibo.ml` calcule cent mille fois les dix premiers termes de la suite de Fibonacci. Ce programme permet de tester l'application de fonction récursive, ainsi que l'utilisation d'opérations arithmétiques de base.

3. Le programme `takc.ml` calcule mille fois le résultat de la fonction de Takeuchi, avec pour paramètres 18, 12, et 6. Ce programme manipule des triplets, et teste donc le garbage collector, ainsi que l'application de fonctions récursives.
4. Le programme `oddeven.ml` calcule dix mille fois la parité des nombres entiers entre 0 et 100. Ce programme permet de tester l'application de fonctions mutuellement récursives.
5. Le programme `floats.ml` réalise dix millions de fois des applications de fonctions trigonométriques (`sin` et `cos`) sur des nombres flottants. Il permet de tester la représentation des flottants, ainsi que les performances des opérations sur ces derniers.
6. Le programme `integr.ml` calcule dix mille fois l'intégrale $\int_0^{10}(x^2 + 2x + 10)\,dx$, par pas de 0.01. Ce programme teste la représentation des nombres flottants, ainsi que la génération dynamique de valeurs fonctionnelles.
7. Le programme `eval.ml` correspond à un évaluateur de formules logiques booléennes, et calcule dix mille fois la valeur de dix formules booléennes. Il permet de tester la définition de types algrébriques, le filtrage par motif, ainsi que l'utilisation du module `Stack`, qui représente une structure de pile.
8. Le programme `sieve.ml` réalise dix mille fois le calcul des nombres premiers inférieurs à 50 par l'algorithme du crible d'Eratosthène. Ce programme induit la création de listes, et teste ainsi les capacités de l'algorithme de *garbage collection*.
9. Le programme `objet.ml` permet de tester la représentation des objets en OCaml : il définit une classe `point`, génère cent mille objets de cette classe, et calcule pour chacun d'entre eux dix fois leur symétrique par rapport à l'origine en appliquant l'appel d'une méthode `sym`.
10. Le programme `functor.ml` permet de tester le mécanisme de modules paramétés (ou *foncteurs*) d'OCaml. Il génère un module permettant de représenter un ensemble d'entiers grâce à l'application du foncteur `Set.Make` de la bibliothèque standard, puis ajoute dix mille fois à un ensemble d'entiers les valeurs contenues dans une liste composée de 50 entiers consécutifs.
11. Le programme `bubble.ml` est une implantation à l'aide de tableaux de l'algorithme de tri à bulles. Il génère un tableau de 60 éléments peuplés de valeurs entières décroissantes et applique dix mille fois le tri à bulles sur ce tableau. Ce programme permet de tester l'implantation des tableaux, l'accès à leurs valeurs, et leur mise à jour.
12. Le programme `jdlv.ml` implante le jeu de la vie de Conway sur une matrice $10 \times 10$. Il calcule les dix mille premières configurations du jeu de la vie à partir d'un état qui représente un oscillateur. Ce programme, qui utilise des enregistrements à champs mutables et des matrices, permet de tester l'implantation de ces derniers.
13. Le programme `share.ml` définit une fonction de filtrage de liste faisant usage du mécanisme d'exceptions d'OCaml afin d'éviter la recopie inutile des éléments de la liste d'origine dans la liste résultante. Il calcule cent mille fois le filtrage de la liste des entiers naturels de 0 à 50 par la fonction $(\lambda x.x > 25)$. Ce programme empile de nombreux récupérateurs d'exceptions et permet ainsi de tester le mécanisme de capture des expressions en OCaml.
14. Le programme `abrsort.ml` réalise dix mille fois le tri par arbres binaires de recherche d'un ensemble d'entiers naturels compris entre 0 et 100. Il permet de tester les performances d'OMicroB sur une structure de données récursive.

15. Enfin, le programme `queens.ml` résout dix mille fois le problème des $n$ reines, avec un nombre $n$ de reines allant de 1 à 6. Ce programme permet de tester les capacités de calcul de la machine virtuelle sur un exemple algorithmique non trivial.

Les mesures sur ces programmes ont été réalisées en deux temps : les programmes sont d'abord testés sur PC par l'intermédiaire d'une compilation d'OMicroB avec le compilateur *gcc*. Dans un second temps, les mêmes tests sont exécutés sur un microcontrôleur AVR ATmega2560.

### 7.1.3 Exécution sur ordinateur : résultats et interprétation

Les programmes abordés dans cette section ont été compilés avec l'option d'OMicroB `-no-shortcut-initialization`, afin d'éviter de déclencher le mécanisme d'évaluation partielle d'OMicroB, et que l'intégralité des calculs soit réalisée à la compilation, puisque ces tests ne font appel à aucune primitive d'entrée/sortie.

L'ensemble des mesures effectuées est réalisé à partir d'une représentation 16 bits des valeurs OCaml, de l'algorithme de garbage-collection *Stop and Copy*, d'une taille de tas de 2500 mots et d'une taille de pile de 500 mots. Le processeur de l'ordinateur utilisé est un Intel Core i5-8259U quadricœur, doté d'une architecture 64 bits, et d'une vitesse d'horloge de 2.3 GHz.

**Vitesse d'exécution :** Les tests réalisés permettent de comparer la vitesse d'exécution de la machine virtuelle OMicroB (en nombre d'instructions bytecode par seconde) et celle de l'interprète de bytecode de l'implantation standard de la ZAM, nommée *ocamlrun*. Ce dernier est utilisé dans une configuration standard : avec des valeurs encodées sur 64 bits, une pile de 256 000 mots, et un tas mineur de 2 méga-octets. Les résultats des mesures réalisées à partir de la configuration décrite précédemment sont présentés dans la table 7.1. En particulier, la colonne nommée « Ratio » représente le rapport entre la durée d'exécution d'un programme avec OMicroB et celle avec ocamlrun : une petite valeur est donc préférable. D'autres résultats de mesures réalisées avec le GC *Mark and Compact*, ainsi qu'à partir de représentations 32 bits et 64 bits des données, sont disponibles en annexes C.1, C.2, et C.3.

À partir des diverses mesures réalisées sur PC, il apparaît que OMicroB a une vitesse d'exécution qui peut dans certains cas être environ 2 à 3 fois plus lente qu'ocamlrun, l'interprète de bytecode standard. Ces résultats sont très corrects, dans la mesure où les capacités d'OMicroB restent dans un ordre de grandeur attendu, et nos optimisations ne visent pas réellement des performances en matière de vitesse, mais plutôt des améliorations en ce qui concerne la consommation mémoire. Cette différence de vitesse peut s'expliquer par de nombreux facteurs. Citons en particulier le fait que la représentation du programme octet par octet dans un tableau de valeurs C induit un surcoût évident, comparé au fait qu'ocamlrun charge le programme en RAM, et lit directement des valeurs dont la longueur correspond à l'architecture du processeur de l'ordinateur (32 ou 64 bits). De plus, les limitations en mémoire dynamique sont différentes entre OMicroB (ici : 2500 mots pour le tas – qui n'est qu'à moitié disponible du fait de l'utilisation du garbage-collector *Stop and Copy*) et ocamlrun (par défaut, le tas mineur d'OCaml est de 2 méga-octets sur une plateforme 64 bits), ce qui entraîne un déclenchement plus fréquent de l'algorithme de nettoyage mémoire d'OMicroB, et ralentit alors l'exécution du programme. À des fins d'illustration, les graphiques de la figure 7.2 représentent les temps d'exécution des programmes `takc.ml` et `abrsort.ml` en fonction de diverses tailles de tas, et de l'algorithme de GC sélectionné (pour une représentation 16 bits des valeurs) : la tendance générale de ce temps est de décroître à mesure que la taille de tas disponible

| Nom | Durée d'exécution avec ocamlrun (secondes) | Durée d'exécution avec OMicroB (secondes) | Ratio | Vitesse d'exécution avec OMicroB (millions d'instr. bytecode/seconde) | Nombre de déclenchements du GC d'OMicroB |
|---|---|---|---|---|---|
| `apply` | 1.20 | 2.14 | 1.78 | 306.33 | 58 |
| `fibo` | 0.55 | 1.14 | 2.07 | 428.82 | 0 |
| `takc` | 1.05 | 3.07 | 2.92 | 383.31 | 226000 |
| `oddeven` | 0.28 | 0.60 | 2.14 | 614.68 | 0 |
| `floats` | 0.56 | 1.05 | 1.87 | 219.46 | 0 |
| `integr` | 0.04 | 0.12 | 3.00 | 222.16 | 42 |
| `eval` | 0.04 | 0.07 | 1.75 | 431.42 | 2352 |
| `sieve` | 0.04 | 0.07 | 1.75 | 486.00 | 3333 |
| `objet` | 0.10 | 0.17 | 1.70 | 389.77 | 11765 |
| `functor` | 0.24 | 0.60 | 2.50 | 392.77 | 30003 |
| `bubble` | 0.93 | 1.55 | 1.66 | 461.49 | 35 |
| `jdlv` | 0.36 | 0.75 | 2.08 | 457.64 | 6666 |
| `share` | 0.11 | 0.25 | 2.27 | 480.24 | 699 |
| `abrsort` | 0.47 | 1.22 | 2.59 | 321.27 | 119924 |
| `queens` | 1.35 | 3.35 | 2.48 | 413.54 | 149999 |

Table 7.1 – Mesures de performances de programmes pour OMicroB (sur PC)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 500 -heap-size 2500*

augmente, jusqu'à se stabiliser dès le moment où le programme consomme moins de mémoire que ce qui lui est disponible.

Enfin, le test `floats` témoigne de l'intérêt de notre représentation immédiate des flottants : sur ce test, OMicroB est en 64 bits presque aussi rapide, et même légèrement plus rapide en 32 bits, qu'ocamlrun (cf. les tables C.2 et C.3 en annexes). En effet, l'absence d'allocation des flottants permet à la fois de limiter les déclenchements du GC, et à la fois de réduire le nombre d'indirections induites par la représentation classique des flottants, sous forme de valeurs encapsulées qui sont allouées sur le tas.

### 7.1.4 Exécution sur microcontrôleur : résultats et interprétation

Dans la suite, nous présentons et interprétons les mesures de performances réalisées lors de l'exécution des programmes de test sur un microcontrôleur. Ce dernier est un microcontrôleur AVR ATmega2560, qui bénéficie de 256 ko de mémoire flash, de 8ko de RAM, et d'une vitesse d'horloge de 16 MIPS. Les programmes sont ici aussi compilés avec l'option `no-shortcut-initialization`, dans une représentation 16 bits des valeurs OCaml, avec une pile de 500 mots au maximum, un tas de 2500 mots, et le garbage collector *Stop and Copy*. Il est à noter qu'en raison de la lenteur relative d'un microcontrôleur par rapport à un ordinateur récent, le nombre de calculs réalisés dans ces tests a été divisé par mille comparativement aux tests sur ordinateur, afin de réduire la durée des mesures. Par exemple, le programme `queens.ml` ne calcule plus que dix fois le problème des $n$ reines (avec $n$ de 1 à 6).

**Vitesse d'exécution :** La table 7.2 représente les résultats des mesures relatives au temps d'exécution des programmes pour une version 16 bits de la machine virtuelle. Les mesures de temps ont été réalisées

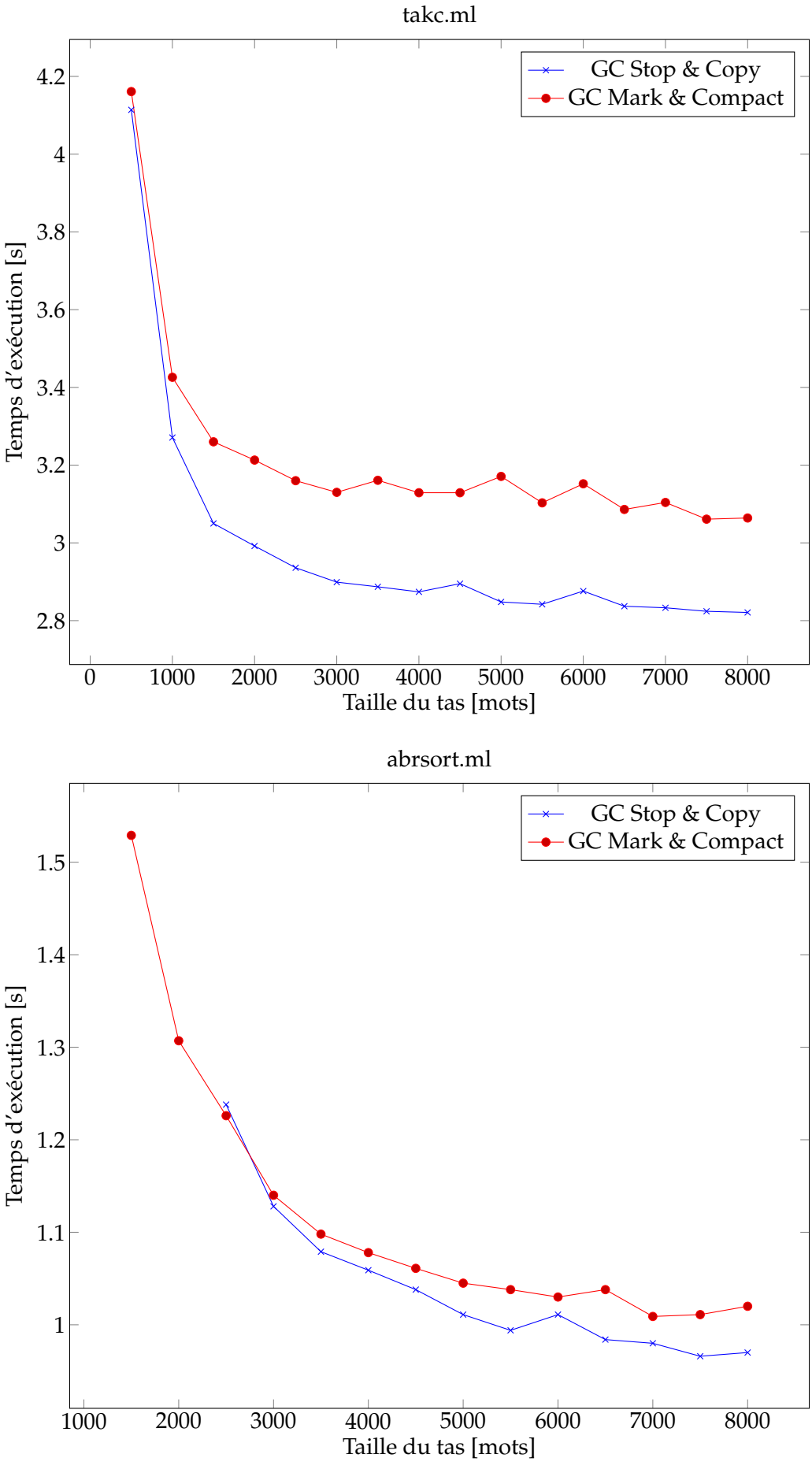



FIGURE 7.2 – Influence de la taille du tas sur le temps d'exécution (sur PC)

| Nom | Durée d'exécution (secondes) | Vitesse (milliers d'instr./seconde) | Nombre de déclenchements du GC |
|---|---|---|---|
| `apply` | 3.579 | 183.191 | 0 |
| `fibo` | 1.984 | 246.473 | 0 |
| `takc` | 5.475 | 214.951 | 228 |
| `oddeven` | 1.334 | 276.536 | 0 |
| `floats` | 7.482 | 30.812 | 0 |
| `integr` | 0.745 | 35.910 | 0 |
| `eval` | 0.143 | 212.048 | 2 |
| `sieve` | 0.195 | 174.953 | 3 |
| `objet` | 0.420 | 162.138 | 12 |
| `functor` | 1.351 | 192.904 | 33 |
| `bubble` | 3.762 | 190.169 | 0 |
| `jdlv` | 2.006 | 171.273 | 6 |
| `share` | 0.572 | 211.891 | 0 |
| `abrsort` | 2.677 | 147.965 | 124 |
| `queens` | 6.598 | 209.987 | 154 |

Table 7.2 – Mesures de vitesse de programmes pour OMicroB (sur ATmega2560)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 500 -heap-size 2500*

grâce à l'utilisation d'une fonction OCaml `Avr.millis : unit -> int` qui retourne le nombre de millisecondes écoulées depuis le lancement du programme sur le microcontrôleur. La différence entre la valeur retournée par cette fonction à la fin du programme et celle retournée au début du programme permet d'estimer sa durée d'exécution. Cette valeur est par la suite transmise, via un protocole de transmission série, à l'ordinateur personnel auquel est branché le microcontrôleur.

Les mesures réalisées nous permettent d'évaluer la vitesse d'exécution d'OMicroB sur AVR à environ 200 000 instructions bytecode par seconde. Ces résultats sont dépendants de nombreux facteurs, tels que les capacités et le niveau d'optimisation du compilateur utilisé (ici, l'option appliquée à `avr-gcc` est `-O2`), la nature des instructions exécutées (dont le temps d'exécution varie selon leur niveau de « complexité »), ou bien des spécificités inhérentes au matériel. À ce propos, les tests `floats.ml` et `integr.ml`, qui manipulent des nombres flottants, se distinguent par une vitesse d'exécution nettement inférieure. Un tel ralentissement s'explique par les spécificités du matériel utilisé : sur un microcontrolleur AVR, les opérations sur les nombres flottants sont réalisées de façon logicielle, et sont ainsi plus lentes que sur un PC où ces opérations sont exécutées par l'unité arithmétique du processeur. De plus, notre représentation des flottants en version 16 bits de la machine virtuelle n'est pas standard, et impose donc des conversions au cours de l'exécution du programme.

La table C.4, en annexe, représente les mesures sur les mêmes programmes dans une version 32 bits de la machine virtuelle. La vitesse mesurée y est, comme attendu, plus lente d'un facteur proche de 2.

**Comparaison avec OCaPIC :** Nous comparons ici les résultats précédents en exécutant une partie des programmes de test avec OCaPIC, la machine virtuelle OCaml destinée aux microcontrôleurs de la famille PIC 18. En particulier, nous exécutons ces programmes pour un microcontrôleur PIC 18f4620, doté de 64 ko de mémoire flash, 4 ko de mémoire RAM, et d'une vitesse de calcul de 10 MIPS. La table 7.3

représente les résultats des mesures de temps d'exécution de ces programmes [2]. OMicroB n'ayant pas à ce jour été totalement porté pour microcontrôleur PIC 18, ces temps sont comparés avec ceux calculés pour ATmega2560, en extrapolant la vitesse de calcul de ce microcontrôleur pour coincider avec les 10 MIPS du PIC 18f4620. En ce sens, la comparaison des performances entre OCaPIC et OMicroB ne peut être ici qu'imprécise et seulement destinée à donner une idée générale de l'ordre de grandeur des performances comparées entre ces deux machines virtuelles. De ces mesures, il résulte que OMicroB est environ 3.6 fois plus lent (à fréquences de calcul égales) que la machine virtuelle OCaPIC. Il est important de rappeler que cette dernière bénéficie d'un interprète de bytecode et un garbage collector réalisés en assembleur, tirant profit d'optimisations spécifiques aux PIC, ce qui peut expliquer ses performances. De plus, certains aspects inhérents au matériel ont des conséquences sur la vitesse d'exécution des programmes : par exemple, un PIC ne nécessite que 2 cycles d'horloge pour lire un octet sur la mémoire flash, contre 12 cycles sur un microcontrôleur AVR.

Par ailleurs, des ébauches d'expérimentations visant à porter OMicroB sur PIC 18 ont mis en exergue l'influence de la qualité du code généré par les compilateurs sur la vitesse des programmes OCaml exécutés : l'utilisation du compilateur `xc8`, fourni par le fabricant, entraîne des vitesses d'exécutions qui sont environ 7 fois plus lentes sur OMicroB que sur OCaPIC, pour un même programme et sur un même microcontrôleur. Cette lenteur relative est en partie explicable par le fait que le code machine généré par ce compilateur est assez mal optimisé. Par exemple, le « *switch* » présent dans l'interprète de bytecode, dont le rôle est de distinguer les instructions en fonction de leur opcode pour permettre leur exécution, a été transformé par `xc8` en une cascade d'instructions de branchement conditionnel imbriquées : la vitesse d'exécution du programme en est de toute évidence impactée.

| **Nom** | **Durée d'exécution avec OMicroB sur ATmega2560 à 10 MIPS** (en secondes) | **Durée d'exécution avec OCaPIC sur PIC 18F4620 à 10 MIPS** (en secondes) | **Ratio OMicroB/OCaPIC** |
|---|---|---|---|
| `apply` | 5.726 | 1.632 | 3.51 |
| `fibo` | 3.174 | 0.860 | 3.69 |
| `takc` | 8.76 | 2.535 | 3.46 |
| `oddeven` | 2.134 | 0.657 | 3.25 |
| `integr` | 1.192 | 0.252 | 4.73 |
| `eval` | 0.229 | 0.064 | 3.58 |
| `sieve` | 0.312 | 0.084 | 3.71 |
| `bubble` | 6.030 | 1.793 | 3.36 |
| `share` | 0.915 | 0.251 | 3.65 |

TABLE 7.3 – Comparaison des temps d'exécution avec OCaPIC
*Options d'OMicroB : -arch 16 -gc SC -stack-size 500 -heap-size 2500*

Néanmoins, les performances temporelles d'OMicroB restent globalement tout à fait acceptables, et le fait que la machine virtuelle OMicroB soit portable est un avantage qu'il est important de prendre en compte. Ainsi, compte tenu de la portabilité de la solution, les performances d'OMicroB permettent de témoigner de la viabilité de l'utilisation d'une machine virtuelle générique. Nous présenterons, au

2. Les autres programmes, trop gourmands en mémoire, n'ont pu être utilisés sur ce matériel.

chapitre 8, un ensemble d'applications qui démontrent la réactivité de la machine virtuelle, ainsi que son adéquation pour l'exécution de programmes embarqués.

**Élements de comparaison avec MicroPython :** Afin de donner un ordre d'idée des performances d'OMicroB par rapport à des solutions pré-existantes et assez fréquemment utilisées, nous réalisons quelques mesures comparatives entre certains exemples de programmes OCaml et leurs équivalents écrits en Python. Ces derniers sont exécutés par l'implémentation MicroPython sur un *pyboard* (version 1.1) doté d'un microcontrôleur STM32F405RG équipé de 1024 kilo-octets de mémoire flash, de 192 kilo-octets de RAM, et d'un processeur qui possède une vitesse de calcul de 168 MHz. De la même façon que pour la comparaison avec OCaPIC, nous extrapolons les résultats des mesures réalisées sur le pyboard afin de les comparer avec les mesures d'OMicroB réalisées sur un ATmega2560 à 16 MHz.

La comparaison entre MicroPython et OMicroB ne peut être que globalement indicative des performances relatives de l'une et de l'autre de ces solutions, tant la différence entre des programmes écrits en OCaml et des programmes écrits en Python est importante. En effet, la programmation dans un style « *camélien* » fait grand usage de fonctions et structures récursives, tandis que la récursivité en Python est assez fortement limitée : il n'y a par exemple en Python pas d'optimisation pour les appels terminaux, qui permet de « dérécursiver » certaines fonctions. De ce fait, un grand nombre de nos programmes de test présentés dans cette section est incompatible avec une exécution avec MicroPython car ils consomment rapidement toute la profondeur de la pile d'appel de Python. Cette pile a par ailleurs une profondeur assez faible d'uniquement 63 niveaux sur un pyboard.

Nous présentons alors, dans la table 7.4 les résultats de mesures réalisées sur un sous-ensemble de nos programmes de test qui correspond à ceux qui sont compatibles avec MicroPython sans avoir à modifier de façon importante leur structure. Il est à noter que le test `bubble` qui réalise le tri à bulles fait, dans notre version Python, usage des *listes* de Python [3], car les tableaux de la librairie Numpy sont inaccessibles en MicroPython. Le test `jdlv` a quant à lui été implanté à l'aide d'objets.

| **Nom** | **Durée d'exécution avec OMicroB sur ATmega2560 à 16 MHz** (en secondes) | **Durée d'exécution avec MicroPython sur pyboard (extrapolée à 16 MHz)** (en secondes) | **Ratio MicroPython/OMicroB** |
|---|---|---|---|
| `fibo` | 1.984 | 5.481 | 2.76 |
| `takc` | 5.475 | 155.620 | 28.42 |
| `bubble` | 3.762 | 4.420 | 1.17 |
| `jdlv` | 2.006 | 3.024 | 1.5 |
| `objet` | 0.420 | 0.630 | 1.5 |

Table 7.4 – Comparaison des temps d'exécution avec MicroPython
*Options d'OMicroB : -arch 16 -gc SC -stack-size 500 -heap-size 2500*

Les résultats sur ces quelques mesures sont très prometteurs : OMicroB est systématiquement plus rapide, à fréquences d'horloges égales. On peut en particulier noter la rapidité supérieure du test `fibo` (sûrement en raison du nombre important d'appels de fonctions) et la bien meilleure performance d'OMicroB sur le test `takc` qui alloue dynamiquement de nombreux n-uplets au cours de son exécution

3. L'accès et la modification d'un élément dans de telles listes a tout de même une complexité en $\mathcal{O}(1)$, comme les tableaux OCaml.

(et déclenche de nombreuses récupérations mémoire), mais les performances d'OMicroB sont également bonnes pour les tests impératifs et objets. Il est à noter par ailleurs que les meilleurs résultats pour les tests `fibo` et `takc` ne viennent pas des optimisations de récursivité terminale du compilateur OCaml puisqu'elle n'est pas réalisée dans `fibo` et que seul l'appel récursif englobant de la fonction de Takeuchi est optimisé.

**Taille des programmes et de l'interprète :** La table 7.5 représente l'empreinte de chaque programme sur la mémoire flash et la mémoire RAM du microcontrôleur. L'empreinte quasiment constante des programmes sur la RAM du microcontrôleur provient du fait que les tailles de pile et de tas sont choisies à la compilation : de ce fait, des tableaux de constantes sont générés statiquement pour représenter ces sections de la mémoire, dont la taille ne peut changer au cours de l'exécution. Il est à noter que la taille des programmes en mémoire `objet` et `functor` en mémoire flash est assez importante. Pour le cas du premier programme, cela est dû à l'importation dans le *bytecode* du programme de tout le mécanisme permettant de créer des objets et de les manipuler. Ce dernier est assez coûteux sur le plan de son occupation en mémoire flash, mais il est toutefois à noter que ce surcoût est stable : l'empreinte mémoire d'un programme qui manipule dix fois plus d'objets et de classes n'a pas une empreinte mémoire dix fois plus importante. En ce qui concerne le programme qui manipule des foncteurs, le surcoût en mémoire est lié au fait qu'*ocamlclean* n'est pas en mesure de détecter statiquement le code mort présent dans un module résultat d'une application de foncteur. Ainsi, dans notre test, le bytecode généré contient l'intégralité du code d'exécution de tout le module résultant de l'application du foncteur `Set.Make`, même si nous ne faisons usage que d'un sous-ensemble limité des fontions de ce module.

| Nom | Empreinte mémoire en Flash (octets) | Empreinte mémoire RAM (octets) |
|---|---|---|
| `apply` | 7626 | 6116 |
| `fibo` | 8614 | 6116 |
| `takc` | 8568 | 6114 |
| `oddeven` | 8230 | 6112 |
| `floats` | 10536 | 6128 |
| `integr` | 10870 | 6122 |
| `eval` | 13420 | 6239 |
| `sieve` | 9550 | 6119 |
| `objet` | 24120 | 6327 |
| `functor` | 20162 | 6241 |
| `bubble` | 11926 | 6217 |
| `jdlv` | 11772 | 6149 |
| `share` | 11122 | 6135 |
| `abrsort` | 13600 | 6219 |
| `queens` | 10704 | 6139 |

Table 7.5 – Mesures de taille de programmes pour OMicroB (sur ATmega2560)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 500 -heap-size 2500*

L'interprète de la machine virtuelle, ainsi que sa bibliothèque d'exécution, ont également un poids notable dans ces mesures. Il est possible d'estimer la taille de l'interprète et du *garbage collector* en mesurant

la taille (en mémoire flash) d'un programme qui calcule seulement la valeur *unit*. Le bytecode associé à ce programme est réduit à la seule instruction bytecode `STOP`. Comme l'optimisation qui consiste à supprimer, dans l'interprète, le code de traitement des instructions bytecode non-utilisées dans le programme est activée par défaut, l'empreinte mémoire du programme exécutable résultant est très faible : dans une version 16 bits d'OMicroB la commande `avr-size` affiche une taille de 1240 octets sur la mémoire flash. Cette valeur correspond ainsi à la taille du plus petit programme OCaml possible compilé avec OMicroB. À l'inverse, lorsque ce même programme est compilé avec l'option `no-clean-intepreter` d'OMicroB, la taille du programme exécutable en mémoire flash est de 21340 octets. Par conséquent, la taille de l'interprète de bytecode ainsi que du garbage collector (qui n'est pas chargé lorsque le code de traitement de toutes les instructions bytecode capables d'allouer de la mémoire est supprimé) est de l'ordre de 20 kilo-octets. Un programme réduit à la seule instruction `CLOSURE` qui alloue une fermeture (et inclut donc le garbage collector) a une taille de 3110 octets, avec le « nettoyage » de l'interprète activé. Ainsi, l'empreinte mémoire du GC *Stop and Copy* est de l'ordre de 2000 octets. Les mêmes mesures sur le garbage collector *Mark and Compact* font état d'une empreinte mémoire de l'ordre de 3000 octets pour ce ramasse-miettes. Dans une configuration 32 bits de la machine virtuelle, l'empreinte mémoire de l'interprète et du garbage collector est de l'ordre de 28 kilo-octets, tandis qu'en 64 bits elle est d'environ 40 kilo-octets.

Il résulte ainsi de ces mesures que sans l'optimisation apportée par la compilation d'un interprète « sur mesure », les programmes utilisant cette machine virtuelle atteindraient rapidement les limites, en matière d'occupation en mémoire flash, des microcontrôleurs visés par notre solution (par exemple, un ATmega32u4 ne possède que 32ko de mémoire flash). Grâce à cette optimisation, les programmes de test ne consomment en moyenne qu'une dizaine de kilo-octets de la mémoire flash, ce qui correspond à la taille de l'interprète spécifique à chaque programme, à sa bibliothèque d'exécution, ainsi qu'au bytecode du programme qui réside également en mémoire flash. Ainsi, cette optimisation offre des résultats non négligeables et permet l'exécution d'OMicroB sur du matériel dans lequel l'inteprète complet n'aurait pas pu être embarqué. Il est à noter qu'un programme faisant usage de toutes les instructions bytecode de la machine virtuelle importerait aussi, à l'évidence, l'ensemble de l'interprète de bytecode. Cependant, ce cas de figure est très rare, et les programmes présentés dans ce manuscrit illustrent le fait qu'une majorité de programmes OCaml ne fait usage que d'un sous-ensemble plutôt restreint (quoique variable d'un programme à l'autre) des instructions bytecode du langage.

## 7.2 Mesures de performances d'OCaLustre

Dans cette section, nous entreprenons d'évaluer les performances du code OCaml produit par OCaLustre, et son adéquation avec les limitations matérielles des microcontrôleurs concernés par cette thèse. Pour ce faire, nous élaborons un exemple de programme OCaLustre réalisant plusieurs calculs, et mesurons la vitesse et l'occupation mémoire du programme exécuté dans OMicroB. Cet exemple représente un *additionneur parallèle à propagation de retenue* qui permet de calculer la somme binaire de mots de 8 bits. Cet additionneur, représenté dans la figure 7.3 est constitué de huit éléments indépendants qui permettent chacun l'addition de deux valeurs représentées par un seul bit. Ces additionneurs 1 bit calculent la somme de deux bits, ainsi que sa retenue éventuelle, et sont dits « *complets* » (en anglais : *full adders*) car ils prennent également en entrée la valeur de la potentielle retenue produite par l'additionneur dédié au calcul du bit de poids inférieur. Le calcul de la somme de deux octets ($a_7a_6a_5a_4a_3a_2a_1a_0$ et $b_7b_6b_5b_4b_3b_2b_1b_0$)

est réalisé en additionnant chaque paire de bits $(a_i, b_i)$ tout en propageant la retenue calculée par chaque additionneur 1 bit vers l'additionneur qui le succède.

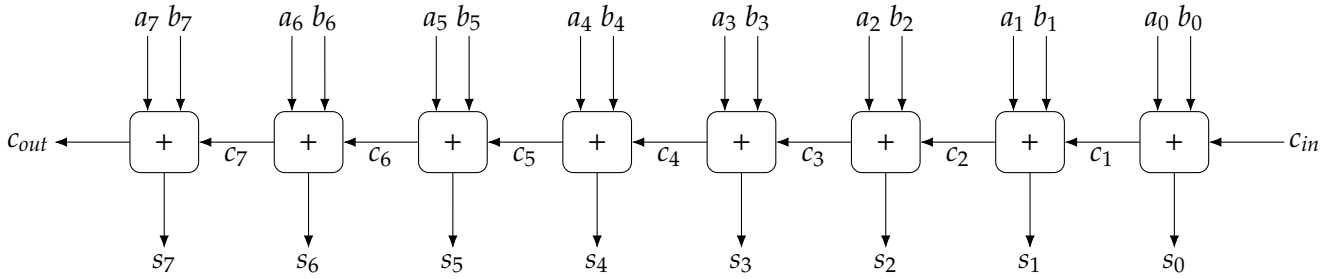



FIGURE 7.3 – Additionneur 8 bits

### 7.2.1 Un additionneur binaire en OCaLustre

Un additionneur 1 bit est concrétisé en OCaLustre par un nœud nommé `fulladder`, dont la définition suit précisemment la représentation standard d'un additionneur sous forme d'un diagramme constitué de portes logiques, tel qu'illustré dans la figure 7.4.

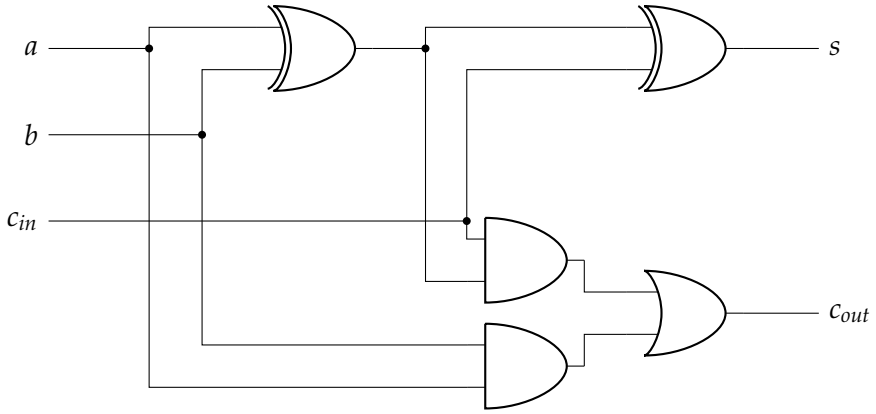



FIGURE 7.4 – Circuit d'un additionneur complet

Le code du nœud `fulladder` – ainsi que de la fonction OCaml `xor` permettant de calculer le « ou exclusif » entre deux booléens – est alors le suivant :

```
let xor a b = if a then not b else b

let%node fulladder (a,b,cin) ~return:(s,cout) =
  x = call xor a b;
  s = call xor x cin;
  and1 = (x && cin);
  and2 = (a && b);
  cout = (and1 || and2)
```

Ce nœud calcule la somme `s` de deux bits `a` et `b` et d'une retenue entrante `cin`, ainsi que la potentielle retenue sortante `cout`. Les valeurs des bits sont ici représentées par l'intermédiaire de booléens.

À partir d'un nœud `fulladder`, il est alors aisé de définir un additionneur capable de calculer la somme de deux valeurs sur 2 bits sous la forme d'un nœud `twobits_adder` dont le rôle est simplement de chaîner les calculs de deux additionneurs :

```
let%node twobits_adder (c0,a0,a1,b0,b1) ~return:(s0,s1,c2) =
  (s0,c1) = fulladder (a0,b0,c0);
  (s1,c2) = fulladder (a1,b1,c1)
```

Ce nœud calcule ainsi trois booléens correspondant à la somme $s_0 s_1$ deux valeurs $a_0 a_1$ et $b_0 b_1$, et à la valeur de sa retenue éventuelle $c_2$.

En suivant le même principe, nous pouvons alors définir un additionneur de valeurs sur quatre bits, puis un additionneur de valeurs représentées sur 8 bits :

```
let%node fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3) ~return:(s0,s1,s2,s3,c4) =
  (s0,s1,c2) = twobits_adder (c0,a0,a1,b0,b1);
  (s2,s3,c4) = twobits_adder (c2,a2,a3,b2,b3)

let%node eightbits_adder (c0,a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7)
                         ~return:(s0,s1,s2,s3,s4,s5,s6,s7,c8) =
  (s0,s1,s2,s3,c4) = fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3);
  (s4,s5,s6,s7,c8) = fourbits_adder (c4,a4,a5,a6,a7,b4,b5,b6,b7)
```

Cet exemple illustre parfaitement la compositionabilité inhérente à l'approche synchrone d'OCaLustre : une définition d'un nœud peut être réutilisée et combinée par un autre nœud, qui lui-même peut être combiné à un autre afin de réaliser aisément des applications de plus en plus complexes, sans que le développeur n'ait à se soucier des relations de causalité entre les différents composants du programme.

### 7.2.2 Résultats

À partir du nœud `eightbits_adder` défini ci-dessus, nous réalisons un programme OCaLustre qui calcule cent mille fois la somme $0b11111111 + 0b11111111$. Cet exemple est alors exécuté, de la même façon que les tests précédents, à l'aide de la machine virtuelle OMicroB, sur PC.

La table 7.6 illustre les mesures de performances réalisées sur ce programme. Notamment, l'empreinte mémoire d'OCaLustre est faible : une pile de 80 mots, et un tas de 400 valeurs OCaml suffisent pour l'exécution d'un tel programme, pour une consommation RAM totale de seulement 1069 octets, et une consommation en flash de 7668 octets.

D'un point de vue de la vitesse d'exécution les performances des programmes OCaLustre dans OMicroB sont bonnes, mais néanmoins ralenties par le nombre important de déclenchements du GC. Comme l'illustre le graphique de la figure 7.5, un tas plus grand permet d'en réduire drastiquement le nombre, et ainsi d'accélerer l'exécution du programme.

L'option de compilation `-na` d'OCaLustre permet de générer un code « impératif » optimisé pour l'embarqué critique, au sens où il n'entraîne pas d'allocations de nouvelles valeurs au cours d'un instant

| Nom | Durée d'exécution (secondes) | Vitesse (millions d'instr./seconde) | Nombre de GC |
|---|---|---|---|
| `adders.ml` | 0.54 | 138.01 | 550054 |

TABLE 7.6 – Mesures de vitesse de l'additionneur avec OMicroB (sur PC)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 80 -heap-size 400*

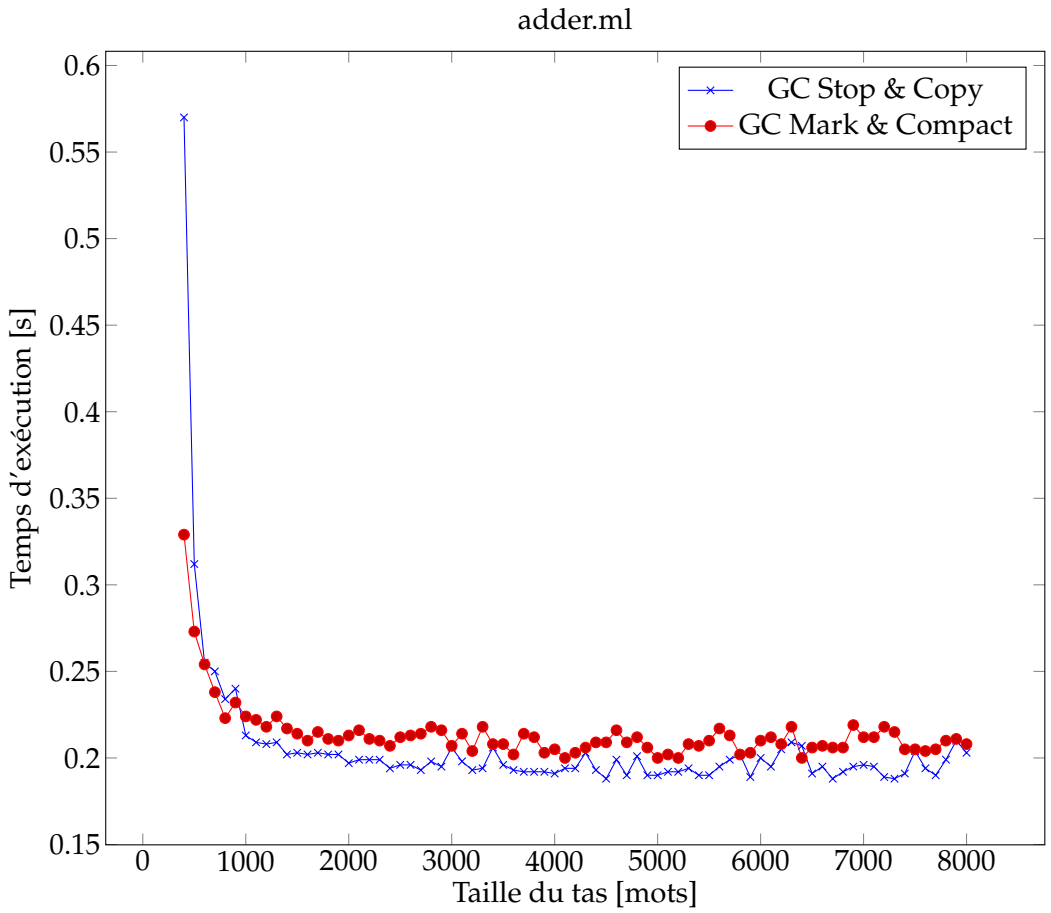



FIGURE 7.5 – Durée d'exécution de l'additionneur en fonction de la taille du tas (sur PC)

synchrone, et évite ainsi le déclenchement du garbage collector pendant l'exécution du programme synchrone. De ce fait, la vitesse d'exécution de l'additionneur est grandement accélérée lors de l'utilisation de cette option (table 7.7).

| Nom | Durée d'exécution (secondes) | Vitesse (millions d'instr./seconde) | Nombre de déclenchements du GC |
|---|---|---|---|
| `adder-noalloc.ml` | 0.21 | 383.14 | 0 |

TABLE 7.7 – Vitesse de l'additionneur sans allocations sur le tas en cours d'instant
*Options d'OMicroB : -arch 16 -gc SC -stack-size 80 -heap-size 400*

**Élements de comparaison avec Lucid Synchrone :** Nous proposons désormais de comparer les performances de ce programme avec celui d'un programme équivalent réalisé dans le langage Lucid Synchrone. Bien que le développement de ce langage soit aujourd'hui abandonné, son compilateur (dans sa version

3, datant d'avril 2006) reste disponible sur le site du projet [⚓19]. Ce dernier produit du code OCaml, permettant la comparaison avec OCaLustre. Cette comparaison permet principalement de vérifier le bien fondé de l'utilisation d'OCaLustre pour la programmation de matériel à faible ressources, en s'assurant que les programmes synchrones réalisés n'auraient pas pu être écrits en Lucid Synchrone tout en conservant des performances équivalentes. Bien entendu, Lucid Synchrone est une extension riche de Lustre, dont l'expressivité est supérieure à celle d'OCaLustre, et sa richesse se répercute aussi dans la complexité du code généré. La comparaison proposée ici ne tient donc pas lieu de comparatif des performances brutes des deux langages, mais d' illustration du gain en place qu'apporte l'usage de notre extension de syntaxe sur les programmes que nous visons, de par sa relative simplicité. La figure 7.8 représente les mesures réalisées sur le programme Lucid Synchrone `adder.ls` (dont le code – très proche de la syntaxe d'OCaLustre – est donné en annexe B), ainsi que sur le programme OCaLustre dans sa version standard (`adder.ml`), et sa version optimisée pour l'embarqué critique (`adder-na.ml`). Les besoins mémoire pour permettre l'exécution du programme Lucid Synchrone étant supérieurs, les mesures ont été effectuées avec une pile de 150 mots, et un tas de 600 mots.

| **Nom** | **Durée d'exécution** (secondes) | **Vitesse** (millions d'instr./seconde) | **Nombre de déclenchements du GC** |
|---|---|---|---|
| `adder.ls` | 0.79 | 222.87 | 440042 |
| `adder.ml` | 0.25 | 298.11 | 110010 |
| `adder-na.ml` | 0.19 | 423.47 | 0 |

Table 7.8 – Vitesses des programmes Lucid Synchrone et OCaLustre (PC)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 150 -heap-size 600*

Il résulte de ces mesures que sur ce petit programme, à configurations de la machine virtuelle égales, les performances de Lucid Synchrone sur cet exemple sont trois à quatre fois moindres que celles d'OCaLustre sur PC. Sur un ATmega2560, les performances en vitesse de Lucid Synchrone sont 4 à 6 fois inférieures aux programmes OCaLustre [4] (table 7.9). En effet, de par la richesse du langage, le code OCaml généré par Lucid Synchrone est bien plus conséquent et plus gourmand en ressources que celui généré par OCaLustre : la figure 7.6 représente, pour le nœud `twobits_adder` le code OCaml généré par Lucid Synchrone, et celui généré par compilation OCaLustre. En particulier, la version Lucid Synchrone fait usage de variants polymorphes, qui sont assez coûteux, pour représenter certaines valeurs. Elle génère également à chaque instant plusieurs références vers des n-uplets, ce qui a pour conséquence de remplir assez rapidement le tas de la machine virtuelle. Ainsi, le fait que le programme Lucid Synchrone nécessite d'augmenter la taille du tas, et de doubler la taille de la pile est aussi assez pénalisant pour une utilisation sur des microcontrôleurs dont les ressources mémoires sont limitées à quelques kilo-octets.

| **Nom** | **Durée d'exécution** (secondes) | **Vitesse** (milliers d'instr./seconde) | **Nombre de GC** | **Taille en flash (octets)** | **Taille en RAM (octets)** |
|---|---|---|---|---|---|
| `adder.ls` | 2.181 | 88.827 | 482 | 9250 | 1627 |
| `adder.ml` | 0.554 | 148.185 | 120 | 7552 | 1609 |
| `adder-na.ml` | 0.366 | 221.453 | 0 | 8472 | 1613 |

Table 7.9 – Vitesses et tailles des programmes Lucid Synchrone et OCaLustre (Atmega2560)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 150 -heap-size 600*

4. Le nombre d'instants calculés a été une nouvelle fois divisé par mille pour nos mesures sur microcontrôleur.

```
let twobits_adder _cl147 (_148__c0, _149__a0, _150__a1, _151__b0, _152__b1) _325
 _self_364 =
  let _self_364 = match !_self_364 with
                | 'St_369(_self_364) -> _self_364
                | _ -> (let _368 = {_366 = ref 'Snil_;
                                    _365 = ref 'Snil_;
                                    _init329 = true} in
                        _self_364 := 'St_369(_368);
                        _368) in
  let _cl339__ = ref false in
  let _338 = ref (false, false) in
  let _cl337__ = ref false in
  let _328 = ref false in
  let _340 = ref (false, false) in
   (if _cl147 then
     (_328 := (or) _325 _self_364._init329;
      _cl337__ := true;
      _338 := fulladder !_cl337__ (_149__a0, _151__b0, _148__c0) !_328 _self_364._365));
   (let (_153__s0, _154__c1) = !_338 in
    (if _cl147 then
      (_cl339__ := true;
       _340 := fulladder !_cl339__ (_150__a1, _152__b1, _154__c1) !_328 _self_364._366));
    (let (_155__s1, _156__c2) = !_340 in
     _self_364._init329 ← (&) !_328 (not _cl147);
     (_153__s0, _155__s1, _156__c2)))
```

```
let twobits_adder () =
  let fulladder1_app = fulladder () in
  let fulladder2_app = fulladder () in
  fun (c0, a0, a1, b0, b1) ->
    let (s0, c1) = fulladder1_app (a0, b0, c0) in
    let (s1, c2) = fulladder2_app (a1, b1, c1) in
    (s0, s1, c2)
```

FIGURE 7.6 – Codes OCaml du nœud `twobits_adder` générés par Lucid Synchrone (haut) et par OCaLustre (bas)

**Mesures sur les exemples du manuscrit** : Le tableau 7.10 représente l'exécution de programmes OCaLustre issus de la plupart des exemples abordés dans ce manuscrit. Le nom de chaque programme correspond au nom de son nœud principal, qui est exécuté un million de fois. Les mesures réalisées valident les performances de vitesse mesurées sur l'additionneur, et la faible consommation mémoire d'OCaLustre. La consommation des ressources matérielles par l'extension OCaLustre est faible : son empreinte mémoire flash est comparable à celles d'autres programmes OCaml, et son empreinte en RAM est également basse. Cette utilisation parcimonieuse des ressources rend OCaLustre compatible avec de nombreux modèles de microcontrôleurs, possédant des capacités mémoires de l'ordre de 2 kilo-octets. Ainsi, OCaLustre apporte un modèle de programmation de haut niveau qui permet de réaliser des programmes synchrones apportant de nombreuses garanties (que ce soit sur le plan du typage – d'horloges, ou des données manipulées – ou de la détection d'incohérences causales au sein des programmes). Ces avantages sont d'autant plus importants qu'ils n'entraînent pas d'augmentation de la consommation en ressources, et permet ainsi le développement de systèmes embarqués reposant sur du matériel aux capacités limitées.

| **Nom** | **Durée d'exécution avec ocamlrun** (secondes) | **Durée d'exécution avec OMicroB** (secondes) | **Ratio** | **Vitesse d'exécution avec OMicroB** (millions d'instr. bytecode/seconde) | **Nombre de GC avec OMicroB** |
|---|---|---|---|---|---|
| `arith` | 0.85 | 1.54 | 1.81 | 337.76 | 357142 |
| `blinker` | 0.02 | 0.04 | 2.00 | 521.67 | 0 |
| `call_cpt` | 0.06 | 0.11 | 1.83 | 488.14 | 19058 |
| `cpt` | 0.01 | 0.03 | 3.00 | 645.06 | 0 |
| `ex_const` | 0.04 | 0.09 | 2.25 | 518.05 | 17721 |
| `ex_norm` | 0.03 | 0.06 | 2.00 | 575.05 | 0 |
| `ex_tuples` | 0.06 | 0.11 | 1.83 | 396.31 | 37411 |
| `fibonacci` | 0.02 | 0.04 | 2.00 | 635.31 | 0 |
| `fibonacci2` | 0.02 | 0.04 | 2.00 | 660.56 | 0 |
| `merge` | 0.03 | 0.06 | 2.00 | 558.22 | 0 |
| `ordo` | 0.06 | 0.13 | 2.16 | 397.50 | 37411 |
| `two_cpt` | 0.03 | 0.08 | 2.66 | 481.79 | 18705 |
| `watch` | 0.06 | 0.13 | 2.16 | 428.58 | 40404 |
| `when` | 0.06 | 0.10 | 1.66 | 395.53 | 18365 |
| `whennot` | 0.03 | 0.05 | 1.66 | 548.65 | 0 |

Table 7.10 – Mesures de performances de programmes OCaLustre (sur PC)
*Options d'OMicroB : -arch 16 -gc SC -stack-size 50 -heap-size 500*

### Conclusion du chapitre

Les différentes mesures réalisées dans ce chapitre valident le fait que notre approche reposant sur l'utilisation d'une machine virtuelle est viable et adaptée pour la programmation de microcontrôleurs dans un langage de haut niveau. Le simple fait que des programmes OCaml puissent être exécutés, à l'aide d'un interprète générique, sur du matériel avec de si faibles ressources, représente une réussite. De plus, les performances de la machine virtuelle OMicroB et de l'extension synchrone OCaLustre sont tout à fait correctes du point de vue de leur vitesse d'exécution, ainsi que de celui de la consommation mémoire des programmes générés : des applications non triviales peuvent être envisagées, pour une exécution sur du matériel très fortement limité en mémoire. Nous présentons ainsi dans le chapitre suivant quelques applications concrètes qui tirent profit d'OMicroB et d'OCaLustre.

# 8 Applications pour montages électroniques

Nous présentons un ensemble d'applications pratiques ou ludiques réalisées avec les solutions logicielles décrites dans ce manuscrit. Ces applications, pouvant être exécutées sur des microcontrôleurs à faibles quantités de ressources mémoires et computationnelles, témoignent de l'intérêt de l'approche haut niveau adoptée dans cette thèse. Il est en effet aisé de réaliser de petits programmes embarqués tirant profit des garanties apportées par le langage OCaml et la surcouche synchrone d'OCaLustre. Nous présentons trois exemples de programme, dont l'intérêt est de faire montre de l'expressivité de notre solution, tout en validant le fait que cette dernière est compatible avec du matériel doté de faibles ressources. En particulier, les programmes réalisés peuvent être exécutés sur du matériel disposant de moins de 8 kilo-octets de RAM. Chaque programme décrit dans la suite est lié à un montage électronique concret. Le premier exemple est un programme permettant de contrôler un lecteur de cartes perforées simple, pour lequel chaque carte contient la représentation binaire d'un octet. Cet exemple témoigne en particulier de la correspondance entre la notion d'*horloge* synchrone et celle d'horloges « physiques », qui gouvernent la cadence d'arrivée des signaux électriques en entrée du programme. Le second programme régit le fonctionnement d'une *tempéreuse à chocolat* : un appareil de cuisine permettant de faire fondre du chocolat à une température précise. Le dernier exemple est l'implémentation d'un jeu vidéo de « Serpent » pour un *Arduboy*, un petit appareil portatif dédié à l'exécution de jeux vidéo simples, classiquement réalisés en C.

L'ensemble des codes sources de ces programmes est disponible en annexe D.

## 8.1 Un lecteur de cartes perforées

Pour notre premier exemple, nous réalisons un programme pour microcontrôleur qui sera intégré à un montage représentant un lecteur de cartes perforées. Chacune de ces dernières contient la représentation d'un octet par l'intermédiaire de deux lignes horizontales contenant des *trous*. La première ligne représente le signal d'*horloge* : si le montage reconnaît un trou dans cette ligne, cela signifie qu'une donnée doit être également lue. Cette *donnée* est représentée sur la deuxième ligne de la carte : un trou représente la valeur binaire 1, tandis que l'absence de trou représente la valeur 0. La figure 8.1 est le schéma d'une carte perforée : cette carte correspond à l'octet représenté en binaire par la valeur $0b10001010$ [1].

### 8.1.1 Montage

Un montage électronique assez simple, représenté par le schéma de la figure 8.2, permet de réaliser un lecteur de telles cartes perforées. Il suffit en effet de connecter deux broches d'entrée numérique (notées $I_0$ et $I_1$ dans la figure) d'un microcontrôleur à deux tiges en métal, et que chacune de ces tiges exerce

1. La lecture d'un octet d'une carte se fait de gauche à droite, avec une représentation *big-endian*.

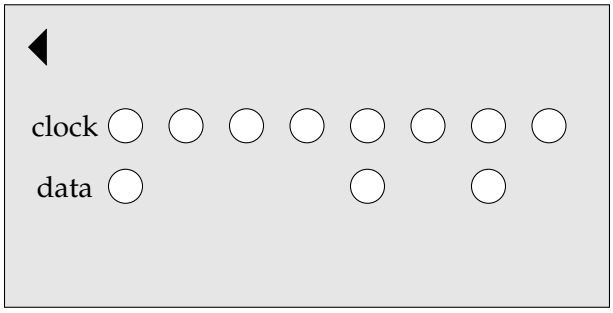



FIGURE 8.1 – Exemple schématisé d'une carte perforée

une légère pression au niveau de chaque ligne de trous de la carte perforée. La carte repose elle-même sur une plaque en métal alimentée par un courant électrique. Dès lors que la carte perforée est glissée entre la plaque et les tiges métalliques, la lecture des valeurs représentées par des trous (ou leur absence) peut être réalisée. En effet, si une tige passe sur un trou de la carte, le contact électrique est réalisé entre cette tige et la plaque de métal, et la broche du microcontrôleur associée est donc reliée à la source de courant électrique : la valeur 1 (*HIGH*) est reconnue par l'appareil. À l'inverse, quand une tige en métal repose sur la carte, le contact n'est pas réalisé, puisque cette dernière n'est pas conductrice, et la broche du microcontrôleur est alors reliée à la masse du circuit (par l'intermédiaire d'une résistance *pull-down*) : c'est la valeur 0 (*LOW*) qui est mesurée. Enfin, la valeur binaire mesurée sur une carte est émise par le microcontrôleur sur un ensemble de huit LED (branchées aux broches notées $O_0$ à $O_7$) dont chacune représentera un des bits de l'octet présent sur la carte : si le bit est à 1, la LED correspondante sera allumée, sinon elle sera éteinte.

### 8.1.2 Programme

L'utilisation d'un signal d'horloge représenté physiquement par une suite de trous sur la carte perforée illustre adéquatement la notion d'horloge dans un programme synchrone. Ainsi, un programme OCaLustre peut être facilement réalisé pour contrôler le montage décrit ci-dessus. Dans ce programme, un signal d'horloge doit être à *vrai* dès que la tige métallique positionnée sur la ligne qui représente le signal horloge se retrouve dans un trou. Néanmoins, tant que la tige reste dans un même trou, une suite de valeurs `true` ne doit pas être constamment produite : c'est en réalité *l'entrée* dans un trou qui doit indiquer un *tick* du signal d'horloge. Pour ce faire, nous définissons un nœud `edge` qui permet de détecter un tel front montant :

```
let%node edge x ~return:e =
  e = (x && (not (true ⋙ x)))
```

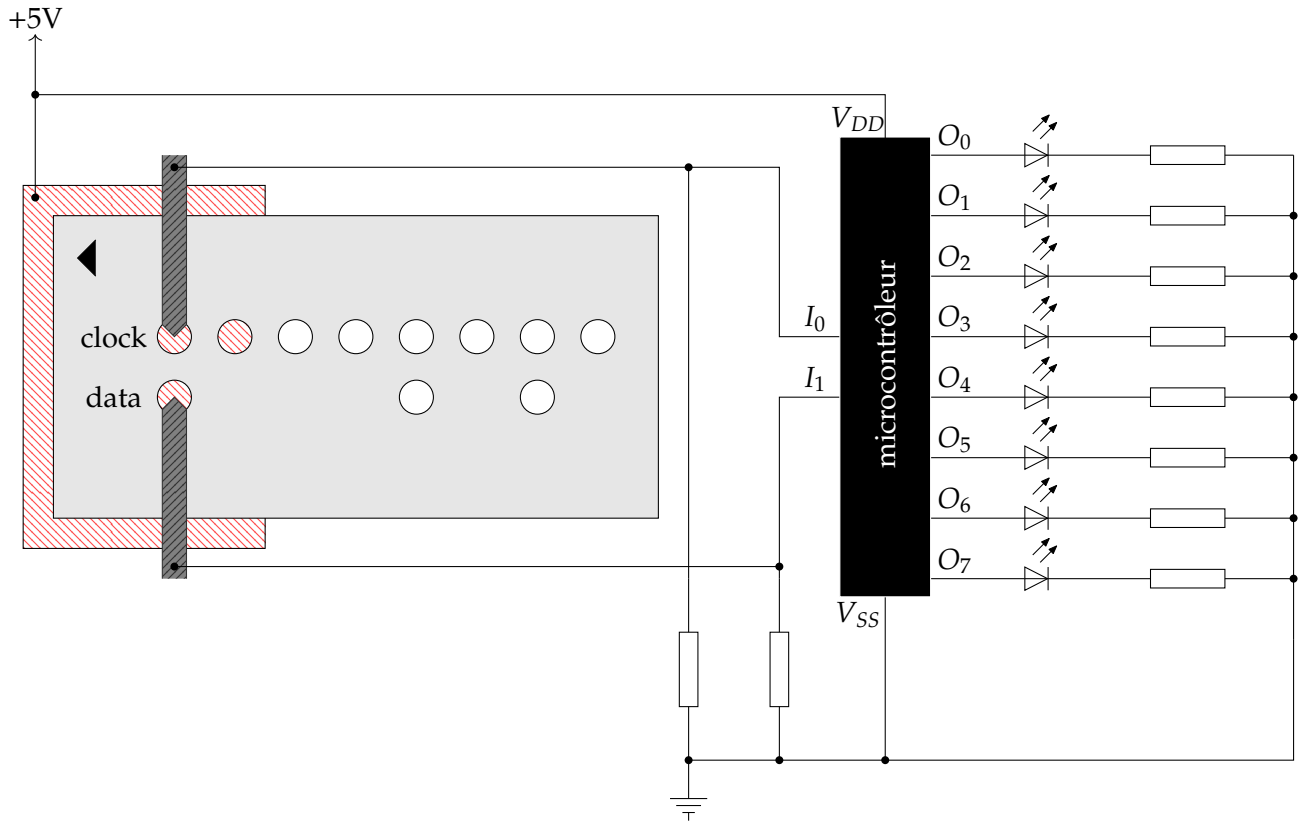



FIGURE 8.2 – Montage de lecteur de carte perforée

Grâce à ce nœud, nous définissons alors le nœud `read_bit` qui permet de lire un bit sur la carte dès lors que le signal d'horloge est vrai :

```
let%node read_bit (top,bot) ~return:(clk,data) =
  clk = edge top;
  data = bot [@when clk]
```

Le flot `top` correspond à l'état de la première ligne de la carte perforée, et le flot `bot` à l'état de la seconde ligne de cette dernière. Il est ici intéressant de noter que la représentation « physique » de l'horloge par des trous sur la carte perforée se traduit directement par la définition d'un signal d'horloge synchrone `clk` qui représente la fréquente d'échantillonnage du signal `data` de données.

Le programme devra détecter le fait que huit bits ont été lus. Pour ce faire, nous déclarons un nœud `count` qui permet de compter le nombre d'instants synchrones, de 0 à l'entier précédant une valeur de réinitialisation nommée `reset` :

```
let%node count (reset) ~return:(cpt) =
  cpt = (0 ⋙ (cpt + 1)) mod reset
```

À chaque fois que le signal d'horloge sera vrai, la valeur du signal des données sera enregistrée dans un tableau de 8 cases. Un tel tableau `t` est alors déclaré en OCaml :

```
let t = Array.make 8 false
```

Dans ce programme, une fois que huit bits ont été lus, l'octet correspondant sera affiché par huit LED branchées au microcontrôleur. On définit alors un nœud principal `read_card` qui représente la lecture complète de la carte perforée : une fois que 8 bits ont été lus, le flot `send` est mis à vrai, afin de signaler au programme OCaml qu'il peut mettre à jour l'état des LED :

```
let%node read_card (top,bot) ~return:(i,data,clk,send) =
  (clk,data) = read_bit (top,bot);
  i = count(8 [@when clk]);
  send = merge clk (i = 7) false
```

L'indice de la case du tableau correspondant au bit en cours de lecture est représenté par le flot `i`. Ce flot n'est présent (et sa valeur n'est incrémentée) que lorsque l'horloge `clk` est vraie.

Sur un microcontrôleur AVR ATmega2560 pour lequel les tiges métalliques seraient reliées aux broches 13 et 12, et les LED seraient connectées aux broches 42 à 49, le code OCaml responsable des entrées/sorties du programme est alors le suivant :

```
open Avr

let t = Array.make 8 false
let clk = PIN13
let data = PIN12
let leds = [| PIN42 ; PIN43; PIN44; PIN45; PIN46; PIN47; PIN48; PIN49 |]

let init () =
  (* réglage des broches en entrée *)
  pin_mode clk INPUT;
  pin_mode data INPUT;
  (* réglage des broches en sortie *)
  Array.iter (fun x -> pin_mode x OUTPUT) leds

let input_clk () = bool_of_level (digital_read clk)
let input_data () = bool_of_level (digital_read data)

(* fonction qui allume/éteint une des LED *)
let update_led i b = digital_write leds.(i) (level_of_bool b)

let output i data clk send =
  if clk then t.(i) ← data;
  if send then Array.iteri update_led t
```

Enfin, le code de la boucle principale du programme qui réalise l'interface entre ces fonctions et le programme synchrone OCaLustre est présenté ci-après :

```
let () =
 (* initialisation du matériel *)
 init ();
 (* création de l'état du noeud principal *)
 let st = read_card_alloc () in
 while true do
   (* lecture des entrées *)
   let c = input_clk () in
   let d = input_data () in
   (* mise à jour de l'état du noeud principal *)
   read_card_step st c d;
   (* émission des sorties *)
   let i = st.read_card_out_i in
   let data = st.read_card_out_data in
   let clk = st.read_card_out_clk in
   let send = st.read_card_out_send in
   output i data clk send
 done
```

Ce code est compatible avec le modèle de compilation d'OCaLustre destiné aux systèmes embarqués temps réel. Ainsi, lors de l'exécution de la boucle principale du programme, aucune nouvelle valeur OCaml n'est allouée sur le tas : le *garbage collector* n'est alors jamais activé lors de l'exécution du programme synchrone, et le temps d'exécution d'un instant synchrone peut alors être mesuré grâce à *Bytecrawler*. L'option `-m` suivie du nom du nœud principal permet de générer automatiquement le code de la boucle principale du programme, ainsi que les prototypes des fonctions qui permettent l'interaction entre le programme synchrone et son environnement. Nous illustrerons l'utilisation de cette option dans les exemples suivants.

### 8.1.3 Analyses statiques du programme

À partir des outils décrits précédemment dans ce manuscrit, nous pouvons réaliser plusieurs analyses statiques sur le code source du programme, ainsi que sur le bytecode qui lui est associé.

**Typage d'horloges :** Les types d'horloges induits par l'algorithme d'inférence d'horloges pour les nœuds du programme du lecteur de cartes perforées sont les suivants [2] :

```
edge :: (x:base) -> (e:base)
read_bit :: (top:base * bot:base) -> (clk:base * data:(base on clk))
count :: (reset:base) -> (cpt:base)
read_card :: (top:base * bot:base) -> (i:(base on clk) * data:(base on clk) * clk:base * send:base)
```

Ces types correspondent bien à ce qui est attendu : en particulier, les valeurs des flots `i` et `data` sont bien présentes uniquement quand le flot d'horloge `clk` est vrai. L'utilisation de l'opérateur de fusion

2. L'affichage des informations de typage des nœuds se fait grâce à l'option `-i` d'OCaLustre.

`merge` pour définir le flot `send` permet de sur-échantilloner la valeur `(cpt=7)` sur l'horloge de base. Ces informations de typage d'horloge sont à prendre en compte lors de la réalisation des fonctions d'interfaçage entre le programme synchrone et le programme OCaml : par exemple, il serait erroné que ces fonctions accèdent à la valeur du flot `i` lorsque le flot `clk` est faux.

**Temps d'exécution pire cas :** Sur un microcontrôleur AVR ATmega2560, le temps d'exécution pire cas d'un instant synchrone de ce programme, qui comprend les appels aux primitives d'entrée/sortie, est estimé par l'outil Bytecrawler à 57162 cycles.

Il est à noter néanmoins que l'utilisation de tableaux dans le programme implique des appels à des primitives C spécifiques pour la lecture et écriture dans des cases de ces tableaux. Ces primitives vérifient, par défaut, que les indices des cases lues ou écrites sont bien inférieurs à la taille du tableau, et lèvent une exception dans le cas contraire. Il n'est cependant pas possible, pour l'outil Bound-T (que nous utilisons également pour estimer le temps d'exécution des primitives), de borner le temps d'exécution d'une fonction pouvant lever une exception. Par conséquent, nous compilons le programme avec l'option `-unsafe` du compilateur *ocamlc*, afin de supprimer ces vérifications des bornes des tableaux, et de rendre alors possible l'estimation des temps d'exécution maximaux des primitives réalisant la lecture ou l'écriture dans un tableau.

Puisque le microcontrôleur utilisé a une fréquence de calcul de 16 MHz, notre lecteur de cartes perforées est ainsi capable de traiter un couple (*clock*, *data*) en environ 3.57 millisecondes. Cette courte durée permet d'assurer que l'utilisateur de carte perforée ne peut de toute évidence pas glisser une carte trop vite dans l'appareil, et fausser la lecture.

### 8.1.4 Consommation mémoire

La taille minimale de pile pour que ce programme s'exécute sans encombre est de 36 valeurs, tandis que celle du tas est de 318 valeurs. Ainsi, dans une configuration 16 bits de la machine virtuelle, il suffit de 879 octets de RAM pour que ce programme puisse s'exécuter. Son empreinte en mémoire flash est d'environ 12.1 kilo-octets, ce qui le rend compatible avec des microcontrôleurs dont les ressources mémoires sont encore plus faibles que le ATMega2560, comme le ATTiny1614, aux 16 kilo-octets de mémoire flash et 2048 octets de RAM, et dont le coût d'achat est de l'ordre de 60 centimes d'euros.

L'activation de l'évaluation anticipée qui consiste à pré-calculer les valeurs issues de l'initialisation des modules OCaml dès la compilation du programme permet d'en réduire d'autant plus la consommation en mémoire : il ne nécessite alors plus qu'une pile de 35 valeurs et un tas de 160 valeurs, pour une empreinte mémoire flash de 10.8 kilo-octets et une consommation de seulement 533 octets de RAM.

## 8.2 Une tempéreuse à chocolat

Pour notre second exemple d'application, nous décrivons le montage et le programme permettant de réaliser une tempéreuse à chocolat. Ce dispositif est un appareil de cuisine permettant de faire chauffer du chocolat à une température indiquée par un utilisateur. Notre tempéreuse est construite à partir d'un montage électronique assez peu onéreux. Dans ce montage, un microcontrôleur ATMega2560 est connecté aux composants électroniques suivants :

— Une résistance électrique chauffante qui convertit un signal électrique en chaleur.
— Une sonde de température qui convertit la température de son milieu en une valeur analogique.

— Un écran à cristaux liquides (LCD) à deux segments de type LCD1602A, souvent fourni dans les kit de développements pour microcontrôleurs.
— Deux boutons-poussoirs.

La figure 8.3 est une représentation schématisée de ce montage électronique.

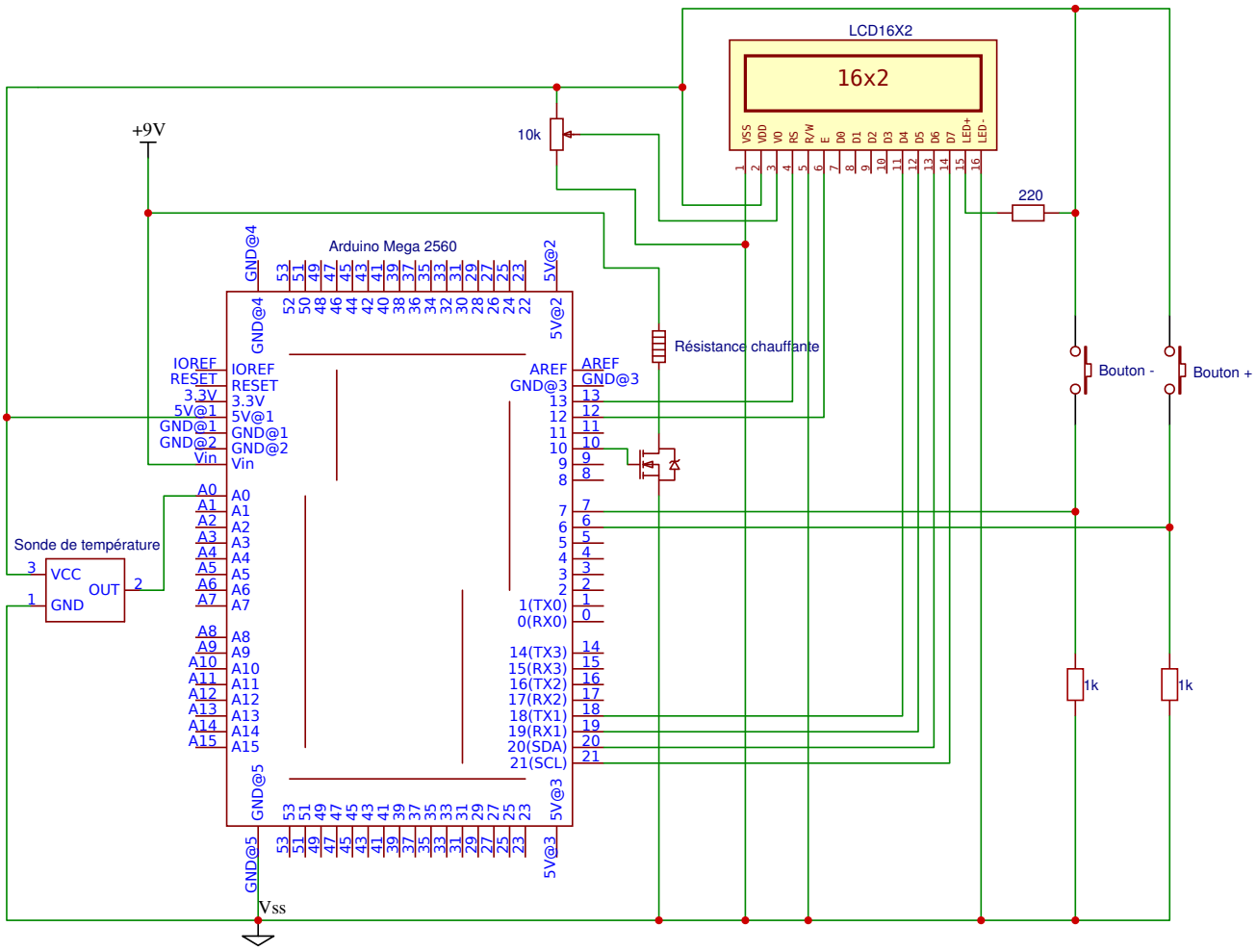



FIGURE 8.3 – Représentation schématique de la tempéreuse à chocolat

Les composants de ce circuit sont utilisés de la façon suivante : la résistance chauffante est placée dans un récipient rempli d'eau, et le chocolat est alors chauffé au bain marie par la tempéreuse qui contrôle la température de l'eau : dès lors que la température actuelle de la préparation est inférieure à la température désirée par l'utilisateur, la résistance chauffante est activée, permettant d'augmenter cette température. L'interface constituée de l'écran LCD des deux boutons poussoirs permet d'augmenter ou de diminuer la valeur de température désirée.

Le rôle du programme associé à ce montage est ainsi d'envoyer un signal électrique sur la broche reliée à la resistance en fonction de la valeur retournée par le capteur de température. La température désirée est calculée à partir des appuis successifs des boutons poussoirs. En parallèle, le programme affiche à tout instant la température actuelle de la préparation, et sa température désirée, sur l'écran LCD.

### 8.2.1 Programme OCaLustre

Le microcontrôleur au cœur de ce montage est chargé de l'exécution du programme en *synchronisant* les différents composants électroniques jouant un rôle dans le fonctionnement de la tempéreuse. Le modèle de programmation synchrone est alors particulièrement bien adapté à un tel programme embarqué : chaque nœud du programme synchrone représente un composant de l'application, qui doit s'exécuter en *même temps* que les autres, et leur interaction régit le fonctionnement du dispositif complet.

Le programme synchrone associé à ce montage est alors très simple : seuls trois nœuds OCaLustre sont nécessaires pour définir le comportement du programme :

```
(** on allume/éteint si on appuie en meme temps sur + et - **)
let%node thermo_on (p,m) ~return:(b) =
  b = (true ⋙ if p && m then not b else b)

(** modification de la température désirée selon le bouton appuyé **)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 ⋙ if p then w+5 else if m then w-5 else w)

(** noeud principal : calcul de la température désirée et de l'état de la résistance **)
(** Les températures sont en dixièmes de degrés Celsius **)
let%node thermo (plus,minus,real_temp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (plus,minus);
  wanted = set_wanted_temp (plus[@when on], minus[@when on]);
  real = real_temp [@when on];
  heat = (real < wanted);
  resistor = merge on heat false
```

Le programme manipule des valeurs de températures sous la forme de valeurs entières qui représentent des dixièmes de degrés Celsius. En particulier, le nœud principal reçoit la valeur des deux boutons poussoir et celle mesurée par la sonde de température, et se charge de produire cinq flots distincts :

— Le flot `on` permet de distinguer l'état actif du dispositif, de l'état de veille : la tempéreuse passe de l'un à l'autre dès que les deux boutons poussoir sont pressés en même temps. Ce flot est l'horloge qui régit la présence de la plupart des flots manipulés par le programme.

— Le flot `wanted` est défini comme le résultat de l'appel au nœud `set_wanted_temp` qui permet d'augmenter ou diminuer la température voulue (initialisée à 32.5 °C), avec une granularité de 0.5 °C, en fonction des boutons pressés par l'utilisateur de la tempéreuse. Ce flot n'est défini qu'à la condition que le flot `on` soit vrai.

— Le flot `real` est le résultat du sous-échantillonnage du flot `real_temp`, qui représente la valeur mesurée par la sonde de température, par l'horloge `on`.

— Le flot `heat` permet de définir la condition à partir de laquelle la résistance doit se mettre en chauffe : si la température réelle est inférieure à la température désirée, alors ce flot est *vrai*. Il est à noter qu'en raison d'une volonté de simplifier les exemples présentés dans ce chapitre, cette version de la tempéreuse est extrêmement naïve : l'inertie de la chauffe de la résistance peut entraîner une montée de la température à des valeurs au-delà de celle de la température voulue.

Un « vrai » dispositif tiendrait compte des différents aspects physiques de l'environnement du montage afin de contrôler plus finement les variations de température, par exemple en stoppant le chauffage dès que la préparation atteint une température proche de la température voulue, ou bien en calculant le rapport cyclique de chauffe nécessaire pour conserver une température stable. Néanmoins, ces modifications dépassent la vocation de ce chapitre, dont le but est d'illustrer par des exemples simples les applications possibles de la programmation synchrone avec OCaLustre. Un programme OCaLustre qui mesure le rapport cyclique de chauffe est toutefois donné en annexe D.2.3.

— Le flot `resistor` résulte de la fusion du flot précédent, et d'un flot constant calculant la valeur `false`. Ce flot permet de renseigner si la résistance chauffante doit être alimentée. Dans le cas où l'appareil est éteint, cette résistance doit être éteinte (`false`), et dans le cas où la tempéreuse est allumée, c'est la valeur du flot `heat` qui définit l'état de la résistance chauffante.

**Typage d'horloge et signature des nœuds** : Les types d'horloges des nœuds inférés automatiquement par OCaLustre lors de leur compilation sont les suivants :

```
thermo_on :: (p:base * m:base) -> (b:base)
set_wanted_temp :: (p:base * m:base) -> (w:base)
thermo :: (plus:base * minus:base * real_temp:base)
              -> (on:base * wanted:(base on on) * real:(base on on) * resistor:base)
```

En adéquation avec la description des différents flots définis par le programme, les flots `wanted` et `real` retournés par le nœud `thermo` sont bien cadencés par le flot `on` : leurs valeurs ne sont ainsi disponibles que lorsque la tempéreuse est active.

### 8.2.2 Boucle d'interaction

L'option `-m <node>` d'OCaLustre permet de générer le code de la machine d'exécution du programme, chargée d'appeler en boucle le nœud `<node>`. Ce nœud tiendra alors le rôle de nœud principal. Cette option génère un fichier `<node>_io.ml` pré-rempli par la signature des fonctions `init_<node>`, `input_<node>` et `output_<node>`, qui correspondent aux fonctions d'interaction entre le noyau synchrone du programme et le reste du code OCaml. De plus, l'option `-d <ms>` permet de cadencer la boucle principale, afin qu'une exécution d'un instant synchrone du programme soit réalisée toutes les `<ms>` millisecondes. Le programme de la tempéreuse à chocolat ne faisant pas intervenir de contraintes de vitesse importantes, une fréquence d'exécution de 1/500ms semble suffisante : le programme est ainsi compilé avec les options `-m thermo -d 500`.

Un fichier `thermo_io.ml` contenant les prototypes des fonctions permettant l'interaction entre la partie synchrone du programme et son environnement est alors généré par OCaLustre. Une fois complété, le code de ces fonctions est le suivant :

```
(*** Fonctions d'interfaçage de l'instant synchrone ***)

(** fonction d'initialisation **)
let init_thermo () =
  (* initialisation de la lecture analogique *)
  Avr.adc_init ();
  (* initialisation de l'écran *)
  LiquidCrystal.lcdBegin lcd 16 2;
  (* initialisation des broches *)
  pin_mode sensor INPUT;
  pin_mode resistor OUTPUT;
  pin_mode plus INPUT;
  pin_mode minus INPUT

(** fonction d'entrée **)
let input_thermo () =
  let plus = digital_read plus in
  let minus = digital_read minus in
  let plus = bool_of_level plus in
  let minus = bool_of_level minus in
  let real_temp = read_temp () in
  (plus,minus,real_temp)

(** fonction de sortie **)
let output_thermo (on,wanted,real,res) =
  if on then
    begin
      print_temp wanted real;
      digital_write resistor (if res then HIGH else LOW)
    end
  else idle ()
```

Ces dernières font usage de fonctions auxiliaires OCaml principalement responsables de la conversion en degrés Celsius de la température mesurée par la sonde électronique, ainsi que de leur affichage sur l'écran LCD. Le code de ces fonctions, ainsi que des déclarations des différentes variables utilisées par le programme, est le suivant :

```
open Avr

(* déclaration de l'écran *)
let lcd = LiquidCrystal.create4bitmode PIN13 PIN12 PIN18 PIN19 PIN20 PIN21

(* déclaration des pins *)
let plus = PIN7
let minus = PIN6
let resistor = PIN10
let sensor = PINA0

(* conversion de température *)
let convert_temp t =
  let f = (float_of_int (1033 - t) /. 11.67) in
  int_of_float (f*.100.)

(* lecture de la température *)
let read_temp () =
  let t = analog_read sensor in
  convert_temp t

(* Affichage des températures sur l'écran LCD *)
let print_temp wanted real =
  let split_temp t =
    let u = t/10 in
    let dec = t mod 10 in
    (u,dec) in
  LiquidCrystal.home lcd;
  let (wu,wd) = split_temp wanted in
  let (ru,rd) = split_temp real in
  LiquidCrystal.print lcd "Wanted T :";
  LiquidCrystal.print lcd ((string_of_int wu)^"."^(string_of_int wd));
  LiquidCrystal.setCursor lcd 0 1;
  LiquidCrystal.print lcd "Actual T :";
  LiquidCrystal.print lcd ((string_of_int ru)^"."^(string_of_int rd))

let idle () =
  LiquidCrystal.home lcd; LiquidCrystal.clear lcd; LiquidCrystal.print lcd "..."
```

### 8.2.3 Consommation mémoire

Le programme de la tempéreuse nécessite au minimum une pile de 61 valeurs et un tas de 448 valeurs, ce qui revient, dans une représentation 16 bits des valeurs OCaml, à une empreinte mémoire totale de 17.3 kilo-octets de mémoire flash et seulement 1277 octets de RAM (avec le GC Stop and Copy [3]). Néanmoins, en raison du fait que l'activation fréquente de l'algorithme de GC ralentit l'exécution du programme

3. Il est à noter qu'en raison de l'utilisation de cet algorithme de GC, le nombre de valeurs OCaml qu'il est réellement possible d'allouer correspond à la moitié de la taille du tas.

(comme nous l'avons illustré au chapitre précédent), il serait regrettable de ne pas bénéficier de l'étendue de la RAM du microcontrôleur utilisé. Un tas de 3800 mots permet ainsi de remplir la quasi-totalité des 8 kilo-octets de RAM du microcontrôleur ATMega2560, en réduisant le nombre de déclenchements de l'algorithme de GC pendant les cent premiers instants synchrones de 699 à seulement 4.

### 8.2.4 Simulation

Le simulateur intégré à OMicroB permet d'exécuter ce programme directement sur ordinateur, principalement dans le but de faciliter le débogage du programme. La figure 8.4 représente l'interface du simulateur pour ce programme : l'état de la resistance chauffante est ici représenté par une LED située entre les deux boutons + et - qui permettent de régler la température. La barre horizontale située en dessous de ces éléments permet de choisir la valeur mesurée sur la broche d'entrée analogique à laquelle est en réalité connectée la sonde de température.

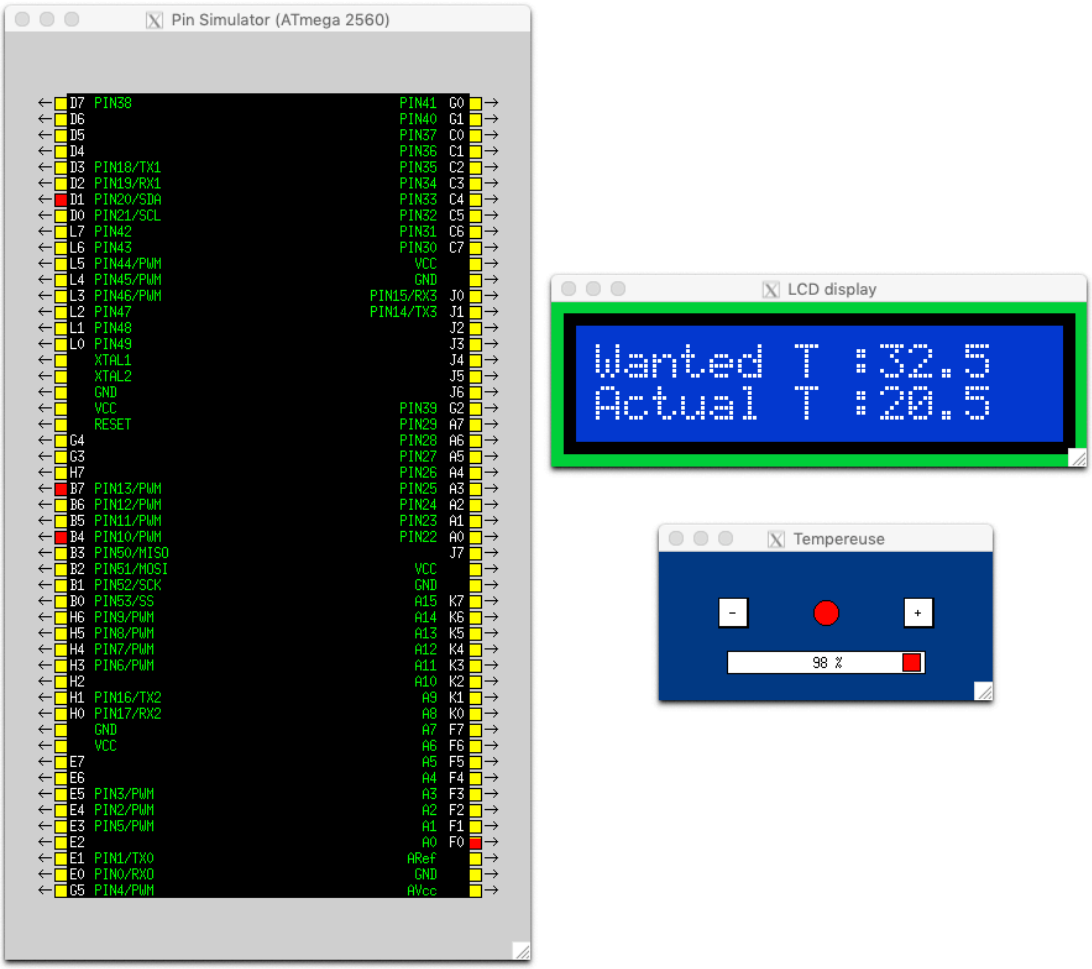



FIGURE 8.4 – Exécution du programme de la tempéreuse dans le simulateur

## 8.3 Un jeu de Serpent

La dernière application présentée dans ce manuscrit est un petit jeu vidéo de « Serpent » dans lequel un serpent, représenté par une suite de cases contiguës, se déplace dans un monde en deux dimensions

afin d'y manger des pommes, qui apparaîssent au fur et à mesure de l'évolution du jeu. Dès que le serpent mange une pomme, il grandit d'une case, et une nouvelle pomme apparaît sur l'écran. Le but du jeu est alors pour le serpent de manger le maximum de pommes afin d'atteindre une certaine taille, tout en évitant d'entrer en collision avec lui même, ce qui le tue (et fait alors perdre le jeu).

Ce jeu vidéo était très populaire à la fin des années 1990 et au début des années 2000, lors de la popularisation du téléphone portable : les téléphones de l'époque, dont les vitesses de calcul étaient de l'ordre de celles des microcontrôleurs étudiés dans cette thèse[4], pouvaient en effet exécuter quelques jeux vidéo simples, adaptés aux performances limitées des processeurs mobiles de l'époque. De ce fait, nous détaillons dans cette section une implémentation de ce jeu adaptée à un petit appareil dont les ressources matérielles sont elles aussi très limitées. Cet appareil, nommé *Arduboy*, est une petite console de jeu portative au format carte de crédit, destinée au développement de petits jeux souvent issus de la mouvance *retrogaming*. Pour des considérations de performances ainsi qu'une utilisation précise des ressources de l'appareil, les programmes développés par la communauté Arduboy sont généralement écrits en C. Nous démontrons pourtant, par cet exemple, qu'il est tout à fait envisageable d'y développer une application avec le couple OMicroB/OCaLustre. Les solutions logicielles présentées dans cette thèse permettent de ce fait l'utilisation d'abstractions de haut niveau pour du matériel sur lequel les ressources mémoires sont très restreintes. La figure 8.5 représente un Arduboy qui exécute le jeu de Serpent.


FIGURE 8.5 – Un Arduboy

L'intégralité des codes sources des modules présentés dans cette section est accessible dans l'annexe D.3.

### 8.3.1 Anatomie d'un Arduboy

Le composant principal d'un Arduboy est un microcontrôleur de la famille AVR : le ATmega32u4. Ce microcontrôleur fait partie de ce que nous considérons être du matériel fortement limité : sa puissance de calcul pourrait théoriquement atteindre 16 MHz, mais étant donné que la batterie de l'Arduboy possède un voltage assez faible (entre 3.4V et 3.7V) les performances réelles de ce microcontrôleurs sont plus de

4. Par exemple, le Nokia 3310 commercialisé en 2000 possédait un processeur Texas Instrument MAD2WD1 de 13 MHz.

l'ordre de 11 MHz à 12 MHz. De plus, le ATmega32u4 possède 32 kio de mémoire flash, et seulement 2.5 kio de RAM, ce qui impose aux programmes réalisés d'avoir une empreinte mémoire très faible.

Un Arduboy contient également plusieurs composants électroniques destinés en majorité à l'interaction utilisateur : il possède en effet un ensemble de 6 boutons poussoirs (quatre qui permettent de représenter des flèches de direction – haut, bas, gauche, et droite – et deux boutons d'action – A et B), un écran matriciel monochrome à technologie OLED (*Organic Light-Emitting Diode*) d'une résolution de $128 \times 64$ pixels, trois LED (une rouge, une verte, et une bleue) positionnées à côté de l'écran, ainsi qu'un haut-parleur piézoélectrique permetant de jouer des sons de basse définition. Pour notre application, nous ferons usage de l'écran afin d'afficher le serpent et la pomme à consommer, des boutons de direction gauche et droite pour déplacer le serpent, ainsi que des LED pour indiquer au joueur si le jeu est gagné (vert) ou perdu (rouge).

Le schéma électronique représentant les composants de l'Arduboy utilisés par notre programme est donné dans la figure 8.6.

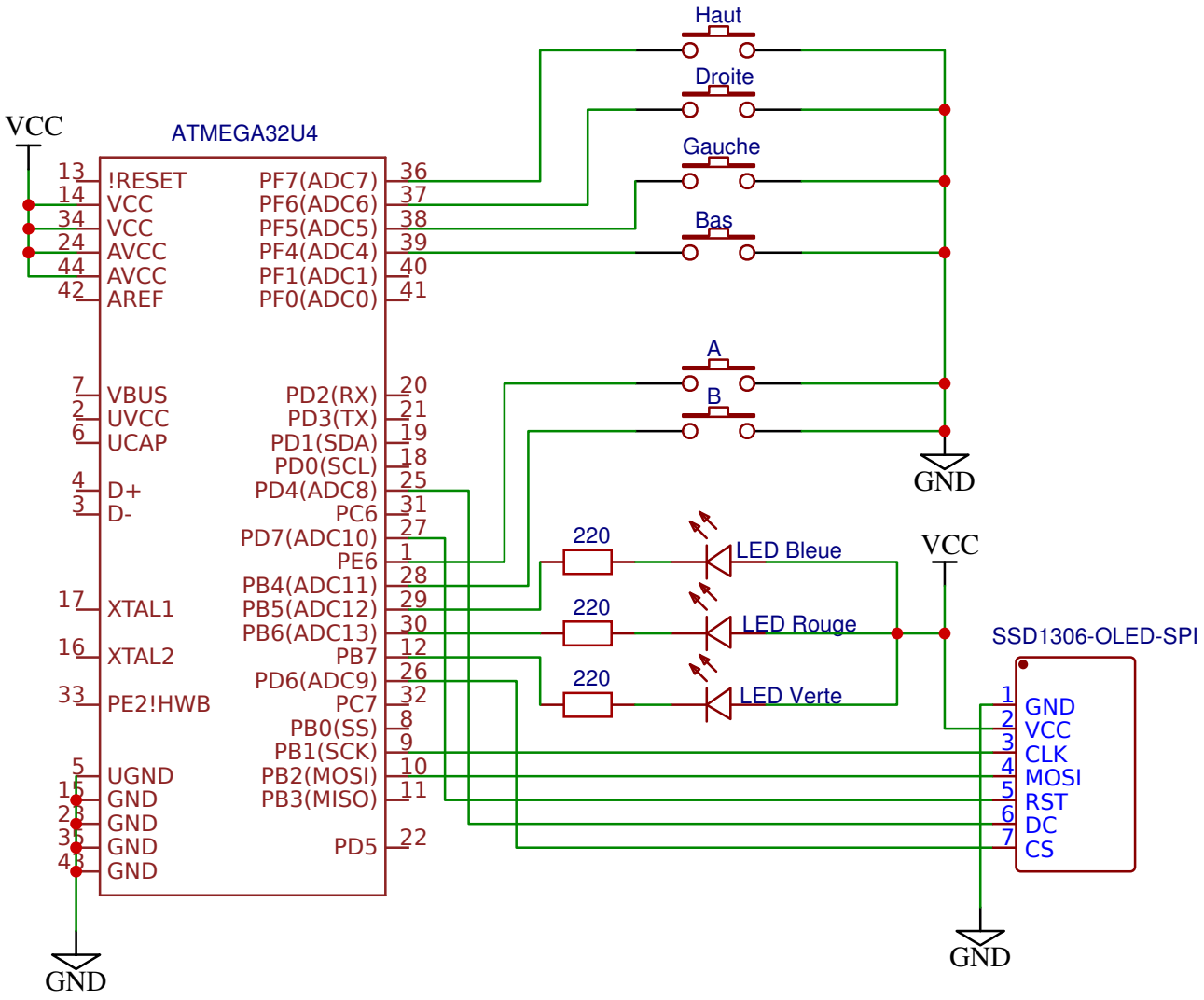



FIGURE 8.6 – Composants de l'Arduboy utilisés par le jeu de Serpent

### 8.3.2 Programme du montage électrique

Le programme réalisé tient compte des spécificités techniques de l'Arduboy : ainsi, nous décrivons en premier lieu le code des modules OCaml permettant l'interaction entre le microcontrôleur et les composants d'entrée/sortie, nécessaire au bon fonctionnement du jeu.

**Liaison SPI et connexion à l'écran OLED** : Dans un Arduboy, le transfert de données entre le microcontrôleur et l'écran OLED (de type SSD 1309) est réalisé via une interface SPI (*Serial Peripheral Interface*) dont sont utilisés deux fils :

— Un fil est relié à la broche SCK (*Serial Clock*) du microcontrôleur et représente le signal d'horloge qui permet de cadencer la communication.
— Un fil est relié à la broche MOSI (*Master Output, Slave Input*) du microcontrôleur et représente le signal de commandes ou de données envoyées à l'écran.

La liaison SPI est implantée de façon matérielle par le microcontrôleur : il suffit de l'initialiser correctement [5] pour qu'ensuite tout octet placé dans le registre `SPDR` soit émis par le microcontrôleur. Le code OCaml qui permet de configurer et d'interagir avec la liaison SPI est représenté par un module *Spi* dont le code est présenté dans la figure 8.7. Ce module fait usage du module *Avr* fourni dans la bibliothèque standard d'OMicroB, qui définit les registres et les broches du microcontrôleur, ainsi que des fonctions permettant d'en changer l'état.

```
(*** Module de gestion de la connexion SPI (Serial Peripheral Interface) ***)

open Avr

(** Initialiser la connexion SPI **)
let begin_spi ~sck ~mosi =
  set_bit SPCR MSTR;
  set_bit SPCR SPE;
  set_bit SPSR SPI2x;
  pin_mode sck OUTPUT;
  pin_mode mosi OUTPUT

(** Arrêter la connexion SPI **)
let end_spi () = clear_bit SPCR SPE

(*** Émettre des données via la connexion SPI **)
let transfer data = write_register SPDR data
```

FIGURE 8.7 – Module `spi.ml` permettant l'interaction avec un périphérique série

De plus, les signaux permettant de contrôler l'écran sont portés par trois fils distincts :

— Un fil relié à la broche D/C (*Data/Command*) de l'écran permet de le positionner en mode « commande » ou en mode « données ».
— Un fil relié à la broche CS (*Chip Select*) de l'écran qui ne permet la communication entre l'écran et le microcontrôleur qu'à condition qu'elle soit au niveau bas (*LOW*).

5. Les détails techniques qui consistent à configurer la laison SPI en modifiant la valeur de certains bits de plusieurs registres sortent du contexte de ce manuscrit.

— Un fil relié à la broche RST (*Reset*) de l'écran qui permet de réinitialiser le microcontrôleur : si cette broche passe du niveau haut (*HIGH*) au niveau bas (*LOW*), l'écran se réinitialise.

De surcroît, afin d'illustrer l'interopératibilité entre OCaml et le langage C, les données de l'écran seront représentées dans un tableau C d'octets (dans lequel chaque octet représente 8 points sur l'écran) faisant office de mémoire tampon. Il est à noter qu'afin de réduire l'empreinte mémoire du programme et de simplifier la lisibilité du jeu, la résolution d'écran réellement utilisée par le Serpent sera de $64 \times 32$ points. L'affichage prendra néanmoins la totalité de la largeur et de la hauteur de l'écran : les « points » du jeu auront ainsi une taille de $2 \times 2$ pixels sur l'écran.

Le module OCaml *Oled*, dont un extrait est présenté dans la figure 8.8, contient les fonctions nécessaires pour la manipulation de l'écran et la manipulation de la mémoire tampon. En particulier, une fonction `Oled.flush` permet de transférer à l'écran une nouvelle image représentant l'état du jeu. Les détails des caractéristiques de l'écran et de sa configuration sont disponibles dans la fiche technique de l'écran SSD 1306 [⚓4].

```
(*** Écrire un point (true=noir/false=blanc) ***)
let draw x y color = write_buffer x y color

(*** Effacer l'écran ***)
let clear () =
 for _i = 0 to 1023 do
   Spi.transfer(0x00)
 done

(*** Envoyer le buffer sur l'écran ***)
let flush () =
  for _i = 0 to 1023 do
    Spi.transfer(get_byte_buffer())
  done

(*** Initialisation de l'écran ***)
let boot ~cs ~dc ~rst =
  digital_write rst HIGH;
  digital_write rst LOW;
  digital_write rst HIGH;
  command_mode cs dc;
  transfer_program boot_program;
  data_mode cs dc;
  clear()
```

FIGURE 8.8 – Module `oled.ml` (extrait) permettant de configurer et contrôler l'écran SSD 1306

**Configuration de l'Arduboy :** Enfin, un module Arduboy (fig. 8.9) permet de configurer les composants de ce dernier via des fonctions de la bibliothèque standard. Les numéros des broches utilisés correspondent à la convention de nommage de la bibliothèque Arduino (en l'occurence, celle de l'Arduino Leonardo qui renferme également un microcontrôleur ATMega32u4).

— La fonction `pin_mode` permet de définir une broche comme une entrée (`INPUT`), une sortie (`OUTPUT`), ou une entrée reliée par une résistance *pull-up*. Les broches d'un AVR possèdent une résistance *pull-up* interne, ce qui a pour incidence que la valeur mesurée sur une broche est au niveau

```
open Avr

(** Les broches utilisées **)
let cs = PIN12
let dc = PIN4
let rst = PIN6
let button_left = PINA2
let button_right = PINA1
let button_down = PINA3
let button_up = PINA0
let button_a = PIN7
let button_b = PIN8
let blue = PIN9
let red = PIN10
let green = PIN11

(** initialisation des LED rgb à anode commune (HIGH = éteinte) **)
let init_led l =
  digital_write l HIGH

(** allumer une des LED rgb à anode commune (LOW = allumé) **)
let light_led l =
  digital_write l LOW

(** initialisation des broches **)
let boot_pins () =
  List.iter (fun x -> pin_mode x INPUT_PULLUP) [button_left; button_right; button_up;
                                                button_down];
  pin_mode button_a INPUT_PULLUP;
  pin_mode button_b INPUT_PULLUP;
  List.iter (fun x -> pin_mode x OUTPUT) [red;green;blue];
  List.iter (fun x -> pin_mode x OUTPUT) [cs;dc;rst];
  List.iter init_led [red;green;blue]

(** initialisation des broches, de la liaison SPI, et de l'écran **)
let init () =
  boot_pins ();
  Spi.begin_spi ~sck:SCK ~mosi:MOSI;
  Oled.boot ~cs:cs ~dc:dc ~rst:rst
```

FIGURE 8.9 – Module `Arduboy.ml` : code de configuration et initialisation du matériel

haut (*HIGH*) lorsqu'elle n'est connectée à aucun composant. Par conséquent, quand les boutons-poussoirs de l'Arduboy sont ouverts la valeur lue est *HIGH*, et quand ils sont fermés la valeur lue est *LOW* car ils sont reliés à la masse du circuit.

— La fonction `digital_write` permet d'écrire une valeur binaire (niveau haut ou niveau bas) sur une broche.

Ce module, dont l'implantation suit le schéma électronique représentant les connexions entre le microcontrôleur et les composants externes, permet de configurer l'ensemble du matériel nécessaire à cette application : en particulier, la fonction `init` permet de configurer les broches utilisées par le programme, d'activer la liaison SPI, et d'initialiser l'écran.

Les différents modules présentés dans cette sous-section implantent en réalité une *bibliothèque* permettant de contrôler le montage interne de l'Arduboy. Cette bibliothèque peut être utilisé pour l'implémentation de notre jeu de Serpent, mais elle peut également être réutilisée pour de nombreux autres projets qui visent ce matériel. En effet, l'ensemble des fonctionnalités présentées dans cette partie traduisent simplement les caractéristiques du montage électronique et les relations entre ses différents composants, qui ne changent pas d'un programme à l'autre. Cette bibliothèque permet d'illustrer le fait que l'utilisation d'un langage de haut niveau comme OCaml est également adaptée à la définition d'interactions de bas niveau, comme il est ici le cas des interactions électroniques entre le microcontrôleur ATmega32u4 et les composants auxquels il est connecté.

### 8.3.3 Programme du jeu

À partir de la bibliothèque décrite dans la section précédente, nous abordons désormais le détail de l'implantation du jeu de Serpent. Ce jeu a la structure d'un programme synchrone qui utilise à la fois un noyau OCaLustre permettant de représenter la réaction aux actions du joueur, ainsi que des fonctions OCaml dont le rôle et de réaliser majoritairement la communication entre ce noyau et les périphériques externes comme l'écran ou les flèches directionnelles.

Dans ce programme, le serpent est un tableau `snake` dont chaque case représente les positions des diverses sections du corps du serpent :

```
let max_size = 15
let snake = Array.make max_size (0,0)
```

Deux valeurs `head` et `tail` représentent l'indice de la case du tableau qui correspond respectivement à la tête et à la queue du serpent. Pour représenter le déplacement du serpent, à chaque instant du programme ces deux pointeurs avanceront chacun vers la case d'indice immédiatement supérieur, et la position de la nouvelle tête du serpent sera inscrite dans la nouvelle case pointée par `head`. Comme l'illustre la figure 8.10, le tableau aura une structure circulaire : lorsque l'un ou l'autre de ces pointeurs sortira du tableau après s'être déplacé, il sera repositionné à l'indice 0.

**Noyau synchrone :** La boucle du jeu sera chargée de calculer la direction du serpent en fonction de l'appui sur le bouton droit ou gauche, de calculer la nouvelle position de sa tête, ainsi que de prendre en compte le fait que le serpent grandit dès que sa tête se retrouve à la même position que la pomme (autrement dit : dès qu'il mange la pomme). Le code de cette boucle est un nœud OCaLustre qui reçoit

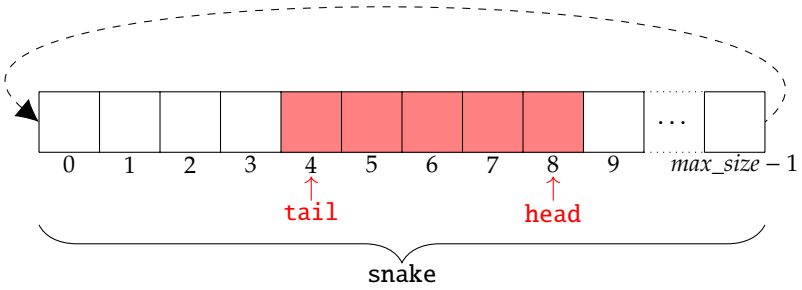



FIGURE 8.10 – Structure du serpent

en entrée la taille maximale du serpent, l'état des boutons gauche et droit, ainsi que les dimensions du monde. Il calcule la nouvelle valeur des pointeurs `head` et `tail`, la nouvelle position de la tête du serpent `(new_x,new_y)`, la position de la pomme `(apple_x,apple_y)`, et indique si le joueur a gagné (`win`) :

```
let%node game_loop (max_size,left,right,width,height)
     ~return:(head,tail,new_x,new_y,apple_x,apple_y,win) =
  (new_x,new_y) = new_head (dir,width,height);
  dir = direction(left,right);
  head = (1 ⋙((head+1) mod max_size));
  eats = (apple_x = new_x && apple_y = new_y);
  (a_x,a_y) = new_apple (width [@when eats], height [@when eats]);
  apple_x = (10 ⋙merge eats a_x (apple_x [@whennot eats]));
  apple_y = (10 ⋙merge eats a_y (apple_y [@whennot eats]));
  tail = merge eats ((0⋙tail)[@when eats]) (0 ⋙((tail+1) mod max_size)[@whennot eats]);
  size = (1 ⋙merge eats ((size + 1) [@when eats]) (size [@whennot eats]));
  win = (size = max_size - 1)
```

Ce nœud définit une horloge `eats`, qui est vraie dès que le serpent mange la pomme. Cette horloge régit l'exécution de plusieurs calculs :

— Quand `eats` est vrai, une nouvelle position `(a_x,a_y)` pour la pomme est calculée par un nœud `new_apple` :

```
(** Tirage aléatoire d'un entier **)
let new_position n = Random.int n

(** Nouvelle position de la pomme **)
let%node new_apple (width,height) ~return:(a_x,a_y) =
  (a_x,a_y) = (call new_position width, call new_position height)
```

— Quand `eats` est vrai, la taille `size` du serpent augmente.

— Si `eats` est vrai, le pointeur vers la queue du serpent (`tail`) n'est pas décalé, ce qui permet de simuler l'agrandissement du serpent.

La boucle de jeu fait également appel à un nœud `direction` pour calculer la direction du serpent :

```
let left_of = function South -> East | North -> West | East -> North | West -> South
let right_of = function South -> West | North -> East | East -> South | West -> North

(** Tourner à gauche **)
let%node left (dir) ~return:ndir =
  ndir = call left_of dir

(** Tourner à droite **)
let%node right (dir) ~return:ndir =
  ndir = call right_of dir

(** Direction du serpent à partir de sa direction précédente **)
let%node direction (l,r) ~return:dir =
  pre_dir = (South ⋙ dir);
  dir = if l then
          (merge l (left (pre_dir [@when l])) (pre_dir [@whennot l]))
        else
          (merge r (right (pre_dir [@when r])) (pre_dir [@whennot r]))
```

En particulier, l'utilisation de l'opérateur ⋙ permet de définir le flot `dir` en fonction de sa valeur précédente : ainsi, au début du programme le serpent se déplace vers le sud puis, lorsque le bouton gauche (resp. droit) est appuyé, il tourne à gauche (resp. droite) par rapport à sa position précédente. Le serpent conserve la même direction si aucun bouton n'est appuyé.

Le nœud `game_loop` fait également appel à un nœud `new_head` qui calcule la nouvelle position de la tête :

```
(** Modulo **)
let nmod x y = (x + y) mod y

(** Nouvelle cordonnée **)
let%node new_coord (dir,max,v,dir1,dir2) ~return:n =
  n = if dir = dir1 then call nmod (v-1) max
      else if dir = dir2 then call nmod (v+1) max
      else v

(** Nouvelle position de la tête du serpent **)
let%node new_head (dir,w,h) ~return:(x,y) =
  x = new_coord (dir,w,0 ⋙ x,West,East);
  y = new_coord (dir,h,0 ⋙ y,North,South)
```

Enfin, le nœud principal `main` du programme OCaLustre se charge de détecter un front montant sur les deux boutons permettant de faire tourner à gauche ou à droite le serpent, et fait appel au nœud implantant la boucle de jeu :

```
(** Front montant (pour les boutons) **)
let%node rising_edge i ~return:o =
  o = (i && (not (false ⋙ i)))

(** Noeud principal **)
  let%node main (max_size,button1,button2,width,height)
    ~return:(head,tail,nx,ny,apple_x,apple_y,win) =
  left = rising_edge (button1);
  right = rising_edge (button2);
  (head,tail,nx,ny,apple_x,apple_y,win) = game_loop(max_size,left,right,width,height)
```

**Boucle principale et fonctions d'interaction :** De façon identique à l'exemple précédent, il est fait usage de l'option `-m` d'OCaLustre afin de générer le code de la machine d'exécution du programme, chargée d'exécuter en boucle son nœud principal.

Une fois l'option `-m main` renseignée, le fichier `main_io.ml` est généré par OCaLustre avec les prototypes des fonctions d'interaction. Après complétion, ce code contient essentiellement les fonctions nécessaires à l'affichage du jeu, ainsi qu'à la lecture des valeurs des boutons de l'Arduboy. Néanmoins, en raison du fait que la version actuelle d'OCaLustre ne gère pas les tableaux, la fonction permettant de vérifier si le serpent n'entre pas en collision avec lui-même (nommée `eats_itself`) est également définie dans ce fichier, directement en OCaml.

### 8.3.4 Consommation mémoire

Dans une configuration 16 bits de la machine virtuelle, et avec le garbage collector Stop and Copy, le programme du jeu de serpent peut s'exécuter avec une pile d'une taille de 60 valeurs au minimum, et d'un tas d'au moins 744 valeurs, pour une empreinte en mémoire flash de 17.1 kilo-octets et de 2204 octets de RAM, ce qui s'approche grandement des 2.5 kilo-octets de RAM disponibles sur le microcontrôleur. L'utilisation de l'évaluation anticipée permet néanmoins de réduire la taille de la pile à 58 valeurs, et l'emploi du garbage collector *Mark and Compact* permet de doubler la zone d'allocation mémoire (et ainsi de réduire la fréquence des appels au GC) sans augmenter la taille de la RAM. Sur les cent premiers instants synchrones du programme, le nombre de déclenchements de l'algorithme de nettoyage mémoire passe ainsi de 49 avec le GC *Stop and Copy* à 18 avec le GC *Mark and Compact*.

## Conclusion du chapitre

Les exemples présentés dans ce chapitre ont démontré la possibilité d'exécuter des programmes simples, mais néanmoins complets, sur des appareils dotés de ressources très limitées. Ces exemples mettent également en exergue les avantages liés à l'utilisation de paradigmes de programmation de haut niveau. Les programmes réalisés profitent des garanties apportées par l'utilisation de langages de haut niveau (estimation du temps d'exécution pire cas, typage, détection de boucle de causalité) tout en étant exécutables sur du matériel dont les ressources sont très limitées. Ainsi, un microcontrôleur dont la RAM est inférieure à 2.5 kilo-octets est capable d'exécuter l'ensemble des programmes décrits dans ce chapitre. Cette très faible empreinte des programmes OCaLustre embarquant la machine virtuelle OMicroB permet d'envisager sereinement l'exécution sur du matériel mieux équipé en ressources mémoires (comme les microcontrôleurs basés sur des processeurs ARM Cortex-M0, dotés de ressources mémoires qui peuvent atteindre plusieurs centaines de kilo-octets de mémoire flash et plusieurs dizaines de kilo-octets de RAM), pour la réalisation de programmes encore plus riches et complexes, bénéficiant de composants avancés comme par exemple des modules Bluetooth, des accéléromètres, ou des écrans tactiles multicolores.

# Conclusion et perspectives

L'objectif de cette thèse était de proposer un ensemble de solutions permettant de programmer des microcontrôleurs dans des langages de haut niveau, afin de démontrer qu'il est tout à fait envisageable de bénéficier des avantages de modèles de programmation plus riches tout en respectant les contraintes d'un tel matériel. Pour ce faire, nous avons décrit dans ce manuscrit une suite de solutions logicielles et d'approches formelles qui permettent une montée progressive en abstraction, apportant à chaque niveau de cette succession d'abstractions de nouvelles garanties sur les programmes réalisés. Dans cette conclusion, nous rappelons les diverses contributions apportées par cette thèse, à l'aune de la montée en abstraction qu'elles illustrent, et des nouvelles garanties qu'elles apportent. Nous discutons par la suite des diverses perspectives et travaux futurs qui seraient appropriés pour la poursuite de notre approche consistant à améliorer la sûreté et l'expressivité de la programmation de microcontrôleurs.

## Contributions

La figure 9.1 résume schématiquement le positionnement de chaque chapitre de ce manuscrit par rapport au gain en abstraction conféré par les différentes approches présentées dans cette thèse.

La première contribution de cette thèse a reposé sur une *approche machine virtuelle* permettant l'exécution de langages multiparadigmes sur du matériel varié. Nous avons alors proposé *OMicroB*, une machine virtuelle pour le langage OCaml, concue avec l'objectif d'être portée sur de nombreux modèles de microcontrôleurs, tout en bénéficiant d'une empreinte mémoire limitée. Configurable et optimisée, cette machine virtuelle a pu être exécutée sur des microcontrôleurs dont les ressources mémoires ne dépassent pas 8 kilo-octets de RAM, et quelques dizaines de kilo-octets de mémoire flash. De surcroît, les performances en matière de vitesse d'OMicroB sont assez prometteuses, puisqu'elles sont plutôt proches de celles de la machine virtuelle OCaml standard, qui ne profite pas des mêmes avantages au niveau de la réduction de l'occupation mémoire. OMicroB offre également une meilleure approche pour le débogage des programmes, d'abord par le fait que sa totale compatibilité avec le bytecode généré par le compilateur standard *ocamlc* permet de profiter des outils d'analyse et de débogage d'OCaml classiques (comme par exemple le débogueur *ocamldebug*), mais également par son simulateur qui permet de représenter les composants d'un circuit électronique auxquels sera relié le microcontrôleur à programmer, et de tester leurs interactions, directement depuis l'ordinateur sur lequel est développée l'application. De plus, la sûreté du typage vérifié statiquement par le compilateur du langage OCaml permet de réduire fortement le nombre d'erreurs potentielles à l'exécution du programme. De par la portabilité de son bytecode, l'expressivité du langage, ainsi que l'utilisation de mécanismes de gestion mémoire automatiques, l'utilisation d'OMicroB offre ainsi une abstraction du matériel sur lequel sera exécuté le programme, permettant de considérer sereinement le déploiement d'un même programme sur plusieurs appareils différents. Nos travaux relatifs à l'approche machine virtuelle ainsi qu'à la réalisation d'OMicroB ont été

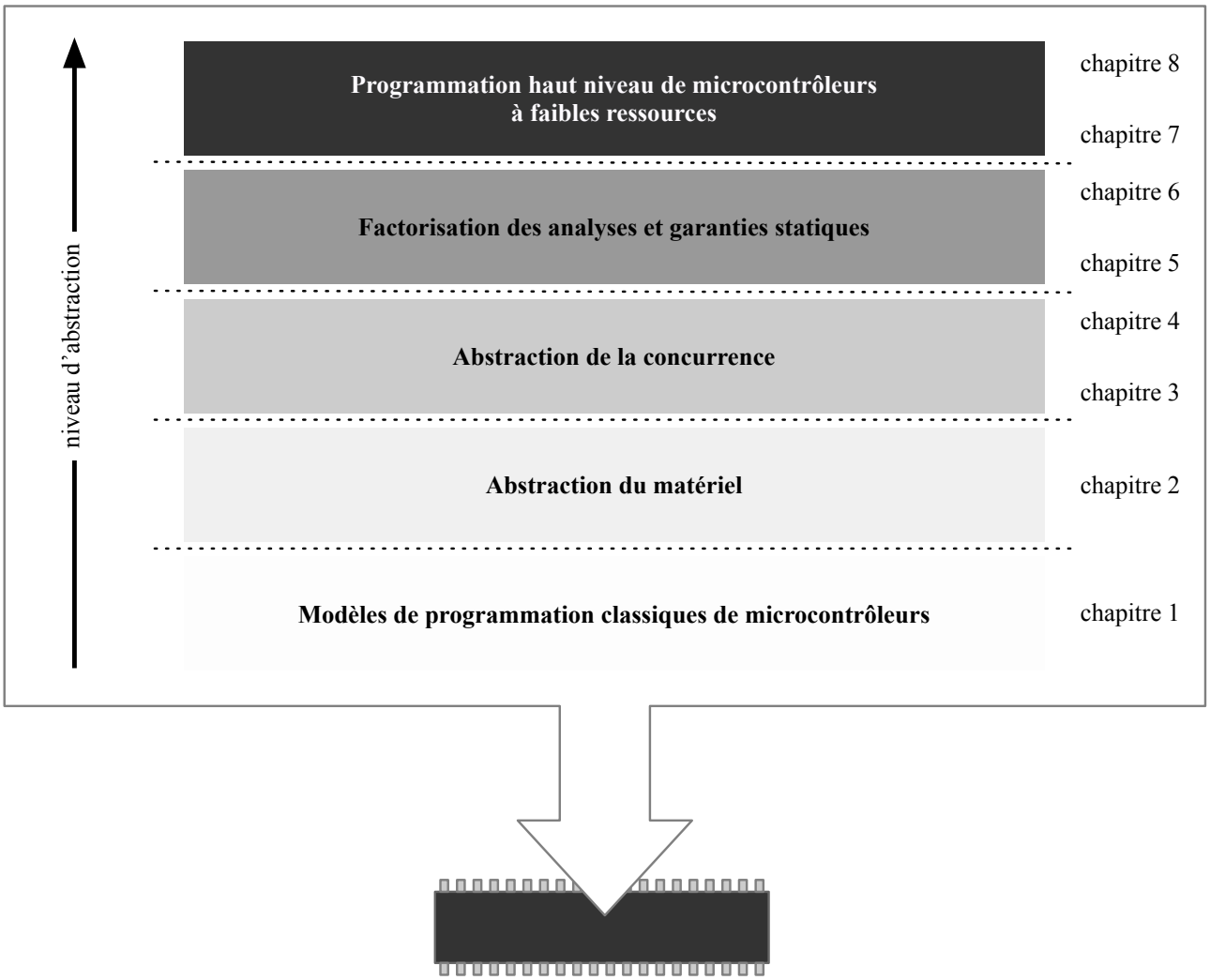



FIGURE 9.1 – Montée en abstraction pour une programmation de haut niveau de microcontrôleurs à faibles ressources

présentés à la conférence ERTS$^2$ (*Embedded Real Time Softwares and Systems*) en 2016 [VVC16] et en 2018 [VVC18a].

L'abstraction du matériel a constitué un premier socle pour les abstractions suivantes. Nous avons en effet abordé par la suite le fait qu'un très grand nombre de systèmes embarqués, dans lesquels les microcontrôleurs ont une place prépondérante, exposent des aspects *concurrents*. Cette concurrence inhérente aux multiples interactions entre un tel système et son environnement justifie l'adjonction d'un modèle de programmation adapté pour le développement de tels systèmes, en supplément des aspects plus *algorithmiques* fournis par les langages généralistes. Nous avons ainsi proposé *OCaLustre*, une extension du langage OCaml, permettant d'utiliser des traits de programmation synchrone pour la réalisation de programmes embarqués, tout en bénéficiant des avantages du langage hôte et de la machine virtuelle OMicroB. Cette extension de langage a un modèle de compilation qui lui confère une empreinte mémoire faible, et donc adaptée aux aspects matériels des microcontrôleurs considérés. L'extension synchrone OCaLustre offre donc une montée en abstraction *de la concurrence* : l'utilisation d'un modèle de programmation synchrone rend en effet implicite l'élaboration et l'interaction des différents éléments concurrents dans un programme. Qui plus est, le choix d'un modèle *à flots de données* permet de représenter facilement les interactions entre le programme et son environnement physique, constitué de composants électroniques qui communiquent par l'émission de valeurs électriques dont la « valeur » change au fil du temps. Cette extension synchrone offre à son tour de nouvelles garanties sur les programmes réalisés, en vérifiant par exemple au moment de la compilation la *cohérence causale* des différents composants logiciels qui interviennent dans un programme, évitant ainsi l'apparition de situations d'interblocage lors de l'exécution du programme. De plus, OCaLustre bénéficie d'une spécification formelle dont plusieurs aspects ont été vérifiés de façon mécanique à l'aide d'outils de formalisation et de preuve. Par exemple, le système des horloges synchrones, permettant de régir l'absence ou la présence de valeurs au cours de l'exécution du programme, a été défini formellement. Un outil permettant de vérifier la cohérence entre ce système et les types d'horloges inférés par le compilateur d'OCaLustre a été alors formalisé et prouvé correct en Coq, avant d'être extrait vers du code OCaml pour l'intégrer au compilateur. Cette spécification formelle du langage et la vérification de certaines de ses propriétés offrent ainsi aux programmes réalisés en OCaLustre un niveau de sûreté accru. La description du langage OCaLustre et de sa compilation a fait l'objet d'une publication dans les actes des JFLA (*Journées Francophones et Langages Applicatifs*) en 2017 [VVC17].

Les abstractions induites par nos travaux ne se font pas au détriment de la vérification de certaines garanties essentielles à l'élaboration de programmes embarqués, parfois critiques, et permet même de *factoriser* certaines analyses. En effet, nous avons ainsi illustré l'avantage de l'utilisation d'un bytecode commun à toutes les implantations de la machine virtuelle OMicroB afin de réaliser l'analyse du temps d'exécution pire cas d'un programme synchrone. L'estimation de ce temps d'exécution est souvent nécessaire au bon comportement d'un système embarqué critique, afin d'assurer que le temps de réaction du programme synchrone est inférieur à la fréquence d'apparition de ses entrées. Nous avons donc proposé une méthode permettant de calculer une borne supérieure du temps d'exécution réel d'un programme OCaLustre via l'analyse des instructions de son bytecode. Cette méthode a ensuite été prouvée correcte sur un langage similaire à un sous-ensemble des instructions bytecode OCaml et implantée dans un outil, nommé *Bytecrawler*, qui permet de calculer le temps d'exécution pire cas d'un programme OCaLustre sans soumettre l'utilisateur de nos solutions à des considérations complexes relatives au code de bas niveau réellement exécuté par l'interprète de la machine virtuelle. L'utilisation du

bytecode constitue ainsi, en quelque sorte, un troisième niveau d'abstraction apporté par notre solution : cette abstraction permet de rendre les analyses plus directes, et moins dépendantes des spécificités du matériel. La description de notre analyse de temps d'exécution pire cas ainsi que sa formalisation et sa preuve de correction ont été présentées au colloque WCET (*International Workshop on Worst-Case Execution Time Analysis*), satellite de la conférence ECRTS (*Euromicro Conference on Real-Time Systems*), en 2019 [VC19].

Les divers outils élaborés au cours de nos travaux, ainsi que la spécification formelle d'OCaLustre et la vérification partielle des propriétés lui étant associées, constituent une chaîne complète qui permet le développement d'applications plus sûres et plus riches sur des microcontrôleurs dont les ressources matérielles sont fortement limitées. Les mesures de performances visant à estimer la consommation en ressources du couple OMicroB/OCaLustre valident la viabilité de notre solution, dont la consommation mémoire est faible. À ce propos, trois exemples d'applications complètes, basées chacune sur un montage électronique différent, ont été décrits, et sont exécutables sur des microcontrôleurs très faiblement dotés en ressources. L'exemple de programmation d'un jeu-vidéo pour un *Arduboy* nous a par ailleurs servi d'application pratique pour l'élaboration d'un tutoriel invité aux JFLA 2018 [VVC18b]. Tous les exemples présentés permettent par ailleurs de démontrer la pertinence de nos différentes contributions, de la puissance expressive du langage OCaml à la sûreté des garanties apportées par l'extension synchrone. La simplicité de déploiement de telles applications est rendue possible grâce à la portabilité des solutions proposées. En effet, l'ensemble des solutions logicielles décrites dans cette thèse est portable : les programmes OCaLustre sont compilés vers du bytecode standard, exécutables sur une machine virtuelle générique, et analysables avec des outils qui s'adaptent aisément à différents microcontrôleurs. Cette portabilité est au cœur des ambitions de cette thèse, visant à permettre le développement d'applications sûres et libérées des considérations spécifiques au matériel utilisé. Le travail d'implémentation réalisé permet d'ores et déjà aux différents prototypes des solutions logicielles présentées dans cette thèse d'être pleinement utilisables, et des applications industrielles sont envisagées dans le cadre du projet LCHIP (*Low Cost Integrity Platform*) [⚓11] en partenariat avec la société CLEARSY. Ce projet consiste en une plateforme d'exécution sûre et peu coûteuse basée sur des microcontrôleurs PIC32. Notre approche machine virtuelle et notre modèle de programmation synchrone pourraient constituer au sein de cette plateforme un mode d'exécution alternatif dans le but d'introduire une redondance de l'exécution des programmes afin d'en vérifier la correction.

## Perspectives

Les divers travaux entrepris dans cette thèse constituent une première approche complète permettant de programmer des microcontrôleurs à l'aide de langages et modèles de programmation de haut niveau, qui apportent des garanties supplémentaires sur les programmes par rapport aux traditionnelles techniques de programmation de microcontrôleurs. Nous proposons, pour conclure ce manuscrit, un ensemble de propositions envisagées visant à augmenter d'autant plus le pouvoir expressif et les garanties apportées par nos solutions, ainsi que leur compatibilité avec du matériel à (très) faibles ressources.

Tout d'abord, la machine virtuelle OMicroB pourrait bénéficier d'optimisations supplémentaires visant à permettre, pour du matériel faiblement équipé en ressources, l'exécution de programmes dont la consommation mémoire est actuellement incompatible avec notre solution. L'une des principales causes de consommation excessive de la RAM par un programme OCaml provient de la persistance dans le

tas de valeurs immutables qui ne sont jamais libérées, comme des chaînes de caractères qui peuvent correspondre aux noms des exceptions utilisées explicitement ou non dans un programme. Par exemple, les exceptions `Out_of_memory` et `Stack_overflow` sont systématiquement déclarées par la bibliothèque d'exécution, et leurs noms perdurent dans le tas pendant toute son exécution. Des travaux en cours visent à déplacer de telles valeurs immutables depuis la RAM du microcontrôleurs vers sa mémoire flash, dont la taille est généralement plus conséquente, mais qui n'est pas destiné à être modifiée pendant l'exécution d'un programme. Le tas serait ainsi distribué à la fois entre la RAM pour traiter les valeurs mutables et la mémoire flash pour traiter les valeurs immutables. Cette optimisation permettrait de libérer un espace non négligeable dans le tas afin de traiter des valeurs réellement dynamiques. De surcroît, nous entreprenons de poursuivre nos travaux de portage d'OMicroB sur du matériel plus varié, afin de valider d'avantage la portabilité de notre approche. Nous visons par exemple les cartes ESP32 dotées de 520 kilo-octets de RAM et 4 mégaoctets de mémoire flash, ainsi que les microcontrôleurs STM32 présents dans les cartes de développement Nucleo dont les plus limitées en ressources bénéficient de 2 kilo-octets de RAM et 16 kilo-octets de mémoire flash.

Les premiers efforts de formalisation et de vérification de certaines propriétés du langage OCaLustre pourraient mener, à terme, à une certification complète du compilateur. En particulier, réaliser la preuve que les programmes compilés respectent la sémantique du langage permettrait de vérifier qu'aucune valeur de flot absent n'est jamais lue lors de l'exécution d'un programme, et de renforcer ainsi la sûreté du typage des horloges synchrones. Des travaux sont par ailleurs envisagés pour extraire directement, depuis Coq, le code réalisant la plupart des étapes de compilation d'un programme OCaLustre, afin d'assurer que ce processus de compilation respecte la spécification formelle adoptée. Nous pourrions à ce sujet nous rapprocher des travaux de Bourke *et al.* sur Vélus [BBD+17], un compilateur Lustre certifié en Coq.

De plus, de nouvelles garanties sur la cohérence des programmes réalisés pourraient être vérifiées par l'intermédiaire de *contrats synchrones*, sur le modèle des outils de *model-checking* Lesar [Rat92] et Kind2 [CMST16]. Des premières tentatives de définition de contrats en OCaLustre avec extraction vers le langage WhyML (compatible avec la plateforme de vérification de programmes Why3 [FP13]) ou vers Isabelle/HOL ont en effet été entreprises. Cependant leur vérification devient rapidement difficile dès que ces contrats font référence à des valeurs précédentes de flots (par exemple, pour décrire qu'un flot a des valeurs croissantes dans le temps) du fait que la preuve de telles propriétés fait intervenir un principe de *k-induction* que les outils de vérification généralistes peinent à exploiter de manière automatique.

Par ailleurs, des travaux initiaux visant à représenter le typage des horloges d'OCaLustre au sein même du système de type d'OCaml, via l'utilisation de types de données algébriques généralisés (GADT), ont été entrepris. Ces travaux qui semblent prometteurs pourraient permettre de laisser au compilateur standard OCaml le rôle de vérifier la sûreté complète du typage d'un programme OCaLustre, à la fois en ce qui concerne le typage de données standard (que la preuve de correction du typage vis-à-vis de la traduction en OCaml permet de vérifier), ainsi que le typage des horloges. Une tel usage du système de type riche d'OCaml représente un avantage supplémentaire à l'utilisation d'un langage de haut niveau.

La spécification du langage OCaLustre ainsi que son prototype ne concerne actuellement que des flots dont les valeurs sont de types simples, comme les booléens ou les entiers. Cette restriction a été décidée afin de respecter la même sémantique que celle de Lustre, ainsi que pour pouvoir proposer un mode de compilation « non allouant » qui soit compatible avec tous les programmes OCaLustre valides. Cependant,

il semblerait avantageux de pouvoir profiter de la richesse du langage hôte, en autorisant la manipulation de valeurs de types plus complexes. Une telle extension permettrait de créer des programmes OCaLustre plus expressifs. Par exemple, un programme OCaLustre pourrait être capable de manipuler des flots de n-uplets, des flots de listes, voire même des flots de fonctions ou de nœuds, permettant d'introduire un modèle de programmation synchrone d'ordre supérieur, sur l'exemple de Lucid Synchrone. L'extension d'OCaLustre à des types de données plus complexes semble directe pour ce qui concerne le modèle de compilation « standard » abordé au chapitre 4, mais rendrait la vérification du WCET difficile du fait de l'allocation dynamique de valeurs qu'elle introduirait.

Enfin, notre solution pourrait bénéficier d'analyses statiques plus poussées sur les programmes OCaml, permettant par exemple de borner la quantité de mémoire allouée par un programme afin d'assurer qu'aucun phénomène de dépassement du tas ne puisse se produire lors de l'exécution de ce programme. Nous envisageons de nous inspirer et d'étendre des travaux proches des nôtres, qui visent la programmation de systèmes embarqués dans un langage à la ML. Ces travaux intègrent au langage un système de régions explicites afin d'estimer statiquement une borne supérieure du nombre maximal de valeurs vivantes dans le tas au cours de l'exécution du programme [SC16a]. Ces analyses pourraient permettre, d'autre part, d'intégrer le temps d'exécution du GC pour le calcul du WCET (quitte à le déclencher volontairement, par exemple au début d'un instant synchrone).

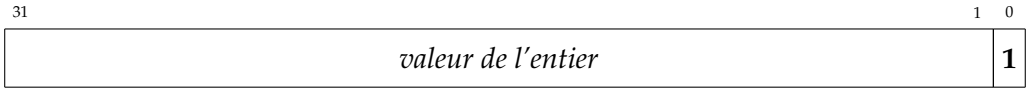
Toutes ces perspectives auraient pour finalité de poursuivre l'approche présentée dans cette thèse, en augmentant la sûreté de la programmation de systèmes embarqués, ainsi qu'en offrant des modèles de programmation d'encore plus haut niveau sur des microcontrôleurs qui conservent, quant à eux, les mêmes faibles ressources.

# A Représentation des valeurs OCaml

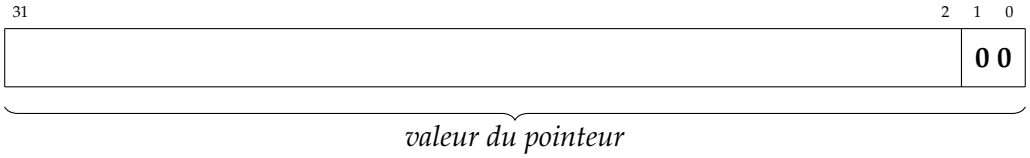
## A.1 Représentation des valeurs dans la ZAM

### A.1.1 Représentation sur 32 bits

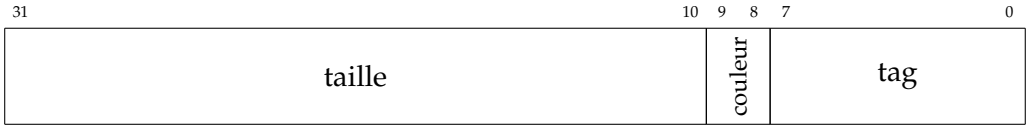
Entier :




Pointeur :



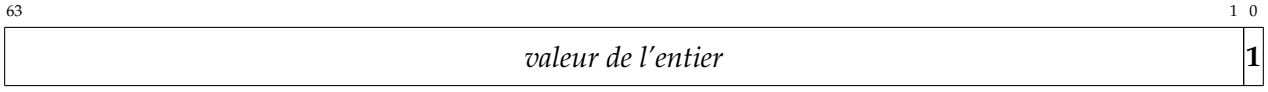

Entête de bloc :



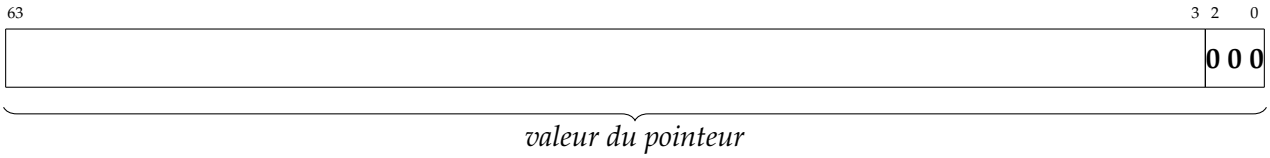

Les flottants sont alloués sur le tas dans des blocs qui contiennent deux valeurs OCaml (ce sont des flottants double précision sur $32 \times 2 = 64$ bits).

### A.1.2 Représentation sur 64 bits

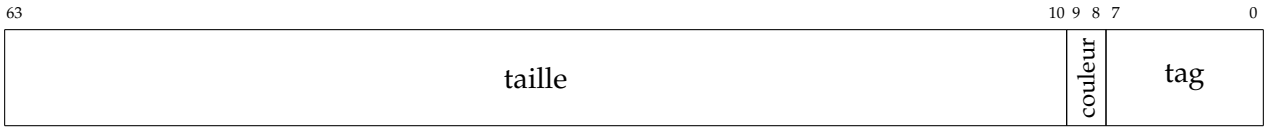
Entier :




Pointeur :




Entête de bloc :

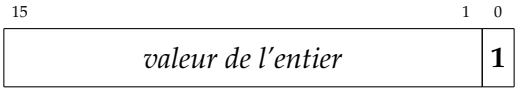



Les flottants sont alloués sur le tas dans des blocs qui contiennent une valeur OCaml (ce sont donc aussi des flottants double précision 64 bits).

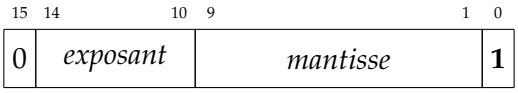
## A.2 Représentation des valeurs dans OMicroB

### A.2.1 Représentation sur 16 bits

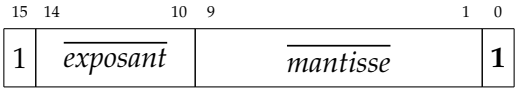
Entier :



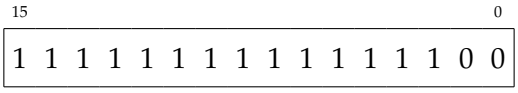

Flottant (positif) :



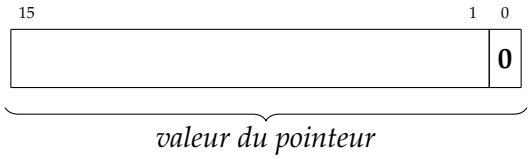

Flottant (négatif) :



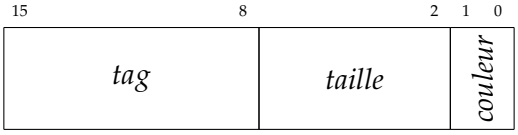

NaN :




Pointeur sur le tas :

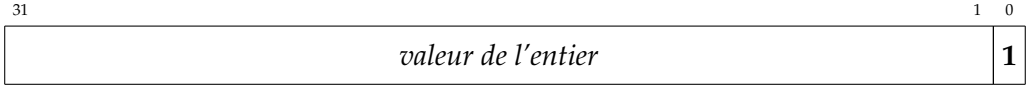



Entête de bloc :

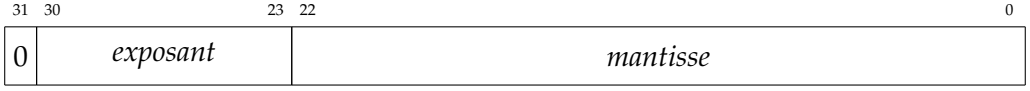



### A.2.2 Représentation sur 32 bits

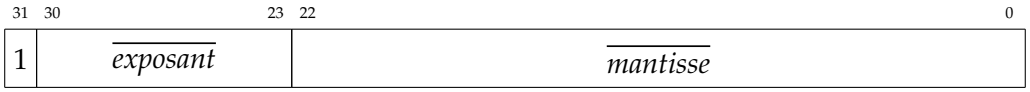
Entier :



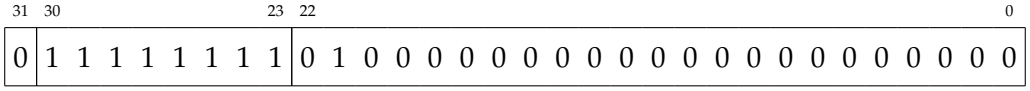

Flottant (positif) :



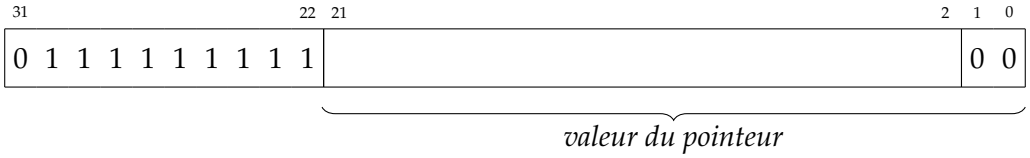

Flottant (négatif) :



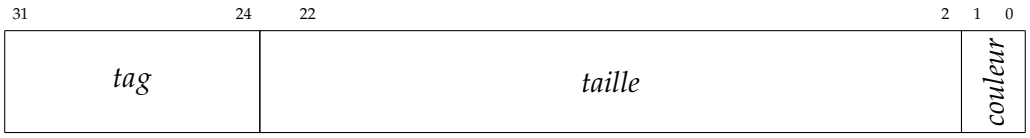
NaN :




Pointeur sur le tas :

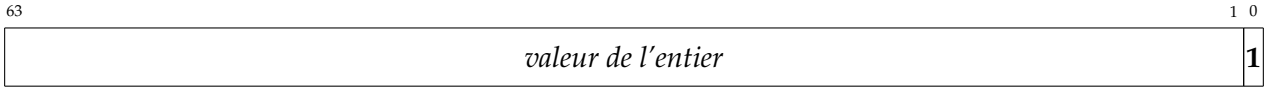



Entête de bloc :



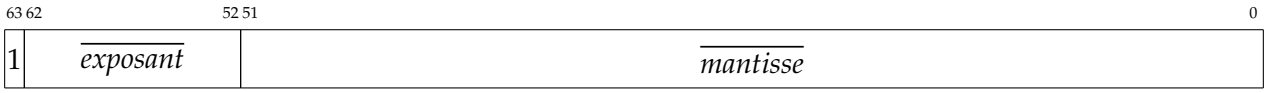

### A.2.3 Représentation sur 64 bits

Entier :

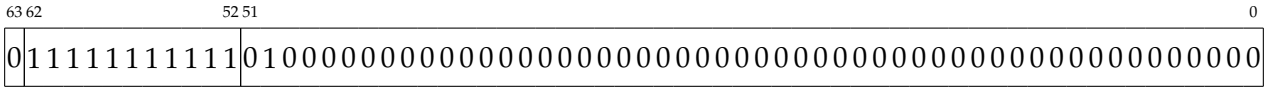



Flottant (positif) :

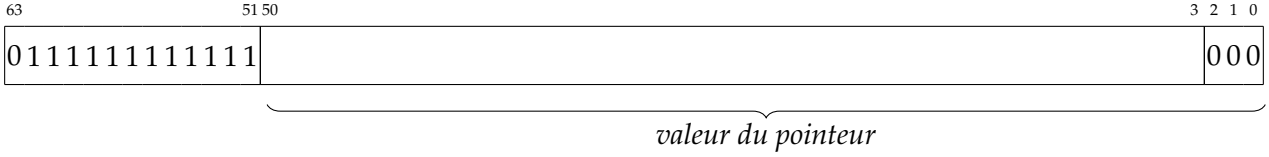

Flottant (négatif) :



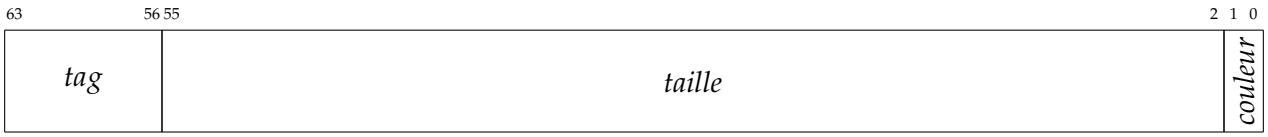

NaN :




Pointeur sur le tas :




Entête de bloc :



# B Code Lucid Synchrone de l'additionneur

```
let xor a b = if a then not b else b

let node fulladder (a,b,cin) =
  let x = (xor a b) in
  let s = (xor x cin) in
  let and1 = (x && cin) in
  let and2 = (a && b) in
  let cout = (and1 || and2) in
  (s,cout)

let node twobits_adder (c0,a0,a1,b0,b1) =
  let (s0,c1) = fulladder (a0,b0,c0) in
  let (s1,c2) = fulladder (a1,b1,c1) in
  (s0,s1,c2)

let node fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3) =
  let (s0,s1,c2) = twobits_adder (c0,a0,a1,b0,b1) in
  let (s2,s3,c4) = twobits_adder (c2,a2,a3,b2,b3) in
  (s0,s1,s2,s3,c4)

let node heightbits_adder (c0,a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7) =
  let (s0,s1,s2,s3,c4) = fourbits_adder (c0,a0,a1,a2,a3,b0,b1,b2,b3) in
  let (s4,s5,s6,s7,c8) = fourbits_adder (c4,a4,a5,a6,a7,b4,b5,b6,b7) in
  (s0,s1,s2,s3,s4,s5,s6,s7,c8)
```

# C Résultats des mesures de performances

| **Nom** | **Durée d'exécution avec ocamlrun** (secondes) | **Durée d'exécution avec OMicroB** (secondes) | **Ratio** | **Vitesse d'exécution avec OMicroB** (millions d'instr. bytecode/seconde) | **Nombre de déclenchements du GC d'OMicroB** |
|---|---|---|---|---|---|
| `apply` | 1.22 | 2.14 | 1.75 | 306.33 | 29 |
| `fibo` | 0.55 | 1.30 | 2.36 | 376.04 | 0 |
| `takc` | 1.05 | 3.11 | 2.96 | 378.38 | 110500 |
| `oddeven` | 0.28 | 0.61 | 2.17 | 604.60 | 0 |
| `floats` | 0.56 | 1.05 | 1.87 | 219.46 | 0 |
| `integr` | 0.04 | 0.12 | 3.00 | 222.16 | 21 |
| `eval` | 0.04 | 0.08 | 2.00 | 377.50 | 1153 |
| `sieve` | 0.04 | 0.07 | 1.75 | 486.00 | 1666 |
| `objet` | 0.08 | 0.17 | 2.12 | 389.77 | 3424 |
| `functor` | 0.26 | 0.58 | 2.23 | 406.31 | 10001 |
| `bubble` | 0.93 | 1.51 | 1.62 | 473.71 | 17 |
| `jdlv` | 0.36 | 0.74 | 2.05 | 463.82 | 2999 |
| `share` | 0.11 | 0.27 | 2.45 | 444.67 | 321 |
| `abrsort` | 0.47 | 1.17 | 2.48 | 335.00 | 38084 |
| `queens` | 1.37 | 3.47 | 2.53 | 399.24 | 56666 |

TABLE C.1 – Mesures de performances avec le GC Mark and Compact (sur PC)
*Options d'OMicroB : -arch 16 -gc MC -stack-size 500 -heap-size 2500*

| **Nom** | **Durée d'exécution avec ocamlrun** (secondes) | **Durée d'exécution avec OMicroB** (secondes) | **Ratio** | **Vitesse d'exécution avec OMicroB** (millions d'instr. bytecode/seconde) | **Nombre de déclenchements du GC d'OMicroB** |
|---|---|---|---|---|---|
| `apply` | 1.21 | 2.27 | 1.87 | 288.79 | 58 |
| `fibo` | 0.54 | 1.16 | 2.14 | 421.43 | 0 |
| `takc` | 1.04 | 2.94 | 2.82 | 400.26 | 221999 |
| `oddeven` | 0.28 | 0.62 | 2.21 | 594.85 | 0 |
| `floats` | 0.59 | 0.57 | 0.96 | 404.28 | 0 |
| `integr` | 0.04 | 0.06 | 1.50 | 440.00 | 41 |
| `eval` | 0.04 | 0.08 | 2.00 | 377.50 | 2307 |
| `sieve` | 0.04 | 0.07 | 1.75 | 486.00 | 3333 |
| `objet` | 0.08 | 0.16 | 2.00 | 414.13 | 10989 |
| `functor` | 0.24 | 0.60 | 2.50 | 392.77 | 30002 |
| `bubble` | 0.93 | 1.43 | 1.53 | 500.21 | 35 |
| `jdlv` | 0.37 | 0.72 | 1.94 | 476.70 | 6666 |
| `share` | 0.11 | 0.26 | 2.36 | 461.77 | 684 |
| `abrsort` | 0.47 | 1.14 | 2.42 | 343.81 | 115198 |
| `queens` | 1.37 | 3.22 | 2.35 | 430.23 | 149999 |

TABLE C.2 – Mesures de performances dans une représentation 32 bits des valeurs (sur PC)
*Options d'OMicroB : -arch 32 -gc SC -stack-size 500 -heap-size 2500*

| **Nom** | **Durée d'exécution avec ocamlrun** (secondes) | **Durée d'exécution avec OMicroB** (secondes) | **Ratio** | **Vitesse d'exécution avec OMicroB** (millions d'instr. bytecode/seconde) | **Nombre de déclenchements du GC d'OMicroB** |
|---|---|---|---|---|---|
| `apply` | 1.26 | 2.36 | 1.87 | 277.77 | 58 |
| `fibo` | 0.58 | 1.21 | 2.08 | 404.01 | 0 |
| `takc` | 1.06 | 2.97 | 2.80 | 396.21 | 219666 |
| `oddeven` | 0.29 | 0.60 | 2.06 | 614.68 | 0 |
| `floats` | 0.58 | 0.75 | 1.29 | 307.25 | 0 |
| `integr` | 0.05 | 0.06 | 1.20 | 444.33 | 41 |
| `eval` | 0.04 | 0.08 | 2.00 | 377.50 | 2272 |
| `sieve` | 0.05 | 0.07 | 1.40 | 486.00 | 3333 |
| `objet` | 0.08 | 0.18 | 2.25 | 368.12 | 10666 |
| `functor` | 0.24 | 0.60 | 2.50 | 392.77 | 30002 |
| `bubble` | 1.00 | 1.55 | 1.55 | 461.49 | 34 |
| `jdlv` | 0.37 | 0.83 | 2.24 | 413.53 | 6666 |
| `share` | 0.11 | 0.25 | 2.27 | 480.24 | 675 |
| `abrsort` | 0.52 | 1.20 | 2.30 | 326.62 | 112604 |
| `queens` | 1.35 | 3.29 | 2.43 | 421.08 | 144999 |

TABLE C.3 – Mesures de performances dans une représentation 64 bits des valeurs (sur PC)
*Options d'OMicroB : -arch 64 -gc SC -stack-size 500 -heap-size 2500*

| **Nom** | **Durée d'exécution** (secondes) | **Vitesse** (milliers d'instr./seconde) | **Nombre de déclenchements du GC** |
|---|---|---|---|
| `apply` | 7.860 | 83.432 | 0 |
| `fibo` | 2.999 | 163.065 | 0 |
| `takc` | 13.730 | 85.714 | 434 |
| `oddeven` | 2.262 | 163.085 | 0 |
| `floats` | 4.780 | 48.229 | 0 |
| `integr` | 0.379 | 69.902 | 0 |
| `eval` | 0.355 | 85.416 | 4 |
| `sieve` | 0.390 | 87.477 | 9 |
| `bubble` | 7.469 | 94.519 | 0 |
| `jdlv` | 3.737 | 91.938 | 16 |
| `share` | 1.373 | 88.275 | 1 |

TABLE C.4 – Mesures de vitesse de programmes pour OMicroB en version 32 bits (sur ATmega2560)
*Options d'OMicroB : -arch 32 -gc SC -stack-size 500 -heap-size 1400*

*Les programmes ne figurant pas dans cette table correspondent à ceux qui ne peuvent s'exécuter avec cette taille de tas réduite pour compenser la taille des valeurs.*

# D Code des applications

## D.1 Programme du lecteur de cartes perforées

```
1  open Avr
2
3  let t = Array.make 8 false
4  let clk = PIN13
5  let data = PIN12
6  let leds = [| PIN42 ; PIN43; PIN44; PIN45; PIN46; PIN47; PIN48; PIN49 |]
7
8  let init () =
9    (* réglage des broches en entrée *)
10   pin_mode clk INPUT;
11   pin_mode data INPUT;
12   (* réglage des broches en sortie *)
13   Array.iter (fun x -> pin_mode x OUTPUT) leds
14
15 let input_clk () = bool_of_level (digital_read clk)
16 let input_data () = bool_of_level (digital_read data)
17
18 (* fonction qui allume/éteint une des LED *)
19 let update_led i b = digital_write leds.(i) (level_of_bool b)
20
21 let output i data clk send =
22   if clk then t.(i) ← data;
23   if send then Array.iteri update_led t
24
25 let%node edge x ~return:e =
26   e = (x && (not (true ⋙ x)))
27
28 let%node read_bit (top,bot) ~return:(clk,data) =
29   clk = edge top;
30   data = bot [@when clk]
31
32 let%node count (reset) ~return:(cpt) =
33   cpt = (0 ⋙ (cpt + 1)) mod reset
34
35 let%node read_card (top,bot) ~return:(i,data,clk,send) =
36   (clk,data) = read_bit (top,bot);
37   i = count(8 [@when clk]);
38   send = merge clk (i = 7) false
39
```

```
let () =
  (* initialisation du matériel *)
  init ();
  (* création de l'état du noeud principal *)
  let st = read_card_alloc () in
  while true do
    (* lecture des entrées *)
    let c = input_clk () in
    let d = input_data () in
    (* mise à jour de l'état du noeud principal *)
    read_card_step st c d;
    (* émission des sorties *)
    let i = st.read_card_out_i in
    let data = st.read_card_out_data in
    let clk = st.read_card_out_clk in
    let send = st.read_card_out_send in
    output i data clk send
  done
```

## D.2 Programme de la tempéreuse

### D.2.1 tempereuse.ml

```
(** on allume/éteint si on appuie en même temps sur + et - **)
let%node thermo_on (p,m) ~return:(b) =
  b = (true ⋙ if p && m then not b else b)

(** modification de la température désirée selon le bouton appuyé **)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 ⋙ if p then w+5 else if m then w-5 else w)

(** noeud principal : calcul de la température désirée et de l'état de la résistance **)
(** Les températures sont en dixièmes de degrés C **)
let%node thermo (plus,minus,real_temp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (plus,minus);
  wanted = set_wanted_temp (plus[@when on], minus[@when on]);
  real = real_temp [@when on];
  heat = (real < wanted);
  resistor = merge on heat false
```

### D.2.2 thermo_io.ml

```
open Avr

(* déclaration de l'écran *)
let lcd = LiquidCrystal.create4bitmode PIN13 PIN12 PIN18 PIN19 PIN20 PIN21

(* déclaration des pins *)
let plus = PIN7
let minus = PIN6
```

```
9  let resistor = PIN10
10 let sensor = PINA0
11
12 (* conversion de température *)
13 let convert_temp t =
14   let f = (float_of_int (1033 - t) /. 11.67) in
15   int_of_float (f*.100.)
16
17 (* lecture de la température *)
18 let read_temp () =
19   let t = analog_read sensor in
20   Serial.write_string "an=";
21   Serial.write_int t;
22   convert_temp t
23
24 (* Affichage des températures sur l'écran LCD *)
25 let print_temp wanted real =
26   let split_temp t =
27     let u = t/10 in
28     let dec = t mod 10 in
29     (u,dec) in
30   LiquidCrystal.clear lcd;
31   LiquidCrystal.home lcd;
32   let (wu,wd) = split_temp wanted in
33   let (ru,rd) = split_temp real in
34   LiquidCrystal.print lcd "Wanted T :";
35   LiquidCrystal.print lcd ((string_of_int wu)^"."^(string_of_int wd));
36   LiquidCrystal.setCursor lcd 0 1;
37   LiquidCrystal.print lcd "Actual T :";
38   LiquidCrystal.print lcd ((string_of_int ru)^"."^(string_of_int rd))
39
40 (*** Fonctions d'entrées/sorties de l'instant synchrone ***)
41
42 (** fonction d'initialisation **)
43 let init_thermo () =
44   (* initialisation de la lecture analogique *)
45   Avr.adc_init ();
46   (* initialisation de l'écran *)
47   LiquidCrystal.lcdBegin lcd 16 2;
48   (* initialisation des broches *)
49   pin_mode sensor INPUT;
50   pin_mode resistor OUTPUT;
51   pin_mode plus INPUT;
52   pin_mode minus INPUT
53
54 (** fonction d'entrée **)
55 let input_thermo () =
56   let plus = digital_read plus in
57   let minus = digital_read minus in
```

```
58   let plus = bool_of_level plus in
59   let minus = bool_of_level minus in
60   let real_temp = read_temp () in
61   (plus,minus,real_temp)
62
63 (** fonction de sortie **)
64 let output_thermo (on,wanted,real,res) =
65   if on then
66     begin
67       print_temp wanted real;
68       digital_write resistor (if res then HIGH else LOW)
69     end
70   else
71     begin
72       LiquidCrystal.home lcd;
73       LiquidCrystal.clear lcd;
74       LiquidCrystal.print lcd "..."
75     end;
```

### D.2.3 Prise en compte du rapport cyclique de chauffe

Le programme OCaLustre suivant tient compte de l'inertie de la montée en température de la préparation : il mesure la proportion à laquelle doit être activée la résistance chauffante.

```
1
2  let%node min(a,b) ~return:c =
3    c = if a < b then a else b
4
5  let%node max(a,b) ~return:c =
6    c = if a > b then a else b
7
8  (** calcul de la proportion de chauffe (en %) **)
9  let%node update_prop (wtemp,ctemp) ~return:(prop) =
10   delta = min (10,max (-10,wtemp-ctemp));
11   delta2 = if delta < 0 then (-delta * delta) else (delta*delta);
12   offset = min (10,delta2);
13   pre_prop = (0 ⋙ prop);
14   prop = min (100,max (0, (pre_prop+offset)))
15
16 let%node timer (number) ~return:(alarm) =
17   time = (0 ⋙ (time + 10)) mod 100;
18   alarm = if (time < number) then true else false
19
20 let%node heat (w,c) ~return:(h) =
21   count = (0 ⋙ count + 1) mod 10;
22   update = (count = 0);
23   (* le rapport cyclique (prop) est mis à jour tous les 10 instants *)
24   prop = merge update
25            (update_prop (w [@when update],c [@when update]))
26            ((0 ⋙ prop) [@whennot update]);
```

```
  h = timer (prop)

(** on allume/éteint si on appuie en même temps sur + et - **)
let%node thermo_on(p,m) ~return:(b) =
  b = (true ⋙ if p && m then (not b) else b)

(** modification de la température désirée selon le bouton appuyé **)
let%node set_wanted_temp (p,m) ~return:(w) =
  w = (325 ⋙ if p then w+5 else if m then w-5 else w)

(** noeud principal : calcul de la température désirée et de l'état de la résistance **)
(** Les températures sont en dixièmes de degrés celsius **)
let%node thermo (plus,minus,real_temp) ~return:(on,wanted,real,resistor) =
  on = thermo_on (plus,minus);
  wanted = set_wanted_temp (plus[@when on], minus[@when on]);
  real = real_temp [@when on];
  heat = heat (wanted,real);
  resistor = merge on heat false
```

## D.3 Programme du Serpent

### D.3.1 spi.ml

```
(*** Module de gestion de la connexion SPI (Serial Peripheral Interface) ***)

open Avr

(** Initialiser la connexion SPI **)
let begin_spi ~sck ~mosi =
  set_bit SPCR MSTR;
  set_bit SPCR SPE;
  set_bit SPSR SPI2x;
  pin_mode sck OUTPUT;
  pin_mode mosi OUTPUT

(** Arrêter la connexion SPI **)
let end_spi () = clear_bit SPCR SPE

(*** Emettre des données via la connexion SPI **)
let transfer data = write_register SPDR data
```

### D.3.2 oled.ml

```
open Avr

(*** Fonctions d'écriture/lecture dans le buffer d'affichage ***)
external write_buffer : int -> int -> bool -> unit = "caml_buffer_write"
external read_buffer : int -> int -> bool = "caml_buffer_read"
external get_byte_buffer : unit -> int = "caml_buffer_get_byte"
```

```
7
8  (*** Programme de boot de l'écran ***)
9  let boot_program =
10   [|
11     0xD5; 0xF0; (* Set display clock divisor = 0xF0 *)
12     0x8D; 0x14; (* Enable charge Pump *)
13     0xA1;       (* Set segment re-map *)
14     0xC8;       (* Set COM Output scan direction *)
15     0x81; 0xCF; (* Set contrast = 0xCF *)
16     0xD9; 0xF1; (* Set precharge = 0xF1 *)
17     0xAF;       (* Display ON *)
18     0x20; 0x00; (* Set display mode = horizontal addressing mode *)
19   |]
20
21 (*** Envoi du programme sur l'écran ***)
22 let transfer_program prog =
23   Array.iter Spi.transfer prog
24
25 (*** Passer l'écran en mode "commande" ***)
26 let command_mode cs dc =
27   digital_write cs HIGH;
28   digital_write dc LOW;
29   digital_write cs LOW
30
31 (*** Passer l'écran en mode "données" ***)
32 let data_mode cs dc =
33   digital_write dc HIGH;
34   digital_write cs LOW
35
36 (*** Envoyer une commande à l'écran ***)
37 let send_lcd_command cs dc com =
38   command_mode cs dc;
39   Spi.transfer com;
40   data_mode cs dc
41
42 (*** Écrire un pixel (true=noir/false=blanc) ***)
43 let draw x y color =
44   write_buffer x y color
45
46 (*** Effacer l'écran ***)
47 let clear() =
48  for _i = 0 to 1023 do
49    Spi.transfer(0x00)
50  done
51
52 (*** Envoyer le buffer sur l'écran ***)
53 let flush () =
54   for _i = 0 to 1023 do
55     Spi.transfer(get_byte_buffer())
```

```
  done

(*** Initialisation de l'écran ***)
let boot ~cs ~dc ~rst =
  digital_write rst HIGH;
  digital_write rst LOW;
  digital_write rst HIGH;
  command_mode cs dc;
  transfer_program boot_program;
  data_mode cs dc;
  clear()
```

### D.3.3 arduboy.ml

```
open Avr

(** Les broches utilisées **)
let cs = PIN12
let dc = PIN4
let rst = PIN6
let button_left = PINA2
let button_right = PINA1
let button_down = PINA3
let button_up = PINA0
let button_a = PIN7
let button_b = PIN8
let blue = PIN9
let red = PIN10
let green = PIN11

(** initialisation des LED rgb à anode commune (HIGH = éteinte) **)
let init_led l =
  digital_write l HIGH

(** allumer une des LED rgb à anode commune (LOW = allumé) **)
let light_led l =
  digital_write l LOW

(** initialisation des broches **)
let boot_pins () =
  List.iter (fun x -> pin_mode x INPUT_PULLUP) [button_left; button_right; button_up;
                                                button_down];
  pin_mode button_a INPUT_PULLUP;
  pin_mode button_b INPUT_PULLUP;
  List.iter (fun x -> pin_mode x OUTPUT) [red;green;blue];
  List.iter (fun x -> pin_mode x OUTPUT) [cs;dc;rst];
  List.iter init_led [red;green;blue]

(** initialisation des broches, de la liaison SPI, et de l'écran **)
let init () =
```

```
  boot_pins ();
  Spi.begin_spi ~sck:SCK ~mosi:MOSI;
  Oled.boot ~cs:cs ~dc:dc ~rst:rst
```

### D.3.4 snake.ml

```

(** Direction du serpent **)
type direction = South | North | East | West

(** Fonctions d'orientation : le serpent tourne à gauche et à droite **
 ** par rapport à sa direction courante **)
let left_of = function
  | South -> East
  | North -> West
  | East -> North
  | West -> South

let right_of = function
  | South -> West
  | North -> East
  | East -> South
  | West -> North

(** Modulo **)
let nmod x y =
  (x + y) mod y

(** Tirage aléatoire d'un entier **)
let new_position n = Random.int n

(** Nouvelle cordonnée en x ou en y **)
let%node new_coord (dir,max,v,dir1,dir2) ~return:n =
  n = if dir = dir1 then call nmod (v-1) max
      else if dir = dir2 then call nmod (v+1) max
      else v

(** Nouvelle position de la tête du serpent **)
let%node new_head (dir,w,h) ~return:(x,y) =
  x = new_coord (dir,w,0 ⋙ x,West,East);
  y = new_coord (dir,h,0 ⋙ y,North,South)

(** Tourner à gauche **)
let%node left (dir) ~return:ndir =
  ndir = call left_of dir

(** Tourner à droite **)
let%node right (dir) ~return:ndir =
  ndir = call right_of dir

```

```
45 (** Direction du serpent à partir de sa direction précédente **)
46 let%node direction (l,r) ~return:dir =
47   pre_dir = (South ⋙ dir);
48   dir = if l then
49           (merge l (left (pre_dir [@when l])) (pre_dir [@whennot l]))
50         else
51           (merge r (right (pre_dir [@when r])) (pre_dir [@whennot r]))
52
53 (** Nouvelle position de la pomme **)
54 let%node new_apple (width,height) ~return:(a_x,a_y) =
55   (a_x,a_y) = (call new_position width, call new_position height)
56
57 (** Boucle de jeu **)
58 let%node game_loop (max_size,left,right,width,height)
59      ~return:(head,tail,new_x,new_y,apple_x,apple_y,win) =
60   (new_x,new_y) = new_head (dir,width,height);
61   dir = direction(left,right);
62   head = (1 ⋙ ((head+1) mod max_size));
63   eats = (apple_x = new_x && apple_y = new_y);
64   (a_x,a_y) = new_apple (width [@when eats], height [@when eats]);
65   apple_x = (10 ⋙ merge eats a_x (apple_x [@whennot eats]));
66   apple_y = (10 ⋙ merge eats a_y (apple_y [@whennot eats]));
67   tail = merge eats ((0⋙ tail)[@when eats]) (0 ⋙ ((tail+1) mod max_size) [@whennot eats]);
68   size = (1 ⋙ merge eats ((size + 1) [@when eats]) (size [@whennot eats]));
69   win = (size = max_size -1)
70
71 (** Front montant (pour les boutons) **)
72 let%node rising_edge i ~return:o =
73   o = (i && (not (false ⋙ i)))
74
75 (** Noeud principal **)
76 let%node main (max_size,button1,button2,width,height) ~return:(head,tail,nx,ny,apple_x,apple_y,win) =
77   left = rising_edge (button1);
78   right = rising_edge (button2);
79   (head,tail,nx,ny,apple_x,apple_y,win) = game_loop(max_size,left,right,width,height)
```

### D.3.5 main_io.ml

```
1  open Avr
2
3  (*** Fonctions du jeu ***)
4
5  (** Le tableau qui contient les positions du corps du serpent **)
6  let max_size = 15
7  let snake = Array.make max_size (0,0)
8
9  (** Teste si le serpent entre en collision avec lui-même *)
10 exception Lose
11 let eats_itself head tail =
12   let f c = if c = snake.(head) then (raise Lose) in
```

```
13   try
14     if tail < head then
15       begin
16         for i = tail to head - 1 do
17           f snake.(i);
18         done
19       end
20     else
21       begin
22         for i = tail to max_size - 1 do
23           f snake.(i);
24         done;
25         for i = 0 to head - 1 do
26           f snake.(i);
27         done;
28       end;
29     false
30   with Lose -> true
31
32 (*** Fonctions d'affichage ***)
33
34 let draw_snake head tail : unit =
35   let (x,y) = snake.(head) in
36   let (x',y') = snake.(tail) in
37   (* on affiche une nouvelle tête *)
38   Oled.draw x y true;
39   (* on efface la queue pour donner l'illusion de déplacement *)
40   Oled.draw x' y' false
41
42 let draw_apple x y : unit =
43   Oled.draw x y true
44
45 (*** Fonctions de la machine d'exécution synchrone ***)
46
47 (** Initialisation du programme synchrone **)
48 let init_main () = Arduboy.init ()
49
50 (** Fonction d'entrée de chaque instant synchrone **)
51 let input_main () =
52   let l = Avr.digital_read Arduboy.button_left in
53   let r = Avr.digital_read Arduboy.button_right in
54   (max_size,bool_of_level l,bool_of_level r,64,32)
55
56 (** Fonction de sortie de chaque instant synchrone **)
57 let output_main (head,tail,nx,ny,apple_x,apple_y,win) =
58   snake.(head) ← (nx,ny);
59   draw_snake head tail;
60   draw_apple apple_x apple_y;
61   Oled.flush ();
```

```
62    if win then
63      begin
64        (* Gagné *)
65        Arduboy.light_led Arduboy.green;
66        Avr.delay 2000;
67        raise Exit
68      end
69    else if eats_itself head tail then
70      begin
71        (* Perdu *)
72        Arduboy.light_led Arduboy.red;
73        Avr.delay 2000;
74        raise Exit
75      end
```

# Références Web


[⚓1] Annexes électroniques du manuscrit.
https://stevenvar.github.io/these.

[⚓2] Certifications de KCG.
http://www.esterel-technologies.com/products/scade-suite/automatic-code-generation/scade-suite-kcg-certification-kits/.

[⚓3] Description de PPX.
http://ocamllabs.io/doc/ppx.html.

[⚓4] Fiche technique de l'écran SSD1306.
https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf.

[⚓5] Global MCU Market 2018 by Manufacturers, Regions, Type and Application, Forecast to 2023.
https://www.orianresearch.com/request-sample/553889.

[⚓6] Instructions de la machine virtuelle OCaml (site du projet Cadmium).
http://cadmium.x9c.fr/distrib/caml-instructions.pdf.

[⚓7] Page de l'IDE Arduino (site de la plateforme Arduino).
https://www.arduino.cc/en/Main/Software.

[⚓8] Page de l'IDE Atmel Studio (site de Microchip).
https://www.microchip.com/mplab/avr-support/atmel-studio-7.

[⚓9] Page de l'outil ocamlclean (site du projet OCaPIC).
http://www.algo-prog.info/ocapic/web/index.php?id=ocamlclean.

[⚓10] Page du compilateur MPLAB (site de Microchip).
https://www.microchip.com/mplab.

[⚓11] Page du projet LCHIP (site de Clearsy).
http://www.clearsy.com/2016/10/4260/.

[⚓12] Page du projet python-on-a-chip (archive).
https://code.google.com/archive/p/python-on-a-chip/.

[⚓13] Site de la machine virtuelle Java HaikuVM.
http://haiku-vm.sourceforge.net.

[⚓14] Site de la machine virtuelle NanoVM.
http://www.harbaum.org/till/nanovm.

[⚓15] Site de la plateforme Arduino.
https://www.arduino.cc.

[⚓16] Site de la plateforme microEJ.
https://www.microej.com.

[⚓17] Site de la suite logicielle Proteus.
https://www.labcenter.com.

[⚓18] Site de l'implémentation Bigloo.
https://www-sop.inria.fr/mimosa/fp/Bigloo.

[⚓19] Site de Lucid Synchrone.
https://www.di.ens.fr/~pouzet/lucid-synchrone/index.html.

[⚓20] Site de micro :bit.
https://microbit.org/.

[⚓21] Site de ReactiveML.
http://rml.lri.fr.

[⚓22] Site du langage Guile.
https://www.gnu.org/software/guile.

[⚓23] Site du programmateur usbpicprog.
http://usbpicprog.org.

[⚓24] Site du projet AVR-Ada.
https://sourceforge.net/projects/avr-ada.

[⚓25] Site du projet AVR Libc.
https://www.nongnu.org/avr-libc.

[⚓26] Site du projet AVR-Rust.
https://github.com/avr-rust.

[⚓27] Site du projet SDCC.
http://sdcc.sourceforge.net.

[⚓28] Site du projet TinyGO.
https://tinygo.org.

[⚓29] Tutoriel OCaml - Objets (site officiel du langage OCaml) .
https://ocaml.org/learn/tutorials/objects.html.

# Bibliographie


[ABC+17] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich et Steve Zdancewic : *Position paper : the science of deep specification*. Philosophical Transactions of the Royal Society A : Mathematical, Physical and Engineering Sciences, 375(2104) :20160331, septembre 2017. https://doi.org/10.1098/rsta.2016.0331. *[cité page 41]*

[Abr96] Jean-Raymond Abrial : *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996, ISBN 0-521-49619-5. *[cité page 41]*

[ACR+08] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa et Miguel Castro : *Preventing Memory Error Exploits with WIT*. Dans *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 263–277, Oakland, CA, USA, mai 2008. IEEE Computer Society. https://doi.org/10.1109/SP.2008.30. *[cité page 38]*

[AGG07] Elvira Albert, Samir Genaim et Miguel Gómez-Zamalloa : *Heap Space Analysis for Java Bytecode*. Dans *Proceedings of the 6th International Symposium on Memory Management (ISMM 2007)*, pages 105–116, Montréal, Québec, Canada, octobre 2007. ACM. https://doi.org/10.1145/1296907.1296922. *[cité page 23]*

[AS87] Bowen Alpern et Fred B. Schneider : *Recognizing Safety and Liveness*. Distributed Computing, 2(3) :117–126, septembre 1987, ISSN 1432-0452. https://doi.org/10.1007/BF01782772. *[cité page 38]*

[Aug13] Cédric Auger : *Compilation certifiée de SCADE/LUSTRE*. Thèse de doctorat, Université Paris-Sud, Orsay, France, 2013. https://tel.archives-ouvertes.fr/tel-00818169. *[2 citations pages 41 et 107]*

[AW77] Edward A. Ashcroft et William W. Wadge : *Lucid, a Nonprocedural Language with Iteration*. Communications of the ACM, 20(7) :519–526, 1977. https://doi.org/10.1145/359636.359715. *[2 citations pages 33 et 79]*

[Bad14] Yusuf Abdullahi Badamasi : *The working principle of an Arduino*. Dans *Proceedings of the 11th International Conference on Electronics, Computer and Computation (ICECCO 2014)*, pages 1–4, Abuja, Nigeria., septembre 2014. https://doi.org/10.1109/ICECCO.2014.6997578. *[cité page 17]*

[BB91] Albert Benveniste et Gérard Berry : *The synchronous approach to reactive and real-time systems*. Proceedings of the IEEE, 79(9) :1270–1282, septembre 1991, ISSN 0018-9219. https://doi.org/10.1109/5.97297. *[cité page 31]*

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet et Lionel Rieg : *A formally verified compiler for Lustre*. Dans *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, pages 586–601, Barcelone, Espagne, juin 2017. ACM. https://doi.org/10.1145/3062341.3062358. *[3 citations pages 41, 107 et 215]*

[BBP18] Timothy Bourke, Lélio Brun et Marc Pouzet : *Towards a verified Lustre compiler with modular reset*. Dans *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES 2018*, pages 14–17, Sankt Goar, Allemagne, mai 2018. ACM. https://doi.org/10.1145/3207719.3207732. *[cité page 41]*

[BC84] Gérard Berry et Laurent Cosserat : *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*. Dans *Seminar on Concurrency*, tome 197 de *Lecture Notes in Computer Science*, pages 389–448, Carnegie-Mellon University, Pittsburg, PA, USA, juillet 1984. Springer. https://doi.org/10.1007/3-540-15670-4_19. *[cité page 32]*

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic et Robert de Simone : *The synchronous languages 12 years later*. Proceedings of the IEEE, 91(1) :64–83, 2003. https://doi.org/10.1109/JPROC.2002.805826. *[cité page 39]*

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt et Gareth Smith : *A trusted mechanised JavaScript specification*. Dans *POPL*, pages 87–100. ACM, 2014. *[cité page 41]*

[BCHP08] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon et Marc Pouzet : *Clock-directed modular code generation for synchronous data-flow languages*. Dans *Proceedings of the 2008 ACM SIGPLAN/-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 121–130, Tucson, AZ, USA, juin 2008. ACM. https://doi.org/10.1145/1375657.1375674. *[2 citations pages 109 et 119]*

[BCL08] Niels Brouwers, Peter Corke et Koen Langendoen : *Darjeeling, a Java compatible virtual machine for microcontrollers*. Dans *Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference (Middleware 2008)*, pages 18–23, Leuven, Belgique, décembre 2008. ACM. https://doi.org/10.1145/1462735.1462740. *[cité page 25]*

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange et Pascal Sainrat : *OTAWA : An Open Toolbox for Adaptive WCET Analysis*. Dans *Proceedings of the 8th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS 2010)*, pages 35–46, Weidhofen/Ybbs, Autriche, 2010. Springer. https://doi.org/10.1007/978-3-642-16256-5_6. *[cité page 40]*

[BDPR17] Timothy Bourke, Pierre Evariste Dagand, Marc Pouzet et Lionel Rieg : *Vérification de la génération modulaire du code impératif pour Lustre*. Dans *Actes des 28èmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, janvier 2017. https://hal.inria.fr/hal-01403830. *[cité page 93]*

[Bel17] Charles Bell : *Introducing MicroPython*, pages 27–57. Apress, Berkeley, CA, 2017, ISBN 978-1-4842-3123-4. https://doi.org/10.1007/978-1-4842-3123-4_2. *[cité page 25]*

[Ber86] Jean-Louis Bergerand : *LUSTRE : un langage déclaratif pour le temps réel*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, 1986. *[cité page 77]*

[BFL18] Clément Ballabriga, Julien Forget et Giuseppe Lipari : *Symbolic WCET computation*. ACM Transactions on Embedded Computing Systems (TECS), 17(2) :39, 2018. *[cité page 168]*

[BN94] Swagato Basumallick et Kelvin Nilsen : *Cache issues in real-time systems*. Dans *In Proceedings of the 1994 ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time System*, Orlando, FL, USA, juin 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.3755. *[cité page 159]*

[Bou98] Hédi Boufaied : *Machines d'exécution pour langages synchrones*. Thèse de doctorat, Université de Nice-Sophia Antipolis, Nice, France, 1998. *[cité page 124]*

[BP13] Timothy Bourke et Marc Pouzet : *Zélus : a synchronous language with ODEs*. Dans *Proceedings of the 16th international conference on Hybrid systems : computation and control (HSCC 2013)*, pages 113–118, Philadelphia, PA, USA, avril 2013. ACM. https://doi.org/10.1145/2461328.2461348. *[cité page 37]*

[BP19] Timothy Bourke et Marc Pouzet : *Clocked arguments in a verified Lustre compiler*. Dans *Actes des 30èmes Journées Francophones des Langages Applicatifs (JFLA 2019)*, page 16, Les Rousses, France, janvier 2019. https://hal.inria.fr/hal-02005639. *[cité page 138]*

[BRS18] Frédéric Bour, Thomas Refis et Gabriel Scherer : *Merlin : a language server for OCaml (experience report)*. Proceedings of the ACM on Programming Languages, 2(ICFP) :103 :1–103 :15, septembre 2018. https://doi.org/10.1145/3236798. *[cité page 101]*

[BSR12] Thomas W. Barr, Rebecca Smith et Scott Rixner : *Design and Implementation of an Embedded Python Run-Time System*. Dans *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 297–308, Boston, MA, USA, juin 2012. USENIX Association. https://www.usenix.org/conference/atc12. *[cité page 26]*

[BV97] Ross Bannatyne et Greg Viot : *Introduction to microcontrollers*. Dans *Proceedings of the 1997 Western Electronics Show and Convention (WESCON/97)*, pages 564–574, Santa Clara, CA, USA, novembre 1997. https://doi.org/10.1109/WESCON.1997.632384. *[cité page 15]*

[Cam16] Giulio Camilleri : *An LLVM Bitcode Interpreter for 16-bit MSP430 Microcontrollers*. Mémoire de licence, Faculty of ICT - University of Malta, mai 2016. https://www.um.edu.mt/library/oar/handle/123456789/13777. *[cité page 24]*

[CH86] Paul Caspi et Nicolas Halbwachs : *A Functional Model for Describing and Reasoning About Time Behaviour of Computing Systems*. Acta Informatica, 22(6) :595–627, 1986. https://doi.org/10.1007/BF00263648. *[cité page 34]*

[Cha92] Emmanuel Chailloux : *An Efficient Way of Compiling ML to C*. Dans *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 37–51, San Francisco, CA, USA, juin 1992. ACM. *[cité page 73]*

[CHP06] Paul Caspi, Grégoire Hamon et Mark Pouzet : *Lucid Synchrone, un langage de programmation des systèmes réactifs*. Systèmes Temps-réel : Techniques de Description et de Vérification Théorie et Outils, 1 :217260, 2006. *[cité page 37]*

[CL14] Raphaëlle Crubillé et Ugo Dal Lago : *On Probabilistic Applicative Bisimulation and Call-by-Value λ-Calculi*. Dans *Proceedings of the 23rd European Symposium on Programming (ESOP 2014)*, tome 8410 de *Lecture Notes in Computer Science*, pages 209–228, Grenoble, France, avril 2014. Springer. https://doi.org/10.1007/978-3-642-54833-8_12. *[cité page 152]*

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel et Cesare Tinelli : *The Kind 2 Model Checker*. Dans *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, pages 510–517, Toronto, Canada, juillet 2016. Springer International Publishing. https://doi.org/10.1007/978-3-319-41540-6_29. *[cité page 215]*

[CP99] Paul Caspi et Marc Pouzet : *Lucid Synchrone : une extension fonctionnelle de Lustre*. Dans *Actes des 10èmes Journées Francophones des Langages Applicatifs (JFLA 99)*, Avoriaz, France, février 1999. INRIA. https://hal.archives-ouvertes.fr/hal-01574464. *[2 citations pages 37 et 79]*

[CP01] Antoine Colin et Isabelle Puaut : *A Modular & Retargetable Framework for Tree-Based WCET Analysis*. Dans *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*, pages 37–44, Delft, Pays-Bas, juin 2001. IEEE Computer Society. https://doi.org/10.1109/EMRTS.2001.933995. *[cité page 40]*

[CP03] Jean-Louis Colaço et Marc Pouzet : *Clocks as First Class Abstract Types*. Dans *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, tome 2855 de *Lecture Notes in Computer Science*, pages 134–155, Philadelphia, PA, USA, octobre 2003. Springer. https://doi.org/10.1007/978-3-540-45212-6_10. *[4 citations pages 89, 101, 118 et 137]*

[CP04] Jean-Louis Colaço et Marc Pouzet : *Type-based initialization analysis of a synchronous dataflow language*. International Journal on Software Tools for Technology Transfer (STTT), 6(3) :245–255, 2004. https://doi.org/10.1007/s10009-004-0160-y. *[cité page 81]*

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs et John Plaice : *Lustre : A Declarative Language for Programming Synchronous Systems*. Dans *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL 87)*, pages 178–188, Munich, Germany, janvier 1987. ACM Press. https://doi.org/10.1145/41625.41641. *[3 citations pages 33, 77 et 104]*

[CPP17] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet : *SCADE 6 : A formal language for embedded critical software development (invited paper)*. Dans *Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 1–11, Sophia Antipolis, France, septembre 2017. IEEE Computer Society. https://doi.org/10.1109/TASE.2017.8285623. *[3 citations pages 12, 36 et 119]*

[Cri92] Régis Cridlig : *An Optimizing ML to C Compiler*. Dans *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 28–36, San Francisco, CA, USA, juin 1992. ACM. *[cité page 73]*

[DF05] Danny Dubé et Marc Feeley : *BIT : A Very Compact Scheme System for Microcontrollers*. Higher-Order and Symbolic Computation, 18(3-4) :271–298, 2005. https://doi.org/10.1007/s10990-005-4877-4. *[cité page 26]*

[DM82] Luis Damas et Robin Milner : *Principal Type-schemes for Functional Programs*. Dans *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212, Albuquerque, New Mexico, janvier 1982. ACM, ISBN 0-89791-065-6. http://doi.acm.org/10.1145/582153.582176. *[cité page 89]*

[Dor08] Francois Xavier Dormoy : *Scade 6 : a model based solution for safety critical software development*. Dans *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS 2008)*, pages 1–9, Toulouse, France, janvier 2008. *[cité page 87]*

[DRM11] Gwenaël Delaval, Eric Rutten et Hervé Marchand : *Intégration de la synthèse de contrôleurs discrets dans un langage de programmation*. Dans *Actes du 8ème colloque francophone sur la modélisation des systèmes réactifs (MSR'11)*, Lille, France, novembre 2011. https://hal.inria.fr/inria-00629104. *[cité page 37]*

[DS00] Stephan Diehl et Peter Sestoft : *Abstract Machines for Programming Language Implementation*. Future Generation Computer Systems, 16(7) :739–751, mai 2000, ISSN 0167-739X. http://doi.org/10.1016/S0167-739X(99)00088-6. *[cité page 22]*

[FD03] Marc Feeley et Danny Dubé : *PICBIT : A Scheme system for the PIC microcontroller*. Dans *Proceedings of the 4th Workshop on Scheme and Functional Programming*, pages 7–15, Boston, MA, USA, novembre 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.5903. *[cité page 26]*

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich : *Why3 - Where Programs Meet Provers*. Dans *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, tome 7792, pages 125–128, Rome, Italie, mars 2013. Springer. https://doi.org/10.1007/978-3-642-37036-6_8. *[cité page 215]*

[Gér13] Léonard Gérard : *Programmer le parallélisme avec des futures en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue dune compilation synchrone*. Thèse de doctorat, Université Paris Sud - Paris XI, septembre 2013. https://tel.archives-ouvertes.fr/tel-00929932. *[2 citations pages 79 et 89]*

[GG87] Thierry Gautier et Paul Le Guernic : *SIGNAL : A declarative language for synchronous programming of real-time systems*. Dans *Proceedings of Functional Programming Languages and Computer Architecture*, tome 274 de *Lecture Notes in Computer Science*, pages 257–277, Portland, Oregon, USA, septembre 1987. Springer. https://doi.org/10.1007/3-540-18317-5_15. *[cité page 35]*

[GGPP12] Léonard Gérard, Adrien Guatto, Cédric Pasteur et Marc Pouzet : *A modular memory optimization for synchronous data-flow languages : application to arrays in a lustre compiler*. Dans *Proceedings of the 2012 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '12)*, pages 51–60, Pékin, Chine, juin 2012. ACM. https://doi.org/10.1145/2248418.2248426. *[cité page 37]*

[GHKT14] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai et Xavier Thirioux : *Testing-Based Compiler Validation for Synchronous Languages*. Dans *Proceedings of the 6th International Symposium on NASA Formal Methods (NFM 2014), Houston*, tome 8430 de *Lecture Notes in Computer*

*Science*, pages 246–251, Houston, TX, USA, avril 2014. Springer. https://doi.org/10.1007/978-3-319-06200-6_19. *[cité page 119]*

[GPPT16] Fei Guan, Long Peng, Luc Perneel et Martin Timmerman : *Open source FreeRTOS as a case study in real-time operating system evolution*. Journal of Systems and Software, 118 :19–35, 2016. https://doi.org/10.1016/j.jss.2016.04.063. *[cité page 29]*

[Gud93] David Gudeman : *Representing Type Information in Dynamically Typed Languages*. Rapport technique 93-27, University of Arizona, Phoenix, AZ, USA, octobre 1993. *[cité page 67]*

[Gut97] Scott B. Guthery : *Java Card (Industry Report)*. IEEE Internet Computing, 1(1) :57–59, 1997. https://doi.org/10.1109/4236.585173. *[cité page 25]*

[Hal03] Dean W. Hall : *PyMite : A Flyweight Python Interpreter for 8-bit Architectures*. Dans *Proceedings of the First Python Community Conference (PyCon 2003)*, pages 26–28, Washington, DC, USA, mars 2003. http://ftp.ntua.gr/mirror/python/pycon/papers/pymite/index.html. *[cité page 26]*

[Hal05] Nicolas Halbwachs : *A synchronous language at work : the story of Lustre*. Dans *Proceedings of the 3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005)*, pages 3–11, Verona, Italie, juillet 2005. IEEE Computer Society. https://doi.org/10.1109/MEMCOD.2005.1487884. *[cité page 33]*

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond et Daniel Pilaud : *The synchronous data flow programming language LUSTRE*. Proceedings of the IEEE, 79(9) :1305–1320, Sep. 1991. https://doi.org/10.1109/5.97300. *[cité page 77]*

[HG16] Caleb Helbling et Samuel Z. Guyer : *Juniper : a functional reactive programming language for the Arduino*. Dans *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016)*, pages 8–16, Nara, Japon, septembre 2016. ACM. https://doi.org/10.1145/2975980.2975982. *[cité page 12]*

[HK07] Trevor Harmon et Raymond Klefstad : *A Survey of Worst-Case Execution Time Analysis for Real-Time Java*. Dans *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–8, Long Beach, CA, USA, mars 2007. IEEE Computer Society. *[cité page 168]*

[HPK+09] Kirak Hong, Jiin Park, Taekhoon Kim, Sungho Kim, Hwangho Kim, Yousun Ko, Jongtae Park, Bernd Burgstaller et Bernhard Scholz : *TinyVM, an efficient virtual machine infrastructure for sensor networks*. Dans *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems (SenSys 2009)*, pages 399–400, Berkeley, CA, USA, novembre 2009. ACM. https://doi.org/10.1145/1644038.1644121. *[cité page 25]*

[HR01] Nicolas Halbwachs et Pascal Raymond : *A tutorial of Lustre*. 2001. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.872. *[cité page 110]*

[HS02] Niklas Holsti et Sam Saarinen : *Status of the Bound-T WCET tool*. Dans *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, Vienne, Autriche, June 2002. http://www.cs.york.ac.uk/rts/wcet2002. *[cité page 160]*

[JGS93] Neil D. Jones, Carsten K. Gomard et Peter Sestoft : *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993, ISBN 0-13-020249-5. *[cité page 72]*

[Jon97] Mike Jones : *What really happened on Mars Rover Pathfinder*. ACM Forum on Risks to the Public in Computers and Related Systems, 19(49), 1997. http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html. *[cité page 30]*

[JRH19] Erwan Jahier, Pascal Raymond et Nicolas Halbwachs : *The Lustre V6 Reference Manual*. 2019. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf. *[cité page 79]*

[JWC08] Jeong Hoon Ji, Gyun Woo et Hwan Gue Cho : *A Plagiarism Detection Technique for Java Program Using Bytecode Analysis*. Dans *Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology*, tome 1, pages 1092–1098. IEEE Computer Society, novembre 2008. https://doi.org/10.1109/ICCIT.2008.267. *[cité page 23]*

[Kah74] Gilles Kahn : *The Semantics of Simple Language for Parallel Programming*. Dans *Proceedings of IFIP Congress 1974*, pages 471–475, Stockholm, Suède, août 1974. *[cité page 34]*

[Kat10] Yoshiharu Kato : *Splish : A Visual Programming Environment for Arduino to Accelerate Physical Computing Experiences*. Dans *Proceedings of the 8th International Conference on Creating, Connecting and Collaborating through Computing ($C^5$ 2010)*, pages 3–10, La Jolla, CA, USA, janvier 2010. IEEE Computer Society. https://doi.org/10.1109/C5.2010.20. *[cité page 12]*

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch et Simon Winwood : *seL4 : formal verification of an OS kernel*. Dans *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009 (SOSP '09)*, pages 207–220, Big Sky, MT, USA, 2009. ACM. http://doi.acm.org/10.1145/1629575.1629596. *[cité page 41]*

[KLW14] Robbert Krebbers, Xavier Leroy et Freek Wiedijk : *Formal C Semantics : CompCert and the C Standard*. Dans *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP 2014)*, tome 8558 de *Lecture Notes in Computer Science*, pages 543–548, Vienne, Autriche, juillet 2014. Springer. https://doi.org/10.1007/978-3-319-08970-6_36. *[cité page 41]*

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish et Scott Owens : *CakeML : A Verified Implementation of ML*. Dans *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, pages 179–191, San Diego, CA, USA, 2014. ACM, ISBN 978-1-4503-2544-8. http://doi.acm.org/10.1145/2535838.2535841. *[cité page 41]*

[Kri07] Jean-Louis Krivine : *A call-by-name lambda-calculus machine*. Higher-Order and Symbolic Computation, 20(3) :199–207, 2007. https://doi.org/10.1007/s10990-007-9018-9. *[cité page 51]*

[Lam77] Leslie Lamport : *Proving the Correctness of Multiprocess Programs*. IEEE Trans. Software Eng., 3(2) :125–143, 1977. https://doi.org/10.1109/TSE.1977.229904. *[cité page 38]*

[Ler90] Xavier Leroy : *The ZINC experiment : an economical implementation of the ML language*. Rapport technique 117, INRIA, Rocquencourt, France, février 1990. https://hal.inria.fr/inria-00070049/file/RT-0117.pdf. *[2 citations pages 26 et 52]*

[LF08] Francesco Logozzo et Manuel Fähndrich : *On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis*. Dans *Proceedings of the 17th International Conference on Compiler Construction (CC 2008)*, tome 4959 de *Lecture Notes in Computer Science*, pages 197–212, Budapest, Hongrie, mars 2008. Springer. https://doi.org/10.1007/978-3-540-78791-4_14. *[cité page 23]*

[LHS10] Michael W. Lew, Thomas B. Horton et Mark Sherriff : *Using LEGO MINDSTORMS NXT and LEJOS in an Advanced Software Engineering Course*. Dans *Proceedings of the 23rd IEEE Conference on Software Engineering Education and Training (CSEE&T 2010)*, pages 121–128, Pittsburgh, Pennsylvania, USA, mars 2010. https://doi.org/10.1109/CSEET.2010.31. *[cité page 25]*

[LL05] Benjamin Livshits et Monica S. Lam : *Finding Security Vulnerabilities in Java Applications with Static Analysis*. Dans *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA, 2005. USENIX Association. https://www.usenix.org/conference/14th-usenix-security-symposium/. *[cité page 23]*

[LM97] Yau-Tsun Steven Li et Sharad Malik : *Performance analysis of embedded software using implicit path enumeration*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 16(12) :1477–1487, 1997. https://doi.org/10.1109/43.664229. *[cité page 39]*

[Lop09] Bruno Cardoso Lopes : *Understanding and writing an LLVM compiler back-end (tutorial)*. Dans *ELC'09 : Embedded Linux Conference*, San Francisco, CA, USA, avril 2009. *[cité page 24]*

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha et Alex Buckley : *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st édition, 2014, ISBN 013390590X, 9780133905908. *[cité page 25]*

[MDLM18] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne et Gilles Muller : *Usuba : Optimizing & Trustworthy Bitslicing Compiler*. Dans *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing (WPMVP@PPoPP 2018)*, pages 4 :1–4 :8, Vienne, Autriche, février 2018. ACM. https://doi.org/10.1145/3178433.3178437. *[cité page 119]*

[Mil78] Robin Milner : *A theory of type polymorphism in programming*. Journal of Computer and System Sciences, 17(3) :348 – 375, 1978, ISSN 0022-0000. https://doi.org/10.1016/0022-0000(78)90014-4. *[2 citations pages 118 et 137]*

[MP05] Louis Mandel et Marc Pouzet : *ReactiveML : a reactive extension to ML*. Dans *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 82–93, Lisbonne, Portugal, juillet 2005. ACM. https://doi.org/10.1145/1069774.1069782. *[cité page 32]*

[MPP10] Louis Mandel, Florence Plateau et Marc Pouzet : *Lucy-n : a n-Synchronous Extension of Lustre*. Dans *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, tome 6120 de *Lecture Notes in Computer Science*, pages 288–309, Québec city,

Québec, Canada, juin 2010. Springer. https://doi.org/10.1007/978-3-642-13321-3_17. *[cité page 34]*

[MS17] Mahesh M et Sivraj P : *DrawCode : Visual tool for programming microcontrollers*. Dans *Proceedings of the 3rd International Conference on Advances in Computing,Communication Automation (ICACCA 2017)*, pages 1–6, Dehradun, Inde, septembre 2017. https://doi.org/10.1109/ICACCAF.2017.8344708. *[cité page 12]*

[MV13] Michel Mauny et Benoît Vaugon : *OCamlCC - Traduire OCaml en C en passant par le bytecode*. Dans *Actes des 24èmes Journées Francophones des Langages Applicatifs (JFLA 2013)*, Aussois, France, février 2013. https://hal.inria.fr/hal-00779721. *[cité page 73]*

[MV14] Michel Mauny et Benoît Vaugon : *Nullable Type Inference*. Dans *Proceedings of the 2014 OCaml Users and Developers Workshop (OCaml 2014)*, Gothenbourg, Suède, septembre 2014. https://hal.inria.fr/hal-01413294. *[cité page 122]*

[NPW02] Tobias Nipkow, Lawrence C. Paulson et Markus Wenzel : *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, tome 2283 de *Lecture Notes in Computer Science*. Springer, 2002, ISBN 978-3-540-45949-1. https://doi.org/10.1007/3-540-45949-9. *[cité page 41]*

[PAM+09] Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury et Jean-Louis Colaço : *Experience report : Using Objective Caml to develop safety-critical embedded tools in a certification framework*. Dans *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 215–220, Édimbourg, Royaume-Uni, août 2009. ACM. https://doi.org/10.1145/1596550.1596582. *[cité page 36]*

[PB19] Basile Pesin et Julien Bissey : *Programmation de haut niveau pour applications sur microcontrôleurs*. Rapport de travaux encadrés de recherche, Sorbonne Université, juin 2019. *[cité page 75]*

[Pes18] Basile Pesin : *Paradigmes de programmation alternatifs sur microcontrôleurs via l'OMicroB*. Rapport de stage, Sorbonne Université, juin 2018. *[cité page 68]*

[PFB+11] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla et David Lesens : *Multi-task Implementation of Multi-periodic Synchronous Programs*. Discrete Event Dynamic Systems, 21(3) :307–338, Sep 2011, ISSN 1573-7594. https://doi.org/10.1007/s10626-011-0107-x. *[cité page 30]*

[Pie02] Benjamin C. Pierce : *Types and programming languages*. MIT Press, 2002, ISBN 978-0-262-16209-8. *[cité page 38]*

[Pla88] John Plaice : *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, 1988. *[cité page 109]*

[Pla89] John Plaice : *Nested clocks : The LUSTRE synchronous dataflow language*. Dans *Proceedings of the 1989 International Symposium on Lucid and Intensional Programming*, pages 1–17, 1989. *[cité page 87]*

[Pou06] Marc Pouzet : *Lucid Synchrone - Tutorial and Reference Manual*. 2006. https://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf. *[cité page 37]*

[PSS98] Amir Pnueli, Michael Siegel et Eli Singerman : *Translation Validation*. Dans *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, pages 151–166, Lisbonne, Portugal, mars 1998. Springer. https://doi.org/10.1007/BFb0054170. *[cité page 138]*

[Rat92] Christophe Ratel : *Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE.* Thèse de doctorat, Université Joseph Fourier, Grenoble, France, 1992. https://tel.archives-ouvertes.fr/tel-00341223. *[cité page 215]*

[Reg12] Pablo Vieira Rego : *Integrating 8-bit AVR Micro-Controllers in Ada*. The Ada User Journal, 33(4) :301, 2012. *[cité page 24]*

[Rey98] John C. Reynolds : *Definitional Interpreters for Higher-Order Programming Languages*. Higher-Order and Symbolic Computation, 11(4) :363–397, 1998. *[cité page 44]*

[RP19] Mario Aldea Rivas et Héctor Pérez : *Leveraging real-time and multitasking Ada capabilities to small microcontrollers*. Journal of Systems Architecture - Embedded Systems Design, 94 :32–41, 2019. https://doi.org/10.1016/j.sysarc.2019.02.015. *[cité page 24]*

[RWT+06] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger et Bernd Becker : *A Definition and Classification of Timing Anomalies*. Dans *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET '06)*, tome 4 de *OpenAccess Series in Informatics (OASIcs)*, Dresden, Allemagne, juillet 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.WCET.2006.671. *[cité page 159]*

[SC16a] Jérémie Salvucci et Emmanuel Chailloux : *Memory Consumption Analysis for a Functional and Imperative Language*. Dans *Proceedings of the 1st international workshop on Resource Aware Computing (RAC 2016)*, tome 330 de *Electronic Notes in Theoretical Computer Science*, pages 27 – 46, Eindhoven, Pays-Bas, avril 2016. https://hal.sorbonne-universite.fr/hal-01420298. *[cité page 216]*

[SC16b] Rémy El Sibaïe et Emmanuel Chailloux : *Synchronous-reactive web programming*. Dans *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2016)*, pages 9–16, Amsterdam, Pays-Bas, novembre 2016. ACM. https://doi.org/10.1145/3001929.3001931. *[cité page 33]*

[Sen07] Koushik Sen : *Concolic testing*. Dans *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 571–572, Atlanta, Georgia, USA, novembre 2007. ACM. https://doi.org/10.1145/1321631.1321746. *[cité page 168]*

[SF09] Vincent St-Amour et Marc Feeley : *PICOBIT : A Compact Scheme System for Microcontrollers*. Dans *Revised Selected Papers of the 21st Internation Symposium on Implementation and Application of Functional Languages (IFL 2009)*, tome 6041 de *Lecture Notes in Computer Science*, pages 1–17, South Orange, NJ, USA, septembre 2009. Springer. https://doi.org/10.1007/978-3-642-16478-1_1. *[cité page 26]*

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana et Lionel Marcé : *Cheddar : a flexible real time scheduling framework*. Dans *Proceedings of the 2004 Annual ACM SIGAda International Conference*

*on Ada*, pages 1–8, Atlanta, GA, USA, novembre 2004. ACM. https://doi.org/10.1145/1032297.1032298. *[cité page 29]*

[SN08] Konrad Slind et Michael Norrish : *A Brief Overview of HOL4*. Dans *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, tome 5170 de *Lecture Notes in Computer Science*, pages 28–32, Montréal, Québec, Canada, août 2008. Springer. https://doi.org/10.1007/978-3-540-71067-7_6. *[cité page 41]*

[SNO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar et Rok Strnisa : *Ott : Effective tool support for the working semanticist*. Journal of Functional Programming, 20(1) :71–122, 2010. https://doi.org/10.1017/S0956796809990293. *[cité page 93]*

[SP06] Martin Schoeberl et Rasmus Pedersen : *WCET Analysis for a Java Processor*. Dans *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '06)*, pages 202–211, Paris, France, 2006. ACM, ISBN 1-59593-544-4. http://doi.acm.org/10.1145/1167999.1168033. *[cité page 23]*

[Spa17] Philip Sparks : *The route to a trillion devices*. rapport technique, ARM, juin 2017. *[cité page 10]*

[Spi89] Michael Spivey : *An introduction to Z and formal specifications*. Software Engineering Journal, 4(1) :40–50, 1989. *[cité page 41]*

[SSD+17] Anatoliy Shamraev, Elena Shamraeva, Anatoly Dovbnya, Andriy Kovalenko et Oleg Ilyunin : *Green Microcontrollers in Control Systems for Magnetic Elements of Linear Electron Accelerators*, pages 283–305. Springer International Publishing, Cham, 2017, ISBN 978-3-319-44162-7. https://doi.org/10.1007/978-3-319-44162-7_15. *[cité page 13]*

[STD85] *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985, pages 1–20, octobre 1985. http://doi.org/10.1109/IEEESTD.1985.82928. *[cité page 66]*

[Tea19] The Coq Development Team : *The Coq Proof Assistant, version 8.9.0*, janvier 2019. https://doi.org/10.5281/zenodo.2554024. *[3 citations pages 13, 41 et 93]*

[VB14] Jérôme Vouillon et Vincent Balat : *From Bytecode to JavaScript : the Js_of_ocaml Compiler*. Software : Practice and Experience, 44(8) :951–972, 2014. https://doi.org/10.1002/spe.2187. *[cité page 33]*

[VC19] Steven Varoumas et Tristan Crolard : *WCET of OCaml Bytecode on Microcontrollers : An Automated Method and Its Formalisation*. Dans *Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, tome 72 de *OpenAccess Series in Informatics (OASICs)*, pages 5 :1–5 :12. Schloss Dagstuhl, juillet 2019. https://doi.org/10.4230/OASIcs.WCET.2019.5. *[cité page 214]*

[VVC16] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *Concurrent Programming of Microcontrollers, a Virtual Machine Approach*. Dans *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS$^2$ 2016)*, Toulouse, France, 2016. https://hal.archives-ouvertes.fr/ERTS2016. *[2 citations pages 30 et 213]*

[VVC17] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs*. Dans *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, 2017. https://hal.archives-ouvertes.fr/JFLA2017. *[cité page 213]*

[VVC18a] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *A Generic Virtual Machine Approach for Programming Microcontrollers : the OMicroB Project*. Dans *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, 2018. https://hal.archives-ouvertes.fr/ERTS2018. *[cité page 213]*

[VVC18b] Steven Varoumas, Benoît Vaugon et Emmanuel Chailloux : *La programmation de microcontrôleurs dans des langages de haut niveau - Cours invité*. Dans *Actes des 29èmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, BANYULS, France, janvier 2018. https://hal.sorbonne-universite.fr/hal-01762414. *[cité page 214]*

[VVF18] Nicolas Valot, Pierre Vidal et Louis Fabre : *Increase avionics software development productivity using Micropython and Jupyter notebooks*. Dans *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS$^2$ 2018)*, Toulouse, France, janvier 2018. https://hal.archives-ouvertes.fr/ERTS2018. *[cité page 25]*

[VWC15] Benoît Vaugon, Philippe Wang et Emmanuel Chailloux : *Programming Microcontrollers in OCaml : The OCaPIC Project*. Dans *Proceedings of the 17th International Symposium on Practical Aspects of Declarative Languages (PADL 2015)*, tome 9131 de *Lecture Notes in Computer Science*, pages 132–148, Portland, OR, USA, juin 2015. Springer. https://doi.org/10.1007/978-3-319-19686-2_10. *[cité page 26]*

[Wal00] David Walker : *A Type System for Expressive Security Policies*. Dans *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, pages 254–267, Boston, Massachusetts, USA, janvier 2000. ACM. https://doi.org/10.1145/325694.325728. *[cité page 95]*

[WDS+10] W. E. Wong, V. Debroy, A. Surampudi, H. Kim et M. F. Siok : *Recent Catastrophic Accidents : Investigating How Software was Responsible*. Dans *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 14–22, juin 2010. https://doi.org/10.1109/SSIRI.2010.38. *[cité page 38]*

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat et Per Stenström : *The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools*. ACM Transactions on Embedded Computing Systems (TECS), 7(3) :36 :1–36 :53, mai 2008, ISSN 1539-9087. https://doi.org/10.1145/1347375.1347389. *[2 citations pages 39 et 40]*

**Résumé**

Les microcontrôleurs sont des circuits imprimés programmables nichés dans de nombreux objets de notre quotidien. En raison de leurs ressources limitées, ils sont souvent programmés dans des langages de bas niveau comme le C, ou en langage assembleur. Ces derniers n'offrent pas les mêmes abstractions et les mêmes garanties que des langages de haut niveau, comme OCaml. Cette thèse propose alors un ensemble de solutions destinées à enrichir la programmation de microcontrôleurs avec des paradigmes de programmation de plus haut niveau. Ces solutions apportent une montée en abstraction progressive, permettant notamment de réaliser des programmes indépendants du matériel utilisé. Nous présentons ainsi une première abstraction du matériel prenant la forme d'une machine virtuelle OCaml, qui permet de profiter des nombreux avantages du langage tout conservant une faible empreinte mémoire. Nous étendons par la suite OCaml avec un modèle de programmation synchrone inspiré du langage Lustre et permettant d'abstraire les aspects concurrents d'un programme. Une spécification formelle du langage est donnée, et plusieurs propriétés de typage sont par la suite vérifiées. Les abstractions offertes par nos travaux induisent par ailleurs la portabilité de certaines analyses statiques pouvant être réalisées sur le bytecode des programmes. Une telle analyse, servant à estimer le temps d'exécution pire-cas d'un programme synchrone, est alors proposée. L'ensemble des propositions de cette thèse constitue une chaîne complète de développement, et plusieurs exemples d'applications concrètes illustrant l'intérêt des solutions offertes sont alors présentés.

**Abstract**

Microcontrollers are programmable integrated circuit embedded in multiple everyday objects. Due to their scarce resources, they often are programmated using low-level languages such as C or assembly languages. These languages don't provide the same abstractions and guarantees than higher-level programming languages, such as OCaml. This thesis offers a set of solutions aimed at extending microcontrollers programming with high-level programming paradigms. These solutions provide multiple abstraction layers which, in particular, enable the development of portable programs, free from the specifics of the hardware. We thus introduce a layer of hardware abstraction through an OCaml virtual machine, that enjoys the multiple benefits of the language, while keeping a low memory footprint. We then extend the OCaml language with a synchronous programming model inspired from the Lustre dataflow language, which offers abstraction over the concurrent aspects of a program. The language is then formally specified and various typing properties are proven. Moreover, the abstractions offered by our work induce portability of some static analyses that can be done over the bytecode of programs. We thus propose such an analysis that consists of estimating the worst case execution time (WCET) of a synchronous program. All the propositions of this thesis form a complete development toolchain, and several practical examples that illustrate the benefits of the given solutions are thus provided.