Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Programmation sûre, expressive et efficace de circuits FPGA

par Loïc Sylvestre

Dirigée par Emmanuel Chailloux et Jocelyn Sérot

Soutenue publiquement le 25 novembre 2024

Devant un jury composé de :

Carlos Agon, Professeur, Sorbonne Université	*Président*
Emmanuel Chailloux, Professeur, Sorbonne Université	*Directeur de thèse*
Jean-Louis Colaço, Ingénieur distingué, ANSYS	*Examinateur*
Marco Danelutto, Professeur, Università di Pisa	*Rapporteur*
Mickaël Dardaillon, Maître de conférences, INSA Rennes	*Examinateur*
Laure Gonnord, Professeure, Grenoble INP	*Examinatrice*
Alan Schmitt, Directeur de Recherche, Inria Rennes	*Rapporteur*
Jocelyn Sérot, Professeur, Université Clermont Auvergne	*Directeur de thèse*

Remerciment

Je remercie chaleureusement mes rapporteurs, Marco Danelutto et Alan Schmitt, pour avoir relu en détail ce manuscrit et m'avoir apporté de précieux conseils qui ont contribué à améliorer la version finale. Tout aussi chaleureusement, je remercie Carlos Agon, Jean-Louis Colaço, Mickaël Dardaillon, et Laure Gonnord d'avoir accepté d'évaluer mon travail.

Un grand merci à vous deux, Emmanuel Chailloux et Jocelyn Sérot, pour le soin que vous avez mis à diriger ma thèse et à m'accompagner. Vous m'avez insufflé l'énergie, la bonne humeur et l'envie de me dépasser. J'ai pris beaucoup de plaisir à travailler avec vous.

Un grand merci aussi à Antoine Miné qui a œuvré dès le départ pour m'accueillir en thèse dans l'équipe APR. APR est un lieu fort chaleureux où faire une thèse et enseigner. Au fil des années, j'y ai côtoyé *Abdelraouf, Ada, Alice, Antoine (G), Annick, Boubacar, Christine, Émilie, Eve, Francesco, Frédéric, Ghiles, Guillaume, Hector, Jules, Julien, Keanu, Mamy, Marco, Martin, Maryse, Mathieu (D), Mathieu (J), Mathieu (P), Mattéo, Mehdi, Michèle, Milla, Mohamed, Pascal, Pierre (N), Pierre (G), Raphaël, Romain, Sébastien, Sylvain, Steven, Thomas, Tong et Vincent* ; merci à toutes et tous. J'en profite pour passer dans les couloirs d'à côté et remercier *Daniela, Emmanuelle, Jean-Paul, Marie-Minerve et Roselyne* pour nos échanges.

J'ai été heureux de prendre part à l'aventure de l'IRILL, depuis 2019, avec *Emmanuel, Hector, Gabriel, Olivier, Ralf et Vincent notamment*. Merci aussi à Simão Melo de Sousa pour notre collaboration (avec *Emmanuel, Hector et João*) dans le cadre du programme franco-portugais Pessoa. Merci à Danaël Carbonneau et Adrien Challe pour leur excellent travail de stage avec Emmanuel et moi. Merci aussi à Colin Gonzales, Basile Pesin, Léo Andrès et Milla Valnet avec qui j'ai eu la joie d'organiser le Meetup OCaml (OUPS).

Les discussions techniques avec Mary Sheeran, Michael Mendler, Marc Pouzet, Dumitru Potop-Butucaru et Robert de Simone – en marges des parties de baby-foot avec *Baptise, Basile, Émilie, Fabien, Grégoire et Paul*, aux JFLA et à SYNCHRON – ont certainement contribué aux résultats présentés dans ce manuscrit. Je me réjouis de rejoindre l'équipe Inria PARKAS en post-doc pour travailler avec Timothy Bourke et Marc Pouzet.

La thèse a été ponctuée de pauses aux *Corsi di Musica Antica a Magnano* : je remercie chaleureusement mes professeurs, Eva et Georges Kiss, Anastas Demitriades, Lucas Tacardi et Albert Muehlboeck ainsi que mes amis et amies, *Yvonne, Rudy, May, Vincenzo, Giacomo*.

Merci à Mikhail Malt pour m'avoir accompagné il y neuf ans à l'IRCAM, alors que je me décidais à étudier l'informatique. J'étais tombé sur « *LISP, langage d'un autre type* » ; le déclic est venu de là.

Je remercie enfin toute ma famille – notamment mes parents Thérèse et Dominique et mon frère Duncan – pour leur soutien, et enfin ma compagne, Aki, pour sa bienveillance.

Résumé

Les FPGA (*Field-Programmable Gate Arrays*) sont des circuits numériques reconfigurables : leur comportement est personnalisable par synthèse logique de spécifications au niveau transfert de registres (niveau RT), dans des langages de description de matériel comme VHDL ou Verilog. Les FPGA sont bien adaptés pour réaliser des systèmes réactifs, directement sous forme de circuits synchrones interagissant avec l'environnement extérieur via des broches d'entrées/sorties – le synthétiseur logique garantissant que les contraintes temporelles du système sont respectées compte tenu de la fréquence d'horloge du FPGA cible. Les FPGA sont par ailleurs utilisés pour implanter des accélérateurs matériels. Cependant, les descriptions au niveau RT de systèmes transformationnels (ou « calculs ») – ayant des latences de plusieurs cycles d'horloge – sont difficiles à mettre au point, maintenir et optimiser manuellement. La synthèse haut niveau (*High-Level Synthesis, HLS*) constitue alors une solution pour exprimer plus simplement les calculs, dans un langage de programmation compilé au niveau RT. L'avantage de cette approche est de cacher aux programmeuses et programmeurs les détails d'implantation, en laissant au compilateur le soin d'ordonnancer les calculs au cours du temps. Cela conduit toutefois à une perte de contrôle sur le comportement temporel et donc la sûreté et l'efficacité des circuits engendrés. À l'heure où les systèmes embarqués, notamment à base de FPGA, doivent exécuter toujours plus de calculs, tout en interagissant avec leur environnement, cette thèse propose un modèle de programmation pour combiner description de matériel (orientée flot de données) et calcul parallèle généraliste (orienté flot de contrôle) en suivant une approche synchrone. Cela donne lieu à la conception et l'implantation d'Eclat, un langage de programmation fonctionnel-impératif, parallèle et synchrone, compilé en VHDL. Eclat est suffisamment précis pour décrire des circuits synchrones au niveau RT. Il permet de programmer simplement des accélérateurs matériels, avec une sémantique temporelle claire et prédictible par laquelle exploiter des compromis temps-espace. Tout programme Eclat est réactif, avec un mécanisme pour plonger les calculs dans les programmes et ainsi mélanger calcul et interaction. Eclat offre aussi de la mémoire partagée (réalisée sous forme de blocs RAM), conciliant concurrence et déterminisme. Il sert à développer des abstractions de programmation telles que des squelettes algorithmiques et des implantations de machines virtuelles pour des langages de haut niveau. Cela répond, à différents niveaux, au besoin d'intégrer des algorithmes généralistes à l'intérieur des applications embarquées réactives à base de FPGA.

Abstract

FPGAs (Field-Programmable Gate Arrays) are reconfigurable digital circuits : their behavior can be customized by logic synthesis of specification at the so-called register transfer level (RT level), in hardware description languages such as VHDL or Verilog. FPGAs are well suited to implement reactive systems, directly as synchronous circuits interacting with the external environment via I/O pins – the logic synthesizer ensuring that timing constraints are met, given the FPGA clock frequency. FPGAs are also used to implement hardware accelerators ; however, RT-level descriptions of transformational systems (or "computations") – with latencies of several clock cycles – are difficult to debug, maintain and manually optimize. High-Level Synthesis (HLS) offers a simpler way of expressing computations, using a programming language compiled at the RT level. The advantage of this approach is to keep the implementation details hidden from the programmer, leaving the compiler responsible for scheduling computations over time. However, this leads to a loss of control over temporal behavior and therefore safety and efficiency for the circuits generated. As embedded systems, especially those based on FPGAs, need to perform more and more computations, while interacting with their environment, this thesis proposes a programming model to combine hardware description (data-flow oriented) and general-purpose parallel computation (control-flow oriented) using a synchronous approach. This programming model forms the basis for the design and implementation of Eclat, a functional-imperative, parallel and synchronous programming language, compiled to VHDL. Eclat is sufficiently precise to describe synchronous circuits at the RT level. It facilitates the programming of hardware accelerators, with a clear and predictable temporal semantics by which to exploit time-space trade-offs. Any Eclat program is reactive, with a mechanism for embedding computations within programs and thereby mix computation and interaction. Eclat also offers shared memory (in the form of RAM blocks), with deterministic concurrency. It is used to develop programming abstractions such as algorithmic skeletons and virtual machine implementations for high-level languages. This addresses, at various levels, the need to run general-purpose algorithms within FPGA-based reactive embedded applications.

Sommaire

1 Introduction **1**

2 Conception et implantation de circuits sur FPGA **13**

3 Mixer calcul et interaction : le langage Eclat **27**

4 Sémantique d'Eclat **53**

5 Implantation d'Eclat sur FPGA **77**

6 Mixer parallélisme synchrone et mémoire partagée **127**

7 Développement d'applications sur FPGA avec Eclat **145**

8 Travaux connexes **183**

9 Conclusion et travaux futurs **195**

A Synthèse d'Eclat sur le FPGA Intel Max10 **203**

B Définition d'Eclat **209**

C Prise en main du compilateur Eclat **233**

Bibliographie **241**

Table des figures **261**

Table des matières **267**

1 Introduction

Les FPGA (*Field-Programmable Gate Arrays*) sont des circuits intégrés numériques reprogrammables [37]. L'architecture d'un FPGA est formée d'*éléments logiques* reliés par une matrice d'interconnexions. Cette matrice comprend aussi d'autres composants matériels comme des blocs d'entrées/sorties et des mémoires. Le circuit en tant que tel est gravé sur le silicium, mais les éléments logiques et interconnexions entre composants matériels restent configurables. L'utilisatrice ou l'utilisateur spécifie le comportement du circuit dans des abstractions de programmation : du niveau *portes logiques* jusqu'aux langages de programmation généralistes [124].

1.1 Contexte

Le potentiel de reprogrammation des FPGA répond à des besoins dans un grand nombre de domaines. Les FPGA offrent en particulier du parallélisme à grain très fin [54] et des entrées/sorties physiques génériques. Ils servent à implanter des applications en médecine [158], en musique [164], en robotique [228], en physique [47], en vision par ordinateur [46], dans les réseaux de télécommunications [218], dans les centres de données [210], *et cætera*. Au sein de ces domaines d'applications, les FPGA tendent d'ailleurs à être de plus en plus utilisés pour l'implantation de systèmes embarqués critiques [23].

La configuration d'un FPGA est engendrée automatiquement par *synthèse logique* [179]. Ce processus transforme un programme, écrit dans un langage de description de matériel (*Hardware Description Language, HDL)* [211], vers un plan détaillé du circuit.

VHDL [67] et Verilog [209] sont les deux principaux langages de description de matériel [73]. Ce sont des langages d'instructions en style impératif *basés sur les évènements* (*event-driven*). Cependant, leur modèle d'exécution diffère fortement des langages de programmation classiques [10], en particulier dans la manière de traiter le parallélisme, la mémoire et la concurrence.

Le niveau d'abstraction des descriptions de matériel est appelé *niveau transfert de registres* (niveau RT) [16]. Au niveau RT, le comportement *fonctionnel* et *temporel* du circuit doit être spécifié précisément sous forme de transferts de données entre registres synchronisés à chaque *tick* d'une horloge physique.

Les outils de synthèse logique réalisent des analyses statiques pour estimer la fréquence d'horloge maximale pour une description de matériel donnée. Ces analyses tiennent compte des temps de propagation de l'électricité codant l'information dans le circuit synthétisé [178]. De cette manière, les durées inférieures à la période d'horloge peuvent être négligées : elles sont conceptuellement [1] *instantanées*.

Cela constitue un *modèle de calcul* [109] qui simplifie la gestion de la concurrence et des communications dans les systèmes tout en servant de cadre sémantique pour la vérification formelle de circuits [120].

1.2 Problématique

Tout circuit, tel que spécifié au niveau RT, lit des entrées et produit des sorties suivant une échelle de temps discret, en faisant abstraction des difficultés d'implantation que cela pose au niveau physique [84]. De ce fait, un circuit (qui est par nature un *système vibratoire* [27]) s'apparente à un *système réactif* [136] capable de réagir à toute variation d'entrées en produisant *instantanément* des sorties.

Il est aussi possible d'encoder au niveau RT des systèmes *transformationnels* [100], qui lisent des entrées provenant de l'environnement, effectuent un calcul et forcent l'environnement à attendre le résultat. De tels systèmes (avec une durée d'exécution bornée ou non) sont largement utilisés dans les services en nuage (*cloud services*) [159] et les accélérateurs matériels à base de FPGA [174].

Considérons par exemple un circuit de calcul de racine carrée entière [137]. Il s'exécute en un nombre variable de cycles d'horloge et comprend une sortie spéciale (souvent nommée ready, enable ou valid) indiquant d'une part si le calcul est prêt à recevoir une nouvelle entrée et d'autre part si le résultat est prêt à être lu. L'entrée du circuit peut varier à chaque cycle. Cependant, le calcul sur une entrée donnée *ignore* les nouvelles entrées et laisse la sortie indéfinie jusqu'à ce que le calcul se termine. Le circuit traite alors une nouvelle entrée.

Des comportements réactifs et transformationnels peuvent cohabiter sur un même FPGA : il suffit de les composer en parallèle au niveau RT. Les circuits correspondants peuvent même communiquer de façon déterministe en se synchronisant sur l'horloge physique. Cela permet de mélanger indifféremment *calculs* d'une part et *interaction* d'autre part sur un FPGA.

Ceci dit, et tant pour les calculs que pour l'interaction, le niveau de détail à fournir pour décrire un système au niveau RT est très important [68]. Les détails d'implantation, et notamment le comportement temporel du circuit, influencent directement le comportement fonctionnel du

1. En pratique, l'on cherche cependant à optimiser ces durées pour augmenter la fréquence d'horloge maximale et ainsi améliorer les performances ou répondre à des contraintes temporelles du système.

système à spécifier [2], tester et vérifier [122]. Cela motive le développement de nouvelles abstractions et de nouveaux traits de programmation pour maintenir des invariants corrects par construction et faciliter la programmation de systèmes réalistes.

1.3 État de l'art

La recherche en programmation FPGA a fait émerger des abstractions de programmation au delà du niveau RT. Nous distinguons principalement deux familles d'approches, qui seront détaillées dans cette section :

— Les approches classiques incluent la *synthèse de haut niveau* [61], les DSL [70], les langages de modélisation au niveau système [69], les implantations matérielles de langages de programmation généralistes et la description de matériel en style fonctionnel [48]. Ces approches offrent des constructions de programmation pour implanter des calculs sur FPGA. Elles varient en termes d'expressivité, de facilité de mise en œuvre, de précision des circuits engendrés et d'efficacité (en particulier dans la manière d'exploiter le parallélisme disponible au niveau RT).

— Les langages synchrones [165] sont spécialement adaptés à la conception et l'implantation de systèmes réactifs, y compris au niveau RT sur FPGA. Par exemple Esterel [29] a été utilisé pour le développement d'applications critiques sur FPGA [96].

Un contrôle fin des détails architecturaux exprimables au niveau RT reste essentiel dans deux types d'applications [75] :

— les applications de calculs efficaces, étant donnée que la fréquence d'horloge d'un FPGA (de l'ordre de 50 MHz sur de petit modèles) est de l'ordre de 100 fois plus faible que sur un processeur moderne,

— les applications à faible latence ou ayant des contraintes temporelles fortes (et en particulier les systèmes réactifs).

1.3.1 Synthèse de haut niveau

La *synthèse de haut niveau* (*High-Level Synthesis, HLS*) [142] est une approche populaire pour le développement d'applications de calcul sur FPGA. Elle consiste à programmer des circuits directement dans un sous-ensemble de C [72], C++ [123], OpenCL [219] ou encore Fortran [177] en s'appuyant sur des compilateurs optimisants (appelés *outils HLS*) pour engendrer une description de matériel au niveau RT. Les outils HLS découpent automatiquement le flot de contrôle du programme source impératif en une séquence de cycles d'horloge au niveau RT. Le parallélisme exploitable dans le programme source est généralement inféré, la sémantique du langage étant purement séquentielle [52]. Les programmeuses et programmeurs ont un contrôle partiel sur les décisions du compilateur en utilisant des annotations (ou *pragmas*)

2. Par exemple, l'accumulation des temps de propagation au sein du circuit peut rendre une solution, pourtant correcte fonctionnellement, invalide.

qui peuvent être ignorées par le compilateur [151]. Des analyses statiques permettent d'estimer le temps d'exécution des circuits engendrés dans certains cas [53].

Les travaux portant sur la HLS se sont attaqués à des traits de programmation avancés comme les pointeurs [185], la mémoire dynamique [223] et la récursion [208].

Pour faciliter le prototypage rapide d'applications [103], les approches à base de HLS cachent les détails d'implantation, et en particulier les aspects temporels, ce qui constitue une limitation majeure. Le comportement temporel des programmes sources est dicté par les résultats de compilation au niveau RT : ces résultats sont imprédictibles et d'une grande variabilité [161], notamment de par les techniques d'optimisations agressives mises en œuvre et les heuristiques d'*exploration de l'espace de conception* (*Design Space Exploration, DSE*) [34].

1.3.2 Langages dédiés

D'autres approches ont cherché à répondre aux difficultés de programmation au niveau RT, tout en gardant un contrôle plus fin sur les détails architecturaux, comparés aux approches HLS. Les langages dédiés (*Domain Specific Languages, DSL*) [115] sont spécialisés pour certains types d'applications. Par exemple, CAPH [187] est un langage spécialisé dans le traitement de flux temps-réel sur FPGA. Il repose sur un modèle d'*acteurs synchrones flot de données* (*synchronous dataflow, SDF*) [127] et propose des traits de programmation de haut niveau (comme du polymorphisme paramétrique et des fonctions d'ordre supérieur). Le langage est *précis au niveau cycle* (*cycle accurate*) : la sémantique formelle de CAPH prédit le comportement temporel des circuits engendrés.

1.3.3 Langages de modélisation au niveau système

Les langages de modélisation de systèmes comme BLUESPEC [152, 1] et SYSTEMC [157] sont basés sur des modèles transactionnels [102] à la frontière entre description de matériel et logiciel embarqué. Par exemple, SYSTEMC offre un mécanisme de canaux qui permet de séparer les calculs d'une part, et les communications d'autre part [89]. Le langage mélange plusieurs niveaux d'abstractions [157] pour la *conception conjointe matérielle-logicielle* (*hardware/software co-design*) [118].

1.3.4 Implantation de langages de programmation existants sur FPGA

Un FPGA peut être programmé dans un langage de programmation classique (comme C, PYTHON ou JAVA) en utilisant un processeur *softcore*. Un *softcore* [134] est un processeur implanté au niveau RT pour être synthétisé sur un FPGA. Il fonctionne suivant la fréquence d'horloge du FPGA cible, qui est comparable à celle d'un microcontrôleur. L'intérêt d'utiliser un *softcore* est que cela permet de personnaliser les entrées/sorties pour les besoins de l'application [3], et de s'interfacer avec d'autres circuits sur un FPGA (y compris avec d'autres *softcores*).

3. Cela sert notamment à implanter des microcontrôleurs pour pallier les problèmes d'obsolescence [92] dans les applications embarquées critiques.

LegUp [44] est une implantation du langage C sur un processeur *softcore* avec accélération matérielle : certaines fonctions C sont compilées au niveau RT et communiquent avec le *softcore*. Pylog [104] propose une approche similaire pour exécuter Python sur un processeur extérieur [2] communiquant avec le circuit FPGA. Un processeur *hardcore* (*Hard Processor System, HPS*) est un processeur pré-fabriqué placé sur une carte FPGA [8]. À la différence d'un *softcore*, un processeur *hardcore* n'est pas reconfigurable, mais sa fréquence de fonctionnement est bien plus rapide que celle du circuit FPGA.

Le langage Hume [97], initialement conçu pour la programmation de logiciel embarqué temps-réel, a été utilisé pour programmer des FPGA. Un programme Hume est formé d'une série de *boîtes*. Un système de coordination en style flot de données relie les boîtes du programme. L'implantation de Hume repose sur une machine virtuelle HAM (*Hume Abstract Machine*) qui a été portée sur plusieurs processeurs *softcore* coopérant en parallèle [3] sur FPGA. D'autre part, un sous-ensemble de Hume a été directement compilé au niveau RT [189] pour exploiter davantage le parallélisme disponible au *niveau portes logiques*. Cela met en exergue un lien entre programmation de systèmes embarqués temps réel et programmation parallèle.

JAIP [212] est une implantation matérielle de la machine virtuelle Java (*Java Virtual Machine*, JVM) écrite en VHDL et appelant un processeur *softcore* pour gérer le chargement dynamique de classes.

JikesRVM [133] est une JVM implantée sur un processeur *softcore* avec accélération matérielle de la gestion automatique de mémoire (*Garbage Collection*, GC).

SHard [180, 22] est une implantation d'un sous-ensemble de Scheme au niveau RT. La bibliothèque d'exécution de SHard comprend un tas dans lequel sont allouées des paires pointées $(car \cdot cdr)$ qui permettent d'encoder des listes, et donc la pile, les fermetures et d'autres structures de données. La récursion générale est supportée par traduction en récursion terminale en style *par passage de continuations* (*Continuation Passing Style, CPS*). La récursion terminale est implantée par un système d'acteurs. SHard offre un type *vecteur* primitif (c'est-à-dire, un tableau mutable Scheme alloué dynamiquement). Il comporte aussi une construction de programmation `(par ((`x_1 e_1`) (`x_2 e_2`)) `e`)` qui s'apparente à la forme spéciale `let` de Scheme, mais évalue les liaisons `(`x_1 e_1`)` et `(`x_2 e_2`)` en parallèle et synchronise les résultats avant d'évaluer l'expression e. Cela permet par exemple d'implanter un tri-fusion parallèle traitant récursivement deux moitiés de vecteur. Chaque fermeture est associée à une zone mémoire unique, dans un bloc de mémoire RAM (BRAM), qui contient les variables libres de la fonction. SHard n'a pas de récupérateur automatique de mémoire.

Functional Hardware [74] est un projet visant à implanter Haskell [105] au niveau RT. Le flot de compilation part d'une représentation intermédiaire du compilateur Haskell (*Glasgow Haskell Compiler, GHC*) et applique une série de transformations préservant la sémantique ; en particulier, les fonctions récursives générales sont réécrites sous forme de fonctions récursives terminales avec pile d'appel explicite [227]. La bibliothèque d'exécution du langage comprend un tas alloué dans une zone de mémoire RAM ainsi qu'un récupérateur automatique de mémoire spécialisé dans la gestion de données non modifiables (purement fonctionnel) [17].

Dans un travail antérieur, nous avons contribué au développement de l'environnement de programmation O2B (*OCaml on Board*) [188] basé sur l'implantation OMicroB (*OCaml on Microcontroller Boards*) [216] de la machine virtuelle OCaml. OCaml [129] est un langage de programmation fonctionnel-impératif, modulaire et objet avec typage statique, polymorphisme paramétrique et inférence de types dans la lignée du langage ML [141].

Dans l'environnement O2B, les programmes OCaml sont compilés en bytecode (en utilisant le compilateur bytecode ocamlc de la distribution OCaml). Le bytecode est embarqué dans un tableau C et lié à l'interprète de la machine virtuelle (VM) écrit en C ainsi qu'à sa bibliothèque d'exécution (*runtime*) qui comprend un gestionnaire automatique de mémoire. Le programme C résultant est compilé puis exécuté sur un processeur *softcore* instancié sur le circuit FPGA.

O2B supporte l'appel de circuits externes (décrits en VHDL au niveau RT) depuis des programmes OCaml. De plus, nous avons proposé une méthode de partitionnement des programmes OCaml en une partie "hôte" (exécutée sur le processeur *softcore*) et des accélérateurs matériels programmés dans un sous-ensemble d'OCaml appelé Macle (*ML Accelerator*) [202].

Macle est compilé en VHDL via une chaîne de compilation que nous avons développée, et qui est représentée sur la figure 1.1. Sur cette figure, une fonction Macle `f(x)` est définie dans le programme source OCaml, compilée en VHDL et synthétisée sous forme de circuit spécialisé sur le FPGA. Le circuit communique avec le processeur *softcore* via un bus mémoire. Le passage d'argument et l'attente du résultat de `f` sont réalisés par une fonction C engendrée par le compilateur Macle à partir du type de `f` ; cette fonction C est appelée (en tant que *fonction externe*) depuis le bytecode OCaml via la FFI (*Foreign function interface)* [91]. De cette manière, le programme OCaml peut appeler des circuits externes écrits en VHDL ou en Macle. Nous avons étendu le langage Macle avec des squelettes de parallélisme implantés au niveau RT dans le but de transformer efficacement des tableaux OCaml alloués en mémoire externe (SDRAM) via le bus mémoire du processeur *softcore* [201].

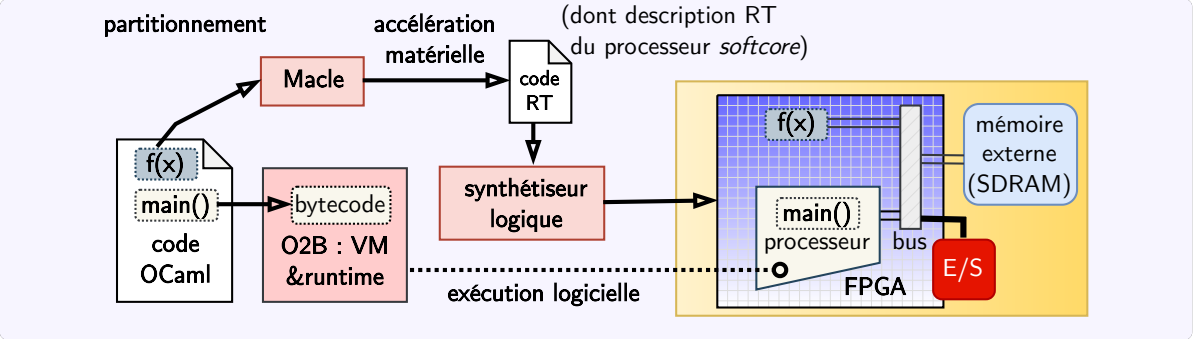


Figure 1.1 Chaîne de compilation de Macle et OCaml sur FPGA

Les *machines à réduction de graphes* constituent une autre technique d'implantation de langage fonctionnel [106]. Cette technique a été mise en œuvre principalement sur des processeurs classiques, mais des implantations sur FPGA ont aussi été proposé [148]. Par exemple, Heron [168] est une implantation matérielle d'un sous-ensemble d'Haskell. La machine Heron et son récupérateur automatique de mémoire [169] ont été implantés au niveau RT en utilisant le langage Cλash [12], qui est un langage de description de matériel en style fonctionnel.

1.3.5 Description de matériel en style fonctionnel

Certaines approches de description de circuits inspirées de la programmation fonctionnelle offrent un contrôle très fin sur la structure du circuit décrit (de plus bas niveau que le niveau RT) tout en maintenant une sémantique clair en terme de transfert de registres (de niveau RT).

Le langage μFP [195, 196] est un langage fonctionnel où chaque fonction représente un circuit. Chaque application de fonction crée une instance d'un circuit (en le dupliquant), connecte à ses entrées des arguments effectifs (qui sont les sorties d'autres circuits) et produit des valeurs en sorties. Des circuits *combinatoires* (c'est-à-dire, *sans* registres) sont représentés par des fonctions non récursives exprimées dans un dialecte du langage FP (*Functional Programming*) [14]. Un opérateur de point fixe μ, qui est spécifique à μFP, sert à décrire des circuits *séquentiels* (c'est-à-dire, *avec* registres) : μ appliqué à une fonction f représente un registre mis à jour par f à chaque *tick* d'horloge. La valeur initiale de l'état est une valeur spéciale notée " **?**" (*don't care*). La fonction f est une *fonction de transition* : elle prend en argument une entrée et un état suivant s' et retourne à la fois une sortie et un état courant s ; μf est une fonction à une entrée et une sortie, l'état étant implicitement géré par l'opérateur μ. La construction du circuit μf à partir d'une fonction f est illustrée graphiquement sur la figure 1.2.

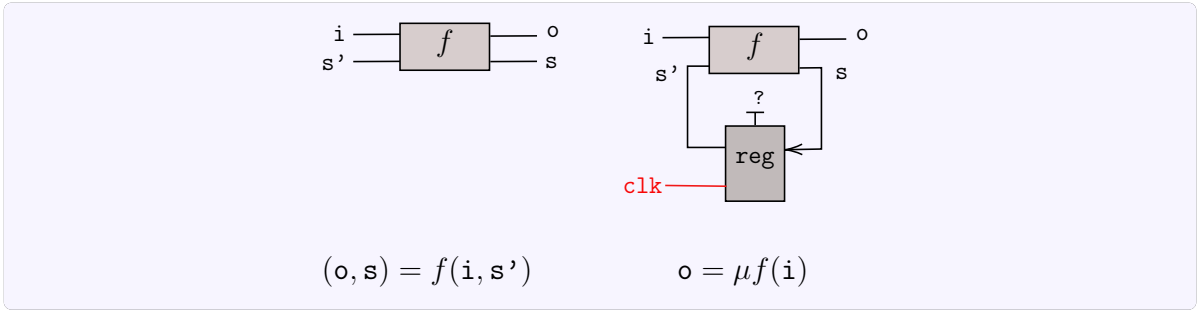


FIGURE 1.2 Construction d'une fonction avec état en μFP

Dans ce contexte, μFP et ses descendants manipulent des flots synchrones [85] ; chaque flot est une suite infinie de valeurs prises par un fil au cours du temps. Le temps s'écoule de façon discrète : c'est une suite de cycles d'horloge.

Le langage de description de matériel DAISY [110, 111] a été développé en même temps que μFP. Il est inspiré de PURE LISP et de SCHEME [112]. Chaque programme DAISY est un système d'équations récursives. Le langage fournit des flots en tant que structure de données inspirée du langage LUCID [11], ce qui permet de décrire naturellement des circuits. Les flots sont des *listes paresseuses* notées $\langle tête \ ! \ queue \rangle$.

DAISY permet d'exprimer des récursions terminales non imbriquées en style direct, à la SCHEME. Le style de programmation est alors orienté flot de contrôle, plutôt que flot de données : le programme calcule la valeur y d'une fonction récursive terminale $F(x)$ appelée avec un argument donnée a et des fonctions combinatoires g, h et p :

$$\begin{aligned} y &\stackrel{\text{def}}{=} F(a) \\ F(x) &\stackrel{\text{def}}{=} g(x) \text{ si } p(x) \text{ sinon } F(h(x)) \end{aligned}$$

La récursion se termine avec la valeur $g(x)$ dès que la condition d'arrêt $p(x)$ est vérifiée, et sinon se poursuit avec un appel récursif en mettant à jour l'argument x avec une fonction $h(x)$.

La définition récursive terminale est automatiquement synthétisée en un système d'équations de flots donné sur la figure 1.3. Celui-ci déroule dans le temps les étapes de récursion jusqu'à ce que la condition d'arrêt p soit vraie. Un signal spécial `rdy` indique à l'environnement l'instant à partir duquel le résultat y est disponible.

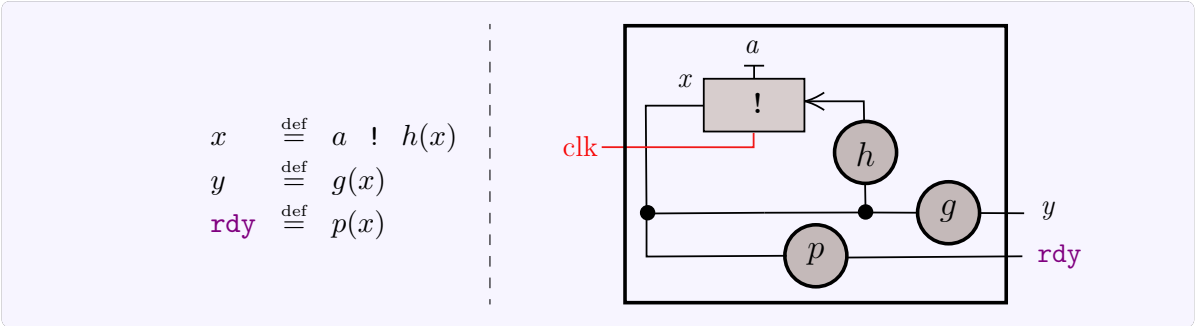


FIGURE 1.3 Implantation de la récursion terminale en DAISY

Cette transformation revient à "découper" en cycles d'horloge la formulation flot de contrôle pour obtenir une formulation flot de données. Cela montre la manière dont sont encodés les systèmes transformationnels au niveau RT et constitue la base des approches par synthèse de haut niveau (HLS, voir section 1.3.1).

1.3.6 Langages synchrones

Les langages synchrones reposent sur l'*Hypothèse synchrone* [165] suivant laquelle l'exécution d'un programme est ordonnée par une suite de *ticks* en *parfait synchronisme* [35]. Les communications *internes* au programme s'effectuent de façon instantanée à chaque *tick*, de même que l'interaction entre le programme et son environnement. Le modèle de programmation résultant constitue une méthodologie éprouvée pour la conception et l'implantation de systèmes embarqués réactifs [21], en logiciel ou en matériel, notamment dans des contextes embarqués critiques.

Il existe deux familles de langages synchrones :

— impératifs orientés flot de contrôle, comme ESTEREL [29],

— ou déclaratifs orientés flot de données comme LUSTRE [162] et SIGNAL [126].

En flot de contrôle, le programme progresse *instantanément* d'un pas dans son exécution, avec possible émission d'événements en sortie, à chaque occurrence d'événements en entrée.

En flot de données, le programme modélise la *fonction de transition*[4] du système réactif de manière analogue à un circuit formé de portes logiques et de registres. Le programme s'exécute

4. Cette notion a été définie à la section 1.3.5 dans le contexte de μFP, qui a aussi une sémantique flot de données synchrone [197].

instantanément, point à point sur des flots de valeurs prises par les entrées au cours du temps (une suite de *ticks*), ce qui produit des flots de valeurs de sorties.

Des extensions de langages synchrones proposent un mélange entre flot de contrôle et flot de données. C'est le cas des automates de modes *à la* ARGOS [135] dans certains dialectes de LUSTRE [57], comme SCADE [58] et VÉLUS [160].

Les implantations de langages synchrones s'appuient sur un langage hôte : généralement C ou ADA pour les implantations logicielles [24], et VHDL ou VERILOG pour les implantations matérielles [130]. Dans les deux cas, une analyse de pire temps d'exécution (*Worst Case Execution Time, WCET)* est nécessaire pour déterminer la fréquence de fonctionnement minimale du programme sur le matériel cible vis-à-vis des contraintes temporelles de l'environnement.

Les primitives du langage hôte sont supposées instantanées (c'est-à-dire, toujours plus rapide que la durée d'un cycle séparant deux *ticks*). Cependant, des mécanismes ont été proposés pour l'appel de tâches asynchrones en ESTEREL (via une construction de programmation appelée `exec`) [30], l'appel de calculs multi-cycles en QUARTZ [41] et la désynchronisation de programmes LUSTRE [55].

Des expériences académiques et industrielles ont été menées pour programmer des applications FPGA dans des langages synchrones, par exemple, en médecine [170]. Ils semble y avoir regain d'intérêt récent pour l'implantation de langages synchrones sur FPGA [7], notamment en vue de leur utilisation dans des applications spatiales [222]. Cela fait suite aux travaux précurseurs des années 1990 et 2000, qui se sont intéressés à l'implantation matérielle d'ESTEREL [25], LUSTRE [176] et SIGNAL [18], puis à l'optimisation de la compilation en circuits [81] avec des applications dans l'industrie [9]. Les fondements mathématiques des langages synchrones ont aussi été étudiés en lien avec le comportement des circuits au *niveau portes logiques* [184].

1.4 Synthèse : vers un langage synchrone généraliste

Les approches traditionnelles, comme la synthèse de haut niveau (HLS), sont largement utilisées pour la conception d'accélérateurs de calculs. Toutefois le comportement temporel des circuits engendrés est non spécifié, à l'exception notable de langages dédiés à la programmation temps-réel, tels que CAPH, ou des langages de modélisation de systèmes comme BLUESPEC.

Les langages synchrones ont été spécialement conçus pour programmer des comportements temporels expressifs de façon composable. Ils offrent de la concurrence déterministe [86], d'abord mis en œuvre pour la conception d'applications réactives embarquées, puis pour la programmation parallèle intensive de multi-processeurs [226]. Des modèles de programmation proches du modèle synchrone, comme le modèle *réacteur* [139], tendent aussi à appliquer des techniques de programmation réactive pour la programmation parallèle haute performance.

Dans cette thèse, nous développons un langage de programmation fonctionnel et parallèle généraliste pour la programmation d'applications réactives intégrant des calculs sur FPGA.

L'enjeu est de tirer parti de la cible FPGA pour concevoir un langage expressif, ici fonctionnel, sachant gérer les interactions et qui soit sûr (via du typage statique et des garanties

de réactivité) tout en étant efficace, car exploitant le parallélisme au niveau *circuit* (pour compenser les pertes de fréquence par rapport aux cibles logicielles conventionnelles).
La principale idée défendue dans le manuscrit est la suivante :

> Puisque dans les approches par synthèse de haut niveau (HLS), un langage généraliste est compilé au niveau RT en découpant l'exécution des programmes en suites de cycles d'horloge, nous pensons que le langage source, généraliste, gagnerait à être spécifié avec une sémantique synchrone. L'approche synchrone, en effet, apporte du déterminisme en programmation parallèle et *aiderait* à estimer les performances temporelles des programmes (en nombre de cycles d'horloge physique). Cela permettrait d'exploiter des techniques essentielles en programmation FPGA, telles que le compromis entre temps et espace et l'interaction avec le monde physique.

Nous mettons en œuvre cette idée à travers la conception et l'implantation d'Eclat[5] : un langage parallèle généraliste avec une sémantique synchrone, des constructions de programmation réactives et un schéma de compilation ciblant le niveau RT.

Eclat se distingue des langages existants implantés sur FPGA par sa manière de mélanger *flot de données* et *flot de contrôle*. Il repose sur un noyau fonctionnel strict inspiré d'OCaml s'inscrivant dans la continuité des techniques de compilation HLS orientées flot de contrôle et des implantations de langages de programmation de haut niveau sur FPGA. Mais à la différence de l'approche HLS traditionnelle, Eclat possède une sémantique synchrone, qui guide les décisions du compilateur :

> "*L'exécution de chaque construction de programmation*
> *est instantanée (elle prend* zéro *cycle)*
> *sauf les appels de fonctions récursives terminales*
> *qui introduisent un retard d'un cycle.*"

Les constructions *instantanées* sont utilisées pour décrire la fonction de transition (orientée flot de données) d'un programme réactif connecté au monde physique. Le programme peut avoir un état interne via une nouvelle construction de programmation appelée `reg` (ou *registre*).

Enfin, une nouvelle construction de programmation appelée `exec` sert à exécuter pas-à-pas des calculs généralistes (arbitrairement longs, mais avec un comportement temporel déterministe), à l'intérieur d'un programme instantané. Chaque calcul redémarre à chaque fois qu'il se termine. Rien ne garantit qu'un calcul termine sur une entrée donnée, mais il reste réinitialisable à tout moment depuis l'extérieur. Cela vise à mélanger calculs et interaction sur FPGA de manière structurée. La totalité d'une application est alors programmée en Eclat, garantissant ainsi la réactivité du système par construction.

Nous montrons, à travers des applications programmées en Eclat, la capacité du langage à s'appuyer sur des abstractions de programmation (calculs, typages, mémoire partagée, squelettes de parallélisme, machines virtuelles) tout en gardant le contrôle sur le comportement

5. *Langage* ***décla****ra****t****if***. Le nom d'Eclat fait référence à des langages déclaratifs existants tels que Lustre [162], Clash [12], Shard [180] et Macle [201].

temporel des programmes sources et la capacité à interagir de façon expressive avec le monde extérieur, suivant l'approche synchrone.

L'implantation matérielle d'Eclat est inspirée de celle du langage Macle [201] avec principalement deux différences. Premièrement, Eclat a une sémantique temporelle (contrairement à Macle) et le compilateur Eclat doit respecter cette sémantique. Deuxièmement, Eclat offre une approche générale pour programmer la configuration complète d'un FPGA, contrairement à Macle qui sert à programmer des accélérateurs matériels spécifiques pour personnaliser l'environnement d'exécution d'un programme OCaml "hôte" basé sur un processeur *softcore*.

Le compilateur Eclat est un logiciel libre disponible en ligne à l'adresse suivante :

https://github.com/lsylvestre/eclat

1.5 Structure du manuscrit

L'organisation des chapitres du manuscrit vise à répondre à deux objectifs : présenter d'une part une formalisation d'Eclat et de sa compilation, et montrer d'autre part la capacité qu'à Eclat à exprimer différents types d'applications, mixant calcul et interaction à plusieurs niveaux d'abstraction. L'ordre de ces chapitres traduit une montée en expressivité, allant de la description de circuit Eclat au niveau RT, jusqu'à l'implantation d'applications logicielles généralistes interagissant avec le monde physique.

Le chapitre 2 donne des prérequis minimaux en description de matériel. Il montre l'architecture interne d'un FPGA, la connexion entre un circuit et des entrées/sorties physiques et les principes de conception de circuits synchrones au niveau RT.

Le chapitre 3 présente Eclat en décrivant sa syntaxe et sa sémantique intuitive sur une série d'exemples.

Le chapitre 4 donne une formalisation de la sémantique d'Eclat. Elle inclut un système de types qui *peut* garantir qu'un programme donné reçoit des entrées et produit des sorties à chaque cycle d'horloge, de façon réactive.

Le chapitre 5 décrit les étapes de compilation d'Eclat vers des descriptions de matériel VHDL synthétisables pour reconfigurer des cartes FPGA.

Le chapitre 6 étend le modèle de programmation d'Eclat avec de la mémoire partagée implantée sous forme de blocs RAM. Cela vise à exploiter de manière sûre et efficace la mémoire disponible dans les circuits FPGAs. Cette possibilité fait d'Eclat un langage parallèle généraliste, en grande partie prédictible, qui supporte les accès mémoire concurrents de façon déterministe et équitable.

Le chapitre 7 présente des applications utilisant Eclat : un application réactive simple (PWM), un jeu vidéo (Pong), un programme de résolution de problèmes (n reines), un framework de parallélisme de haut niveau et une implantation de la machine virtuelle OCaml et de sa bibliothèque d'exécution en Eclat. Plusieurs mesures expérimentales sont données et montre l'effectivité d'Eclat et sa capacité et exploiter de façon efficace différentes formes de parallélisme sur une carte FPGA.

Le chapitre 8 présente les travaux connexes, tels que les langages SAFL [146], CλASH [12], CHISEL [13] et HANDEL-C [43].

Le chapitre 9 conclut ce travail et décrit une expérience d'enseignement menée avec ECLAT comme langage support. Il discute enfin des perspectives de recherche pour la programmation en ECLAT dans un contexte embarqué critique mixte avec du calcul et de l'interaction.

L'annexe A donne des exemples de circuits engendrés par le compilateur ECLAT associé à ce manuscrit. L'annexe B donne une version étendue de la syntaxe et de la sémantique d'ECLAT. L'annexe C présente, à l'aide d'exemples, comment utiliser le compilateur.

2 Conception et implantation de circuits sur FPGA

Les caractéristiques principales des circuits reconfigurables FPGA ainsi que les approches existantes pour les programmer ont été présentées au chapitre 1. Dans ce nouveau chapitre, nous détaillons l'architecture de ces circuits et leur programmation au *niveau transfert de registres* (niveau RT) en lien avec leur réalisation au *niveau portes logiques*.

2.1 Architecture

La figure 2.1 schématise l'architecture d'un FPGA. Celle-ci est constituée d'une matrice d'interconnexions configurable reliant principalement des *éléments logiques*[1] (*Logic Elements, LE*) et des blocs d'entrées/sorties (*E/S*).

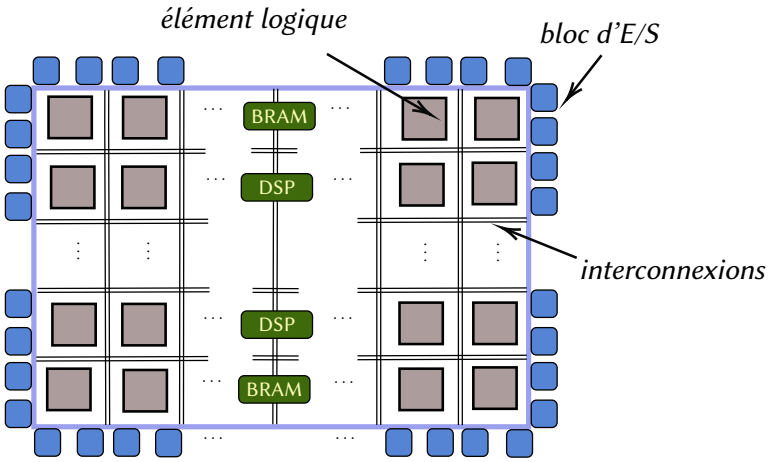


FIGURE 2.1 Architecture d'un FPGA

Cette matrice relie aussi des blocs de mémoire SRAM (*Block Random-Access Memory, BRAM*) [225] et des processeurs de signal numérique[2] (*Digital Signal Processors, DSP*). Tous ces éléments et leurs interconnexions sont configurables par *synthèse logique* et *placement/routage* à partir d'une description de matériel au niveau RT.

1. Plusieurs milliers à plusieurs millions d'éléments logiques suivant les modèles de FPGA.
2. Ces derniers sont notamment utilisés pour implanter efficacement des multiplieurs [66].

La synthèse logique d'une description de circuit engendre une *net-list*, qui spécifie la structure du circuit au *niveau portes logiques* ; puis le *placement/routage* instancie cette *net-list* sur la matrice d'interconnexions. Il en résulte un fichier de configuration (*bitstream*) qui, lorsqu'il est chargé dans la mémoire du FPGA [220], reconfigure statiquement [3] celui-ci pour implanter matériellement la fonctionnalité souhaitée.

La structure d'un élément logique comprend une ou plusieurs tables associatives (*Lookup tables, LUT*), une bascule *D* (*flip flop*), plusieurs multiplexeurs et parfois un additionneur [4] [132], comme schématisé sur la figure 2.2. L'élément logique qui y est montré comporte deux LUT, chacune réalisant une fonction combinatoire (`f` ou `g`) qui, étant donnés trois bits d'entrée, produit un bit de sortie. L'additionneur (`full_add`) réalise des opérations arithmétiques (d'addition et de soustraction sur 1 bit) étant donnée une retenue `ci` en entrée. Enfin la bascule *D* réalise un registre 1 bit synchronisé par l'horloge physique du FPGA (`clk`).

Cet élément logique est programmable via d'une part la fonction réalisée par les LUTs `f` et `g` et d'autre part les trois bits de sélections des multiplexeurs [5] (`sel0`, `sel1` et `sel2`). Ces derniers sont configurés à la valeur `0` ou `1`, par le synthétiseur logique, pour sélectionner un chemin personnalisé parmi les différents comportements réalisables avec ce seul élément logique.

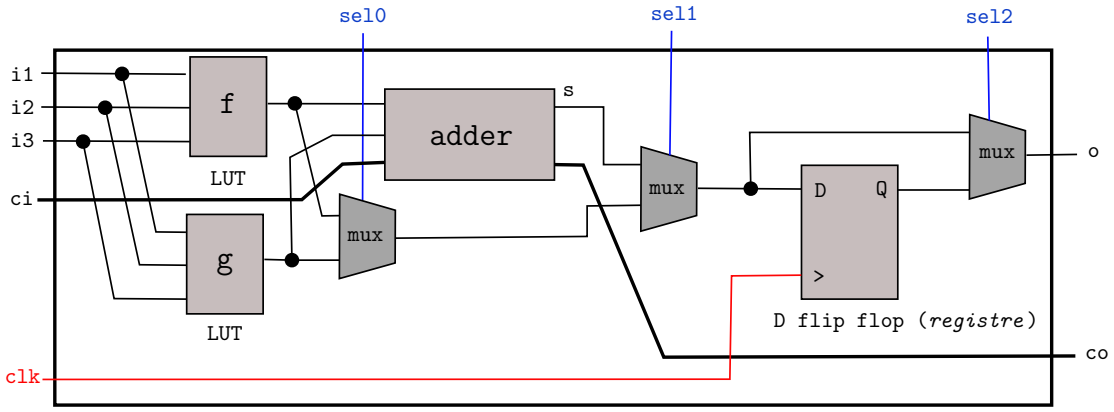


FIGURE 2.2 Structure d'un élément logique avec additionneur intégré

2.2 Interagir avec l'environnement extérieur

L'un des aspects principaux de la programmation FPGA est l'interaction avec un environnement extérieur (le *monde physique*).

Une carte FPGA est un système comprenant un circuit FPGA et d'autres composants matériels *pré-câblés*. Ces composants sont accessibles, depuis le FPGA, via ses blocs d'entrées/sor-

3. Certains FPGA sont reconfigurables dynamiquement. Cette caractéristique est notamment utilisée dans des missions spatiales critiques pour résister aux pannes tout en limitant l'intervention humaine [31].

4. L'architecture d'un FPGA varie suivant le constructeur, tels XILINX/AMD, INTEL (anciennement ALTERA), LATTICE ou encore ATMEL.

5. Un multiplexeur sélectionne l'une de ses entrées de données sur sa sortie en fonction de son entrée de sélection.

ties. Il s'agit de capteurs et actionneurs (comme par exemple des boutons et des LED), d'une mémoire externe (*Synchronous Dynamic Random-Access Memory, SDRAM*) [87] et parfois même d'un ou plusieurs processeurs *hardcore* [8].

La figure 2.3 montre un système en fonctionnement sur la plateforme d'évaluation DE10-lite de Terasic. Cette carte FPGA très simple intègre le circuit FPGA MAX10 d'Intel. Ce FPGA comporte 500 blocs d'entrées/sorties, 50 000 éléments logiques [6] et 1638 Kbits de mémoire RAM (BRAM). Certaines entrées/sorties sont pré-câblées à des dispositifs physiques présents sur la carte : une horloge de 50 MHz, dix interrupteurs, deux boutons, dix LED, six afficheurs 7 segments et une mémoire SDRAM externe.

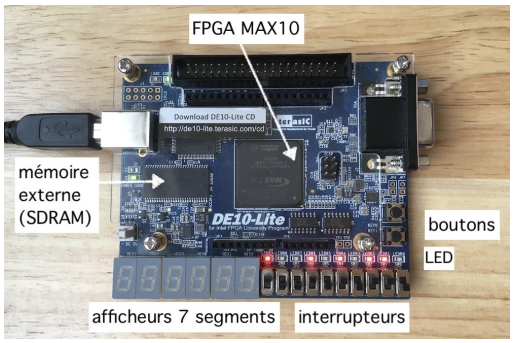


Figure 2.3 Une carte FPGA (DE10-lite) embarquant le circuit FPGA MAX10 d'Intel

Sur la figure 2.4, nous avons programmé au niveau RT un circuit *volontairement très simple*. Le langage de description de matériel utilisé ici est VHDL. Ce circuit connecte l'entrée `SW` à la sortie `LEDR`. `SW` désigne la rangée de dix interrupteurs tandis que `LEDR` désigne la rangée de dix LED. Autrement dit, chaque interrupteur allume ou éteint une LED. Cependant, la rangée de LED s'éteint (indépendamment des valeurs des interrupteurs) lorsque le bouton `KEY(0)` est pressé. Les entrées/sorties du circuit sont déclarées aux lignes 4 à 9. L'*architecture* du circuit est définie aux lignes 11 à 21. Celle-ci comporte un *processus combinatoire* exécuté à chaque changement de valeur de l'entrée `KEY` ou de l'entrée `SW`, comme spécifié par la liste `(KEY,SW)` à la ligne 14 (ou *liste de sensibilité* dans la terminologie VHDL).

La figure 2.5 montre la *net-list* engendrée par le *synthétiseur logique* Quartus II d'Intel à partir du programme de la figure 2.4. Les entrées du circuit sont représentées à gauche et les sorties sont à droite. La constante `10h0` est la constante `0` en hexadecimal (`h`) sur 10 bits. Le composant central est un multiplexeur qui propage en sortie l'une ou l'autre de ses entrées (sur 10 bits) suivant le sélecteur connecté à l'entrée `KEY(0)`. Cela correspond à l'interruption de la rangée de LED lorsque le bouton `KEY(0)` est pressé.

La figure 2.6 donne la réalisation matérielle du circuit sur le FPGA MAX 10. Le circuit comprend dix éléments logiques (`LOGIC_CELL_COMB`) qui implantent le multiplexeur 10 bits et 22 blocs d'entrées/sorties (`IO_BUF`) correspondant aux deux boutons, dix interrupteurs et dix LED.

6. Il existe des modèles de FPGA beaucoup plus puissants, comme le Stratix 10 GX/SX d'Intel qui comporte 10 millions d'éléments logiques.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

-- Interface
entity top is
  port (signal KEY : in std_logic_vector(0 to 1);     -- deux boutons
        signal SW : in std_logic_vector(0 to 9);      -- dix interrupteurs
        signal LEDR : out std_logic_vector(0 to 9)); -- dix LED
end entity;

-- Implantation
architecture rtl of top is
begin
  process (KEY,SW) begin        -- à chaque nouvelle entrée KEY ou SW
    if KEY(0) = '0' then        -- si le premier bouton n'est pas pressé
      LEDR <= SW;               -- alors connecter les interrupteurs aux LED
    else                        -- sinon
      LEDR <= "0000000000";     -- éteindre les LED
    end if;
  end process;
end architecture;
```

FIGURE 2.4 Description d'un circuit en VHDL

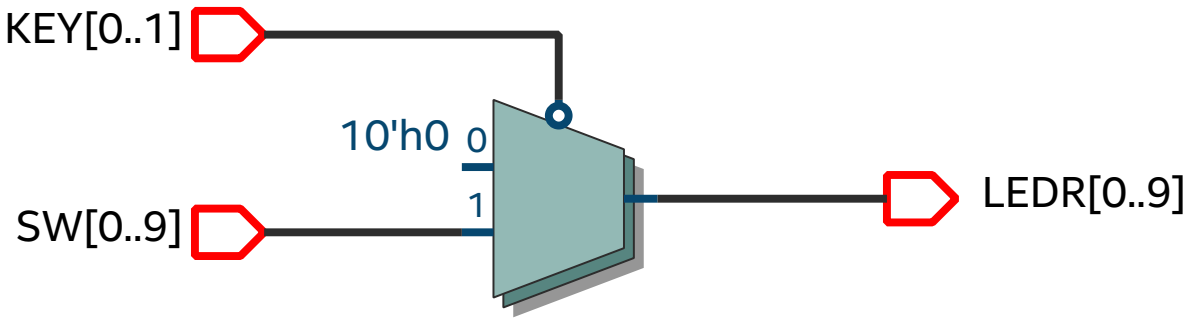


FIGURE 2.5 *net-list* obtenue par synthèse du module VHDL `top` de la figure 2.4

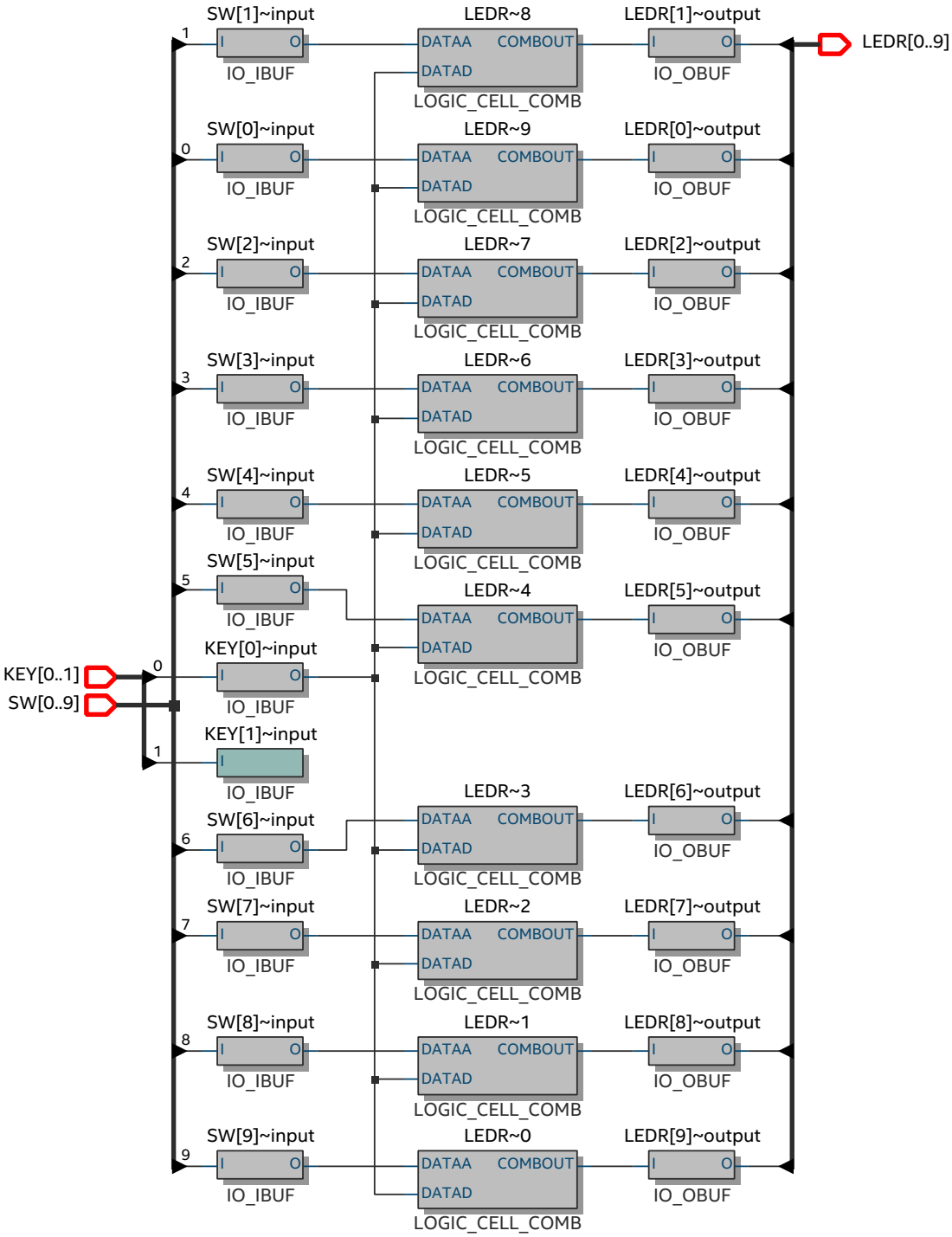


FIGURE 2.6 Réalisation du circuit `top` de la figure 2.5 sur le FPGA MAX 10

2.3 Circuits synchrones

Dans la section précédente, nous avons décrit un circuit très simple au niveau RT (figure 2.4). Le niveau RT permet de composer, explicitement ou par synthèse, des portes logiques et des registres. Il y a classiquement deux sortes de circuits : les fonctions combinatoires (qui dépendent uniquement de leurs entrées, comme le circuit mentionné ci-dessus) et les fonctions séquentielles (qui dépendent de leurs entrées et d'un état interne).

2.3.1 Fonctions combinatoires

Les circuits numériques reposent sur un codage de l'information par des signaux électriques. Le niveau logique `0` (*resp.* `1`) est représenté par un signal de tension *basse* (*resp. haute*). Ces signaux sont manipulés par des portes logiques (*not*, *and*, *or*, *xor*, *nor*, *nand*) réalisées avec des transistors, le plus souvent en technologie CMOS. La figure 2.7 montre le schéma d'un inverseur (porte *not*) en technologie CMOS. Celui-ci est obtenu en combinant deux transistors. `Vdd` désigne la tension haute de l'alimentation (correspondent au niveau logique `1`). Ce circuit inverse le niveau logique de son entrée `i` en sortie `o` : le niveau logique `0` devient `1` et le niveau `1` devient `0`.

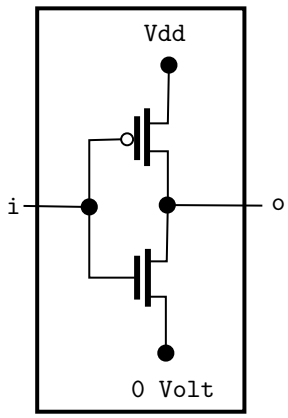


FIGURE 2.7 Inverseur CMOS

Les fonctions *combinatoires*, qui calculent des valeurs de sortie à partir de valeurs d'entrée au cours du temps, sont définissables en composant des portes logiques existantes. La sortie $O(t+\tau)$ d'une fonction combinatoire f au temps $t+\tau$ dépend de l'entrée $I(t)$ de la fonction f au temps t :

$$O(t+\tau) = f(I(t))$$

pour tout $t \geq 0$ et avec τ le temps de propagation du signal électrique dans le circuit réalisant la fonction f.

Au niveau conception, on fait en général abstraction des temps de propagation dans un premier temps et on modélise une fonction combinatoire, soit par un système d'équations, soit par une table de vérité dans le cadre de l'algèbre de Boole.

Exemple. Le circuit suivant est un *demi-additionneur* (*half-adder*) ; il compose des portes logiques élémentaires (xor et and) pour additionner deux bits a et b sans retenue en entrée et retourner la somme s ainsi qu'une retenue co :

$$\texttt{half_add}(a,b) = (s, co) \text{ où } \begin{cases} s = a \text{ xor } b \\ co = a \text{ and } b \end{cases}$$

Les circuits sont couramment représentés sous forme diagrammatique. La Figure 2.8 schématise le circuit `half_add` défini ci-dessus.

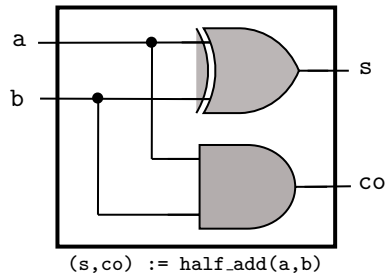


FIGURE 2.8 Demi-additionneur 1-bit

Un additionneur complet 1-bit prend deux valeurs a et b, plus une retenue d'entrée c (*carry in*) et retourne la somme s de a et b ainsi qu'une retenue sortante co (*carry out*). Ce circuit s'obtient en composant deux instances du circuit `half_add` :

$$\texttt{full_add}(a,b,ci) = (s, co) \text{ où } \begin{cases} (s_1, c_1) = \texttt{half_add}(a,b) \\ (s_2, c_2) = \texttt{half_add}(ci, s_1) \\ s = s_2 \\ co = c_1 \text{ or } c_2 \end{cases}$$

La figure 2.9 donne le circuit correspondant ; il comprend deux instances du circuit `half_add`, la sortie `s1` du premier étant connectée à l'entrée du second.

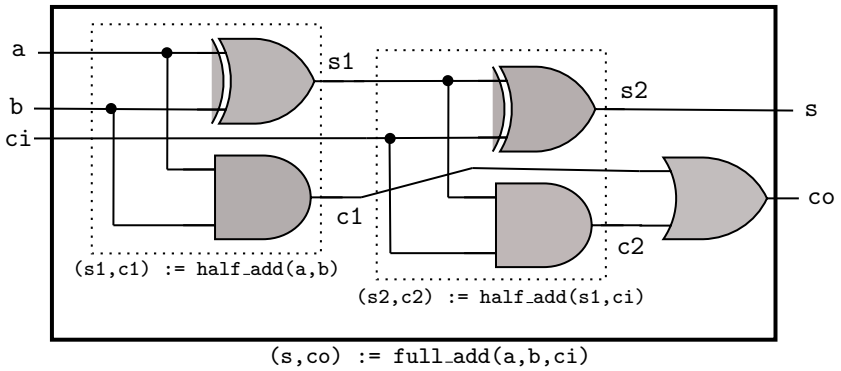


FIGURE 2.9 Additionneur complet 1-bit

2.3.2 Fonctions séquentielles

En rebouclant des portes logiques élémentaires, il est possible de construire des circuits *séquentiels* dont le comportement temporel sort du cadre des fonctions combinatoires.

Sur la figure 2.10, est défini un verrou (en anglais *latch*).

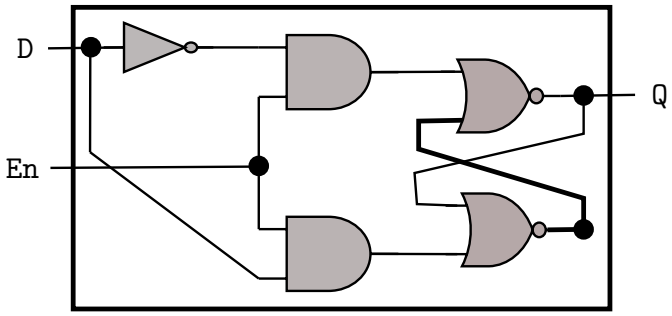


FIGURE 2.10 Verrou

Le verrou a deux entrées : un signal `D` (*donnée*) à mémoriser et un signal de contrôle `En` (*enable*). Ce circuit comprend une porte *not*, deux portes *and* et deux portes *nor*[7]. Lorsque `En` vaut `1`, la sortie `Q` vaut `D`. Lorsque `En` vaut `0`, `Q` conserve la valeur qu'avait l'entrée `D` lors du passage à `0` de `En` : les nouvelles valeurs de `D` sont ignorées. Cela se résume par la table suivante, où X désigne la valeur ignorée et $Q_{précédent}$ désigne la valeur mémorisée.

En	D	Q	
0	X	$Q_{précédent}$	pas de changement
1	0	0	écrit 0
1	1	1	écrit 1

Le chronogramme de la figure 2.11 illustre le comportement temporel du verrou pour des valeurs d'entrées `D` et `En` données évoluant au cours du temps (qui s'écoule de la gauche vers la droite). À ce stade, le temps est continu ; il n'y a pas encore de notion d'horloge. Lorsque l'entrée `En` est à `1`, la sortie `Q` recopie l'entrée `D`.

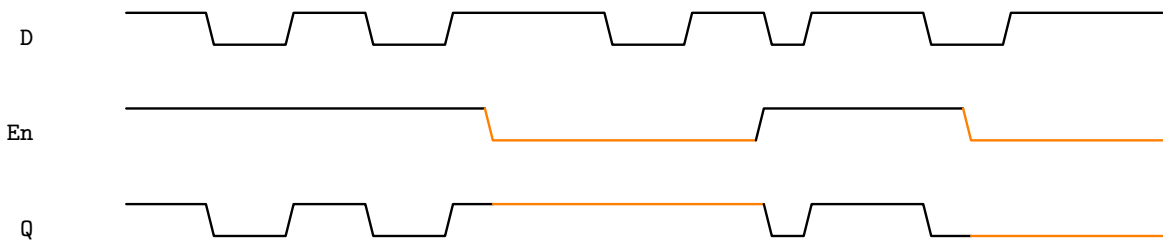


FIGURE 2.11 Comportement temporel d'un verrou (ou *latch*)

7. $\text{nor}(a, b) \equiv \text{not}(\text{or}(a, b))$.

La figure 2.12 donne la définition d'une bascule D (*flip-flop*) ; elle s'obtient en combinant deux verrous. La bascule D fonctionne comme suit : elle copie en sortie `Q` son entrée `D` à chaque fois que le signal d'horloge `clk` passe de 0 à 1 (*front montant d'horloge)* ; la valeur copiée est la valeur qu'avait `D` au moment du front montant de l'entrée `clk`.

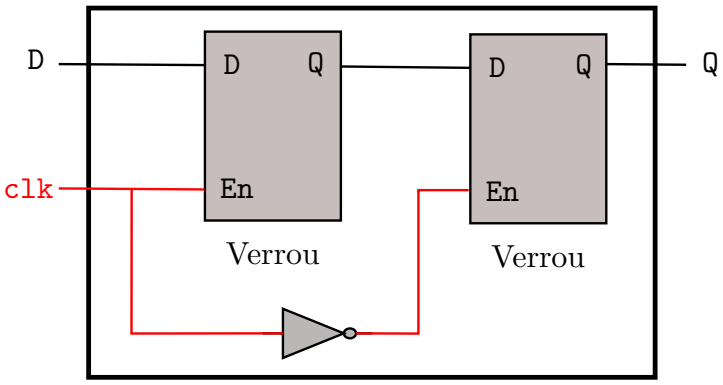


FIGURE 2.12 Bascule D

Les bascules D servent à réaliser des *registres* synchronisés par l'horloge. Cela permet notamment d'échantillonner un signal, venant par exemple d'une entrée physique comme l'illustre le chronogramme figure 2.13 pour un suite de valeurs d'entrée D donnée. Contrairement à la figure 2.11, le comportement temporel est ici synchronisé sur les fronts montants d'horloge (notés t_1, t_2, $\cdots$ sur la figure 2.13 et appelés "*ticks*"). La recopie de l'entrée `D` en sortie `Q` induit une latence de *au plus* un cycle. En particulier, la valeur de la sortie `Q` au cycle t_0 est indéfinie. L'entrée D peut varier librement entre deux fronts montants de `clk` mais les valeurs de D sont ignorées sauf celle qui précède immédiatement le deuxième front montant.

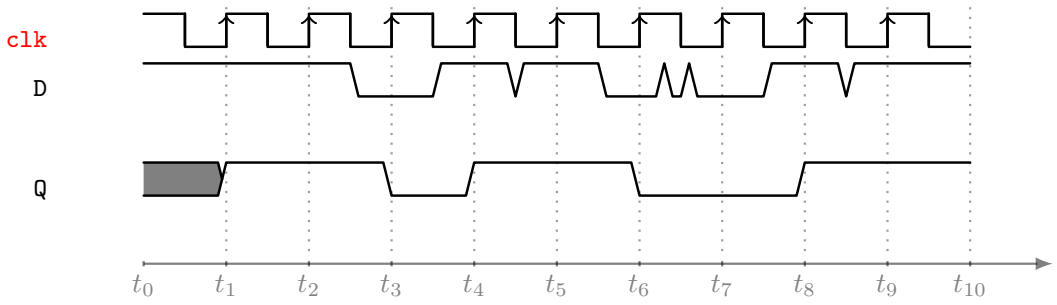


FIGURE 2.13 Comportement temporel d'une bascule D

La sortie d'une bascule D peut reboucler sur l'entrée pour implanter un schéma d'accumulation schématisé sur la figure 2.14. La sortie X de la bascule est connectée en entrée de la fonction combinatoire f. La sortie de f est connectée à l'entrée X' de la bascule ; X' est pris en compte à chaque front montant de l'horloge `clk`. On représente au dessus de la bascule sa valeur initiale[8] x_0.

8. Le schéma ne représente pas le dispositif matériel réalisant cette initialisation.

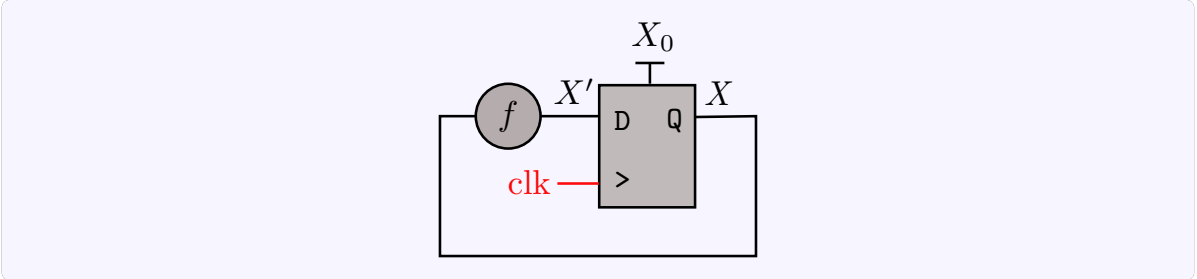


FIGURE 2.14 Schéma simplifié d'un circuit séquentiel

2.4 Modèles d'implantation des circuits synchrones

Les *circuits séquentiels* sont des circuits ayant un état interne ; ils sont construits en combinant des fonctions combinatoires et des registres. Les circuits séquentiels synchrones sont classiquement réalisés sous forme de *machines à états*, dont il existe principalement deux modèles d'implantation : les *Machines de Moore* [143] et les *Machines de Mealy* [138].

La structure de ces machines est donnée sur la figure 2.15 dans le cas Moore (à gauche) et Mealy (à droite). Elle repose sur un registre (au centre de chaque machine) qui mémorise l'état interne du système (ou *état courant*, noté X). L'état est mis à jour à chaque front montant de l'horloge `clk`. La nouvelle valeur de l'état (ou *état futur*, noté X') est calculée par la fonction combinatoire f à partir de l'état courant et de l'entrée i. La sortie o est définie par la fonction combinatoire g qui dépend :

— du seul état courant X dans une machine de Moore

— de l'état courant X et de l'entrée i dans une machine de Mealy.

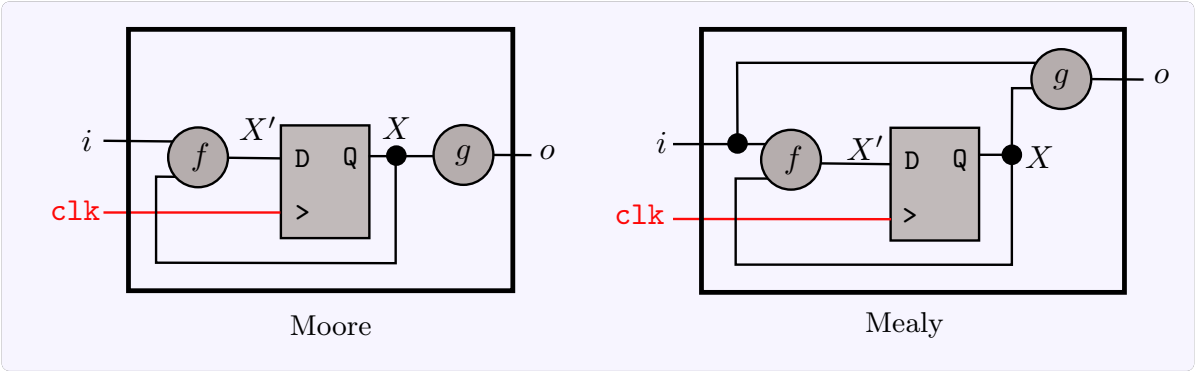


FIGURE 2.15 Structure des machines de Moore et Mealy

Le fonctionnement de ces machines est défini par des équations qui transcrivent les dépendances entre les entrées et les sorties des différents éléments du circuit séquentiel (c'est-à-dire, les deux fonctions f et g ainsi que le registre).

Les machines de Moore sont définies par :

$$\begin{aligned} X' &\stackrel{\text{def}}{=} f(X, i) \\ o &\stackrel{\text{def}}{=} g(X) \end{aligned}$$

Les machines de Mealy sont définies par :

$$\begin{aligned} X' &\stackrel{\text{def}}{=} f(X, i) \\ o &\stackrel{\text{def}}{=} g(X, i) \end{aligned}$$

Notons que ces équations établissent une relation combinatoire entre, d'une part, l'état présent X et l'entrée i, et d'autre part, l'état futur X' et la sortie o. Ces équations définissent la *fonction de transition* dictant l'évolution du circuit au cours du temps.

À chaque front montant de l'horloge `clk`, la sortie X du registre prend en compte la valeur X' et maintient cette valeur jusqu'au front montant suivant. La valeur de la sortie o est définie en continu suivant la seconde équation. Toutefois, dans le cas d'une machine de Moore, la valeur de la sortie ne dépend que de l'état courant : elle ne peut donc varier que lors des fronts montants de l'horloge. Pour une machine de Mealy, par contre, la sortie peut varier instantanément en réponse à tout changement de valeur de l'entrée entre deux *ticks* d'horloge.

2.5 Hypothèse de synchronisme au niveau RT

Les équations de Moore et de Mealy sont souvent formulées de la façon suivante, avec $t \in \mathbb{N}$:

$$\begin{aligned} X_{t+1} &\stackrel{\text{def}}{=} f(X_t, i_t) \\ o_t &\stackrel{\text{def}}{=} g(X_t) \end{aligned}$$

pour la machine de Moore et

$$\begin{aligned} X_{t+1} &\stackrel{\text{def}}{=} f(X_t, i_t) \\ o_t &\stackrel{\text{def}}{=} g(X_t, i_t) \end{aligned}$$

pour la machine de Mealy.

Il s'agit d'une abstraction synchrone du comportement du circuit où t est une grandeur *discrète*. Par exemple, X_t désigne la valeur de l'état courant au cycle d'horloge t ; autrement dit, X est un flot. Implicitement, cette abstraction se base sur l'hypothèse que i, X, X' et o ne changent de valeur que sur les fronts montants. Or la justification de cette abstraction est loin d'être évidente, pour trois raisons :

— Premièrement, l'entrée i peut varier à tout moment (voire même de multiple fois) au cours d'un cycle d'horloge. Or l'état futur X' désigne l'entrée d'un registre : la valeur de cette entrée est *prise en compte* uniquement au moment du front montant d'horloge : les variations entre les *ticks* d'horloge sont ignorées.

— Deuxièmement, dans le modèle des machines de Mealy, la valeur de la sortie change à la fois au moment du front montant d'horloge et à chaque variation d'entrée, ce qui engendre des valeurs intermédiaires non prédites par les équations synchrones définies ci-dessus.

La figure 2.16 met en évidence ce phénomène sur un chronogramme en faisant varier l'entrée au milieu du cycle (sur les fronts descendants d'horloge). Dans la première moitié de chaque cycle, la sortie dépend de l'état courant et de l'entrée commencée au cycle précédent. Ces valeurs sont représentées en rouge sur la figure.

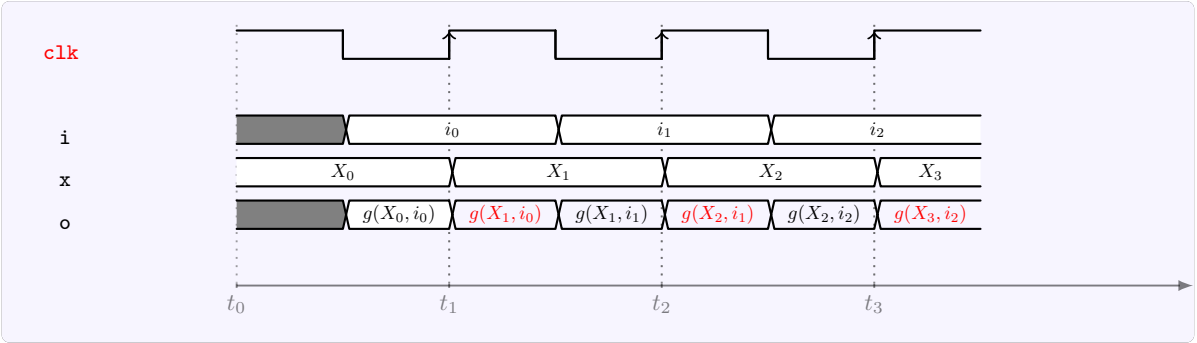


FIGURE 2.16 Comportement temporelle d'une machine de Mealy

— Troisièmement, au niveau physique, l'entrée ne doit jamais varier juste avant le front montant car elle doit rester stable pendant un certain temps, appelé *temps de setup* [181], avant d'être prise en compte au moment du front montant. Si cette propriété n'est pas satisfaite, des problèmes de métastabilité des registres peuvent apparaître, rendant le circuit non fonctionnel de façon non déterministe.

Pour justifier le premier point, il convient de faire l'hypothèse que la dynamique de l'environnement physique (l'intervalle de temps le plus court entre deux entrées) n'est pas plus rapide que celle du circuit : il y a au plus une variation d'entrée par cycle, ce qui justifie le modèle synchrone. Suivant le type d'applications, cette hypothèse est raisonnable en pratique puisque la fréquence d'horloge du FPGA cible est de l'ordre de la centaine de MHz, soient une période de l'ordre de la dizaine de nano-secondes, ce qui est très court. Si les variations d'entrée sont plus rapides que l'horloge de FPGA, alors il faut changer de FPGA cible pour disposer d'une horloge suffisamment rapide.

Pour justifier les points deux et trois, il suffit de *resynchroniser* la machine à états : cela consiste à placer un registre au niveau de l'entrée pour que celle-ci soit prise en compte, tout comme l'état, à chaque *tick* d'horloge. De plus, le registre permet d'éviter les phénomènes de métastabilité dans la machine à états lorsque l'entrée varie dans la *zone de setup* avant le *tick*.

Notons que cette resynchronisation de l'entrée induit forcément un retard dans la prise en compte de cette entrée, retard pouvant aller jusqu'à une période d'horloge, si la variation de l'entrée se produit juste après le front d'horloge. Ce retard est le prix à payer pour éviter les aléas de fonctionnement dues aux violations de *setup* dans le cas d'une entrée non synchronisée. Compte tenu des périodes d'horloges mises en jeu ici (de l'ordre de la dizaine de ns), ce retard est parfaitement acceptable [9] y compris pour des applications réactives, quitte à utiliser un FPGA avec un fréquence d'horloge plus haute pour satisfaire les contraintes de l'environnement.

9. En pratique, il peut même il y avoir plusieurs étages de resynchronisation de l'entrée. En effet, en échantillonnant l'entrée "primaire", par définition non contrainte, on peut potentiellement déclencher une violation

2.6 Composition de machines à états

Dans la suite du manuscrit nous nous plaçons dans un cadre où les entrées physiques sont resynchronisées, avec l'hypothèse que l'environnement ne varie pas plus vite que l'horloge du circuit. Dans cette section, nous nous intéressons à la composition de machines à états, de type Moore ou Mealy.

La figure 2.17 donne deux réalisations au *niveau portes logiques* d'un circuit d'accumulation. La réalisation de gauche (`sum_moore`) est une machine de Moore : sa sortie `s` ne dépend que de l'état courant, c'est-à-dire la sortie de la bascule D, qui induit une retard d'un cycle. Celle de droite (`sum_mealy`) est une machine de Mealy : sa sortie `o` dépend instantanément de la valeur de l'entrée `i`.

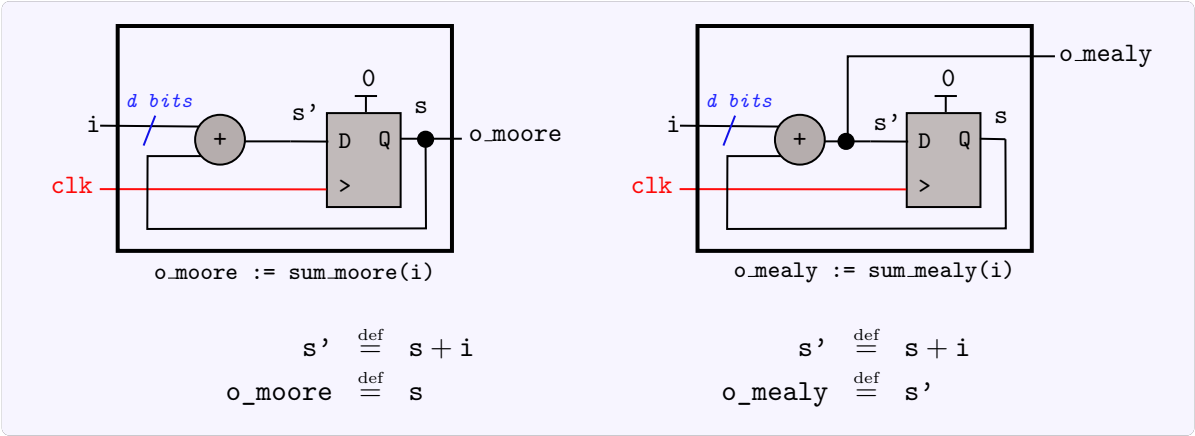


FIGURE 2.17 Deux réalisations d'un circuit d'accumulation

Le chronogramme sur la figure 2.18 montre que la machine de Moore répond au cycle suivant, tandis que la machine de Mealy répond instantanément.

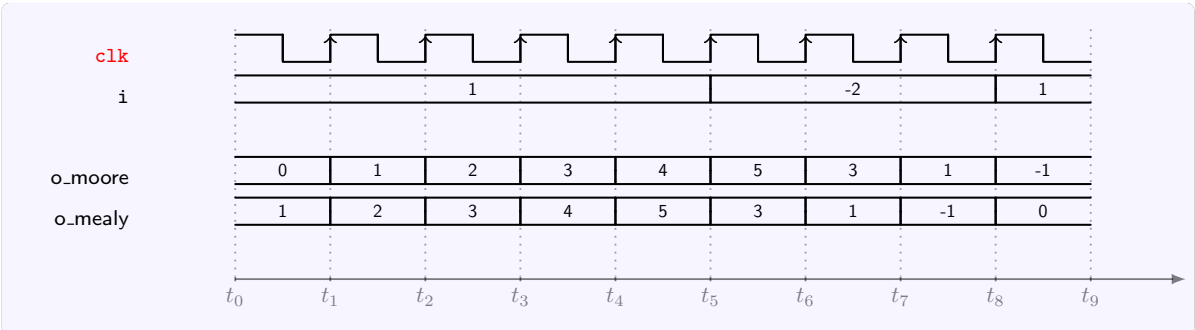


FIGURE 2.18 Comportement temporel de machines de Moore et de Mealy

de *setup*. Cette violation peut entraîner l'apparition d'une valeur instable en sortie du registre. Mais la métastabilité est forcément transitoire (quelques nanosecondes). Une cascade de rééchantillonnages permet de réduire la probabilité d'observer une telle valeur sur l'entrée "finale" à une valeur arbitrairement basse.

Considérons maintenant la composition de machine à états. La figure 2.19 compose *en série* des circuits en connectant la sortie d'un circuit à l'entrée d'un autre circuit. Les circuits considérés sont des instances de machines de Moore et de machines de Mealy. Le chronogramme associé montre que les retards d'un cycle s'accumulent lorsque plusieurs machines de Moore sont composées en série. La composition de machines de Mealy en série, par contre, n'introduit aucun retard.

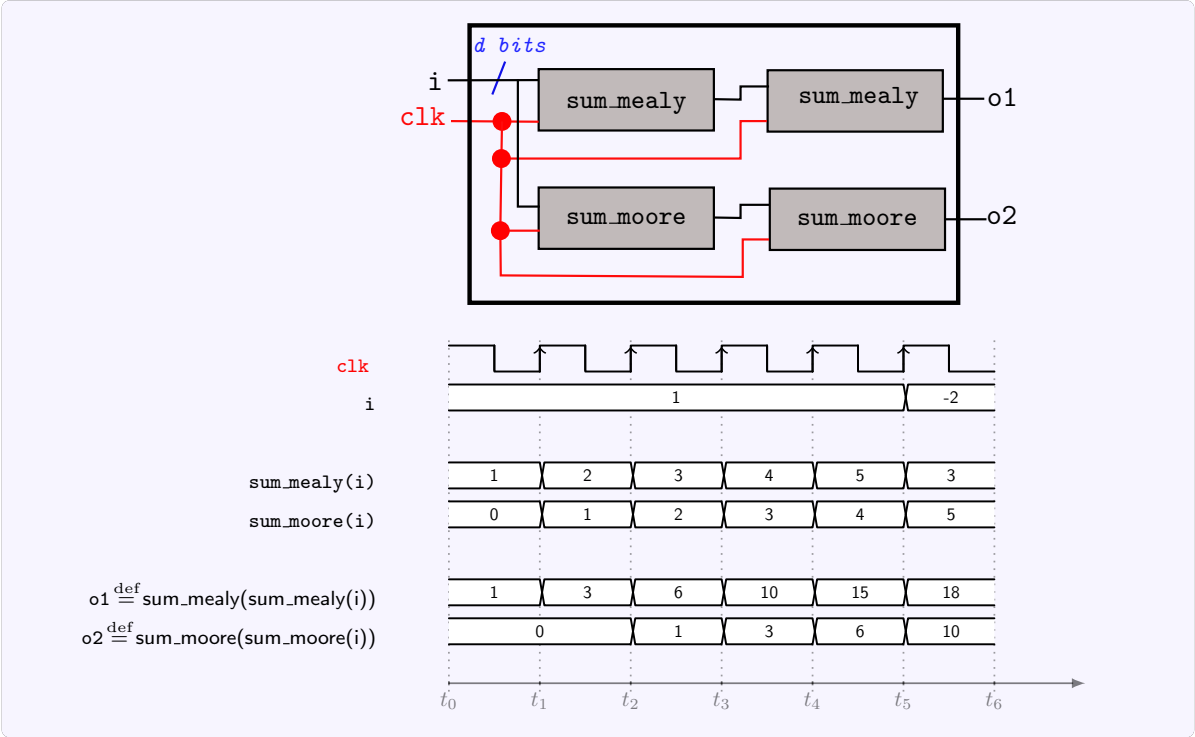


FIGURE 2.19 Compositions de machines à états

Dans la suite du manuscrit, nous nous plaçons dans un modèle de machine de type Mealy, qui est bien adapté pour la conception et l'implantation modulaire de systèmes réactifs. Les entrées physiques sont supposées *déjà* resynchronisées (en plaçant un registre entre chaque broche d'entrée et le circuit). À chaque cycle d'horloge, les circuits lisent une entrée et produisent une sortie avec une dépendance instantanée entre l'entrée (resynchronisée) et la sortie. Connecter la sortie d'un circuit à l'entrée d'un autre circuit introduit une dépendance entre l'entrée du premier circuit et la sortie du second ; cette dépendance aussi est instantanée.

En pratique, cette dépendance peut influencer le chemin critique (c'est-à-dire le temps de propagation maximal reliant les entrées aux sorties du circuit) et donc la fréquence maximale d'horloge. L'outil de synthèse logique calcule une approximation de ce pire temps de propagation. Cela s'apparente au calcul de pire temps d'exécution (*Worst Case Execution Time, WCET)* des systèmes embarqués réactifs. Si la fréquence maximale est trop basse vis-à-vis de l'horloge physique du FPGA cible, le synthétiseur indique statiquement que le circuit n'est pas réalisable correctement sur le FPGA.

3 Mixer calcul et interaction : le langage ECLAT

Disparaître peu à peu,
non simultanément, sur le geste du chef.
Ce geste sera donné après que le piano
aura commencé à jouer
Pierre Boulez, 1964
Éclat pour 15 instruments.

Ce chapitre présente ECLAT, un langage de programmation généraliste qui unifie *flot de données* et *flot de contrôle* dans un style de programmation applicatif statiquement typé *à la OCAML*. ECLAT est spécialement conçu pour faciliter la programmation de *systèmes embarqués réactifs* sur des circuits FPGA et pour incorporer des *calculs* arbitrairement longs dans les systèmes réactifs. Le langage repose sur l'hypothèse synchrone [165] : sa sémantique (définie au chapitre 4) et son implantation (présentée au chapitre 5) mettent en correspondance le temps logique du modèle synchrone avec l'horloge physique du FPGA cible.

3.1 Présentation générale

ECLAT est un langage d'expressions avec des constructions de programmation généralistes modélisées de façon synchrone : l'exécution d'une expression e est une suite possiblement infinie de réductions entrecoupées de *ticks* d'horloge et calculant une valeur v. L'intervalle entre deux *ticks* d'horloge est appelé un *cycle*. En ECLAT, chaque construction de programmation se réduit en *zéro cycle*, sauf l'appel de fonction récursive qui introduit un retard d'*un cycle* (à la manière de l'instruction `pause` d'ESTEREL [206]).

Certaines expressions s'exécutent toujours en *zéro cycle* : elles sont dites *instantanées*. Le caractère *instantané* d'une expression est garanti par typage statique. Par exemple, le type $\mathbf{bool} \xrightarrow{\mathbf{0}} \mathbf{bool}$ désigne les fonctions qui prennent un booléen et retournent *instantanément* un booléen. À l'inverse, le type $\mathbf{bool} \xrightarrow{\mathbf{1}} \mathbf{bool}$ désigne des fonctions qui prennent un booléen et retournent un booléen *en un temps inconnu* (zéro, un ou plusieurs cycles) qui dépend généralement des entrées du programme.

Un programme ECLAT est un système réactif modélisé par une fonction de transition instantanée. Celle-ci est connectée aux entrées/sorties physiques du circuit FPGA cible. Le programme est ré-exécuté à chaque cycle d'horloge avec l'entrée courante, ce qui a pour effet de transformer point à point un flot d'entrées en un flot de sorties et donc *interagir* avec l'environnement.

Le point essentiel est alors de pouvoir mélanger calculs et interaction de façon sûre, c'est-à-dire, sans risquer de perdre des entrées ou produire des sorties en retard.

À cette fin, ECLAT fournit une construction de programmation appelée `exec` qui exécute *pas à pas* une expression arbitrairement longue à l'intérieur du programme réactif. Ce mécanisme se distingue des *futures* [55] (notamment en LUSTRE), des *coroutines* [19] ou de l'appel de *tâches asynchrones* [30] (notamment en ESTEREL). En effet, les calculs ECLAT (qui sont ainsi exécutés pas à pas) progressent de façon synchronisée (*lockstep*). Cela assure le déterminisme et l'équité y compris en présence de parallélisme dans les calculs, voire de concurrence (comme ce sera le cas au chapitre 6).

ECLAT offre aussi une construction de programmation appelée `reg` (ou *registre*) pour sauvegarder des valeurs entre les cycles à la manière de l'opérateur μ de μFP [195].

La conditionnelle `if` e_1 `then` e_2 `else` e_3 en ECLAT est une structure de contrôle qui exécute uniquement la branche *then* ou la branche *else* en fonction de la valeur de la condition e_1 (comme en OCAML). Par exemple, si un bloc `exec` est placé dans la branche *then* d'une conditionnelle, le calcul sous le `exec` ne progresse que lorsque la branche *then* est exécutée, ce qui a pour effet de ralentir le calcul. Ainsi, ECLAT se caractérise par un *temps logique* [199] hérité des langages synchrones.

Organisation du chapitre La syntaxe d'ECLAT est définie à la section 3.2 en expliquant informellement le rôle de chaque construction du langage, la manière dont celui-ci est typé statiquement et la représentation des valeurs à l'exécution. Le mélange de calculs et d'interactions en ECLAT est expliqué à la section 3.3 sur une série d'exemples. Suivent des exemples variés qui illustrent les traits de programmation généralistes (section 3.4) et des techniques de programmation synchrone applicables en ECLAT (section 3.5). La formalisation de la sémantique d'ECLAT fera l'objet du chapitre 4.

3.2 Syntaxe

ECLAT est basé sur un λ-calcul en appel par valeur, typé statiquement avec polymorphisme paramétrique à la ML [63]. La syntaxe d'ECLAT est définie sur la figure 3.1. Elle comprend plusieurs catégories syntaxiques qui sont présentées dans les sous-sections suivantes.

La sous-section 3.2.1 présente le langage d'expressions. La sous-section 3.2.2 présente le langage de types. La sous-section 3.2.3 présente le langage de durées, qui sert à garantir la réactivité des programmes par typage statique. Les constantes et les opérateurs prédéfinis d'ECLAT sont présentés à la sous-section 3.2.4. Des abréviations et constructions dérivées sont ensuite introduites à la sous-section 3.2.5.

programme	π	::=	`let` f = `fun` `(`p`:`$\mathring{\tau}$`)` `:` $\mathring{\tau} \to e$
motif	p	::=	`()` | x | `(`p`,`p`)`
expression	e	::=	
constante			c
variable		|	x
opérateur		|	op
fonction		|	`fun` `(`p`:`τ`)` `:` $\mathring{\tau} \to e$
fonction récursive		|	`fix` f `(fun (`p`:`τ`)` `:` $\mathring{\tau} \to e$`)`
définition locale		|	`let` p`:`τ = e `in` e
application		|	$e\ e$
paire séquentielle		|	`(`e`,`e`)`
paire parallèle		|	`(`$e \| \| e$`)`
conditionnelle		|	`if` e `then` e `else` e
exec		|	$\texttt{exec}_\ell\ e$ `default` e `reset` e
registre		|	$\texttt{reg}_\ell$ `(fun (`p`:`$\mathring{\tau}$`)` `:` $\mathring{\tau} \to e$`)` `init` e
constante	c	::=	`()` | `true` | `false` | `(`l `:int<`θ`>)` | `(`c`,`c`)` | $\{c_1, \cdots c_n\}$
opérateur	op	::=	`not` | `&` | `or` | `xor` | `fst` | `snd`
			| `+` | `-` | `*` | `/` | `mod` | `<` | `>` | `<=` | `>=` | `=` | `resize_int<`θ`>`
			| `vect_create<`θ`>` | `vect_nth` | `vect_copy_with` | `vect_size`
type	τ	::=	
type de base			$\mathring{\tau}$
type de paire		|	τ `*` τ
type fonctionnel		|	$\tau \xrightarrow{\delta} \mathring{\tau}$
inconnue		|	$'a$
type de base	$\mathring{\tau}$	::=	
type de paire			$\mathring{\tau}$ `*` $\mathring{\tau}$
type de base prédéfini		|	`unit` | `bool` | `int<`θ`>` | $\mathring{\tau}$ `vect<`θ`>`
inconnue		|	$'B$
taille	θ	::=	
littéral			n
inconnue		|	$'N$
durée pire cas	δ	::=	
instantanée			$\mathbf{0}$
non-instantanée		|	$\mathbf{1}$
composition séquentielle		|	$\delta + \delta$
composition parallèle		|	$\max(\delta, \delta)$
inconnue		|	$'d$
schéma de type	σ	::=	
type			τ
type quantifié		|	$\forall\ 'v \cdot \sigma$
variable quantifiée	$'v$	::=	$'a$ | $'B$ | $'N$ | $'d$

FIGURE 3.1 Syntaxe d'ECLAT

3.2.1 Langage d'expressions

On se donne un ensemble dénombrable de variables dénotées par les lettres x, y, z, f, g et h. Un programme π est une fonction `let` f = `fun` `(`p`:`$\mathring{\tau}$`)` `:` $\mathring{\tau}' \to e$ (identifiée [1] par un nom f) s'appliquant point à point à un flot d'entrées, ce qui produit un flot de sorties. Au cours de l'exécution, la valeur v en tête du flot d'entrées est liée au motif p tandis que la valeur v' de l'expression e (qui dépend de p) est placée en tête du flot de sortie. Un motif p est soit le motif vide `()`, soit une variable x, soit une paire de motifs `(`p_1`,`p_2`)`. Les motifs sont utilisés dans les liaisons locales (`let/in`) et les fonctions (`fun` et `fix`) pour déconstruire des valeurs. Chaque fonction récursive `fix` f `(fun (`p`:`τ`)` `:` $\mathring{\tau}' \to e$`)` lie à son nom f sa propre définition dans l'expression e. Un opérateur *op* est une fonction prédéfinie unaire. Par exemple, l'opérateur d'addition `+` s'applique à une paire d'entiers et retourne un entier.

Deux constructions de programmation permettent de créer des paires. D'une part, `(`e_1`,`e_2`)` évalue *séquentiellement* l'expression e_1 en une valeur v_1, puis l'expression e_2 en v_2, puis construit la paire `(`v_1`,`v_2`)`. D'autre part, `(`$e_1 \| e_2$`)` évalue *en parallèle* l'expression e_1 en v_1 et l'expression e_2 en v_2, de façon synchronisée, puis construit la paire `(`v_1`,`v_2`)`. Lorsque les expressions e_1 et e_2 sont instantanées, il n'y a pas de différence entre les expressions `(`e_1`,`e_2`)` et `(`$e_1 \| e_2$`)`, qui sont toutes deux calculées instantanément avec un ordre de réduction gauche-droite.

La construction `exec` e `default` e_0 `reset` e_1 est une expression instantanée. Elle représente l'évaluation de l'expression e déroulée dans le temps (en zéro, un ou plusieurs cycles) et *répétée* indéfiniment. La valeur retournée par `exec` est soit **(1)** la paire `(`v`,true)` (lorsque l'évaluation de e se termine et produit une valeur v), soit **(2)** la paire `(`v_0`,false)` avec v_0 la valeur de l'expression instantanée e_0 (dite *valeur par défaut*). Chaque fois que l'évaluation de e se termine, elle recommence au cycle suivant dans l'environnement courant. L'expression e peut contenir des variables libres ; c'est pourquoi, le temps d'exécution de e dépend en général de l'entrée du programme. Les valeurs des variables libres dans e ne changent pas au cours de l'exécution : elles sont figées à chaque redémarrage du calcul de e. L'expression booléenne e_1 qui suit le mot-clé `reset` est instantanée : lorsqu'elle vaut `true`, le calcul de e est réinitialisé *immédiatement* au début du cycle (il redémarre son exécution dans le même cycle).

La construction `reg (fun (`p`:`$\mathring{\tau}$`)` `:` $\mathring{\tau} \to e$`)` `init` e_0 est une expression instantanée ; elle représente un registre initialisé avec la valeur de l'expression instantanée e_0. Chaque fois que cette construction est exécutée, la fonction instantanée `(fun (`p`:`$\mathring{\tau}$`)` `:` $\mathring{\tau} \to e$`)` est appliquée à la valeur mémorisée dans le registre. La valeur obtenue est retournée instantanément en sortie tandis que l'état du registre est mis à jour au *tick* d'horloge suivant avec la nouvelle valeur. Autrement dit, chaque bloc `reg` est une machine de Mealy.

Dans la syntaxe abstraite (figure 3.1), les constructions `reg` et `exec` sont annotées avec une étiquette ℓ qui n'apparaît pas dans les programmes sources. Elle sert à identifier chaque bloc `reg` ou `exec` avec un nom unique. Les annotations de types dans les lieurs `let/in`, `fun`, `fix` et `reg` peuvent être omises. Elles sont alors inférées par le système de types.

1. Ce nom f est le nom du circuit engendré par le compilateur ECLAT. Le nom sert à instancier le circuit ou à le composer avec d'autres circuits (cf. le mécanisme d'appel de fonctions externes, section 5.4 page 89).

3.2.2 Langage de types

Un type est soit un type de base $\mathring{\tau}$ (comme par exemple le type **bool** des booléens), soit un type de paire, soit un type fonctionnel, soit une inconnue de type. Le constructeur de type fonctionnel $\tau \xrightarrow{\delta} \mathring{\tau}'$ garantit que le type de retour est toujours un type de base $\mathring{\tau}'$. Cela autorise une forme limitée d'ordre supérieur, où les fonctions peuvent recevoir des fonctions en arguments mais ne peuvent pas en retourner. Le paramètre δ du constructeur $\tau \xrightarrow{\delta} \mathring{\tau}'$ représente un temps d'exécution pire cas ; son rôle est détaillé à la section 3.2.3. Une paire de types de base est aussi un type de base. Une distinction est faite entre inconnue de type $'a$ et inconnue de type de base $'B$. Par exemple, la *fonction identité* `fun` $x \to x$ a nécessairement un type de la forme $'B \xrightarrow{\delta} 'B$ puisque, le type de retour doit être un type de base (par construction).

Eclat offre du polymorphisme paramétrique à la ML [63]. Les noms introduits par la construction `let/in` sont associés à des schémas de types (désignés par la métavariable σ). Ces schémas de types sont obtenus en généralisant, dans le type de chaque fonction liée à un nom par un `let/in`, les inconnues de types de base ($'B$), de types ($'a$), de tailles ($'N$) et de durées ($'d$) qui n'apparaissent pas libres dans l'environnement de typage. Un schéma de type est soit un type τ, soit une quantification de variable $(\forall\ 'v\ .\sigma)$ en tête d'un schéma de type σ.

3.2.3 Langage de durées

Comme le langage d'expressions d'Eclat permet d'exprimer à la fois des calculs arbitrairement longs et des comportements instantanés, une analyse statique est nécessaire pour garantir qu'un programme Eclat termine toujours instantanément quelle que soit son entrée et quels que soient les calculs exécutés. Cette analyse doit abstraire le pire temps d'exécution de tout programmes[2] ; par exemple, la durée de la fonction `compose` définie par `fun ((`f`,`g`),`x`)` $\to$ $f(g\ x)$ dépend des durées des arguments fonctionnels f et g.

C'est pourquoi nous proposons une analyse de pire temps d'exécution à base de typage statique. Cette analyse est définie au chapitre suivant (section 4.1) par un *système de types et d'effet* [205] où l'effet modélise le passage du temps au cours de l'exécution.

La *durée* δ qui apparaît sur le constructeur de type fonctionnel $\tau \xrightarrow{\delta} \mathring{\tau}'$ spécifie un temps d'exécution pire cas pour toute fonction de ce type.

La durée **0** représente l'évaluation *instantanée* d'une expression quelle que soit l'entrée du programme. Symétriquement, la durée **1** représente l'évaluation d'une expression dont on ne peut garantir l'instantanéité. Les constructeurs $\delta + \delta$ et $\max(\delta, \delta)$ représentent respectivement la composition séquentielle et la composition parallèle des durées.

Avec cette abstraction binaire (**0**/**1**), la somme de deux durées non-instantanées est non-instantanée : $\mathbf{1} + \mathbf{1} \stackrel{\text{def}}{=} \mathbf{1}$. Par conséquent, les constructeurs $\delta + \delta$ et $\max(\delta, \delta)$ sont équivalents. Nous gardons toutefois deux constructeurs séparés dans le but d'établir une correspondance entre le système de types (c'est-à-dire, la sémantique statique) et la sémantique dynamique *synchrone* d'Eclat (chapitre 4).

2. À la manière des calculs de WCET (*Worst Case Execution Time*) pour les systèmes embarqués réactifs.

L'évaluation des constantes est instantanée ($\mathbf{0}$). L'appel d'une fonction de type $\tau \xrightarrow{\delta} \mathring{\tau}'$ prend dans le pire cas une durée δ. Chaque appel de fonction récursive terminale (qu'il soit direct ou récursif) induit un retard d'un cycle, et par conséquent, toute fonction récursive terminale `fix` f `(fun (`p`:`τ`) : `$\mathring{\tau}'$ `→` e`)` a un type de la forme $\tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'$. La durée de `(`$e_1 \| e_2$`)` est le maximum entre la durée de e_1 et la durée de e_2. La durée de `let` $p = e_1$ `in` e_2 est la somme des durées de e_1 et e_2. La durée (pire cas) de `if` e `then` e_1 `else` e_2 est la durée de l'expression e plus le maximum entre la durée de e_1 et la durée de e_2 (sans tenir compte de la valeur de e). Ce dernier cas souligne que l'analyse proposée est imprécise, mais sûre. Cela suffit en pratique pour vérifier qu'un programme ECLAT donné est instantané et modélise donc bien un système réactif.

L'unification des durées est réalisée en appliquant les règles d'équivalence définies sur la figure 3.2. Le langage de durées est suffisamment simple pour que l'inférence de durées soit décidable : la présentation du système de types (section 4.1) est dirigée par la syntaxe.

$$
\begin{aligned}
\mathbf{0} &\simeq \mathbf{0} \\
\mathbf{1} &\simeq \mathbf{1} \\
\mathbf{0} &\not\simeq \mathbf{1} \\
\mathbf{0} + \delta &\simeq \delta \\
\mathbf{1} + \delta &\simeq \mathbf{1} \\
\delta_1 + \delta_2 &\simeq \delta_2 + \delta_1 \\
\max(\mathbf{0}, \delta) &\simeq \delta \\
\max(\mathbf{1}, \delta) &\simeq \mathbf{1} \\
\max(\delta_1, \delta_2) &\simeq \max(\delta_2, \delta_1)
\end{aligned}
$$

FIGURE 3.2 Règles d'unification des durées

3.2.4 Constantes et opérateurs prédéfinis

Les constantes d'ECLAT sont l'unité `()`, les booléens (`true` et `false`), les entiers signés `(`l `: int<`θ`>)` où l désigne un entier littéral, les paires de constantes `(`c_1`,`c_2`)` et les vecteurs `{`c_1`,` $\cdots c_n$`}`. Elles sont manipulables par un ensemble d'opérateurs dont les schémas de type sont listés sur la Figure 3.3. Cela inclut des opérateurs logiques, comme l'opérateur `&` qui désigne le *et* logique. À noter qu'en ECLAT, l'application d'opérateurs est toujours stricte [3].

Les opérateurs arithmétiques d'ECLAT sont modulaires. Par exemple, multiplier deux entiers 32 bits produit un entier 32 bits ; les débordements arithmétiques ne produisent pas d'erreurs d'exécution. Les entiers de tailles différentes ne sont pas compatibles. Les divisions et modulos sont instantanés, mais peuvent déclencher une erreurs d'exécution en cas de division par zéro. Les conversions entre types entiers de tailles différentes se font explicitement au moyen

3. Contrairement aux opérateurs `&&` et `||` de C, OCAML et JAVA par exemple.

de l'opérateur `resize_int<`θ`>` : si la taille θ est plus grande que le type de l'entier argument, alors le résultat est une copie de l'argument avec extension de signe [4] ; si elle est plus petite, le résultat concatène le signe et les $\theta - 1$ bits de poids faibles de l'entier argument.

Les vecteurs sont fonctionnels (comme les tableaux SCADE [58] par exemple). Ce sont des valeurs immédiates (associées au constructeur de type de base $\mathring{\tau}$ `vect<`θ`>`) qui peuvent, par conséquent, être passées en argument de fonctions. L'opérateur `vect_create<`θ`>` crée un vecteur de taille θ dont tous les éléments sont initialisés avec la valeur reçue en argument. Etant donnés un entier i et un vecteur *vec*, l'opérateur `vect_nth(`*vec*`,`i`)` retourne l'élément à l'indice i dans le vecteur *vec*. L'opérateur `vect_copy_with(`*vec*`,`i`,`v`)` crée une copie du vecteur *vec* argument en remplaçant par v l'élément situé à l'indice i donné. L'opérateur `vect_size` retourne la taille (c'est-à-dire, le nombre d'éléments) du vecteur argument.

$$
\begin{array}{rcl}
\texttt{not} & : & \texttt{bool} \xrightarrow{0} \texttt{bool} \\
op & : & (\texttt{bool} * \texttt{bool}) \xrightarrow{0} \texttt{bool} \quad \text{si } op \in \{\texttt{\&}, \texttt{or}, \texttt{xor}\} \\
op & : & \forall\, \texttt{'}N \cdot (\texttt{int<}\ \texttt{'}N\ \texttt{>} * \texttt{int<}\ \texttt{'}N\ \texttt{>}) \xrightarrow{0} \texttt{int<}\ \texttt{'}N\ \texttt{>} \quad \text{si } op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{mod}\} \\
op & : & \forall\, \texttt{'}N \cdot (\texttt{int<}\ \texttt{'}N\ \texttt{>} * \texttt{int<}\ \texttt{'}N\ \texttt{>}) \xrightarrow{0} \texttt{bool} \quad \text{si } op \in \{\texttt{<}, \texttt{>}, \texttt{=}, \texttt{>=}, \texttt{<=}\} \\
\texttt{resize_int<}\theta\texttt{>} & : & \forall\, \texttt{'}N \cdot \texttt{int<}\ \texttt{'}N\ \texttt{>} \xrightarrow{0} \texttt{int<}\theta\texttt{>} \\
\texttt{fst} & : & \forall\, \texttt{'}B_1 \cdot \forall\, \texttt{'}B_2 \cdot (\ \texttt{'}B_1\ *\ \texttt{'}B_2\ ) \xrightarrow{0} \texttt{'}B_1 \\
\texttt{snd} & : & \forall\, \texttt{'}B_1 \cdot \forall\, \texttt{'}B_2 \cdot (\ \texttt{'}B_1\ *\ \texttt{'}B_2\ ) \xrightarrow{0} \texttt{'}B_2 \\
\texttt{vect_create<}\theta\texttt{>} & : & \forall\, \texttt{'}B \cdot \texttt{'}B \xrightarrow{0} \texttt{'}B\ \ \texttt{vect<}\theta\texttt{>} \\
\texttt{vect_nth} & : & \forall\, \texttt{'}N_1 \cdot \forall\, \texttt{'}N_2 \cdot \forall\, \texttt{'}B \cdot (\ \texttt{'}B\ \ \texttt{vect<}\ \texttt{'}N_1\ \texttt{>} * \texttt{int<}\ \texttt{'}N_2\ \texttt{>}) \xrightarrow{0} \texttt{'}B \\
\texttt{vect_copy_with} & : & \forall\, \texttt{'}N_1 \cdot \forall\, \texttt{'}N_2 \cdot \forall\, \texttt{'}B \cdot (\ \texttt{'}B\ \ \texttt{vect<}\ \texttt{'}N_1\ \texttt{>} * \texttt{int<}\ \texttt{'}N_2\ \texttt{>} *\ \texttt{'}B\ ) \xrightarrow{0} \textbf{unit} \\
\texttt{vect_size} & : & \forall\, \texttt{'}N \cdot \forall\, \texttt{'}B \cdot \texttt{'}B\ \ \texttt{vect<}\ \texttt{'}N\ \texttt{>} \xrightarrow{0} \texttt{int<16>}
\end{array}
$$

FIGURE 3.3 Typage des opérateurs

3.2.5 Syntaxe concrète

La syntaxe concrète d'ECLAT inclut plusieurs abréviations syntaxiques et des constructions dérivées. La plus importante est la construction `let rec` dérivée de `let/in` et `fix` pour la définition de fonctions récursives terminales.

4. Les bits de poids forts sont mis à *zéro* (si c'est un nombre positif) ou à *un* (si c'est un nombre négatif) suivant la représentation des entiers en complément à deux.

3.2.5.1 Déclaration globales

Comme indiqué précédemment, un programme ECLAT est une fonction qui transforme un flot d'entrées (l'argument) en un flot de sorties (le résultat). Dans la syntaxe abstraite d'ECLAT, cette fonction est de la forme `let` $f =$ `fun` $(p{:}\mathring{\tau})$: $\mathring{\tau}' \to e$. Cependant, dans la syntaxe concrète, un programme ECLAT est une séquence de déclarations formant un environnement de définition global *hyperstatique*[5] [167] comme dans la boucle d'interaction OCAML. Il s'agit simplement d'un sucre syntaxique :

$$\underbrace{\left(\begin{array}{l}\texttt{let } x_1{:}\tau_1 = v_1\texttt{;;}\\ \ldots\\ \texttt{let } x_n{:}\tau_n = v_n\texttt{;;}\\ \texttt{let } f{:}\mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}' = v\texttt{;;}\end{array}\right)}_{\text{syntaxe concrète}} \stackrel{\text{def}}{=} \underbrace{\left(\begin{array}{l}\texttt{let } f = \texttt{ fun } (y{:}\mathring{\tau}) : \mathring{\tau}' \to\\ \quad\texttt{let } x_1{:}\tau_1 = v_1 \texttt{ in}\\ \quad\ldots\\ \quad\texttt{let } x_n{:}\tau_n = v_n \texttt{ in}\\ \quad v\ y\end{array}\right)}_{\text{syntaxe abstraite}} \text{ où } y \text{ est un nom frais.}$$

Chaque déclaration globale est de la forme `let` $x_i{:}\tau_i = v_i$`;;`. La variable globale x_i introduite est alors visible uniquement dans les déclarations suivantes, mais elle peut être redéfinie tout en maintenant la portée lexicale. La dernière définition globale d'un programme doit être une fonction et constitue le *point d'entrée* : c'est cette fonction qui interagit avec l'environnement physique.

3.2.5.2 Abréviations

Les expressions et les types peuvent être sur-parenthésés, comme par exemple `(`f`(`g`(`x`)))`. Dans un motif p, les variables peuvent être notées "`_`" lorsqu'elles ne sont pas utilisées ; ce symbole "`_`" peut aussi apparaître dans les annotations de types pour désigner un type quelconque qui sera inféré par le système de types. Par exemple, `(fun ((`x`,_):(bool * _)) : bool` $\to x$`)` est une fonction qui prend une paire en argument et retourne sa composante de gauche, laquelle doit être un booléen. À cela s'ajoute un ensemble d'abréviations définies sur la figure 3.4.

Les types des fonctions instantanées (*resp.* non instantanées) sont notés $\tau \Rightarrow \mathring{\tau}'$ (*resp.* $\tau \to \mathring{\tau}'$). Les définitions de fonctions (qu'elles soient locales ou globales) sont abrégées en `let` f `(`$p{:}\tau$`)`$:\mathring{\tau}' = e$ avec ou sans annotations de types. L'application des opérateurs binaires usuels est notée de façon infixe (avec précédences standards des opérateurs).

Les annotations de types omises sont remplacées par des inconnues de types. Si la branche *else* d'une conditionnelle est omise, elle est remplacée par la constante `()`. Si la clause `reset` de la construction `exec` est omise, elle est remplacée par la constante `false` (auquel cas le calcul n'est jamais réinitialisé).

5. Dans un environnement global hyperstatique, la portée est lexicale et on ne peut pas faire référence à une variable tant qu'elle n'a pas été définie.

$$
\begin{array}{rcl}
\tau \Rightarrow \mathring{\tau}' & \stackrel{\text{def}}{=} & \tau \xrightarrow{\mathbf{0}} \mathring{\tau}' \\
\tau \to \mathring{\tau}' & \stackrel{\text{def}}{=} & \tau \xrightarrow{\mathbf{1}} \mathring{\tau}' \\
e_1\ op\ e_2 & \stackrel{\text{def}}{=} & op(e_1, e_2) \\
(e{:}\tau) & \stackrel{\text{def}}{=} & \texttt{let}\ x{:}\tau = e\ \texttt{in}\ x \quad \text{où } x \text{ est nom frais.} \\
\texttt{let}\ p = e\ \texttt{in}\ e' & \stackrel{\text{def}}{=} & \texttt{let}\ p\ \texttt{:_} = e\ \texttt{in}\ e' \\
\texttt{fun}\ (p{:}\tau) \to e & \stackrel{\text{def}}{=} & \texttt{fun}\ (p{:}\tau)\ \texttt{:}\ _ \to e \\
\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau) \to e) & \stackrel{\text{def}}{=} & \texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ \texttt{:}\ _ \to e) \\
\texttt{fun}\ p \to e & \stackrel{\text{def}}{=} & \texttt{fun}\ (p\ \texttt{:_})\ \texttt{:_} \to e \\
\texttt{fix}\ f\ (\texttt{fun}\ p \to e) & \stackrel{\text{def}}{=} & \texttt{fix}\ f\ (\texttt{fun}\ (p\ \texttt{:_})\ \texttt{:_} \to e) \\
l & \stackrel{\text{def}}{=} & (l{:}\texttt{int<_>}) \quad \text{pour } l \text{ un littéral entier positif} \\
\texttt{let}\ f\ p = e\ \texttt{in}\ e' & \stackrel{\text{def}}{=} & \texttt{let}\ f = \texttt{fun}\ p \to e\ \texttt{in}\ e' \\
\texttt{let}\ f\ p = e\texttt{;;} & \stackrel{\text{def}}{=} & \texttt{let}\ f = \texttt{fun}\ p \to e\texttt{;;} \\
\texttt{if}\ e_1\ \texttt{then}\ e_2 & \stackrel{\text{def}}{=} & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ () \\
\texttt{reg}\ (\texttt{fun}\ p \to e)\ \texttt{init}\ e_0 & \stackrel{\text{def}}{=} & \texttt{reg}\ (\texttt{fun}\ (p\ \texttt{:}\ _)\ \texttt{:}\ _ \to e)\ \texttt{init}\ e_0 \\
\texttt{exec}\ e\ \texttt{default}\ e_0 & \stackrel{\text{def}}{=} & \texttt{exec}\ e\ \texttt{default}\ e_0\ \texttt{reset}\ \texttt{false}
\end{array}
$$

FIGURE 3.4 Abréviations dans la syntaxe concrète d'ECLAT

3.2.5.3 Constructions dérivées

La figure 3.5 définit trois constructions dérivées disponibles en ECLAT. La construction `let rec` est dérivée de `let/in` et de `fix` de façon standard. La séquence $e_1\texttt{;}e_2$ est dérivée de `let/in`. ECLAT offre des n-uplets qui constituent une généralisation des paires ; ils peuvent être utilisés aussi bien dans des motifs que dans des expressions.

$$
\begin{array}{rcl}
\texttt{let rec}\ f\ p = e\ \texttt{in}\ e' & \stackrel{\text{def}}{=} & \texttt{let}\ f = \texttt{fix}\ f\ (\texttt{fun}\ p \to e)\ \texttt{in}\ e' \\
e\texttt{;}e' & \stackrel{\text{def}}{=} & \texttt{let}\ () = e\ \texttt{in}\ e' \\
(p_1, p_2, \cdots\ p_n) & \stackrel{\text{def}}{=} & ((p_1, p_2), \cdots\ p_n) \\
(e_1, e_2, \cdots\ e_n) & \stackrel{\text{def}}{=} & ((e_1, e_2), \cdots\ e_n) \\
(e_1 \| e_2 \| \cdots\ e_n) & \stackrel{\text{def}}{=} & ((e_1 \| e_2) \| \cdots\ e_n) \\
\texttt{let}\ p_1 = e_1\ \texttt{and}\ \cdots\ p_n = e_n\ \texttt{in}\ e' & \stackrel{\text{def}}{=} & \texttt{let}\ (p_1, \cdots\ p_n) = (e_1 \| \cdots\ e_n)\ \texttt{in}\ e' \\
\texttt{pause}\ e & \stackrel{\text{def}}{=} & (\texttt{fix}\ f\ (\texttt{fun}\ () \to e))() \text{ où } f \text{ est un nom frais.} \\
\texttt{halt}\ e & \stackrel{\text{def}}{=} & (\texttt{fix}\ f\ (\texttt{fun}\ () \to f\ e))() \\
 & & \text{où } f \text{ est un nom frais.}
\end{array}
$$

FIGURE 3.5 Constructions dérivées disponibles en ECLAT

La construction `let/and` est dérivée de `let/in` et de $(e_1 \| e_2)$. Elle s'évalue comme suit : $\texttt{let}\ p_1 = e_1\ \texttt{and}\ p_2 = e_2\ \texttt{in}\ e$ évalue parallèlement e_1 en v_1 et e_2 en v_2 en synchronisant la

terminaison des deux expressions, puis évalue l'expression e où p_1 est lié à v_1 et p_2 à v_2. Cette construction existe aussi en version n-aire. La construction (**pause** e) effectue intuitivement une *pause* d'un cycle avant d'exécuter e. La construction **halt** e est une boucle infinie déroulée dans le temps ; elle sert à modéliser les erreurs d'exécution.

3.2.6 Extension de l'environnement d'exécution Eclat

Lors du développement de circuits réalistes, il est souvent nécessaire d'utiliser des composants matériels existants. C'est pourquoi ECLAT fournit une directive **external** $f : \mathring{\tau} \xrightarrow{'d} \mathring{\tau}'$;; pour déclarer des fonctions externes en vue de les utiliser, par simple application de fonction $f(e)$, dans les programmes ECLAT. Les types des arguments des fonctions externes doivent être des types de base (car les fonctions ne sont pas des valeurs manipulables dans le circuit engendré). Les fonctions peuvent avoir un état interne et leur temps d'exécution δ peut être non instantané.

La directive **external** f peut être précédée par le mot clé **shared**, ce qui force la fonction f a n'être instanciée qu'une seule fois [6] ; l'instance de f est alors implicitement protégée par un verrou suivant la même technique que celle employée au chapitre 6 pour séquentialiser les accès à une mémoire partagée. Si la fonction externe n'est pas déclarée **shared**, chaque appel à cette fonction dans le programme source créé une nouvelle instance du circuit correspondant.

Cette section a présenté les principaux éléments syntaxiques d'ECLAT et leur sémantique intuitive. Nous parcourons dans le reste de ce chapitre une série d'exemples de programmation en ECLAT pour illustrer l'expressivité du langage.

3.3 Principes de programmation en Eclat

Cette section explique le mélange de calcul et d'interaction exprimable en ECLAT à travers une série d'exemples simples.

3.3.1 Récursion terminale pour le calcul

Commençons par définir en ECLAT la *fonction de Fibonacci* telle que :

$$\begin{aligned} \mathit{Fibonacci}(0) &\stackrel{\text{def}}{=} 0 \\ \mathit{Fibonacci}(1) &\stackrel{\text{def}}{=} 1 \\ \mathit{Fibonacci}(n) &\stackrel{\text{def}}{=} \mathit{Fibonacci}(n-1) + \mathit{Fibonacci}(n-2) \end{aligned}$$

La récursion générale n'est pas directement exploitable en ECLAT : une analyse statique rejète les définitions récursives qui ne sont pas récursives terminales. C'est pourquoi la définition de *Fibonacci* doit être réécrite de façon récursive terminale [7]. Il en existe une définition bien

6. Dans ce cas, il n'y a pas de parallélisme.

7. Cela nécessite en général une pile explicite tenant compte du nombre d'étapes de calcul, comme dans l'implantation du problème des *N reines* en ECLAT qui nous présenterons à la section 7.5.

connue telle que :

$$\begin{array}{lcl} \texttt{fibonacci}(n) & \stackrel{\text{def}}{=} & \texttt{fib}(n, 0, 1) \\ \texttt{fib}(0, a, b) & \stackrel{\text{def}}{=} & a \\ \texttt{fib}(i, a, b) & \stackrel{\text{def}}{=} & \texttt{fib}(i-1, b, a+b) \quad \text{si } i \neq 0 \end{array}$$

Cette fonction `fibonacci` est traduite en ECLAT sur la figure 3.6. À noter que c'est aussi une fonction OCAML valide ; sa sémantique intuitive est d'ailleurs la même qu'en OCAML.

```
1 let fibonacci n =
2   let rec fib (i, a, b) =
3     if i = 0 then a
4     else fib(i - 1, b, a + b)
5   in fib(n, 0, 1) ;;
```

FIGURE 3.6 Un calcul en ECLAT

La fonction `fibonacci`(n) est un calcul non instantané : son exécution prend exactement $n+1$ cycles d'horloge. En effet, chaque appel à la fonction récursive terminale `fib` (y compris l'*appel direct*) induit un retard d'un cycle.

Supposons que la fonction `fibonacci` soit utilisée[8] comme point d'entrée du programme ECLAT. Celle-ci interagit alors directement avec un flot d'entrées en produisant un flot de sorties. Ces deux flots sont synchronisés par l'horloge physique. L'exécution de ce programme est représentée par un chronogramme sur la figure 3.7. Les traits verticaux désignent les *ticks* d'horloge (numérotés $t_0, t_1, \cdots t_n$ en bas du chronogramme). Les valeurs prises par les variables du programme au cours du temps sont représentées par des flots.

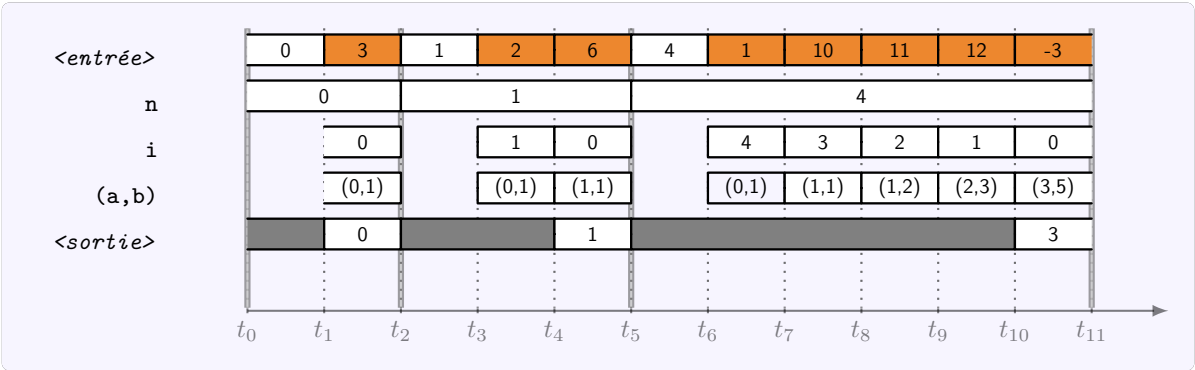


FIGURE 3.7 Exécution d'un programme ECLAT

L'entrée du programme (`<entrée>`) est un flot de valeurs émises à chaque cycle (juste *après* le *tick* d'horloge). La fonction `fibonacci` (point d'entrée du programme) est appelée implicitement avec des valeurs de ce flot.

8. Avec l'option `-relax` du compilateur (voir annexe C) qui autorise le programme à être non instantané.

La sémantique d'ECLAT est en appel par valeur. C'est pourquoi le paramètre formel `n` de `fibonacci` reçoit la valeur courante de l'entrée au moment de l'appel de fonction et il reste figé pendant l'appel. Dès que le programme se termine, il retourne une valeur en sortie (*<sortie>*) et attend un cycle ; puis il redémarre avec une nouvelle entrée.

Tout appel à `fib` conduit à un retard d'un cycle. L'exécution des autres constructions de programmation d'ECLAT n'induit aucun retard supplémentaire. Les paramètres `i`, `a` et `b` de la fonction `fib` changent à chaque appel. Aux cycles commençant par t_0, t_2 et t_5, les paramètres `i`, `a` et `b` n'ont pas encore reçu de valeur. Cela vient justement de ce retard d'un cycle (dû à l'appel récursif) avant l'entrée dans le corps de la fonction. On représente cela par un espace blanc qui met en évidence la *durée de vie* de ces liaisons.

À chaque fois que le programme se termine, il redémarre avec une nouvelle entrée. Les traits gris verticaux à t_0, t_2, t_5 et t_{11} désignent les *ticks* où le programme démarre (ces traits seront omis dans la suite du document).

Nous arrivons maintenant à un point crucial : ce programme n'est pas *instantané* et cela pose deux problèmes. D'une part, des entrées (représentées en orange) sont ignorées. D'autre part, la valeur retournée en sortie est retardée, d'où la présence de valeurs indéfinies (représentées en gris). C'est pourquoi le système de types d'ECLAT rejette ce programme que nous considérons comme non sûr. Pour appeler `fibonacci` à l'intérieur d'un programme ECLAT, il faut utiliser la construction **`exec`**. Cela n'impacte en rien l'expressivité du langage.

3.3.2 Exécution d'un calcul pas à pas

La figure 3.8 montre un programme *instantané* utilisant la construction **`exec`** d'ECLAT. Il exécute pas à pas un appel à `fibonacci`, qui est ici une fonction auxiliaire et non plus un programme. Le calcul progresse à l'intérieur du **`exec`** et retourne *instantanément* une paire `(o,rdy)` qui contient :

— soit le résultat de l'appel à `fibonacci` (associé au booléen `true`),

— soit une valeur par défaut (associée au booléen `false`).

Si l'entier `n` argument est négatif, le flot de contrôle ne passe pas par le bloc **`exec`** et celui-ci n'est donc pas exécuté. Cela souligne le fait qu'ECLAT est basé sur un temps logique (analogue à la notion d'*horloge* des langages synchrones flot de données).

```
1 let main n =
2   if n < 0 then 0 else
3   let (o,rdy) = exec fibonacci(n) default 42
4   in o ;;
```

FIGURE 3.8 Un programme instantané exécutant un calcul long

La clause **`reset`** du bloc **`exec`** est ici omise (suivant l'abréviation introduite sur la figure 3.4). Le bloc **`exec`**, dans ce cas, n'est jamais réinitialisé pendant l'exécution.

L'exécution de ce programme est tracée sur le chronogramme à la figure 3.9. À chaque passage dans le **exec** (suivant le flot de contrôle du programme), ce dernier retourne instantanément une valeur (qui est soit `42`, soit le résultat de l'appel à `fibonacci`). Par conséquent, ici, la fonction `main` est instantanée. Le calcul progresse sous le **exec** à chaque fois qu'il est exécuté. Par exemple, si la fonction `main` est appelée tous les trois cycles, le calcul progresse trois fois plus lentement. La valeur retournée par **exec** est une paire `(o,rdy)` où `rdy` est un booléen qui vaut `true` lorsque le résultat `o` est disponible. Quand `rdy` vaut `false`, `o` contient une valeur par défaut obtenue en évaluant la clause **default** (qui doit être une expression instantanée). Lorsque le calcul se termine, il redémarre à la prochaine exécution du **exec**. Le calcul sous le **exec** s'exécute sur une entrée et ignore les nouvelles entrées jusqu'à ce qu'il se termine (comme c'était le cas sur la figure 3.7 avec des entrées ignorées représentées en orange).

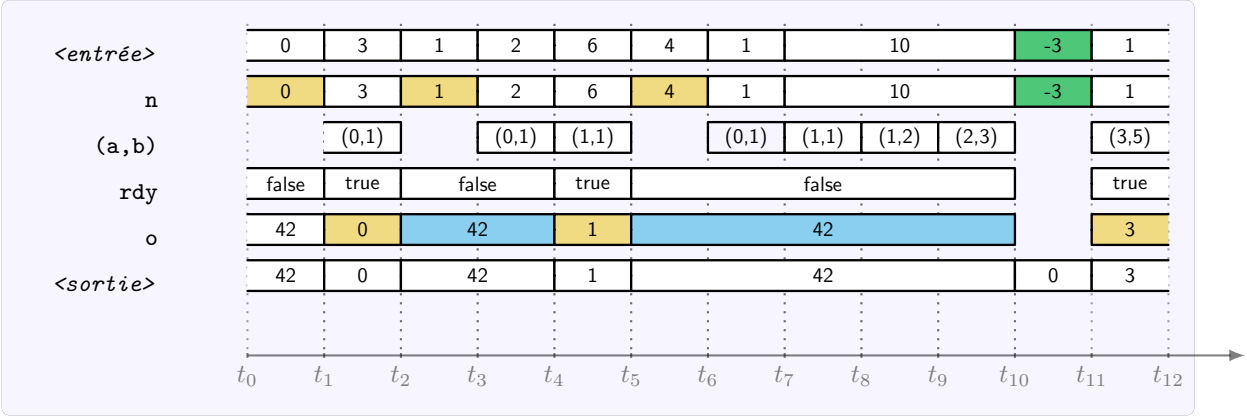


FIGURE 3.9 Exécution pas à pas suivant un temps logique

L'appel `fibonacci(`n`)` prend $n+1$ cycles si $n \geq 0$ (sans compter la pause d'un cycle entre la fin du calcul et son redémarrage). En particulier, `fibonacci(0)` prend un cycle (ce qui correspond au coût de l'appel direct à la fonction locale `fib` récursive terminale). La fonction `main` prend *zéro* cycle car `fibonacci` est sous un **exec** qui retourne toujours instantanément une valeur.

Au *tick* t_{10}, l'entrée `i` est négative ; c'est donc la branche *then* de la conditionnelle (ligne 2) qui est exécutée. La branche *else*, qui contient le **exec**, n'est pas exécutée, ce qui a pour effet de suspendre la progression du calcul `fibonacci(n)`. C'est pourquoi le résultat `fibonacci(4) = 3`, qui devait arriver à t_{10}, arrive finalement à t_{11} lorsque l'entrée `i` est à nouveau positive.

Dans la suite, on ne considérera que des programmes instantanés (comme ici `main`). Les flots *<entrée>* et *<sortie>* seront omis et implicitement connectés à l'entrée et à la sortie du programme (ici `n` et `o`).

3.3.3 Circuits séquentiels pour l'interaction

Les circuits séquentiels synchrones implantés au niveau RT comportent des registres pour sauvegarder des valeurs au cours du temps, ce qui permet d'implanter un compteur par exemple.

La construction `reg (fun` $p \rightarrow e$`) init` e_0 d'ECLAT représente un registre initialisé à la valeur de e_0 et mis à jour avec la fonction `fun` $p \rightarrow e$. La nouvelle valeur est instantanément connectée à la sortie et elle est recopiée dans le registre à la fin du cycle d'horloge.

Ainsi, la fonction `counter` définie sur la figure 3.10 s'exécute instantanément à chaque fois qu'elle est appelée, comme le montre le chronogramme associé. La sortie `o` vaut la valeur précédente `c` (initialement `0`) incrémentée ou non suivant la valeur de l'entrée `i`. La construction `reg` est une expression qui retourne une valeur. Par conséquent, la fonction `counter` de la figure 3.10 peut aussi s'écrire comme suit :

```
let counter i =
  reg (fun c -> if i then c + 1 else c) init 0 ;;
```

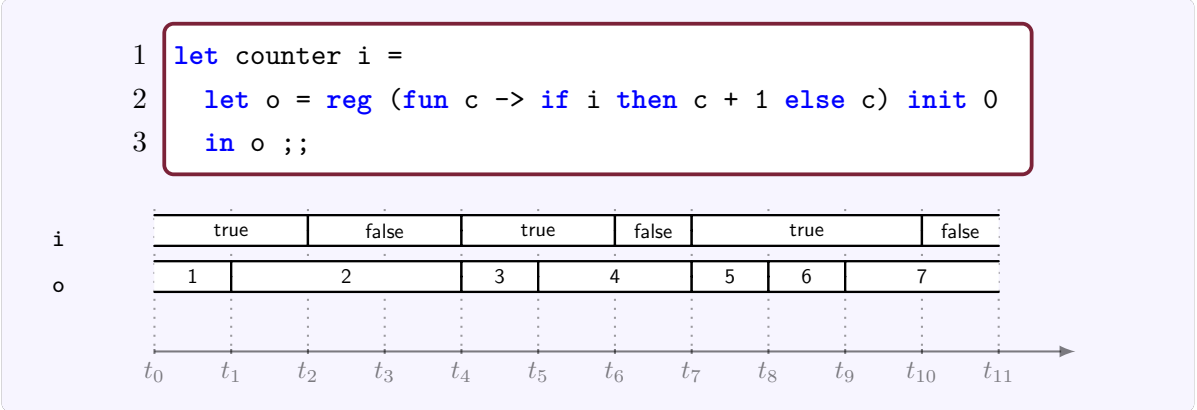


FIGURE 3.10 Un compteur en ECLAT

3.3.4 Mélange de calcul et d'interaction

Nous avons montré que la construction `exec` d'ECLAT permet d'exécuter des calculs pas à pas (comme la fonction non instantanée `fibonacci` appelée sur la figure 3.9). La sortie `rdy` (fournie par `exec`) permet de sonder le résultat du calcul pour savoir à quel cycle celui-ci est disponible. Les calculs se comportent alors comme des fonctions instantanées (au même titre que la fonction `counter` sur la figure 3.10) et peuvent donc être exécutés dans le système réactif.

La figure 3.11 définit un programme ECLAT qui compose d'une part la fonction `fibonacci` (non instantanée) et d'autre part la fonction `counter` (instantanée). La sortie `c` contient le nombre de cycles où la variable `rdy` a pris la valeur `true` depuis l'instant initial (t_0), c'est-à-dire le nombre de calculs de Fibonacci achevés. La valeur par défaut du bloc `exec` dépend de l'entrée `n`. Les points importants ici sont la composition *instantanée* du calcul et du compteur et la disponibilité de l'entrée `n` à chaque cycle.

```
1 let main n =
2   let (o,rdy) = exec fibonacci(n) default (42 + n) in
3   let c = counter(rdy) in
4   (o,c) ;;
```

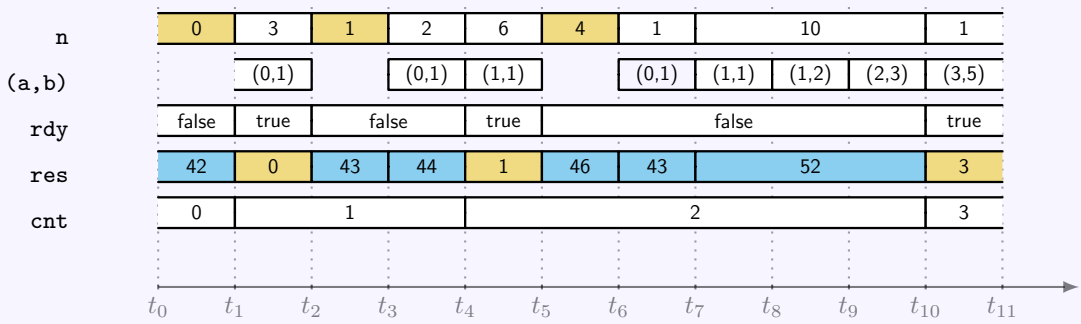


FIGURE 3.11 Un programme ECLAT mixant calcul et interaction

3.3.5 Composition de calculs

Le but d'ECLAT n'est pas d'offrir une syntaxe alternative, sous forme de fonctions récursives terminales, pour décrire des calculs en style flot de données. Les calculs réalisent directement des algorithmes en style flot de contrôle. Dans ce contexte, un point essentiel est la composition des calculs.

L'exemple donné sur la figure 3.12 illustre les deux formes de composition, séquentielle et parallèle, disponibles en ECLAT. Ce programme calcule la valeur de :

$$Fibonacci(3) + Fibonacci(Fibonacci(4+1))$$

Les synchronisations sont implicites, à la fois pour la séquence `let` $x = e_1$ `in` e_2 et pour la paire parallèle `(`$e_1 \| e_2$`)`. La durée de la séquence ("e_1, puis e_2") est l'addition des durées de e_1 et e_2. La durée de la paire parallèle ("e_1 et e_2 avec synchronisation des deux résultats") est le maximum des durées de e_1 et e_2.

Ainsi, sur le chronogramme, la valeur de `n` est disponible dès t_0. Puis, toujours à t_0, la valeur de `n` est utilisée pour calculer `fibonacci(n)`. Comme `n` = `5`, le calcul prend 6 cycles. Dès t_6, le résultat est lié au nom `x` et le calcul parallèle `(fibonacci(3)` ‖ `fibonacci(x))` commence. Le calcul `fibonacci(3)` prend 4 cycles et le calcul `fibonacci(x)` prend 6 cycles car `x` = `5`. Lorsque les deux calculs sont terminés, leurs résultats sont additionnés, ce qui donne la valeur `7` retournée en sortie ; alors `rdy` passe à `true` pendant un cycle. Après cette pause d'un cycle, le calcul sous le `exec` redémarre au *tick* t_{13}.

```
1 let main () =
2   let (o,rdy) = exec (let n = 4 + 1 in
3                       let x = fibonacci(n) in
4                       let (y, z) = ( fibonacci(3) || fibonacci(x) )
5                       in y + z)
6               default 0
7   in o ;;
```

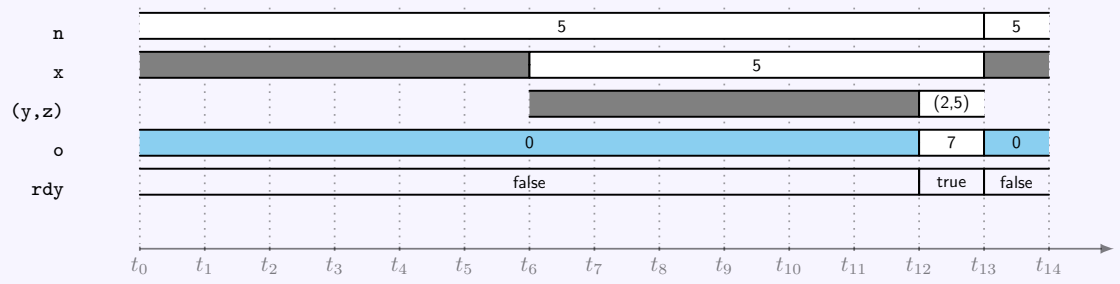


FIGURE 3.12 Compositions séquentielles et parallèles en ECLAT

3.4 Expressivité du langage

Puisqu'ECLAT vise à offrir un équilibre entre calculs et interactions, en utilisant les même constructions de programmation pour décrire les uns et les autres, on peut se demander si les calculs expressibles sont suffisamment généralistes et si l'expressivité des calculs a un impact sur la sûreté des interactions. Le but de cette section est de montrer, par une série d'exemples, qu'ECLAT est à la fois sûr et expressif.

À la section 3.4.1, nous montrons comment programmer simplement en ECLAT, sous forme de fonctions récursives terminales, des calculs irréguliers dont le temps d'exécution n'est pas prédictible. L'expressivité qui en résulte pose cependant question : comment garantir la sûreté du système réactif étant donné que la terminaison des calculs n'est pas garantie (ce qui équivaut à un *dead-lock*). C'est pourquoi, à la sous-section 3.4.2, nous montrons que la clause **reset** de la construction **exec** permet de réinitialiser un calcul et le relancer instantanément avec de nouvelles entrées, ce qui permet toujours d'éviter un potentiel *dead-lock*.

La section 3.4.3 montre que d'une part, les calculs peuvent utiliser la construction de paire parallèle $(e_1 \| e_2)$, et que d'autre part, le programme réactif peut exécuter pas à pas plusieurs calculs en les plaçant dans des blocs **exec** composés séquentiellement. De cette manière, les calculs progressent en parallèle tout en étant davantage corrélés aux entrées du programme, ce qui favorise la gestion concurrente des entrées/sorties.

Enfin, la section 3.4.4 montre qu'ECLAT dispose d'une structure de données élémentaire (appelée *vecteur*) ainsi que des traits de programmation fonctionnelle comme l'ordre supérieur, le polymorphisme paramétrique et l'imbrication de récursions terminales. Cela offre un niveau

d'abstraction bien plus haut que ce que l'on trouve dans les principaux langages de description de matériel habituellement utilisés pour programmer des circuits FPGA.

3.4.1 Calcul irrégulier

L'exemple `fibonacci` de la figure 3.6 a l'avantage de montrer clairement le lien entre les *ticks* d'horloge et les appels de fonctions récursives terminales en ECLAT : `fibonacci(`n`)` prend $n+1$ cycles pour tout entier n positif. Cependant, la durée des calculs expressibles en ECLAT n'est pas toujours prédictible.

La *suite de Syracuse* est un exemple classique de calcul irrégulier difficilement prédictible [108]. Elle est définie pour tout $n \geq 1$ par :

$$\begin{array}{rcll} F(1) & \stackrel{\text{def}}{=} & 1 & \\ F(n) & \stackrel{\text{def}}{=} & F(n/2) & \text{si } n \text{ est pair} \\ F(n) & \stackrel{\text{def}}{=} & F(3 \times n + 1) & \text{si } n \text{ est impair} \end{array}$$

La fonction F retourne `1` si elle termine. D'après la conjecture de *Collatz*, la fonction $F(n)$ termine toujours et atteint `1` quel que soit $n \geq 1$. Le *temps de vol* de la suite de *Collatz* de racine n, qui désigne le nombre d'étapes de récursion dans F, est défini par :

$$\begin{array}{rcll} \texttt{collatz}(n) & \stackrel{\text{def}}{=} & \texttt{loop}(n) & \\ \\ \texttt{loop}(1) & \stackrel{\text{def}}{=} & 1 & \\ \texttt{loop}(n) & \stackrel{\text{def}}{=} & 1 + \texttt{loop}(n/2) & \text{si } n \text{ est pair} \\ \texttt{loop}(n) & \stackrel{\text{def}}{=} & 1 + \texttt{loop}(3 \times n + 1) & \text{si } n \text{ est impair} \end{array}$$

Cela peut se réécrire sous une forme récursive terminale comme suit :

$$\begin{array}{rcll} \texttt{collatz}(n) & \stackrel{\text{def}}{=} & \texttt{loop}(n, 1) & \\ \\ \texttt{loop}(1, t) & \stackrel{\text{def}}{=} & t & \\ \texttt{loop}(i, t) & \stackrel{\text{def}}{=} & \texttt{loop}(i/2, t + 1) & \text{si } i \text{ est pair} \\ \texttt{loop}(i, t) & \stackrel{\text{def}}{=} & \texttt{loop}(3 \times i + 1, t + 1) & \text{si } i \text{ est impair} \end{array}$$

Cette formulation est programmée en ECLAT sur la figure 3.13. De façon intéressante, l'implantation de *Collatz* en ECLAT retourne un entier `t` (le temps de vol) en `t` cycles. Par exemple, le chronogramme figure 3.13 montre que `collatz(1)` retourne `1` en 1 cycle (soit un appel direct à la fonction `loop`) ; `collatz(8)` retourne `4` en 4 cycles (soit un appel direct et trois appels récursifs). Cela illustre le lien fondamental entre le comportement fonctionnel et le comportement temporel des programmes en ECLAT.

L'argument `n` est un entier signé. Or, `collatz(n)` n'est pas défini pour `n` < 1. C'est pourquoi sur le chronogramme, le calcul `collatz(-2)` qui commence au *tick* t_7 ne termine jamais, car dans ce cas :

— `loop(-2)` appelle `loop((-2)/2)`, c'est-à-dire `loop(-1)` ;

— `loop(-1)` appelle `loop((3*(-1))+1)`, c'est-à-dire `loop(-2)`.

Le système est alors bloqué : bien que le calcul continue de se dérouler à chaque cycle, il ignore les nouvelles entrées et retourne indéfiniment la valeur `0` par défaut.

```
1  let collatz n =
2    let rec loop(i,t) =
3      if i = 1 then t
4      else if i mod 2 = 0 then loop(i / 2, t + 1)
5           else loop(3 * i + 1, t + 1)
6    in
7    loop(n,1) ;;
8
9  let main n =
10   let (o,rdy) = exec collatz n default 0
11   in o ;;
```

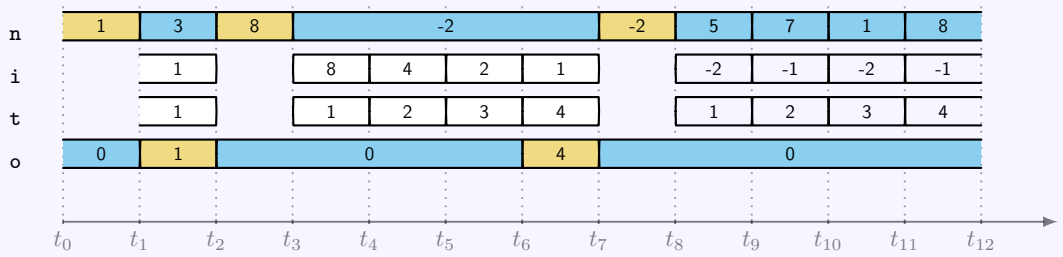


FIGURE 3.13 Un calcul long irrégulier en ECLAT

3.4.2 Réinitialisation

La construction **exec** peut être réinitialisée via sa clause **reset**. Cela permet d'éviter qu'un calcul, trop long ou ne terminant pas, ne bloque le système (comme c'est le cas sur la figure 3.13). Cela constitue aussi un trait de programmation pour interagir, de façon expressive, avec l'environnement : à tout moment il est possible d'interrompre le calcul pour prendre en compte de nouvelles entrées.

La figure 3.14 montre un exemple de calcul irrégulier réinitialisable. À chaque fois que l'entrée `r` du programme vaut `true` (ici à t_8 et t_{10}), le calcul redémarre avec une nouvelle entrée.

3.4.3 Désynchroniser les calculs parallèles

En ECLAT, le programme réactif peut exécuter de multiples calculs en parallèle, via la construction **exec**. L'effet est similaire à la construction $(e_1 \| e_2)$ avec cependant une différence notable : $(e_1 \| e_2)$ synchronise *implicitement* les résultats des expressions e_1 et e_2 exécutées en parallèle (l'expression qui termine en premier attend que la seconde se termine) ; ce qui n'est pas le cas lorsque l'on compose directement des blocs **exec**. Utiliser plusieurs blocs **exec** est utile pour la gestion *concurrente* et *non instantanée* des entrées/sorties.

La figure 3.15 montre la différence de comportement entre ces deux manières de composer des calculs en parallèle en ECLAT. Les noms `x` et `y` sont liés à deux calculs, chacun étant exécuté

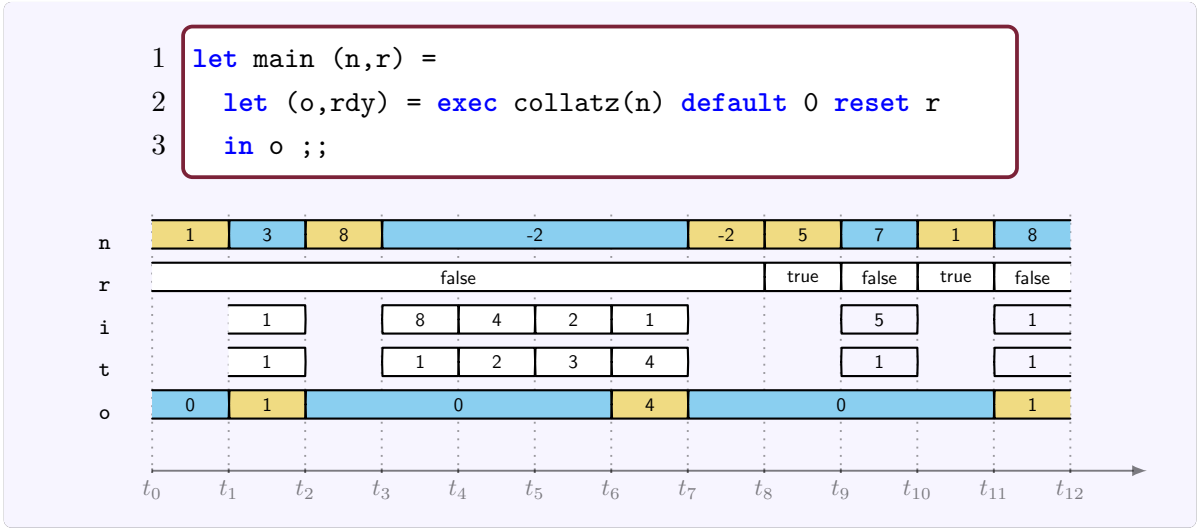


FIGURE 3.14 Réinitialisation d'un calcul sous un `exec`

pas à pas avec `exec`. À chaque fois qu'un calcul se termine, il redémarre avec une nouvelle entrée. Le nom `o` est lié à un calcul de la forme $(e_1 \| e_2)$ qui ne termine que lorsque les expressions e_1 et e_2 se sont terminées. Par exemple, ce calcul redémarre à t_5 avec les entrées courantes `(2,8)`. Le résultat du calcul (la paire `(2,4)`) est retourné à t_9 ; puis ce calcul redémarre à t_{10} avec la nouvelle entrée `(1,1)`, produisant le résultat `(1,1)` à t_{11}.

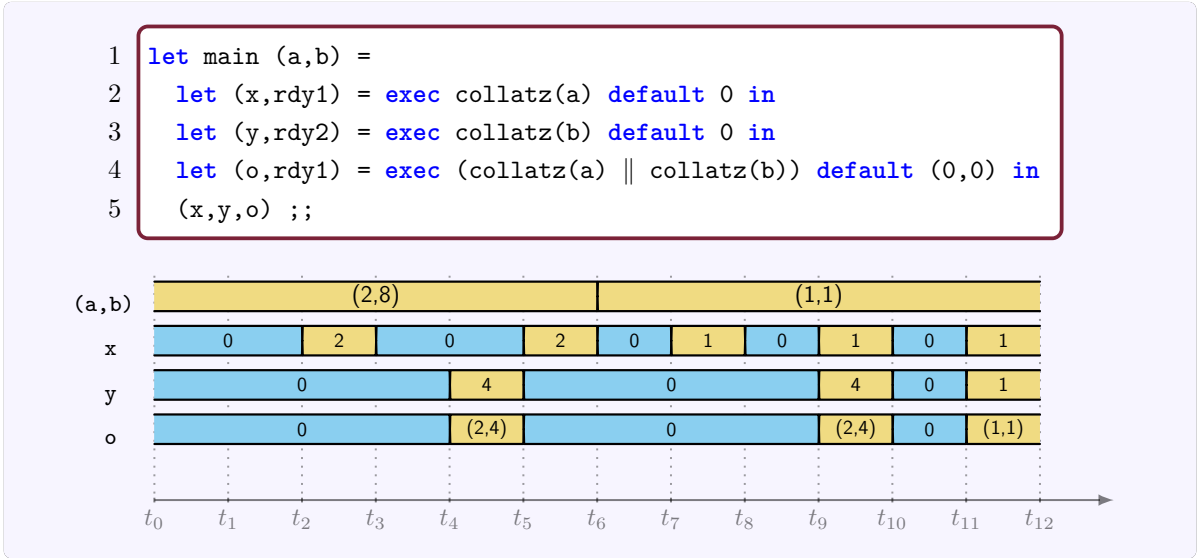


FIGURE 3.15 Composition séquentielle de blocs `exec`

3.4.4 Traits de programmation applicatifs

Le style applicatif d'ECLAT est un choix de conception important qui vise à offrir un haut niveau d'abstraction tout en produisant des circuits qui se veulent efficaces et avec un compor-

tement temporel précis.

Dans ce contexte, ECLAT s'apparente aux langages de description de matériel fonctionnels (tels μFP ou CλASH) par certains de ses traits comme les vecteurs fonctionnels, le polymorphisme paramétrique (de type et de taille) et une forme limitée d'ordre supérieur qui permet de passer des fonctions en paramètres d'autre fonctions. De plus, la récursion terminale, qui introduit un retard d'un cycle à chaque appel et peut être imbriquée, offre un réel gain d'expressivité puisque cela permet de programmer des algorithmes séquentiels en style direct avec des synchronisations implicites.

Le programme sur la figure 3.16 est un exemple simple qui mélange ces différents traits. Il définit un combinateur `map` qui, étant donnés une fonction `f` et un vecteur `a` de taille $'N$, applique f point à point sur chaque élément du vecteur `a` et construit un vecteur résultat. Cet exemple illustre l'utilisation des opérateurs prédéfinis `vect_size`, `vect_nth` et `vect_copy_with`. Les types des vecteurs sont paramétrés par leur taille. Une contrainte de type force le vecteur résultat à avoir la même taille que le vecteur argument. La fonction `map` est polymorphe paramétrique : son schéma de type abstrait non seulement les types des éléments des vecteurs arguments et résultats, mais aussi la taille de ces deux vecteurs. Comme en OCAML, les variables de types *quantifiées* le sont implicitement en tête des types. La fonction `map` définit localement une fonction récursive `loop` dans laquelle figurent deux variables libres : la fonction `f` et le vecteur `a` qui sont les deux arguments de `map`.

```
1  let map ((f,a) : (('B1 => 'B2) * 'B1 vect<'N>)) : 'B2 vect<'N> =
2    let rec loop (i,acc) =
3      if i < vect_size a then
4        let x = vect_nth (a,i) in
5        let y = f x in
6        loop(i+1,vect_copy_with(acc,i,y))
7      else acc
8    in loop(0,a) ;;
9
10 let main (a:int<8> vect<12>) : int<8> vect<12> =
11   let zero : int<8> vect<12> = vect_create<12>(1) in
12   let inc x = x + 1 in
13   let (o,rdy) = exec map(inc,a) default zero
14   in o ;;
```

FIGURE 3.16 Définition d'un type vecteur et parcours de vecteur en ECLAT

Le temps d'exécution exact de `map` est `N` + 1 cycles (puisque la fonction f est instantanée de par l'annotation [9] de type de la forme $\tau \Rightarrow \mathring{\tau}'$) et d'autre part, chaque appel à la fonction récursive locale `loop` (qu'il soit direct ou récursif) induit un retard d'un cycle. Dans cet exemple,

9. Cette annotation est prise en compte lors du typage du programme : si la fonction argument n'est pas instantanée, alors le programme est rejeté.

la fonction `map` applique la fonction `inc` $\stackrel{\text{def}}{=}$ `fun x → x + 1` pour créer un nouveau vecteur en incrémentant l'ensemble des éléments du vecteur argument.

3.5 Programmation synchrone

Cette section vise à montrer qu'ECLAT est suffisamment expressif pour exprimer des programmes synchrones classiques.

3.5.1 Mise à jour de registres en plusieurs cycles

La construction **exec** e **default** e_0 **reset** e_1 évalue e de façon répétée et rend disponible un résultat valide pendant exactement un cycle, à l'instant où l'évaluation de e se termine. La valeur par défaut (donnée par e_0) est retournée le reste du temps. Or, il est fréquent de vouloir conserver le résultat valide jusqu'à l'arrivée du résultat suivant. Cela se programme en ECLAT en combinant les constructions **exec** et **reg**, comme illustré sur la figure 3.17.

La fonction `current` (figure 3.17) est une variante d'**exec** : elle prend en argument une fonction `f`, son argument `x`, une valeur par défaut `d` et un booléen `r` pour réinitialiser ou non le calcul de f. Le registre contient la valeur à mémoriser et sa fonction de mise à jour intègre un bloc **exec**. Si le calcul sous le **exec** se termine, alors le registre est mis à jour avec la nouvelle valeur. Il est inchangé le reste du temps.

```
1  let current ((f,x),(d,r)) =
2    let (o,_) =
3      reg (fun (pre_v,_) ->
4            let (v,rdy) = exec f(x) default d reset r in
5            if rdy then (v,rdy) else (pre_v,rdy))
6      init (d,false)
7    in o ;;
8
9  let main n =
10   current((fibonacci,n),(-42,false)) ;;
```

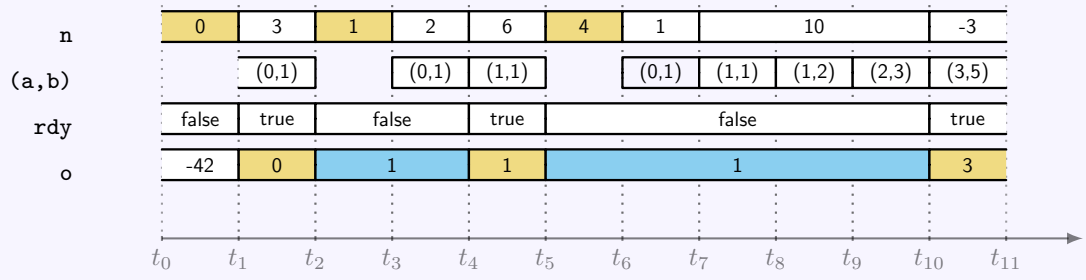


FIGURE 3.17 Mise à jour d'un registre par un calcul long

3.5.2 Retarder un flot

Au *niveau portes logiques*, chaque registre implante un retard, son entrée est recopiée sur sa sortie à chaque *tick* d'horloge. Or, le comportement de l'opérateur **reg** d'ECLAT est assez différent : lorsqu'il est exécuté (ce qui dépend du flot de contrôle), il calcule *instantanément* une sortie en fonction de la valeur contenue dans le registre ; la valeur de sortie est ensuite connectée au registre pour mettre à jour celui-ci au *tick* d'horloge suivants.

Cette construction **reg** est suffisamment générale pour implanter un retard, comme l'opérateur **fby** de LUSTRE, qui prend un flot en entrée et retourne un flot retardé de un cycle.

La figure 3.18 définit en ECLAT une fonction `fby`. Cette fonction a le schéma de type $\forall\ 'B \cdot (\ 'B\ *\ 'B\ ) \xrightarrow{0} 'B$. Elle utilise un bloc **reg** contenant une paire formée de l'ancienne valeur mémorisée et de la valeur courante). L'argument `x` sert à donner une valeur pour le premier cycle. Les valeurs suivantes de `x` sont ignorées, comme le suggère le chronogramme.

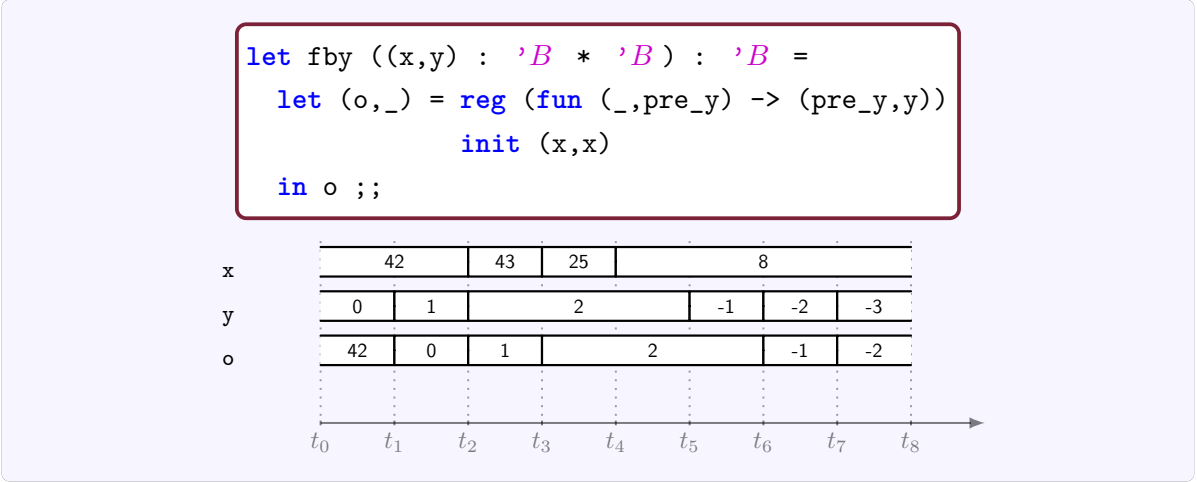


FIGURE 3.18 Retarder un flot en ECLAT

3.5.3 Encoder le contrôle en style flot de données

Un exemple classique en programmation synchrone est le circuit ABRO. Ce circuit attend que ses entrées `A` et `B` aient pris la valeur `true` pour retourner `true` en sortie `o` pendant un cycle, avec de plus une entrée `R` pour réinitialiser ce comportement. Les entrées `A` et `B` et `R` peuvent varier dans n'importe quel ordre, y compris simultanément. Le programme doit gérer les différents scénarios possibles de façon exhaustive et déterministe.

Cela correspond à l'automate de Mealy figure 3.19. L'automate comprend quatre états reliés par des transitions. Il est au plus dans un état à chaque cycle. Les transitions sont de la forme $q \xrightarrow{I/O} q'$ et se lisent : "l'automate dans l'état q passe dans l'état q' et émet le signal O si le signal I est présent". L'important ici est la quantité de détails dans l'automate pour traiter chaque cas : soit `A` est lue, puis `B` ; soit `B` est lue puis `A` ; soit `A` et `B` sont lues simultanément ; à cela s'ajoute le traitement de l'entrée `R` pour réinitialiser le circuit à tout moment. L'intérêt d'un langage synchrone est de pouvoir définir succinctement et composer ce type de comportement.

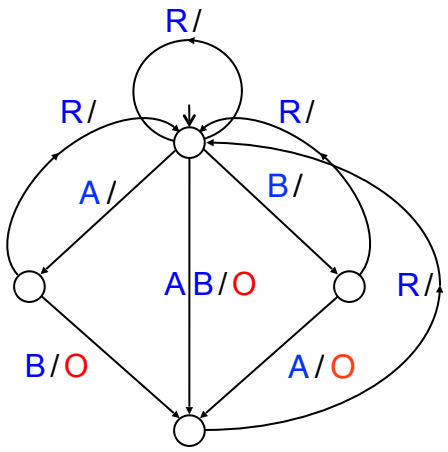


FIGURE 3.19 L'automate d'ABRO *(source Gérard Berry)*

Une implantation d'ABRO est définie en ECLAT sur la figure 3.20. C'est une formulation flot de données qui consiste à définir un circuit `await` soutenant la valeur `true` en sortie dès que son entrée `i` prend la valeur de `true`. Une seconde entrée est utilisée pour réinitialiser ce comportement. La fonction `fby` permet alors de programmer un détecteur de front montant sur une entrée booléenne. Le circuit `abro` compose simplement deux instances de `await` en parallèle et détecte, avec `edge`, l'instant où les deux `await` retournent `true`.

```
1  let await (i,r) =
2    let o = reg (fun s ->
3                  (s or i) & not r)
4            init false
5    in o ;;
```

```
6  let edge i =
7    (not (fby(false,i))) & i ;;
8
9  let abro (a,b,r) =
10   let o = edge (await(a,r) &
11                 await(b,r))
12   in o ;;
```

FIGURE 3.20 ABRO en ECLAT : "attendre `A` et `B` puis émettre une fois `O` sauf si `R`"

L'exemple `abro` est classiquement utilisé pour montrer la composition des comportements, de façon concise. La fonction `abro` se généralise en une fonction `abcro` avec une entrée `c` supplémentaire en composant deux instances de `abro`.

Deux implantations possibles d'`abcro` sont définies sur la figure 3.21. Celle de gauche comporte deux instances de `abro` :

— la première instance de `abro` s'applique sur `a` et `b` et produit une sortie `t` ;

— la seconde s'applique à `t` et `c`.

Celle de droite compose directement trois instances de `await`. L'avantage de la version `abcro` est

que la définition `abro` d'origine n'a pas été modifiée, gagnant ainsi en modularité [10]. Cela évite les erreurs de programmation dues aux *copier-coller* (`abcro-flat` étant justement une version modifiée de `abro`) et permet de vérifier les composants d'un programme de façon modulaire.

```
1 let abcro (a,b,c,r) =
2   let t = abro(a,b,r) in
3   let o = abro(t,c,r)
4   in o ;;
```

```
1 let flat_abcro (a,b,c,r) =
2   let o = edge ((await(a,r) &
3                  await(b,r)) &
4                  await(c,r))
5   in o ;;
```

FIGURE 3.21 Deux implantations possible du circuit ABCRO

À noter cependant que la version `abcro` comporte quatre instances de `await` tandis que la version `flat_abcro` n'en comporte que trois. Si l'on compile ces deux versions en VHDL (en utilisant le compilateur ECLAT) et que l'on synthétise la description de matériel obtenue sur un FPGA Intel (en utilisant le synthétiseur logique QUARTUS), on observe que :

— `abcro` comporte 13 éléments logiques (dont 8 registres) et son chemin critique conduit à une fréquence maximale de 414 MHz *(voir net-list A.6 avec lien arrière pour revenir)*.

— `abcro_flat` comporte 9 éléments logiques (dont 6 registres) et son chemin critique conduit à une fréquence maximale de 751 MHz.

La version modulaire est donc moins efficace. Chaque instance de `await` contient un bloc **`reg`**. La réalisation de chaque bloc **`reg`** contient deux registres [11]. La dépendance de donnée `t` entre les deux instances d'`abro` composées séquentiellement dans `abcro` cause un quasi doublement du chemin critique ($751/414 \approx 1.8$) ; cela n'a pas d'impact tant que la fréquence maximale reste inférieure à la fréquence de l'horloge physique du FPGA.

3.5.4 Observateur synchrone

Un observateur synchrone est un programme synchrone qui vérifie une propriété fonctionnelle d'un autre programme [95].

Par exemple, le programme sur la figure 3.22 définit un compteur incrémenté à chaque fois que son entrée `event` vaut `true`. Nous programmons un observateur pour tester si la valeur

10. C'est le fameux principe *"WTO, Write Things Once"* défendu par Gérard Berry notamment [28].

11. L'un des deux registres transporte l'état proprement dit (mis à jour progressivement au cours de l'exécution du programme). L'autre registre contient un booléen utilisé pour gérer l'initialisation du bloc **`reg`**, c'est à dire l'activation de la clause **`init`** à la première exécution de celui-ci. Par exemple, considérons le programme suivant :

$$\texttt{let main } (a,b) \texttt{ = if } b \texttt{ then (reg (fun } x \to x+1\texttt{) init } a\texttt{) else 0}$$

le bloc **`reg`** est initialisé à la valeur de l'entrée a dès que l'entre b vaut `true`.

du compteur est toujours positive (compte-tenu des risques de débordements arithmétiques) et éteindre une LED sur le circuit FPGA cible dès que cette propriété n'est plus vérifiée.

La propriété `always_positive` à vérifier est directement exprimée en ECLAT : le registre (lignes 11 et 12) contient initialement la valeur `true` et prend définitivement `false` dès que la valeur courante `o` de la sortie à tester est négative. La fonction `observer` prend en argument une fonction `f`, une entrée `i` et une propriété `p` à tester sur la sortie `o` de `f(i)`. L'observateur retourne la sortie `o` ainsi qu'un booléen `ok` qui vaut `p(i,o)`. Le point d'entrée (`main`) instancie l'observateur avec le compteur et la propriété `always_positive` à vérifier.

```
1  let count(event) =
2    reg (fun c ->
3          if event then c + 1
4          else c) init 0 ;;
5
6  let observer ((f,i),p) =
7    let o = f(i) in
8    let ok = p(i,o) in
9    (o,ok) ;;
```

```
10 let always_positive (_,o) =
11   reg (fun s -> s & 0 < o)
12   init true ;;
13
14 let main(event) =
15   let (o,ok) =
16     observer((count,event),
17              always_positive)
18   in ok ;;
```

FIGURE 3.22 Observateur synchrone en ECLAT

3.5.5 Modéliser un environnement

L'environnement physique peut être modélisé de façon synchrone par un programme ECLAT. Cela permet de modéliser l'interaction entre un système réactif et son environnement réagissant en *boucle fermée* : à chaque cycle, l'environnement utilise la sortie précédente du système pour déterminer une nouvelle entrée, puis le système calcule une sortie en fonction de cette entrée.

Par exemple, la figure 3.23 comporte trois fonctions : la fonction `sum` implante un système réactif qui retourne la somme de l'historique de ses entrées ; la fonction `environment` incrémente son entrée si celle-ci est inférieure à 10 et sinon retourne -1.

La fonction `main` (qui est le point d'entrée, figure 3.23) compose l'environnement et le système réactif. L'exécution du système est représentée sur un chronogramme dans le bas de la figure 3.23. Le flot `pre_o` correspond au flot `o` retardé d'un cycle et initialisé à la valeur `0` de la clause `init` du bloc `reg`. La nouvelle entrée `i` est calculée en fonction de `pre_o` par la fonction `environment`. Le programme calcule la sortie courante `o` en fonction de l'entrée `i`.

```
1  let sum i =
2    reg (fun s -> s + i) init 0 ;;
3
4  let environment o =
5    if o < 10 then o + 1
6    else (-1) ;;
```

```
7  let main () =
8    reg (fun pre_o ->
9      let i = environment pre_o in
10     let o = sum i in
11     o)
12   init 0 ;;
```

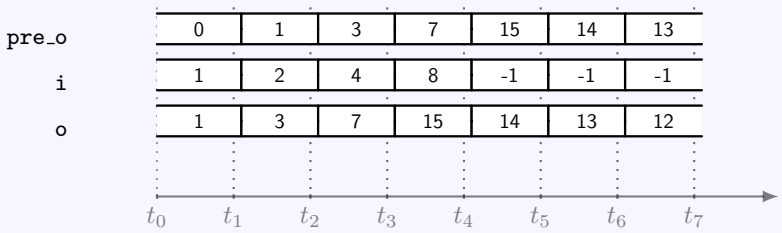


FIGURE 3.23 Modéliser un environnement physique de façon synchrone

3.6 Conclusion

Ce chapitre a présenté la syntaxe d'ECLAT et sa sémantique intuitive ainsi que des exemples illustrant les différents aspects de la programmation en ECLAT. Les domaines d'application visés sont l'implantation d'accélérateurs de calcul, la conception de systèmes embarqués réactifs et la description de circuits au niveau transfert de registres. Tout l'intérêt d'ECLAT est de pouvoir mélanger ces différents types d'applications au sein d'un même programme, avec à la fois un contrôle fin sur les détails temporels bas-niveau et un style de programmation généraliste, fonctionnel-impératif *à la ML*. Vu la finesse de programmation en ECLAT, les programmeuses et programmeurs ont besoin d'une sémantique explicitant le comportement temporel des programmes. La sémantique d'ECLAT est définie au chapitre suivant.

4 Sémantique d'Eclat

Ce chapitre présente une formalisation d'Eclat, incluant d'une part un système de types et d'autre part une sémantique opérationnelle synchrone. Cela vise à offrir aux programmeuses et programmeurs un contrôle fin sur le comportement fonctionnel et temporel des circuits réalisant les programmes Eclat, en conciliant parallélisme et déterminisme, avec une garantie de réactivité par typage statique.

4.1 Sémantique statique : typage et analyse de réactivité

Eclat est un langage statiquement typé. Son système de types est basé sur celui de ML [141], ce qui constitue un bon compromis entre expressivité et simplicité de programmation. La construction `let` expose du polymorphisme paramétrique en liant des noms à des schémas de types en vue de les instancier dans différents contextes, ce qui apporte de la modularité.

Le caractère *instantané* d'un programme est une propriété dynamique qui conditionne sa sûreté d'exécution et sa réactivité. Cette propriété doit être satisfaite pour toute exécution du programme, quelles que soient ses entrées ; nous la vérifions donc *statiquement*.

Eclat incorpore cette analyse statique dans un *système de types et d'effets* [205] où les effets sont des *durées*. Cela modélise le passage du temps au cours de l'exécution des programmes.

Le jugement $\Gamma_0 \vdash \pi : \tau|\mathbf{0}$ signifie "dans l'environnement initial Γ_0, le programme π est instantané et a le type τ". Pour un programme π donné, le système infère à la fois le type du programme (les annotations de types étant généralement omises) et une estimation *pire cas* de son temps d'exécution ; le programme est rejeté s'il est mal typé ou s'il n'est pas instantané.

Le jugement $\Gamma \vdash e : \tau|\delta$ signifie "dans l'environnement de typage Γ, l'expression e a le type τ et un temps d'exécution inférieur ou égal à δ cycles". Un programme π est bien typé si c'est une fonction bien typée de type $\mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'$ dans l'environnement initial Γ_0 (règle Ty-Prog).

$$\textsc{Ty-Prog}\ \frac{\Gamma_0 \vdash \texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau}' \to e : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'|\mathbf{0}}{\Gamma_0 \vdash \texttt{let}\ f = \texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau}' \to e : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'|\mathbf{0}}$$

Un environnement de typage Γ est une fonction partielle des noms (x) vers des schémas de types (σ). On note $\Gamma[x : \sigma]$ l'extension de l'environnement de typage Γ avec le nom x de schéma de type σ. La fonction $\Gamma(x)$ de recherche dans l'environnement de typage est définie par $\Gamma[x : \sigma](x) = \sigma$ et $\Gamma[x : \sigma](y) = \Gamma(y)$ si $x \neq y$. On note $\Gamma[p : \tau]$ l'extension de l'environnement de typage Γ avec le motif p de schéma type trivial τ, définie par :

$$\begin{array}{lcl} \Gamma[\texttt{()} : \texttt{unit}] & = & \Gamma \\ \Gamma[x : \tau] & = & \Gamma[x : \tau] \\ \Gamma[\texttt{(}p\texttt{,}p'\texttt{)} : \tau \texttt{ * } \tau'] & = & \Gamma[p : \tau][p' : \tau'] \end{array}$$

Construire $\Gamma[p : \tau]$ suppose que p et τ soient *compatibles* structurellement ; par exemple le motif $\texttt{(}p\texttt{,}p'\texttt{)}$ n'est pas compatible avec le type `unit`. La compatibilité entre un motif p et un type τ est une relation $p \sim \tau$ définie par :

$$\begin{array}{rcll} \texttt{()} & \sim & \texttt{unit} & \\ x & \sim & \tau & \\ \texttt{(}p\texttt{,}p'\texttt{)} & \sim & \tau \texttt{ * } \tau' & \text{si } p \sim \tau \text{ et } p' \sim \tau' \end{array}$$

Le jugement de typage des expressions est défini par un ensemble de règles d'inférence. Les règles Ty-Let-poly et Ty-Var implantent le polymorphisme paramétrique de ML. Le type de certaines valeurs fonctionnelles liées à un nom x par un `let` est généralisé, formant un schéma de type $\sigma = \text{Gen}(\tau, \Gamma)$ associé à x dans un nouvel environnement de typage $\Gamma[x : \sigma]$. On rappelle que tout type est aussi un schéma de type trivial sans quantification (cf le langage de types d'Eclat sur la figure 3.1 page 29).

On note $\mathcal{FV}(\tau)$ (*resp.* $\mathcal{FV}(\Gamma)$) l'ensemble des variables $\texttt{'}v$ qui apparaissent libres dans le type τ (*resp.* l'environnement Γ). La fonction $\text{Gen}(\tau, \Gamma)$ construit un schéma de type en généralisant, dans τ, toutes les variables $\texttt{'}v$ (de *types*, de *types de base*, de *tailles* ou de *durées*) qui n'apparaissent pas libres dans l'environnement de typage Γ :

$$\text{Gen}(\tau, \Gamma) \stackrel{\text{def}}{=} \forall\ \texttt{'}v_1\ \cdots\ \texttt{'}v_n\ \cdot \tau \text{ où } \{\ \texttt{'}v_1\ , \cdots\ \ \texttt{'}v_n\ \} \stackrel{\text{def}}{=} \mathcal{FV}(\tau) \setminus \mathcal{FV}(\Gamma)$$

La relation $\tau \leq \sigma$, qui se lit "τ est une instance du schéma de type σ", est définie par :

$$\begin{array}{lcll} \tau & \leq & \tau & \\ \tau & \leq & \forall\ \texttt{'}a\ \cdot \sigma & \text{s'il existe } \tau' \text{ tel que } \tau \leq \sigma[\ \texttt{'}a\ \mapsto \tau'] \\ \tau & \leq & \forall\ \texttt{'}B\ \cdot \sigma & \text{s'il existe } \mathring{\tau}' \text{ tel que } \tau \leq \sigma[\ \texttt{'}B\ \mapsto \mathring{\tau}'] \\ \tau & \leq & \forall\ \texttt{'}d\ \cdot \sigma & \text{s'il existe } \delta \text{ tel que } \tau \leq \sigma[\ \texttt{'}d\ \mapsto \delta] \\ \tau & \leq & \forall\ \texttt{'}N\ \cdot \sigma & \text{s'il existe } \theta \text{ tel que } \tau \leq \sigma[\ \texttt{'}N\ \mapsto \theta] \end{array}$$

Dans certains cas, la liaison `let` $p = e$ n'est pas généralisée (règle Let-mono). Un critère syntaxique poly(e) est en effet utilisé pour limiter la généralisation. Cela permet d'éviter, par exemple, que l'expression `(42:int<` $'N_0$ `>)` soit associée au schéma de type $\forall$ $'N$ `·int<` $'N$ `>`. Le critère poly(e) est vrai dans les cas suivants (et faux pour tout autre forme d'expression) :

$$\text{poly}(\texttt{fun (}p\texttt{:}\tau\texttt{) : }\mathring{\tau}' \to e)$$
$$\text{poly}(\texttt{fix } f \texttt{ (fun (}p\texttt{:}\tau\texttt{) : }\mathring{\tau}' \to e))$$

La construction `let` (règles Ty-Let-poly et Ty-Let-Mono) réalise une composition séquentielle : la liaison est d'abord calculée avec un temps d'exécution δ, suivie du corps du `let` avec un temps d'exécution δ'. Le temps d'exécution est alors exactement $\delta + \delta'$, la synchronisation "`in`" étant instantanée.

Ty-Let-poly

$$\frac{\begin{array}{c}\text{poly}(e) \text{ est vrai} \\ \Gamma \vdash v : \tau|\delta \qquad \sigma \stackrel{\text{def}}{=} \text{Gen}(\tau, \Gamma) \\ \Gamma[x:\sigma] \vdash e' : \tau'|\delta'\end{array}}{\Gamma \vdash \texttt{let } x\texttt{:}\tau = e \texttt{ in } e' : \tau'|\delta + \delta'}$$

Ty-Let-Mono

$$\frac{\text{poly}(e) \text{ est faux ou } p \text{ n'est pas une variable} \qquad \Gamma \vdash e : \tau|\delta \qquad p \sim \tau \qquad \Gamma[p:\tau] \vdash e' : \tau'|\delta'}{\Gamma \vdash \texttt{let } p\texttt{:}\tau = e \texttt{ in } e' : \tau'|\delta + \delta'}$$

Le typage d'une occurrence de variable x (par la règle Ty-Var) assigne à cette occurrence de x une instance du schéma type σ associé à x dans l'environnement de typage. L'accès aux valeurs des variables est instantané (**0**).

Ty-Var
$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau|\mathbf{0}}$$

L'évaluation des constantes est instantanée (**0**) comme spécifié par les règles suivantes. Les éléments d'un vecteur doivent être du même type, qui doit être un type de base.

Ty-Unit
$$\Gamma \vdash \texttt{()} : \texttt{unit}|\mathbf{0}$$

Ty-True
$$\Gamma \vdash \texttt{true} : \texttt{bool}|\mathbf{0}$$

Ty-False
$$\Gamma \vdash \texttt{false} : \texttt{bool}|\mathbf{0}$$

Ty-Int
$$\Gamma \vdash \texttt{(}l\texttt{:int<}\theta\texttt{>)} : \texttt{int<}\theta\texttt{>}|\mathbf{0}$$

Ty-Vect
$$\frac{\Gamma \vdash c_i : \mathring{\tau}|\mathbf{0} \qquad i \in \{1, \cdots n\}}{\Gamma \vdash \texttt{\{}c_1\texttt{,} \cdots c_n\texttt{\}} : \mathring{\tau}\ \texttt{vect<}\theta\texttt{>}|\mathbf{0}}$$

Le type d'une paire `(`e_1`,`e_2`)` – y compris si cette paire est une constante `(`c_1`,`c_2`)` – est de la forme τ_1 `*` τ_2 avec e_1 de type τ_1 et pire temps d'exécution δ_1 et e_2 de type τ_2 et pire temps d'exécution δ_2 (règle Ty-Pair). Les deux expressions sont évaluées séquentiellement ; le temps d'exécution résultant est $\delta_1 + \delta_2$.

Ty-Pair
$$\frac{\Gamma \vdash e_1 : \tau_1|\delta_1 \qquad \Gamma \vdash e_2 : \tau_2|\delta_2}{\Gamma \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} : \tau_1 \texttt{ * } \tau_2|\delta_1 + \delta_2}$$

Le type d'une paire parallèle $(e_1 \| e_2)$ est de la forme $\tau_1 * \tau_2$ avec e_1 de type τ_1 et pire temps d'exécution δ_1 et e_2 de type τ_2 et pire temps d'exécution δ_2 (règle Ty-Par). Les deux expressions sont évaluées en parallèle ; le temps d'exécution résultant est $\max(\delta_1, \delta_2)$.

$$\textsc{Ty-Par}\ \frac{\Gamma \vdash e_1 : \tau_1|\delta_1 \qquad \Gamma \vdash e_2 : \tau_2|\delta_2}{\Gamma \vdash (e_1 \| e_2) : \tau_1 * \tau_2|\max(\delta_1, \delta_2)}$$

Remarque : De par l'abstraction **0/1** pour les durées, si $\Gamma \vdash (e_1 \| e_2) : \tau_1 * \tau_2|\delta$, alors $\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2|\delta$ et réciproquement.

L'application de fonction évalue l'expression en position fonctionnelle, puis l'argument, puis le corps de la fonction. Le temps d'exécution pire cas est donc la somme des temps d'exécution de ces trois expressions (règle Ty-App). En Eclat toutefois, les expressions non-instantanées ne peuvent pas retourner de fonctions ; c'est pourquoi l'expression en position fonctionnelle doit être instantanée (**0**).

$$\textsc{Ty-App}\ \frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\delta} \mathring{\tau}|\mathbf{0} \qquad \Gamma \vdash e_2 : \tau_2|\delta_2}{\Gamma \vdash e_1\ e_2 : \mathring{\tau}|\delta_2 + \delta}$$

Le type de chaque opérateur est une instance d'un schéma de type σ prédéfini dans un environnement de typage des opérateurs Δ (règle Ty-Operator). Les schémas de types $\Delta(op)$ des opérateurs ont été listés sur la figure 3.3.

$$\textsc{Ty-Operator}\ \frac{\Delta(op) = \sigma \qquad \tau \leq \sigma}{\Gamma \vdash op : \tau|\mathbf{0}}$$

Toute fonction $(\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)$ est une valeur dont la création est instantanée (**0**) ; son type est de la forme $\tau \xrightarrow{\delta} \mathring{\tau}'$ où τ est le type de l'argument, δ est le pire temps d'exécution du corps e de la fonction et $\mathring{\tau}'$ est le type du résultat, lequel doit être un type de base (règle Ty-Fun). La notation $p \sim \tau$ introduite précédemment indique que le motif p est compatible avec le type τ ; par exemple, si $(p_1, p_2) \sim \tau$, alors τ est de la forme $\tau_1 * \tau_2$ avec $p_1 \sim \tau_1$ et $p_2 \sim \tau_2$.

$$\textsc{Ty-Fun}\ \frac{p \sim \tau \qquad \Gamma[p : \tau] \vdash e : \mathring{\tau}'|\delta}{\Gamma \vdash \texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e : \tau \xrightarrow{\delta} \mathring{\tau}'|\mathbf{0}}$$

Toute fonction récursive $\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)$ est aussi une valeur dont la création est instantanée (**0**). Cependant, son type est de la forme $\tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'$ car l'appel direct prend un cycle, qui s'ajoute à la durée δ de l'appel proprement dit. Le corps e est typé dans un environnement où f est présent et de type $\tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'$. Tout appel direct à f prend un cycle (**1**) avant d'évaluer le corps e en δ cycles. Le temps d'exécution réel de e (qui est abstrait par δ) est infini dans le pire cas [1] (règle Ty-Fix).

1. Inférer le temps d'exécution d'une fonction récursive (qui peut ne pas terminer) est indécidable dans le cas général. L'abstraction **0/1** mise en œuvre ici pour représenter les temps d'exécution vise uniquement à vérifier que certaines expressions sont *toujours* instantanées (**0**).

$$\textsc{Ty-Fix}\ \frac{p \sim \mathring{\tau} \qquad \Gamma[f : \tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'][p : \tau] \vdash e : \mathring{\tau}'|\delta}{\Gamma \vdash \texttt{fix}\ f\ \texttt{(fun (}p\texttt{:}\tau\texttt{) :}\ \mathring{\tau}' \to e\texttt{)} : \tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'|\mathbf{0}}$$

Les conditionnelles évaluent d'abord leur condition, puis la branche *then* ou la branche *else*. Le temps d'exécution *pire cas* d'une conditionnelle est donc le temps d'exécution de la condition auquel s'ajoute le temps d'exécution maximal parmi les deux branches (règle Ty-If). Pour des raisons d'implantation, le type d'une conditionnelle doit être un type de base. Par exemple, l'expression `(if true then + else -)` est rejetée car `+` et `-` ont un type fonctionnel.

$$\textsc{Ty-If}\ \frac{\Gamma \vdash e_1 : \texttt{bool}|\delta_1 \qquad \Gamma \vdash e_2 : \mathring{\tau}|\delta_2 \qquad \Gamma \vdash e_3 : \mathring{\tau}|\delta_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathring{\tau}|\delta_1 + \max(\delta_2, \delta_3)}$$

La construction `reg` représente un registre initialisé à une valeur de type $\mathring{\tau}$ et mis à jour avec une fonction instantanée de type $\mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}$. La valeur de retour de la construction `reg` est la valeur de type $\mathring{\tau}$ nouvellement mise à jour (règle Ty-Reg). Le type $\mathring{\tau}$ est un type de base car la fonction s'applique au résultat qu'elle produit (la valeur précédemment stockée dans le registre), en rappelant qu'en Eclat, le type de retour d'une fonction est toujours un type de base.

$$\textsc{Ty-Reg}\ \frac{\Gamma \vdash \texttt{fun (}p\texttt{:}\mathring{\tau}\texttt{) :}\ \mathring{\tau} \to e : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}|\mathbf{0} \qquad \Gamma \vdash e_0 : \mathring{\tau}|\mathbf{0}}{\Gamma \vdash \texttt{reg (fun (}p\texttt{:}\mathring{\tau}\texttt{) :}\ \mathring{\tau} \to e\texttt{) init}\ e_0 : \mathring{\tau}|\mathbf{0}}$$

La construction `exec` e_1 `default` e_2 `reset` e_3 représente un calcul évalué pas à pas et retournant soit une valeur par défaut (donnée par l'expression instantanée e_2), soit le résultat du calcul de e_1. Tout comme dans le cas `reg`, la valeur retournée par l'expression sous le `exec` doit avoir un type de base (règle Ty-Exec). Elle est associée à un booléen qui vaut `true` lorsque l'évaluation de e_1 se termine, et `false` sinon. Enfin, l'expression e_3 pour la réinitialisation conditionnelle du calcul doit être de type `bool`.

$$\textsc{Ty-Exec}\ \frac{\Gamma \vdash e_1 : \mathring{\tau}|\delta \qquad \Gamma \vdash e_2 : \mathring{\tau}|\mathbf{0} \qquad \Gamma \vdash e_3 : \texttt{bool}|\mathbf{0}}{\Gamma \vdash \texttt{exec}\ e_1\ \texttt{default}\ e_2\ \texttt{reset}\ e_3 : \mathring{\tau}\ \texttt{*}\ \texttt{bool}|\mathbf{0}}$$

Remarque : Comme l'indique la règle Ty-Exec, le temps d'exécution de l'expression `exec` e_1 `default` e_2 `reset` e_3 est toujours nul ($\mathbf{0}$), quelle que soit la durée de l'expression e_1 sous le `exec`. Rappelons que `halt()` est une expression qui ne termine jamais. Ainsi :

— `(exec 42+1 default 0 reset` e_3`)` retourne toujours instantanément la valeur `(43,true)` quelle que soit e_3 ; en effet, lorsque e_3 s'évalue en `true`, le calcul `42+1` est réinitialisé puis s'exécute dans le même cycle et retourne la valeur `43` ;

— l'expression `(exec halt() default 0 reset` e_3`)` retourne toujours instantanément la valeur par défaut `(0,false)` : elle peut être réinitialisée avec e_3, mais cela redémarre l'exécution de `halt()` qui ne termine jamais ;

— l'expression `(exec (if x then 42 else halt()) default 0 reset x)` rend toujours instantanément `(42,true)` quand `x` vaut `true` ou `(0,false)` quand `x` vaut `false`. En effet, quand `x` vaut `true`, alors l'expression sous le `exec` est réinitialisée et s'évalue en `42` ; quand au contraire `x` vaut `false`, le calcul `halt()` ne termine pas et c'est donc la valeur `0` qui est retournée par défaut.

Exemples

La dérivation de typage suivante montre que la fonction *compose*, définie en ECLAT par `fun ((`f`,`g`),`x`)` $\to$ $f(g(x))$, est bien typée. Le type de *compose* (qui est de la forme $\tau \xrightarrow{\delta} \mathring{\tau}'$) garde trace des temps d'exécution des deux arguments fonctionnels f et g composés séquentiellement.

$$\textsc{Ty-Fun}\ \frac{\textsc{Ty-App}\ \dfrac{\textsc{Ty-Var}\ \dfrac{(\mathring{\tau}_1 \xrightarrow{\delta_2} \mathring{\tau}_2) \leq \Gamma(f)}{\Gamma \vdash f : \mathring{\tau}_1 \xrightarrow{\delta_2} \mathring{\tau}_2 | \mathbf{0}} \qquad \textsc{Ty-App}\ \dfrac{\textsc{Ty-Var}\ \dfrac{(\tau_x \xrightarrow{\delta_1} \mathring{\tau}_1) \leq \Gamma(g)}{\Gamma \vdash g : \tau_x \xrightarrow{\delta_1} \mathring{\tau}_1 | \mathbf{0}} \qquad \dfrac{\tau_x \leq \Gamma(x)}{\Gamma \vdash x : \tau_x | \mathbf{0}}}{\Gamma \vdash g(x) : \mathring{\tau}_1 | \delta_1 + \mathbf{0}}}{\overbrace{\emptyset[((f,g),x) : \tau]}^{\Gamma} \vdash f(g(x)) : \mathring{\tau}_2 | \delta_1 + \delta_2} \quad ((f,g),x) \sim \tau}{\emptyset \vdash \texttt{fun}\ (((f,g),x) : \tau)\ :\ \mathring{\tau}_2 \to f(g(x)) : \underbrace{((\mathring{\tau}_1 \xrightarrow{\delta_2} \mathring{\tau}_2) * (\tau_x \xrightarrow{\delta_1} \mathring{\tau}_1)) * \tau_x)}_{\tau} \xrightarrow{\delta_1+\delta_2} \mathring{\tau}_2 | \mathbf{0}}$$

L'analyse de réactivité réalisée par le système de types est sûre, mais elle peut être imprécise. Par exemple, étant donnée une expression e_1 non instantanée, l'expression `(if true then () else` e_1`)` est considérée comme non instantanée (**1**). En effet, la règle TY-IF calcule la durée maximale entre les deux branches, et ce indépendamment de la valeur de la condition. Or, à l'exécution, cette expression termine toujours en zéro cycle car la branche *else* n'est jamais exécutée.

$$\textsc{Ty-If}\ \frac{\textsc{Ty-True}\ \dfrac{}{\emptyset \vdash \texttt{true} : \texttt{bool} | \mathbf{0}} \qquad \textsc{Ty-Unit}\ \dfrac{}{\emptyset \vdash () : \texttt{unit} | \mathbf{0}} \qquad \dfrac{\vdots}{\emptyset \vdash e_1 : \texttt{unit} | \mathbf{1}}}{\emptyset \vdash \texttt{if true then () else } e_1 : \texttt{unit} | \underbrace{\mathbf{0} + \max(\mathbf{0}, \mathbf{1})}_{=\mathbf{1}}}$$

Polymorphisme de taille

La gestion des tailles des valeurs (autrement dit, le nombre de fils nécessaire pour représenter ces valeurs) est un problème classique en description de matériel, avec une tension entre précision (pour tailler le matériel sur mesure) et abstraction (pour offrir de la modularité et de la sûreté [2]). Dans ce contexte, ECLAT supporte une forme limitée de typage dépendant [39].

2. En garantissant, par exemple, qu'un additionneur traite des entiers de même taille (en nombre de bits), sans pour autant dupliquer la description de l'additionneur pour chaque taille d'entier considérée.

Les entiers et vecteurs d'Eclat sont des valeurs dont les tailles sont connues à la fois statiquement et à l'exécution. Ces tailles apparaissent explicitement dans les programmes sources et dans les valeurs. Chaque constante entière `(`l`:int<`θ`>)` est formée d'un littéral entier l sur θ bits. Chaque vecteur `{`c_1`,` $\cdots$ c_n`}` est une collection de n éléments de même type. Les fonctions `(fun (`p`:`τ`)` : $\mathring{\tau}'$ $\rightarrow$ e`)` et `(fix` f `(fun (`p`:`τ`)`: $\mathring{\tau}'$ $\rightarrow$ e`))` sont annotées avec le type τ de leur argument et leur type de retour $\mathring{\tau}'$. Les opérateurs `resize_int<`θ`>` et `vect_create<`θ`>` sont eux-mêmes paramétrés par des annotations de tailles. Enfin, les opérateurs peuvent dépendre des tailles des valeurs manipulées : l'opérateur `+` par exemple attend deux entiers de taille n (en nombre de bits) et retourne un entier de taille n.

Or, dans le corps d'une fonction polymorphe paramétrique f, chaque annotation de taille peut être une variable $'N$ liée dans le schéma de type de f. Les variables de tailles doivent donc être instanciées *pendant l'exécution*, à chaque appel à f.

La figure 4.1 montre un exemple de programme exploitant le polymorphisme de taille pour manipuler en tant que valeur la taille d'un entier à l'exécution. La fonction `size_int` a le schéma de type $\forall\ 'N\ \cdot$ `int<`$'N$`>` $\xrightarrow{0}$ `int<16>`. Le paramètre de taille $'N$ est instancié :

— avec la taille `12` lors de l'appel à `size_int`, ligne 5,

— puis avec la taille `8` lors de l'appel `size_int`, ligne 6.

```
1 let size_int = fun (x:int<'N>) : int<16> ->
2   vect_size(vect_create<'N>(true)) ;;
3
4 let main () =
5   size_int (0:int<12>)
6   + size_int (0:int<8>) ;;
```

Figure 4.1 Exemple de polymorphisme de taille en Eclat

Au cours de chacun de ces deux appels, un vecteur de taille $'N$ (valant respectivement `12` puis `8`) est créé et la taille de ce vecteur est retournée. Les deux valeurs résultantes sont additionnées ; le programme retourne donc la valeur `(20:int<16>)`, c'est-à-dire `12`+`8` sur 16 bits.

Dans la section suivante, nous définissons la sémantique synchrone d'Eclat en se focalisant sur le comportement temporel des programmes bien typés. Cette sémantique doit gérer le polymorphisme de taille. Cependant, pour simplifier la présentation, nous limitons l'expressivité du système de types (ce qui conduit à rejeter d'avantage de programmes) et nous exploitons *à l'exécution* l'information de type contenue dans les valeurs :

(1) Nous amendons la règle Ty-Fun (*resp.* Ty-Fix) pour limiter les inconnues de tailles dans le type de retour de chaque fonction : étant donnée une fonction `(fun (`p`:`τ`)` : $\mathring{\tau}'$ $\rightarrow$ e`)` (*resp.* une fonction récursive terminale `(fix` f `(fun (`p`:`τ`)` : $\mathring{\tau}'$ $\rightarrow$ e`))`), toute inconnue de taille $'N$ qui apparaît libre dans le corps e mais qui n'apparaît pas libre

dans le type τ de l'argument est remplacée par la taille `32`. Par exemple, la fonction `(fun (`x`:int< 'N >) → (0:int< 'N >))` a le type `int< 'N >` $\xrightarrow{\mathbf{0}}$ `int< 'N >` mais la fonction `(fun (`x`:int< 'N`$_1$`>) → (0:int< 'N`$_2$`>))` a le type `int< 'N`$_1$`>` $\xrightarrow{\mathbf{0}}$ `int<32>`.

(2) chaque valeur v possède une information de type à l'exécution, notée $\mathbf{type}(v)$, qui est encodée dans v et qui peut être extraite par un accès dynamique : cela permet de résoudre et propager les informations de tailles à l'exécution compte tenu de la limitation **(1)**.

L'information de type $\mathbf{type}(v)$ est définie en fonction de v sur la figure 4.2. Pour résoudre les inconnues de tailles dans une expression e, on note $e[\texttt{'}N := n]$ la substitution de l'inconnue de taille $\texttt{'}N$ par le littéral de taille n dans l'expression e. Plus généralement, on définit figure 4.2 la substitution $e[\tau \mapsto \tau']$ du type τ par le type τ' dans l'expression e. Cela servira à propager les contraintes de tailles *à l'exécution* à chaque application de fonction.

$$
\begin{array}{lcl}
\mathbf{type}(\texttt{true}) & \stackrel{\text{def}}{=} & \texttt{bool} \\
\mathbf{type}(\texttt{false}) & \stackrel{\text{def}}{=} & \texttt{bool} \\
\mathbf{type}(\texttt{()}) & \stackrel{\text{def}}{=} & \texttt{unit} \\
\mathbf{type}(\texttt{(}l\texttt{:int<}n\texttt{>)}) & \stackrel{\text{def}}{=} & \texttt{int<}n\texttt{>} \\
\mathbf{type}(\texttt{\{}c_1\texttt{,}\cdots c_n\texttt{\}}) & \stackrel{\text{def}}{=} & \mathring{\tau}\ \texttt{vect<}n\texttt{>} \quad \text{où } \mathring{\tau} \stackrel{\text{def}}{=} \mathbf{type}(c_1) \\
\mathbf{type}(\texttt{fun (}p\texttt{:}\tau\texttt{) : }\mathring{\tau}' \to e) & \stackrel{\text{def}}{=} & \tau \xrightarrow{\texttt{'}d} \mathring{\tau}' \\
 & & \text{où } \texttt{'}d \text{ est une inconnue de durée fraîche} \\
\mathbf{type}(\texttt{fix } f\ \texttt{(fun (}p\texttt{:}\tau\texttt{) : }\mathring{\tau}' \to e\texttt{)}) & \stackrel{\text{def}}{=} & \tau \xrightarrow{\mathbf{1}} \mathring{\tau}' \\
 & & \\
e[\tau \mapsto \tau] & \stackrel{\text{def}}{=} & e \\
e[\texttt{int< 'N >} \mapsto \texttt{int<}n\texttt{>}] & \stackrel{\text{def}}{=} & e[\texttt{'}N := n] \\
e[\mathring{\tau}\ \texttt{vect< 'N >} \mapsto \mathring{\tau}'\ \texttt{vect<}n\texttt{>}] & \stackrel{\text{def}}{=} & e[\texttt{'}N := n][\mathring{\tau} \mapsto \mathring{\tau}'] \\
e[(\tau_1\ \texttt{*}\ \tau_2) \mapsto (\tau_1'\ \texttt{*}\ \tau_2')] & \stackrel{\text{def}}{=} & e[\tau_1 \mapsto \tau_1'][\tau_2 \mapsto \tau_2'] \\
e[(\tau_1 \xrightarrow{\delta} \mathring{\tau}_2) \mapsto (\tau_1' \xrightarrow{\delta'} \mathring{\tau}_2')] & \stackrel{\text{def}}{=} & e[\tau_1 \mapsto \tau_1'][\mathring{\tau}_2 \mapsto \mathring{\tau}_2']
\end{array}
$$

FIGURE 4.2 Accès dynamique aux types des valeurs et instantiation d'inconnues de tailles

Cette section a présenté le système de types d'ECLAT, qui intègre une analyse de temps d'exécution pour garantir par typage que le programme produit toujours instantanément des sorties en fonction des entrées. Cette analyse va de paire avec la sémantique synchrone d'ECLAT, qui spécifie opérationnellement la progression des calculs au cours du temps. Cette sémantique suppose que les programmes soient bien typés ; les programmes mal typés sont rejetés statiquement.

4.2 Sémantique synchrone

Le comportement fonctionnel et temporel des programmes Eclat est défini par une sémantique opérationnelle dite *comportementale*. C'est un style de sémantique que l'on retrouve à la fois dans les langages synchrones orientés flot de contrôle [29] et ceux orientés flot de données [45]. Habituellement, dans sa forme la plus générale, elle se modélise par un relation de réduction $e/\mu \xhookrightarrow[v]{v'} e'/\mu'$ qui signifie : "l'expression e avec la mémoire μ et l'entrée v retourne instantanément la sortie v' et se réduit en l'expression e' dans la mémoire μ'. L'exécution du programme est alors une suite de réductions (une réduction à chaque *tick*) comme suit :

$$e_1/\mu_1 \xhookrightarrow[v_1]{v'_1} e_2/\mu_2 \xhookrightarrow[v_2]{v'_2} \cdots\ e_n/\mu_n \xhookrightarrow[v_n]{v'_n} \cdots$$

Le flot de sorties $v'_1 \bullet v'_2 \bullet\ \cdots\ v'_n \bullet\ \cdots$ découle des étapes de calcul $e_1\ e_2\ \cdots\ e_n \cdots$ et du flot d'entrées $v_1 \bullet v_2 \bullet\ \cdots\ v_n \bullet \cdots$. En Eclat, toutefois, les calculs n'interagissent pas avec l'extérieur ; la suite de réductions (une à chaque *tick*) est de la forme :

$$f\,(v)/\mu \hookrightarrow e_1/\mu_1 \hookrightarrow \cdots\ e_n/\mu_n \hookrightarrow \underbrace{v'/\mu'\ \cdots\ v'/\mu'}_{\text{n'évolue plus}}\ \cdots$$

Autrement dit, la fonction f appliquée à l'entrée v se réduit jusqu'à atteindre une valeur v'. En particulier, si la fonction f est instantanée, alors pour toute entrée v, $f\,(v)/\mu \hookrightarrow v'/\mu'$: la fonction f appliquée à v dans l'état mémoire μ retourne instantanément la sortie v' et le nouvel état mémoire μ'. La fonction f peut donc s'appliquer point à point sur les éléments du flot d'entrée et retourner le flot de sorties.

4.2.1 Ensemble des valeurs

L'ensemble $\mathbb{V}$ des valeurs Eclat est défini sur la Figure 4.3. Il est présenté comme un sous-ensemble du langage d'expressions ; ce choix est motivé par le style de sémantique opérationnelle utilisé à la section 4.2. Une valeur est soit une constante, soit une paire de valeurs, soit une valeur fonctionnelle (y compris les opérateurs).

valeur	v	::=
constante		c
paire		$\mid$ `(`v`,`v`)`
opérateur		$\mid$ op
fonction		$\mid$ `fun (`p`:`τ`) :` $\mathring{\tau}' \to e$
fonction récursive		$\mid$ `fix` f `(fun (`p`:`τ`) :` $\mathring{\tau}' \to e$`)`

Figure 4.3 Ensemble des valeurs Eclat

4.2.2 Jugements d'évaluation et de réduction

La sémantique comportementale d'ECLAT est définie sur deux niveaux syntaxiques : celui des *programmes* – de la forme `let` f = `(fun` $(p{:}\tau)$ `:` $\mathring{\tau}' \to e$`)` comme présenté figure 3.1 – et celui des *expressions*.

Le jugement $\mu \vdash \pi(v) \Rightarrow v', \mu'$ signifie "dans l'état mémoire μ, le programme π avec l'entrée v retourne instantanément la sortie v' dans l'état mémoire μ'".

On appelle configuration une paire e/μ où e est une expression et μ un état mémoire associé. Le jugement $e/\mu \hookrightarrow e'/\mu'$ signifie : "la configuration e/μ se réduit en e'/μ' instantanément (en zéro cycle)". Les valeurs sont des expressions. Le processus d'évaluation est modélisé comme une suite de réductions :

$$e/\mu \hookrightarrow e_1/\mu_1 \hookrightarrow \ \cdots \ \hookrightarrow e_n/\mu_n$$

telle que e_n est une valeur.

4.2.3 Notations

La mémoire μ est une fonction partielle des emplacements (notés ℓ) vers des valeurs. On note $\mu[\ell \mapsto v]$ la mise à jour de la mémoire μ avec la valeur v à l'emplacement ℓ. L'expression à l'emplacement ℓ dans la mémoire μ est notée $\mu(\ell)$. Similairement aux environnements de typage, elle est définie par $\mu[\ell \mapsto v](\ell) = v$ et $\mu[\ell' \mapsto v](\ell) = \mu(\ell)$ si $\ell \neq \ell'$.

Dans ce chapitre, la mémoire est modifiée de manière implicite par les constructions `reg` et `exec` [3]. Pour modéliser cela dans la sémantique opérationnelle, nous annotons ces constructions avec une étiquette unique ℓ (qui représente un emplacement en mémoire), c'est-à-dire : `reg`$_\ell$ `(fun` $p \to e$`) init` e_0 et `exec`$_\ell$ e `default` e_0 `reset` e_1.

Ces annotations ont été omises dans la définition de la syntaxe d'ECLAT (figure 3.1) pour deux raisons. D'une part, ce sont des étiquettes fraîches qui n'apparaissent pas dans la syntaxe concrète du langage. D'autre part, elles sont spécifiques à ce type de sémantique (qui décrit opérationnellement l'ordonnancement des calculs au cours de l'évaluation) et n'ont pas d'incidence sur le typage des programmes.

On note $e[x \mapsto v]$ la substitution sans capture du nom x par la valeur v dans l'expression e. Elle sera définie de façon non-standard à la section 4.3 ; le but est de garantir que chaque construction `reg`$_\ell$ ou `exec`$_\ell$ du programme est rattachée à un emplacement ℓ qui *reste* unique au cours de la réduction.

Par extension, la substitution $e[p \mapsto v]$ d'un motif p par une une expression v dans l'expression e est définie par :

$$\begin{array}{lcll} e[\texttt{()} \mapsto \texttt{()}] & \stackrel{\text{def}}{=} & e & \\ e[x \mapsto v] & \stackrel{\text{def}}{=} & e[x \mapsto v] & \text{(définie à la section 4.3)} \\ e[\texttt{(}p\texttt{,}p'\texttt{)} \mapsto \texttt{(}v_1\texttt{,}v_2\texttt{)}] & \stackrel{\text{def}}{=} & e[p \mapsto v_1][p' \mapsto v_2] & \end{array}$$

3. Les accès mémoire explicites, dans un style impératif, seront ajoutés au chapitre 6.

Dans le cas $e[(p,p') \mapsto (v_1,v_2)]$, on suppose que v_1 et v_2 sont des valeurs closes[4] pour éviter toute capture d'une variable de v_1 apparaissant dans p'.

On note **vars-of-pat**(p) l'ensemble des noms introduits dans un motif p :

$$\begin{aligned}
\textbf{vars-of-pat}(()) &\stackrel{\text{def}}{=} \emptyset \\
\textbf{vars-of-pat}(x) &\stackrel{\text{def}}{=} \{x\} \\
\textbf{vars-of-pat}((p,p')) &\stackrel{\text{def}}{=} \textbf{vars-of-pat}(p) \cup \textbf{vars-of-pat}(p')
\end{aligned}$$

4.2.4 Réduction des programmes Eclat

Le jugement d'évaluation des programmes est modélisé par la règle Eval-Prog ci-dessous. Celle-ci spécifie que le programme `let` $f =$ `fun` $(p{:}\tau)$: $\mathring{\tau}' \to e$ appelé avec la valeur d'entrée v s'évalue instantanément en v' si l'application `(fun` $p \to e$`)` v (qui est une expression) se réduit instantanément en v'.

Eval-Prog
$$\frac{\texttt{(fun } (p{:}\tau) \texttt{ : } \mathring{\tau}' \to e\texttt{)}\ v/\mu \hookrightarrow v'/\mu'}{\mu \vdash \texttt{(let } f = \texttt{fun } (p{:}\tau) \texttt{ : } \mathring{\tau}' \to e\texttt{)(}v\texttt{)} \Rightarrow v', \mu'}$$

Chaque programme π est la fonction de transition d'un système réactif avec un état interne μ (une mémoire). Le programme Eclat est appliqué à un flot d'entrées vs et produit un flot de sorties vs'. On modélise cela par le jugement $\mu \vdash \pi\ vs \Downarrow vs'$. Ce jugement est défini par la règle Interaction. Un flot est une suite infinie de valeurs $(v_0 \bullet v_1 \bullet \cdots v_n \bullet \cdots)$. Le programme transforme point à point le flot d'entrées en un flot de sorties.

Interaction
$$\frac{\mu \vdash \pi(v) \Rightarrow v'/\mu' \qquad \mu' \vdash \pi\ vs \Downarrow vs'}{\mu \vdash \pi\ (v\ \bullet vs) \Downarrow (v' \bullet vs')}$$

La règle Interaction est importante pour comprendre le fonctionnement des programmes Eclat. Cela justifie en particulier le comportement des constructions `reg` et `exec`, qui retourne instantanément une valeur et sauvegardent en mémoire la continuation du calcul. Le calcul sauvegardé en mémoire peut ainsi être repris lors de la prochaine exécution des blocs `reg` ou `exec`.

4.2.5 Noyau applicatif

Les valeurs se réduisent instantanément en elles-mêmes (règle Reduce-Val).

Reduce-Val
$$v/\mu \hookrightarrow v/\mu$$

4. Le jugement d'évaluation doit respecter cette contrainte. Il elle définit de manière à ce que les expressions réduites soient closes (car bien typées) en préservant le typage par réduction (mais cela n'a pas encore été démontré).

La réduction d'une conditionnelle **if** e **then** e_1 **else** e_2 réduit d'abord la condition. Si l'expression obtenue (autrement dit, le *réduit*) est la valeur `true`, alors la branche *then* est réduite immédiatement (règle REDUCE-IF-TRUE). Si le réduit est la valeur `false`, alors c'est la branche *else* qui est réduite immédiatement (règle REDUCE-IF-FALSE). Si le réduit n'est pas une valeur, alors, il remplace la condition e d'origine, ce qui donne **if** e' **then** e_1 **else** e_2 (règle REDUCE-IF-PAUSE).

$$\textsc{Reduce-If-True}\quad \frac{e/\mu \hookrightarrow \texttt{true}/\mu' \qquad e_1/\mu' \hookrightarrow e_1'/\mu''}{\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2/\mu \hookrightarrow e_1'/\mu''}$$

$$\textsc{Reduce-If-False}\quad \frac{e/\mu \hookrightarrow \texttt{false}/\mu' \qquad e_2/\mu' \hookrightarrow e_2'/\mu''}{\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2/\mu \hookrightarrow e_2'/\mu''}$$

$$\textsc{Reduce-If-Pause}\quad \frac{e/\mu \hookrightarrow e'/\mu' \qquad e' \notin \mathbb{V}}{\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2/\mu \hookrightarrow \textbf{if } e' \textbf{ then } e_1 \textbf{ else } e_2/\mu'}$$

Les composantes e_1 et e_2 d'une paire séquentielle (e_1, e_2) sont évaluées l'une après l'autre, de la gauche vers la droite : l'expression (e_1, e_2) est équivalente à **let** $x_1 = e_1$ **in** (**let** $x_2 = e_2$ **in** (x_1, x_2)) avec x_1 et x_2 des noms frais.

$$\textsc{Reduce-Pair-Val}\quad \frac{e_1/\mu \hookrightarrow v_1/\mu' \qquad e_2/\mu' \hookrightarrow e_2'/\mu''}{(e_1, e_2)/\mu \hookrightarrow (v_1, e_2')/\mu''}$$

$$\textsc{Reduce-Pair-Pause}\quad \frac{e_1/\mu \hookrightarrow e_1'/\mu' \qquad e_1' \notin \mathbb{V}}{(e_1, e_2)/\mu \hookrightarrow (e_1', e_2)/\mu'}$$

Les composantes des paires parallèles sont évaluées simultanément, mais de façon synchronisée avec les *ticks* d'horloge, et en forçant un ordre de réaction gauche-droite (qui n'engendre aucun surcoût à l'exécution et présente l'avantage d'être pleinement déterministe et prédictible). Si les deux réduits obtenus sont des valeurs v_1 et v_2, alors la paire (v_1, v_2) est immédiatement créée (règle REDUCE-PAR-VAL). Sinon, une paire parallèle est reformée avec les deux réduits (règle REDUCE-PAR-PAUSE). L'expression $(e_1 \| e_2)$ est équivalente à (e_1, e_2) si e_1 et e_2 sont instantanées (règles REDUCE-PAIR-VAL et REDUCE-PAR-VAL).

$$\textsc{Reduce-Par-Val}\quad \frac{e_1/\mu \hookrightarrow v_1/\mu' \qquad e_2/\mu' \hookrightarrow v_2/\mu''}{(e_1 \| e_2)/\mu \hookrightarrow (v_1, v_2)/\mu''}$$

$$\textsc{Reduce-Par-Pause}\quad \frac{e_1/\mu \hookrightarrow e_1'/\mu' \qquad e_2/\mu' \hookrightarrow e_2'/\mu'' \qquad e_1' \notin \mathbb{V} \text{ ou } e_2' \notin \mathbb{V}}{(e_1 \| e_2)/\mu \hookrightarrow (e_1' \| e_2')/\mu''}$$

La réduction d'une liaison locale **let** $p{:}\tau = e_1$ **in** e_2 réduit d'abord l'expression e_1. Si le réduit de e_1 est une valeur v, alors le motif p est substitué par v dans e_2 et la réduction se poursuit (règle REDUCE-LET-VAL). Si le réduit de e_1 est une expression e_1' qui n'est pas une valeur, alors e_1' remplace e_1 dans l'expression d'origine, ce qui donne **let** $p{:}\tau = e_1'$ **in** e_2 (règle REDUCE-LET-PAUSE).

REDUCE-LET-VAL
$$\frac{e_1/\mu \hookrightarrow v/\mu' \qquad e_2[p \mapsto v]/\mu' \hookrightarrow e'/\mu''}{\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2/\mu \hookrightarrow e'/\mu''}$$

REDUCE-LET-PAUSE
$$\frac{e_1/\mu \hookrightarrow e_1'/\mu' \qquad e_1' \notin \mathbb{V}}{\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2/\mu \hookrightarrow \texttt{let}\ p{:}\tau = e_1'\ \texttt{in}\ e_2/\mu'}$$

La réduction d'une application $(e_1\ e_2)$ réduit d'abord l'expression e_1 en position fonctionnelle puis l'argument e_2. L'évaluation de l'expression e_1 est instantanée (garantie par typage) ; c'est pourquoi il n'y a pas de règle pour le cas $e_1/\mu \hookrightarrow e_1'/\mu'$ et $e_1' \notin \mathbb{V}$.

Si le réduit de e_2 est une expression e_2' qui n'est pas une valeur, alors $(e_1\ e_2)$ se réduit en $(v_1\ e_2')$ (règle REDUCE-APP-PAUSE). Si au contraire l'argument e_2' est une valeur v_2, alors la valeur v_1 en position fonctionnelle s'applique à v_2.

Si v_1 est un opérateur op, alors l'application de op à l'argument v_2 est donnée par une fonction sémantique ϑ définie sur la figure 4.4 (sous-section 4.2.6), puis la réduction se poursuit. La fonction ϑ prend l'état mémoire courant et retourne un nouvel état mémoire ; cela sert à modéliser des effets de bord [5]. Les erreurs d'exécution (telles les divisions par zéro) sont modélisées sous forme de boucles infinies déroulées dans le temps via la construction `halt` e (dérivée de `fix`) qui se réduit toujours en `halt` e.

Si l'expression en position fonctionnelle se réduit en $(\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)$ et l'argument se réduit en une valeur v, alors les contraintes de tailles sont résolues et propagées dans le corps e en fonction de $\mathbf{type}(v)$, le motif p est substitué par l'argument v dans l'expression obtenue et la réduction se poursuit (règle REDUCE-APP-FUN).

Si l'expression en position fonctionnelle se réduit en $(\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e))$ et l'argument se réduit en une valeur v, alors les contraintes de tailles sont résolues et propagées dans le corps e en fonction de $\mathbf{type}(v)$, le nom f est substitué par sa définition, le motif p est substitué par l'argument v dans l'expression obtenue et la réduction est bloquée (règle REDUCE-APP-FIX). Pour modéliser cela, on introduit la notation $\widehat{e}$ (c'est-à-dire, l'expression e décorée avec un chapeau) inspirée de la sémantique opérationnelle d'ESTEREL [207].

REDUCE-APP-OP
$$\frac{\vartheta \text{ est définie figure 4.2.6} \qquad e_1/\mu \hookrightarrow op/\mu' \qquad e_2/\mu' \hookrightarrow v_2/\mu'' \qquad \vartheta(op, v_2, \mu'') \hookrightarrow e'/\mu_3}{e_1\ e_2/\mu \hookrightarrow e'/\mu_3}$$

REDUCE-APP-FUN
$$\frac{e_1/\mu \hookrightarrow (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)/\mu' \qquad e_2/\mu' \hookrightarrow v/\mu'' \qquad e[\tau \mapsto \mathbf{type}(v)][p \mapsto v]/\mu'' \hookrightarrow e'/\mu_3}{e_1\ e_2/\mu \hookrightarrow e'/\mu_3}$$

REDUCE-APP-PAUSE
$$\frac{e_1/\mu \hookrightarrow v_1/\mu' \qquad e_2/\mu' \hookrightarrow e_2'/\mu'' \qquad e_2' \notin \mathbb{V}}{e_1\ e_2/\mu \hookrightarrow v_1\ e_2'/\mu''}$$

REDUCE-APP-FIX
$$\frac{e_1/\mu \hookrightarrow \overbrace{(\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau) \to e))}^{w}/\mu' \qquad e_2/\mu' \hookrightarrow v/\mu'' \qquad e' \stackrel{\text{def}}{=} e[\tau \mapsto \mathbf{type}(v)][f \mapsto w][p \mapsto v]}{e_1\ e_2/\mu \hookrightarrow \widehat{e'}/\mu''}$$

L'expression $\widehat{e}$ est une expression non encore évaluée : pour tout expression e, l'expression $\widehat{e}$ n'est pas une valeur (et en particulier, pour toute valeur v, l'expression $\widehat{v}$ n'est pas une

5. Ce sera le cas au chapitre 6 lorsque nous ajouterons à ECLAT des tableaux modifiables en place.

valeur). Nous choisissons de modéliser la notation $\widehat{e}$ comme un sucre syntaxique équivalent à l'expression (`let () = () in` e). Si e se réduit en e', alors $\widehat{e}$ se réduit en e' (règle dérivée Reduce-Hat). La notation $\widehat{e}$ dans la règle permet d'insérer une pause à chaque appel de fonction récursive (règle Reduce-App-Fix), y compris si le corps de la fonction est une valeur. Par exemple, `let` $x =$ `5 in` $x+$`1` se réduit en `6`, mais `let` $x =$ `(fix` f `(fun () ->` `5)) () in` $x+$`1` se réduit en `let` $x =$ $\widehat{\texttt{5}}$ `in` $x+$`1`; il devra être réduit à nouveau pour obtenir la valeur `6`, d'où un temps d'exécution d'un cycle.

$$\textsc{Reduce-Hat}\ \frac{e/\mu \hookrightarrow e'/\mu'}{\widehat{e}\,/\mu \hookrightarrow e'/\mu'}$$

Exemple : l'expression suivant additionne les entiers `8` et `3` en quatre cycles (autrement dit, le résultat est obtenu instantanément lors du cinquième *tick*).

$$\underbrace{\texttt{(fix add (fun } (a,b) \texttt{ -> if b = 0 then } a \texttt{ else add(a+1,b-1)))}}_{\phi_{\mathsf{add}}}\ \texttt{(8,3)}$$

En effet :

1. étant donné un état mémoire μ, cette expression se réduit instantanément en :

$$\widehat{e_0}\,/\mu \text{ où } e_0 \stackrel{\text{def}}{=} \texttt{if 3 = 0 then 8 else } \phi_{\mathsf{add}}\texttt{(8+1,3-1)}$$

car :

$$\textsc{Reduce-App-Fix}\ \frac{\textsc{Reduce-Val}\ \dfrac{}{\phi_{\mathsf{add}}/\mu \hookrightarrow \phi_{\mathsf{add}}/\mu} \qquad \textsc{Reduce-Val}\ \dfrac{}{\texttt{(8,3)}/\mu \hookrightarrow \texttt{(8,3)}/\mu}}{\phi_{\mathsf{add}}\texttt{(8,3)}/\mu \hookrightarrow \widehat{e_0}\,/\mu}$$

2. Puis, au *tick* suivant, la configuration $\widehat{e_0}\,/\mu$ obtenue se réduit instantanément en :

$$\widehat{e_1}\,/\mu \text{ où } e_1 \stackrel{\text{def}}{=} \texttt{if 2 = 0 then 9 else } \phi_{\mathsf{add}}\texttt{(9+1,2-1)}$$

car :

$$\textsc{Reduce-App-Op}\ \textsc{Reduce-If-False}\ \frac{\vdots \quad \vdots \quad \vdots}{\texttt{0 = 3}/\mu \hookrightarrow \texttt{false}/\mu} \qquad \textsc{Reduce-App-Fix}\ \frac{\textsc{Reduce-Val}\ \dfrac{}{\phi_{\mathsf{add}}/\mu \hookrightarrow \phi_{\mathsf{add}}/\mu} \qquad \vdots}{\phi_{\mathsf{add}}\texttt{(8+1,3-1)}/\mu \hookrightarrow \widehat{e_0}\,/\mu}$$

$$\underbrace{\texttt{if 3 = 0 then 8 else } \phi_{\mathsf{add}}\texttt{(8+1,3-1)}}_{\widehat{e_0}}\,/\mu \hookrightarrow \widehat{e_1}\,/\mu$$

3. Puis, au *tick* suivant, la configuration $\widehat{e_1}\,/\mu$ obtenue se réduit instantanément en :

$$\widehat{e_2}\,/\mu \text{ où } e_2 \stackrel{\text{def}}{=} \texttt{if 1 = 0 then 10 else } \phi_{\mathsf{add}}\texttt{(10+1,1-1)}$$

4. Puis, au *tick* suivant, la configuration $\widehat{e_2}\,/\mu$ obtenue se réduit instantanément en :

$$\widehat{e_3}\,/\mu \text{ où } e_3 \stackrel{\text{def}}{=} \texttt{if 0 = 0 then 11 else } \phi_{\mathsf{add}}\texttt{(11+1,0-1)}$$

5. Puis, au *tick* suivant, la configuration $\widehat{e_3}/\mu$ se réduit instantanément en $\texttt{11}/\mu$, car :

$$\textsc{Reduce-App-Op}\ \textsc{Reduce-If-True}\ \frac{\dfrac{\vdots\quad\vdots\quad\vdots}{\texttt{0 = 0}/\mu \hookrightarrow \texttt{true}/\mu} \qquad \textsc{Reduce-Val}\ \dfrac{}{\texttt{11}/\mu \hookrightarrow \texttt{11}/\mu}}{\underbrace{\texttt{if 0 = 0 then 11 else } \phi_{\mathsf{add}}\texttt{(11+1,0-1)}}_{e_2}/\mu \hookrightarrow \texttt{11}/\mu}$$

Avant de présenter la réduction des constructions **reg** et **exec**, qui forment le noyau réactif d'ECLAT (sous-section 4.2.7), nous détaillons la comportement des opérateurs du langage.

4.2.6 Réduction des appels d'opérateurs

La fonction ϑ utilisée dans la règle REDUCE-APP-OP spécifie la réduction de chaque opérateur. Elle est définie sur la figure 4.4. Les opérateurs s'appliquent à des arguments déjà évalués (y compris les opérateurs logiques `&` et `or`). Le résultat n'est pas toujours une valeur ; par exemple, $\vartheta(\texttt{xor}, \texttt{(true,}v\texttt{)}, \mu)$ se réécrit en $\texttt{not(}v\texttt{)}$. À noter que nous aurions pu définir les opérateurs logiques avec des conditionnelles. Par exemple : $\vartheta(\texttt{not}, v, \mu) \stackrel{\text{def}}{=} \texttt{(if } v \texttt{ then false else true)}/\mu$ et $\vartheta(\texttt{xor}, (v_1\texttt{,}v_2), \mu) \stackrel{\text{def}}{=} \texttt{if } v_1 \texttt{ then not(}v_2\texttt{) else } v_2/\mu$. Nous avons préféré une présentation plus synthétique, sous forme de table de vérité, pour bien distinguer les opérateurs logiques (combinatoires) d'une part, et la *structure de contrôle* **if/then/else** d'autre part.

Les opérateurs arithmétiques s'appliquent à des entiers de même taille. Lorsque le résultat est un entier, il est de même taille que les entiers arguments. Les entiers (de type `int<`n`>`) sont signés avec une représentation n bits en complément à 2 dont un bit de signe. Les vecteurs sont fonctionnels : ils sont manipulés comme toute autre constante. En particulier, l'opérateur `vect_copy_with` crée une copie de vecteur argument où l'un de ces éléments (une constante) remplacé par une autre constante.
Deux cas d'erreurs peuvent se produire lors de la réduction des appels d'opérateurs (figure 4.4) :

1. les divisions par zéro ;
2. les indices hors des bornes de vecteurs ;

Dans les deux cas, la configuration e/μ courante est *gelée* : elle se réduit en **halt** e/μ. La construction dérivée **halt** est définie figure 3.5 par : $\texttt{(fix } f\ \texttt{(fun () } \rightarrow f\ e\texttt{))()}$ où f est un nom frais, modélisant ainsi une boucle infinie.

La réduction des appels d'opérateurs vise à garantir la sûreté, via :

— une propriété de progrès

— et une propriété de préservation du typage par réduction.

Le comportement des opérateurs vise à vérifier ces propriétés, y compris dans les deux cas d'erreurs listés ci-dessus. Pour autant, les programmes mal typés, doivent être rejetés avant l'exécution car ils risquent, justement, de ne pas pouvoir se réduire. Par exemple, le triplet $(\texttt{+}, \texttt{(true,false)}, \mu)$ n'appartient pas au domaine de ϑ, ce qui empêche l'application de la règle REDUCE-APP-OP ; c'est pourquoi cette expression est rejetée par le système de types.

$$\vartheta(\texttt{not},\texttt{true},\ \mu) \stackrel{\text{def}}{=} \texttt{false}/\mu$$
$$\vartheta(\texttt{not},\texttt{false},\mu) \stackrel{\text{def}}{=} \texttt{true}/\mu \qquad \vartheta(\texttt{xor},(\texttt{false},v),\mu) \stackrel{\text{def}}{=} v/\mu$$
$$\vartheta(\texttt{or},(\texttt{false},v),\mu) \stackrel{\text{def}}{=} v/\mu \qquad \vartheta(\texttt{xor},(\texttt{true},\ v),\mu) \stackrel{\text{def}}{=} \texttt{not}(v)/\mu$$
$$\vartheta(\texttt{or},(\texttt{true},\ v),\mu) \stackrel{\text{def}}{=} \texttt{true}/\mu \qquad \vartheta(\texttt{fst},(v_1,v_2),\mu) \stackrel{\text{def}}{=} v_1/\mu$$
$$\vartheta(\texttt{\&},(\texttt{false},v),\mu) \stackrel{\text{def}}{=} \texttt{false}/\mu \qquad \vartheta(\texttt{snd},(v_1,v_2),\mu) \stackrel{\text{def}}{=} v_2/\mu$$
$$\vartheta(\texttt{\&},(\texttt{true},\ v),\mu) \stackrel{\text{def}}{=} v/\mu$$

$$\vartheta(\texttt{+},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>})),\mu) \stackrel{\text{def}}{=} (l\texttt{:int<}n\texttt{>})/\mu \quad \text{où } l \stackrel{\text{def}}{=} (l_1+l_2) \mod 2^n$$

note : les entiers (de type `int<`n`>`) sont signés avec une représentation n bits en complément à 2 dont un bit de signe qui fait partie du littéral l.

$$\vartheta(\texttt{-},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>})),\mu) \stackrel{\text{def}}{=} (l\texttt{:int<}n\texttt{>})/\mu \quad \text{où } l \stackrel{\text{def}}{=} (l_1-l_2) \mod 2^n$$
$$\vartheta(\texttt{*},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>})),\mu) \stackrel{\text{def}}{=} (l\texttt{:int<}n\texttt{>})/\mu \quad \text{où } l \stackrel{\text{def}}{=} (l_1\times l_2) \mod 2^n$$
$$\vartheta(\texttt{/},((\ell_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>})),\mu) \stackrel{\text{def}}{=} (l\texttt{:int<}n\texttt{>})/\mu \quad \text{où } l \stackrel{\text{def}}{=} (l_1 \div l_2) \mod 2^n$$
si $l_2 \neq 0$
$$\vartheta(\texttt{mod},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>})),\mu) \stackrel{\text{def}}{=} (l\texttt{:int<}n\texttt{>})/\mu \quad \text{où } l \stackrel{\text{def}}{=} (l_1\ \%\ l_2) \mod 2^n$$
si $l_2 \neq 0$

$$\vartheta(\texttt{<},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>}))) \stackrel{\text{def}}{=} l/\mu \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 < l_2 \text{ sinon } \texttt{false}$$
$$\vartheta(\texttt{>},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>}))) \stackrel{\text{def}}{=} l/\mu \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 > l_2 \text{ sinon } \texttt{false}$$
$$\vartheta(\texttt{=},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>}))) \stackrel{\text{def}}{=} l/\mu \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 = l_2 \text{ sinon } \texttt{false}$$
$$\vartheta(\texttt{<=},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>}))) \stackrel{\text{def}}{=} l/\mu \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 \leq l_2 \text{ sinon } \texttt{false}$$
$$\vartheta(\texttt{>=},((l_1\texttt{:int<}n\texttt{>}),(l_2\texttt{:int<}n\texttt{>}))) \stackrel{\text{def}}{=} l/\mu \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 \geq l_2 \text{ sinon } \texttt{false}$$

$$\vartheta(\texttt{resize_int<}n\texttt{>},(l\texttt{:int<}m\texttt{>}),\mu) \stackrel{\text{def}}{=} (l'\texttt{:int<}n\texttt{>}) \text{ où } l' \stackrel{\text{def}}{=} l \mod 2^n$$

$$\vartheta(\texttt{vect_create<}n\texttt{>},c,\mu) \stackrel{\text{def}}{=} \{\underbrace{c,\cdots c}_{n \text{ fois}}\}/\mu$$

note : $n > 0$ (à vérifier statiquement)

$$\vartheta(\texttt{vect_size},\{v_1,\cdots v_n\},\mu) \stackrel{\text{def}}{=} (n\texttt{:int<16>})/\mu$$
$$\vartheta(\texttt{vect_nth},(\{v_0,\cdots v_{n-1}\},i),\mu) \stackrel{\text{def}}{=} v_i/\mu \quad \text{si } i \in \{0,\cdots n-1\}$$
$$\vartheta(\texttt{vect_copy_with},(v,i,c),\mu) \stackrel{\text{def}}{=} v'/\mu \text{ où } v \stackrel{\text{def}}{=} \{v_0,\cdots v_{i-1},v_i,\ v_{i+1}\cdots v_{n-1}\}$$
et $v' \stackrel{\text{def}}{=} \{v_0,\cdots v_{i-1},c,\ v_{i+1}\cdots v_{n-1}\}$
si $i \in \{0,\cdots n-1\}$

Division par zéro

$$\vartheta(op,\overbrace{(v,((0\texttt{:int<}n\texttt{>})))}^{c},\mu) \stackrel{\text{def}}{=} \texttt{halt}(op\ c)/\mu \quad \text{où } op \in \{\texttt{/},\texttt{mod}\}$$

Indice hors des bornes

$$\vartheta(\texttt{vect_nth},\overbrace{(\{v_0,\cdots v_{n-1}\},i)}^{args},\mu) \stackrel{\text{def}}{=} \texttt{halt (vect_nth } args)/\mu \quad \text{si } i \notin \{0,\cdots n-1\}$$
$$\vartheta(\texttt{vect_copy_with},(v,i,c),\mu) \stackrel{\text{def}}{=} \texttt{halt (vect_copy_with } (v,i,c))/\mu$$
où $v \stackrel{\text{def}}{=} \{v_0,\cdots v_{n-1}\}$
si $i \notin \{0,\cdots n-1\}$

FIGURE 4.4 Sémantique des opérateurs

4.2.7 Noyau réactif

Lors de la première réduction de $\texttt{reg}_\ell\ \texttt{(fun}\ p \to e\texttt{)}\ \texttt{init}\ e_0$, l'emplacement mémoire ℓ n'est associé à aucune valeur en mémoire (ce qui est noté $\ell \notin \mathsf{dom}(\mu)$). Le système initialise alors l'emplacement mémoire ℓ en y rangeant la valeur de l'expression instantanée e_0, puis il évalue à nouveau $\texttt{reg}_\ell\ \texttt{(fun}\ p \to e\texttt{)}\ \texttt{init}\ e_0$ dans le même cycle (règle Reduce-Reg-Init).

Une fois que l'emplacement mémoire ℓ est initialisé et contient une valeur v, chaque réduction de l'expression $\texttt{reg}_\ell\ \texttt{(fun}\ p \to e\texttt{)}\ \texttt{init}\ e_0$ consiste à mettre à jour le registre en appliquant la fonction de mise à jour $\texttt{(fun}\ p \to e\texttt{)}$ à la valeur courante v. La valeur v' obtenue est immédiatement rangée en mémoire à l'emplacement ℓ associé et instantanément retournée (règle Reduce-Reg-Step).

Reduce-Reg-Step

$$\frac{\mu(\ell) = v \qquad \texttt{(fun (}p\texttt{:}\mathring{\tau}\texttt{) : }\mathring{\tau} \to e\texttt{)}\ v/\mu \hookrightarrow v'/\mu'}{\texttt{reg}_\ell\ \texttt{(fun (}p\texttt{:}\mathring{\tau}\texttt{) : }\mathring{\tau} \to e\texttt{)}\ \texttt{init}\ e_0/\mu \hookrightarrow v'/\mu'[\ell \mapsto v']}$$

Reduce-Reg-Init

$$\frac{\ell \notin \mathsf{dom}(\mu) \qquad e_0/\mu \hookrightarrow v/\mu' \qquad \texttt{reg}_\ell\ \texttt{(fun (}p\texttt{:}\mathring{\tau}\texttt{) : }\mathring{\tau} \to e\texttt{)}\ \texttt{init}\ e_0/\mu'[\ell \mapsto v] \hookrightarrow v'/\mu''}{\texttt{reg}_\ell\ \texttt{(fun (}p\texttt{:}\mathring{\tau}\texttt{) : }\mathring{\tau} \to e\texttt{)}\ \texttt{init}\ e_0/\mu \hookrightarrow v'/\mu''}$$

Comme pour un bloc `reg`, lors de la première réduction de $\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1$, l'emplacement ℓ n'est associé à aucune valeur en mémoire ; c'est pourquoi la mémoire est initialisée en y chargeant le calcul e à exécuter pas à pas ; puis l'expression $\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1$ est immédiatement ré-évaluée (règle Reduce-Reg-Init).

Une fois que l'emplacement mémoire ℓ est initialisé et contient la continuation e_ℓ du calcul, chaque réduction de l'expression $\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1$ d'origine réduit e_ℓ. Si l'expression e' obtenue est une valeur v_ℓ, la valeur $\texttt{(}v_\ell\texttt{,true)}$ est retournée et l'emplacement ℓ est "déchargé" de la mémoire résultante μ'' via l'opération $\mu''[\ell \mapsto \mathsf{none}]$ (règle Reduce-Exec-Val). De cette manière, à la prochaine exécution du bloc `exec`, le calcul à l'emplacement ℓ n'est plus en mémoire (car $\mu(\ell) = \mathsf{none}$) et la règle Reduce-Reg-Init s'applique donc, ce qui a pour effet de recharger le calcul d'origine en mémoire. Sinon (si e' n'est pas une valeur), la continuation $\texttt{(fun (():unit)} \to e'\texttt{)}$ est rechargée en mémoire à l'emplacement ℓ, l'expression par défaut e_0 (qui est instantanée) est évaluée en v_0 et la valeur $\texttt{(}v_0\texttt{,false)}$ est retournée. Enfin, dans le cas où l'expression booléenne e_1 dans la clause `reset` s'évalue en `true`, la continuation $\texttt{(fun (():unit)} \to e\texttt{)}$ (où e est l'expression d'origine dans le `exec`) est rechargée en mémoire à l'emplacement ℓ (elle écrase la continuation courante e_ℓ) et l'expression $\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1$ redémarre dans le même cycle (règle Reduce-Exec-Reset).

REDUCE-EXEC-INIT

$$\frac{\ell \notin \mathsf{dom}(\mu) \text{ ou } \mu(\ell) = \mathsf{none} \qquad \texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu'[\ell \mapsto (\texttt{fun}\ (()\texttt{:unit}) \to e)] \hookrightarrow v/\mu'}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu \hookrightarrow v/\mu'}$$

REDUCE-EXEC-VAL

$$\frac{\mu(\ell) = (\texttt{fun}\ (()\texttt{:unit}) \to e_\ell) \qquad e_1/\mu \hookrightarrow \texttt{false}/\mu' \qquad e_\ell/\mu' \hookrightarrow v_\ell/\mu''}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu \hookrightarrow (v_\ell\texttt{,true})/\mu''[\ell \mapsto \mathsf{none}]}$$

REDUCE-EXEC-DEFAULT

$$\frac{\mu(\ell) = (\texttt{fun}\ (()\texttt{:unit}) \to e_\ell) \qquad e_1/\mu \hookrightarrow \texttt{false}/\mu' \qquad e_\ell/\mu \hookrightarrow e'/\mu'' \qquad e' \notin \mathbb{V} \qquad e_0/\mu'' \hookrightarrow v_0/\mu_1}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu \hookrightarrow (v_0\texttt{,false})/\mu_1[\ell \mapsto (\texttt{fun}\ (()\texttt{:unit}) \to e')]}$$

REDUCE-EXEC-RESET

$$\frac{e_1/\mu \hookrightarrow \texttt{true}/\mu' \qquad \texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ \texttt{false}/\mu'[\ell \mapsto (\texttt{fun}\ (()\texttt{:unit}) \to e)] \hookrightarrow e_3/\mu''}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu \hookrightarrow e_3/\mu''}$$

Remarque : L'environnement d'exécution est figé à chaque démarrage du calcul. En effet, les lieurs (`fun`, `fix`, `let/in` et `reg`) se réduisent en appliquant la fonction de substitution sans capture $e[p \mapsto v]$ qui remplace chaque nom introduit par sa valeur. Or, le système de types d'ECLAT vise à garantir que seules les expressions closes sont effectivement réduites [6] Par conséquent, à chaque exécution d'un bloc ($\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1$), les sous-expressions e, e_0 et e_1 sont closes. La continuation ($\texttt{fun}\ (()\texttt{:unit}) \to e$) chargée en mémoire (règle REDUCE-EXEC-INIT) est donc spécialisée pour les entrées au démarrage du calcul.

Par exemple, la fonction locale `delay` définie ci-dessous retourne, un cycle plus tard, l'ancienne valeur de `x` et non sa valeur courante. La fonction `delay` a un temps d'exécution d'un cycle et elle est appelée une fois tous les deux cycles [7].

```
1  let main x =
2    let (o,rdy) =
3      exec
4        (* le calcul démarre aux ticks 0, 2, 4, 6 etc. *)
5        let rec delay () =
6          (* entrée dans le corps de la fonction aux ticks 1, 3, 5, 7 etc. *)
7          x
8        in delay()
9     default 42
10   in o ;;
```

6. C'est une propriété voulue (mais non démontrée) de la sémantique d'ECLAT : la préservation du typage par réduction (*subject reduction*).

7. En effet, `exec` redémarre le calcul après un cycle à chaque fois qu'il se termine.

Exemple : à la fin de la sous-section 4.2.5, nous avons montré que la configuration :

$$\phi_{\mathsf{add}}\texttt{(8,3)} \text{ où } \phi_{\mathsf{add}} \stackrel{\text{def}}{=} \texttt{(fix add (fun } (a,b) \rightarrow \texttt{if b = 0 then } a \texttt{ else add(a+1,b-1)))}/\mu$$

s'évalue comme suit :

$$\phi_{\mathsf{add}}\texttt{(8,3)}/\mu \hookrightarrow e_0/\mu_0 \hookrightarrow e_1/\mu_1 \hookrightarrow e_2/\mu_2 \hookrightarrow e_3/\mu_3 \hookrightarrow \texttt{11}/\mu_4$$

où les mémoires μ_i sont égales à μ car ϕ_{add} et sans effet de bord.
Considérons maintenant le programme f suivant :

$$\texttt{let } f \texttt{ () = (exec}_\ell\ \phi_{\mathsf{add}}\texttt{(8,3) init 42 reset false)}$$

1. étant donnée une mémoire initiale μ telle que $\ell \notin \mathsf{dom}(\mu)$, la réduction instantanée du programme f avec l'entrée `()` et la mémoire μ produit la configuration $\texttt{(42,false)}/\mu_0$ où $\mu_0 \stackrel{\text{def}}{=} \mu[\ell \mapsto \texttt{(fun (():unit)} \rightarrow e_0\texttt{)}]$, car :

$$\textsc{Reduce-App-Fun}\ \frac{\textsc{Reduce-Exec-Init}\ \dfrac{\textsc{Reduce-Exec-Default}\ \dfrac{\textsc{Reduce-App-Fix}\ \dfrac{\vdots \quad \vdots}{\phi_{\mathsf{add}}\texttt{(8,3)}/\mu^\star \hookrightarrow e_0/\mu^\star} \qquad \mu^\star(\ell) = \kappa_0 \quad \vdots \quad \vdots}{e^\star/\mu^\star \hookrightarrow \texttt{(42,false)}/\underbrace{\mu^\star[\ell \mapsto e_0]}_{\mu_0}} \qquad \ell \notin \mathsf{dom}(\mu) \quad \text{où } \mu^\star \stackrel{\text{def}}{=} \mu[\ell \mapsto (\underbrace{\texttt{fun (():unit)} \rightarrow \phi_{\mathsf{add}}\texttt{(8,3)}}_{\kappa_0}\texttt{)}]}{e^\star[\texttt{()} \mapsto \texttt{()}]/\mu \hookrightarrow \texttt{(42,false)}/\mu_0} \qquad \vdots \quad \vdots}{\texttt{(fun () } \rightarrow \underbrace{\texttt{(exec}_\ell\ \phi_{\mathsf{add}}\texttt{(8,3) init 42 reset false)}}_{e^\star}\texttt{) ()}/\mu \hookrightarrow \texttt{(42,false)}/\mu_0}$$

2. au *tick* suivant, la réduction du programme f avec l'entrée `()` et la mémoire μ_0 produit la configuration $\texttt{(42,false)}/\mu_1$ où $\mu_1 \stackrel{\text{def}}{=} \mu[\ell \mapsto \texttt{(fun (():unit)} \rightarrow e_1\texttt{)}]$.
3. au *tick* suivant, la réduction du programme f avec l'entrée `()` et la mémoire μ_1 produit la configuration $\texttt{(42,false)}/\mu_2$ où $\mu_2 \stackrel{\text{def}}{=} \mu[\ell \mapsto \texttt{(fun (():unit)} \rightarrow e_2\texttt{)}]$.
4. au *tick* suivant, la réduction du programme f avec l'entrée `()` et la mémoire μ_2 produit la configuration $\texttt{(42,false)}/\mu_3$ où $\mu_3 \stackrel{\text{def}}{=} \mu[\ell \mapsto \texttt{(fun (():unit)} \rightarrow e_3\texttt{)}]$.
5. au *tick* suivant, la réduction du programme f avec l'entrée `()` et la mémoire μ_2 produit la configuration $\texttt{(11,true)}/\mu_3$ où $\mu_3 \stackrel{\text{def}}{=} \mu[\ell \mapsto \mathsf{none}]$.
6. au *tick* suivant, la réduction du programme f avec l'entrée `()` et la mémoire μ produit la configuration $\texttt{(42,false)}/\mu_0$ où $\mu_0 \stackrel{\text{def}}{=} \mu[\ell \mapsto \texttt{(fun (():unit)} \rightarrow e_0\texttt{)}]$.

4.2.8 Fonctions extérieures

Dans le style de sémantique adopté ici (et qui se base sur la notion de substitution), seules les expressions closes peuvent être réduites. Les expressions closes sont des expressions qui ne contiennent pas de variables libres. Le corps de chaque définition de fonction est propagé au niveau des sites d'appel au cours de la réduction. Dans ce cadre, on suppose que les appels de fonctions extérieures (définies dans l'environnement de typage initial) ont été substituées par leur définition au préalable. Nous expliciterons cette substitution dans la version étendue d'ECLAT présentée en annexe B (règle EVAL-EXTERNAL-DECL). Nous avons ici fait le choix de modéliser les fonctions externes comme s'il s'agissait de fonctions ECLAT, compte tenu du fait que ces fonctions externes peuvent être des calculs longs et qu'elles doivent se comporter comme des fonctions ECLAT. En pratique, elles sont écrites dans le langage cible du compilateur ECLAT, à savoir VHDL (voir section 5.4).

Cette section a présenté la sémantique opérationnelle d'ECLAT en modélisant la réduction des constructions d'ECLAT par un ensemble de règles de réduction. Les règles REDUCE-LET-VAL (dont découle la sémantique des lieurs **fun**, **fix** et **reg**) utilise une substitution $e[p \mapsto v]$ qui est dite *sans capture*. Par exemple $x[x \mapsto v] \stackrel{\text{def}}{=} v$ mais $(\textbf{fun}\ x \to e)[x \mapsto v] \stackrel{\text{def}}{=} \textbf{fun}\ x \to e$ car le nom x n'apparaît pas libre dans $\textbf{fun}\ x \to e$. Nous avons défini la substitution $e[p \mapsto v]$ d'un motif p par une valeur v dans une expression e ; en particulier, $e[(p_1,p_2) \mapsto (v_1,v_2)] \stackrel{\text{def}}{=} e[p_1 \mapsto v_1][p_2 \mapsto v_2]$. Nous avons cependant caché une difficulté dans la définition de la fonction $e[x \mapsto v]$ et qui concerne la duplication des blocs **reg** et **exec** à l'exécution. Nous traitons ce point à la section suivante.

4.3 Substitution sans capture et mémoire locale

Cette section définit la fonction de substitution sans capture $e[x \mapsto v]$ telle qu'elle doit être mis en œuvre dans la sémantique comportementale d'ECLAT (section 4.2). Cette substitution est non standard. Elle renomme *à la volée* les emplacements ℓ des blocs $\textbf{reg}_\ell$ et $\textbf{exec}_\ell$ dans la valeur v à instancier. Ce renommage est déterministe : il consiste à préfixer par le chemin jusqu'à chaque occurrence de x à remplacer dans l'arbre de syntaxe abstraite de l'expression e.

Considérons par exemple la fonction

$$\phi_{\mathsf{abro}} \stackrel{\text{def}}{=} (\textbf{fun}\ ((\texttt{a},\texttt{b}),\texttt{r}) \to (\texttt{await(a,r)}\ \texttt{\&}\ \texttt{await(b,r)}))$$

C'est une variante de ABRO[8] sans détection de front montant (edge) : "attendre A et attendre B, puis maintenir O jusqu'à l'arrivée de R". Elle est représentée sur la partie gauche de la figure 4.5. La variable await est libre dans ϕ_{abro}. Nous utilisons la fonction $\phi_{\mathsf{abro}}[\texttt{await} \mapsto v]$ pour remplacer chaque occurrence de await par une instance de v dans ϕ_{abro}. Nous obtenons l'expression représentée sur le côté droit de la figure 4.5. Chaque occurrence de await a été remplacée par une instance $\mathsf{prefix}(id, v)$ où id est le chemin depuis la racine dans l'arbre de syntaxe abstraite l'expression ϕ_{abro}.

8. la version d'origine a été présentée à la sous-section 3.5.3.

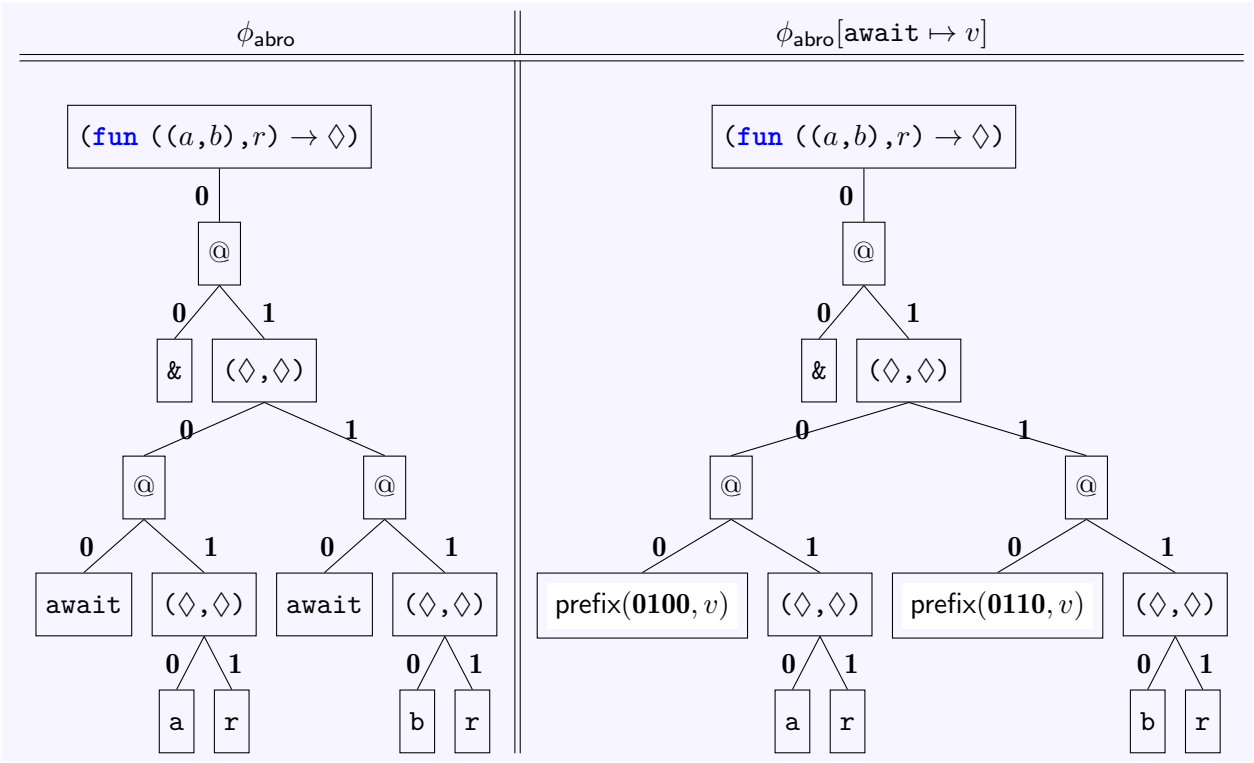


FIGURE 4.5 Substituer chaque occurrence de variable par une instance d'une valeur

Posons maintenant $v \stackrel{\text{def}}{=}$ `(fun` $i \to$ `reg`$_\ell$ `(fun` $s \to (s$ `or` $i)$ & `not(`r`)) init false)`. Alors :

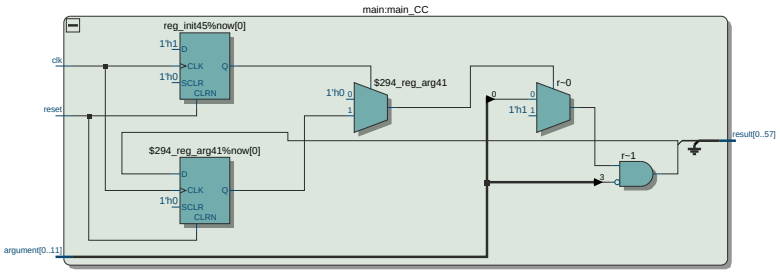
— prefix(**0100**, v) $\stackrel{\text{def}}{=}$ `(fun (i,r)` $\to$ `reg`$_{(\mathbf{0100}\cdot\ell)}$ `(fun` $s \to (s$ `or` $i)$ & `not(`r`)) init false)`

— prefix(**0110**, v) $\stackrel{\text{def}}{=}$ `(fun (i,r)` $\to$ `reg`$_{(\mathbf{0110}\cdot\ell)}$ `(fun` $s \to (s$ `or` $i)$ & `not(`r`)) init false)`

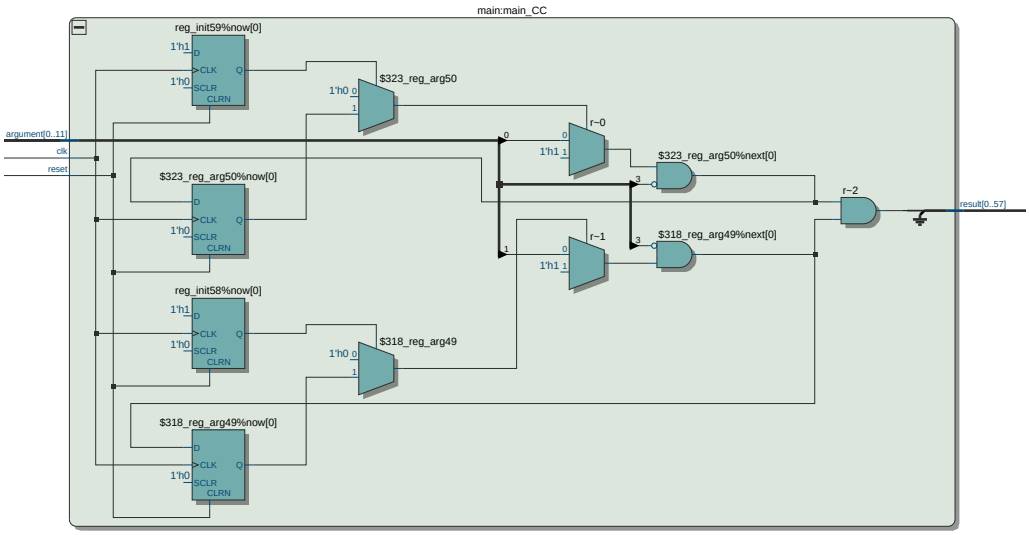
Ainsi, l'expression $\phi_{\mathsf{abro}}[\texttt{await} \mapsto v]$ obtenue est :

```
fun ((a,b),r) ->
  ( ((fun i -> reg_{0100·ℓ} (fun s -> (s or i) & not(r))) (a,r))
    &
    ((fun i -> reg_{0110·ℓ} (fun s -> (s or i) & not(r))) (b,r)) )
```

Chaque occurrence de la variable `await` dans `abro` a été remplacée par une instance renommée du circuit `await`. Le renommage suit la forme de l'arbre de syntaxe abstraite d'`abro`. De cette manière, à chaque exécution du programme, les registres sont dupliqués en renommant l'emplacement ℓ associé au bloc `reg`, tout en se souvenant de l'ancien emplacement renommé. Cela correspond à la fonctionnalité souhaitée au niveau RT (obtenue par compilation en VHDL et synthèse logique) : étant donné un circuit `await` (figure 4.6a), instancier deux fois `await` pour obtenir `abro` (figure 4.6b).



(a) Circuit `await` : "attendre A puis soutenir O jusqu'à l'arrivée de R"



(b) Variante d'`ABRO` : "attendre A et B puis soutenir O jusqu'à l'arrivée de R"

FIGURE 4.6 Exemple d'instanciation au niveau matériel

Chaque instance de `await` comporte deux registres de 1 bit avec une convention de nommage dans le code VHDL engendré :

— `reg_arg` pour stocker la valeur courante du bloc **`reg`**,

— et `reg_init` qui est un booléen initialement à `false` et passant à `true` lors de la première exécution du bloc **`reg`** ; cela sert à initialiser le bloc **`reg`**.

Maintenant que nous avons illustré le comportement de la fonction de substitution $e[x \mapsto v]$, nous pouvons en donner la définition (figure 4.7). Elle utilise deux fonctions auxiliaires :

— $e[x \overset{\ell}{\mapsto} v]$ qui collecte le chemin ℓ faisant office de préfixe,

— et $\mathsf{prefix}(\ell, v)$ qui renomme tous les emplacements dans v en les préfixant par ℓ.

La substitution $e[x \mapsto v]$ est définie par $e[x \overset{\varepsilon}{\mapsto} v]$ où ε est le préfixe vide tel que $\varepsilon \cdot \ell = \ell$. À mesure que la fonction $e[x \overset{\ell}{\mapsto} v]$ parcourt la structure de l'arbre de syntaxe de e, le préfixe ℓ s'allonge. La substitution $x[x \overset{\ell}{\mapsto} v]$ remplace l'occurrence de la variable x par une instance de la valeur v préfixée avec ℓ, c'est-à-dire $\mathsf{prefix}(\ell, v)$.

Les cas intéressants dans la définition de $\mathsf{prefix}(\ell, v)$ sont $\mathsf{prefix}(\ell, \mathtt{reg}_{\ell'}\ (\mathtt{fun}\ p \to e)\ \mathtt{init}\ e_0)$ et $\mathsf{prefix}(\ell, \mathtt{exec}_{\ell'}\ e\ \mathtt{default}\ e_0\ \mathtt{reset}\ e_2)$, qui assignent aux blocs **`reg`** et **`exec`** un nouvel

emplacement noté $\ell \cdot \ell'$. L'emplacement ℓ' correspond au nom unique identifiant le bloc `reg` ou `exec` ; cet identifiant est ajouté dans le code source par l'analyseur syntaxique. La définition complète de la fonction de substitution $e[x \mapsto v]$ est donnée en annexe (figure B.27).

$$
\begin{array}{rcl}
e[x \mapsto v] & \stackrel{\text{def}}{=} & e[x \overset{\varepsilon}{\mapsto} v] \\
\hline
c[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & c \\
x[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \mathsf{prefix}(\ell, v) \\
y[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & y \text{ si } y \neq x \\
(\texttt{fun } (p{:}\tau) : \mathring{\tau}' \to e)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{fun } (p{:}\tau) : \mathring{\tau}' \to e \text{ si } x \in \textbf{vars-of-pat}(p) \\
(\texttt{fun } (p{:}\tau) : \mathring{\tau}' \to e)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{fun } (p{:}\tau) : \mathring{\tau}' \to e[x \overset{(\ell\cdot 0)}{\longmapsto} v] \quad \text{si } x \notin \textbf{vars-of-pat}(p) \\
(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{if } e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v] \texttt{ then } e_2[x \overset{(\ell\cdot 1)}{\longmapsto} v] \texttt{ else } e_1[x \overset{(\ell\cdot 2)}{\longmapsto} v] \\
\hline
\mathsf{prefix}(\ell, c) & \stackrel{\text{def}}{=} & c \\
\mathsf{prefix}(\ell, x) & \stackrel{\text{def}}{=} & x \\
\mathsf{prefix}(\ell, (\texttt{fun } (p{:}\tau) : \mathring{\tau}' \to e)) & \stackrel{\text{def}}{=} & \texttt{fun } (p{:}\tau) : \mathring{\tau}' \to \mathsf{prefix}(\ell, e) \\
\mathsf{prefix}(\ell, \left[\begin{array}{l} \texttt{reg}_{\ell'}\ (\texttt{fun } (p{:}\mathring{\tau}) : \mathring{\tau} \to e) \\ \texttt{init } e_0 \end{array}\right]) & \stackrel{\text{def}}{=} & \left[\begin{array}{l} \texttt{reg}_{\ell\cdot\ell'}\ (\texttt{fun } (p{:}\mathring{\tau}) : \mathring{\tau} \to \mathsf{prefix}(\ell, e)) \\ \texttt{init } \mathsf{prefix}(\ell, e_0) \end{array}\right] \\
\mathsf{prefix}(\ell, \texttt{exec}_{\ell'}\ e \texttt{ default } e_0 \texttt{ reset } e_2) & \stackrel{\text{def}}{=} & \left[\begin{array}{l} \texttt{exec}_{\ell\cdot\ell'}\ \mathsf{prefix}(\ell, e) \texttt{ default } \mathsf{prefix}(\ell, e_0) \\ \texttt{reset } \mathsf{prefix}(\ell, e_2) \end{array}\right]
\end{array}
$$

FIGURE 4.7 Substitution sans-capture non-standard d'ECLAT (extrait)

Remarque : La solution proposée supporte l'imbrication de registres ainsi que la présence de blocs `exec` sous les registres ; cependant, les blocs `exec` et les registres ne sont pas autorisés [9] sous un `exec`. En effet, la technique de renommage présentée fait l'hypothèse que le chemin allant de la racine de l'arbre de syntaxe (AST) jusqu'à chaque nœud `reg` ou `exec` ne change pas entre deux cycles d'exécution. Or, dans le cas d'un `exec`, la réduction pas-à-pas est destructive : elle change la racine de l'AST au cours de la réduction.

Discussion : Une autre solution existe dans la littérature [144, 59]. Elle consiste à abstraire les noms de registres en introduisant une construction $\Lambda\ \ell.e$ (appelée *grand lambda*) à la manière de l'abstraction de type dans le λ-calcul polymorphe explicitement typé [145].

Nous avons choisi de ne pas suivre cette approche, et ce pour deux deux raisons. Tout d'abord, cela pose un problème de modularité ; par exemple, si une fonction f comporte un registre identifié avec une étiquette ℓ, et si une fonction `g` appelle deux fois `f`, alors `g` se retrouve paramétrée par deux étiquettes ; cela revient à rendre global dans le programme source l'ensemble des étiquettes. D'autre part, si les étiquettes sont explicites, deux registres peuvent avoir la même étiquette, ce que nous cherchons à interdire car cela a peu de sens en termes de circuit [10]. Des étiquettes implicites (comme nous le proposons) empêchent ce partage par construction et ne requiert *pas* de discipline de types particulière (comme du typage linéaire).

9. Cela est garanti par une analyse syntaxique sur la representation intermédiaire du compilateur.
10. Par exemple, le registre aurait alors plusieurs valeurs initiales et pourrait être mis à jour plusieurs fois par

4.4 Conclusion

Dans ce chapitre, nous avons montré non seulement *ce que calculent* les programmes ECLAT, mais aussi *dans quel ordre* et *à quel moment*. Cela a l'avantage de placer la question de l'efficacité au cœur du langage. Plus fondamentalement, l'objet du langage est de traiter formellement l'interaction avec l'environnement physique. C'est en effet une composante essentielle de la programmation FPGA, mais qui (en termes de sûreté de fonctionnement) est difficile à contrôler, prédire et garantir au cycle d'horloge près sans un modèle de programmation adapté.

De fait, les choix de conception d'ECLAT dépendent de choix d'implantation. Ce constat est vrai, non seulement pour certaines limitations du langage (comme la distinction entre types et types de base), mais surtout pour la justification des traits de programmation les plus saillants :

— expressions instantanées,

— récursion terminale synchronisée à chaque appel direct ou récursif,

— parallélisme synchrone,

— description de circuits séquentiels avec `reg`,

— et enfin `exec` pour faire le lien entre la vision flot de données (*fonctions instantanées*) et la vision flot de contrôle (*fonctions non instantanées*).

Ces traits de programmation font l'hypothèse que le compilateur *saura* découper l'exécution des calculs en des séquences de *ticks* d'horloge, comme spécifié par la sémantique du langage.

cycle, comme dans $\Lambda\ell \cdot \texttt{reg}_\ell$ `(fun` $x \to \texttt{reg}_\ell$ `(fun` $x \to x + 1$`) init 43) init 42` où les deux blocs `reg` ont la même étiquette ℓ.

5 Implantation d'Eclat sur FPGA

Les deux chapitres précédents étaient consacrés à la conception d'Eclat : un langage généraliste avec une sémantique synchrone pour faciliter la programmation de circuits FPGA.

Nous nous intéressons maintenant à l'implantation du langage. L'approche proposée consiste à compiler Eclat en VHDL, qui est un langage de description de matériel largement supporté par les outils de synthèse logique. Cela débouche sur une chaîne de compilation complète, automatique de bout en bout, pour reconfigurer des circuits FPGA depuis Eclat.

5.1 Interprétation d'Eclat en termes de circuits

Eclat comporte un petit nombre de constructions de programmation qui visent à capturer un grand nombre de cas d'usage, y compris la description de circuits (orientée flot de données) et la programmation de calculs (orientée flot de contrôle). De ce fait, il n'est pas étonnant que les constructions d'Eclat aient une interprétation naturelle en termes de circuits :

- chaque constante est un vecteur de bits (autrement dit, un entier de taille fixée) ;
- chaque paire séquentielle (e_1, e_2) groupe les valeurs produites par les circuits e_1 et e_2 (c'est une concaténation de vecteurs de bits) ;
- chaque paire parallèle $(e_1 \| e_2)$ juxtapose les circuits e_1 et e_2 et synchronise leurs résultats ;
- chaque liaison locale `let` $x = e_1$ `in` e_2 est une dépendance de donnée (autrement dit un fil) entre e_1 et e_2 ;
- chaque conditionnelle s'implante avec des multiplexeurs ;
- chaque construction `reg` est une machine de Mealy ;
- chaque opérateur prédéfini est une fonction combinatoire ;
- chaque appel de fonction non récursive est une instanciation de circuit (c'est-à-dire une duplication de celui-ci) ;
- chaque appel de fonction récursive terminale est une transition dans un *diagramme états-transitions* (*state diagram*) réalisé par une machine de Mealy ;
- enfin, chaque construction `exec` sert de passerelle entre flot de données et flot de contrôle.

Cette correspondance informelle entre les constructions de programmation d'ECLAT et leur implantation sous forme de circuits offre aux programmeuses et programmeurs un contrôle fin sur les détails architecturaux.

Compte tenu des possibilités d'imbrications disponibles en ECLAT ainsi que des contraintes imposées par la sémantique synchrone présentée au chapitre 4, la compilation d'ECLAT en VHDL requiert un effort important. Le problème est exposé à la section suivante pour motiver nos choix d'implantation et l'organisation du reste du chapitre.

5.2 Principes d'implantation

Il s'agit de compiler au niveau RT un langage, ECLAT, qui mélange calculs et interactions. Les calculs sont orientés flot de contrôle et retournent un résultat après une durée spécifiée en nombre cycles (et imposée) par la sémantique. Les interactions sont orientées flot de données et retournent instantanément des sorties en fonction des entrées.

5.2.1 Encodage des calculs sous forme d'automates

Concentrons nous d'abord sur l'implantation des calculs au niveau RT. Une approche classique consiste à les encoder sous forme de machines à états (aussi appelées *automates*) [114].

Dans la suite, on distingue la notion d'*automate* (vu comme une représentation ou une construction de programmation) de celle de *machine à états* (qui est la réalisation d'un automate au niveau RT, voir section 2.4) [186].

Un automate est formé d'un ensemble d'états (qui sont dénotés par la lettre q) et d'un ensemble de transitions reliant ces états. nous nous inspirons des formalises d'*automates avec variables* [149]. La syntaxe des transitions est définie sur la figure 5.1.

$$q \xrightarrow{\overline{cond}\quad \overline{(x_1, \cdots x_n)\ \texttt{:=}\ (e_1, \cdots e_n)}} q'$$

où
- q est un état source
- q' est un état destination
- *cond* est une expression booléenne combinatoire (et donc instantanée)
- $(x_1, \cdots x_n)$ est un n-uplet de variables
- $(e_1, \cdots e_n)$ est un n-uplet d'expressions combinatoires.

FIGURE 5.1 Définition du langage de transitions utilisé

La transition se lit comme suit :

> "à chaque *tick* d'horloge, si l'automate est dans l'état source q et si la condition *cond* est vraie, alors les variables $x_1, \cdots x_n$ reçoivent les valeurs des expressions

$e_1, \cdots e_n$ et l'automate passe dans l'état destination q'".

Illustrons l'encodage d'un calcul ECLAT sous forme d'automate. Nous considérons le programme qui est défini sur la figure 5.2. C'est une variante simplifiée du programme présenté sur la figure 3.8 ; nous reviendrons sur la version d'origine à la sous-section 5.2.2.

```
1 let fibonacci n =
2   let rec fib (i, a, b) =
3     if i = 0 then a
4     else fib(i - 1, b, a + b)
5   in fib(n, 0, 1) ;;
6
7 let main n =
8   let (o,rdy) = exec fibonacci(n) default 42
9   in o ;;
```

FIGURE 5.2 Un calcul ECLAT à compiler

Une implantation de ce programme sous forme d'automate est donnée sur la figure 5.3. L'automate a deux états : un état initial appelé `Idle` et un état de calcul `Fib` (qui correspond à la fonction récursive terminale `fib` définie aux lignes 2 à 5 de la figure 5.2). Chaque appel à `fib` dans le programme source correspond à une transition vers l'état destination `Fib`. Le passage de paramètre est encodé par des affectations de variables. L'appel direct à `fib` correspond à la transition centrale reliant l'état `Idle` à l'état `Fib` (avec la condition `true`).

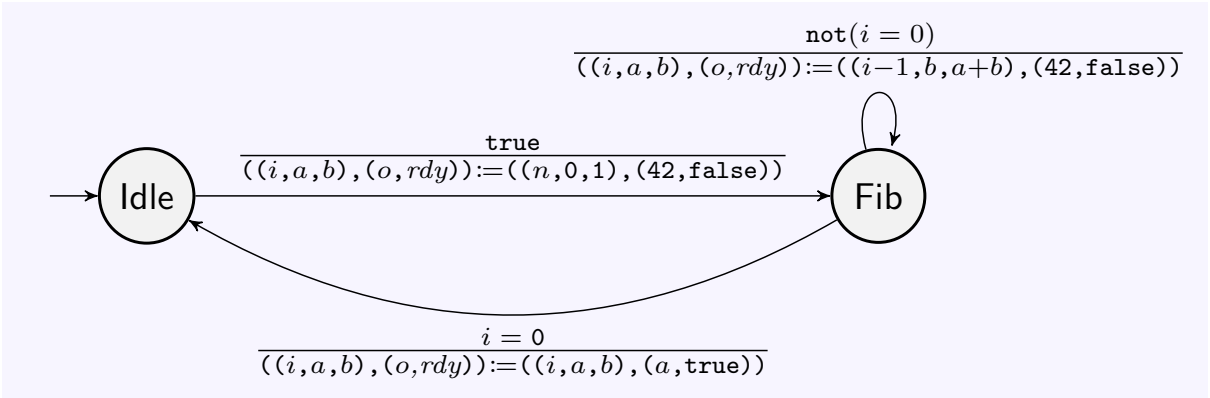


FIGURE 5.3 Encodage du programme de la figure 5.2 sous forme d'automate

Au cours du calcul, la sortie `o` de l'automate reçoit la valeur par défaut `42` et la sortie `rdy` reçoit `false` pour signifier que le calcul n'est pas terminé. Les deux branches de la conditionnelle du programme source correspondent à deux transitions mutuellement exclusives : l'une s'active si $i = 0$ (c'est le *cas de base*, à la ligne 3 de la figure 5.2) ; l'autre s'active si `not(`$i = 0$`)`, ce qui modélise la progression du calcul récursif terminal.

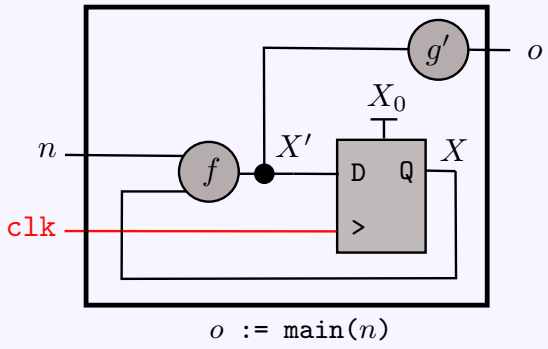
L'implantation d'ECLAT ne requiert pas de pile ni de tas [1] (ceci grâce au fait que toute définition de fonction récursive *non-terminale* est rejetée statiquement). Lorsque la récursion atteint le cas de base $i = \texttt{0}$, le calcul se termine : le résultat `a` est assigné à la sortie `o`, la sortie `rdy` prend la valeur `true` et l'automate repasse dans l'état initial `Idle`. Cela modélise bien le comportement temporel de la construction **`exec`** (à la ligne 8 de la figure 5.2) avec en particulier le redémarrage du calcul (qui correspond au retour vers l'état `Idle`) à chaque fois que celui-ci se termine.

Maintenant que nous avons encodé le programme ECLAT sous forme d'automate, il reste à le traduire en un circuit, c'est-à-dire, une machine de Mealy avec entrées physiques resynchronisées suivant les hypothèses énoncées à la section 2.5.

Comme cela a été présenté à la section 2.4, les machines de Mealy sont définies par deux fonctions combinatoires f et g telles que :

$$\begin{array}{rcl} X_{t+1} & \stackrel{\text{def}}{=} & f(X_t, i_t) \\ o_t & \stackrel{\text{def}}{=} & g(X_t, i_t) \end{array} \qquad \text{où} \begin{cases} i_t \text{ est l'entrée courante,} \\ X_t \text{ est l'état courant,} \\ X_{t+1} \text{ est l'état suivant,} \\ o_t \text{ est la sortie courante.} \end{cases}$$

La traduction d'un automate en une machine de Mealy consiste simplement à définir *par cas* la fonction f. Chaque cas correspond à une transition dans l'automate. De cette manière, à partir de l'automate de la figure 5.3, nous obtenons la machine de Mealy donnée figure 5.4.



$$\begin{array}{rcll} X_0 & \stackrel{\text{def}}{=} & \texttt{(Idle,(0,0,0),(0,0))} & \\ f\texttt{((Idle,(}i\texttt{,}a\texttt{,}b\texttt{),(}o\texttt{,}rdy\texttt{)),}n\texttt{)} & \stackrel{\text{def}}{=} & \texttt{(Fib,(}n\texttt{,0,1),(42,false))} & \\ f\texttt{((Fib,(}i\texttt{,}a\texttt{,}b\texttt{),(}o\texttt{,}rdy\texttt{)),}n\texttt{)} & \stackrel{\text{def}}{=} & \texttt{(Fib,(}i-1\texttt{,}b\texttt{,}a+b\texttt{),(42,false))} & \text{si } \texttt{not}(i = 0) \\ f\texttt{((Fib,(}i\texttt{,}a\texttt{,}b\texttt{),(}o\texttt{,}rdy\texttt{)),}n\texttt{)} & \stackrel{\text{def}}{=} & \texttt{(Idle,(}i\texttt{,}a\texttt{,}b\texttt{),(}a\texttt{,true))} & \text{si } i = 0 \\ g(X, n) & \stackrel{\text{def}}{=} & g'(f(X, n)) & \\ g'\texttt{((}q\texttt{,(}i\texttt{,}a\texttt{,}b\texttt{),(}o\texttt{,}rdy\texttt{)),}n\texttt{)} & \stackrel{\text{def}}{=} & o & \end{array}$$

FIGURE 5.4 Implantation du programme de la figure 5.2 sous forme de machine de Mealy

1. Mais nous pouvons programmer une pile et un tas *en* ECLAT (voir sections 7.5 et 7.9).

La fonction f calcule l'état suivant X' en fonction de l'état courant X et de l'entrée n. La fonction g de la machine de Mealy (qui calcule la sortie o en fonction de l'état suivant X' et de l'entrée n) est ici définie par la fonction combinatoire g' extrayant la sortie o dans l'état suivant X' calculé par f.

En résumé, cet exemple simple (`fibonacci`, figure 5.2) donne le principe d'implantation des calculs d'ECLAT. De façon générale, chaque calcul est encodé sous forme d'automate et réalisé par une machine de Mealy [2]. Lorsque le calcul est en cours d'exécution, la sortie contient une valeur par défaut fixée par la construction **exec** englobante. L'état de la machine de Mealy est un n-uplet qui comprend à la fois une *variable d'état* (codant le nom de la fonction récursive terminale à appeler au cycle suivant), les valeurs des paramètres de fonctions récursives terminales du programme et la sortie courante.

Fondamentalement, la compilation d'ECLAT encode le flot de contrôle en flot de données. C'est ce qui justifie (et rend possible) le mélange de calcul et d'interaction en ECLAT.

5.2.2 Combiner flot de données et flot de contrôle

Le calcul considéré sur la figure 5.2 était une simplification d'un programme présenté au chapitre 3. Nous avons montré comment encoder ce calcul simplifié sous forme d'automate et comment construire structurellement un circuit à partir de cet automate orienté flot de contrôle. Considérons maintenant le programme original (figure 3.8) redonné ci-dessous :

```
1 let main n =
2   if n < 0 then 0 else
3   let (o,rdy) = exec fibonacci(n) default 42
4   in o ;;
```

Ce programme fait progresser le calcul `fibonacci` uniquement lorsque l'entrée `n` est positive. L'encodage de ce programme sous forme d'automate peut consister à ajouter à l'automate précédent (figure 5.3) de nouvelles transitions pour modéliser tous les cas où l'entrée `n` est négative. L'automate résultant est donné sur la figure 5.5.

Cette méthode de compilation est inefficace : elle conduirait rapidement à faire exploser le nombre de transitions et dupliquer inutilement les fonctions combinatoires qui apparaissent dans les transitions.

Pour engendrer une implantation efficace, nous choisissons de travailler directement sur la représentation flot de données de l'automate de calcul (figure 5.4). Cette représentation est une machine de Mealy dont nous avons défini la fonction f et la fonction g (laquelle est dérivée de f et d'une fonction auxiliaire g').

2. C'est-à-dire un circuit séquentiel qui calcule instantanément sa sortie et un état suivant en fonction de son état courant et de son entrée à chaque cycle.

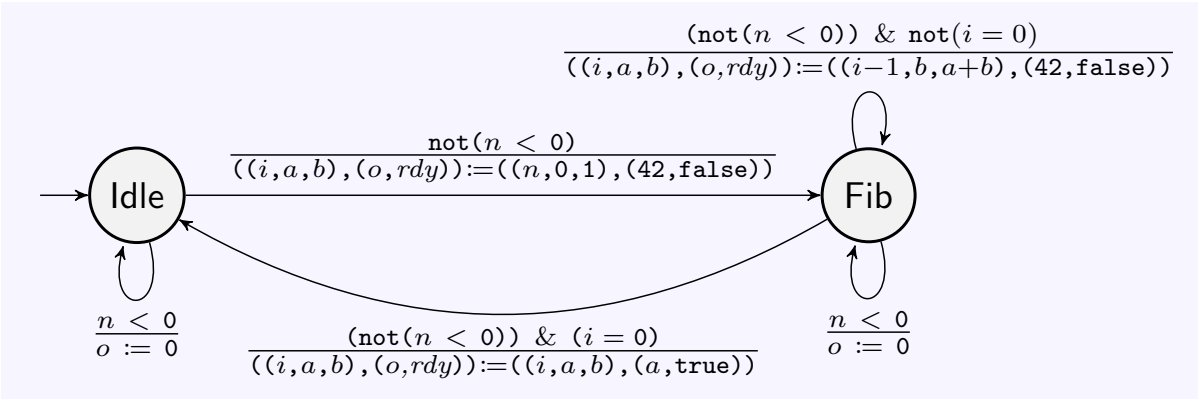


FIGURE 5.5 Implantation inefficace du programme ECLAT de la figure 3.8

Nous obtenons le circuit donné à la figure 5.6. La conditionnelle (`if n < 0 then 0 else` $\cdots$) qui se trouve dans le programme source est une structure de contrôle : elle n'évalue qu'une seule de ses deux branches en fonction de la condition. C'est pourquoi, nous l'implantons en utilisant deux (et non pas un) multiplexeurs pilotés par la condition `n < 0`. L'un sélectionne la sortie ; l'autre empêche la mise à jour des registres par la fonction f lorsque `n < 0`.

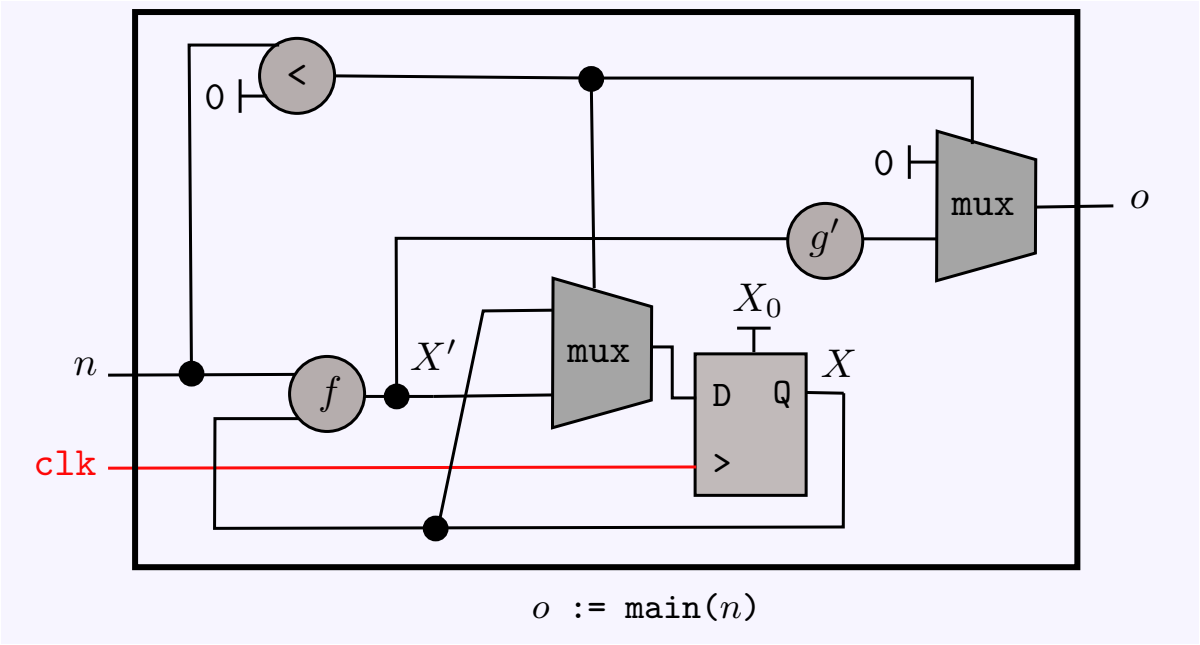


FIGURE 5.6 Implantation efficace du programme ECLAT de la figure 3.8

Cet exemple met en évidence un point de choix crucial dans l'implantation d'ECLAT : nous avons besoin d'un langage intermédiaire orienté flot de données capable de réaliser de telles opérations sur la structure des circuits. Or ce langage doit aussi permettre l'encodage d'automates orienté flot de contrôle. C'est pourquoi nous définirons à la section 5.5 un langage *ad hoc* appelé MINI-HDL. Il sera compilé en VHDL et servira de cible pour implanter ECLAT.

5.2.3 De la nécessité de dupliquer les circuits

La duplication de circuit est une technique courante en description de matériel. Par exemple, au chapitre 2, nous avons présenté un additionneur complet 1-bit (figure 2.9). Pour construire un circuit capable d'additionner des entiers sur 32 bits, nous pouvons connecter en série 32 instances de cet additionneur 1-bit. Le circuit résultant est 32 fois plus large. Son implantation sur FPGA consomme 32 fois plus d'éléments logiques.

Un point essentiel est qu'on ne peut pas utiliser deux fois un même circuit sans le dupliquer au cours d'un même cycle d'horloge. Par exemple, si l'on veut implanter l'expression ECLAT `(a + b) + c`, nous devons utiliser deux additionneurs. Il en va de même pour l'expression `let f(x,y) = x + y in f(f(a,b),c)` compte tenu de la sémantique d'ECLAT qui prédit que cette expression est instantanée.

À la sous-section 5.2.1, nous avons montré que chaque calcul ECLAT (comme par exemple `fibonacci(`n`)`) est encodé sous la forme d'un automate à partir duquel un circuit est réalisé. On peut alors se demander comment implanter l'expression `(fibonacci(i)` $\|$ `fibonacci(j))`. Nous choisissons pour cela d'adapter la technique présentée à la sous-section 5.2.2 : la composition parallèle de deux calculs ECLAT juxtapose leurs implantations et synchronise leurs résultats. De cette manière, nous ne construisons pas explicitement un produit synchrone d'automates, qui risquerait de faire exploser le nombre d'états et donc la taille du circuit. Il s'agit littéralement de deux circuits placés côte à côte, ce qui conduit à dupliquer la fonction `fibonacci` pour respecter la sémantique d'ECLAT : les deux calculs doivent progresser de façon synchronisée avec une pause d'un cycle à chaque appel récursif.

Pour les mêmes raisons, la construction **`exec`** induit une duplication. Par exemple, la composition séquentielle suivante est implantée en juxtaposant deux instances du circuit réalisant le calcul de `fibonacci` :

```
1 let (x,rdy1) = exec fibonacci(i) default 0 in
2 let (y,rdy2) = exec fibonacci(j) default 0 in
3 (x,y)
```

5.2.4 Duplication et partage

Les conceptrices et concepteurs de matériel ont fréquemment besoin de partager des circuits ou des ressources (voir chapitre 6) quitte à introduire des temps de synchronisation additionnels dans les calculs. C'est pourquoi l'implantation d'ECLAT a pour caractéristique essentielle le partage des appels de fonctions récursives terminales (avec des limitations discutées à la sous-section 5.2.6). Le mot clé "**`rec`**" sert alors d'annotation pour forcer une fonction [3] à être partagée en contrepartie d'un retard d'un cycle à chaque appel.

Par exemple, tentons d'additionner trois entiers a, b et c avec un seul additionneur. Une modélisation possible d'un tel circuit est donnée sur la figure 5.7. L'automate comporte trois

3. Cette fonction ne s'appelle pas nécessairement elle même.

états : `Idle`, `Add` et `Pause`. Le calcul commence dans l'état `Idle` ; les variables `x` et `y` reçoivent des copies de a et b puis l'automate passe dans l'état `Add`. Dans l'état `Add`, l'automate additionne `x` et `y`, met le résultat dans `t` puis passe dans l'état `Pause` qui sert de point de synchronisation. La première fois où l'automate entre dans l'état `Pause`, `x` reçoit `t` et `y` reçoit c, puis l'automate passe dans l'état `Add` : c'est cela qui permet de réutiliser l'additionneur se trouvant dans la transition reliant l'état `Add` à l'état `Pause`. La deuxième fois où l'automate entre dans l'état `Pause`, l'automate repasse dans l'état `Idle` (le calcul se termine).

Pour distinguer les arrivées dans l'état `Pause`, la variable `id` identifie les deux continuations possibles depuis `Pause`, l'une poursuivant le calcul dans l'état `Add` et l'autre terminant le calcul.

La variable `o` désigne le résultat du calcul. Lorsque le calcul est en cours, elle reçoit une valeur par défaut [4] (ici `0`) à la manière de ce que fait ECLAT avec **`exec`**.

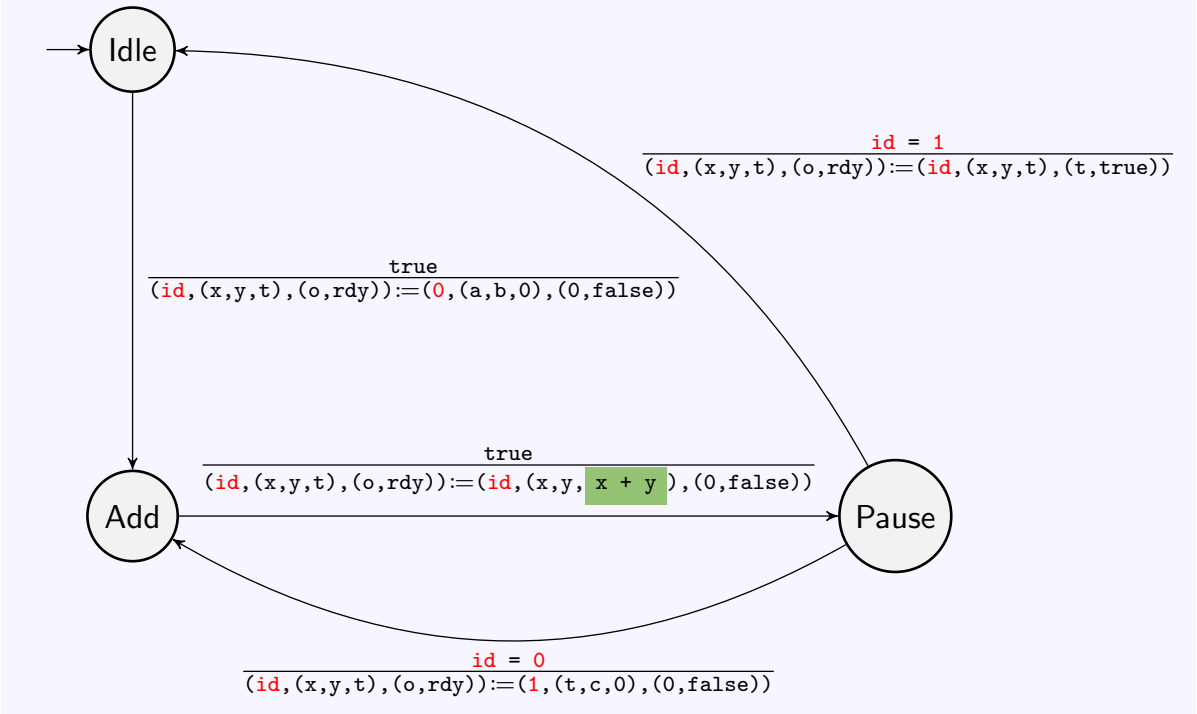


FIGURE 5.7 Automate de calcul de $(a + b) + c$ utilisant un seul additionneur

ECLAT est suffisamment précis pour programmer cet automate. Nous en donnons une définition possible sur la figure 5.8. Comme attendu, elle a le même comportement temporel que l'automate (figure 5.7) et utilise un seul additionneur. En effet, les définitions des fonctions `delay` (ligne 1) et `add` (ligne 2) sont annotées avec le mot clé **`rec`**. Les appels à ces fonctions sont donc partagés aux prix d'un cycle de retard à chaque appel. Cela montre l'interêt du partage en tant que trait de programmation à part entière d'ECLAT, qui est complémentaire à la sémantique temporelle du langage.

4. C'est un choix de modélisation, pour cet automate donné, qui s'inspire de la construction **`exec`** d'ECLAT.

```
1  let rec delay() = () ;;      (* attend un cycle puis termine *)
2  let rec add(x,y) = x + y ;; (* attend un cycle puis retourne x + y *)
3
4  let main (a,b,c) = (* calcule (a + b) + c avec un seul additionneur *)
5    let (o,rdy) = exec
6                    let t = add(a,b) in
7                    delay();
8                    let res = add(t,c) in
9                    delay();
10                   res
11                default 0
12   in (o,rdy)
```

FIGURE 5.8 Calcul de $(a + b) + c$ en ECLAT utilisant un seul additionneur

5.2.5 Des synchronisations instantanées

Les synchronisations en ECLAT sont instantanées. C'est en particulier le cas pour la composition séquentielle `let` $p = e_1$ `in` e_2. Ce n'est pas sans poser de difficultés d'implantation car, si e_1 est un appel à une fonction récursive terminale par exemple, le retour de l'appel doit être instantané [5]. Cependant, cela a des avantages en termes de programmabilité. Ainsi, à partir de la figure 5.8, il est tout à fait possible d'écrire en ECLAT :

```
let t = add(a,b) in
add(t,c)
```

En retirant ainsi les deux appels à `delay`, le calcul se termine alors en seulement deux cycles (au lieu de quatre cycles sur les figures 5.7 et 5.8) tout en maintenant le partage du circuit `+`. Or, cette expression n'est pas représentable avec le formalisme d'automates de la figure 5.7 ; en effet, cela conduirait nécessairement à faire figurer l'opérateur `+` dans deux transitions distinctes sortant de l'état `Add`, et donc briser le partage. Le langage intermédiaire MINI-HDL, que nous introduirons à la section 5.5 pour compiler ECLAT, devra être en mesure de représenter finement ce type de comportement temporel avec partage.

Les synchronisations instantanées constituent un choix de conception fondamental d'ECLAT. Grâce à cela, toutes les constructions de programmation d'ECLAT (à l'exception des fonctions récursives terminales qui sont spécifiques aux calculs) ont le même comportement fonctionnel et temporel aussi bien dans les calculs que dans les interactions.

Ce choix de conception d'ECLAT vise aussi à offrir une certaine simplicité de programmation. La précision qui en résulte aide d'ailleurs à réduire le temps d'exécution des calculs décrits.

5. Toute transition dans un automate synchrone prend un cycle. L'encodage des synchronisations instantanées dans un automate implique donc des manipulations complexes des transitions. Ces manipulations viennent s'ajouter à la problématique de partage des appels de fonctions récursives terminales propre à l'implantation d'ECLAT sur FPGA.

5.2.6 Traits applicatifs et imbrications : un partage limité

Les traits de programmation applicatifs d'ECLAT, tels l'ordre supérieur et le polymorphisme paramétrique, constituent bien sûr un volet important de la compilation d'ECLAT. Cela pose d'ailleurs des difficultés d'implantation qui sont spécifiques à la cible FPGA.

Premièrement, contrairement aux constantes d'ECLAT, les valeurs fonctionnelles ne sont pas représentables en pratique sous forme de vecteurs de bits : ce sont des valeurs dans la sémantique, mais ce ne sont pas des valeurs dans les circuits engendrés. Elles doivent donc être transformées à la compilation, tout en préservant la sémantique des programmes. On s'appuie pour cela sur le système de types d'ECLAT qui rejette certaines classes de programmes fonctionnels que l'on ne sait pas compiler efficacement sur FPGA. Rappelons en particulier le type fonctionnel $\tau \xrightarrow{\delta} \mathring{\tau}$ qui force, par construction, le type de retour à être un type de base $\mathring{\tau}$ (cf. le langage des types d'ECLAT défini figure 3.1).

Deuxièmement, concernant le polymorphisme paramétrique, toute représentation uniforme des valeurs aurait un coût rédhibitoire en termes de taille et de précision des circuits décrits.

Pour ces deux raisons, le choix d'implantation actuel du compilateur ECLAT consiste à *inliner* systématiquement les fonctions d'ordre supérieur et les fonctions polymorphes paramétriques. Ce sont des exceptions dans la méthodologie introduite à la sous-section 5.2.4 qui visait à partager les appels de fonctions récursives terminales. Ces exceptions s'ajoutent aux cas où une fonction récursive terminale est appelée dans plusieurs blocs **exec** ou dans des branches parallèles : ce point (qui a été discuté à la section 5.2.3) s'explique par l'impossibilité physique d'exécuter un même circuit plusieurs fois (c'est-à-dire, avec plusieurs entrées) au cours d'un cycle d'horloge.

Il reste enfin à prendre en compte les imbrications avancées exprimables en ECLAT. Pour respecter la sémantique, le partage ne peut être réalisé que si la fonction appelée ne contient pas de blocs **reg**. On rappelle que l'imbrication de blocs **exec** est interdite. Aussi, par construction, les fonctions récursives ne contiennent jamais de blocs **exec** [6].

La figure 5.9 donne une vue synthétique des différents cas de duplication et de partage des appels de fonctions en ECLAT. Cette règle restera valable au chapitre 6 lorsque nous ajouterons une mémoire partagée.

Les appels à une fonction f peuvent être partagée si et seulement si f est récursive terminale, monomorphe avec un argument $x : \mathring{\tau}$ où $\mathring{\tau}$ est un type de base [7] et ne contenant pas de blocs **reg**. À noter que la duplication d'une fonction f ne duplique pas les fonctions définies localement dans f, sauf si f est une fonction d'ordre supérieur. Le partage reste localisé dans un bloc **exec** ou une branche donnée de paire parallèle `(`$e_1 \| e_2$`)`.

6. Si tel était le cas, la récursion ne serait pas *gardée* par un **exec** : le programme serait alors non instantané et il ne serait donc pas bien typé.

7. Les types de bases sont les types des constantes, sachant que les paires de constantes sont des constantes (cf. la figure 3.1).

5.2.7 Synthèse

Cette section a présenté les principaux points de choix relatifs à l'implantation matérielle d'Eclat. Les calculs Eclat sont compilés sous forme d'automates en respectant la sémantique présentée au chapitre 4. Cette sémantique impose que les synchronisations entre calculs soient instantanées. Elle autorise aussi des imbrications de traits de programmation combinant flot de contrôle et flot de données. C'est pourquoi la compilation d'Eclat requiert un langage intermédiaire adapté (voir section 5.5).

Par exemple, la compilation d'une paire parallèle $(e_1 \| e_2)$ consiste à encoder séparément les expressions e_1 et e_2 sous forme d'automates, puis à juxtaposer leurs représentations flot de données. Il faut alors synchroniser leurs sorties de telle sorte que les deux calculs réalisés démarrent toujours au même cycle d'horloge. Cette approche évite l'explosion combinatoire du nombre d'états inhérente aux produits d'automates. Elle justifie aussi l'une des principales motivation d'Eclat, à savoir le mélange de calculs (orientés flot de contrôle) et d'interactions (orientées flot de données).

À cela s'ajoute les traits de programmation applicatifs d'Eclat, comme les fonctions d'ordre supérieur, dans la lignée des langages de description de matériel fonctionnels [48].

Un circuit, par nature, traite une entrée par cycle ; il ne peut pas "être appelé" deux fois dans un même cycle. Les duplications de circuits sont donc courantes, comme le fait de connecter en série 32 instances (ou copies) d'un additionneur complet 1-bit pour former un additionneurs complet 32-bit. Dans ce contexte, le partage de circuits ou de ressources (voir chapitre 6) est une technique essentielle en conception de matériel pour réduire la taille des circuits réalisés.

Un choix d'implantation déterminant d'Eclat a consisté à partager les appels de fonctions récursives terminales. Cela offre aux programmeuses et programmeurs un contrôle sur la taille des circuits depuis le programme source (sous certaines conditions, voir figure 5.9) en annotant ou non les déclarations de fonctions avec le mot clé "`rec`". Par exemple, la réalisation matérielle de l'expression `let rec f(x,y) = x + y in f(f(a,b),c)` contient un seul additionneur tandis que `let f(x,y) = x + y in f(f(a,b),c)` en contient deux. La première prend deux cycles alors que la seconde est instantanée.

type de fonction appelée	comportement temporel	choix d'implantation
opérateur	démarrage instantané	duplication
fonction non-récursive	démarrage instantané	duplication
fonction récursive terminale monomorphe dont l'argument est un type de base et ne contenant pas de blocs `reg`	démarrage retardé d'un cycle	partage au sein d'un même bloc `exec` ou d'une même branche paralèlle
autres fonctions récursives terminales	démarrage retardé d'un cycle	duplication

Figure 5.9 Règles de partage et de duplication des fonctions Eclat

Ces caractéristiques complexifient encore un peu plus la compilation du langage ; c'est à notre avis le prix a payer pour une programmation de FPGA qui soit :

— *sûre*, en termes d'interaction avec les entrées/sorties ;

— *expressive*, en termes d'imbrication de calculs et d'interactions ;

— et *efficace*, de par la capacité à contrôler à la fois le temps d'exécution des calculs et le compromis entre partage et duplication.

5.3 Chaîne de compilation

La chaîne de compilation d'ECLAT est schématisée sur la figure 5.10. Le programme source (un fichier texte) passe d'abord par une phase d'analyse syntaxique qui construit l'arbre de syntaxe abstraite du programme. Le programme est ensuite typé. Puis, par une série de transformations de programmes préservant la sémantique, le programme est *normalisé*, c'est-à-dire traduit dans un sous-ensemble d'ECLAT appelé CORE-ECLAT. Ce langage intermédiaire est ensuite compilé vers MINI-HDL, un langage de description de matériel *ad hoc* orienté flot de données. MINI-HDL est alors traduit en VHDL. La description de matériel résultante est simulée sur PC et synthétisée par des outils de synthèse logique comme le synthétiseur QUARTUS d'INTEL. L'annexe C est un guide d'utilisation du compilateur ECLAT.

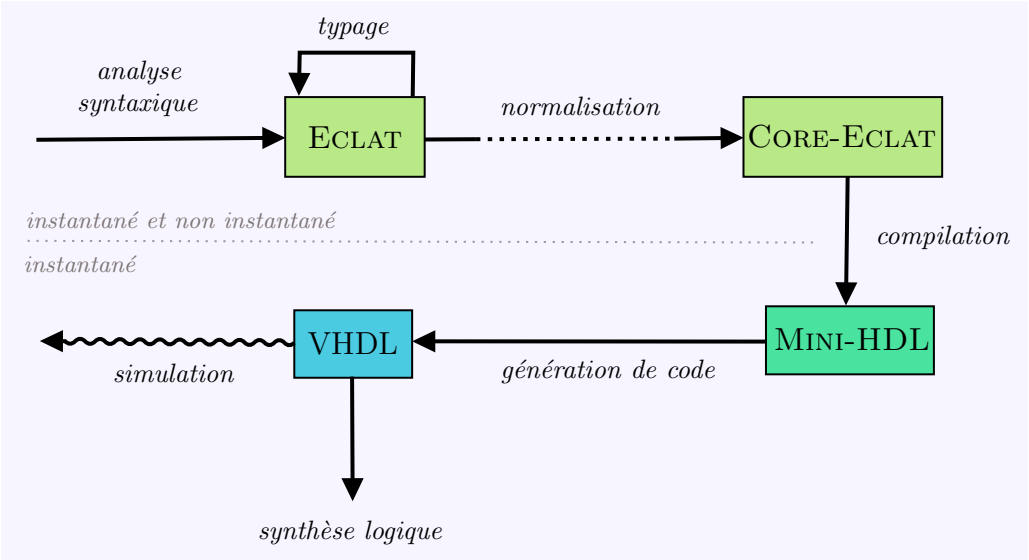


FIGURE 5.10 Chaîne de compilation d'ECLAT sur FPGA

La suite de ce chapitre adopte un angle de présentation *bottom-up* qui vise à définir, par étapes, le schéma de compilation d'ECLAT vers VHDL.

Nous présentons d'abord succinctement l'interopérabilité en ECLAT, qui donne une vue complète du langage en lien avec son implantation en VHDL.

MINI-HDL est défini à la section 5.5. Sa sémantique est proche de celle d'ECLAT tout en étant plus simple (car tout programme MINI-HDL termine instantanément). MINI-HDL peut

être vu comme un sous-ensemble synthétisable de VHDL. Il comprend cependant des constructions communes avec ECLAT (la conditionnelle par exemple) et hérite même des constantes, des opérateurs et du langage de types de base d'ECLAT. MINI-HDL introduit de plus une construction de traitement par cas (**case/with**) et une construction d'automate (**automaton**) dérivée de **case/with**. C'est un point essentiel pour la compilation de CORE-ECLAT vers MINI-HDL.

La génération de code VHDL à partir de MINI-HDL est définie à la section 5.6 (y compris l'instanciation de circuits extérieurs). Cette étape est grandement facilitée par les choix de conception de MINI-HDL, qui est un langage d'instructions inspiré de VHDL.

CORE-ECLAT est présenté à la section 5.7. C'est un forme normalisé d'ECLAT spécialement conçue pour être encodable directement sous forme d'automates.

La compilation de CORE-ECLAT vers MINI-HDL, qui réalise cette encodage sous forme d'automates, est définie à la section 5.8. Le schéma de compilation proposée est relativement complexe, compte tenu des difficultés motivées à la section 5.2.

Finalement, la section 5.9 décrit la normalisation d'ECLAT vers CORE-ECLAT.

5.4 Interopérabilité

L'application $f(x)$ en ECLAT peut faire référence à une fonction externe f implantée en VHDL (autrement dit, un *circuit extérieur*). Il faut pour cela que la fonction externe f soit déclarée dans le programme source avec la directive **external** $f : \mathring{\tau} \xrightarrow{\delta} \mathring{\tau}$ (voir sous-section 3.2.6 page 36). Par convention, chaque programme ECLAT **let** $f =$ **fun** $(p{:}\mathring{\tau}) : \mathring{\tau}' \to e$ est compilé sous la forme d'un composant VHDL appelé aussi f ; cela rend possible l'appel de programme ECLAT (compilé en VHDL) depuis d'autres programmes.

Pour implanter ce mécanisme d'interopérabilité entre ECLAT et VHDL, le langage intermédiaire MINI-HDL (qui sera présenté au chapitre suivant) comporte une instruction dédiée $(x, rdy) \coloneqq \mathbf{run}_i\ f\ a$. Celle-ci crée statiquement (par duplication) une instance i d'un circuit extérieur f en lui passant la valeur d'entrée a et en récupérant le résultat x ainsi qu'une sortie *rdy* indiquant si l'appel est terminé. Cela permet "d'appeler" des circuits instantanés (c'est-à-dire, tels que *rdy* ne vaut jamais `false`) ou réalisant des calculs longs. Les circuits extérieurs sont généralement écrit en VHDL.

La durée de la fonction externe f appelée depuis un programme ECLAT n'est pas nécessairement instantanée. Le compilateur ECLAT réalise les synchronisations nécessaires pour que f puisse être utilisée dans des calculs (orienté flot de contrôle) avec attente du résultat.

Il faut alors considérer le cas où le calcul est suspendu, par exemple lorsque le calcul se situe sous un **exec**, lui même placé dans la branche *else* d'une conditionnelle (comme sur dans l'exemple de la figure 3.8). Ce cas de figure a été largement discuté à la section 5.2. C'est la raison pour laquelle les circuits extérieurs de MINI-HDL ont une entrée booléenne implicite `go` qui indique les cycles où le circuit doit progresser dans son exécution ou non. À cela s'ajoute une sortie `rdy` implicite qui indique quand le circuit (de calcul ou non) se termine (à la manière de la construction **exec** d'ECLAT). Si le circuit extérieur n'est pas un circuit de calcul, sa sortie `rdy` vaut toujours `true`.

La figure 5.11 donne un exemple de programme ECLAT déclarant deux fonctions externes : une fonction instantanée `counter` et une fonction `fibonacci`. La conditionnelle à la ligne 6 suspend l'exécution de sa partie *else* lorsque `suspend` vaut `true` (ce qui correspond au concept de temps multiforme des langages synchrones). Dans ce cas, l'exécution des blocs `exec` et `reg` et en particulier la progression des appels de fonctions externes doit être suspendue.

Ces fonctions externes doivent être définies en VHDL [8] (ou engendrées). Leur argument doit être un type de base comme précisé à la section 3.2.6. Elles reçoivent une entrée implicite `go` (en plus de l'argument) et retourne à la fois le résultat et une sortie `rdy` implicite.

Lorsque l'entrée `go` est à `false`, l'exécution de la fonction externe est suspendue ; ainsi, le compteur à la ligne 10 de la figure 5.11 n'est incrémenté que lorsque l'entrée `suspend` est `false`. Chaque appel de fonction externe dans `main` attend que la sortie `rdy` soit à `true` avant de lire le résultat. En particulier, si la fonction externe termine instantanément (en mettant sa sortie `rdy` à `true`), l'appel dans `main` termine aussi instantanément.

```
1  external counter : bool => int<16> ;;
2
3  external fibonacci : int<32> -> int<32> ;;
4
5  let main ((i1,i2,suspend) : int<32> * int<32> * bool) : int<16> =
6    if suspend then -1 else (
7      let (o,_) = exec
8                    let (x1,x2) = (fibonacci(i1) || fibonacci(i2)) in
9                    y1 + y2
10                 default 0 reset false in
11     let c = counter(o) in
12     c) ;;
```

FIGURE 5.11 Appels de fonctions externes depuis ECLAT

Remarque : la réinitialisation des appels de fonctions externes (via la clause `reset` d'un bloc `exec` englobant) n'est actuellement pas supportée par le compilateur ECLAT. Une analyse statique (définie sur la représentation interne du compilateur) détecte, dans les programmes, les blocs `exec` contenant des appels de fonctions externes sous un bloc `exec` avec clause `reset` non réduite à `false` ; ces programmes sont alors rejetés à la compilation.

Les fonctions externes (comme les fonctions `counter` et `fibonacci` déclarées sur la figure 5.11) peuvent être définies directement en ECLAT, comme le montre les figures 5.12 et 5.13. Cela constitue une forme de compilation séparée où les programmes ECLAT appellent d'autres programmes ECLAT.

Ces deux fonctions sont instantanées, chacune prend une entrée `go` indiquant si elle en

8. Puisqu'ECLAT est actuellement compilé en VHDL, comme présenté dans ce chapitre.

cours d'appel, et une sortie `rdy` indiquant l'instant où l'appel se termine. Lorsqu'il est appelé, le compteur termine instantanément et `rdy` vaut `true`. Le calcul de Fibonacci est encapsulé dans un bloc **`exec`** (lignes 9 à 12) ; ainsi le calcul peut être suspendu à tout moment lorsque l'entrée `go` passe à `false`. C'est une technique classique d'implantation du temps logique dans les langages synchrones [41].

```
1  let counter ((go,i) : bool * bool) : int<16> * bool =
2    if go then
3      let o = reg (fun c -> if i then c + 1 else c) init 0
4      in (o,true)
5    else (0,false) ;;
```

FIGURE 5.12 Définition d'un circuit externe en ECLAT

```
6  let fibonacci ((go,n) : bool * int<32>) : int<32> * bool =
7    if go then (
8      let (o,rdy) = exec
9                      let rec fib (i,a,b) =
10                       if i = 1 then a else fib(i - 1, b, a + b)
11                     in fib(n,0,1)
12                   default 0
13      in (o,rdy)
14   ) else (0,false) ;;
```

FIGURE 5.13 Définition d'un calcul externe en ECLAT

Cette section a présenté l'interoprabilité en ECLAT pour deux raisons. D'une part, c'est un trait de programmation interessant que nous utiliserons au chapitre 7 pour programmer l'application Pong (section 7.4). D'autre part, cela clarifie le fonctionnement de l'instruction $\texttt{run}_i$ que nous allons introduire en MINI-HDL, pour l'appel de circuits externes avec une entrée `go` implicite et une sortie `rdy`. Ces deux paramètres, `go` et `rdy`, seront explicités à la section suivante, dans le schéma de traduction de MINI-HDL vers VHDL.

5.5 Abstraction de la cible : Mini-HDL

Cette section présente MINI-HDL, un langage de description de matériel statiquement typé, inspiré de VHDL. C'est un langage synchrone flot de données avec horloge globale implicite, comme μFP [196], CλASH [12] et LUSTRE [175] par exemple. Chaque programme MINI-HDL est une fonction de transition qui transforme un flot d'entrées en un flot de sorties.

5.5.1 Syntaxe et sémantique intuitive

En MINI-HDL, les connexions entre les entrées et les sorties sont exprimées en style impératif dans un langage d'instructions *à la VHDL*, ce qui facilite justement la traduction de MINI-HDL vers VHDL. Plus fondamentalement, ce choix de conception consistant à proposer un langage d'instructions a pour but de faciliter la compilation d'ECLAT : le style impératif constitue une méthode de conception *comportementale*, par opposition aux méthodes *structurelles* [191]. Par exemple, l'instruction conditionnelle (`if/then/else`) de VHDL et de MINI-HDL n'est pas un simple multiplexeur sélectionnant l'une de ses entrées ; c'est une structure de contrôle qui est synthétisée (par le synthétiseur logique) en une combinaison de multiplexeurs [9].

La figure 5.14 définit la syntaxe de MINI-HDL. Chaque programme Π est un circuit `let` f `(`$p : \mathring{\tau}$`) returns (`$p' : \mathring{\tau}'$`)` $=$ s. Il est identifié par un nom f, possède une entrée p et une sortie p' annotées avec leurs types de base respectifs. Le corps du programme est une instruction qui s'exécute toujours instantanément et assigne une valeur à la sortie.

programme	Π	$::=$ `let` f `(`$p : \mathring{\tau}$`) returns (`$p : \mathring{\tau}$`)` $=$ s
instruction	s	$::=$
ne rien faire		`()`
séquence		$\mid$ s `;` s
conditionnelle		$\mid$ `if` a `then` s `else` s
traitement par cas		$\mid$ `case` a `with` $\overline{c_i \rightarrow s_i}$ `otherwise` s
variable locale		$\mid$ `var` $x : \mathring{\tau}$ `in` s
affectation		$\mid$ $p \coloneqq a$
instanciation		$\mid$ `(`x`,`rdy`)` $\coloneqq$ `run`$_i$ f a
atome	a	$::=$ $x \mid c \mid$ `(`a`,`a`)` $\mid$ op a
motif	p	$::=$ $x \mid$ `(`x`,`p`)`
constante	c	$::=$ `()` $\mid$ `true` $\mid$ `false` $\mid$ `(`l `: int<`$\mathring{\theta}$`>)` $\mid$ `(`c`,`c`)` $\mid$ $\{c_1, \cdots c_n\}$
opérateur combinatoire	$\mathring{op}$	$::=$ `not` $\mid$ `&` $\mid$ `or` $\mid$ `xor` $\mid$ `fst` $\mid$ `snd`
		$\mid$ `+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `mod` $\mid$ `<` $\mid$ `>` $\mid$ `<=` $\mid$ `>=` $\mid$ `=` $\mid$ `resize_int<`$\mathring{\theta}$`>`
		$\mid$ `vect_create<`$\mathring{\theta}$`>` $\mid$ `vect_nth` $\mid$ `vect_copy_with` $\mid$ `vect_size`
type de base	$\mathring{\tau}$	$::=$ `unit` $\mid$ `bool` $\mid$ `int<`$\mathring{\theta}$`>` $\mid$ $\mathring{\tau}$ `*` $\mathring{\tau}$ $\mid$ $\mathring{\tau}$ `vect<`$\mathring{\theta}$`>`
taille	$\mathring{\theta}$	$::=$ n

FIGURE 5.14 Syntaxe de MINI-HDL

Le langage d'instructions de MINI-HDL comprend la construction `case/with` déjà évoquée, qui permet notamment de représenter des automates (nous y reviendrons à la figure 5.17). Les autres instructions sont principalement `()` (*ne rien faire*), la séquence, la conditionnelle (comme en ECLAT), la déclaration de variable (`var/in`) et l'affectation $p \coloneqq a$ où p est une paire imbriquée de variables `(`x_1`,(`x_2`,` $\cdots x_n$`))`. À cela s'ajoute une construction d'instanciation de circuit externe $x \coloneqq$ `run`$_i$ $f(a)$ qui dénote l'appel de l'instance i du circuit f avec l'argument

9. cf. la conditionnelle réalisée avec deux multiplexeurs sur la figure 5.6 page 82 : l'un des deux multiplexeurs empêche l'exécution de la branche *else*.

a et l'assignation du résultat de f à la variable x. Cela sert à appeler des fonctions externes (écrites par exemple en VHDL) depuis ECLAT (compilé en MINI-HDL).

L'argument a dans `(`x`,`rdy`)` $:=$ **`run`**$_i$ $f(a)$ est une expression combinatoire (aussi appelée *atome*), qui apparaît aussi en partie droite des affectations et comme condition pour l'instruction **`if/then/else`**. Les atomes sont définis comme un sous-ensemble du langage d'expressions d'ECLAT (variables, constante, paire d'atomes ou application d'opérateur). Les constantes MINI-HDL sont les constantes ECLAT. Les opérateurs MINI-HDL sont les opérateurs d'ECLAT. Les types de base et tailles de MINI-HDL sont les types de base et tailles d'ECLAT ne comportant pas d'inconnues de types de base ni d'inconnue de taille.

Dans la suite, on note n déclarations consécutives de variables x_i`:`$\mathring{\tau}$ avec l'abréviation **`var`** x_1`,` x_2`,` $\cdots x_n$ `:` $\mathring{\tau}$ **`in`** s. Comme en ECLAT, les applications d'opérateurs sont notées de façon infixe, l'information de type `(`l `:` `int<`$\mathring{\theta}$`>)` des constantes entières peut être omise et nous utilisons des n-uplets pour simplifier l'écriture des paires imbriquées.

Nous introduisons enfin des types énumérés `enum(A | B | C |` $\cdots$`)` implantés avec des entiers ; il s'agit d'un simple sucre syntaxique (`A` $\stackrel{\text{def}}{=}$ `0`, `B` $\stackrel{\text{def}}{=}$ `1`, `C` $\stackrel{\text{def}}{=}$ `2`, $\cdots$) pour faciliter l'écriture et la lecture des automates.

Chaque sortie ou variable locale de type $\mathring{\tau}$ est implicitement initialisée avec une constante par défaut $d(\mathring{\tau})$, qui est une séquence de bits à `0`. La définition de $d(\mathring{\tau})$ est donnée figure 5.15. Par exemple, si une variable MINI-HDL est de type **`bool * int<8>`**, alors sa valeur par défaut est **`(false,(0:int<8>))`**. On retrouve ce choix de conception dans d'autres langages de programmation, comme JAVA par exemple.

$$
\begin{aligned}
d(\texttt{unit}) &\stackrel{\text{def}}{=} \texttt{()} \\
d(\texttt{bool}) &\stackrel{\text{def}}{=} \texttt{false} \\
d(\texttt{int<}n\texttt{>}) &\stackrel{\text{def}}{=} \texttt{(0:int<}n\texttt{>)} \\
d(\mathring{\tau}_1 \texttt{ * } \mathring{\tau}_2) &\stackrel{\text{def}}{=} \texttt{(}\, d(\mathring{\tau}_1)\, \texttt{,}\, d(\mathring{\tau}_2)\, \texttt{)} \\
d(\mathring{\tau}\ \texttt{vect<}n\texttt{>}) &\stackrel{\text{def}}{=} \texttt{\{}\underbrace{d(\mathring{\tau})\texttt{,} \cdots d(\mathring{\tau})}_{n \text{ fois}}\texttt{\}}
\end{aligned}
$$

FIGURE 5.15 Valeurs par défaut d'une variable MINI-HDL en fonction de son type

La figure 5.16 donne un exemple de programme MINI-HDL encodant un automate. Il décrit le circuit de la figure 5.6 qui fait progresser le calcul *Fibonacci* uniquement lorsque l'entrée n est positive. Pour représenter l'état, nous utilisons ici un type énuméré à deux valeurs (`0` pour `Idle` et `1` pour `Fib`). L'automate est initialement dans l'état `Idle`, qui est la constante par défaut d(`enum(Idle|Fib)`) de ce type énuméré [10].

L'expression **`case/with`** (lignes 7 à 12, figure 5.16) encode fidèlement l'automate de la figure 5.3, ou plus exactement, sa version flot de données définie à la figure 5.4. La variable

10. La valeur par défaut est la première constante déclarée en partant de la droite ; c'est celle qui est encodée avec l'entier 0.

`fib_arg` (présent dans la version MINI-HDL mais pas dans l'automate) anticipe un choix de conception dans le schéma de compilation de CORE-ECLAT vers MINI-HDL : chaque fonction récursive f a *un* argument appelé par convention f`_arg` et qui peut être une imbrications de paire comme ici.

```
1  let main(n:int<16>) returns (o:int<16>) =
2    var state : enum(Idle | Fib) in (* Idle par défaut *)
3    var fib_arg : int<16> * int<16> * int<16> in
4    var a, b, i : int<16> in
5    var rdy : bool in
6    if n < 0 then o := 0 else (
7    case state with
8    | Idle -> rdy := false; fib_arg := (0, 1, n); state := Fib
9    | Fib -> (a, b, i) := fib_arg;
10             if i = 0 then o := a; rdy := true; state := Idle
11             else fib_arg := (b, a + b, i - 1) ; state := Fib
12   otherwise ();
13   if rdy then () else o := 42
14   )
```

FIGURE 5.16 Encodage du circuit de la figure 5.6

Les deux transitions sortant de l'état `Fib` dans l'automate (figure 5.3) correspondent au cas `Fib ->` $\cdots$ de l'instruction **`case/with`** dans le programme MINI-HDL. Chaque cas est une instruction MINI-HDL qui peut aussi contenir des instructions **`if/then/else`** ou **`case/with`** ; cela permet un contrôle fin du partage dans les circuits tout en étant suffisamment expressif pour encoder les synchronisations instantanées d'ECLAT (voir sous-section 5.2.5).

Rappelons que ce programme (figure 5.16) est instantané (comme tous les programmes MINI-HDL). À chaque cycle, il s'exécute de bout en bout et les sorties sont modifiées aux cours de l'exécution.

5.5.2 Proximité entre Eclat et Mini-HDL

On notera un ressemblance avec le programme ECLAT original (figure 3.8), dont nous redonnons le code ici :

```
1  let fibonnacci(n) =
2    let rec fib (a,b,i) = ...
3    in fib (0,1,n) ;;
4  let main n =
5    if n < 0 then 0 else
6    let (o,rdy) = exec fibonacci(n) default 42
7    in o ;;
```

Le `if/then/else` dans le programme ECLAT correspond à un `if/then/else` dans le programme MINI-HDL (ligne 6, figure 5.16). Profitons de cette remarque pour souligner que le schéma de compilation de CORE-ECLAT n'est pas toujours aussi direct, du fait des synchronisations instantanées imposées par la sémantique d'ECLAT. Par exemple, l'expression ECLAT :

```
1 let rec loop(z) = ... in
2 let x = if n < 0 then fibonacci(n) else 42 in
3 loop(x * 4)
```

sera compilée en une instruction MINI-HDL de cette forme :

```
1 case state with
2 | Idle -> if n < 0 then state := Fib
3           else (x := 42; loop_arg := x * 4; state := Loop)
4 | Fib -> (a, b, i) := fib_arg;
5          if i = 0 then (x := a; loop_arg := x * 4 ; state := Loop)
6          else ...
7 | Loop -> ...
8 otherwise ()
```

La continuation du calcul, à savoir `loop(x * 4)`, a été dupliquée dans le code MINI-HDL engendré pour satisfaire les contraintes temporelles du programme source. Notons que la partie dupliquée est *uniquement* la partie instantanée située entre le point de synchronisation et le prochain appel récursif terminal (ici la fonction `loop`).

5.5.3 Extension de Mini-HDL avec des automates

Pour faciliter la compréhension du schéma de compilation de CORE-ECLAT vers MINI-HDL, nous introduisons deux constructions dérivées pour représenter des automates plus directement. Ce sont deux instructions définies sur la figure 5.17.

$$\begin{pmatrix} \texttt{automaton}_{\mathcal{L}} \\ \mid f_1 \rightarrow s_1 \ \cdots \\ \mid f_n \rightarrow s_n \\ \texttt{end} \end{pmatrix} \equiv \begin{pmatrix} \texttt{case state}_{id} \texttt{ with} \\ \mid c_1 \rightarrow s_1 \ \cdots \\ \mid c_n \rightarrow s_n \\ \texttt{otherwise ()} \end{pmatrix} \text{ où } c_i = const(id, f_i)$$

$$\texttt{continue}_{id}\ f \quad \equiv \quad \texttt{state}_{id} \coloneqq const(id, f)$$

FIGURE 5.17 Constructions dérivées de MINI-HDL

La construction $\texttt{automaton}_{\mathcal{L}}$ représente un automate identifié par une étiquette $\mathcal{L}$ et qui comporte une liste ordonnée de transitions. Chaque transition est une paire formée d'un état et d'une instruction. L'état initial est le premier de la liste de transition, par convention. Il

est toujours encodé par l'entier 0, c'est-à-dire la valeur par défaut de la variable d'état. Cette dernière est implicite ; elle correspond au nom $\texttt{state}_{\mathcal{L}}$ dans la définition de la construction $\texttt{automaton}_{\mathcal{L}}$ de la figure 5.17. La construction $\texttt{continue}_{id}\ f$ est un branchement vers l'état f de l'automate id ; cela consiste à assigner à la variable d'état le numéro $const(id, f)$ associé à l'état f ciblé dans l'automate id.

À noter que la variable d'état doit être déclarée séparément, en dehors de l'automate ; par convention, elle s'appelle $\texttt{state}_{\mathcal{L}}$ pour chaque automate $\mathcal{L}$ (comme sur la figure 5.17). Ce choix de conception vise à simplifier le schéma de compilation ; par exemple, l'état d'un automate peut-être réinitialisé *depuis l'extérieur*, par une simple `continue`) ; c'est de cette manière que nous encoderons la cause `reset` de la construction `exec` ainsi que le redémarrage des paires parallèles $(e_1 \| e_2)$ d'ECLAT.

Un automate est une structure de contrôle (définie comme une variante de l'instruction `case/with`). À chaque fois qu'un automate s'exécute, il déclenche une transition qui met à jour la variable d'état, puis termine, ce qui n'est pas sans rappeler la construction `exec` d'ECLAT.

La figure 5.18 donne un exemple de programme MINI-HDL utilisant la construction dérivée `automaton`. Ce programme est équivalent au programme MINI-HDL de la figure 5.16, à l'exception de la ligne 5 qui réinitialise l'automate *depuis l'extérieur* si l'entrée `n` vaut 123. Cela revient à ajouter une clause "`reset (n = 123)`" à la construction `exec` dans le programme ECLAT original (figure 3.8). Le point virgule à la ligne 5 fait partie d'une séquence : l'instruction ligne 5 est d'abord exécutée, avant l'instruction lignes 6 à 14. De cette manière, lorsque l'automate est réinitialisé (ligne 5), il redémarre dans le même cycle.

```
1  let main(n:int<16>) returns (o:int<16>) =
2    var state_L : enum(Idle | Fib) in (* Idle par défaut *)
3    var fib_arg : (int<16> * int<16>) * int<16> in
4    var rdy : bool in
5    if n = 123 then continune_L Idle else ();
6    if n < 0 then o := 0 else (
7    automaton_L
8    | Idle -> rdy := false; fib_arg := ((0,1),n); continue_L Fib
9    | Fib -> (a, b, i) := fib_arg;
10             if i = 0 then o := a; rdy := true; continue_L Idle
11             else fib_arg := ((b,a+b), i-1) ; continue_L Fib
12   end;
13   if rdy then () else o := 42
14   )
```

FIGURE 5.18 Encodage du circuit de la figure 5.6 avec constructions dérivées

Comme toute autre instruction MINI-HDL comportant des sous-instructions, les blocs

`automaton` peuvent être imbriqués (autrement dit, ce sont des automates hiérarchiques). Sur la figure 5.19 par exemple, l'automate interne est exécuté uniquement lorsque l'automate externe est dans l'état `On`. À la ligne 5, lorsque l'entrée `resume` vaut `true`, les deux automates sont réinitialisés en manipulant directement leur variable d'état. Cette caractéristique sera utilisée dans le schéma de compilation d'ECLAT vers MINI-HDL, pour réinitialiser un bloc `exec` et pour le démarrage synchronisé des paires parallèles $(e_1 \| e_2)$ d'ECLAT.

```
1  let f((start, stop, resume):(bool * bool * bool)
2      returns (o:int<8>) =
3    var state_L : (Off|On) in (* Off par défaut *)
4    var state_L2 : (A|B|C) in (* A par défaut *)
5    if resume then (continue_L Off; continue_L2 A) else
6    automaton_L
7    | Off -> if start then continue_L On else ()
8    | On ->
9        (automaton_L2
10       | A -> o := 1; continue_L2 B
11       | B -> o := 2; continue_L2 C
12       | C -> o := 3; continue_L2 A
13       end;
14       if stop then continue_L Off else ())
15    end
```

FIGURE 5.19 Imbrication d'automates en MINI-HDL

Les automates peuvent être composés en parallèle ; il suffit pour cela de les faire progresser dans le même cycle d'horloge, comme sur la figure 5.20. Ce programme ajoute 3 à sa sortie tous les trois cycles. Le flot de sortie `o` qui en résulte est

$$0 \bullet 0 \bullet 0 \bullet 3 \bullet 3 \bullet 3 \bullet 6 \cdots$$

Dans ce programme, l'automate $\mathcal{L}_1$ effectue instantanément une transition et termine ; puis l'automate $\mathcal{L}_2$ effectue instantanément une transition et termine. Le programme est ré-exécuté à chaque cycle, ce qui conduit les deux automates à progresser en parallèle en se synchronisant à chaque cycle.

À noter que de cette manière, le comportement du parallélisme et purement déterministe : un automate est exécuté, puis le second à chaque cycle. Les deux automates ainsi exécutés pourraient tout à fait lire ou écrire une même variable ; c'est d'ailleurs le cas ici à la ligne 12. Après synthèse logique, les accès concurrents correspondent à des dépendances de données (autrement dit des fils) entre les deux circuits juxtaposés.

```
1  let f(i:unit) returns (o:int<8>) =
2    var x, y : int<8> in
3    var state_{L1} : (A|B|C) in (* A par défaut *)
4    var state_{L2} : (Q1|Q2) in (* Q1 par défaut *)
5    automaton_{L1}
6    | A -> x := 1; continue_{L1} B
7    | B -> x := 2; continue_{L1} C
8    | C -> x := 3; continue_{L1} A
9    end;
10   automaton_{L2}
11   | Q1 -> y := 0; continue_{L2} Q2
12   | Q2 -> if x = 1 then o := y else ();
13           y := y + 1; continue_{L2} Q2
14   end
```

FIGURE 5.20 Composition parallèle d'automates en MINI-HDL

MINI-HDL joue un rôle central dans la chaîne de compilation d'ECLAT vers VHDL (schématisée figure 5.10 page 88). C'est à la fois le langage cible utilisé pour compiler ECLAT et le langage d'entrée du générateur de code VHDL. il est donc important de définir sa sémantique.

5.5.4 Sémantique

MINI-HDL est typé statiquement. Le typage des programmes MINI-HDL est défini par une relation qui associe un type à chaque programme bien typé. La relation de typage est notée :

$$\Gamma_0 \vdash_{\mathsf{prog}} \Pi : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}$$

ce qui se lit : "dans l'environnement de programmes externes Γ_0 (qui assigne à chaque nom de programme f un type de la forme $\mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'$) le programme Π est bien typé". La définition du jugement $\vdash_{\mathsf{prog}}$ fait intervenir deux jugements auxiliaires :

$$\Gamma \vdash_{\mathsf{s}} s : \mathtt{unit}$$

pour typer les instructions (s pour *statement*) et :

$$\Gamma \vdash_{\mathsf{a}} a : \mathring{\tau}$$

pour typer les atomes (a pour *atom*). Le premier se lit "dans l'environnement de typage Γ, l'instruction s est bien typée". Le second se lit "dans l'environnement de typage Γ, l'atome a est bien typé, de type de base $\mathring{\tau}$". Les environnements de typage sont ceux d'ECLAT avec des schémas de types réduits à des types [11].

11. En ECLAT, un schéma de type σ est soit un type τ, soit un type quantifié de la forme $\forall\ 'v\ \cdot \sigma$.

Le jugement $\vdash_{\mathsf{prog}}$ est défini par la règle Ty-prog : le programme est bien typé si son corps est une instruction bien typée dans l'environnement prédéfini Γ_0 étendu avec l'entrée p de type $\mathring{\tau}$ et la sortie p' de type $\mathring{\tau}'$.

$$\textsc{Ty-prog}\ \frac{\Gamma_0[p : \mathring{\tau}][p' : \mathring{\tau}'] \vdash_{\mathtt{s}} s : \mathtt{unit}}{\vdash_{\mathtt{prog}} \mathtt{let}\ f\ (p : \mathring{\tau})\ \mathtt{returns}\ (p' : \mathring{\tau}') = s}$$

Du fait que tout atome Mini-HDL est par définition une expression Eclat, le typage des atomes de Mini-HDL découle directement du système de types d'Eclat (règle Ty-Atom). Le jugement $\Gamma \vdash e : \tau$ (ici restreint à un type de base $\mathring{\tau}$) a été introduit au chapitre 4 pour Eclat.

$$\textsc{Ty-Atom}\ \frac{\Gamma \vdash a : \mathring{\tau}}{\Gamma \vdash_{\mathtt{a}} a : \mathring{\tau}}$$

Le typage des instructions de Mini-HDL est défini sur la figure 5.21. Les règles Ty-S-Skip, Ty-S-Seq, Ty-S-Set, Ty-S-If, Ty-S-Case et Ty-S-VarIn sont standards. Par exemple, l'affectation d'un atome a à une motif p est bien typée si les noms dans p sont présents dans l'environnement de typage et si a est du même type que p (règle Ty-S-Set). On reprend la notation $\mu[p \mapsto v]$ introduite dans la sémantique Eclat pour désigner la mémoire obtenue en affectant aux variables dans p les valeurs correspondantes dans v.

$$\textsc{Ty-S-Skip}\ \Gamma \vdash_{\mathtt{s}} () : \mathtt{unit}$$

$$\textsc{Ty-S-Seq}\ \frac{\Gamma \vdash_{\mathtt{s}} s_1 : \mathtt{unit} \qquad \Gamma \vdash_{\mathtt{s}} s_2 : \mathtt{unit}}{\Gamma \vdash_{\mathtt{s}} s_1\,;\ s_2 : \mathtt{unit}}$$

$$\textsc{Ty-S-Set}\ \frac{\Gamma(x) : \mathring{\tau} \qquad \Gamma \vdash_{\mathtt{a}} a : \mathring{\tau}}{\Gamma \vdash_{\mathtt{s}} x \coloneqq a : \mathtt{unit}}$$

$$\textsc{Ty-S-If}\ \frac{\Gamma \vdash_{\mathtt{a}} a : \mathtt{bool} \qquad \Gamma \vdash_{\mathtt{s}} s_1 : \mathtt{unit} \qquad \Gamma \vdash_{\mathtt{s}} s_2 : \mathtt{unit}}{\Gamma \vdash_{\mathtt{s}} \mathtt{if}\ a\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 : \mathtt{unit}}$$

$$\textsc{Ty-S-Run}\ \frac{\Gamma(f) = (\mathtt{bool} * \mathring{\tau}) \xrightarrow{\mathbf{0}} (\mathring{\tau}' * \mathtt{bool}) \qquad \Gamma \vdash_{\mathtt{a}} a : \mathring{\tau} \qquad \Gamma(x) : \mathring{\tau}' \qquad \Gamma(rdy) : \mathtt{bool}}{\Gamma \vdash_{\mathtt{s}} (x, rdy) \coloneqq \mathtt{run}_i\ f\ a : \mathtt{unit}}$$

$$\textsc{Ty-S-Case}\ \frac{\Gamma \vdash_{\mathtt{a}} a : \mathring{\tau} \qquad i \in \mathcal{I} \qquad \Gamma \vdash_{\mathtt{a}} c_i : \mathring{\tau} \qquad \Gamma \vdash_{\mathtt{s}} s_i : \mathtt{unit} \qquad \Gamma \vdash_{\mathtt{s}} s : \mathtt{unit}}{\Gamma \vdash_{\mathtt{s}} \mathtt{case}\ a\ \mathtt{with}\ \{c_i \to s_i\}_{i \in \mathcal{I}}\ \mathtt{otherwise}\ s : \mathtt{unit}}$$

$$\textsc{Ty-S-VarIn}\ \frac{\Gamma[x : \mathring{\tau}] \vdash_{\mathtt{s}} s : \mathtt{unit}}{\Gamma \vdash_{\mathtt{s}} \mathtt{var}\ x : \mathring{\tau}\ \mathtt{in}\ s : \mathtt{unit}}$$

FIGURE 5.21 Typage des instructions Mini-HDL

La règle Ty-S-Run vérifie le bon typage des appels de circuits externes. L'appel de l'instance i d'un circuit externe f est bien typé si f est bien associé à un type $(\mathtt{bool} * \mathring{\tau}) \xrightarrow{\mathbf{0}} (\mathring{\tau}' * \mathtt{bool})$ dans l'environnement de typage avec $\mathring{\tau}$ le type de l'argument effectif et $\mathring{\tau}'$ le type de la variable résultat. Les types $\mathring{\tau}$ et $\mathring{\tau}'$ doivent être des types de base. Les deux booléens correspondent aux paramètres `go` et `rdy` déjà évoqués dans le cadre de l'interopérabilité entre Eclat et VHDL (section 5.4).

La sémantique opérationnelle de MINI-HDL est définie par une relation $\mu, cs \vdash \Pi \Downarrow cs'$ qui se lit : "étant donnée une mémoire μ et un flot d'entrées cs, le programme Π engendre le flot le flot de sorties cs'". La relation $\Downarrow$ est définie par application de la relation d'évaluation des instructions, point à point sur les éléments du flot d'entrées. La relation $s/\mu \hookrightarrow_{\mathtt{s}} \mu'$ signifie "l'instruction s dans l'état mémoire μ s'évalue instantanément et produit un état mémoire μ'. La définition de $s/\mu \hookrightarrow_{\mathtt{s}} \mu'$ fait aussi intervenir une relation d'évaluation des atomes, notée $\mu \vdash a \hookrightarrow_{\mathtt{a}} c$ et qui se lit "dans l'état mémoire μ, l'atome a s'évalue instantanément en la constante c".

Sans perte de généralité, on suppose que tous les noms déclarés dans le programme MINI-HDL sont uniques. C'est une contrainte fondamentale en VHDL et, d'ailleurs, le schéma de compilation de CORE-ECLAT vers MINI-HDL le garantit. La sémantique comportementale de MINI-HDL respecte cet invariant.

La figure 5.22 définit les relations $\Downarrow$, $\hookrightarrow_{\mathtt{s}}$ et $\hookrightarrow_{\mathtt{a}}$ par un ensemble de règles d'inférence. Dans la règle REDUCE-PROG, le programme est appliqué instantanément, point à point, à chaque élément du flot d'entrée $v \bullet vs$, ce qui engendre un flot de sortie $v' \bullet vs'$ (de façon coinductive) ; l'état mémoire change potentiellement à chaque évaluation du programme Π, c'est-à-dire, à chaque cycle.

REDUCE-PROG

$$\frac{s/\mu[p \mapsto c][p' \mapsto d(\mathring{\tau}')] \hookrightarrow_{\mathtt{s}} \mu' \qquad \mu'(p') = c' \qquad \mu', cs \vdash \Pi \Downarrow cs'}{\mu, (c \bullet cs) \vdash \underbrace{\mathtt{let}\ f\ (p : \mathring{\tau})\ \mathtt{returns}\ (p' : \mathring{\tau}') = s}_{\Pi} \Downarrow (c' \bullet cs')}$$

REDUCE-ATOM

$$\frac{a/\mu \hookrightarrow c/\mu}{\mu \vdash a \hookrightarrow_{\mathtt{a}} c}$$

REDUCE-S-SEQ

$$\frac{s_1/\mu \hookrightarrow_{\mathtt{s}} \mu' \qquad s_2/\mu' \hookrightarrow_{\mathtt{s}} \mu''}{s_1\,;\ s_2/\mu \hookrightarrow_{\mathtt{s}} \mu''}$$

REDUCE-S-SKIP

$$()/\mu \hookrightarrow_{\mathtt{s}} \mu$$

REDUCE-S-SET

$$\frac{\mu \vdash a \hookrightarrow_{\mathtt{a}} c}{p := a/\mu \hookrightarrow_{\mathtt{s}} \mu[p \mapsto c]}$$

REDUCE-S-IF-TRUE

$$\frac{\mu \vdash a \hookrightarrow_{\mathtt{a}} \mathtt{true} \qquad s_1/\mu'}{\mathtt{if}\ x\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2/\mu \hookrightarrow_{\mathtt{s}} \mu'}$$

REDUCE-S-CASE

$$\frac{\mu \vdash a \hookrightarrow_{\mathtt{a}} c_j \qquad j \in \mathcal{I} \qquad s_j/\mu \hookrightarrow_{\mathtt{s}} \mu'}{\mathtt{case}\ x\ \mathtt{with}\ \{c_i \to s_i\}_{i \in \mathcal{I}}\ \mathtt{otherwise}\ s/\mu \hookrightarrow_{\mathtt{s}} \mu'}$$

REDUCE-S-IF-FALSE

$$\frac{\mu \vdash a \hookrightarrow_{\mathtt{a}} \mathtt{false} \qquad s_2/\mu \hookrightarrow_{\mathtt{s}} \mu'}{\mathtt{if}\ x\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2/\mu \hookrightarrow_{\mathtt{s}} \mu'}$$

REDUCE-S-CASE-OTHERWISE

$$\frac{\mu \vdash a \hookrightarrow_{\mathtt{a}} c_j \qquad j \notin \mathcal{I} \qquad s/\mu \hookrightarrow_{\mathtt{s}} \mu'}{\mathtt{case}\ x\ \mathtt{with}\ \{c_i \to s_i\}_{i \notin \mathcal{I}}\ \mathtt{otherwise}\ s/\mu \hookrightarrow_{\mathtt{s}} \mu'}$$

REDUCE-S-VARIN

$$\frac{s/\mu[x \mapsto d(\mathring{\tau})] \hookrightarrow_{\mathtt{s}} \mu'}{\mathtt{var}\ x : \mathring{\tau}\ \mathtt{in}\ s/\mu \hookrightarrow_{\mathtt{s}} \mu'}$$

REDUCE-S-RUN

$$\frac{\mathbf{INSTANCE}(f, i) = \langle \mathtt{let}\ f\ p : \mathring{\tau}\ \mathtt{returns}\ p' : \mathring{\tau}' = s \rangle \qquad \mathtt{var}\ p : \mathring{\tau}\ \mathtt{in}\ p := a\,;\ s\,;(x\,, rdy)\ \mathbf{:=}\ \mathbf{atom\text{-}of\text{-}pat}(p')/\mu \hookrightarrow_{\mathtt{s}} \mu'}{(x\,, rdy) := \mathtt{run}_i\ f\ a \hookrightarrow_{\mathtt{s}} \mu'}$$

FIGURE 5.22 Sémantique comportementale de MINI-HDL

La sémantique comportementale de MINI-HDL s'inspire fortement de celle d'ECLAT ; cela vise à faciliter le raisonnement (informel) sur l'encodage des constructions de programmation d'ECLAT en MINI-HDL (cf. le schéma de compilation de la section 5.8 page 108).

La règle REDUCE-ATOM spécifie qu'un atome a s'évalue en une constante c dans une mémoire μ si a, vu comme une expression ECLAT, se réduit en c et laisse la mémoire intacte. L'évaluation d'une déclaration de variable $x : \mathring{\tau}$ place instantanément x en mémoire, en l'associant à la constante par défaut $d(\mathring{\tau})$, puis évalue instantanément l'instruction dans la portée de la variable (règle REDUCE-S-VARIN). Cette règle fait l'hypothèse que les déclarations de variables, l'entrée et la sortie du programme introduisent des noms uniques (ce qui s'obtient par renommage préalable des variables du programme). L'évaluation de l'instruction `()` laisse la mémoire intacte (règle REDUCE-S-SKIP). L'évaluation de l'affectation met instantanément à jour la mémoire (règle REDUCE-S-SET). L'évaluation d'une séquence d'instructions évalue instantanément la première instruction puis la seconde instruction (règle REDUCE-S-SEQ). L'évaluation d'une conditionnelle évalue instantanément l'instruction *then* ou *else* en fonction de la condition (règles REDUCE-S-IF-TRUE et REDUCE-S-IF-FALSE). L'évaluation de l'instruction **`case/with`** sélectionne et évalue instantanément l'instruction correspondant à la valeur testée[12] (règle REDUCE-S-CASE) ou l'instruction donnée par la clause **`otherwise`** (règle REDUCE-S-CASE-OTHERWISE).

L'évaluation d'un appel à une instance i d'un circuit extérieur f suppose que le circuit f soit défini dans un environnement global $\mathbf{INSTANCE}(f, i)$ contenant des instances i de programmes extérieurs f (règle REDUCE-S-RUN). Chaque instance a son propre état mémoire, ce que l'on modélise avec la mémoire globale μ en supposant que les variables et liaisons introduites par chaque instance soient uniques. Dans cette règle, la fonction $\mathbf{atom\text{-}of\text{-}pat}(p)$ convertit un motif p (c'est-à-dire une imbrication de paires de variables) en un atome équivalent :

$$\begin{aligned} \mathbf{atom\text{-}of\text{-}pat}(x) &\stackrel{\text{def}}{=} x \\ \mathbf{atom\text{-}of\text{-}pat}(\ (p,p')\ ) &\stackrel{\text{def}}{=} (\ \mathbf{atom\text{-}of\text{-}pat}(p)\ ,\ \mathbf{atom\text{-}of\text{-}pat}(p')\ ) \end{aligned}$$

Cette section a présenté informellement comment programmer des automates en MINI-HDL, comment les imbriquer et comment les composer en parallèle dans le but d'encoder les traits de programmation d'ECLAT. La sémantique de MINI-HDL a été définie ; elle sert de référence pour compiler ECLAT. Dans la section suivante, nous présentons la traduction de MINI-HDL en VHDL. La correction de cette traduction est assurée par deux faits, à savoir :

(1) la très grande proximité syntaxique et sémantique entre MINI-HDL et VHDL ;

(2) la simplicité du sous-ensemble synthétisable de VHDL utilisé pour implanter MINI-HDL.

Le sous-ensemble mentionné dans le point **(2)** se limite à des expressions et des instructions VHDL, couplé à des règles de codage standards pour implanter une machine de Mealy en VHDL. Cela évacue *de facto* les aspects les plus complexe de la sémantique de VHDL [83] en se restreignant à un sous-ensemble impératif instantané classiquement décrit par une sémantique opérationnelle proche de celle que nous proposons pour MINI-HDL.

12. On suppose que les différents cas $c_i \to s_i$ sont mutuellement exclusifs, comme en VHDL.

5.6 Traduction de Mini-HDL en VHDL

Mini-HDL est essentiellement un sous-ensemble de VHDL pour de décrire de façon concise la fonction de transition d'une machine de Mealy telle que présentée au chapitre 2 (section 2.4).

5.6.1 Représentation des valeurs

Chaque constante de Mini-HDL (qui est aussi une constante Eclat) est représentable par un vecteur de bits VHDL de type `std_logic_vector` dont la taille $size(\mathring{\tau})$ doit être fixée statiquement (elle est inférée à partir du type de la constante). Les paires et les vecteurs sont traduits en appliquant l'opérateur `&` de concaténation de vecteurs de bits de VHDL.

$\mathcal{C}_{\mathsf{ty}}(\mathring{\tau}) \stackrel{\text{def}}{=}$ `std_logic_vector`$(0$ `to` $n)$ où $n = size(\mathring{\tau}) - 1$

$\mathcal{C}_{\mathsf{c}}$`(true)`	$\stackrel{\text{def}}{=}$	`"1"`	$size$`(bool)`	$\stackrel{\text{def}}{=}$	1
$\mathcal{C}_{\mathsf{c}}$`(false)`	$\stackrel{\text{def}}{=}$	`"0"`			
$\mathcal{C}_{\mathsf{c}}$`(())`	$\stackrel{\text{def}}{=}$	`"0"`	$size$`(unit)`	$\stackrel{\text{def}}{=}$	1
$\mathcal{C}_{\mathsf{c}}$`((`l`:int<`n`>))`	$\stackrel{\text{def}}{=}$	`"`$b_n b_{n-1} \cdots b_1$`"` où $b_n b_{n-1} \cdots b_1$ est la représentation du littéral entier l sur n bits	$size$`(int<`n`>)`	$\stackrel{\text{def}}{=}$	n
$\mathcal{C}_{\mathsf{c}}$`((`c`,`c'`))`	$\stackrel{\text{def}}{=}$	$\mathcal{C}_{\mathsf{c}}(c)$ `&` $\mathcal{C}_{\mathsf{c}}(c')$	$size(\mathring{\tau} * \mathring{\tau}')$	$\stackrel{\text{def}}{=}$	$size(\mathring{\tau}) + size(\mathring{\tau}')$
$\mathcal{C}_{\mathsf{c}}$`({`c_1`,` $\cdots$ c_n`}`	$\stackrel{\text{def}}{=}$	$\mathcal{C}_{\mathsf{c}}(c_1)$`&` $\cdots$ `&` $\mathcal{C}_{\mathsf{c}}(c_n)$	$size(\mathring{\tau}$ `vect<`n`>)`	$\stackrel{\text{def}}{=}$	$n \times size(\mathring{\tau})$

Figure 5.23 Traduction des types et constantes Mini-HDL vers VHDL

5.6.2 Traduction des expressions combinatoires

La traduction des atomes Mini-HDL est définie sur la figure 5.24. Chaque atome est traduit en une expression combinatoire VHDL, incluant les constantes et les opérateurs combinatoires. Chaque opérateur $\mathring{op}$ de Mini-HDL est traduit en un opérateur VHDL noté $[\![\mathring{op}]\!]$.

$\mathcal{C}_{\mathsf{a}}(c)$	$\stackrel{\text{def}}{=}$	$\mathcal{C}_{\mathsf{c}}(c)$
$\mathcal{C}_{\mathsf{a}}(x)$	$\stackrel{\text{def}}{=}$	x
$\mathcal{C}_{\mathsf{a}}(\mathring{op}\ a)$	$\stackrel{\text{def}}{=}$	$[\![\mathring{op}]\!$ `(` $\mathcal{C}_{\mathsf{a}}(a)$ `)`
$\mathcal{C}_{\mathsf{a}}$`((`a`,`a'`))`	$\stackrel{\text{def}}{=}$	$\mathcal{C}_{\mathsf{a}}(a)$ `&` $\mathcal{C}_{\mathsf{a}}(a')$

Figure 5.24 Traduction des atomes Mini-HDL vers VHDL

5.6.3 Traduction des instructions

Nous nous intéressons maintenant à la traduction des instruction Mini-HDL en VHDL. Chaque instruction Mini-HDL existe aussi en VHDL, à l'exception **(1)** des variables locales (`var/in`) d'une part, qui doivent être globalisées, et **(2)** des instanciations de circuits externes (`run`) d'autre part, qui requièrent un encodage spécifique (voir sous-section 5.6.5).

Pour globaliser les déclarations de variables **(1)**, nous définissons sur la figure 5.25 une fonction **collect-var-decl**(s) calculant l'ensemble des déclaration de variables $x : \mathring{\tau}$ introduites par la construction `var/in` dans l'instruction s. Cette fonction suppose que les déclarations locales ont toutes des noms distincts (ce qui s'obtient par un renommage préalable des variables), comme déjà supposé dans la sémantique de Mini-HDL.

$$
\begin{array}{rcl}
\textbf{collect-var-decl}(\texttt{()}) & \stackrel{\text{def}}{=} & \emptyset \\
\textbf{collect-var-decl}(\texttt{var}\ x : \mathring{\tau}\ \texttt{in}\ s) & \stackrel{\text{def}}{=} & \{x : \mathring{\tau}\} \cup \textbf{collect-var-decl}(s) \\
\textbf{collect-var-decl}(s\texttt{;}\ s') & = & \textbf{collect-var-decl}(s) \cup \textbf{collect-var-decl}(s') \\
\textbf{collect-var-decl}(\texttt{if}\ x\ \texttt{then}\ s\ \texttt{else}\ s') & \stackrel{\text{def}}{=} & \textbf{collect-var-decl}(s) \cup \textbf{collect-var-decl}(s') \\
\textbf{collect-var-decl}\left(\begin{array}{l}\texttt{case}\ x\ \texttt{with} \\ \left(c_i \rightarrow s_i\right)_{i\in\mathcal{I}} \\ \texttt{otherwise}\ s'\end{array}\right) & = & \left(\bigcup_{i\in\mathcal{I}} \textbf{collect-var-decl}(s_i)\right) \cup \textbf{collect-var-decl}(s') \\
\textbf{collect-var-decl}(p \coloneqq a) & \stackrel{\text{def}}{=} & \emptyset \\
\textbf{collect-var-decl}(\texttt{(}x\texttt{,}rdy\texttt{)} \coloneqq \texttt{run}_i\ f\ a) & \stackrel{\text{def}}{=} & \emptyset
\end{array}
$$

Figure 5.25 Collecte de l'ensemble des variables déclarées dans une instruction Mini-HDL

La figure 5.26 définit le schéma de traduction des instructions Mini-HDL vers VHDL. L'instruction `()` de Mini-HDL correspond à l'instruction `null` de VHDL. La séquence, l'affectation, la conditionnelle et la construction `case/with` de Mini-HDL sont identiques à des instructions VHDL (avec de légères différences dans la syntaxe concrète).

Les variables (`var/in`) sont globalisées en appliquant la méta-fonction **collect-var-decl**(.) sur le programme entier.

La traduction de la construction `((`x`,`rdy`)` $\coloneqq$ `run`$_i$ f a`)` de Mini-HDL utilise une convention de nommage. Le signal VHDL f`_argument_`i est destiné à transporter l'argument de l'instance i du circuit externe f. La variable f`_result_`i reçoit le résultat de cette même instance. Le signal f`_argument_rq_`i détermine à chaque cycle (via son premier bit à `0` ou `1`) si l'instance i du circuit f doit être exécutée ou non (c'est le fameux signal "go" présenté à la section 5.4). Au début de l'exécution du programme (à chaque cycle), le premier bit de f`_argument_rq_`i vaut `0` par défaut. Il passe à `1` lors de l'exécution de l'instruction `run`. À la fin de l'exécution (instantanée) du programme, le signal f`_argument_`i reçoit la valeur de f`_argument_rq_`i (sachant que les bit restant dans f`_argument_rq_`i code l'argument, voir figure 5.26).

$$
\begin{array}{rcl}
\mathcal{C}_{\mathsf{s}}(()) & \stackrel{\text{def}}{=} & \texttt{null;} \\
\mathcal{C}_{\mathsf{s}}(s;\ s') & \stackrel{\text{def}}{=} & \mathcal{C}_{\mathsf{s}}(s)\ \mathcal{C}_{\mathsf{s}}(s') \\
\mathcal{C}_{\mathsf{s}}(x := a) & \stackrel{\text{def}}{=} & x\ \texttt{:=}\ \mathcal{C}_{\mathsf{a}}(a)\texttt{;} \\
\mathcal{C}_{\mathsf{s}}((x_1, \cdots x_n) := (a_1, \cdots a_n)) & \stackrel{\text{def}}{=} & x_1\ \texttt{:=}\ \mathcal{C}_{\mathsf{a}}(a_1)\texttt{;}\ \cdots x_n\ \texttt{:=}\ \mathcal{C}_{\mathsf{a}}(a_n)\texttt{;} \\
\mathcal{C}_{\mathsf{s}}(\texttt{if}\ x\ \texttt{then}\ s\ \texttt{else}\ s') & \stackrel{\text{def}}{=} & \texttt{if}\ x\ \texttt{then}\ \mathcal{C}_{\mathsf{s}}(s)\ \texttt{else}\ \mathcal{C}_{\mathsf{s}}(s')\ \texttt{end if;} \\
\mathcal{C}_{\mathsf{s}}\left(\begin{array}{l}\texttt{case}\ x\ \texttt{with} \\ \mid c_1 \to s_1 \mid \cdots \\ \mid c_n \to s_n \\ \texttt{otherwise}\ s'\end{array}\right) & \stackrel{\text{def}}{=} & \left[\begin{array}{l}\texttt{case}\ x\ \texttt{is} \\ \quad \texttt{when}\ c_1\ \texttt{=>}\ \mathcal{C}_{\mathsf{s}}(s_1)\ \cdots \\ \quad \texttt{when}\ c_n\ \texttt{=>}\ \mathcal{C}_{\mathsf{s}}(s_n) \\ \quad \texttt{when others}\ \mathcal{C}_{\mathsf{s}}(s') \\ \texttt{end case;}\end{array}\right. \\
\mathcal{C}_{\mathsf{s}}(\texttt{var}\ x : \mathring{\tau}\ \texttt{in}\ s) & \stackrel{\text{def}}{=} & \mathcal{C}_{\mathsf{s}}(s) \\
\mathcal{C}_{\mathsf{s}}((x, rdy) := \texttt{run}_i\ f\ a) & \stackrel{\text{def}}{=} & \left[\begin{array}{l} f\texttt{_argument_rq_}i\ \texttt{:= "1" \&}\ \mathcal{C}_{\mathsf{a}}(a)\texttt{;} \\ x\ \texttt{:=}\ f\texttt{_result_}i\texttt{(1 to length'_result_}i\ \texttt{- 1);} \\ rdy\ \texttt{:=}\ f\texttt{_result_}i\texttt{();}\end{array}\right.
\end{array}
$$

FIGURE 5.26 Traduction des instructions MINI-HDL vers VHDL

5.6.4 Traduction du programme entier

La figure 5.27 définit le schéma de traduction général d'un programme MINI-HDL en VHDL. Cette traduction est un encodage de machine de Mealy suivant une règle de codage standard dite "*à deux processus*".

Un processus synchrone (lignes 16 à 23) est contrôlé par l'horloge globale `clk`. On utilise une convention de nommage pour désigner l'état courant `sig_`x_i et l'état suivant `next_`x_i. À chaque front montant de l'horloge (détecté à la ligne 20 par la primitive `rising_edge(clk)` de VHDL), l'état courant `sig_`x_i est mis-à-jour (ligne 21) avec le nouvel état `next_`x_i calculé par le programme au cycle précédent. Ce processus est réinitialisable de manière asynchrone par le signal `reset` (lignes 18 et 19) pour assigner à l'état (`sig_`x_i) sa valeurs par défaut $d(\mathring{\tau}_i)$, comme spécifié dans la sémantique de MINI-HDL.

Le calcul du nouvel état est réalisé par un processus combinatoire (lignes 25 à 35). Chaque signal `sig_`x_i formant l'état courant est placé dans une variable x_i (ligne 29). L'exécution du programme (ligne 31) met instantanément à jour les variables en fonction de l'entrée du circuit, appelée `argument` par convention. Les nouvelles valeurs des sorties sont assignées aux variables du motif p'. Ces variables sont connectées à la sortie du circuit, désigné par le signal `result` suivant notre convention de nommage (ligne 33). Ainsi, la sortie `result` dépend instantanément de l'entrée `argument`. Enfin, chaque variable x_i du programme est assignée au signal `next_`x_i associé pour définir le nouvel état qui remplacera l'état courant au *tick* suivant (ligne 21).

Les fonctions **`EXTRA-DÉCLARATIONS`**, **`EXTRA-CONNEXIONS`** et **`EXTRA-LISTE`** servent à instancier les circuits externes tandis que **`GO-DÉCLARATIONS`**, **`GO-ABSENT`** et **`GO-RQ`** servent à communiquer avec ceux-ci, en empêchant leur exécution si la construction **`run`** associée n'est pas exécutée au cours de ce cycle. Ces six fonctions sont présentées dans la sous-section suivante.

$\mathcal{C}_{\mathsf{ci}}\Big($ **let** $f\ (p : \mathring{\tau})$ **returns** $(p' : \mathring{\tau}')\ = s_{\mathsf{main}}\ \Big) \stackrel{\text{def}}{=}$

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity f in
5    port( signal clk      : in std_logic; -- horloge global
6          signal reset    : in std_logic; -- reset global
7          signal argument : in Cty(τ̊);      -- entrées
8          signal result   : out Cty(τ̊') );  -- sorties
9  end entity;
10 architecture rtl of f is
11   (signal sig_x_i : Cty(τ̊);)_(x_i:τ̊)∈V
12   (signal next_x_i : Cty(τ̊);)_(x_i:τ̊)∈V
13   EXTRA-DÉCLARATIONS(S)
14 begin
15   EXTRA-CONNEXIONS(S)
16   process(reset,clk)
17   begin
18     if reset = '1' then
19       (sig_x_i <= Ca(d(τ̊_i)) )_(x_i:τ̊_i)∈V
20     elsif rising_edge(clk) then
21       (sig_x_i <= next_x_i;)_(x_i:τ̊_i)∈V
22     end if;
23   end process;
24
25   process(argument, EXTRA-LISTE(S), sig_x_1, ··· sig_x_n)
26     (variable x_i : Cty(τ̊_i);)_(x_i:τ̊_i)∈V
27     GO-DÉCLARATIONS(S)
28   begin
29     (x_i := sig_x_i;)_(x_i:τ̊_i)∈V
30     GO-ABSENT(S)
31     Cs(S)
32     GO-RQ(S)
33     result <= Ca(atom-of-pat(p'));
34     (next_x_i <= x_i;)_(x_i:τ̊_i)∈V
35   end process;
36 end architecture;
```

où $\begin{cases} \mathcal{V} = \{x_1 : \mathring{\tau}_1,\ \cdots\ x_n : \mathring{\tau}_n\} \stackrel{\text{def}}{=} \textbf{collect-var-decl}(S) \\ S \stackrel{\text{def}}{=} \left[\begin{array}{l} \textbf{var}\ p : \mathring{\tau}\ \textbf{in} \\ p \coloneqq \texttt{argument}; \\ s_{\mathsf{main}} \end{array}\right. \end{cases}$

FIGURE 5.27 Traduction d'un programme MINI-HDL en VHDL

5.6.5 Instanciation des circuits externes

La fonction **EXTRA-DÉCLARATIONS(S)** utilisée figure 5.27 déclare, un à un, chaque circuit externe f : `(bool * `$\mathring{\tau}$`)` $\xrightarrow{\mathbf{0}}$ `(`$\mathring{\tau}'$ `* bool)` instancié par des occurrences de $(x, rdy) \coloneqq \mathbf{run}_i\ f\ a$ dans le corps du programme S. Ces déclarations sont de la forme :

```
1 component f port (clk : std_logic;
2                   reset : std_logic;
3                   argument : C_ty(bool * τ̊);
4                   result : C_ty(τ̊' * bool));
5 end component;
```

Elle définit aussi les déclarations des signaux utilisés pour communiquer avec chaque instance :

```
1 signal f_argument_i : C_ty(bool * τ̊);
2 signal f_result_i : C_ty(τ̊' * bool);
```

La fonction **EXTRA-LISTE(S)** déclare ces signaux dans la *liste de sensibilité* suivant les règles de codage standard VHDL (cf. section 2.2).

Les booléens associés à l'argument et au résultat correspondent aux entrées/sorties implicites `go` et `rdy` présentées à la section 5.4. Pour respecter la sémantique, chaque instance d'un circuit externe VHDL *devrait* geler son exécution à chaque fois que son entrée `go` (en l'occurrence le premier bit de f`_argument_`i) vaut `0` (cf. le circuit externe de la figure 5.13 défini directement en ECLAT pour être compilé en VHDL et appelé par un autre programme ECLAT).

La fonction **EXTRA-CONNEXIONS(S)** implante, une à une, chaque instance de fonction externe f_i appelée par les instructions $\mathbf{run}_i$ dans S :

```
1 f_i : component f port map (clk => clk,
2                             reset => reset,
3                             argument => f_argument_i,
4                             result => f_result_i);
```

Pour chaque instance i de chaque circuit externe f, la fonction **GO-ABSENT** met à `0` le premier bit (à gauche) de la variable f`_argument_rq_`i tandis que **GO-RQ** assigne à chaque signal f`_argument_`i la valeur de la variable f`_argument_rq_`i. Les variables f`_argument_rq_`i sont déclarées par la fonction **GO-DÉCLARATIONS**.

En résumé, MINI-HDL est un langage intermédiaire flot de données qui sert à décrire des machines de Mealy de façon précise et concise. C'est un langage d'instructions proche de VHDL, ce qui facilite la traduction des programmes MINI-HDL en VHDL. Les programmes MINI-HDL correspondent à des processus combinatoires VHDL réalisant la fonction de transition d'une machine de Mealy (figure 5.27, lignes 25 à 35). Le processus séquentiel associé qui met à jour l'état de la machine de Mealy (figure 5.27, lignes 16 à 23) est implicite : il est synthétisé à partir des variables du programmes. Cela vise à décrire en MINI-HDL des machines de Mealy bien formées par construction, tout en ayant une sémantique opérationnelle proche de celle d'ECLAT. Nous pouvons alors compiler en MINI-HDL un sous-ensemble normalisé d'ECLAT appelé CORE-ECLAT, puis traduire en VHDL le code MINI-HDL obtenu.

5.7 Core-Eclat : un sous-ensemble normalisé d'Eclat

Les programmes ECLAT bien typés ne sont pas compilés directement en VHDL : des transformations de programmes sont d'abord appliquées (voir section 5.9) et la sortie de ces transformations correspond à un sous-ensemble normalisé d'ECLAT appelé CORE-ECLAT. Ce langage intermédiaire est ensuite compilé en MINI-HDL, lui même étant traduit en VHDL.

La syntaxe de CORE-ECLAT est définie sur la figure 5.28. C'est un langage d'expressions *monomorphe* : les fonctions ne peuvent pas être appelées avec des arguments de types différents. Ce langage est aussi *de premier ordre* : les fonctions CORE-ECLAT ne prennent pas de fonctions en arguments et, de même qu'en ECLAT, elle ne retournent pas de fonctions.

programme normalisé	$\mathring{\pi}$	$::= \texttt{let}\ f = \texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau}' \rightarrow \mathcal{A}$
automate	$\mathcal{A}$	$::= \mathring{e}$
expression normalisée	$\mathring{e}$	$::= a$
appel externe ou récursif		$\mid f\ a$
liaison locale		$\mid \texttt{let}\ p{:}\mathring{\tau} = \mathring{e}\ \texttt{in}\ \mathring{e}$
récursion locale		$\mid \texttt{let rec}\ f\ (p{:}\mathring{\tau}) = \mathring{e}\ \texttt{in}\ \mathring{e}$
conditionnelle		$\mid \texttt{if}\ a\ \texttt{then}\ \mathring{e}\ \texttt{else}\ \mathring{e}$
paire parallèle		$\mid (\mathcal{A}_1 \| \mathcal{A}_2)$
registre		$\mid \texttt{reg (fun}\ (x{:}\mathring{\tau})\ :\ \mathring{\tau} \rightarrow \mathring{e})\ \texttt{init}\ a$
bloc exec		$\mid \texttt{exec}\ \mathcal{A}\ \texttt{default}\ a\ \texttt{reset}\ a$
atome	a	$::= x \mid c \mid (a,a) \mid op\ a$
motif	p	$::= x$
constante	c	$::= () \mid \texttt{true} \mid \texttt{false} \mid (l : \texttt{int<}\mathring{\theta}\texttt{>}) \mid (c,c) \mid \{c_1, \cdots c_n\}$
opérateur combinatoire	$\mathring{o}p$	$::= \texttt{not} \mid \texttt{\&} \mid \texttt{or} \mid \texttt{xor} \mid \texttt{fst} \mid \texttt{snd}$
		$\mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{mod} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{resize_int<}\mathring{\theta}\texttt{>}$
		$\mid \texttt{vect_create<}\mathring{\theta}\texttt{>} \mid \texttt{vect_nth} \mid \texttt{vect_copy_with} \mid \texttt{vect_size}$
type de base	$\mathring{\tau}$	$::= \texttt{unit} \mid \texttt{bool} \mid \texttt{int<}\mathring{\theta}\texttt{>} \mid \mathring{\tau} \mid \mathring{\tau}\ \texttt{*}\ \mathring{\tau} \mid \texttt{vect<}\mathring{\theta}\texttt{>}$
taille	$\mathring{\theta}$	$::= n$
schéma de type normalisé	$\mathring{\sigma}$	$::= \mathring{\tau} \mid \mathring{\tau} \xrightarrow{\delta} \mathring{\tau}$

FIGURE 5.28 Syntaxe de CORE-ECLAT

Tous les motifs sont réduits à des variables, la déstructuration de motif est réalisée explicitement via les opérateurs de projections de paires `fst` et `snd`.

CORE-ECLAT est un sous-ensemble d'ECLAT : tout programme CORE-ECLAT est un programme ECLAT et a la même sémantique (section 3). Le système de types de CORE-ECLAT découle de celui d'ECLAT (chapitre 3) mais tout schéma de type lié dans l'environnement de typage doit être soit un type de base $\mathring{\tau}$ soit un type fonctionnel de la forme $\mathring{\tau} \xrightarrow{\delta} \mathring{\tau}'$ où $\mathring{\tau}$ et $\mathring{\tau}'$ sont des types de base. Cette restriction vient de l'impossibilité de représenter les valeurs fonctionnelles en pratique dans les circuits engendrés (voir 5.2.6 page 86).

La classe a des atomes est équivalente à celle introduite en MINI-HDL (tout en étant un sous-ensemble du langage d'expressions d'ECLAT).

Une nouvelle catégorie syntaxique $\mathcal{A}$ appelée *automate* est introduite ; c'est une expression qui ne comporte pas d'occurrences de noms de fonctions libres. Chaque automate $\mathcal{A}$ dans le programme CORE-ECLAT correspond à un bloc `automaton`$_{\mathcal{L}}$ dans le programme MINI-HDL engendré à partir de $\mathcal{A}$. L'absence de noms de fonctions libres vise à simplifier la compilation de CORE-ECLAT vers MINI-HDL ; en effet, cela évite l'ajout d'un environnement de compilation associant à ces noms fonctions leurs définitions. Un automate $\mathcal{A}$ est soit le corps d'un programme, soit une expression calculée sous un `exec`, soit une expression dans l'une des branches de la construction parallèle $(\mathcal{A}_1 \| \mathcal{A}_2)$.

La classe $\mathring{op}$ désigne les opérateurs combinatoires d'ECLAT. Au chapitre 6, nous introduirons de nouveaux opérateurs qui ne seront pas combinatoires ; d'où la distinction faite entre opérateur op d'ECLAT et opérateur combinatoire $\mathring{op}$ de CORE-ECLAT et MINI-HDL.

Le schéma de compilation présenté à la section suivante suppose que tout nom de liaison (introduit par `fun`, `let/in`, `let/rec/in` ou `reg`) est unique. Le programme CORE-ECLAT doit être renommé au préalable, avec des noms uniques, en préservant la portée lexicale.

5.8 Compilation de Core-Eclat vers Mini-HDL

La compilation de CORE-ECLAT vers MINI-HDL constitue le point central du compilateur ECLAT. C'est cette étape qui réalise les choix d'implantation motivés à la section 5.2 (page 78).

Rappelons qu'un *automate* $\mathcal{A}$ est une classe syntaxique de CORE-ECLAT (voir figure 5.28) tandis qu'un bloc `automaton` est une instruction MINI-HDL (figure 5.17 page 95).

Le schéma de compilation proposé est défini par induction sur la structure des programmes CORE-ECLAT. Il met en œuvre un jugement principal et deux jugements auxiliaires, qui seront détaillés par la suite, et dont voici les principales caractéristiques :

— Le compilé d'un programme $\mathring{\pi}$ s'obtient en compilant son corps $\mathcal{A}$ (sous-section 5.8.1) ;

— Le compilé d'un automate $\mathcal{A}$ est un bloc `automaton` (sous-section 5.8.2) ;

— Le compilé d'une expression $\mathring{e}$ dans un automate $\mathcal{A}$ est une instruction MINI-HDL qui écrit le résultat du calcul de $\mathring{e}$ dans une variable destination (sous-section 5.8.3). Cela ressemble aux techniques de compilation SSA (*static single assignment*) [163].

À cela s'ajoute une étape de compilation supplémentaire (sous-section 5.8.4) qui gère le retour des appels de fonctions.

Chaque définition de fonction récursive terminale f dans un automate $\mathcal{A}$ correspond à un état f dans le compilé de $\mathcal{A}$ (qui est un bloc `automaton`). Les arguments de la fonction f sont stockés dans une variable spéciale appelée par convention f`_arg`. Les appels à f correspondent à des transitions vers l'état f. La valeur de retour de l'appel à f est écrit dans une variable spéciale appelée par convention f`_result`. Pour revenir à l'appelant, nous utilisons une variable spéciale appelée par convention f`_id` et qui identifie (par un entier) la continuation du site d'appel. Une continuation est une expression CORE-ECLAT compilée en MINI-HDL. Par exemple, étant donnée une variable destination y et une continuation k, la continuation de l'appel à f dans l'expression `let x =` f `10 in x + 1` est l'instruction `x` $\coloneqq$ f`_result; y` $\coloneqq$ `x + 1;` k.

Par convention, l'instruction $\texttt{continue}_{\mathcal{L}}\ f^\sharp$ (où f est annoté avec le symbole $\sharp$) désigne la continuation numéro f`_id` de la fonction f, où $\mathcal{L}$ est le nom du bloc $\texttt{automaton}_{\mathcal{L}}$ directement englobant la définition de la fonction f (c'est à dire celui qui contient l'état f). Cette continuation est construite après la compilation puis stockée dans le code compilé sous la forme d'une "table de sauts" sélectionnant dynamiquement la bonne continuation en fonction de la valeur de f`_id`. Les tables de sauts [13] sont construites à l'aide de la fonction $\textbf{dispatch}[\![(\cdot)]\!]_{(\cdot)}$ définie à la sous-section 5.8.4, laquelle remplace chaque occurrence de $\texttt{continue}_{\mathcal{L}}\ f^\sharp$ par l'instruction :

$$\begin{array}{l}\texttt{case}\ f\texttt{_id}\ \texttt{with}\\ |\ n_1 \to s_1\ \cdots\\ |\ n_m \to s_m\\ \texttt{otherwise}\ \texttt{()}\end{array}$$

où s_j est la continuation du site d'appel de f numéro n_j. Cela sert à implanter les retours des appels de fonctions.

5.8.1 Compilation des programmes

Le jugement $\vdash \mathcal{C}[\![\mathring{\pi}]\!] \rightsquigarrow \Pi$ définit le compilé d'un programme $\mathring{\pi}$. Il signifie : "le compilé de $\mathring{\pi}$ est Π". Il est défini par la règle suivante où $\mathcal{L} \vdash \mathcal{C}[\![\mathcal{A}]\!] \rightsquigarrow s$ est le jugement de compilation des automates (voir sous-section 5.8.2).

COMPILE-MAIN

$$\frac{\mathcal{L} \text{ est un nom frais} \qquad \mathcal{L} \vdash \mathcal{C}[\![\mathcal{A}]\!] \rightsquigarrow s \qquad \mathcal{FV}(s) \stackrel{\text{def}}{=} \{x_1 : \mathring{\tau}_1, \cdots x_n : \mathring{\tau}_n\} \qquad s' \stackrel{\text{def}}{=} \texttt{var}\ x_1 : \mathring{\tau}_1\ \texttt{in}\ \cdots\ \texttt{var}\ x_n : \mathring{\tau}_n\ \texttt{in}\ (s\ \texttt{; result} \coloneqq \texttt{result}_{\mathcal{L}})}{\vdash \mathcal{C}[\![\texttt{let}\ f = \texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau}' \to \mathcal{A}]\!] \rightsquigarrow \texttt{let}\ f\ (p{:}\mathring{\tau})\ \texttt{returns}\ (\texttt{result}{:}\mathring{\tau}') = s'}$$

Le corps $\mathcal{A}$ du programme est compilé en une instruction s (qui est un bloc `automaton`, voir sous-section 5.8.2) qui assigne un résultat dans une variable spéciale appelée par convention $\texttt{result}_{\mathcal{L}}$; puis la valeur de cette variable est assignée à la sortie du programme, appelée par convention `result`. Les variables libres (et leurs types) dans s (notées $\mathcal{FV}(s)$) doivent être déclarées en amont.

13. Le terme *table de continuations* désigne une structure de données utilisées par le compilateur pour collecter des fragments de code engendré (ou *continuations*) au cours de la compilation. Une *table de sauts* désigne une instruction dans le code engendré et qui est obtenue à partir d'informations collectées dans une table de continuations.

5.8.2 Compilation des automates

Le jugement $\mathcal{L} \vdash \mathcal{C}[\![\mathcal{A}]\!] \rightsquigarrow s$ définit le compilé d'un automate $\mathcal{A}$. Il signifie : "étant donné un nom $\mathcal{L}$, l'automate $\mathcal{A}$ est compilé en une instruction s ; cette instruction est un bloc $\texttt{automaton}_{\mathcal{L}}$ identifié par $\mathcal{L}$ et dont l'état initial est $\texttt{Idle}_{\mathcal{L}}$". Ce jugement est défini par la règle suivante qui fait intervenir à la fois le jugement $\rho, \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s, ts/\omega$ (sous-section 5.8.3) et la fonction $\textbf{dispatch}[\![s_i]\!]_\omega$ qui traite le retour instantané des appels de fonctions (sous-section 5.8.4).

COMPILE-AUTOMATON

$$\frac{\mathcal{A} \text{ est une expression } \mathring{e} \qquad \mathcal{L} \text{ est un nom frais} \qquad k \stackrel{\text{def}}{=} (\texttt{rdy}_{\mathcal{L}} \coloneqq \texttt{true};\ \texttt{continue}_{\mathcal{L}}\ \texttt{Idle}_{\mathcal{L}}) \qquad \emptyset, \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}\ \textbf{TO}\ \texttt{result}_{\mathcal{L}}\ \textbf{THEN}\ k\ \textbf{IS}\ s, \{f_1 : s_1, \cdots\ f_n : s_n\}/\omega \qquad s^\star \stackrel{\text{def}}{=} \textbf{dispatch}[\![s]\!]_\omega \qquad s_i^\star \stackrel{\text{def}}{=} \textbf{dispatch}[\![s_i]\!]_\omega \qquad i \in \{1, \cdots n\}}{\vdash \mathcal{C}[\![\mathcal{A}]\!] \rightsquigarrow \texttt{automaton}_{\mathcal{L}}\ \texttt{|}\ \texttt{Idle}_{\mathcal{L}} \to \texttt{rdy}_{\mathcal{L}} \coloneqq \texttt{false};\ s^\star\ \texttt{|}\ f_1 \to s_1^\star \cdots\ \texttt{|}\ f_n \to s_n^\star\ \texttt{end}}$$

Cette règle a l'avantage de donner une vue d'ensemble de la compilation de CORE-ECLAT vers MINI-HDL. Elle repose toutefois sur des notions nombreuses qui seront introduites progressivement à la sous-section 5.8.3. C'est pourquoi nous en donnons une explication intuitive.

L'automate $\mathcal{A}$ est par définition une expression $\mathring{e}$. En compilant $\mathring{e}$, nous obtenons une instruction s "à exécuter immédiatement", un ensemble de transitions $\{f_1 : s_1, \cdots\ f_n : s_n\}$ correspondant aux n déclarations de fonctions récursives présentes dans $\mathcal{A}$ et une table de continuations ω pour gérer le retour instantané des appels de fonctions. À partir de ces éléments, nous formons un bloc $\texttt{automaton}_{\mathcal{L}}$ qui contient :

— une transition reliant l'état initial $\texttt{Idle}_{\mathcal{L}}$ à l'instruction s ;

— les n transitions $f_i \to s_i$ obtenues en compilant l'expression $\mathring{e}$.

Le nom $\mathcal{L}$ sert à identifier différents éléments dans le bloc $\texttt{automaton}_{\mathcal{L}}$ (comme $\texttt{Idle}_{\mathcal{L}}$ par exemple) en suivant une convention de nommage. Les calculs ECLAT redémarrent lorsqu'ils se terminent ; c'est pourquoi la compilation $\mathring{e}$ est paramétrée par une continuation k qui effectue un branchement vers l'état initial via l'instruction $\texttt{continue}_{\mathcal{L}}\ \texttt{Idle}_{\mathcal{L}}$. Une variable spéciale appelée par convention $\texttt{rdy}_{\mathcal{L}}$ doit valoir `true` lorsque le calcul se termine ; c'est pourquoi, la transition sortant de l'état $\texttt{Idle}_{\mathcal{L}}$ assigne la valeur `false` à $\texttt{rdy}_{\mathcal{L}}$ tandis que la continuation k qui revient dans l'état $\texttt{Idle}_{\mathcal{L}}$ lui assigne la valeur `true`. À la manière de SSA, la compilation de $\mathring{e}$ écrit un résultat dans une variable ; cette variable s'appelle $\texttt{result}_{\mathcal{L}}$ par convention.

Le bloc $\texttt{automaton}_{\mathcal{L}}$ obtenu doit encore être complété (c'est le passage de s à $s^\star$ et s_i à $s_i^\star$). En effet, les retours des appels de fonctions récursives d'ECLAT doivent rendre le contrôle instantanément à l'appelant, mais cela n'est pas réalisable par le jugement de compilation des expressions (qui est dirigé par la syntaxe) car toute transition exécutée dans l'automate prend un cycle. C'est pourquoi le jugement collecte dans une *table de continuations* ω les continuations des sites d'appels, absentes dans s et dans les s_i ; pour être insérées dans un second temps. Du fait que chaque appel à une fonction récursive terminale f est partagé, cette insertion nécessite un *dispatch* pour sélectionner la bonne continuation de f à l'exécution en fonction de la variable $f\texttt{_id}$ ayant sauvegardé l'identifiant du site d'appel (voir sous-section 5.8.4).

5.8.3 Compilation des expressions

Le jugement $\rho, \mathcal{L} \vdash$ **COMPILE** $\mathring{e}$ **TO** x **THEN** k **IS** $s, ts/\omega$ définit le compilé d'une expression $\mathring{e}$. Il signifie : "dans l'environnement de compilation ρ, le compilé de $\mathring{e}$ (qui écrit son résultat dans x puis exécute k) est un triplet formé d'une instruction s, d'un ensemble de transitions ts et d'une table de continuations ω pour construire le bloc `automaton`$_{\mathcal{L}}$ englobant identifié par $\mathcal{L}$".

L'environnement de compilation ρ est une fonction partielle qui associe à un nom de fonction récursive f une marque $\rho(f) \in \{\mathsf{tail\text{-}call}, \mathsf{direct}\}$ qui indique si, dans l'expression $\mathring{e}$ à compiler, les occurrences de f sont des appels récursifs ou des appels directs.

L'ensemble de transitions ts sert à construire explicitement l'encodage de l'automate $\mathcal{A}$ dans lequel se situe l'expression $\mathring{e}$ à compiler. La table de continuations ω sert à implanter le retour des appels de fonctions en cas de partage (voir sous-section 5.2.4 page 83) avec synchronisation instantanée comme imposé par la sémantique d'ECLAT (voir sous-section 5.2.5 page 85).

La table de continuations vide est notée $\emptyset$. Une table de continuations ω est une fonction partielle qui associe à un nom de fonction f un ensemble indexé de continuations possibles. Cet ensemble est une fonction partielle qui associe à un entier n (ou *numéro d'instance*) une instruction MINI-HDL (ou *continuation*). Chaque numéro d'instance correspond à un site d'appel de fonction récursives dans le programme CORE-ECLAT à compiler. La n-ième continuation de la fonction f dans la table ω est notée $\omega(f \cdot n)$. L'extension de la table ω avec la continuation $n \mapsto s$ de la fonction f est notée $\omega[f \cdot n \mapsto s]$. La fusion de tables de continuations est notée $\omega_1 \mathbin{@} \omega_2$. Elle est définie comme suit : si $f \cdot n$ est associé à une continuation s dans ω_1 ou dans ω_2, alors $f \cdot n$ est associé à s dans $\omega_1 \mathbin{@} \omega_2$. La table $\omega_1 \mathbin{@} \omega_2$ suppose que si $f \cdot n$ est liée dans ω_1 (*resp.* ω_2), alors il n'est pas lié dans ω_2 (*resp.* ω_1).

Le compilé d'un atome a vers x suivi de k est une séquence qui assigne à x l'atome a puis exécute immédiatement l'instruction k.

COMPILE-ATOM

$$\rho, \mathcal{L} \vdash \textbf{COMPILE } a \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } (x := a;\ k), \emptyset/\emptyset$$

Le compilé d'un appel récursif terminal $f(a)$ vers x suivi de k (figure 5.29) assigne l'argument a au paramètre formel de f (nommé f`_arg` par convention) puis continue dans l'état f en ignorant la continuation k.

COMPILE-TAIL-CALL

$$\frac{\rho(f) = \mathsf{tail\text{-}call} \quad \texttt{(* la continuation k est ignorée *)} \qquad s \overset{\text{def}}{=} \left[\begin{array}{l} f\texttt{_arg} := a; \quad \texttt{(* l'argument est sauvegardé dans f_arg *)} \\ \texttt{continue}_{\mathcal{L}}\ f \quad \texttt{(* l'automate passe dans l'état f, i.e., pause d'un cycle *)} \end{array}\right.}{\rho, \mathcal{L} \vdash \textbf{COMPILE } f\ a \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s, \emptyset/\emptyset}$$

FIGURE 5.29 Compilation d'un appel récursif terminal

Le compilé d'un appel direct $f(a)$ vers x suivi de k (figure 5.30) assigne l'argument a au paramètre formel de f, (f`_arg` par convention) puis assigne un identifiant n unique à la variable désignant l'instance courante de f (f`_id` par convention) puis continue dans l'état f. La continuation k est précédée de l'instruction $x \coloneqq f$`_result` (où f`_result` désigne par convention le résultat de l'appel de fonction) ; l'instruction obtenue est rangée à l'emplacement $f \cdot n$ dans la table de continuations en cours de construction.

COMPILE-DIRECT-CALL

$$\rho(f) = \mathsf{direct} \qquad n \text{ est un identifiant unique}$$

$$k_f \stackrel{\text{def}}{=} \left[\begin{array}{l} \texttt{(* la continuation du site d'appel récupère le résultat f_result *)} \\ x \coloneqq f\texttt{_result}; \ \texttt{(* et écrit celui-ci dans la variable destination x *)} \\ k \ \texttt{(* puis exécute la continuation k *)} \end{array}\right.$$

$$\omega \stackrel{\text{def}}{=} \{f : \{n : k_f\}\} \qquad \texttt{(* la continuation de f est associée au numéro n *)}$$
$$\texttt{(* du site d'appel dans la table de continuations *)}$$

$$s \stackrel{\text{def}}{=} \left[\begin{array}{l} f\texttt{_arg} \coloneqq a; \ \texttt{(* l'argument est sauvegardé dans f_arg *)} \\ f\texttt{_id} \coloneqq n; \ \texttt{(* le numéro du site d'appel n est sauvegardé dans f_id *)} \\ \texttt{continue}_{\mathcal{L}}\ f \ \texttt{(* l'automate passe dans l'état f, i.e., pause d'un cycle *)} \end{array}\right.$$

$$\rho, \mathcal{L} \vdash \textbf{COMPILE } f\ a \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s, \emptyset/\omega$$

FIGURE 5.30 Compilation de l'appel direct à une fonction récursive

Le compilé d'un appel de fonction externe $f(a)$ vers x suivi de k (figure 5.31) communique avec l'instance i du circuit externe f (via l'instruction $\texttt{run}_i$).

COMPILE-EXTERNAL-CALL

$$f \notin \mathsf{dom}(\rho) \qquad i \text{ est un identifiant frais} \qquad \textit{sync}, \textit{result} \text{ et } \textit{rdy} \text{ sont des noms frais}$$

$$s \stackrel{\text{def}}{=} \left[\begin{array}{l} (\textit{result}, \textit{rdy}) \coloneqq \texttt{run}_i\ f\ a; \ \texttt{(* communique avec l'instance i de f *)} \\ \texttt{(* le circuit retourne un résultat (result) et un sortie *)} \\ \texttt{(* additionnelle rdy. Si rdy est vrai, le calcul se termine *)} \\ \texttt{if } \textit{rdy} \texttt{ then } (x \coloneqq \textit{result};\ k) \ \texttt{(* x reçoit résult, on continue avec k *)} \\ \texttt{else continue}_{\mathcal{L}}\ \textit{sync} \ \texttt{(* sinon, l'automate reste dans l'état sync *)} \end{array}\right.$$

$$s_0 \stackrel{\text{def}}{=} s \quad \texttt{(* l'instruction de démarrage est une copie de s (duplication) *)}$$
$$\texttt{(* pour commence l'appel immédiatement. NB : l'appel peut être instantané *)}$$

$$\rho, \mathcal{L} \vdash \textbf{COMPILE } f\ a \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s_0, \{\textit{sync} : s\}/\emptyset$$

FIGURE 5.31 Compilation d'un appel de fonction externe

Toute fonction f qui n'est pas liée dans l'environnement d'exécution est considérée externe. Le fonction f peut être instantanée ou non, mais son implantation est un circuit qui, par définition, produit instantanément des sorties. C'est pourquoi, par convention, un tel circuit externe à une sortie *rdy* indiquant le cycle où $f(a)$ se termine. Ce booléen n'apparaît pas en ECLAT dans la déclaration du type de f. L'appel $f(a)$ être instantané (d'après la sémantique). L'au-

tomate engendré sonde la sortie `rdy`. Pour que la communication commence dès le démarrage de l'appel, la portion de circuit réalisant le sondage est dupliquée.

Le compilé d'une définition de fonction **let rec** $f\ y = \mathring{e}_1$ **in** $\mathring{e}_2$ vers x suivi de k est le compilé de $\mathring{e}_2$ avec un ensemble de transitions étendu par le compilé de e_1 où le paramètre y est substitué par f`_arg`, ce qui est noté $\mathring{e}_1[f\texttt{_arg} \mapsto y]$ comme au chapitre 4.

Le compilé de $\mathring{e}_1[f\texttt{_arg} \mapsto y]$ dont il est question (appelons le s_1) est obtenu avec un environnement de compilation enrichi $\rho \cup \{f : \textsf{tail-call}\}$ (qui indique que tout appel à f depuis $\mathring{e}_1$ est récursif terminal). La variable destination est par convention f`_result` (c'est-à-dire la variable de retour de la fonction f). La continuation n'est pas encore connue ; elle est donc remplacée par l'instruction $\texttt{continue}_{\mathcal{L}}\ f^\sharp$ où la marque $f^\sharp$ désigne un point de *dispatch* de l'ensemble des continuations associées à f dans la table de continuations qui sera connue à la fin de compilation de l'automate courant. Finalement, l'état f (c'est-à-dire le nom de la fonction récursive terminale) est associé au l'instruction compilée s_1 ; la transition obtenue est rangée dans l'ensemble de transitions en cours de construction.

COMPILE-FUNREC

$$\frac{\begin{array}{c} k_f \stackrel{\text{def}}{=} \texttt{continue}_{\mathcal{L}}\ f^\sharp \ \texttt{(* retour au site d'appel courant *)} \\ (\rho[f : \textsf{tail-call}]), \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}_1[f\texttt{_arg} \mapsto y]\ \textbf{TO}\ f\texttt{_result}\ \textbf{THEN}\ k_f\ \textbf{IS}\ s_1, ts_1/\omega_1 \\ \rho[f : \textsf{direct}], \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}_2\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s_2, ts_2/\omega_2 \end{array}}{\rho, \mathcal{L} \vdash \textbf{COMPILE}\ \texttt{let rec}\ f(y{:}\mathring{\tau}) = \mathring{e}_1\ \texttt{in}\ \mathring{e}_2\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s_2, ts_1 \cup ts_2 \cup \{f : s_1\}/\omega_1 @ \omega_2}$$

Le compilé d'une liaison locale **let** $y = \mathring{e}_1$ **in** $\mathring{e}_2$ vers x suivi de k est le compilé de $\mathring{e}_1$ vers y suivi du compilé de $\mathring{e}_2$ vers x suivi de k. La variable est déclarée globalement par la règle COMPILE-MAIN (sous-section 5.8.1) à la fin du processus de compilation (on rappelle que tous les noms dans le programme CORE-ECLAT à compiler sont uniques).

COMPILE-LET-VAR

$$\frac{\begin{array}{c} \rho, \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}_2\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s_2, ts_2/\omega_2 \\ \rho, \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}_1\ \textbf{TO}\ y\ \textbf{THEN}\ s_2\ \textbf{IS}\ s_1, ts_1/\omega_1 \qquad (\mathring{e}_1 \text{ n'est pas un atome}) \end{array}}{\rho, \mathcal{L} \vdash \textbf{COMPILE}\ \texttt{let}\ y{:}\mathring{\tau} = \mathring{e}_1\ \texttt{in}\ \mathring{e}_2\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s_1, ts_1 \cup ts_2/\omega_1 @ \omega_2}$$

Comme cas particulier de cette dernière règle, le compilé d'une liaison locale **let** $y = a$ **in** $\mathring{e}$ vers x suivi de k est une affectation de l'atome a à la variable y immédiatement suivie par le compilé de $\mathring{e}$ vers x suivi de k.

COMPILE-LET-VAR-ATOM

$$\frac{\rho, \mathcal{L} \vdash \textbf{COMPILE}\ \mathring{e}\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s, ts/\omega}{\rho, \mathcal{L} \vdash \textbf{COMPILE}\ \texttt{let}\ y{:}\mathring{\tau} = a\ \texttt{in}\ \mathring{e}\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ (y \coloneqq a;\ s), ts/\omega}$$

Le compilé d'une conditionnelle vers x suivi de k est une conditionnelle combinant les compilés des branches *then* et *else* vers x suivis de k. Cela a pour conséquence de dupliquer la continuation, comme discuté à la sous-section 5.5.2.

COMPILE-IF

$$\frac{\rho, \mathcal{L} \vdash \textbf{COMPILE } \mathring{e}_i \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s_i, ts_i/\omega_i \qquad i \in \{1,2\} \qquad s \stackrel{\text{def}}{=} (\texttt{if } a \texttt{ then } s_1 \texttt{ else } s_2)}{\rho, \mathcal{L} \vdash (\textbf{COMPILE } \texttt{if } a \texttt{ then } \mathring{e}_1 \texttt{ else } \mathring{e}_2 \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s, ts_1 \cup ts_2/\omega_1@\omega_2}$$

On peut optimiser cette règle pour éviter la duplication de la continuation k lorsque les deux branches sont instantanées (COMPILE-IF-OPTIM).

COMPILE-IF-OPTIM

$$\frac{\begin{array}{c}s_1 \text{ et } s_2 \text{ sont instantanées} \qquad \rho, \mathcal{L} \vdash \textbf{COMPILE } \mathring{e}_i \textbf{ TO } x \textbf{ THEN } () \textbf{ IS } s_i, ts_i/\omega_i \\ i \in \{1,2\} \qquad s \stackrel{\text{def}}{=} (\texttt{if } a \texttt{ then } s_1 \texttt{ else } s_2)\texttt{; } k\end{array}}{\rho, \mathcal{L} \vdash (\textbf{COMPILE } \texttt{if } a \texttt{ then } \mathring{e}_1 \texttt{ else } \mathring{e}_2 \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s, ts_1 \cup ts_2/\omega_1@\omega_2}$$

Dans le cas général, lorsque l'une des deux expressions est non-instantanée, elle comprend des appels récursifs *terminaux* qui *coupent* la continuation (règle COMPILE-TAIL-CALL) ; cette règle alternative n'est alors pas applicable (car elle change le comportement du programme).

Le compilé d'un registre `(reg (fun` $y \to \mathring{e}$`) init` a`)` vers x suivi de k (figure 5.32) initialise y à la valeur de a au démarrage, puis exécute le compilé de $\mathring{e}$ vers y (dès le démarrage).

COMPILE-REG

$$\frac{\rho, \mathcal{L} \vdash \textbf{COMPILE } \mathring{e} \textbf{ TO } y \textbf{ THEN } () \textbf{ IS } s, \emptyset/\emptyset \qquad \textit{started} \text{ est un nom frais}}{\rho, \mathcal{L} \vdash \textbf{COMPILE } \texttt{reg (fun } (y\texttt{:}\mathring{\tau}\texttt{) : } \mathring{\tau} \to \mathring{e}\texttt{) init } a \textbf{ TO } x \textbf{ THEN } k \textbf{ IS } s', \emptyset, \emptyset}$$

$$s' \stackrel{\text{def}}{=} \left[\begin{array}{l}\texttt{(* initialise y au début, i.e., quand started vaut false par défaut *)} \\ \texttt{(if } \textit{started} \texttt{ then () else } (y \coloneqq a\texttt{; } \textit{started} \coloneqq \texttt{true})\texttt{); (* puis started à true *)} \\ s\texttt{; (* exécute l'instruction de mise à jour de y en fonction de y *)} \\ x \coloneqq y\texttt{; } k \texttt{ (* destination x à la valeur y du registre, puis exécute k *)}\end{array}\right.$$

FIGURE 5.32 Compilation de `reg`

Le compilé de `exec` $\mathcal{A}$ `default` a_{default} `reset` a_{reset} ciblant x avec la continuation k (défini figure 5.33) compile un automate $\mathcal{A}'$ obtenu en substituant dans $\mathcal{A}$ les variables libres (notées $\mathcal{FV}(\mathcal{A})$) par des noms frais x'_i. Ce renommage sert à figer l'environnement au début du calcul (en affectant aux variables x'_i les valeurs x_i uniquement lorsque le calcul commence).

L'automate est identifié avec un nom frais $\mathcal{L}2$. Le code obtenu est transformé pour obtenir une séquence de trois instructions :

— tout d'abord, l'état de l'automate est réinitialisé (`continue`$_{\mathcal{L}2}$ `Idle`$_{\mathcal{L}2}$) si a_{reset} vaut `true` (sachant qu'il vaut `Idle`$_{\mathcal{L}2}$ par défaut au commencement de l'exécution du programme) ;

— puis vient la définition de l'automate proprement dit, avec la copie de l'environnement ($x'_i := x_i$) lorsque le calcul commence.

— puis le résultat x reçoit le résultat de $\mathcal{A}$ (à savoir `result`$_{\mathcal{L}2}$ par convention) avec le booléen `true` lorsque le calcule se termine (c'est-à-dire, si `rdy`$_{\mathcal{L}2}$ vaut `true` par convention) et l'atome a_{default} avec le booléen `false` sinon.

COMPILE-EXEC

$\mathcal{L}2$ et les x'_i sont des noms frais

$$\mathcal{L}2 \vdash \mathcal{C}[\![\mathcal{A}[(x_i \mapsto x'_i)_{x_i \in \mathcal{FV}(\mathcal{A})}]]\!] \rightsquigarrow \left(\begin{array}{l} \texttt{automaton}_{\mathcal{L}2} \\ \texttt{| Idle}_{\mathcal{L}2} \to s \\ \texttt{| } f_1 \to s_1 \texttt{ | } \cdots \\ \texttt{| } f_n \to s_n \\ \texttt{end} \end{array}\right)$$

$s' \stackrel{\text{def}}{=}$

```
(* en cas de reset, retourner dans l'état initial *)
(if a_reset then (continue_L2 Idle_L2) else ());
(* puis, dans tous les cas, faire progresser l'automate d'un pas *)
automaton_L2
| Idle_L2 → (* avec copie des entrées dans l'état initial *)
    ((x'_i := x_i;)_{x_i ∈ FV(A)} s)
| f_1 → s_1 | ⋯
| f_n → s_n
end;
(* si le calcul est terminé *)
( if rdy_L2 then (* alors la variable destination reçoit une paire *)
    x := (result_L2, true) (* formée du résultat et du booléen rdy=true *)
  else (* sinon la variable destination reçoit une paire *)
    x := (a_default, false) (* formée de la valeur *)
  (* de l'expression par défaut et du booléen rdy=false *)
); k (* puis la continuation est exécutée *)
```

$\rho, \mathcal{L} \vdash$ **COMPILE** `exec` $\mathcal{A}$ `default` a_{default} `reset` a_{reset} **TO** x **THEN** k **IS** $s', \emptyset, \emptyset$

FIGURE 5.33 Compilation d'`exec`

Le compilé d'une paire parallèle $(\mathcal{A}_1 \| \mathcal{A}_2)$ vers x suivi de k est obtenu en compilant les deux automates $\mathcal{A}_i$. Il en résulte deux blocs $\texttt{automaton}_i$ qui sont composés séquentiellement (en suivant l'ordre gauche-droite du parallèle d'ECLAT). Autrement dit, les deux blocs sont juxtaposés (comme expliqué sur la figure 5.20). Chaque bloc $\texttt{automaton}_i$ écrit `true` dans sa variable associée $\texttt{rdy}_i$ dès que le calcul qu'il réalise se termine. Lorsque les deux calculs sont terminés, la variable x reçoit la paire des résultats (règles COMPILE-PAR et COMPILE-PAR0).

COMPILE-PAR

$$\frac{\begin{array}{c}\mathcal{L}_1,\ \mathcal{L}_2 \text{ et } \mathit{sync} \text{ sont des noms frais} \\ i \in \{1,2\} \qquad \begin{cases} \mathcal{L}_i \vdash \mathcal{C}[\![\mathcal{A}_i]\!] \rightsquigarrow \texttt{automaton}_{\mathcal{L}_i}\{\texttt{Idle}_{\mathcal{L}_i} : s_i\} \cup \{ts_i\}\ \texttt{end} \\ s_i' \stackrel{\text{def}}{=} \texttt{automaton}_{\mathcal{L}_i}\ \{\texttt{Idle}_{\mathcal{L}_i} : \texttt{()}\} \cup \{ts_i\}\ \texttt{end} \end{cases} \\ ts_1 \neq \emptyset \text{ ou } ts_2 \neq \emptyset \end{array}}{\rho, \mathcal{L} \vdash \textbf{COMPILE}\ (\mathcal{A}_1 \| \mathcal{A}_2)\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s, \{\ t\ \}/\emptyset}$$

$$s \stackrel{\text{def}}{=} \left[\begin{array}{l} s_1\texttt{;}\ s_2\texttt{;}\ \textit{(* exécuter immédiatement le premier pas des deux automates *)} \\ \texttt{continue}_{\mathcal{L}}\ \mathit{sync}\ \textit{(* puis (pause) exécuter le reste dans l'état sync *)} \end{array}\right.$$

$$t \stackrel{\text{def}}{=} \mathit{sync} \mapsto \left(\begin{array}{l} \textit{(* une fois dans l'état sync *)} \\ s_1'\texttt{;}\ s_2'\texttt{;}\ \textit{(* faire progresser les deux automates d'un pas *)} \\ \texttt{if rdy}_{\mathcal{L}_1}\texttt{\&}\ \texttt{rdy}_{\mathcal{L}_2}\ \textit{(* si les deux se sont terminés *)} \\ \texttt{then}\ \textit{(* alors construire la paire résultat, puis *)} \\ \quad (x \coloneqq (\texttt{result}_{\mathcal{L}_1}\texttt{,result}_{\mathcal{L}_2})\texttt{;}\ k)\ \textit{(* exécuter la continuation k *)} \\ \texttt{else}\ \textit{(* sinon attendre *)} \\ \quad \texttt{continue}_{\mathcal{L}}\ \mathit{sync}\ \ \textit{(* ainsi les deux calculs progressent *)} \\ \quad \textit{(* pas à pas, de façon synchronisée (lockstep) *)} \end{array}\right)$$

COMPILE-PAR0

$$\frac{\mathcal{L}_1 \text{ et } \mathcal{L}_2 \text{ sont des noms frais} \qquad i \in \{1,2\} \qquad \mathcal{L}_i \vdash \mathcal{C}[\![\mathcal{A}_i]\!] \rightsquigarrow \underbrace{\texttt{automaton}_{\mathcal{L}_i}\{\texttt{Idle}_{\mathcal{L}_i} : s_i\}\ \texttt{end}}_{s_i'}}{\rho, \mathcal{L} \vdash \textbf{COMPILE}\ (\mathcal{A}_1 \| \mathcal{A}_2)\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ (s_1'\texttt{;}\, s_2'\texttt{;}\, x \coloneqq (\texttt{result}_{\mathcal{L}_1}\texttt{,result}_{\mathcal{L}_2})\texttt{;}\ k), \emptyset/\emptyset}$$

FIGURE 5.34 Compilation des paires parallèles

Le point important ici est de respecter la sémantique synchrone d'ECLAT, ce qui requiert de résoudre plusieurs problèmes d'implantation illustrés ci-dessous.

Exemple : soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates CORE-ECLAT réalisés en MINI-HDL par deux blocs $\texttt{automaton}_i$ notés $\mathcal{C}[\![\mathcal{A}_1]\!]$ et $\mathcal{C}[\![\mathcal{A}_2]\!]$. Comme précédemment, ces blocs $\texttt{automaton}_i$ communiquent respectivement avec l'extérieur à l'aide de la variable $\texttt{result}_i$ de type $\mathring{\tau}_i$ et la variable de synchronisation $\texttt{rdy}_i$. Soit $\mathring{e}_{\mathsf{before}}$ une expression instantanée et soit s_{before} la réalisation de $\mathring{e}_{\mathsf{before}}$ en MINI-HDL : s_{before} assigne la valeur de $\mathring{e}_{\mathsf{before}}$ à une variable y de type $\mathring{\tau}_y$. Sans perte

de généralité, supposons que l'exécution de $\mathring{e}_{\mathsf{before}}$ produise un effet de bord [14].
On considère la compilation de l'expression $\mathring{e}_{\mathsf{exemple}}$ suivante :

$$\mathring{e}_{\mathsf{exemple}} \stackrel{\text{def}}{=} \texttt{(let } y = \mathring{e}_{\mathsf{before}} \texttt{ in } (\mathcal{A}_1 \| \mathcal{A}_2)\texttt{)}$$

Intuitivement, il s'agit de construire une instruction MINI-HDL faisant progresser $\mathcal{C}[\![\mathcal{A}_1]\!]$ et $\mathcal{C}[\![\mathcal{A}_2]\!]$ en parallèle de façon synchrone (c'est-à-dire, dans le même *tick* d'horloge), comme suit :

```
1  var rdy_L, result_L : τ̊1 * τ̊2,
2      rdy1 : bool, rdy2 : bool,
3      result1 : τ̊1, result2 : τ̊2, y : τ̊y in
4  var state_L : (Idle_L) in (* Idle_L par défaut *)
5  | Idle_L ->
6     s_before; C[[A1]]; C[[A2]];
7     if rdy1 & rdy2 then (result_L := (result1,result2); rdy_L := true)
8     else rdy_L := false;
9     continue_L Idle_L
10 end
```

Toutefois, ce code MINI-HDL ne respecte pas la sémantique de l'expression $\mathring{e}_{\mathsf{exemple}}$, et cela pour au moins une raison : l'instruction s_{before} (avec effet de bord) est exécutée à chaque *tick*, ce qui conduit à une répétition de l'effet de bord, contrairement au comportement attendu de l'expression $\mathring{e}_{\mathsf{exemple}}$ (à savoir, exécuter e_{before} uniquement au démarrage, puis exécuter $(\mathcal{A}_1 \| \mathcal{A}_2)$).

Pour y remédier, il faut forcer l'instruction s_{before} à n'être exécutée que lorsque le calcul démarre. Voici alors une solution envisageable :

```
1  var rdy_L, result_L : τ̊1 * τ̊2,
2      rdy1 : bool, rdy2 : bool,
3      result1 : τ̊1, result2 : τ̊2, y : τ̊y in
4  var state_L : (Idle_L|Par) in (* Idle_L par défaut *)
5  automaton_L
6  | Idle_L -> rdy_L := false; s_before; continue_L Par
7  | Par -> C[[A1]]; C[[A2]];
8           if rdy1 & rdy2 then
9             (result_L := (result1,result2); rdy_L := true)
10          else continue_L Idle_L
11 end
```

Cependant, là encore, cette réalisation ne respecte pas la sémantique de l'expression $\mathring{e}_{\mathsf{exemple}}$; en effet, le calcul $(\mathcal{A}_1 \| \mathcal{A}_2)$ démarre *un cycle trop tard*. De plus, les deux automates $\mathcal{A}_i$ devraient s'attendre mutuellement avant de terminer, ce qui n'est pas le cas dans cette réalisation car les

14. Par exemple, un appel de fonction externe `print_int` qui affiche la valeur d'un entier en mode simulation.

deux blocs $\mathcal{C}[\![\mathcal{A}_i]\!]$ ont été engendrés par la règle COMPILE-AUTOMATON (sous-section 5.8.2) de manière à ce que tous deux retournent dans leur état initial respectif lorsqu'ils se terminent.

Pour résoudre ces problèmes d'implantation, nous pouvons remarquer que chaque bloc $\texttt{automaton}_i$ peut être déconstruit en :

$$\texttt{automaton}_i \ \{\ \texttt{Idle}_i \ \texttt{->}\ s_i\ \}\ \cup\ ts_i\ \texttt{end}$$

où $\texttt{Idle}_i$ est un état initial tandis que ts_i est un ensemble de transitions. Appellons s_i (*resp.* ts_i) la *tête* (*resp.* le *reste*) du bloc $\mathcal{C}[\![\mathcal{A}_i]\!]$. Nous pouvons alors réorganiser le code MINI-HDL ainsi :

```
1  var rdy_L, result_L : τ̊1 * τ̊2
2      rdy1, rdy2 : bool,
3      result1 : τ̊1, result2 : τ̊2, y : τ̊y in
4  var state_L : (Idle_L|Par) in (* Idle_L par défaut *)
5  var state1 : (Idle1|···) in
6  var state2 : (Idle2|···) in
7  automaton_L
8  | Idle_L -> rdy_L := false; s_before; s1; s2; continue_L Par
9  | Par -> automaton_1 | Idle1 -> () | ts1 end;
10          automaton_2 | Idle2 -> () | ts2 end;
11          if rdy1 & rdy2 then (result_L := (result1,result2); rdy_L := true)
12          else continue_L Idle_L
13 end
```

Cette solution correspond bien [15] au comportement de l'expression e_{exemple} à condition que la branche $\mathcal{A}_1$ *ou* la branche $\mathcal{A}_2$ soit non-instantanée (règle COMPILE-PAR, figure 5.34).

Dans le cas où $\mathcal{A}_1$ et $\mathcal{A}_2$ sont toutes deux instantanées (autrement dit, si leur réalisation $\mathcal{C}[\![\mathcal{A}_i]\!]$ est de la forme $\texttt{automaton}_i\ \texttt{|}\ \texttt{Idle}_i\ \texttt{->}\ s_i\ \texttt{end}$) la paire parallèle $(\mathcal{A}_1 \| \mathcal{A}_2)$ a le même comportement qu'une paire séquentielle $(\mathcal{A}_1, \mathcal{A}_2)$ (règle COMPILE-PAR0 en bas de la figure 5.34), comme déjà mentionné au chapitre 4. Par exemple, l'expression instantanée :

$$\mathring{e}_{\mathsf{exemple}} \stackrel{\text{def}}{=} \texttt{(let}\ y = \mathring{e}_{\mathsf{before}}\ \texttt{in (1 ||}\ \texttt{2+3))}$$

est compilé comme suit, sans transition vers un état intermédiaire `Par` (car cette transition aurait pris un cycle, ce qui n'est pas compatible avec la sémantique d'ECLAT) :

15. En particulier, lorsque le calcul réalisé par chaque ensemble de transitions ts_i se termine, une transition est prise vers l'état initial $\texttt{Idle}_i$; or $\texttt{Idle}_i$ est associé à l'instruction `()` : autrement dit, le calcul ne redémarre pas. Plus tard dans l'exécution de e_{exemple}, lorsque les deux variables rdy_i valent `true`, le bloc $\texttt{automaton}_\mathcal{L}$ englobant retourne dans l'état `Idle` : le calcul redémarrant, les instructions s_1 et s_2 sont à nouveau exécutées, ce qui a pour effet de réinitialiser les blocs $\texttt{automaton}_1$ et $\texttt{automaton}_2$ (car les deux s_i contiennent une transition $\texttt{continue}_i\ Q$).

```
1 var rdy_L : bool, result_L : τ̊1 * τ̊2, y : τ̊y in
2 var state_L : (Idle_L) in (* Idle_L par défaut *)
3 automaton_L
4 | Idle_L -> s_before; result_L := (1,2+3); rdy_L := true; continue_L Idle_L
5 end
```

5.8.4 Retour instantané des appels de fonctions

Comme déjà indiqué, le retour instantané d'un appel de fonction n'est pas réalisable par le jugement de compilation des expressions (qui est dirigé par la syntaxe), car la moindre transition dans l'automate prend un cycle.

Remarque : Pour faire une analogie avec la programmation assembleur sur processeur classique, cela revient à demander à sauter à un point du code sans effectuer d'instruction de branchement : ce n'est possible que si l'instruction ciblée est l'instruction suivante.

C'est pourquoi le jugement de compilation des expressions collecte une *table de continuations* ω et place dans l'automate des instructions fictives `(continue`$_{\mathcal{L}}$ $f^\sharp$`)` qui devront être remplacées par la bonne continuation associée à f dans la table de continuations. Lorsque plusieurs choix sont possibles, celle-ci doit être sélectionnée dynamiquement par un traitement par cas (`case/with` sur la variable `f_id` qui, par convention, identifie par un entier n le site d'appel de l'appelant (voir règle Compile-Direct-call). La continuation de ce site d'appel a été placée à l'emplacement $f \cdot n$ dans la table de continuations au moment de l'appel (voir aussi règle Compile-Direct-call).

Pour insérer les continuations dans le bloc `automaton` après avoir compilé un automate $\mathcal{A}$, la règle Compile-automaton (sous-section 5.8.2) applique une transformation dédiée **dispatch**$[\![s]\!]_\omega$.

La transformation **dispatch**$[\![s]\!]_\omega$ est définie sur la figure 5.35. Elle remplace chaque instruction fictive `(continue`$_{\mathcal{L}}$ $f^\sharp$`)` présente dans l'instruction s par une instruction `case/with` sélectionnant la bonne continuation en fonction de `f_id`. Chaque continuation k placée dans le `case/with` résultant doit aussi être transformée avec **dispatch**$[\![k]\!]_\omega$.

Cette section a donné le schéma de compilation de Core-Eclat vers Mini-HDL. Celui-ci spécifie en détail les mécanismes à l'œuvre pour partager les appels de fonctions récursives, implanter les synchronisations instantanées dictées par la sémantique et compiler le parallélisme.

Nous pouvons maintenant implanter Eclat entier (qui comporte des traits de programmation applicatifs avancés) en traduisant Eclat vers Core-Eclat. La section suivante décrit cette dernière étape.

5.9 Normalisation d'Eclat vers Core-Eclat

Chaque programme source Eclat est normalisé à travers une série de transformations préservant la sémantique, pour former un programme Core-Eclat équivalent. La chaîne de transformations est représentée sur la figure 5.36. Si le programme est mal typé ou si certaines

$$
\begin{array}{lcl}
\textbf{dispatch}[\![\texttt{continue}_{\mathcal{L}}\ f^{\sharp}]\!]_{\omega} & \stackrel{\text{def}}{=} & \left[\begin{array}{l}\texttt{case}\ f\texttt{_id}\ \texttt{with}\\ \mid n_1 \rightarrow \textbf{dispatch}[\![k_1]\!]_{\omega} \mid \cdots\\ \mid n_m \rightarrow \textbf{dispatch}[\![k_m]\!]_{\omega}\\ \texttt{otherwise}\ ()\end{array}\right.\\
 & & \text{avec } \omega(f) \stackrel{\text{def}}{=} \{n_1 : k_1, \cdots\ n_m : k_m\}\\
\textbf{dispatch}[\![\texttt{continue}_{\mathcal{L}}\ f]\!]_{\omega} & \stackrel{\text{def}}{=} & \texttt{continue}_{\mathcal{L}}\ f\\
\textbf{dispatch}[\![\texttt{if}\ x\ \texttt{then}\ s\ \texttt{else}\ s']\!]_{\omega} & \stackrel{\text{def}}{=} & \texttt{if}\ x\ \texttt{then}\ \textbf{dispatch}[\![s]\!]_{\omega}\ \texttt{else}\ \textbf{dispatch}[\![s']\!]_{\omega}\\
\textbf{dispatch}[\![s\texttt{;}\ s']\!]_{\omega} & \stackrel{\text{def}}{=} & \textbf{dispatch}[\![s]\!]_{\omega}\texttt{;}\ \textbf{dispatch}[\![s']\!]_{\omega}\\
\textbf{dispatch}[\![\texttt{var}\ x\texttt{:}\mathring{\tau}\ \texttt{in}\ s]\!]_{\omega} & \stackrel{\text{def}}{=} & \texttt{var}\ x\texttt{:}\mathring{\tau}\ \texttt{in}\ \textbf{dispatch}[\![s]\!]_{\omega}\\
\textbf{dispatch}[\![p \coloneqq a]\!]_{\omega} & \stackrel{\text{def}}{=} & p \coloneqq a\\
\textbf{dispatch}[\![()]\!]_{\omega} & \stackrel{\text{def}}{=} & ()\\
\textbf{dispatch}[\![\underbrace{\texttt{automaton}\ \cdots\ \texttt{end}}_{s}]\!]_{\omega} & \stackrel{\text{def}}{=} & s\\
\textbf{dispatch}[\![(x\texttt{,}rdy) \coloneqq \texttt{run}_i\ f\ a]\!]_{\omega} & \stackrel{\text{def}}{=} & (x\texttt{,}rdy) \coloneqq \texttt{run}_i\ f\ a
\end{array}
$$

FIGURE 5.35 Insertion de `case/with` pour l'implantation du retour de fonction

fonctions récursives définies dans le programme ne sont pas récursives terminales, alors le programme est rejeté. Sinon, il est normalisé.

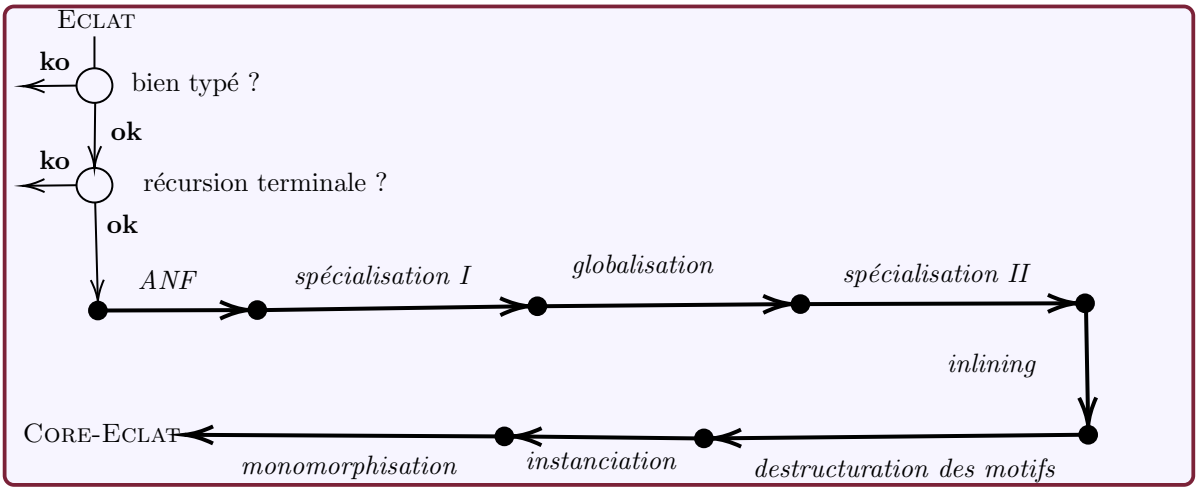


FIGURE 5.36 Normalisation d'ECLAT

5.9.1 Mise en forme ANF (*Administrative Normal Form*)

L'exécution des programmes ECLAT réalisent des synchronisations en de nombreux points de la syntaxe des expressions. Une synchronisation est nécessaire notamment :

— dans la condition des `if`, *e.g.*, `if` <calcul> `then 42 else 43`;

— dans les composantes de paires séquentielles, *e.g.*, `(`<calcul>`,42)`;

— en argument de fonction, *e.g.*, f(<calcul>) ;

— et plus fondamentalement dans les liaisons locales, *e.g.*, `let` x = <calcul> `in x+1`.

Pour simplifier la compilation, nous regroupons les différents cas de composition séquentielle dans la seule construction `let/in`. Toute sous-expression qui n'est pas un atome a dans un *contexte d'évaluation* (désigné par le symbole <calcul> dans les exemples ci-dessus) est liée par un `let`. Les atomes sont les constantes, les variables et les applications d'opérateurs combinatoires (comme défini figure 5.28). Par exemple, l'expression `if` <calcul> `then 42 else 43` se réécrit en `let` x = <calcul> `in (if` x `then 42 else 43)` où x est un nom frais. Cette transformation est appelée *mise sous forme ANF* (*Administrative Normal Form* [116]) dans la littérature.

5.9.2 Spécialisation I

Nous cherchons d'abord à obtenir un programme dit *du premier ordre*, c'est-à-dire un programme où tous les arguments de fonctions sont typés avec des types de base. Cela se vérifie à partir d'informations collectées en amont lors de la phase de typage statique. À chaque fois que l'argument d'une fonction f n'est pas un type de base, la fonction f est *inlinée* : chaque site d'appel à f est remplacé par la définition de f en liant les arguments aux paramètres formels (avec un `let`). Si la fonction f est récursive, la définition récursive est dupliquée au niveau de chaque site d'appel direct à f et les arguments qui ne sont pas typables avec des types de base sont éliminés : ainsi la fonction récursive résultante est bien du premier ordre.

5.9.3 Globalisation (par λ-lifting)

L'étape suivante consiste à globaliser l'ensemble des définitions de fonctions par λ-lifting [113]. Cette transformation de programme est illustrée sur la figure 5.37. Le programme source comporte un certain nombre de définitions de fonctions locales (récursives ou non). La transformation s'effectue en deux étapes. La première étape consiste à expliciter l'environnement lexical de chaque fonction en passant ses variables libres en paramètres. Par exemple, la fonction `let g x = x + y` devient `let g (x,y) = x + y` et l'appel `g z` devient `g (z,y)`. Sur la figure 5.37, les variables nouvellement introduites sont représentées avec un fond coloré ; les variables en position fonctionnelle (comme `f` dans `f v`) ne sont pas traitées comme des variables libres pour éviter d'engendrer de nouvelles fonctions d'ordre supérieur, qui ont déjà été éliminées dans l'étape de spécialisation (sous-section 5.9.2). Cet exemple montre que la première étape n'a pas permis de rendre closes toutes les fonctions. La fonction `let f z = g (z,y)` comporte une variable libre `y` nouvellement introduite. C'est pourquoi la transformation doit être itérée jusqu'à atteindre un point fixe [65]. Une fois que toute les fonctions sont closes, une seconde transformation s'applique pour globaliser l'ensemble des fonctions du programmes.

5.9.4 Spécialisation II

La globalisation s'applique sur des programmes *du premier ordre* (où tous les arguments de fonctions sont typables avec des types de base). Une fois le programme globalisé, toutes les

(a) expression source

```
let y = 1 in
let infinite_loop (n) =
  infinite_loop (n+y)
in
let foo v =
  let g x = x + y in
  let f z = g z in
  f v
in
foo 42
```

(b) expliciter l'environnement (1)

```
let y = 1 in
let infinite_loop (n,y) =
  infinite_loop (n+y,y)
in
let foo (v,y) =
  let g (x,y) = x + y in
  let f z = g (z,y) in
  f v
in
foo (42,y)
```

(c) expliciter l'environnement (2)

```
let y = 1 in
let infinite_loop (n,y) =
  infinite_loop (n+y,y)
in
let foo (v,y) =
  let g (x,y) = x + y in
  let f (z,y) = g (z,y) in
  f (v,y)
in
foo (42,y)
```

(d) globaliser les fonctions

```
let infinite_loop (n,y) =
  infinite_loop (n+y,y)
in
let g (x,y) = x + y in
let f (z,y) = g (z,y) in
let foo (v,y) =
  f (v,y)
in
let y = 1 in
foo (42,y)
```

FIGURE 5.37 Globalisation par λ-lifting

fonctions sont closes : les variables libres des anciennes fonctions sont passées comme paramètres supplémentaires, explicitant ainsi leurs environnements. Or, pour assurer la correction du processus de normalisation, il faut prendre garde à ce qu'après l'ajout de paramètres supplémentaires, le programme soit toujours du premier ordre. Pour ce faire nous appliquons une seconde passe de spécialisation (appelée *spécialisation II*). Ce sera particulièrement nécessaire lorsque nous ajouterons à ECLAT des déclarations de tableaux locaux (qui ne seront pas des types de base) au chapitre suivant. Nous reviendrons sur cette étape et sur la terminaison du processus de compilation à la sous-section 6.3.2.

5.9.5 Intégration des appels de fonctions non récursives

La transformation suivante consiste à *inliner* toutes les fonctions non récursives. L'*inlining* est largement utilisée dans les compilateurs pour atteindre de meilleurs performances. Toutefois ici, le but est différent : l'*inlining* d'une fonction ECLAT correspond à l'instanciation d'un circuit, c'est-à-dire la duplication de celui-ci pour qu'il puisse s'exécuter simultanément sur des entrées différentes. L'*inlining* est connue comme étant susceptible d'augmenter de façon exponentielle la taille de certains programmes.

L'étape d'*inlining* étant placée après l'étape de globalisation, les définitions de fonctions récursives locales ne sont pas dupliquées [16], comme le montre la figure 5.38.

(a) expression source

```
let fibonacci(n) =
  let rec fib(i,a,b) =
    if i = 0 then a
    else fib(i-1,b,a+b)
  in fib(n,1,1)
in
let x = fibonacci(10) in
fibonacci(x)
```

(b) expression après *inlining*

```
let rec fib(i,a,b) =
  if i = 0 then a
  else fib(i-1,b,a+b)
in
let x = fib(10,1,1) in
fib(x,1,1)
```

FIGURE 5.38 Globalisation puis intégration des appels de fonctions non-récursives

5.9.6 Déconstruction des motifs

En CORE-ECLAT, les motifs dans les constructions `let/in`, `let/rec` et `reg` doivent être des variables. Or, en ECLAT, les lieurs introduisent plus généralement un motif p qui peut être soit un nom, soit la constante `()`, soit une paire de motifs. Une transformation de programme est donc nécessaire pour réduire le langage de motifs à de simples variables. C'est pourquoi la normalisation d'ECLAT vers CORE-ECLAT doit déconstruire en profondeur les motifs dans les liaisons en les remplaçant par des appels aux opérateurs `fst` et `snd` de projection de paires.

5.9.7 Instanciation

Comme cela a été expliqué à la section 5.7, CORE-ECLAT comprend une catégorie syntaxique $\mathcal{A}$ (pour *automate*) qui n'existe pas en ECLAT ; un automate est une expression ECLAT qui ne comporte pas de noms de fonctions libres et qui correspond à une machine de Mealy dans le code VHDL engendré. Les automates se situent directement dans les branches des

16. À l'exception de celles qui prennent des paramètres fonctionnels (déjà éliminées par spécialisation) et des appels de fonctions récursives polymorphes (qui seront dupliqués par monomorphisation, voir sous-section 5.39).

paires parallèles $(\mathcal{A}_1 \| \mathcal{A}_2)$ et sous les bloc `exec`. Tout appel à une fonction récursive f dans un automate correspond à une transition vers l'état f. Les transitions sortantes de f réalisent la définition de la fonction f. Or, à ce stade de la compilation d'ECLAT, il peut arriver qu'une fonction récursive f soit définie (avec `let/rec`) à l'extérieur de l'automate $\mathcal{A}$ qui appelle f. C'est pourquoi une étape d'instanciation est nécessaire pour déplacer les définitions de fonctions récursives en profondeur, de sorte que tout automate $\mathcal{A}$ qui appelle une fonction récursive f contiennent aussi la définition de f. Autrement dit, après instanciation, aucun nom de fonction récursive appelée dans un automate $\mathcal{A}$ ne doit apparaître libre dans $\mathcal{A}$.

5.9.8 Monomorphisation

Comme discuté à la sous-section 5.2.6, la compilation d'ECLAT vers un circuit doit *monomorphiser* [40] les programmes, c'est-à-dire faire en sorte que chaque schéma de types dans le programme soit réduit à un type, afin que la taille de chaque variable (en nombre de bits) soit connue statiquement. On notera que l'évaluation des programmes, dans la sémantique d'ECLAT, est déjà une forme de monomorphisation et les substitutions dans la sémantique correspondent aux étapes d'*inlining*. Cela aide à raisonner sur la correction de la chaîne de transformations.

La figure 5.39 donne un exemple de monomorphisation d'un programme ECLAT. Le programme source comporte une fonction `delay` qui étant donnée une valeur `x`, attend un cycle et retourne `x`. La fonction est appelée une première fois avec un argument booléen, puis une seconde fois avec un argument entier. Ce programme est bien typé dans système de types d'ECLAT avec polymorphisme paramétrique. Pour empêcher une fonction d'être appelée avec des valeurs de types différents, il suffit de dupliquer sa définition au niveau de chaque site d'appel. Les sites d'appel des fonctions monomorphes sont laissés intacts (*cf* les appels à `inc` sur la figure 5.39).

(a) expression source

```
let rec delay x = x in
let a = delay true in
let b = delay (42 : int<32>) in
let inc c = c + 1 in
if a then inc(b) else inc(0)
```

(b) expression monomorphisée

```
let a = let rec delay x = x in
        delay true
in
let b = let rec delay x = x in
        delay (42 : int<32>)
in
let inc c = c + 1 in
if a then inc(b) else inc(0)
```

FIGURE 5.39 Monomorphisation

5.10 Conclusion

Cette section a présenté une chaîne de compilation complète pour Eclat ciblant VHDL pour la reconfiguration de circuits FPGA connectés aux entrées/sorties physiques. L'approche proposée a consisté à appliquer à Eclat des transformations de programmes classiques préservant la sémantique (comme le λ-lifting et l'*inlining*) pour obtenir un programme exprimé dans un sous-ensemble d'Eclat appelé Core-Eclat. Core-Eclat est ensuite compilé dans un langage intermédiaire appelé Mini-HDL, lui même étant traduit en VHDL. L'intérêt de Mini-HDL est son modèle de programmation synchrone flot de données, à mi-chemin entre VHDL et Eclat. La sémantique comportementale de Mini-HDL est similaire à une sémantique existante [83] d'un sous-ensemble de VHDL. Elle est aussi très proche de la sémantique comportementale d'Eclat, les deux langages partageant d'ailleurs directement des éléments syntaxiques et sémantiques. Cela vise à faciliter les raisonnements quant à l'équivalence entre un programme Eclat et sa traduction en VHDL.

6 Mixer parallélisme synchrone et mémoire partagée

L'appel de fonctions externes depuis ECLAT (cf. la sous-section 3.2.6 et la section 5.4) donne accès à toutes sortes de composants matériels disponibles dans un FPGA, et notamment des blocs de mémoire RAM. Cependant, l'accès à ces blocs depuis un langage de description de matériel comme VHDL se fait suivant des patrons d'accès spécifiques de très bas niveau.

Pour offrir de la mémoire partagée dans un style fonctionnel-impératif avec une sémantique claire, ce chapitre enrichit ECLAT avec des tableaux primitifs modifiables en place. Cela permet d'encoder des variables partagées par exemple (sous forme de tableaux à un élément) ou de stocker des données relativement larges (de l'ordre du million de bits, y compris sur de petits modèles de circuits FPGA).

Les tableaux en ECLAT ne sont pas des types de base ; il n'y a pas de tableaux de tableaux et les tableaux ne peuvent pas être retournés par les fonctions. Ces choix de conception sont directement liés au choix d'implantation des tableaux qui vise à concilier parallélisme et mémoire partagée. Chaque tableau est implanté par un tableau VHDL instanciant une mémoire RAM indépendante ; la mémoire n'est pas centralisée. L'accès à chaque mémoire se fait en un cycle : il s'agit donc d'un calcul (au sens où il n'est pas instantané) ; l'interaction avec l'environnement ne peut utiliser la mémoire qu'à travers la construction **`exec`**. Les accès concurrents à des tableaux différents peuvent s'effectuer simultanément. En revanche, les accès à un même tableau sont séquentialisés à l'aide d'un *verrou* associé à chaque tableau. La gestion des verrous est implicite dans les programmes sources, mais le comportement fonctionnel et temporel des programmes avec accès mémoire est rendu explicite par la sémantique :

— les "pas" de calculs synchronisés dans chaque paire parallèle $(e_1 \| e_2)$ sont exécutés séquentiellement : e_1 progresse d'un pas puis e_2 progresse d'un pas dans le même cycle (c'est ce que dit la règle REDUCE-PAR-PAUSE du chapitre 4, sous-section 4.2.7) ;

— dans une séquence e_1 `;` e_2 (qui est une construction dérivée de **`let/in`**), à l'instant où e_1 termine, e_2 commence son exécution et c'est la première expression exécutée après e_1 ;

— à chaque appel récursif et à chaque accès à un tableau dont le verrou est déjà pris, le calcul se suspend localement jusqu'à la fin du cycle.

6.1 Mémoire accessible dans un circuit FPGA

Comme présenté au chapitre 2 (sur la figure 2.1), les circuits FPGA comprennent, non seulement de la logique programmable (*éléments logiques*), mais aussi des blocs de mémoire RAM (*Random-Access Memory*), c'est-à-dire, des zones mémoire accessibles à la fois en lecture et en écriture. Dans sa forme la plus simple, une RAM présente :

— trois entrées, à savoir `w_enable` indiquant le démarrage d'une écriture, l'adresse `addr` d'un mot à accéder (en lecture *et* écriture) et le mot `data_in` à écrire,

— ainsi qu'une sortie `data_out` contenant le mot lu à l'adresse `addr`.

Pour utiliser de la mémoire dans une description de matériel au niveau RT, on peut soit instancier des blocs RAM de façon structurelle, soit décrire le comportement d'une mémoire RAM en suivant un patron d'accès bien précis (qui est donné sur la figure 6.1) ; on laisse alors le synthétiseur logique détecter ce patron par *inférence de RAM* [221] et instancier un ensemble de blocs RAM à la place.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity ram_1024_int is
6    port(
7     signal clk      : in std_logic;
8     signal addr     : in natural range 0 to 9;
9     signal data_in  : in integer;
10    signal w_enable : in std_logic := '0';
11    signal data_out : out integer);
12 end entity;
13 architecture rtl of ram_1024_int is
14 type ram_t is array (natural range 0 to 1023)
15    of integer;
16 signal ram : ram_t;
17 begin
18   process (clk)
19     begin
20       if (rising_edge(clk)) then
21         data_out <= ram(addr);
22         if w_enable = '1' then
23           ram(addr) <= data_in;
24         end if;
25       end if;
26    end process;
27 end  architecture;
```

FIGURE 6.1 Description d'une mémoire RAM de 1024 mots en VHDL

Le comportement temporel du bloc RAM de la figure 6.1 est le suivant : à chaque front montant de l'horloge, la sortie `data_out` reçoit la valeur du mot à l'adresse `addr` dans la mémoire `ram`. Si l'écriture est activée via l'entrée `w_enable`, alors l'emplacement mémoire à l'adresse `addr` dans `ram` reçoit la valeur du mot `data_in` à écrire.

Ce patron d'accès correspond à une machine de Moore : la transaction en écriture ou en lecture se fait toujours sur les fronts montants de l'horloge globale (lignes 20 à 25 de la figure 6.1), d'où un retard d'un cycle à chaque accès mémoire.

Pour lire une valeur dans un bloc RAM, le circuit connecté à ce bloc RAM doit positionner l'entrée `addr` à l'indice souhaité et s'assurer que `w_enable` est à `0` (pour ne pas écraser la valeur

stockée en mémoire à cette adresse), puis attendre le prochain *tick* d'horloge pour que la sortie du bloc RAM soit mise à jour, donnant ainsi accès à la valeur demandée.

Pour écrire une valeur dans le bloc RAM, le circuit doit positionner l'entrée `addr` à l'indice souhaité, `w_enable` à `1` et `data_in` à la valeur à écrire ; puis attendre le cycle suivant pour terminer l'écriture en positionnant `w_enable` à `0`.

6.2 Extension d'Eclat avec mémoire partagée

Cette section formalise l'extension d'ECLAT avec des tableaux modifiables en place. Ces tableaux sont implantés sous forme de blocs RAM suivant le patron d'accès de la figure 6.1. Comme évoqué plus haut, chaque accès au tableau prend un cycle, puisque le patron d'accès requis s'avère être une machine de Moore (dont les spécificités temporelles ont été présentées à la section 2.4). Les accès mémoire en ECLAT ne sont donc pas instantanés : ils servent essentiellement à programmer des calculs généralistes.

Nous modélisons le comportement temporel des accès de tableaux dans la sémantique synchrone d'ECLAT, coïncidant avec leur implantation matérielle au niveau RT (figure 6.1).

Tout l'intérêt de spécifier ces accès de façon synchrone est que, de cette manière, nous pouvons ainsi concilier parallélisme et mémoire partagée, c'est-à-dire supporter les accès mémoire concurrents, de façon sûre, prévisible et équitable, tout en exploitant le parallélisme physique au niveau du circuit pour accélérer les calculs.

6.2.1 Opérations sur les tableaux

L'extension de la syntaxe d'ECLAT avec des tableaux est définie à la figure 6.2. De nouvelles opérations sur les tableaux sont ajoutées. Nous introduisons un nouveau constructeur de type $\mathring{\tau}$ `array<`θ`>` pour typer chaque tableau : celui-ci est paramétré par un type de base $\mathring{\tau}$, qui donne le type des éléments du tableau, et une taille de tableau θ (en nombre d'éléments). Chaque tableau est une valeur `(`ℓ`:`$\mathring{\tau}$ `array<`θ`>)` aussi appelée *adresse*. Les tableaux ne sont pas des constantes, contrairement aux vecteurs.

opérateur	op	::=	$\cdots$ | `length` | `get` | `set` | `%acquire`$_\ell$ | `%release`$_\ell$
expression	e	::=	$\cdots$ | `make<`θ`>` c | `create`$_{\mathring{\tau}}$`<`θ`>` `()` | `(`ℓ`:`$\mathring{\tau}$ `array<`θ`>)`
valeur	v	::=	$\cdots$ | `(`ℓ`:`$\mathring{\tau}$ `array<`θ`>)`
type	τ	::=	$\cdots$ | $\mathring{\tau}$ `array<`θ`>`
taille	θ	::=	$\cdots$

FIGURE 6.2 Syntaxe et sémantique d'ECLAT avec mémoire partagée

Nous ajoutons les adresses dans le langage d'expressions afin que toute valeur soit aussi

une expression, ce qui est requis dans le style de sémantique opérationnelle mise en œuvre pour ECLAT. Les adresses $(\ell{:}\mathring{\tau}$ `array<`θ`>`$)$ n'apparaissent pas dans les programmes sources : elles sont créées par la construction $($`make<`θ`>` $c)$ qui alloue statiquement un tableau de θ éléments initialisés à la constante c. Pour des raisons liées à l'implantation, l'exécution de la construction `make<`θ`>` prend $\theta + 1$ cycles. La construction $\texttt{create}_{\mathring{\tau}}\texttt{<}\theta\texttt{>()}$ est une variante *instantanée* de `make<`θ`>` qui initialise le tableau à des valeurs par défaut non spécifiées de type $\mathring{\tau}$. Dans la suite, nous utiliserons les abréviations :

$$\begin{array}{rcl} \ell & \stackrel{\text{def}}{=} & (\ell\texttt{:}\,'B\ \texttt{array<}\,'N\,\texttt{>}) \\ & & \text{où } 'B \text{ est une inconnue de type de base fraîche} \\ & & \text{et } 'N \text{ est une inconnue de taille fraîche.} \\ \texttt{create<}\theta\texttt{>()} & \stackrel{\text{def}}{=} & \texttt{create}_{'B}\texttt{<}\theta\texttt{>()} \\ & & \text{où } 'B \text{ est une inconnue de type de base fraîche.} \end{array}$$

Remarque : Suivant les modèles de FPGA ciblés, les blocs RAM ne sont pas initialisables statiquement à une valeur autre que `0`. Un parcours du tableau sera donc requis pour initialiser son contenu à l'exécution, avec un débit de *au plus* un élément de tableau par cycle.

L'opérateur `length` retourne instantanément la taille du tableau. Les opérateurs `get` et `set` permettent respectivement de lire et écrire un tableau ℓ. L'accès prend exactement un cycle auquel s'ajoute des cycles d'attentes en cas d'accès concurrents. Elles sont définies opérationnellement en utilisant des opérations internes (préfixées par le symbole `%`) qui n'apparaissent pas dans les programmes sources.

Les opérations $\texttt{\%acquire}_\ell$ et $\texttt{\%release}_\ell$ gèrent un verrou qui est associé à chaque tableau ℓ et explicité par une variable qui est notée $\texttt{lock}(\ell)$ dans la sémantique. Cela permet de séquentialiser les accès effectués en parallèle, puisque ceux-ci ne peuvent pas être réalisés simultanément avec des blocs RAM [1].

On rappelle que la construction dérivée `pause` e retarde d'un cycle le calcul de l'expression e. Elle a été définie par `(fix` f `(fun () ->` e`))()` où f est un nom frais (figure 3.5). Nous allons nous en servir à chaque requête d'accès à un tableau pour reproduire d'une part, le comportement temporel de la machine de Moore sous-jacente implantant le patron d'accès, et d'autre part l'attente en cas d'accès lorsqu'un verrou est déjà pris.

Toutes ces opérations viennent s'ajouter à l'environnement de typage des opérateurs (qui a été défini figure 3.3). Le type des tableaux n'est pas un type de base ; or la partie droite d'un type fonctionnel doit être un type de base par construction. C'est pourquoi `make` et `create` (qui "retourne" un tableau) sont des nouvelles constructions du langage et non pas des opérateurs.

Comme l'indiquent les schémas de types figure 6.3, les opérations `get` et `set` ne sont pas instantanées : chaque instance de leur schéma de type est de la forme $\tau \xrightarrow{\mathbf{1}} \mathring{\tau}'$ où $\mathbf{1}$ est une durée pire cas de zéro, un ou plusieurs cycles. Les autres opérations listées ici sont instantanées ($\mathbf{0}$).

1. Certains types de blocs RAM comprennent deux ports d'écriture, ce qui permet d'effectuer jusqu'à deux écritures et une lecture simultanément. Mais pour privilégier la prédictibilité et la simplicité du modèle synchrone d'ECLAT nous effectuons *au plus* un accès par cycle et par tableau.

$$\textsc{Ty-Array}\quad \Gamma \vdash (\ell{:}\mathring{\tau}\ \texttt{array<}\theta\texttt{>}) : \mathring{\tau}\ \texttt{array<}\theta\texttt{>}|\mathbf{0}$$

$$\textsc{Ty-Make}\quad \frac{\Gamma \vdash e : \tau|\mathbf{0}}{\Gamma \vdash \texttt{make<}\theta\texttt{>}\ e : \tau\ \texttt{array<}\theta\texttt{>}|\mathbf{1}} \qquad \textsc{Ty-Create}\quad \frac{\text{'}B \text{ est une inconnue de type de base fraîche}}{\Gamma \vdash \texttt{create<}\theta\texttt{>}\ \texttt{()} : \text{'}B\ \texttt{array<}\theta\texttt{>}|\mathbf{0}}$$

$$\begin{aligned}
\texttt{length} &: \forall\, \text{'}N \cdot \forall\, \text{'}B \cdot \text{'}B\ \texttt{array<}\text{'}N\texttt{>} \xrightarrow{\mathbf{0}} \texttt{int<16>}\\
\texttt{get} &: \forall\, \text{'}N_1 \cdot \forall\, \text{'}N_2 \cdot \forall\, \text{'}B \cdot (\text{'}B\ \texttt{array<}\text{'}N_1\texttt{>} * \texttt{int<}\text{'}N_2\texttt{>}) \xrightarrow{\mathbf{1}} \text{'}B\\
\texttt{set} &: \forall\, \text{'}N_1 \cdot \forall\, \text{'}N_2 \cdot \forall\, \text{'}B \cdot ((\text{'}B\ \texttt{array<}\text{'}N_1\texttt{>} * \texttt{int<}\text{'}N_2\texttt{>}) * \text{'}B) \xrightarrow{\mathbf{1}} \texttt{unit}\\
\texttt{\%acquire}_\ell &: \texttt{unit} \xrightarrow{\mathbf{0}} \texttt{unit}\\
\texttt{\%release}_\ell &: \texttt{unit} \xrightarrow{\mathbf{0}} \texttt{unit}
\end{aligned}$$

FIGURE 6.3 Typage d'ECLAT avec mémoire partagée

Le comportement de chaque opérateur de tableaux ECLAT est formalisé sur la figure 6.4. Il est basé sur la sémantique opérationnelle comportementale d'ECLAT (définie au chapitre 4) et consiste principalement à affiner la fonction ϑ de réduction des opérateurs (définie avec la règle REDUCE-APP-OP au chapitre 4).

On rappelle quelques notations. Une mémoire μ est une fonction partielle des adresses vers des valeurs. Les adresses mémoire sont dénotées par la variable ℓ. On note $\mu(\ell)$ la valeur stockée à l'adresse ℓ dans μ et $\mu[\ell \mapsto v]$ la nouvelle mémoire obtenue en assignant la valeur v à l'adresse ℓ dans μ. L'adresse du $i$ème élément de tableau est notée $\ell \cdot i$.

Afin de simplifier et alléger la présentation, on alloue les tableaux dans la même mémoire μ que celle utilisée pour gérer l'état des registres et des blocs `exec`. On suppose pour cela que les adresses ℓ des tableaux sont distinctes des adresses associées aux constructions $\texttt{reg}_\ell$ et $\texttt{exec}_\ell$.

Les constructions de création d'un tableau (`make<`θ`>` et `create<`θ`>`) renvoient une adresse ℓ liée à une valeur représentant le nouveau tableau en mémoire (règles REDUCE-MAKE et REDUCE-CREATE). Dans la sémantique, cette valeur en mémoire est codée par une paire $(\mathit{verrou}, \mathit{contenu})$ où :

- *verrou* est un booléen indiquant si le verrou associé au tableau est détenu ou non,
- *contenu* représente les n éléments du tableau sous la forme d'un n-uplet.

Par exemple, un tableau de quatre entiers n_1, n_2, n_3 et n_4 est représenté en mémoire par la valeur $(\texttt{false}, (n_1, n_2, n_3, n_4))$.

Les *pause* effectuées par les opérateurs `get` et `set` reflètent exactement le comportement temporel de l'implantation matérielle du langage (cas ϑ-GET et ϑ-SET), c'est-à-dire l'accès à un bloc de mémoire RAM.

Lorsque le verrou associé au tableau ℓ est déjà détenu, tout autre accès à ℓ est retardé d'un cycle jusqu'à ce que le verrou soit libéré (cas ϑ-GET-WAIT et ϑ-SET-WAIT).

Les accès hors des bornes sont détectés à l'exécution : dans ce cas la mémoire n'est pas modifiée et une erreur d'exécution se produit. L'erreur est modélisée par une boucle infinie (via la construction dérivée `halt` définie figure 3.5) comme pour les accès hors des bornes des vecteurs (cf. figure 4.4). L'erreur peut être rattrapée en réinitialisant le calcul via la clause `reset` de la construction `exec` englobante.

Les accès simultanés à la mémoire sont séquentialisés par le verrou $\texttt{lock}(\ell)$ associé à chaque tableau ℓ comme spécifié, par les cas ϑ-ACQUIRE et ϑ-RELEASE.

REDUCE-MAKE

$$\frac{\ell \text{ est une adresse fraîche} \qquad \mathring{\tau} \stackrel{\text{def}}{=} \mathbf{type}(c)}{\texttt{make<}n\texttt{>}\ c/\mu \longrightarrow \underbrace{\texttt{pause(); }\cdots\texttt{pause();}}_{n+1 \text{ fois } \texttt{pause}}\ (\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})/\mu[\ell \mapsto (\texttt{false},\underbrace{(c,\cdots\ c)}_{n \text{ fois}})]}$$

REDUCE-CREATE

$$\frac{\ell \text{ est une adresse fraîche} \qquad d_1(\mathring{\tau}),\cdots\ d_n(\mathring{\tau}) \text{ sont des valeurs non spécifiées de type } \mathring{\tau}}{\texttt{create}_{\mathring{\tau}}\texttt{<}n\texttt{>}\ ()/\mu \longrightarrow (\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})/\mu[\ell \mapsto (\texttt{false},d_1(\mathring{\theta}),\cdots\ d_n(\mathring{\theta}))]}$$

ϑ-LENGTH $\quad \vartheta(\texttt{length},(\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),\mu) \stackrel{\text{def}}{=} (n{:}\texttt{int<16>})/\mu$

ϑ-GET-WAIT $\quad \vartheta(\texttt{get},((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i),\mu) \stackrel{\text{def}}{=} (\texttt{pause ();}\ \texttt{get}((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i))/\mu$
si $\mu(\ell) = (\texttt{true},v)$

ϑ-GET $\quad \vartheta(\texttt{get},((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i),\mu)$
$\stackrel{\text{def}}{=} (\%\texttt{acquire}_\ell\ ();\ \texttt{pause ();}\ \%\texttt{release}_\ell();\ c_i)/\mu$
si $\mu(\ell) = (\texttt{false},(c_0,\cdots c_i,\cdots v_{n-1}))$

ϑ-SET-WAIT $\quad \vartheta(\texttt{set},((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i,c),\mu)$
$\stackrel{\text{def}}{=} (\texttt{pause ();}\ \texttt{set}((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i,c))/\mu$
si $\mu(\ell) = (\texttt{true},v)$

ϑ-SET $\quad \vartheta(\texttt{set},((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i,c_i'),\mu)$
$\stackrel{\text{def}}{=} (\%\texttt{acquire}_\ell();\ \texttt{pause ();}\ \%\texttt{release}_\ell())/\mu'$
où $\mu' = \mu[\ell\cdot i \mapsto (\texttt{false},(c_0,\cdots\ c_{i-1},c_i',c_{i+1},\cdots c_{n-1}))]$
si $\mu(\ell) \stackrel{\text{def}}{=} (\texttt{false},(c_0,\cdots\ c_{i-1},c_i,c_{i+1},\cdots c_{n-1}))$

ϑ-ACQUIRE $\quad \vartheta(\%\texttt{acquire}_\ell,(),\mu) \stackrel{\text{def}}{=} ()/\mu[\ell \mapsto (\texttt{true},v)]$ si $\mu(\ell) = (\texttt{false},v)$

ϑ-RELEASE $\quad \vartheta(\%\texttt{release}_\ell,(),\mu) \stackrel{\text{def}}{=} ()/\mu[\ell \mapsto (\texttt{false},v)]$ si $\mu(\ell) = (\texttt{true},v)$

Indice hors des bornes

ϑ-GET-ERR $\quad \vartheta(\texttt{get},\overbrace{((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i)}^{args},\mu) \stackrel{\text{def}}{=} \texttt{halt (get}\ args\texttt{)}/\mu$ si $i \notin \{0,\cdots n-1\}$

ϑ-SET-ERR $\quad \vartheta(\texttt{set},\overbrace{((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}),i,c)}^{args},\mu) \stackrel{\text{def}}{=} \texttt{halt (set}\ args\texttt{)}/\mu$ si $i \notin \{0,\cdots n-1\}$

FIGURE 6.4 Sémantique d'ECLAT avec mémoire partagée

Comme il y a un verrou par tableau, les accès parallèles à des tableaux différents sont exécutés simultanément, c'est-à-dire au cours du même cycle d'horloge. Par exemple, l'expression définie sur la partie gauche de la figure 6.5 se comporte comme indiqué dans le chronogramme de la partie droite :

au cycle 0 : les deux tableaux (de taille 1) sont créés instantanément, puis la branche gauche du parallèle (`x`) acquiert le verrou associé au tableau `a` (ce qui est noté $\mathcal{A}_{\mathtt{a}}$ dans le chronogramme), puis elle commence immédiatement un accès mémoire matérialisé par une *pause* (notée "-") comme spécifié par l'appel à `%pause` dans la règle ArraySet (figure 6.4) ; puis la branche droite (`y`) fait de même avec le tableau `b` ($\mathcal{A}_{\mathtt{b}}$) ; On notera l'ordre de réduction gauche-droite des branches parallèle, comme spécifié par les règles ReduceParVal et ReduceParPause définies au chapitre 4.

au cycle 1 : `x` relâche le verrou de `a` (ce qui est noté $\mathcal{R}_{\mathtt{a}}$) ; puis, dans le même cycle, `y` relâche le verrou de `b` ($\mathcal{R}_{\mathtt{b}}$).

```
1 let a = create<1> () in
2 let b = create<1> () in
3 let (x,y) = (set(a,0,42) || set(b,0,43)) in ()
```

cycles :	0	1
x :	$\mathcal{A}_{\mathtt{a}}$; -	$\mathcal{R}_{\mathtt{a}}$
y :	$\mathcal{A}_{\mathtt{b}}$; -	$\mathcal{R}_{\mathtt{b}}$

FIGURE 6.5 Accès simultanés à deux tableaux différents

Dans un souci de sûreté et de simplicité, nous rejetons statiquement dans les programmes sources tout bloc `exec` *réinitialisable* (par une expression autre que `false`) comportant des accès à un tableau défini hors de ce bloc `exec`. Par exemple, le programme suivant est rejeté :

```
let main i =
  let t = create<1> () in
  let _ = exec set(t,0,42) default () reset i in
  (o,rdy) ;;
```

En effet, avec la sémantique présentée figure 6.4, si le calcul est réinitialisé alors qu'un verrou est pris, le verrou n'est pas relâché, ce qui cause blocage.

Cette vérification est réalisée par une analyse statique sur la représentation intermédiaire du compilateur (Core-Eclat).

6.2.2 Programmation concurrente

La sémantique d'Eclat fournit aux programmeuses et programmeurs un mécanisme prévisible pour gérer la concurrence :

- *ordre de réduction gauche-droite* : si deux expressions e_{left} et e_{right}, commençant au même cycle, sont mises en parallèle (par exemple, $(e_{\mathsf{left}} \| e_{\mathsf{right}})$), alors e_{left} se réduit toujours avant e_{right} ;

- *atomicité de la composition séquentielle* : si l'accès à un tableau ℓ est suivi séquentiellement par l'expression e (par exemple, `(get(`ℓ`,0);`e`)`), alors e se réduit toujours avant tout accès concurrent à ℓ non encore effectué. En particulier, e peut contenir des accès à ℓ, qui seront alors prioritaires.

Par exemple, il n'y a pas de *data race* dans le programme défini sur la figure 6.6. En déroulant les règles de réduction ECLAT, nous avons la garantie qu'il s'évalue toujours en `43`, comme le montre le chronogramme :

au cycle 0 : le tableau `a` de taille `1` est créé ; puis la première liaison (`x`) prend le verrou et commence une écriture (`-`) ; puis la deuxième liaison (`y`) tente de prendre le verrou mais ne le peut pas (✗) car le verrou est déjà tenu par `x`.

au cycle 1 : `x` relâche le verrou et *de façon atomique aussi loin que possible* : il acquiert à nouveau ce même verrou et commence une écriture ; `y` doit alors attendre, car le verrou est déjà tenu par `x`.

au cycle 2 : `x` relâche le verrou et renvoie la valeur `44` ; puis `y` acquiert le verrou et commence une lecture.

au cycle 3 : `y` relâche le verrou et le programme retourne la valeur `43`.

```
1 let a = create<1> () in
2 let (x,y) = ((set(a,0,42);
3                 set(a,0,43); 44)
4               || get(a,0))
5 in y
```

cycles :	0	1	2	3
x :	$\mathcal{A}_a$; -	$\mathcal{R}_a$;$\mathcal{A}_a$ -	$\mathcal{R}_a$	
y :	✗	✗	$\mathcal{A}_a$; -	$\mathcal{R}_a$

FIGURE 6.6 Section critique en ECLAT

Notons que cet ordonnancement strict des opérations n'empêche pas l'écriture de certains types de programmes dont le comportement ne peut pas être prédit statiquement (si ce n'est en déroulant les règles de réductions). Considérons par exemple une fonction `f`, définie ci-dessous, qui comporte trois accès mémoire (deux écritures en parallèle suivies d'une lecture). L'ordre d'écriture (et donc la valeur finalement lue) dépend de la durée des deux appels à une fonction `collatz` qui a été définie sur la figure 3.13. Or, il s'agit là de deux calculs irréguliers dont les durées dépendent fortement des arguments `i` et `j` de `f`, et donc de l'environnement physique à chaque appel à `f`.

```
let f (i,j) =
   let _ = ( (collatz(i); set((a,0),1))
           || (collatz(j); set((a,0),42)) ) in get(a,0) ;;
```

La figure A.5 (en annexe A) montre une *net-list* de circuit synthétisée à partir d'un calcul ECLAT concurrent similaire à celui discuté ci-dessus. Cela illustre le fait que la concurrence dans les programmes ECLAT se traduit en dépendances de données entre branches parallèles,

c'est-à-dire des fils reliant les réalisations matérielles de deux branches. Ces dépendances sont résolues statiquement, rendant le circuit totalement déterministe bien que non prédicitible (car dépend de l'évolution de l'environnement au cours du temps).

6.2.3 Equité

Lorsque l'on programme en ECLAT, on peut se demander si la gestion des accès concurrents à la mémoire est équitable (*fair*).

Propriété : *Les accès aux tableaux ECLAT sont équitables* : reprendre un verrou à l'infini demande une fonction récursive, qui implique une pause à chaque appel et donc l'opportunité pour les suivants de prendre le verrou. Lorsqu'un suivant prend le verrou, les précédents perdent la priorité compte tenu de l'ordre de réduction gauche-droite des branches parallèles.

Par exemple, soit n expressions $(e_1 \| e_2 \| \cdots e_n)$ mises en parallèle et soit $i < j < k < n$. Si e_j détient un verrou tandis que e_i et e_k attendent sur ce verrou, alors e_k prendra toujours le verrou avant e_i. Cela découle de l'*ordre de réduction gauche-droite* des branches parallèles en ECLAT (section 6.2.2) : à chaque cycle, e_i est réduit avant e_j, et donc avant que e_j ne relâche le verrou, qui peut alors être pris par e_k avant e_i. Dès que e_k termine sa réduction en ayant relâché le verrou, le cycle courant se termine, puis e_1 est prioritaire pour prendre le verrou puisqu'il se réduit en premier.

Nous illustrons se comportement sur la figure 6.7. Dans ce programme, chaque accès i au tableau débute dès que le verrou de `a` est libre ; la politique de priorité suit systématiquement l'ordre de réduction gauche-droite de construction parallèle $(e_1 \| e_2)$ d'ECLAT.

Le chronogramme de la figure 6.7 donne la trace d'exécution correspondante. Un point intéressant se produit aux cycles 4 et 5 :

au cycle 4 : aucun accès mémoire ne commence ; en effet, `x` *ne peut pas* prendre le verrou car `z` le détient déjà depuis le cycle 3 ; puis, `y` *ne peut pas* prendre le verrou pour la même raison ; puis `z` relâche le verrou et continue avec `f(2)` ;

au cycle 5 : `x` prend le verrou (qui a été relâché par `z` au cycle 4) et début un accès ; puis `y` *ne peut pas* prendre le verrou car il est déjà détenu par `x` ; on se retrouve alors exactement dans la même configuration qu'au cycle 1, ce qui illustre la manière dont les accès concurrents s'exécutent en ECLAT de façon équitable.

Notons que la construction $(e \| e')$ n'est pas commutative (c'est une spécificité de ECLAT), mais reste associative. Réduire $((e_1 \| e_2) \| e_3)/\mu$ revient à réduire e_1/μ en e'_1/μ_1 et e_2/μ_1 en e'_2/μ_2, et ensuite e_3/μ_2 en e'_3/μ_3. Réduire $(e_1 \| (e_2 \| e_3))/\mu$ revient à réduire e_1/μ en e'_1/μ_1, et ensuite e_2/μ_1 en e'_2/μ_2 et e_3/μ_2 en e'_3/μ_3. Cela amène respectivement à $((e'_1 \| e'_2) \| e'_3)/\mu_3$ et $(e'_1 \| (e'_2 \| e'_3))/\mu_3$.

```
1 let a = create<3>() in
2 let rec f (i) = set(a,i,i); f(i) in (* boucle infinie *)
3 let ((x,y),z) = ( (f(0) ‖ f(1)) ‖ f(2) ) in ()
```

cycles :	0	1	2	3	4	5	
x :	f(0)	$\mathcal{A}_a$; -	$\mathcal{R}_a$;f(0)	✗	✗	$\mathcal{A}_a$; -	···
y :	f(1)	✗	$\mathcal{A}_a$; -	$\mathcal{R}_a$;f(1)	✗	✗	···
z :	f(2)	✗	✗	$\mathcal{A}_a$; -	$\mathcal{R}_a$;f(2)	✗	···

FIGURE 6.7 Entrelacement équitable des accès mémoire dans un programme ECLAT

6.3 Compilation d'Eclat avec mémoire partagée

Dans cette section, on amende le flot de compilation d'ECLAT pour implanter l'extension d'ECLAT avec mémoire partagée (section 6.2). Il faut pour cela adapter les deux niveaux de langages intermédiaires (CORE-ECLAT et MINI-HDL), puis étendre le schéma de normalisation d'ECLAT vers CORE-ECLAT, la compilation de CORE-ECLAT vers MINI-HDL et la traduction de MINI-HDL vers VHDL.

6.3.1 Adapter Core-Eclat

La figure 6.8 définit l'extension de CORE-ECLAT avec des tableaux en mémoire partagée. On notera l'absence des opérateurs `%acquire` et `%release` qui n'apparaissent pas dans les programmes sources ECLAT et ont été introduites uniquement pour modéliser les accès mémoire dans la sémantique opérationnelle du langage.

expression normalisée	$\mathring{e}$	$::= \cdots \mid$ `let` $\ell =$ `create`$_{\mathring{\tau}}$`<`$\mathring{\theta}$`>` `()` `in` $\mathring{e}$ $\mid$ `get` (ℓ, a) $\mid$ `set` (ℓ, a, a)
atome	a	$::= \cdots \mid$ `length` x
taille normalisée	$\mathring{\theta}$	$::= n$
schéma de type normalisé	$\mathring{\sigma}$	$::= \cdots \mid \mathring{\tau}$ `array<`$\mathring{\theta}$`>`

FIGURE 6.8 Extension de la syntaxe de CORE-ECLAT avec mémoire partagée

La construction `make<`θ`>` est réécrite en `create`$_{\mathring{\theta}}$`<`θ`>` avec `let/rec` pour parcourir le tableau et ainsi forcer son initialisation à la valeur donnée.

6.3.2 Adapter la normalisation d'Eclat vers Core-Eclat

L'ajout de tableaux dans les programmes sources ECLAT implique des adaptations dans le schéma de normalisation d'ECLAT vers CORE-ECLAT. En particulier, les fonctions récursives terminales ne peuvent plus être partagées lorsqu'elles prennent des tableaux (de type

$\mathring{\tau}$ `array<`θ`>`) en paramètre ; en effet, les tableaux ne sont pas des valeurs MINI-HDL : ils ne peuvent pas être stockés dans des variables temporaires (comme le fait le compilateur ECLAT pour implanter le passage de paramètre). Par conséquent, les fonctions prenant des tableaux en paramètres sont *inlinées*, à la manière de ce qui est fait pour spécialiser les fonctions d'ordre supérieur (section 5.9.2).

Il faut noter que la globalisation par λ-lifting (sous-section 5.9.3) ajoute des paramètres de fonctions supplémentaires (l'environnement), y compris des fonctions des tableaux (à l'exception des déclarations globales, voir sous-section 3.2.5.1). Le processus combiné de spécialisation et globalisation doit donc être répété jusqu'à parvenir à un point fixe. Ce processus a été conçu suivant un argument sur la *terminaison* de la compilation :

(i) après globalisation, il n'y a plus de fonctions locales et le nombre de fonctions dans le programme ne change pas ;

(ii) la spécialisation réduit le nombre de fonctions, puisque les fonctions prenant des fonctions ou des tableaux en argument sont *inlinées* ; La duplication de code qui en résulte ne fait pas augmenter le nombre de fonctions du programme (d'après *(i)*).

(iii) nous rejetons une classe de programmes que nous ne savons pas compiler ; cela correspond au cas où le corps d'une fonction récursive crée localement une fonction ou un tableau et passe cette fonction ou ce tableau en paramètre de l'appel récursif : il s'agit d'une forme d'allocation dynamique qui n'est pas compatible avec les choix d'implantation d'ECLAT.

6.3.3 Adapter Mini-HDL

Des constructions d'accès mémoire et des verrous explicites sont ajoutés à MINI-HDL (figure 6.9). L'instruction `read`$_\ell$ a lit à l'indice a du tableau ℓ, ce qui a pour effet de placer la valeur lue dans une variable spéciale appelée `data`$_\ell$ par convention. L'instruction `write`$_\ell$ `(`a`,`a'`)` assigne la valeur de a' à l'élément d'indice calculé a dans le tableau ℓ. L'instruction `stop_writing`$_\ell$ `()` termine l'écriture dans ℓ. Les accès concurrents doivent être séquentialisés explicitement à l'aide de verrous [2]. En effet, le comportement est laissé non spécifié en cas d'accès simultanés à un tableau ℓ, qu'ils soient en lecture (R/R), en écriture (W/W) ou les deux à la fois (R/W). Les construction `acquire`$_\ell$ `()` et `release`$_\ell$ `()` rendent les manipulations de verrous explicites en modifiant une variable booléenne spéciale (appelée par convention `lock`$_\ell$) associée à chaque tableau.

Nous devons ajouter des règles de typage pour les nouvelles constructions introduites en MINI-HDL. La déclaration de tableau `static/in` étend l'environnement de typage avec un nom de tableau ℓ, de type $\mathring{\tau}$ `array<`$\mathring{\theta}$`>`, ainsi qu'un verrou `lock`$_\ell$ associé à ℓ, de type `bool` (règle TY-STATIC-ARRAY).

2. MINI-HDL est un langage intermédiaire qui constitue une cible de compilation sémantiquement plus simple que VHDL. Ce n'est pas un langage de programmation sûr. La compilation des constructions d'ECLAT garantit par construction que le code MINI-HDL engendré maintient certains invariants. Par exemple, un verrou ne doit pas être relâché s'il est déjà libre, il ne doit pas être pris s'il est déjà pris, et les accès mémoire doivent toujours être protégés par des verrous ; ce à quoi veille le schéma de compilation de CORE-ECLAT vers MINI-HDL.

instruction	s	$::= \cdots$
déclaration de tableau		$\mid$ `static` ℓ : $\mathring{\tau}$ `array<`n`>` `in` s
lecture		$\mid$ `read`$_\ell$ a
écriture		$\mid$ `write`$_\ell$ (a, a') $\mid$ `stop_writing`$_\ell$ `()`
gestion de verrous		$\mid$ `acquire`$_\ell$ `()` $\mid$ `release`$_\ell$ `()`

FIGURE 6.9 Syntaxe de MINI-HDL étendue avec le mémoire RAM

$$\textsc{Ty-static-array}\quad \frac{\Gamma[\ell : \mathring{\tau}\ \texttt{array<}\mathring{\theta}\texttt{>}][\texttt{lock}_\ell : \texttt{bool}] \vdash_{\texttt{s}} s : \texttt{unit}}{\Gamma \vdash_{\texttt{s}} \texttt{static}\ \ell\ :\ \mathring{\tau}\ \texttt{array<}\mathring{\theta}\texttt{>}\ \texttt{in}\ s : \texttt{unit}}$$

Les instructions `read`$_\ell$ et `write`$_\ell$ sont bien typées si le tableau ℓ est bien présent et du bon type dans l'environnement de typage et si l'indice est un entier (règles TY-READ et TY-WRITE).

$$\textsc{Ty-Read}\quad \frac{\Gamma(\ell) = \mathring{\tau}\ \texttt{array<}\mathring{\theta}\texttt{>} \qquad \Gamma(x) = \texttt{int<}\mathring{\theta}\texttt{>}}{\Gamma \vdash_{\texttt{s}} \texttt{read}_\ell\ x : \texttt{unit}}$$

$$\textsc{Ty-Write}\quad \frac{\Gamma(\ell) = \mathring{\tau}\ \texttt{array<}\mathring{\theta}\texttt{>} \qquad \Gamma(x) = \texttt{int<}\mathring{\theta}\texttt{>} \qquad \Gamma \vdash_{\texttt{a}} a : \mathring{\tau}}{\Gamma \vdash_{\texttt{s}} \texttt{write}_\ell\ (x, a) : \texttt{unit}}$$

Les instruction `acquire`$_\ell$ et `release`$_\ell$ sont bien typées si le verrou `lock`$_\ell$ est bien présent et de type `bool` dans l'environnement de typage (règles TY-ACQUIRE et TY-RELEASE).

$$\textsc{Ty-Stop-writing}\quad \frac{\Gamma(\ell) = \mathring{\tau}\ \texttt{array<}\mathring{\theta}\texttt{>}}{\Gamma \vdash_{\texttt{s}} \texttt{stop_writing}_\ell\ \texttt{()} : \texttt{unit}}$$

$$\textsc{Ty-Acquire}\quad \frac{\Gamma(\texttt{lock}_\ell) = \texttt{bool}}{\Gamma \vdash_{\texttt{s}} \texttt{acquire}_\ell : \texttt{unit}}$$

$$\textsc{Ty-Release}\quad \frac{\Gamma(\texttt{lock}_\ell) = \texttt{bool}}{\Gamma \vdash_{\texttt{s}} \texttt{release}_\ell : \texttt{unit}}$$

La figure 6.10 définit la sémantique comportementale des nouvelles constructions de tableaux de MINI-HDL. La déclaration de tableau (`static/in`) initialise en mémoire le pointeur de lecture `data`$_\ell$, la taille `length`$_\ell$ et le verrou `lock`$_\ell$ associés au tableau ℓ déclaré (règle REDUCE-ARRAY-DECL). La valeur initiale de `data`$_\ell$ est la valeur par défaut $d(\mathring{\tau})$ inférée à partir de l'annotation de type $\mathring{\tau}$ (on reprend ici la fonction d'initialisation $d(\cdot)$ définie au chapitre 5, figure 5.15). La valeur initiale du verrou est `false` pour signifier que le verrou est disponible.

L'instruction `acquire`$_\ell$ `()` prend le verrou `lock`$_\ell$ si celui-ci est disponible [3] (règle REDUCE-S-ACQUIRE). L'instruction `release`$_\ell$ `()` relâche le verrou `lock`$_\ell$ associé à ℓ (règle REDUCE-S-RELEASE) ; la prémisse $\mu(\texttt{lock}_\ell) = \texttt{true}$ signifie qu'il n'est pas autorisé de relâcher un

3. Le programme doit consulter le verrou `lock`$_\ell$ au préalable pour s'assurer que celui-ci peut être pris, comme dans la règle COMPILE-GET (section 6.3.4).

REDUCE-ARRAY-DECL

$$\frac{n > 0 \qquad \mu' \stackrel{\text{def}}{=} \mu[(\ell \cdot 0) \mapsto d(\mathring{\tau})] \ \cdots \ [(\ell \cdot n-1) \mapsto d(\mathring{\tau})] \qquad s/\mu'[\texttt{data}_\ell \mapsto d(\mathring{\tau})][\texttt{length}_\ell \mapsto n][\texttt{lock}_\ell \mapsto \texttt{false}] \hookrightarrow_{\texttt{s}} \mu''}{\texttt{static}\ \ell : \mathring{\tau}\ \texttt{array<}n\texttt{>}\ \texttt{in}\ s/\mu \hookrightarrow_{\texttt{s}} \mu''}$$

REDUCE-S-STOP-WRITING

$$\frac{\mu(\texttt{lock}_\ell) = \texttt{true}}{\texttt{stop-writing}_\ell\ ()/\mu \hookrightarrow_{\texttt{s}} \mu}$$

REDUCE-S-ACQUIRE

$$\frac{\mu(\texttt{lock}_\ell) = \texttt{false}}{\texttt{acquire}_\ell\ ()/\mu \hookrightarrow_{\texttt{s}} \mu[\texttt{lock}_\ell \mapsto \texttt{true})]}$$

REDUCE-S-RELEASE

$$\frac{\mu(\texttt{lock}_\ell) = \texttt{true}}{\texttt{release}_\ell\ ()/\mu \hookrightarrow_{\texttt{s}} \mu[\texttt{lock}_\ell \mapsto \texttt{false})]}$$

REDUCE-S-WRITE

$$\frac{\mu(\texttt{lock}_\ell) = \texttt{true} \qquad n \stackrel{\text{def}}{=} \mu(\ell \cdot \texttt{length}) \qquad \mu(x) = i \qquad 0 \le i < n \qquad \mu \vdash a \hookrightarrow_{\texttt{a}} c}{\texttt{write}_\ell\ (x\texttt{,}a)/\mu \hookrightarrow_{\texttt{s}} \mu[\ell \cdot i \mapsto c)]}$$

REDUCE-S-READ

$$\frac{\mu(\texttt{lock}_\ell) = \texttt{true} \qquad n \stackrel{\text{def}}{=} \mu(\ell \cdot \texttt{length}) \qquad \mu(x) = i \qquad 0 \le i < n \qquad \mu(\ell \cdot i) = c}{\texttt{read}_\ell\ x/\mu \hookrightarrow_{\texttt{s}} \mu[\texttt{data}_\ell \mapsto c]}$$

FIGURE 6.10 Sémantique comportementale de MINI-HDL avec mémoire partagée

verrou déjà relâché. L'écriture d'une valeur c à l'indice i dans un tableau ℓ lie c à l'adresse mémoire $\ell \cdot i$ (règle REDUCE-S-WRITE). L'instruction `stop_writing ()` est un marqueur de fin d'écriture ; il sert à positionner l'entrée `w_enable` du bloc RAM à `0` (cf. figure 6.1). La lecture de l'élément d'indice i dans un tableau ℓ rend accessible la valeur de cet élément dans $\texttt{data}_\ell$ (règle REDUCE-S-READ).

6.3.4 Adapter la compilation de Core-Eclat vers Mini-HDL

La compilation des tableaux CORE-ECLAT vers MINI-HDL requiert uniquement de traiter les constructions `create<`$\mathring{\theta}$`>`, `get` et `set`. L'expression `length` a appartient à la classe syntaxique des atomes, qui appartient à la fois à CORE-ECLAT et à MINI-HDL.

Les constructions `get` et `set` doivent effectuer une *pause* entre le début et la fin de l'accès mémoire ; cela se modélisera par une transition $\texttt{continue}_{id}\ f_{\mathsf{read}}$ dans la machine à états en cours de construction par le schéma de compilation de CORE-ECLAT vers MINI-HDL.

Les déclarations de tableaux CORE-ECLAT sont traduites en déclaration de tableau MINI-HDL (`static/in`). On rappelle que l'identifiant id désigne le bloc $\texttt{automaton}_{id}$ en cours de définition.

COMPILE-ARRAY-DECL

$$\frac{\rho/id \vdash \textbf{COMPILE}\ \mathring{e}\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s, ts/\omega \qquad s' \stackrel{\text{def}}{=} \texttt{static}\ y : \mathring{\tau}\ \texttt{array<}n\texttt{>}\ \texttt{in}\ s}{\rho/id \vdash \textbf{COMPILE}\ \texttt{let}\ y = \texttt{create}_{\mathring{\tau}}\texttt{<}n\texttt{>}\ ()\ \texttt{in}\ \mathring{e}\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s', ts/\omega}$$

Le compilé de la construction `get (`y`,`a`)` vers x suivi de k (c'est-à-dire un accès en lecture au tableau y) attend sur le verrou $\texttt{lock}_y$, puis prend le verrou et lance une lecture à l'indice donné par l'expression a dans le tableau y, puis attend un cycle (via le changement d'état $\texttt{continue}_{id}\ f_{\mathsf{read}}$ qui modélise cette *pause* d'un cycle) puis assigne à x la donnée lue $\texttt{data}_y$, puis relâche le verrou et exécute k (règle COMPILE-GET). Comme précédemment, ts désigne un ensemble de transitions locales dans la machine à états englobante en cours de construction.

COMPILE-GET

f_{wait} et f_{read} sont des noms frais

$$\begin{array}{rcl} s & \stackrel{\text{def}}{=} & \left[\begin{array}{l} \texttt{if lock}_y\ \texttt{then continue}_{id}\ f_{\mathsf{wait}} \\ \texttt{else (acquire}_y\ \texttt{(); read}_y\ a\texttt{; continue}_{id}\ f_{\mathsf{read}}\texttt{)} \end{array}\right. \\ ts & \stackrel{\text{def}}{=} & \left\{ (f_{\mathsf{wait}} : s),\ (f_{\mathsf{read}} : \texttt{(release}_y\ \texttt{(); } x \coloneqq \texttt{data}_y\texttt{; } k)) \right\} \end{array}$$

$$\rho/id \vdash \textbf{COMPILE}\ \ \texttt{get (}y\texttt{,}a\texttt{)}\ \ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s, ts/\emptyset$$

Similairement, le compilé de la construction `set ((`y`,`a`),`a'`)` vers x suivi de k (c'est-à-dire un accès en écriture au tableau y) attend sur le verrou $\texttt{lock}_y$, puis prend le verrou et lance une écriture de la valeur de a' à l'indice donné par l'expression a dans le tableau y, puis attend un cycle (via le changement d'état $\texttt{continue}_{id}\ f_{\mathsf{write}}$), puis arrête l'écriture, relâche le verrou, assigne la valeur `()` à la variable résultat x et exécute k (règle COMPILE-SET).

COMPILE-SET

f_{wait} et f_{write} sont des noms frais

$$\begin{array}{rcl} s & \stackrel{\text{def}}{=} & \left[\begin{array}{l} \texttt{if lock}_y\ \texttt{then continue}_{id}\ f_{\mathsf{wait}} \\ \texttt{else (acquire}_y\ \texttt{(); write}_y\ \texttt{(}a\texttt{,}a'\texttt{); continue}_{id}\ f_{\mathsf{write}}\texttt{)} \end{array}\right. \\ ts & \stackrel{\text{def}}{=} & \left\{ (f_{\mathsf{wait}} : s),\ (f_{\mathsf{write}} : \texttt{(stop_writing}_y\ \texttt{(); release}_y\ \texttt{(); } x \coloneqq \texttt{(); } k)) \right\} \end{array}$$

$$\rho/id \vdash \textbf{COMPILE}\ \ \texttt{set (}y\texttt{,}a\texttt{,}a'\texttt{)}\ \ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s, ts/\emptyset$$

Les verrous servent aussi à implanter le partage de fonction externes (lorsqu'elles sont déclarées avec le mot clé `share`). Ce partage impose que la fonction externe appelée démarre son exécution lors du *tick* suivant (règle COMPILE-SHARED-EXTERNAL-CALL).

COMPILE-SHARED-EXTERNAL-CALL

$f \notin \mathsf{dom}(\rho)$ $\quad$ $wait_f$, $sync$, $result$ et rdy sont des noms frais

$$\begin{array}{rcl} s & \stackrel{\text{def}}{=} & \left[\begin{array}{l} (result, rdy) \coloneqq \texttt{run}_0\ f\ a\texttt{;} \\ \texttt{if}\ rdy\ \texttt{then (release}_f\texttt{(); } x \coloneqq result\texttt{; } k\texttt{) else continue}_{id}\ sync \end{array}\right. \\ s_0 & \stackrel{\text{def}}{=} & \texttt{if lock}_f\ \texttt{then continue}_{id}\ wait_f\ \texttt{else (acquire}_f\ \texttt{(); continue}_{id}\ sync\texttt{)} \end{array}$$

$$\rho/id \vdash \textbf{COMPILE}\ f\ a\ \textbf{TO}\ x\ \textbf{THEN}\ k\ \textbf{IS}\ s_0, \{wait_f : s_0, sync : s\}/\emptyset$$

Finalement, nous ajoutons des cas à la fonction **dispatch**$[\![]\!]$ qui gère le retour instantané des appels de fonctions récursives :

$$\begin{array}{rcl} \textbf{dispatch}[\![\texttt{static}\ \ell : \tau\ \texttt{in}\ s]\!]_\omega & \stackrel{\text{def}}{=} & \texttt{static}\ \ell : \tau\ \texttt{in}\ \textbf{dispatch}[\![s]\!]_\omega \\ \textbf{dispatch}[\![\texttt{release}_\ell\ \texttt{()}]\!]_\omega & \stackrel{\text{def}}{=} & \texttt{release}_\ell\ \texttt{()} \\ \textbf{dispatch}[\![\texttt{acquire}_\ell\ \texttt{()}]\!]_\omega & \stackrel{\text{def}}{=} & \texttt{acquire}_\ell\ \texttt{()} \\ \textbf{dispatch}[\![\texttt{read}_\ell\ a]\!]_\omega & \stackrel{\text{def}}{=} & \texttt{read}_\ell\ a \\ \textbf{dispatch}[\![\texttt{write}_\ell\ \texttt{(}a\texttt{,}a'\texttt{)}]\!]_\omega & \stackrel{\text{def}}{=} & \texttt{write}_\ell\ \texttt{(}a\texttt{,}a'\texttt{)} \\ \textbf{dispatch}[\![\texttt{stop_writing}_\ell\ \texttt{()}]\!]_\omega & \stackrel{\text{def}}{=} & \texttt{stop_writing}_\ell\ \texttt{()} \end{array}$$

6.3.5 Adapter la traduction de Mini-HDL vers VHDL

La traduction des nouvelles constructions syntaxiques de MINI-HDL (parmi les atomes et instructions) est définie comme suit. L'expression VHDL x`'length` donne la taille du tableau VHDL x (en nombre d'éléments).

$$
\begin{aligned}
\mathcal{C}_{\mathsf{a}}(\texttt{data}_x) &\stackrel{\text{def}}{=} \texttt{data_}x \\
\mathcal{C}_{\mathsf{a}}(\texttt{length } x) &\stackrel{\text{def}}{=} x\texttt{'length} \\
\mathcal{C}_{\mathsf{a}}(\texttt{lock}_x) &\stackrel{\text{def}}{=} \texttt{lock_}x \\
\\
\mathcal{C}_{\mathsf{s}}(\texttt{acquire}_x\ ()) &\stackrel{\text{def}}{=} \texttt{lock_}x\ \texttt{:= "1";} \\
\mathcal{C}_{\mathsf{s}}(\texttt{release}_x\ ()) &\stackrel{\text{def}}{=} \texttt{lock_}x\ \texttt{:= "0";} \\
\mathcal{C}_{\mathsf{s}}(\texttt{write}_x\ (a,a')) &\stackrel{\text{def}}{=} \left\lceil \begin{array}{l} \texttt{w_enable_}x\ \texttt{:= "1";} \\ \texttt{addr_}x\ \texttt{:=}\ \mathcal{C}_{\mathsf{a}}(a)\texttt{;} \\ \texttt{write_data_}x\ \texttt{:=}\ \mathcal{C}_{\mathsf{a}}(a')\texttt{;} \end{array} \right. \\
\mathcal{C}_{\mathsf{s}}(\texttt{stop_writing}_x\ ()) &\stackrel{\text{def}}{=} \texttt{w_enable_}x\ \texttt{:= "0";} \\
\mathcal{C}_{\mathsf{s}}(\texttt{read}_x\ a) &\stackrel{\text{def}}{=} \texttt{addr_}x\ \texttt{:=}\ \mathcal{C}_{\mathsf{a}}(a)\texttt{;} \\
\mathcal{C}_{\mathsf{s}}(\texttt{static } x\ :\ \mathring{\tau}\ \texttt{array<}n\texttt{> in } s) &\stackrel{\text{def}}{=} \mathcal{C}_{\mathsf{s}}(s)
\end{aligned}
$$

La fonction de compilation $\mathcal{C}_{\mathsf{ci}}(\cdot)$ (définie sur la figure 5.27) doit être amendée en sorte que l'architecture VHDL engendrée comporte les patrons d'accès aux blocs de mémoire RAM. On étend pour cela les fonctions **EXTRA-DÉCLARATIONS**(S) et **EXTRA-CONNEXIONS**(S).

Dans **EXTRA-DÉCLARATIONS**(S), nous ajoutons les déclarations suivantes, pour chaque déclaration `static` x : $\mathring{\tau}$ `array<`n`> in` s qui apparaît dans S :

```
type type_x is array (natural range 0 to n-1) of C_ty(τ̊);
signal x : type_x;
signal lock_x : C_ty(bool) := "0";
signal write_data_x : C_ty(τ̊) := d(τ̊);
signal data_x : C_ty(τ̊) := d(τ̊);
signal addr_x : integer range 0 to n-1 := 0;
signal w_enable_x : C_ty(bool) := "0";
```

Dans **EXTRA-CONNEXIONS**(S), nous ajoutons de nouvelles instantiations (s'exécutant concurremment du reste du programme). Chaque déclaration `static` x : $\mathring{\tau}$ `array<`n`> in` s qui apparaît dans S donne lieu à l'ajout d'un processus de la forme :

```
process (clk)
  begin
    if (rising_edge(clk)) then
      data_x <= x(addr_x);
    if w_enable_x = "1" then
      x(addr_x) <= write_data_x;
    end if;
  end if;
end process;
```

Il reste enfin à amender la fonction **EXTRA-LISTE**(S) en y ajoutant les nouveaux signaux utilisés par chaque tableau x, à savoir `lock_`x, `data_`x, `w_enable_`x, `addr_`x et `write_data_`x.

6.4 Discussion

Les calculs en ECLAT, tels que présentés au chapitres 3 à 5, étaient purement *déclaratifs*[4]. Ils étaient programmés en définissant des fonctions récursives terminales et en les composant séquentiellement ou en parallèle. Dans ce contexte, modéliser le déroulement des calculs de façon synchrone présentait un intérêt limité : cela permettait essentiellement de fixer la durée des calculs longs, qui influe directement sur le flot de contrôle du programme réactif et donc sur les valeurs de sorties.

Par exemple, le programme suivant fait osciller sa sortie entre `41` et `42` à chaque cycle. En effet, le calcul `delay ()` a une durée d'*exactement* un cycle, d'où l'alternance entre la valeur par défaut `41` et le résultat de `delay ()`, à savoir `42`.

```
let main () =
  let (o,rdy) = exec
                  let rec delay () = 42 in
                  delay ()
                default 41
  in o ;;
```

L'extension d'ECLAT avec mémoire partagée augmente considérablement l'expressivité des calculs que l'on peut programmer dans le langage. Les calculs sont alors intrinsèquement concurrents, avec *a priori* les difficultés que cela peut poser en terme de programmation.

Or justement, le modèle synchrone d'ECLAT concilie concurrence et déterminisme, évitant ainsi les principaux écueils de la programmation parallèle avec mémoire partagée, tout en maintenant une implantation matérielle efficace exploitant le parallélisme physique.

4. Rappelons que c'est de là que vient le nom *éclat*.

Pour cela, la construction $(e_1 \| e_2)$ d'ECLAT synchronise à chaque cycle les calculs e_1 et e_2, mais force l'ordre de réduction de la gauche vers la droite dans le cycle, comme en REACTIVE C [36]. C'est un choix non standard, guidé par l'implantation, mais qui offre en retour du déterminisme et un comportement relativement intuitif sans surcoût à l'exécution (en nombre de cycles d'horloge).

Les accès mémoire en ECLAT sont protégés au moyen de verrous. De par le modèle d'exécution intrinsèquement séquentiel et synchronisé d'ECLAT (caractérisé par cet ordre de réduction *gauche-droite* des branches parallèles) la prise de verrous est toujours équitable et ne requiert aucun mécanisme dynamique d'arbitrage (*e.g.*, un jeton tournant). Au chapitre suivant (à la section 7.7), nous utiliserons ces aspects avancés de programmation concurrente en ECLAT pour développer des fermes de calculs et nous nous en servirons pour implanter des squelettes de parallélisme efficaces et prédictibles.

7 Développement d'applications sur FPGA avec Eclat

Ce chapitre montre, à travers une série de programmes et de *benchmarks*, l'applicabilité d'Eclat pour le développement d'applications sur FPGA, tant du point de vue de l'expressivité que de la sûreté et de l'efficacité.

Les points clés étudiés sont : la description de circuits, la conception de systèmes réactifs avec mémoire partagée, le développement d'abstractions de programmation (ici des squelettes de parallélisme et une machine virtuelle). Tout l'intérêt d'Eclat est de pouvoir répondre à l'ensemble des problématiques posées par ces points clés.

La section 7.1 décrit en détails la méthodologie mise en œuvre dans ce chapitre pour évaluer les performances d'Eclat sur FPGA. À la section 7.2, nous définissons trois constructions dérivées d'Eclat utilisées dans ce chapitre. À la section 7.3, nous utilisons Eclat pour décrire un circuit interagissant avec des entrées/sorties physiques sur le FPGA. Il définit et instancie dix fois un modulateur de largeur d'impulsion pour faire varier l'intensité d'une rangée de LED.

À la section 7.4, nous présentons un jeu vidéo simple (le fameux jeu de Pong) utilisant des entrées/sorties plus avancées (accéléromètre et écran VGA) et réalisant des calculs de trajectoire.

À la section 7.5, nous présentons un programme Eclat calculant le nombre de solutions au problème des n reines pour un n donné. Une pile explicite est utilisée pour représenter l'espace de recherche (c'est-à-dire, l'ensemble des configurations à considérer, parmi lesquelles se trouvent les solutions). Cela montre qu'il est possible d'implanter des algorithmes récursifs généralistes en Eclat en combinant récursivité terminale et codage de structures de données.

À la section 7.6, nous proposons deux optimisations du problème des n reines en exploitant le compromis temps-espace, c'est-à-dire, réalisant un circuit plus gros (en nombre de cellules logiques) mais qui calcule plus vite.

À la section 7.7, nous présentons une application en style impératif avec mémoire partagée : le *Jeu de la vie*. Nous montrons que la mémoire partagée induit un coût important en terme de temps d'exécution, car les accès concurrents *doivent* être séquentialisés [1]. Nous proposons alors une technique pour exploiter le parallélisme à grain fin sur les éléments de tableaux (en mémoire partagée) via le compromis temps-espace tout en traitant séquentiellement les éléments de tableaux pour ainsi atteindre des gains de vitesse importants.

1. Cette contrainte vient du fonctionnement même des blocs de mémoire RAM.

À la section 7.8, nous décrivons des implantations de squelettes de parallélisme (`map`, `reduce` et `pipe`) en Eclat, opérant sur de la mémoire partagée. Le point central de cette section est la prédictibilité des performances. Nous proposons par exemple une version parallèle du problème des n reines utilisant les squelettes `map` et `reduce` : les gains sont comparables à ceux obtenus à la section 7.6, mais avec un meilleur contrôle sur le degré de parallélisme et sur le compromis temps-espace.

À la section 7.9, nous décrivons une implantation expérimentale de la machine virtuelle OCaml et sa bibliothèque d'exécution en Eclat (incluant un gestionnaire automatique de mémoire). L'interprète de la machine virtuelle (ainsi réalisé sous la forme d'un circuit) permet d'exécuter des programmes OCaml compilés en bytecode (par le compilateur `ocamlc`) stocké dans un tableau Eclat. Nous proposons aussi un mécanisme d'interopérabilité entre le bytecode d'OCaml et Eclat qui permet de s'affranchir (au moins en partie) de la barrière d'abstraction entre langage implanté (OCaml) et langage d'implantation (Eclat).

7.1 Méthodologie expérimentale

Nous ciblons un FPGA Intel Max10 intégré sur une carte Terasic DE10-Lite. Ce FPGA fonctionne à une fréquence de 50 MHz ; il comprend 50K éléments logiques (LE), 1638 Kbits (204 Ko) de mémoire RAM (BRAM) et 64 MB de mémoire externe (SDRAM). Le Max10 est un petit modèle de FPGA. La carte comprend des dispositifs physiques pré-câblés connectés aux entrées/sorties du FPGA. Cela inclut deux boutons, dix interrupteurs, dix LED, six afficheurs 7 segments, un accéléromètre et une sortie VGA. Tous ces dispositifs seront mis en œuvre ds les applications décrites ds ce chapitre.

Le choix d'un si petit modèle de FPGA vise à montrer qu'Eclat est suffisamment précis pour programmer des applications et exploiter le potentiel de calcul et de reconfiguration des FPGA, même les plus petits. L'approche reste compatible avec des modèles de FPGA ayant d'avantage de LE et de mémoire RAM. Il est à noter cependant que ces modèles sont généralement intégrés sur de véritables systèmes sur puce (*System on a Chip, SoC*) embarquant un processeur. Le processeur "hôte" appelle alors des circuits spécialisés réalisés sur le FPGA. Le but d'Eclat est de faciliter la programmation de ces circuits spécialisés, indépendamment d'un processeur présent ou non sur la carte FPGA.

7.1.1 Flot de conception de circuit

Chaque programme Eclat est compilé en VHDL à l'aide du compilateur Eclat. Le code VHDL obtenu est synthétisé par la chaîne d'outils Quartus 22.1 Lite [2].

Après synthèse, Quartus indique l'utilisation des ressources par le circuit engendré (en nombre de LE et en pourcentage d'occupation de la BRAM) ainsi que la fréquence maximale de fonctionnement du circuit. Cette fréquence maximale doit être supérieure à 50 MHz pour que le circuit soit correctement cadencé par l'horloge globale. Tous les programmes Eclat présentés

2. C'est une version récente, sortie en avril 2022.

dans ce manuscrit répondent aux contraintes temporelles (*timing requirements*) de la cible DE10-Lite : ils peuvent fonctionner en toute sécurité (vis-à-vis de ces contraintes temporelles) en utilisant l'horloge globale de 50 MHz.

7.1.2 Mesure des temps d'exécution

L'estimation du temps d'exécution des calculs programmés en ECLAT est mesurée, à la fois par simulation RTL en utilisant le logiciel GHDL [3] (c'est-à-dire, l'exécution d'une description de matériel VHDL par un simulateur sur CPU), et par exécution du circuit correspondant (engendré par QUARTUS) sur la carte DE10-Lite.

1. Les durées *inférieures à une seconde* sont mesurées par simulation RTL et confirmées par exécution sur la carte :
   - la simulation nous donne un temps d'exécution exact (en nombre de cycles [4]) ;
   - l'exécution sur la carte confirme que le résultat apparaît en quelque dixièmes de secondes et que cela correspond approximativement au nombre de cycles obtenu par simulation.
2. Les durées *supérieures à une seconde* sont mesurées uniquement sur la carte FPGA ; en effet, les temps de simulation RTL seraient alors très longs (plusieurs heures). La mesure du temps d'exécution sur FPGA se fait, de deux manière à la fois :
   - Premièrement, nous instrumentons le circuit avec un compteur de cycles (qui s'implante directement en ECLAT par `reg (fun` $c \rightarrow c+1$`) init` 0), aussi bien en synthèse qu'en simulation RTL. Les six afficheurs 7 segments de la carte DE10-lite peuvent représenter un entier hexadécimal sur 24 bits ($16^6 = 16\,777\,216$). C'est bien inférieur aux 50 millions de cycles par seconde de la carte (fonctionnant à 50MHz) lorsque les temps d'exécution sont important, mais cela suffit pour déduire une durée approchée (en négligeant les bits de poids faibles). Cela concorde avec l'observation (c'est-à-dire, la durée mesurée sur la vidéo). Obtenir une mesure exacte (en nombre de cycles) reste possible (en affichant alternativement les bits de poids forts et les bits de poids faibles), mais la précision de la mesure approchée est largement suffisante.
   - Deuxièmement, nous signalons le début et la fin du calcul par des affichages distincts sur les sorties physiques de la carte ; nous filmons l'exécution complète (allant jusqu'à une heure de calcul suivant les *benchmarks* réalisés) pour ensuite déterminer le temps d'exécution sur la vidéo (avec une précision de l'ordre du dixième de seconde, car les LED prennent un certain laps de temps à s'allumer).

3. `http://ghdl.free.fr`

4. Ce nombre est obtenu à l'aide d'un compteur (`reg (fun` $c \rightarrow c+1$`) init` 0) programmé directement en ECLAT ; le nombre de cycles écoulés est affiché par l'opérateur `print_int` d'ECLAT lors du passage à `true` d'une variable *rdy*. Cette variable *rdy* est connectée à la seconde sortie d'un bloc `exec` contenant le calcul dont on souhaite mesurer le temps d'exécution. La primitive `print_int` est utilisable uniquement en mode simulation. Pour la synthèse sur des cartes FPGA, `print_int` (comme d'autres primitives de simulation présente aussi en VHDL qui ne sont pas synthétisables) est éliminée par le compilateur ECLAT via l'option de compilation `-no-print` (voir annexe C).

7.1.3 Comparer Eclat sur FPGA et OCaml sur CPU

Des comparaisons sont données entre, d'une part, l'exécution sur FPGA de programmes Eclat, et d'autre part, l'exécution sur CPU de programmes similaires calculant la même valeur en suivant un même algorithme avec des structures de données proches. Le langage de programmation utilisé sur CPU est OCaml ; ce choix est motivé par la proximité syntaxique et sémantique des deux langages. Les programmes OCaml implantés sont *séquentiels* ; nous n'utilisons pas les traits de programmation parallèle d'OCaml (qui sont fournis à partir de la version 5 du langage), le but étant de montrer les différents gains de vitesse qui peuvent être obtenus en Eclat, sur FPGA, par rapport à une version séquentielle simple sur CPU.

Le CPU utilisé pour ces comparaisons est un processeur Intel core i7 avec 16 Go de RAM DDR3 et une fréquence de fonctionnement de 2.2 GHz. Le code OCaml est compilé en code natif à l'aide du compilateur OCaml `ocamlopt` (version `4.13`) avec optimisations activées (`-O2`). Si le temps d'exécution du calcul à mesurer sur CPU est trop court, nous l'exécutons de façon répétée des milliers de fois dans une boucle `for`, dans le programme OCaml.

D'un point de vue méthodologique, la comparaison entre OCaml sur CPU et Eclat sur FPGA a ses limites, car les deux cibles sont de nature très différentes. C'est pourquoi nous veillons à ne pas introduire de biais dans l'interprétation des résultats. Il faut pour cela tenir compte des différences suivantes :

— la fréquence du FPGA MAX 10 est 44 fois moindre que la fréquence du CPU (50MHz vs. 2.2GHz) ;

— la mémoire partagée d'Eclat est implantée sous forme de blocs RAM (BRAM) ; le temps d'accès à un bloc RAM est de un cycle (avec *au plus* un accès par cycle). La mémoire partagée sur CPU est une mémoire externe DDR3 avec trois niveaux de cache et des temps d'accès bien plus complexes à évaluer.

L'intérêt de cette comparaison est de motiver l'utilisation d'Eclat sur FPGA auprès des publics familiers des langages de programmation généralistes (comme OCaml). C'est aussi de comprendre les facteurs de gain possibles en utilisant des FPGA.

7.2 Macro-constructions dans le compilateur Eclat

Le compilateur effectue des propagations de copies dans les programmes sources. C'est pourquoi il autorise des variables là où des tailles était attendues ; *e.g.*, `vect_create<size>(true)` où `size` est une variable globale.

Dans ce contexte, le compilateur Eclat offre deux constructions dérivées (`generate` et `parfor`) ainsi qu'une fonction primitive `vect_mapi` pour dupliquer un calcul de multiples fois :

— étant donnés une expression e et deux entiers littéraux n et k, la construction dérivée

`parfor` $x = n$ `to` k `do` e `done` engendre l'expression :

$$\texttt{let}\ y = (e_n \| \cdots \| e_k)\ \texttt{in}\ () \qquad \text{avec} \begin{cases} y \text{ un nom frais} \\ i \in \{n, \cdots k\} \\ e_i \stackrel{\text{def}}{=} \texttt{let}\ x = i\ \texttt{in}\ \ e \end{cases}$$

Comme le résultat est `()`, les e_i font en général un effet de bord sur une mémoire partagée. Le compilateur autorise n et k à être des expressions évaluables statiquement (comme par exemple une variable résolue par propagation de constantes).

— étant donnés une fonction f, une expression e et un entier littéral n, la construction dérivée `generate` $f\ e_0\ n$, engendre l'expression :

$$f(0, f(1, f(\cdots, f(n-1, e_0)) \cdots))$$

C'est un schéma d'accumulation de type *fold right* déroulé statiquement et où la fonction f est dupliquée n fois. Ici aussi, n peut être une expression évaluable statiquement.

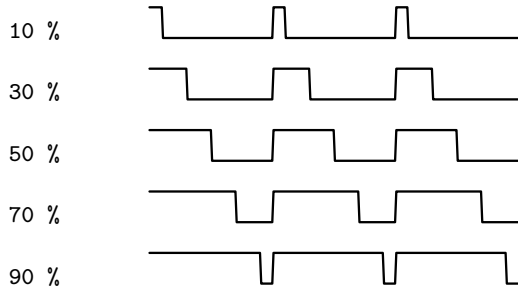
— La fonction primitive `vect_mapi(f,v)` applique la fonction `f` (de durée D) à chaque élément du vecteur `v` (en prenant comme premier argument la position dans le vecteur) et retourne un nouveau vecteur. Si la fonction `f` est instantanée (c'est-à-dire, $D = \mathbf{0}$), `vect_mapi(f,v)` est instantanée. La fonction `f` est dupliquée sur chacun des éléments du vecteur. Le schéma de type de `vect_mapi` est :

$$\forall\ 'N \cdot \forall\ 'd \cdot \forall\ 'B1 \cdot \forall\ 'B2 \cdot (\texttt{int} \times\ 'B1 \xrightarrow{'d}\ 'B2) \times\ 'B1\ \texttt{vect<}'N\texttt{>} \xrightarrow{'d}\ 'B2\ \texttt{vect<}'N\texttt{>}$$

Ces constructions seront utilisées dans la suite du chapitre. On ne peut pas les définir directement en ECLAT ; par exemple, on pourrait programmer une fonction *mapi* sur les vecteurs ECLAT en utilisant une fonction récursive terminale, mais dans ce cas il y aurait une pause d'un cycle entre chaque traitement d'un élément du vecteur.

7.3 Faire varier l'intensité d'une rangée de LEDs

Dans cette section, nous montrons la capacité d'ECLAT à décrire de façon précise et concise des circuits au niveau RT interagissant avec des entrées/sorties physiques. Nous ciblons la carte DE10-lite qui embarque un FPGA MAX 10 ; cette carte comporte dix LEDs qui peuvent être contrôlées depuis le FPGA. Si la sortie contrôlant une LED est à 1, la LED est allumée ; si elle est à 0 la LED est éteinte. Pour faire varier l'intensité lumineuse d'une LED, nous pouvons utiliser la technique de *modulation de largeur d'impulsions* (*Pulse Width Modulation, PWM*) [200]. Cette technique consiste à émettre une forme d'onde carrée (uniquement des 0 et des 1) et à en faire varier le rapport cyclique (*duty cycle*), c'est-à-dire le pourcentage de temps moyen passé à 1, comme ci-dessous :



Sur la figure 7.1 est défini en ECLAT un générateur d'onde triangulaire `triangle` (ligne 1 à 4) qui prend en argument une période en nombre de cycles. À chaque fois que `triangle` est appelé, il retourne instantanément la valeur de son état interne incrémenté modulo la période, puis met à jour l'état avec cette nouvelle valeur à la fin du cycle d'horloge. La fonction `modulation` (ligne 6 à 9) appelle `triangle`, qui retourne un entier `c` oscillant entre 0 et la période ; `modulation` retourne `true` si `c` est plus petit qu'une valeur `duty_cycle` supposée comprise entre 0 et la période et qui représente le rapport cyclique (entre 0 et 1) multiplié par la période.

```
1 let triangle(period) =
2   reg (fun c ->
3         if c >= period then 0 else c + 1
4       ) init 0 ;;
5
6 let modulation(duty_cycle,period) =
7   let c = triangle(period) in
8   let o = (c < duty_cycle) in
9   o ;;
```

FIGURE 7.1 Modulateur de largeur d'impulsions en ECLAT

Sur la figure 7.2 est définie une fonction `control_duty_cycle` qui permet de varier le rapport cyclique via deux entrées : `inc` pour l'augmenter et `dec` pour le diminuer.

Nous voulons contrôler ces deux entrées via deux boutons. Or, étant donné la fréquence d'horloge de 50MHz du FPGA MAX 10, appuyer sur un bouton pendant quelques millisecondes, représente plusieurs dizaines de milliers de cycles au cours desquels l'entrée `inc` ou `dec` vaut `true`. Nous voulons augmenter d'un cran le rapport cyclique à chaque fois que le bouton est pressé ; pour réaliser cela, nous avons besoin d'un détecteur de front montant dont la sortie vaut `true` pendant un cycle si d'une part son entrée et `true` et d'autre part la valeur de l'entrée au cycle précédent était `false`. Ce détecteur de front montant (`edge`) est défini à la ligne 1 ; il a déjà été défini dans l'exemple `ABRO` de la figure 3.20 et utilise une fonction auxiliaire `fby` qui a été définie sur la figure 3.18. La fonction `edge` détecte les changements de valeurs sur une entrée. La fonction `control_duty_cycle` augmente (*resp.* diminue) le rapport cyclique lorsque

le bouton `inc` (*resp.* `dec`) dévient `true`

```
1  let edge i = (not (fby(false,i))) & i ;;
2
3  let min(a,b) = if a < b then a else b ;;
4  let max(a,b) = if a > b then a else b ;;
5
6  let period = 1024 ;;
7  let step = period / 16 ;;
8
9  let control_duty_cycle(inc,dec) =
10   let dc = reg (fun v ->
11                 if edge(inc) then min(v + step, period) else
12                 if edge(dec) then max(v - step, 0) else
13                 v) init period
14   in dc ;;
```

FIGURE 7.2 Faire varier le rapport cyclique avec deux boutons

Nous définissons deux fonctions combinatoires, `min` et `max` (lignes 4 et 5), que nous appelons aux lignes 12 et 13 pour garder le rapport cyclique dans l'intervalle compris entre 0 et la période.

Finalement, sur la figure 7.3, nous définissons aux lignes 1 à 4 une fonction `pwm`, qui compose le variateur de rapport cyclique (`control_duty_cycle`) et le modulateur de largeur d'impulsion (`modulation`). Le point d'entrée du programme (`main`) est défini aux lignes 5 et 6.

La fonction `main` prend en entrées les valeurs des dix interrupteurs (`sw`) et des deux boutons (`inc` et `dec`). Nous utilisons la construction primitive `vect_mapi` (définie à la section 7.2) pour transformer point à point les valeurs des interrupteurs (qui sont des booléens *ouvert/fermé*) et donner en résultat les valeurs des dix LED en instanciant dix fois la fonction `pwm`. Les boutons servent à augmenter ou diminuer le rapport cyclique de chaque LED si l'interrupteur correspondant est ouvert : c'est pourquoi nous faisons deux conjonctions `b & inc_button` et `b & dec_button` où `b` et la valeur (*ouvert/fermé*) du `i`-ème interrupteur.

```
1  let pwm((inc,dec) : bool * bool) =
2    let dc = control_duty_cycle(inc,dec) in
3    modulation(dc,period) ;;
4
5  let main ((sw,(inc_button,dec_button)) : bool vect<10> * (bool * bool)) =
6    vect_mapi((fun (i,b) -> pwm(b & inc_button, b & dec_button)), sw) ;;
```

FIGURE 7.3 Point d'entrée qui instancie dix PWM

Il reste à faire l'association entre les paramètres du programme (*resp.* la valeur de retour) et les entrées (*resp.* les sorties physiques. Cette correspondance se fait en ligne de commandes avec l'option `-top="SW:10,KEY:2|LEDR:10"` du compilateur ECLAT (voir l'annexe C). Le format de cette option comporte les déclarations d'entrées suivies du caractère `|` suivies des déclarations de sorties. Chaque déclaration est de la forme $x{:}n$ où x est le nom d'une entrée (*resp.* d'une sortie) physique telle que définie dans un fichier de *mapping* et la taille de cette entrée (*resp.* cette sortie) en nombre de bits. Le compilateur vérifie que les tailles spécifiées en entrées et sorties sont compatibles avec le type de la fonction `main`.

La figure 7.4 donne une photographie de la carte FPGA en fonctionnement. Elle est reconfigurée avec cette application : chaque LED à son propre rapport cyclique mis à jour avec les deux boutons si l'interrupteur correspondant est ouvert, d'où les niveaux d'intensité différents de chaque LED.

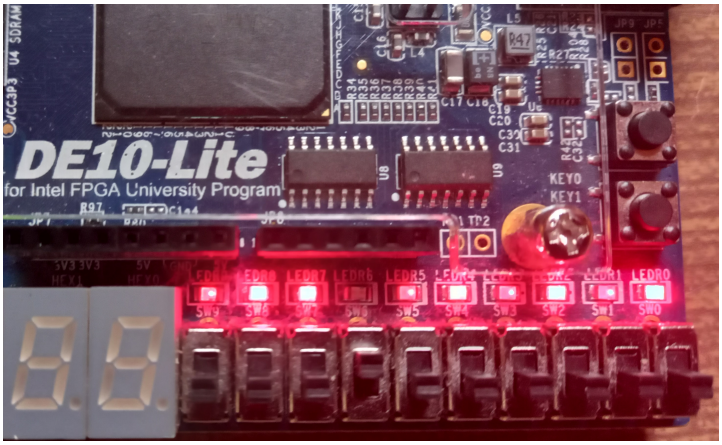


FIGURE 7.4 Variations de l'intensité des LED de la carte DE10-lite par PWM

7.4 Jeu de Pong

Le jeu de Pong est un jeu d'arcade en deux dimensions. Il se joue à deux joueurs ; l'un est placé sur le bord gauche de l'écran, l'autre sur le bord droit. Chaque joueur possède une raquette qu'il utilise pour renvoyer une balle dans l'autre camp. La balle rebondit sur les raquettes et sur les bords. Si la balle atteint le bord gauche (*resp.* droit), le joueur de droite marque un point (*resp.* perd un point).

Dans le cadre d'un stage à l'IRILL [5], Adrien Challes a réalisé une implantation de ce jeu en ECLAT. Le joueur de droite est contrôlé par l'utilisatrice ou l'utilisateur, le mouvement de la raquette est déterminé par l'inclinaison de l'accéléromètre (qui est un dispositif physique) présent sur la carte DE10-lite. La vitesse de la raquette influence la direction de la balle (pour obtenir un effet réaliste). Le joueur de gauche est émulé par le circuit ; il estime la trajectoire de la balle et déplace sa raquette à une certaine vitesse pour essayer de rattraper la balle. Le

5. Initiative pour la Recherche et l'Innovation sur le Logiciel Libre.

score s'affiche sur les afficheurs 7 segments. Lorsque le premier interrupteur est enclenché, la partie est suspendue et le mot `PAUSE` s'affiche sur les afficheurs 7 segments.

La gestion de l'accéléromètre est effectuée en VHDL, en utilisant le mécanisme[6] d'appel de fonctions externes d'Eclat. Le programme complet représente 700 lignes de VHDL (pour la gestion de l'accéléromètre) et 260 lignes d'Eclat (pour la gestion du protocole VGA, des entrées/sorties et de la logique du jeu).

La figure 7.5 est un instantané de cette implantation du jeu de Pong. La partie est en pause ; la balle est représentée par un carré rouge ; les raquettes sont des rectangles blancs à gauche et à droite de l'écran.

Figure 7.5 Jeu de pong mis en pause

Le protocole VGA requiert une fréquence de 25 MHz, tandis que celle du FPGA est de 50

6. Ce mécanisme a aussi été développé dans le cadre de ce stage.

MHz. Par conséquent, le programme ECLAT doit envoyer des données tous les deux cycles aux sorties transitant par le port VGA en respectant le protocole. Celui-ci consiste à envoyer la couleur de chaque pixel un à un avec un temps de synchronisation fixe (en nombre de cycles) à la fin de chaque ligne et entre chaque image.

Les raquettes et la balle se meuvent suivant une trajectoire (`dx`,`dy`). Les nouvelles positions des raquettes et de la balle sont calculées 50 fois par seconde (tous les un million de cycles).

Dans une première version, le calcul de la nouvelle position était instantanée : c'était une fonction combinatoire qui comportait des dizaines d'additions, soustractions et comparaisons pour gérer les collisions entre la balle, les raquettes et les bords de l'écran. Or, la fréquence maximale de fonctionnement du circuit synthétisé (calculée par le synthétiseur logique) était de 20 MHz, ce qui était bien inférieur à la fréquence d'horloge de 50 MHz du FPGA MAX 10.

Nous avons donc utilisé en ECLAT une technique de conception de matériel (appelée *retiming*) qui consiste à insérer des registres dans les grosses fonctions combinatoires : cela a pour effet de raccourcir la longueur *chemin critique*[7] et donc augmenter la fréquence.

En style flot de données, insérer des registres et assez fastidieux. C'est pourquoi nous avons eu recours à la construction **`exec`** pour ainsi mettre à jour les positions via un *calcul* prenant plusieurs cycles d'horloge. Ce calcul, dont un extrait est listé sur la figure 7.6, correspond à l'expression combinatoire de départ mais dans laquelle nous avons inséré des *pauses*. La construction **`pause`**`()` est un construction dérivée d'ECLAT (définie figure 3.5 par **`let rec`** f`()` `=` `()` **`in`** f`()` avec f un nom frais). Chaque **`pause`** introduit un retard d'un cycle. Nous laissons au compilateur (et au synthétiseur logique) le soin d'insérer les registres à chaque *pause* pour réaliser le comportement temporel spécifié pour ces calculs. Dans cette deuxième version du jeu, grâce à cette utilisation d'**`exec`** pour décrire le circuit, la fréquence maximale est passée au delà de 50 MHz et le circuit synthétisé fonctionne donc correctement.

7.5 Un algorithme généraliste : les n reines

Dans cette section, nous programmons en ECLAT le problème des n reines bien connu [198] : placer n reines sur un échiquier $n \times n$ sans qu'il y ait de reine en prise par une autre reine selon les règles des échecs. Les reines peuvent être en prise verticalement, horizontalement ou en diagonale (de deux manières).

Le nombre d'arrangements à examiner est très élevé : $n!$ [198]. Le nombre exhaustif de solutions pour $n = 27$ a été découvert en 2016 [166] (à savoir environ $2,3 \cdot 10^{17}$ solutions[8], l'espace de recherche étant bien plus grand).

7. C'est-à-dire, la "distance" maximale - en termes temporels - pour aller d'un point A à un point B dans le circuit. C'est cette distance qui détermine la fréquence maximale d'horloge, compte tenu des temps de propagation du courant électrique.

8. Ce calcul s'est exécuté pendant un an sur 14 FPGAs connectés programmés en VHDL. C'est à ce jour le plus grand n pour lequel le nombre de solutions a été trouvé. Le précédent record pour $n = 26$ avait été découvert en 2009 par les mêmes auteurs en utilisant une infrastructure similaire basée sur 10 FPGAs. Les FPGAs utilisés pour ces implantations étaient puissants, comme par exemple le FPGA Stratix IV GX d'Intel.

```
1  let (pos_dx,pos_dy) = ball_pos_delta in
2  pause();
3  let collision_top_wall = ball_y - pos_dy < 0 in
4  pause();
5  let collision_bottom_wall = ball_y + ball_h + pos_dy >= 480 in
6  pause();
7  let collision_left_wall = ball_x - pos_dx < 0 in
8  pause();
9  let collision_right_wall = ball_x + ball_w + pos_dx >= 640 in
10 pause();
11 ···
```

FIGURE 7.6 Insertion de pauses dans la fonction de déplacement des objets du jeu de Pong

Une méthode de résolution de ce problème par récurrence [213] a déjà été mise en œuvre dans le langage CλASH (et implantée sur un modèle de FPGA proche de celui que nous utilisons dans ce chapitre) pour montrer la méthodologie de conception de circuit en CλASH, dans l'écosystème HASKELL [121].

L'implantation de cet exemple en ECLAT, montre que le langage est suffisamment expressif pour programmer ce type de solution. En particulier, cela montre qu'il est possible d'implanter des piles explicites avec des structures de données, et ainsi programmer des algorithmes récursifs sur FPGA avec ECLAT malgré l'absence de support natif pour la récursion générale.

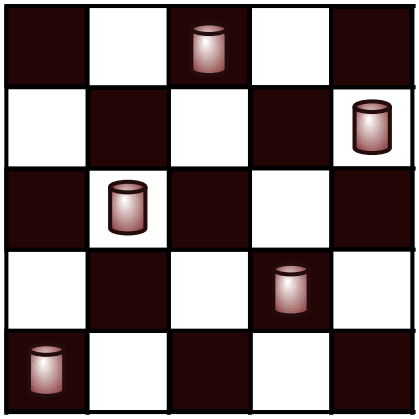
Dans toute solution correcte au problème des n-reines, chaque colonne de l'échiquier est occupée par exactement une reine. Elle peut donc être représentée par une séquence de n entiers $(p_1, \cdots p_n)$ où p_{col} indique le numéro de ligne de la reine sur la colonne $\mathsf{col} \in \{1, \cdots n\}$. La figure 7.7 donne une exemple de solution pour un échiquier de taille $n = 5$.

Deux reines aux lignes p et q sont dites *sûres* si elle sont placées (verticalement) l'une par rapport à l'autre de sorte que les contraintes du problème des n reines soient satisfaites, c'est-à-dire **(1)** les reines p et q ne sont pas sur la même ligne et **(2)** les reines p et q ne sont pas sur la même diagonale.

Le critère **(2)** se vérifie en comparant la distance horizontale d et la distance verticale $|p-q|$ entre les reines p et q. Cela se programme en ECLAT par une fonction combinatoire :

```
1 let abs_sub (a, b) =
2   if b > a then b-a else a-b ;;
3
4 let safe (p, q, d) =
5   not(p = q) & not(abs_sub(q, p) = d);;
```

Une reine p est sûre vis-à-vis d'un ensemble de k reines parmi l'ensemble $\mathtt{qs} \stackrel{\text{def}}{=} \{q_1, \cdots q_m\}$



$(1 \quad 3 \quad 5 \quad 2 \quad 4)$

FIGURE 7.7 Représentation d'une solution au problème des n reines pour $n = 5$

avec $k \leq m$ (ce qui est notée `safe_all (`k`,`p`,qs)`) si et seulement si, pour tout i dans l'intervalle $[1..k]$, `safe(`q_i`, `p`, `$k+1-i$`)` est vraie. La pile `qs` est représentée par un vecteur d'entiers ECLAT, c'est-à-dire, une valeur immédiate (de type `int<8> vect<`n`>`) concaténant les représentations des n entiers.

```
 7  let rec safe_all_aux (qs, i, k, p) =
 8     if i > k then true
 9     else if safe(vect_nth(qs,i-1), p, k+1-i) then
10       safe_all_aux(qs, i+1, k, p)
11     else false ;;
12
13 let safe_all (k, p, qs) =
14   safe_all_aux (qs, 1, k, p) ;;
```

Sur la figure 7.8, la recherche exhaustive des solutions au problème des n reines est définie par une fonction doublement récursive terminale `loop(first_i, col, i, n, qs, nb)` avec une pile `qs` explicite.

L'argument `col` donne le nombre de colonnes déjà traitées ; l'argument `i` $\geq$ `first_i` est l'indice candidat de la ligne `col` $+ 1$ où placer une reine qui soit sûre par rapport à l'ensemble de reines déjà placées ; `n` est le nombre de reines (ce nombre ne varie pas au cours du calcul) ; `qs` représente l'échiquier, les `col` premières valeurs dans `qs` ont déjà été fixées et les colonnes suivantes restant à déterminer ; `nb` est le nombre de solutions déjà trouvées. La récursion s'arrête lorsque `i` $>$ `n` et `col` $<$ `1` (ligne 18). Pour tout `i` $\geq$ `n`, si la configuration `qs` avec la reine `i` est sûre jusqu'à la colonne `col` (ligne 18), alors la recherche se poursuit avec la colonne `col-1` en traitant les lignes candidates à partir de `first_i`.

La figure 7.9 donne le temps d'exécution mesuré du programme ECLAT `queens(`n`)` synthétisé sur le FPGA Max10. Comme attendu, le temps d'exécution de `queens(`n`)` rend cette

```
16 let rec loop (first_i, col, i, n, qs, nb) =
17  if i > n then
18    ( if col < first_i then nb
19      else loop(first_i, col-1, vect_nth(qs,col-1)+1, n, qs, nb)
20   )
21   else if safe_all (col,i,qs) then
22        let qs = vect_copy_with(qs,col,i) in
23        let nb = if col+1 = n then nb+1 else nb in
24        loop(first_i, col+1, first_i, n, qs, nb)
25    else loop(first_i, col, i+1, n ,qs, nb) ;;
26
27 let queens n =
28   (* la taille n est une variable résolue par le compilateur *)
29   let qs = vect_create<n>(1) in
30   loop (1, 1, 1, n, qs, 0) ;;
```

FIGURE 7.8 Recherche exhaustive du nombre de solutions au problème des n reines

fonction inutilisable au delà de $n = 16$ (où il se compterait en jours puis en années). Pour $n = 16$, le calcul prend 45 minutes.

Il est alors naturel de chercher à paralléliser le programme ECLAT pour réussir à obtenir un résultat pour $n \geq 17$. Dans la section suivante, nous proposons deux optimisations qui, en se cumulant, nous permettent d'obtenir une solution au problème des n reines pour $n = 18$ en seulement 24 minutes (sur le FPGA MAX 10).

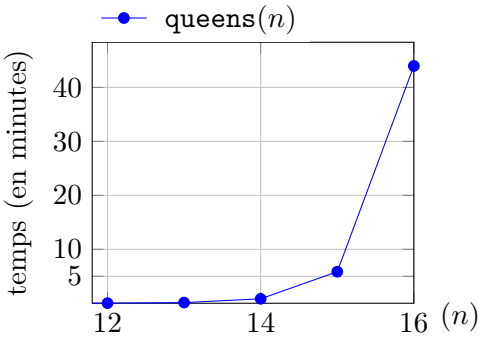


FIGURE 7.9 Temps d'exécution de `queens(n)` en fonction de n

7.6 Compromis temps-espace

En ECLAT, chaque appel de fonction récursive induit un retard de un cycle. En particulier, la fonction `safe_all_aux` s'exécute en *au moins* un cycle et *au plus* `n+1` cycles. En déroulant les n étapes de récursion, nous obtenons un circuit plus gros, mais où `safe_all` termine instantanément. C'est un exemple classique de compromis temps-espace sur FPGA.

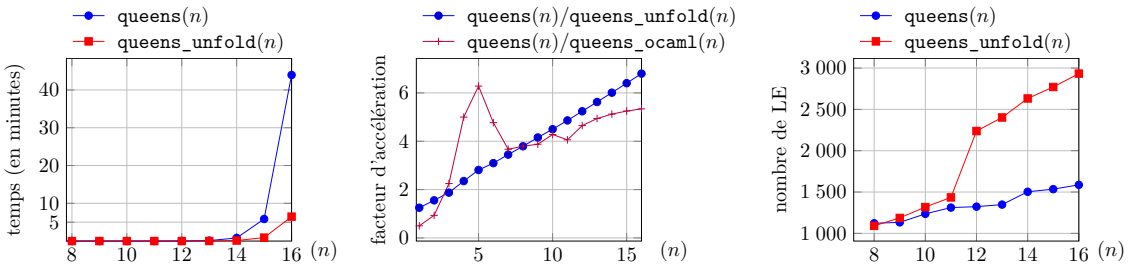
En utilisant la construction `generate` définie à la section 7.2, nous pouvons dérouler les étapes de récursion de la fonction `safe_all` comme suit :

```
let safe_all (k, p, qs) =
  let f (i, acc) =
    acc & ( (i+1 > k) or safe(vect_nth(qs, i), p, k - i) )
  in
  generate f true (vect_length(qs)) ;;
```

La figure 7.10 présente les performances d'une nouvelle version de `queens`(n) (appelée `queens_unfold`(n)) avec déroulement de `safe_all`, vis-à-vis de la fonction `queens`(n) originale (définie sur la figure 7.8).

La figure 7.10a donne les temps d'exécution mesurés de `queens_unfold`(n) et `queens`(n) pour différents n (jusqu'à 16).

La figure 7.10b donne le facteur d'accélération de `queens_unfold`(n) par rapport à `queens`(n) ; celui-ci croît linéairement avec n. Nous reportons aussi (figure 7.10b) le ratio entre la version `queens`(n) originale traduite en OCAML sur CPU (et qui est appelée `queens_ocaml`(n)). Cette version `queens_ocaml` est compilée par le compilateur `ocaml-opt` avec optimisation activée (`-O2`) comme expliqué à la section 7.1. La version OCAML sur CPU à des temps d'exécution comparables à la version ECLAT déroulée sur FPGA. On note cependant un infléchissement pour n compris entre 2 et 7 ; ce phénomène sera étudié sur la figure 7.13d.



(a) temps d'exécution (b) gain de vitesse (c) utilisation des ressources

FIGURE 7.10 Accélération du problème des n reines par déroulement de la fonction `safe_all`

La figure 7.10c donne les tailles de circuits engendrés à partir des fonctions `queens`(n) et `queens_unfold`(n) respectivement : comme attendu, le nombre d'éléments logiques (LE) est plus

important dans la version déroulée et le ratio du nombre de LE `queens_unfold`(n)/`queens`(n) croit avec n (il est de 1,1 pour $n = 11$; 1,7 pour $n = 12$; 1,75 pour $n = 14$; 1,85 pour $n = 16$).

Nous pouvons aussi décomposer le problème des n reines en n sous-problèmes (figure 7.11) : chaque sous-problème calcule le nombre de solutions telles que la ligne i à la colonne 1 est fixée (entre 1 à n). Pour accumuler les résultats des n branches parallèles (ligne 8), nous utilisons une variable partagée (codée par un tableau `r` de une case) initialisée à `0` (ligne 4). Un point important est que le point de reprise (c'est-à-dire, l'argument `first_i` dans la définition de `loop`) qui est ici la colonne numéro **2** (ligne 7) et non plus 1 comme à la figure 7.8 (ligne 30).

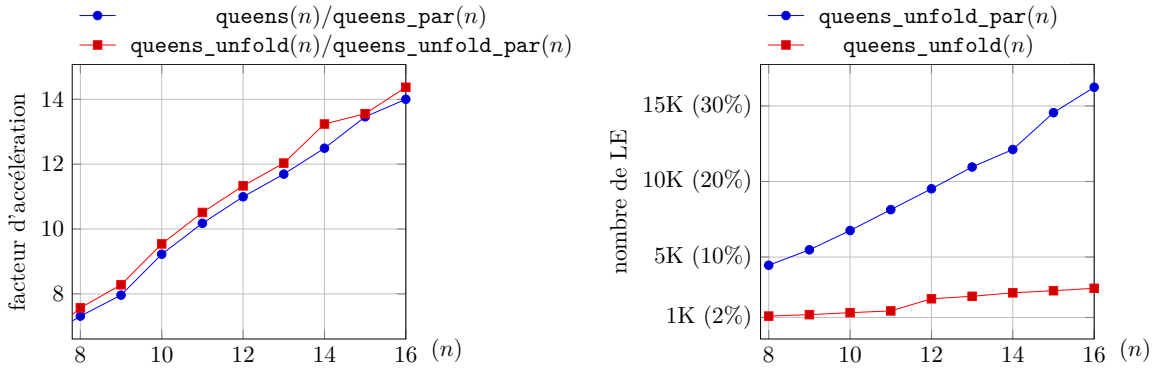
```
1  let queens n =
2    let qs = vect_create<n>(1) in
3    let r = create<1> in
4    set(r,0,0);
5    par_for i = 1 to n do
6      let qs_i = vect_copy_with(v,0,i) in
7      let nb_i = loop (2, 1, 1,n,qs_i,0) in
8      set(r,0,nb_i + get(r,0))
9    done;
10   (get(r,0));;
```

FIGURE 7.11 Décomposition du problème des n reines en n sous problèmes parallélisés

La figure 7.12 donne les gains en vitesse et l'augmentation du nombre de LE dû à la parallélisation de `queens_unfold` (appelée `queens_unfold_par`). Cela illustre très clairement le compromis temps-espace en programmation de FPGA : exécuter n circuits en parallèle donne un gain de vitesse allant jusqu'à n (figure 7.12a) mais occupe jusqu'à n fois plus de ressources (figure 7.12b). Le nombre d'éléments logiques (LE) utilisés (figure 7.12b) est donné avec le pourcentage correspondant parmi les 50 000 LE disponibles sur le FPGA cible (comme expliqué à la section 7.1).

La figure 7.13 sépare les différents gains de vitesse entre la version ECLAT non optimisée (`queens`), les deux optimisations séparées (`queens_unfold` et `queens_par`) et le cumul de ces deux versions (`queens_unfold_par`) vis-à-vis d'une version séquentielle sur CPU (`queens_ocaml`).

La figure 7.13c montre que la version ECLAT non optimisée est quatre fois plus lente que la version OCAML native (sur CPU) ; il est donc nécessaire d'exploiter le parallélisme sur FPGA pour être compétitif avec un CPU. Les deux optimisations proposées (figure 7.13b) sont de 1 à 3 fois plus rapides que la version OCAML native pour les entiers n considérés et ces facteurs d'accélérations augmentent avec n. L'optimisation `queens_par` est plus performante que `queens_unfold` ; cela s'explique par le fait que la fonction récursive `safe` (dans `queens`) vérifie séquentiellement des configurations partielles (de longueurs inférieures à n) et



(a) gain de vitesse

(b) utilisation des ressources

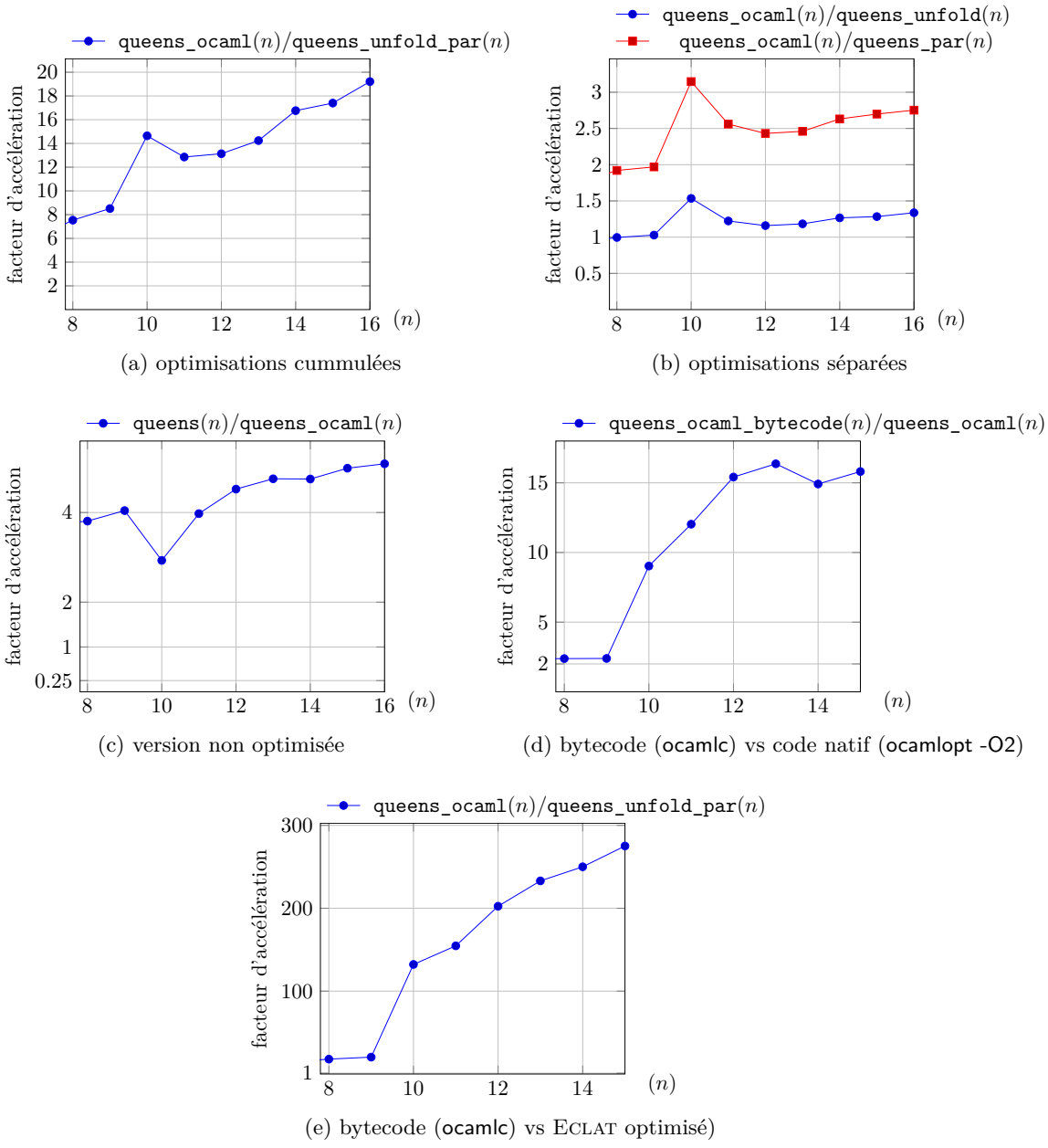
FIGURE 7.12 Compromis temps-espace dans la parallélisation des n reines

s'arrête dès qu'une reine rend la configuration non sûre. Dérouler la fonction `safe` (ce que fait `queens_unfold`) fournit donc à gain de vitesse inférieur à n (qui a été donné sur la figure 7.10b). L'optimisation `queens_par` a une limitation similaire car les n positions de départ fixées conduisaient à des temps de recherche déséquilibrés ; c'est pourquoi le facteur d'accélération qui a été donné sur la figure 7.12a est non optimal.

Finalement, le cumul des deux optimisations (`queens_unfold_par`), qui est montré sur la figure 7.13a, conduit à une réelle accélération sur FPGA, supérieure à un facteur n pour n reines, vis-à-vis de la version native sur CPU (`queens_ocaml`).

Aux figures 7.13d et 7.13e, nous donnons aussi une comparaison vis-à-vis du même programme OCAML compilé en bytecode (queens_ocaml_bytecode) et exécuté par l'implantation standard ocamlrun de la machine virtuelle OCAML. Pour $n = 15$, la version ECLAT optimisée est presque 300 fois plus rapide que la version séquentielle compilée en bytecode, et ce facteur d'accélération augmente avec n. Le gain de vitesse n'est pas donné pour $n = 16$ en raison du temps d'exécution trop long de la version bytecode (18 minutes pour $n = 15$).

Cet exemple montre la différence fondamentale entre programmation de logiciel et de matériel. Un circuit FPGA est beaucoup plus lent qu'un processeur classique ; on ne peut donc pas prétendre à être compétitif avec un CPU (dont la fréquence de fonctionnement est de l'ordre de 50 fois plus rapide) lorsque l'on exécute du code séquentiel sur un FPGA. Cependant, avec un modèle de programmation adapté, il est possible de programmer précisément le circuit pour obtenir des gains significatifs, notamment en jouant sur la duplication des calculs au niveau matériel et donc le parallélisme. La section suivante poursuit l'investigation des possibilités de programmation parallèles disponibles en ECLAT en considérant, de plus, la mémoire partagée.



(a) optimisations cummulées

(b) optimisations séparées

(c) version non optimisée

(d) bytecode (ocamlc) vs code natif (ocamlopt -O2)

(e) bytecode (ocamlc) vs ECLAT optimisé)

FIGURE 7.13 Facteur d'accélération des n reines en ECLAT par rapport à OCAML sur CPU

7.7 Mémoire partagée : le Jeu de la vie

Cette section s'intéresse à l'utilisation de la mémoire partagée dans les programmes ECLAT à travers la programmation d'une application simple.

Le *Jeu de la vie* (*Game of Life*) [62] est un automate cellulaire qui décrit l'évolution d'une grille à deux dimensions théoriquement infinie. En pratique, la grille est souvent réalisée sous la forme d'un tore : la grille a *nbc* colonnes et *nbl* lignes ; on identifie la colonne 0 (*resp.* la ligne 0) à la colonne *nbc* (*resp.* la ligne *nbl*).

Les éléments de la grille sont appelés *cellules* par analogie avec un processus biologique. Chaque cellule possède un statut *vivant* ou *mort* et se situe à une certaine position (i, j) entourée par huit cellules voisines :

$$\begin{array}{lll} (i-1, j-1) & (i-1, j) & (i-1, j+1) \\ (i, j-1) & & (i, j+1) \\ (i+1, j-1) & (i+1, j) & (i+1, j+1) \end{array}$$

Les règles du Jeu de la vie, qui définissent une nouvelle grille à partir d'une grille précédente, peuvent s'énoncer comme suit :

> Une cellule c dans la grille précédente est vivante dans la nouvelle grille si dans la grille précédente, (1) elle est vivante et elle possède soit deux, soit trois cellules voisines vivantes, ou (2) si elle est morte et possède trois cellules voisines vivantes.

Dans la suite de cette section, nous présentons trois programmes ECLAT implantant le Jeu de la vie sur FPGA. La première version représente la grille par un tableau ECLAT (c'est-à-dire, une mémoire partagée). Nous montrons l'impact de ce choix sur les performances : les accès mémoire à un même tableau sont (implicitement) séquentialisés par un verrou et le temps d'accès domine le temps de calcul ; cela limite fortement le parallélisme exploitable sur le FPGA.

La seconde version représente la grille par un vecteur ECLAT, c'est-à-dire une constante de grande taille (par exemple, 4096 bits pour représenter une grille de 64×64 cellules). Les performances temporelles sont alors accrues car le calcul de la nouvelle grille est instantané. Cela illustre à nouveau le compromis temps-espace : un calcul est répliqué sur chaque cellule de la grille en sorte que toutes soient mises à jour en parallèle. Cette méthode, cependant, ne passe pas à l'échelle : au delà d'une taille de grille relativement petite (de l'ordre de 72×72), le programme n'est plus synthétisable sur le modèle de FPGA utilisé (Max 10 d'INTEL) car il occupe trop de ressources (en nombre de cellules logiques).

La troisième version propose un compromis entre les versions 1 et 2, en représentant la grille par un tableau de vecteurs ECLAT, ce qui permet d'exploiter le parallélisme à grain fin sur les lignes, tout en stockant un grand nombre de lignes dans la mémoire partagée.

7.7.1 Version 1 : représenter la grille par un tableau

Dans un langage fonctionnel-impératif comme ECLAT, la grille du Jeu de la vie peut être représentée par un tableau de booléens (`true` pour *vivant* et `false` pour *mort*). La cellule située à la colonne i de la ligne j désige l'élément à l'indice $i + j \times nbl$ du tableau, où nbl est le nombre de lignes que comporte la grille. Une manière de programmer le Jeu de la vie consiste alors à utiliser deux tableaux de booléens : un tableau *source* pour la grille précédente et un tableau *destination* pour la nouvelle grille construite suivant les règles du Jeu de la vie à partir de la grille précédente. Une fois la nouvelle grille calculée, elle remplace la grille précédente par copie du contenu du tableau destination vers le tableau source.

Suivant cette méthode, nous définissons sur la figure 7.14 une fonction `array_life` (lignes 28 à 43) qui transforme *en place* une grille `src` de taille $N = \texttt{nbc} \times \texttt{nbl}$. Cette fonction utilise un tableau temporaire `dst` défini localement à la ligne 29. Ce tableau représente la nouvelle grille construite suivant les règles du Jeu de la vie (lignes 31 à 42) et recopiée dans le tableau `src` à la ligne 43 en utilisant la fonction `copy` définie aux lignes 22 à 25. La fonction `copy(src,dst)` effectue un parcours des N éléments du tableau via une fonction récursive terminale `aux`. Celle-ci comporte N lectures de `src`, N écritures de `dst` et $N + 1$ appels à `aux`. Chaque lecture, chaque écriture et chaque appel à `aux` prend un cycle, d'où un temps d'exécution de exactement $3N + 1$ cycles d'horloge.

La fonction `sum_neighborhood` (ligne 16 à 19) calcule le nombre de cellules vivantes voisines de la cellule à la position `(i, j)` sur la grille `g` de taille $\texttt{nbc} \times \texttt{nbl}$. Il en résulte huit lectures par cellule de la grille à traiter. La grille `src` est parcourue par indices `i` et `j` suivant une double récursion aux ligne 30 à 42. la valeur `cell` de chaque cellule est lue dans la grille précédente `src` en cours de traitement (ligne 34), et la nouvelle valeur est écriture dans la grille destination `dst` (ligne 39).

Sur la figure 7.14, les accès aux tableaux `src` et `dst` sont effectués séquentiellement. Le temps d'exécution de la fonction principale `array_life` (lignes 28 à 44) est de $14N + nbl + 2$ cycles où N est le nombre de cellules dans la grille et nbl le nombre de lignes. En effet, aux lignes 31 à 43, il y a $N + 1$ appels à la fonction `loop`, $9N$ lectures de `src` (lignes 35 et 37) et N écritures de `dst`. À cela s'ajoute les $3N + 1$ cycles de la fonction `copy` appelée ligne 44.

Etant donnée la structure de données de tableau utilisée pour représenter la grille, ce temps d'exécution est presque optimale, car les accès à un même tableau sont nécessairement exécutés séquentiellement (de par le verrou associé à chaque tableau et les contraintes d'accès aux blocs RAM).

Cela conduit à une perte de vitesse d'un facteur $3,8$ par rapport à une exécution du même programme séquentiel sur CPU, mettant ainsi en évidence un goulot d'étranglement dans l'accès aux éléments d'un même tableau (en mémoire partagée) qui limite les possibilités de parallélisation et les performances, compte tenu de la différence de fréquence entre le CPU à 2.2 GHz et le FPGA à 50MHz, qui est d'un facteur 44.

```
let read_cell(src, i, j, nbl) = get(src, i + j * nbl) ;;

let set_cell(dst, i, j, nbl, v) = set(dst, i + j * nbl, v) ;;

(* gestion des indices sur les bords de la grille (structure torique) *)
let border(k,nb) =
  if k = (-1) then nb else
  if k = nb then 0 else k ;;

(* retourne 1 si la cellule (i,j) est vivante : un accès mémoire *)
let alive(g,i,j,nbc,nbl) =
  let cell : bool = read_cell(g, border(i,nbc), border(j,nbl), nbl) in
  if cell then 1 else 0 ;;

(* compte les voisins vivants : 8 accès mémoire *)
let sum_neighborhood(g,i,j,nbc,nbl) =
  alive(g,i-1,j-1,nbc,nbl) + alive(g,i,j-1,nbc,nbl) + alive(g,i+1,j-1,nbc,nbl) +
  alive(g,i-1,j,nbc,nbl) +                            alive(g,i+1,j,nbc,nbl) +
  alive(g,i-1,j+1,nbc,nbl) + alive(g,i,j+1,nbc,nbl) + alive(g,i+1,j+1,nbc,nbl) ;;

(* copier le contenu d'un tableau dans un autre : 2 * length(src) accès mémoire *)
let copy (src,dst) =
 let rec aux(i) = (* length(src) + appels à aux, un cycle par appel *)
   if i < length(src) then (set(dst,i,get(src,i)); aux(i+1)) else ()
 in aux(0) ;;

(* construction de la grille suivante : modification de la grille src en place *)
let array_life ((src, nbc, nbl) : (bool array< 'N > * int<16> * int<16>)) =
  let dst = create< 'N >() in (* tableau temporaire *)
  (* pour chaque cellule de la grille *)
  let rec loop(i,j) =
    if i = nbc then () else
    if j = nbl then loop(i+1,0)  (* l'appel à loop prend un cycle, un par ligne *)
    else ((* lire sa valeur : une lecture en mémoire (un cycle) *)
      let cell = read_cell(src, i, j, nbl) in
      (* compte les voisins vivants : 8 lectures en mémoire (8 cycles) *)
      let sum : int<4> = sum_neighborhood(src,i, j,nbc,nbl) in
      let new_cell = (cell & sum = 2) or (sum = 3) in
      (* écrire la nouvelle valeur : une écriture en mémoire (1 cycle) *)
      set_cell(dst,i,j,nbl,new_cell);
      loop(i,j+1) (* l'appel à loop prend un cycle *)
  ) in
  loop(0,0); (* l'appel à loop prend un cycle *)
  copy(dst,src) ;; (* copie en 2*nbc*nbl cycles *)
```

FIGURE 7.14 Jeu de la vie, version 1

7.7.2 Version 2 : représenter la grille par un vecteur

Bakhteri *et al.* [15] ont présenté une implantation VERILOG du Jeu de la vie capable de traiter l'ensemble des cellules du monde en un cycle. Les auteurs utilisent un petit FPGA similaire au Max 10 que nous utilisons ; cela les conduit à un gain de vitesse d'un facteur 36,7 par rapport à une implantation GPU et un gain 2908 par rapport à une implantation optimisée sur CPU.

ECLAT est suffisamment précis pour programmer cette solution, qui est donnée sur la figure 7.15. La grille est représentée par un vecteur de vecteurs de booléens.

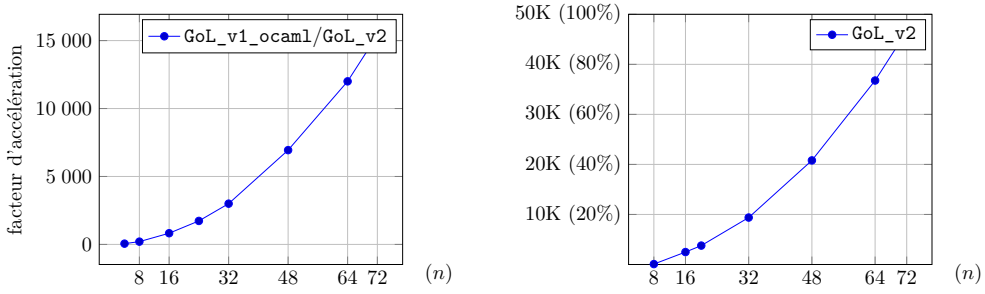
```
1  (* lecture d'une cellule *)
2  let read_cell(src, i, j) =
3    let line = vect_nth(src, i) in
4    let cell = vect_nth(line,j) in
5    cell ;;
6
7  (* gestion des indices sur les bords de la grille (structure torique) *)
8  let border(k,n) =
9    if k = (-1) then n - 1 else
10   if k = n then 0 else k ;;
11
12 (* retourne 1 si la cellule (i,j) est vivante *)
13 let alive(g,i,j,nbc,nbl) =
14   let cell : bool = read_cell(g, border(i,nbc), border(j,nbl)) in
15   if cell then 1 else 0 ;;
16
17 (* compte les voisins vivants *)
18 let sum_neighborhood(g,i,j,nbc,nbl) =
19   alive(g,i-1,j-1,nbc,nbl) + alive(g,i,j-1,nbc,nbl) + alive(g,i+1,j-1,nbc,nbl) +
20   alive(g,i-1,j,nbc,nbl)   +                          alive(g,i+1,j,nbc,nbl) +
21   alive(g,i-1,j+1,nbc,nbl) + alive(g,i,j+1,nbc,nbl) + alive(g,i+1,j+1,nbc,nbl) ;;
22
23 (* construction de la grille suivante : construit une nouvelle grille *)
24 let vect_life ((grid,nbc,nbl) : ((bool vect<'NBC>) vect<'NBL> * int<8> * int<8>)) =
25   let next_line(i,line) =
26     let next_cell(j,cell) =
27       let sum : int<4> = sum_neighborhood(grid,i,j,nbc,nbl) in
28       let new_cell = (cell & s = 2) or (s = 3) in
29       new_cell
30     in
31     vect_mapi(next_cell,line) in
32   vect_mapi(next_line,grid) ;;
```

FIGURE 7.15 Jeu de la vie, version 2

Nous utilisons la construction `vect_mapi` définie à la section 7.2. Elle s'applique sur les lignes (à la ligne 32) et sur les cellules de chaque ligne (ligne 31) pour transformer la grille précédente de façon instantanée (en *zéro* cycle). Cela permet de construire une nouvelle grille à chaque cycle suivant la règle du Jeu de la vie.

La figure 7.16 compare les performances de cette version sur FPGA (`GoL_v2`, appelée à chaque cycle) pour une grille de taille n^2 vis-à-vis de celle obtenue avec la version 1 sur CPU vue comme un programme OCaml séquentiel (`GoL_v1_ocaml`).

La figure 7.16a donne le facteur d'accélération tandis que la figure 7.16b donne le nombre d'éléments logiques (LE) et le pourcentage correspondant d'utilisation des LE (jusqu'à 50K LE disponibles sur le FPGA Max10). Comme à la section 7.6, les courbes illustrent parfaitement le type de compromis temps-espace que l'on peut obtenir en utilisant Eclat. L'accélération maximale observée est significative : $\times 15000$ pour une grille de $72 \times 72 \approx 5000$ cellules. Au-delà de cette limite, le programme ne tient pas sur le circuit FPGA Max10 utilisé [9] ; c'est aussi le cas dans l'implantation Verilog par Bakhteri *et al.* [15].



(a) gain de vitesse (b) nombre d'éléments logiques

Figure 7.16 Compromis temps espace pour le Jeu de la vie sans mémoire partagée

7.7.3 Version 3 : représenter la grille par un tableau de vecteurs

Pour surmonter la limitation de taille de la version parallèle à grain fin décrite ci-dessus, nous représentons la grille, non pas comme un vecteur unique, mais comme un *tableau* de vecteurs : chaque vecteur encode une ligne. Avec cette formulation, les lignes sont des éléments (très larges) stockés en mémoire partagée, et sur lesquels le parallélisme à grain fin reste exploitable individuellement.

Le programme est défini sur la figure 7.17. Tout d'abord, nous définissons une fonction `alive(l,j)` qui retourne 1 si la `j`-ème cellule de la ligne `l` est vivante et sinon `0` (lignes 7 à 9).

La fonction `sum_neighborhood` calcule le nombre de cellules vivantes voisines de la cellule `j` de la ligne `l1` étant données la ligne précédente (`l0`) et la ligne suivante (`l2`).

9. Les FPGA haut de gamme ont beaucoup plus de LE ; par exemple, le Stratix 10 d'Intel comporte 2.8 millions de LE (c'est-à-dire 56 fois plus que dans le Max10).

La fonction `next_cell_with_lines(l0,l1,l2,j)` détermine si la cellule à l'emplacement `j` de la ligne `l1` devient vivante ou morte (lignes 18 à 20) en fonction de sa valeur (`cell`) et du nombre de ses voisines vivantes (`sum`).

```
1  (* gestion des indices sur les bords d'une ligne (structure torique) *)
2  let border(j,n) =
3    if j = (-1) then n - 1 else
4    if j = n then 0 else j ;;
5
6  (* retourne 1 si la cellule (i,j) est vivante *)
7  let alive(l,j) =
8    let cell = vect_nth(l,border(j,vect_size l)) in
9    if cell then 1 else 0 ;;
10
11 (* compte les voisins vivants *)
12 let sum_neighborhood(l0,l1,l2,j) =
13   alive(l0,j-1) + alive(l1,j-1) + alive(l2,j-1) +
14   alive(l0,j)                   + alive(l2,j) +
15   alive(l0,j+1) + alive(l1,j+1) + alive(l2,j+1) ;;
16
17 (* détermine la nouvelle valeur de la cellule j de la ligne l1 *)
18 let next_cell_with_lines(j,cell,l0,l1,l2) =
19   let sum : int<4> = sum_neighborhood(l0,l1,l2,j) in
20   (cell & sum = 2) or (sum = 3) ;;
21
22 (* construction de la grille suivante : modifie le tableau en place *)
23 let vect_array_life (grid : bool vect<'NBC> array<'NBL>) : unit =
24   let first_line = get(grid,0) in
25   let rec aux (l0,l1,i) : unit =
26     if i < length(grid) then
27     (let l2 = if i = length(grid)-1 then first_line else get(grid,i+1) in
28      let next (j,cell) = next_cell_with_lines(j,cell,l0,l1,l2) in
29      set(grid,i,(vect_mapi(next,line1)));
30      aux(l1,l2,i+1)) else ()
31   in
32   aux(get(grid,length(grid)-1),first_line,0) ;;
```

FIGURE 7.17 Jeu de la vie, version 3

La fonction principale `vect_array_life` (lignes 23 à 32) parcourt l'ensemble des lignes de la grille en utilisant une fonction auxiliaire `aux` récursive terminale (lignes 25 à 32). La fonction `aux` lit la ligne suivante `l2` et met à jour la ligne courante `l1` en fonction de la ligne précédente `l0`, et ainsi de suite. La ligne qui suit la dernière ligne est une copie de la première ligne (`first_line` sauvegardée ligne 24) avant que celle-ci ne soit modifiée par la fonction `aux`.

Pour une grille de *nbl* lignes, cette troisième version du Jeu de la vie lit $nbl + 1$ lignes, indépendamment du nombre *nbc* de cellules par ligne. La transformation est effectuée en place, il y a *nbl* écritures de tableaux. Il y a une pause par appel à la fonction `aux` (1 appel direct + *nbl* appels récursifs). Par conséquent, le temps d'exécution total est exactement de $(1+1+1)nbl + 1 = 3nbl+1$ cycles pour traiter une grille complète de taille $N = nbl \times nbc$. Le temps d'exécution augmente linéairement avec *nbl* tandis que le nombre d'éléments logiques (LE) croît avec *nbc* exactement comme sur la figure 7.16b.

Ces mesures sont rapportées sur la figure 7.18, qui résume les performances (ici le débit en nombre de cellules traitées par cycle ainsi que l'utilisation des ressources en nombre d'éléments logiques et en nombre de blocs RAM) des versions du Jeu de la vie décrites dans cette section.

implantation	taille de la grille	LE	RAM	débit
v1 (`array_life`)	1024×512	2K	1Mbit	1 cellule tous les 14 cycles
Bakhteri *et al.* [15] (VERILOG)	64×64	68K	0	4096 cellules par cycle
v2 (`vect_life`)	72×72	47K	0	5184 cellules par cycle
v3 (`vect_array_life`)	2048×512	49K	1Mbit	2048 cellules tous les 3 cycles

FIGURE 7.18 Compromis temps-espace pour différentes implantations du Jeu de la vie

La version 3 a un débit 680 fois plus lent que la version 2 (en nombre de cellules par cycle), mais elle peut traiter des grilles 200 fois plus larges. Elle n'utilise pas de tableau intermédiaire, contrairement à la version 1, ce qui lui permet de traiter une grille deux fois plus large avec un débit $(2048/3) \times 14 = 10000$ fois plus rapide.

Ces trois versions du Jeu de la vie en ECLAT montrent l'importance du choix de structures de données (entre vecteurs et tableaux). La version 1 sur FPGA est 3,8 fois plus lente que sur CPU (vu comme un programme OCAML compilé en code natif). La version 2 sur FPGA est 15000 fois plus rapide que la version 1 sur CPU, mais elle n'est applicable que sur de petites grilles (jusqu'à $72 \times 72 \approx 5000$ cellules avec le modèle de FPGA utilisé). La version 3 est $10000/3,8 \approx 2630$ fois plus rapide que la version 1 sur CPU et elle peut traiter des grilles d'un million de cellules.

7.8 Développement de squelettes de parallélisme en Eclat

Cette section présente des implantations parallèles des squelettes algorithmiques `map`, `reduce` et `pipe` en utilisant ECLAT. Elle met l'accent sur la composabilité et donne un exemple de prédiction des performances, corroborée expérimentalement.

7.8.1 Implantation de `map`

Le squelette `map` est classiquement utilisé pour appliquer une fonction `f` donnée à chaque élément d'un tableau. Une implantation parallèle de `map`, appelée `par_map`, est donnée sur la figure 7.19 (lignes 13-19). Elle prend en paramètres un degré de parallélisme `p`, une fonction `f`,

un tableau source `src` et un tableau destination `dst`. Elle applique `f` aux éléments de `src`, en parcourant `p` parties du tableau source en parallèle (un appel à `map_slice` par sous-tableau) à l'aide de la macro-construction **`parfor`** présentée dans la section 7.2. Si la longueur de `src` n'est pas un multiple de `p`, les éléments restants sont traités séquentiellement (ligne 19).

Le tableau résultat `dst` doit être passé en argument car, comme indiqué précédemment, les fonctions ECLAT ne peuvent pas renvoyer de tableaux (car ce ne sont pas des valeurs immédiates).

```
1  let map_slice(a,b,f,src,dst) =
2    let rec aux(i) =
3      if i < b then
4        (let x = get(src,i) in
5         let y = f(x) in
6         set(dst,i,y);
7         aux(i+1))
8      else ()
9    in
10   if a >= b then ()
11   else aux(a) ;;
12
```

```
13 let par_map(p,f,src,dst) =
14   let n = length src in
15   let d = n/p in
16   parfor i = 0 to p-1 do
17     map_slice(d*i,d*(i+1),f,src,dst)
18   done;
19   map_slice(d*p,n,f,src,dst) ;;
20
21 let map((p,f,src),k) :
22   ((_*_*_ array< 'N >) * _) =
23   let dst = create< 'N >() in
24   par_map(p,f,src,dst); k(dst) ;;
```

FIGURE 7.19 Une implantation parallèle de `map`

La fonction `map` (lignes 20-22) est une extension de `par_map`, évitant de passer explicitement le tableau résultat, ce qui facilite la composition des traitements sur les tableaux.

Pour cela, `map` prend (en plus de `p`, `f` et `src`) une *continuation* `k` (c'est-à-dire, une fonction qui encapsule la suite de l'exécution) ; `map` alloue localement un tableau destination `dst` (ligne 21), appelle `par_map(p,f,src,dst)` et applique ensuite `k` à `dst`.

Par exemple, transformer un tableau avec une fonction `f` puis transformer le tableau résultant avec une fonction `g` (pour un degré donné de parallélisme `p`) puis appliquer `k` sur le tableau résultant s'écrit `map((p,f,src),(fun a -> map((p,g,a),k)))`. En pratique, on utilise une abréviation, $f\ e_1$ `@@` e_2 pour f`(`e_1`,`e_2`)`. Dans l'exemple ci-dessus avec deux `maps`, cela donne `map(p,f,src) @@ (fun a -> (map(p,g,a) @@ k))`.

7.8.2 Fermes de travailleurs

Sur la figure 7.20 est définie une autre implantation parallèle `map_farm`, utilisant une *ferme de travail* pour appliquer *à la demande* la fonction `f` aux éléments du tableau `src`. Cette méthode est bien adaptée lorsque `f` a un temps d'exécution irrégulier en fonction des éléments de `src` (par exemple, la fonction `collatz`, figure 3.13). Les travailleurs communiquent par l'intermédiaire d'une variable partagée `r` (implantée comme un tableau de taille 1) contenant l'indice du prochain élément du tableau à traiter.

Chaque fois qu'un travailleur est disponible, il : **(1)** lit l'indice donné par `r` ; **(2)** incrémente `r` ; **(3)** traite un élément ; et ce jusqu'à ce que l'entier `i` référencé par `r` atteigne la fin du tableau source.

Le point important est que les étapes **(1)** et **(2)**, c'est-à-dire la lecture et l'écriture de `r` (lignes 3 à 5), sont toujours exécutées ensemble de façon atomique (comme expliqué à la section 6.2.2).

```
1 let map_worker (f,r,src,dst) =
2   let rec loop() =
3     let i = get(r,0) in
4     if i >= length src then () else
5       let ((),x) = (set(r,0,i + 1)
6                     || get(src,i)) in
7       (set(dst,i,f(x)); loop())
8   in loop() ;;
```

```
9  let map_farm(((p,f,src),k)
10   : ((_*_*(_ array< 'N >))*_)) : unit =
11   let r = create<1>() in
12   set(r,0,0);
13   let dst = create< 'N >() in
14   parfor _ = 1 to p do
15     map_worker(f,r,src,dst)
16   done;
17   k(dst) ;;
```

FIGURE 7.20 Implantation parallèle de `map` utilisant une ferme de travailleurs

7.8.3 Implantation de `reduce`

Sur la figure 7.21 est définie une implantation du squelette de parallélisme `reduce` à l'aide d'une ferme de travailleurs, en supposant que la fonction de réduction `f` de `reduce` est commutative et associative. Suivant le degré `p` de parallélisme souhaité, `p` travailleurs sont placés en parallèle (avec `parfor`, ligne 14 à 18). Chaque travailleur à son propre accumulateur. Tant qu'il reste des éléments à traiter dans le tableau `src`, chaque travailleur pioche le prochain élément à traiter et met à jour l'accumulateur en appliquant `f` à cet élément et à l'ancienne valeur de l'accumulateur. Lorsqu'un travailleur termine, il écrit la valeur de son accumulateur dans une case du tableau `a` (ligne 16). Une fois que tous les travailleurs ont terminé, le tableau `a` est réduit avec `f` en utilisant une version séquentielle de `reduce`, appelée `reduce_seq`. À noter que l'utilisation d'une version séquentielle de `reduce` ici évite d'instantier `p` nouveaux circuits réalisant `f`. En effet, cela aurait été couteux en ressources matérielles avec un avantage limité en terme de temps d'exécution puisque, généralement, la taille N de tableau domine le degré de parallélisme `p` (par exemple, $N = 3000$ et $\texttt{p} = 16$).

7.8.4 Implantation de `pipe`

À la figure 7.22 est défini un squelette `pipe3`. Fonctionnellement, `pipe3((f1,f2,f3,src),k)` est équivalent à `map(1,f1,src) @@ fun a1 -> map(1,f2, a1) @@ fun a2 ->map((1,f3,a2),k)`, mais ces `map`s successifs (ou *étages de pipelines*) sont parallélisés. L'implantation décrite ici pourrait facilement être généralisée [10] à la main pour obtenir plus de parallélisme (c'est-à-dire,

10. Cependant, ECLAT n'offre pas encore d'outils de métaprogrammation assez puissant pour écrire une version générique de `pipe` (en nombre d'étages de pipeline) dans le langage.

```
1  let reduce_worker (f,r,v0,src) =
2    let rec loop(acc) =
3      let i = get(r,0) in
4      if i < length src then (
5        let ((),v) = (set(r,0,i+1)
6                    || get(src,i)) in
7        loop (f(v,acc))) else acc
8    in loop(v0) ;;
```

```
let reduce_farm(p,f,v0,src) =                          9
  let r = create<1>() in                               10
  set(r,0,0);                                          11
  let a = create<p>() in                               12
  parfor i = 1 to p do                                 13
    let x = reduce_worker(f,r,v0,src)                  14
    in set(a,i-1,x)                                    15
  end;                                                 16
  reduce_seq(f,v0,a) ;;                                17
```

FIGURE 7.21 Implantation parallèle de `reduce` utilisant une ferme de travailleurs

plus d'étages de pipeline).

Sur le côté gauche de la figure 7.22 est définie une fonction `semi_pipe`, qui consomme progressivement les éléments x_i de `src` dès que leur position i est inférieure à l'indice référencé par `rI`. La fonction `f` est appliquée à chaque élément x_i consommé. Ensuite, `semi_pipe` assigne la valeur résultante dans le tableau `dst` (ligne 9) et incrémente la variable partagée `r0` (ligne 10) pour notifier que `dst` est prêt à être traité jusqu'à l'indice i.

Sur le côté droit de la figure 7.22, `pipe3` crée des tableaux intermédiaires et des variables partagées (ces dernières étant initialisées à 0, sauf pour `r0`, qui prend la valeur `n = length src` pour notifier que `src` est prêt à être consommé). Puis `pipe3` appelle ensuite en parallèle les `semi_pipe`s (avec un point de synchronisation entre eux), puis passe le tableau `dst` à la continuation `k`.

```
1  let semi_pipe(f,(rI,src),
2                (r0,dst)) =
3    let rec loop(i) =
4      if i < length src then (
5        let (j,x) = (get(rI,0)
6                    || get(src,i))
7        in if i < j then
8             let y = f x in
9             set(dst,i,y);
10            set(r0,0,i+1);
11            loop(i+1)
12          else (loop(i)))
13     else ()
14   in loop(0)  ;;
```

```
let pipe3 (((f1,f2,f3,src),k)                          15
  : ((_*_*_*(_ array< 'N >))*_)) : unit =              16
  let r0 = create<1>() in                              17
  let (a1,r1) = (create<'N>(), create<1>()) in         18
  let (a2,r2) = (create<'N>(), create<1>()) in         19
  let (a3,r3) = (create<'N>(), create<1>()) in         20
  let n = length src in                                21
  let _ = (set(r0,0,n) || set(r1,0,0)                  22
        || set(r2,0,0) || set(r3,0,0)) in              23
  let _ =                                              24
    ((semi_pipe(f1,(r0,src),(r1,a1))                   25
    || semi_pipe(f2,(r1,a1),(r2,a2)))                  26
    || semi_pipe(f3,(r2,a2),(r3,a3)))                  27
  in k(a3) ;;                                          28
```

FIGURE 7.22 Implantation d'un pipeline à trois étages en ECLAT

7.8.5 Prédiction des performances

Un aspect essentiel de la sémantique synchrone d'ECLAT est sa capacité à prédire le temps d'exécution de programmes parallèles. Illustrons cela par un exemple. Le squelette `par_map` (défini à la figure 7.19) appelle `p` fois `map_slice(a,b,f,src,dst)` en parallèle. Chacun de ces appels comprend un appel direct (1_D cycle) à la fonction `aux` (qui prend un cycle d'horloge) suivi d'une lecture (1_L cycle), d'un appel à `f`, d'une écriture (1_E cycle) et d'un appel récursif terminal (1_R cycle) par itération. Soit $T(\mathtt{f}_i)$ le temps d'exécution de `f` lorsqu'elle est appliquée au i-ème élément du tableau `src` et soit N la taille de `src`. Le temps d'exécution de `map_slice` (en cycles) est :

$$T_{\mathtt{map_slice}}(\mathtt{a},\mathtt{b}) = 1_D + \sum_{i=\mathtt{a}}^{\mathtt{b}-1} \Big(1_L + T(\mathtt{f}_i) + 1_E + 1_R\Big)$$

Par souci de simplicité, supposons que `f` est sans état [11] et que N est un multiple de `p`. En raison du caractère équitable (*fair*) de l'exécution (comme expliqué à la section 6.2.2), le temps d'exécution de `map_par` correspond au temps d'exécution maximal parmi les $T_{\mathtt{map_slice}}$, plus une quantité de temps $T_{\mathtt{stalls}}$ (ou *cycles perdus*) passée à séquentialiser les accès concurrents [12] :

$$T_{\mathtt{map_par}} = \max_{i=0}^{\mathtt{p}-1} \Big(T_{\mathtt{map_slice}}(i \times N/\mathtt{p}, (i+1) \times N/\mathtt{p})\Big) + T_{\mathtt{stalls}}$$

Si le temps de calcul domine le temps d'accès, alors $T_{\mathtt{stalls}}$ devient négligeable. Notons que $T_{\mathtt{stalls}}$ est clairement inférieur à $2N$, qui est le coût d'un parcours séquentiel complet de `src` (N lectures) avec mise à jour de `dst` (N écritures).

Le chronogramme de la figure 7.23 représente le début de l'exécution de `map_par` pour un calcul régulier tel que $T(\mathtt{f}_i) = T_{\mathtt{f}}$ et $T_{\mathtt{f}} = \mathtt{p} - 2$. Les cycles perdus sont notés ✗. Il y a $\mathtt{p} - 1$ cycles perdus (✗) pendant l'amorçage, puisque chaque appel à `map_slice` en parallèle tente simultanément de lire `src`, mais qu'un seul acquiert le verrou ($\mathcal{A}_{\mathtt{src}}$) à la fois. Notons aussi que, dans ce cas précis, une fois l'amorçage terminé, il n'y a plus de cycle perdu. Nous pouvons en déduire que, pour toute fonction sans état `f` telle que $T_{\mathtt{f}} \geq \mathtt{p} - 2$, nous avons $T_{\mathtt{stalls}} = \mathtt{p} - 1$ cycles. En d'autres termes, si $T_{\mathtt{f}} \geq \mathtt{p} - 2$, le temps de calcul de `f` domine les temps d'accès à la mémoire.

Ce résultat théorique est vérifié en pratique. Par exemple, pour $N = 3200$, $\mathtt{p} = 16$ et `f` telle que $T_{\mathtt{f}} = \mathtt{p} - 2 = 14$ cycles, le temps d'exécution mesuré de `par_map` est de 3416 cycles, c'est-à-dire $(1 + \max_{i=0}^{\mathtt{p}-1}(1+14+1+1)N/\mathtt{p}) + (\mathtt{p}-1)$. C'est 15.93 fois mieux que pour $\mathtt{p} = 1$, ce qui est quasiment optimal vis-à-vis du degré de parallélisme. La figure 7.24 donne les facteurs d'accélération mesurés (à gauche) et les temps d'exécution (à droite) pour `map_par` en fonction de `p` et $T_{\mathtt{f}}$ par rapport à $\mathtt{p} = 1$. Cela suggère clairement que, si le rapport $T_{\mathtt{f}}/\mathtt{p}$ est faible, les accès dominent le calcul et la vitesse n'augmente plus avec `p`. On peut en déduire que, par

11. Les programmeuses et programmeurs sont libres de composer des comportements complexes (comme par exemple, des accès à des tableaux globaux depuis la fonction `f` à paralléliser) : cela reste déterministe, avec un compromis entre expressivité et prédictibilité.

12. Chaque `map_slice` lit `src` et écrit `dst` de manière concurrente.

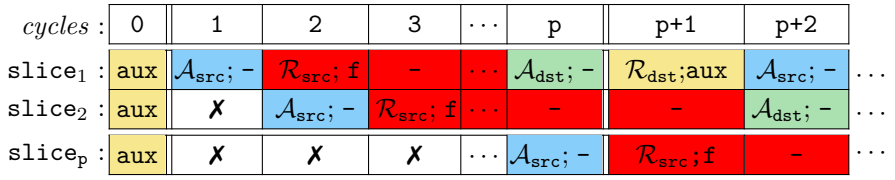


FIGURE 7.23 Amorçage de `map_par` avec $T(\mathtt{f}_i) = T_{\mathtt{f}} = \mathtt{p} - 2$

exemple, il est inutile de paralléliser `f` plus de huit fois si $T_{\mathtt{f}} = 6$, car le gain de vitesse pour $\mathtt{p} = 8$ est optimal.

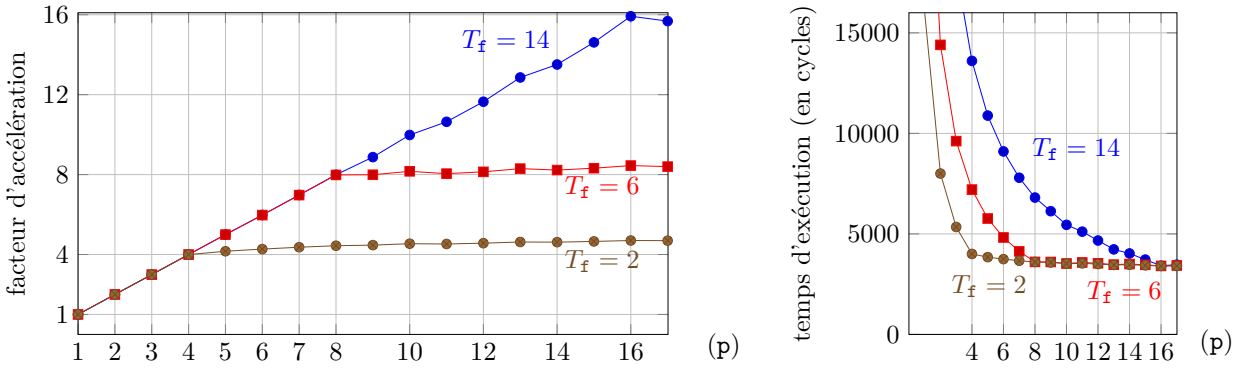


FIGURE 7.24 Accélération et temps d'exécution pour `map_par` en fonction de `p` et $T_{\mathtt{f}}$

Pour des squelettes plus avancés comme `map_farm` (figure 7.20) et `pipe` (figure 7.22), la prédictibilité au cycle d'horloge près est impraticable. Toutefois, l'estimation des performances reste possible et intuitive. Cela permet de contrôle finement le degré de parallélisme pour explorer le compromis temps-espace, comme nous le montrons dans l'exemple suivant (Section 7.8.6).

7.8.6 Les n reines avec des squelettes

La section 7.5 a montré qu'il était possible d'obtenir un gain d'accélération de l'ordre de n pour le problème des n reines en instanciant n sous-problèmes.

Cette réalisation parallèle du problème des n reines a cependant un défaut : elle ne permet pas de choisir le degré de parallélisme `p`, puisque celui-ci est imposé par la taille du problème (c'est-à-dire, $\mathtt{p} = \mathtt{n}$).

Nous proposons donc, sur la figure 7.25, une nouvelle implantation du problème des n-reines avec un degré de parallélisme `p` décorrélé de n. Pour cela, nous créons un tableau de `n` entiers initialisés de `1` à n (chaque entier désigne un sous-problème) ; puis nous parcourons le tableau avec le squelette `map_farm` traitant successivement `p` sous-problèmes en parallèle. Le temps d'exécution mesuré pour un degré de parallélisme p donné est optimal et la taille du circuit engendré dépend linéairement de `p` (constant) et non plus de `n`.

Dans la table donnée figure 7.26, nous comparons les temps d'exécution de `queens_unfold`(8),

```
1  let queens_skel (p,n,k) =
2    if p > n then -1 else
3    let v = vect_create<n>(1) in
4    let r = create<1>() in
5    let a = create<n>() in
6    let rec init_array(i) =
7      if i < n then (set(a,i,i+1); init_array(i+1)) else ()
8    in init_array(0);
9    let f(i) =
10     let v = vect_copy_with(v,0,i) in
11     loop (2, 1, 1, n, v, 0)
12   in
13   map_farm(p,f,a) @@ fun r ->
14     reduce((+),0,r) ;;
```

FIGURE 7.25 Parallélisation des n reines (adaptée de `queens_par`) avec des squelettes

`queens_par`(8) et `queens_skel`(p,8,k) pour différents p et en négligeant le temps d'exécution de la continuation k. Dans le cas ou $p = 8$, le surcoût induit par l'utilisation de squelettes (comprenant la gestion d'une ferme de calcul dans le squelette `map_farm` défini sur la figure 7.20) par rapport à `queens_par` est : $2489/2446 = 1,018$. Tout l'intérêt de la version avec squelettes est d'avoir une plus grande maîtrise sur le degré de parallélisme p et donc sur la taille du circuit (en nombre de LE). On peut remarquer que doubler la valeur de p double le nombre de LE ; pour $p = 8$, l'utilisation de squelettes induit un surcoût en nombre de LE par rapport à `queens_par`(8).

appel	durée (en nombre de cycles)	nombre d'éléments logiques (LE)
`queens_unfold`(8)	18513	1091
`queens_skel`(1, 8, k)	18574	865
`queens_skel`(2, 8, k)	9308	1507
`queens_skel`(4, 8, k)	4820	3446
`queens_skel`(8, 8, k)	2489	6974
`queens_unfold_par`(8)	2446	4457

FIGURE 7.26 Contrôle du degré de parallélisme dans le calcul des n reines à base de squelettes

Les squelettes de programmation constituent une barrière d'abstraction qui permet de s'affranchir de certains détails de bas niveau (comme les communication entre travailleurs dans un ferme). L'intérêt d'ECLAT est d'être suffisamment précis pour que l'on puisse programmer directement ces squelettes dans le langage, et les utiliser librement dans les programmes.

7.9 Implantation matérielle de la machine virtuelle OCaml

L'ensemble de ce chapitre montre l'expressivité d'Eclat pour la programmation d'applications sur FPGA. On peut cependant noter qu'en Eclat, il n'y a pas de structures de données dynamiques ou irrégulières, qui sont pourtant utiles pour programmer certains types d'algorithmes. Dans cette section on montre comment s'affranchir de cette limitation en proposant une implantation de la machine virtuelle OCaml et d'une bibliothèque d'exécution réduite pour OCaml. Les flottants et les traits orientés objets ne sont actuellement pas supportés dans le prototype présenté, mais cela montre l'avantage de cette approche "machine virtuelle" pour la programmation de FPGA, dans la lignée des travaux sur O2B [201] : accélération matérielle de l'environnement d'exécution OCaml, appels d'accélérateurs matériels et interaction avec les entrées/sorties du FPGA cible.

7.9.1 La machine virtuelle OCaml

La machine virtuelle OCaml [128] est une machine à pile adaptée de la machine de Krivine [119] avec évaluation en appel par valeur plutôt qu'en appel par nom.

Elle est capable d'allouer des fermetures, avec un modèle d'exécution optimisé pour les fonctions unaires *à la ML* afin d'éviter l'allocation de fermetures intermédiaires. Le jeu d'instructions de la machine comprend 148 instructions manipulant une pile, un tas, un segment de données global et des registres spécialisés tels que le pointeur de pile.

L'implantation matérielle que nous proposons suit de près la spécification le la machine virtuelle OCaml. En guise de limitation, elle ne prend pas en charge les opérations en virgule flottante et les objets.

7.9.2 Chaîne de compilation pour OCaml sur FPGA

La figure 7.27 schématise le flot de compilation proposé pour OCaml sur FPGA.

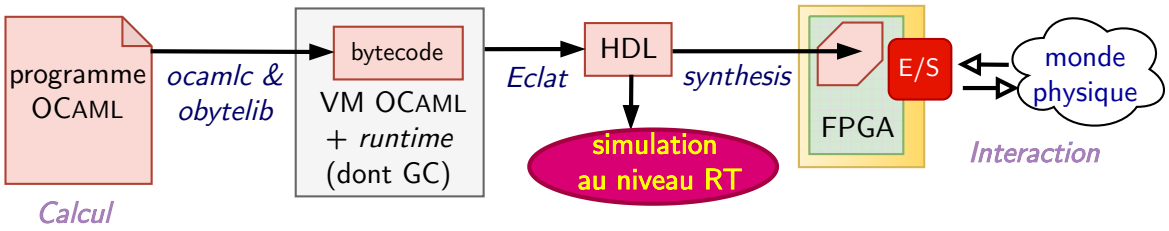


Figure 7.27 Implantation d'OCaml sur FPGA en utilisant Eclat

Le bytecode produit par le compilateur de bytecode OCaml `ocamlc` est d'abord traité à l'aide de la bibliothèque `obytelib` [13] pour obtenir une représentation plus compacte que nous codons comme un fichier Eclat nommé `bytecode.ecl`.

Ce fichier contient :

13. `https://github.com/bvaugon/obytelib`

1. une fonction d'initialisation `init_data` pour charger dans la mémoire de la machine les données globales du programme OCaml ;
2. une fonction `external_call` répartissant les appels de fonctions externes depuis l'interprète de bytecode OCaml ;
3. un tableau global `code` encodant chaque instruction de bytecode du programme OCaml sous la forme d'une séquence d'entiers (par exemple, la séquence "`19 ; 2`" signifie l'instruction `POP(2)` où `19` est l'*opcode* de l'instruction de bytecode `POP`, laquelle attend un paramètre : ici l'entier littéral `2`).

Ce fichier engendré est ajouté au code source de la machine virtuelle OCaml (écrite elle aussi en Eclat). Le byte-code d'un programme OCaml, la machine virtuelle et sa bibliothèque d'exécution avec récupérateur automatique de mémoire (GC) forment un seul programme Eclat. Ce programme est compilé en VHDL par le compilateur Eclat. La description matérielle résultante peut être soit utilisée pour simuler l'exécution du bytecode au niveau transfert de registres (RTL), soit synthétisée à l'aide d'un outil de synthèse logique.

7.9.3 Un environnement d'exécution paramétrable

Le compilateur OCaml standard adopte une représentation uniforme des valeurs. Toute valeur OCaml est soit un entier (*i.e.*, une valeur immédiate) soit un pointeur vers un bloc de mémoire. Un *bit de marque* est utilisé pour différencier les entiers des pointeurs (pour le GC). Sur les architectures CPU, les adresses sont alignées sur la taille du mot (par exemple, 32 ou 64 bits). Par conséquent, leur bit de poids faible est toujours égal à zéro. L'implantation standard de la machine virtuelle OCaml (appelée `ocamlrun`) utilise ce bit comme bit de marquage **0**. Inversement, les entiers ont le bit de marquage **1**. Par conséquent, les entiers natifs OCaml sont des entiers de 31 bits (ou des entiers de 63 bits en fonction de la taille du mot) décalés vers la gauche.

Notre implantation en Eclat utilise un encodage différent, définie figure 7.28a. Celle-ci tient compte du fait que les n-uplets en Eclat sont des valeurs immédiates. Chaque valeur OCaml, représentée en Eclat par un nouveau type `value`, est une paire formée d'un entier (de type `long`) et d'un bit de marque (de type `bool`). Le type `long` est un entier dont la taille est personnalisable.

Toute la mémoire nécessaire pour la machine est allouée statiquement dans un tableau global `ram`, de taille personnalisable, dans lequel sont placées les données globales, la pile et le tas (Fig. 7.28b). Les accès et les modifications de ce tableau Eclat sont séquentiels. Chaque bloc de mémoire alloué dans ce tableau est constitué d'un en-tête (codant une taille, une étiquette et, éventuellement, des informations supplémentaires pour le gestionnaire de mémoire) suivi de valeurs OCaml contiguës. Par exemple, toute liste OCaml est encodée soit par l'entier 0 (représentant la liste vide) soit par un pointeur sur un bloc de trois cellules mémoire contiguës : la première pour l'en-tête, la suivante pour la tête de la liste et la dernière pour la queue de liste.

```
type long = int<31> ;;
type value = long * bool ;;

let is_int (_,b) = b ;;
let long_val (n,_) = n ;;
let val_long (n) = (n,true) ;;
```

(a)

```
let ram = make<16384> (0,true) ;;

let data_start = 0 ;;
let stack_start = 1000 ;;
let heap_start = 4000 ;;
let heap_size = 6000 ;;
```

(b)

FIGURE 7.28 Taille de mot paramétrable et partitionnement de la mémoire pour OCAML

Nous avons implanté un GC de type Stop&Copy [50] pour gérer la mémoire dynamique dans le tableau `ram`. Ce GC nécessite de doubler la taille du tas pour l'organiser en deux compartiments (*semi-espaces*) : l'interprète de bytecode alloue les blocs de mémoire dans l'un des deux compartiments. Lorsque ce compartiment est plein, l'exécution du programme est interrompue afin de copier, de ce compartiment vers l'autre compartiment, tous les blocs accessibles depuis les racines du GC (c'est-à-dire les valeurs contenues dans la pile, le segment de données global et les registres) ; les adresses sont ensuite mises à jour dans les valeurs restantes et les rôles des deux compartiments sont échangés.

7.9.4 Efficacité

Sans surprise, l'exécution du bytecode OCAML est beaucoup plus lente qu'un programme ECLAT fonctionnellement équivalent. La figure 7.29 montre la perte de vitesse à utiliser notre implantation de la machine virtuelle OCAML sur FPGA pour calculer une solution au problème des n reines, vis-à-vis de la version ECLAT non optimisée. Les codes ECLAT et OCAML sont fonctionnellement équivalent (et il s'agit du même code OCAML que celui utilisé pour la comparaison vis-à-vis d'un CPU). La version OCAML (`queens_vm_ocaml`) est 200 fois plus lente que la version ECLAT.

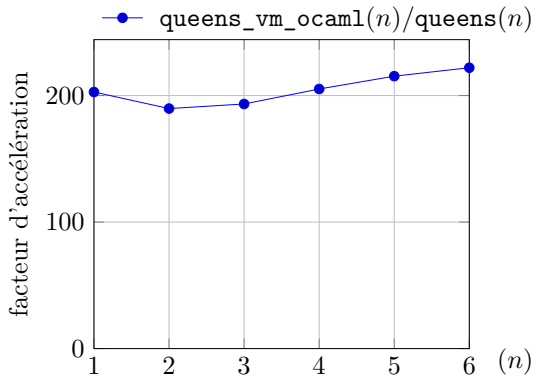


FIGURE 7.29 Perte de vitesse du calcul des n reines en bytecode OCAML sur FPGA vs ECLAT

L'implantation de cette machine virtuelle OCaml et de sa bibliothèque d'exécution utilise 44 % des éléments logiques et 33 % des blocs de mémoire RAM disponibles sur le FPGA Max10.

Ces performances sont certes décevantes mais le but de cette expérience de réalisation de la machine virtuelle OCaml est de montrer l'expressivité d'Eclat en tant que langage d'implantation et de donner accès à des traits de programmation disponibles en OCaml mais pas en Eclat, comme par exemple les exceptions, les structures de données avec partage, la gestion mémoire, les valeurs fonctionnelles et la récursion générale.

Nous avons réalisé une comparaison expérimentale entre cette implantation de la machine virtuelle OCaml et l'implantation O2B (citée dans l'état de l'art, section 1.3.4) basée sur un processeur *softcore*. Ces résultats, qui ont été publiés [203], montre que sur de petits programmes exploitants les différents traits de programmation d'OCaml, notre implantation est en moyenne 45 fois plus rapide qu'O2B sur *softcore*.

7.9.5 Interopérabilité entre Eclat et OCaml

Pour accélérer certains calculs ou étendre l'environnement d'exécution d'OCaml avec de nouvelles primitives qui ne sont pas définissables en OCaml, nous avons porté, dans notre implantation matérielle d'OCaml en Eclat, le mécanisme d'appel de fonctions externes (*Foreign Function Interface*, FFI) d'OCaml. Cela permet aux programmes OCaml, exécutés par notre implantation de machine virtuelle, d'appeler des fonctions externes définies en Eclat. En outre, la machine virtuelle elle-même est définie comme une fonction Eclat, qui peut être intégrée dans des applications Eclat interagissant avec l'environnement extérieur.

7.9.6 Appel de fonctions Eclat depuis les programmes OCaml

Supposons que, pour améliorer les performances, nous voulions utiliser l'implantation Eclat de la fonction `fibonacci` (définie figure 3.6) depuis un programme OCaml, au lieu d'une version bytecode écrite en OCaml [14].

La fonction `fibonacci` ne peut pas être appliquée directement à des valeurs OCaml puisque OCaml et Eclat n'ont pas la même représentation des valeurs (confère le type "`value`" et les primitives `long_val/val_long` définies figure 7.28a).

C'est pourquoi nous utilisons une fonction `fibonacci_glue` écrite en Eclat qui convertit une valeur OCaml `v` en valeur Eclat et la transmet à `fibonacci` :

```
let fibonacci_glue ((v,_),st) =
  let n = fibonacci(long_val(v)) in
  (val_long(n),st)
```

Par convention, on fait en sorte que de telles fonctions appelées à partir d'OCaml prennent un paramètre supplémentaire `st` (l'état de la machine), qui peut être modifié par la fonction, et qui est renvoyé.

14. Les deux versions étant, d'ailleurs, parfaitement similaires.

La fonction `fibonacci_glue` peut alors être appelée depuis les programmes OCaml en utilisant la FFI OCaml [15], par exemple :

```
external fibonacci_ext : int -> int = "fibonacci_glue"

let _ =
  List.iter print_int (
    List.map2 fibonacci_ext [5;6;7;8; ···]
  )
```

Cela permet des gains de performance importants en appelant, en OCaml, des accélérateurs matériels écrits en Eclat.

Par exemple, la fonction `fibonacci_ext` est expérimentalement 65 fois plus rapide qu'une fonction OCaml équivalente compilée en bytecode et exécutée par notre interpréteur de bytecode OCaml écrit en Eclat et ciblant le même FPGA.

De plus, les fonctions externes Eclat peuvent avoir des implantations parallèles (utilisant la construction Eclat $(e_1 \| e_2)$ ou la construction dérivée `let/and/in`) pour augmenter encore les facteurs d'accélération, en fonction du degré de parallélisme disponible. Par exemple, appeler 16 fois `fibonacci` en parallèle avec le même argument est $16 \times 65 \approx 1000$ fois plus rapide qu'une version bytecode OCaml exécutée par notre implantation d'OCaml en Eclat.

7.9.7 Accéder à des structures de données OCaml depuis Eclat

Les fonctions externes Eclat peuvent opérer sur des structures de données OCaml en utilisant les primitives définies dans la bibliothèque d'exécution OCaml, puisque celle-ci est aussi écrite en Eclat.

La figure 7.30 définit par exemple une fonction Eclat `length` calculant la longueur d'une liste OCaml (c'est-à-dire un pointeur, de type `value`, comme défini sur la figure 7.28a). Elle utilise deux primitives : `is_empty` (de type `value` ⇒ `bool`) pour vérifier si une liste est vide, et `list_tail` (de type `value` → `value`) pour accéder à la queue de la liste OCaml.

```
val length : value → int
```

```
let length(lst) =
  let rec aux(lst,acc) =
    if is_empty(lst) then acc
    else aux(list_tail(lst),acc + 1)
  in aux(lst,0)
```

Figure 7.30 fonction Eclat calculant la longueur d'une liste OCaml

15. Ce mécanisme général pour faire interopérer OCaml avec son langage d'implantation (ici Eclat) est très similaire à celui utilisé dans les implantations OCaml écrites en C, sur CPU, avec des fonctions externes C.

Expérimentalement, cette fonction ECLAT `length` est 12 fois plus rapide qu'une version équivalente OCAML compilée en bytecode et exécutée par l'interprète de bytecode OCAML écrit en ECLAT.

Ce facteur d'accélération est raisonnable car la fonction `length` effectue peu de calculs (une seule incrémentation, `acc + 1`) : le temps de calcul est dominé par les accès séquentiels à la mémoire, qui prennent le même temps en ECLAT et OCAML.

7.9.8 Embarquer du code OCaml dans des applications réactives

Notre implantation d'OCAML est écrite en ECLAT comme une fonction instantanée `ocaml_vm` (en utilisant les constructions **exec** et **reg** présentées au chapitre 3). L'exécution d'une instruction de bytecode prend toujours plusieurs cycles d'horloge (en particulier en cas de déclenchement de GC). C'est pourquoi la fonction `ocaml_vm` a une sortie spéciale *busy* correspondant à la sortie *rdy* de la construction **exec** utilisée à l'intérieur de l'application. Elle dispose également d'une sortie *stop* qui signale la fin de l'exécution du programme OCAML. D'autres entrées et sorties peuvent être ajoutées à la fonction `ocaml_vm`, cela dépend des besoin de l'application embarquée.

La figure 7.31 est un exemple simple d'application réactive écrite en ECLAT. Elle prend en entrée deux boutons (`button1` et `button2`) comme entrées et contrôle trois LEDs en sortie. Le point d'entrée (la fonction `main`) est appelé à chaque cycle d'horloge : elle exécute l'interprète de bytecode OCAML (spécialisée pour un programme OCAML donné) en appelant la fonction `ocaml_vm` avec `button2` en argument.

```
val ocaml_vm : bool ⇒ (bool * bool * value)
val main : (bool * bool) ⇒ (bool * bool * bool)
```

```
let main(button1,button2) =
  if button1 then (false,false,false)
  else (let (stop,busy,result) = ocaml_vm (button2) in
        let (red,green) = (busy,stop) in
        let blue = stop && (long_val(result) == 42) in
        (red,green,blue))
```

FIGURE 7.31 Une application réactive écrite en ECLAT et embarquant du code OCAML

Lorsqu'on appuie sur le bouton `bouton1`, l'exécution de la machine est suspendue et les trois voyants s'éteignent. Sinon, le voyant rouge indique que le machine est occupée ; la LED verte indique que l'exécution du bytecode OCAML est terminée ; la LED bleue indique que le bytecode OCAML a produit, dans un registre dédié `result`, une valeur égale à 42.

L'implantation d'OCAML sur FPGA avec ECLAT est un exemple marquant du mélange possible entre calcul et interaction en ECLAT : Cela montre clairement qu'ECLAT n'impose aucune limitation d'expressivité dans les calculs décrits. C'est aussi une voie alternative pour

l'implantation d'OCaml sur FPGA, bien plus efficace que l'approche existante avec la plateforme O2B sur un processeur *softcore* [201]. La bibliothèque d'exécution pour OCaml écrite en Eclat est personnalisable ; en particulier, on peut y intégrer *facilement* des accélérateurs matériels afin de les appeler, en tant que fonctions externes, depuis les programmes OCaml. À noter que les *benchmarks* OCaml qui ont été réalisés sont des programmes assez petits [203], exploitant les différents traits du langages. La bibliothèque d'exécution n'est pas complète : elle n'a actuellement pas de support pour les flottants, les objets, les comparaisons polymorphes (*i.e.*, la fonction `Stdlib.compare` d'OCaml) et l'égalité structurelle [16] Dans des travaux futurs, nous compléterons cette bibliothèque d'exécution et évaluerons cette implantation d'OCaml avec de plus gros programmes. D'autres pistes de recherche incluent la coopération de plusieurs instances de la machine virtuelle fonctionnant en parallèle, avec leur propre pile et tas, mais communiquant via des registres ou une mémoire partagée. Des squelettes de parallélisme comme ceux implantés à la section 7.8 pourraient être mis en œuvre sur des tableau OCaml dynamiquement alloués par la machine virtuelle, comme dans l'expérience O2B [201]. Enfin, nous pourrons estimer les performances (en nombre de cycles) des instructions de bytecode avec des applications possibles dans des applications embarquées temps réel.

Conclusion

Eclat répond aux difficultés qui se posent lorsque l'on programme une application sur FPGA : accéder à la mémoire, exploiter le parallélisme à grain fin à travers le compromis temps-espace, trouver la bonne abstraction de programmation pour un problème donné et interagir avec le monde physique. L'évaluation expérimentale réalisée montre qu'Eclat offre de bonnes performances en termes de temps d'exécution et de taille des circuits engendrés, avec un contrôle très fin sur les aspects architecturaux de bas niveau et un style de programmation généraliste de haut niveau.

16. Le parcours récursifs des valeurs OCaml nécessite une pile explicite puisque les fonctions Eclat doivent être récursives terminales.

8 Travaux connexes

Notre travail sur le langage Eclat, pour le mélange d'interaction et de calculs sur FPGA, cherche à réconcilier sous un modèle synchrone **(1)** la description de matériel en style fonctionnel et **(2)** la composition de calculs par *synthèse de haut niveau* (HLS).

Ces deux approches ont été présentées dans l'état de l'art sur la programmation FPGA (aux sections 1.3.1 et 1.3.5) comme des réponses aux difficultés de programmation au niveau RT. Or ces réponses sont complémentaires. La description de matériel en style fonctionnel repose sur le concept de *fonctions instantanées*, opérant sur des *flots*, pour spécifier et composer des circuits de façon synchrone. La HLS, au contraire, a pour but de faciliter la programmation de calculs, s'exécutant en un nombre arbitraire de cycles, en s'appuyant sur des compilateurs optimisants ciblant le niveau RT.

Ce chapitre détaille les avantages et les limitations de ces deux approches, puis il présente des travaux connexes se situant à leur jonction ainsi que le positionnement d'Eclat par rapport à ces travaux.

8.1 Description de matériel en style fonctionnel

Les HDL en style fonctionnel (voir [48] pour un survey) sont capables de définir très précisément des circuits sous forme de *fonctions instantanées*. Ils font usage de fonctions d'ordre supérieur (appelés *combinateurs*) comme *map* et *scan* [99]. Cela constitue un puissant mécanisme de génération de code pour exploiter le parallélisme à grain fin disponible sur le circuit FPGA. Par exemple appliquer une fonction f à tous les éléments d'un tableau a de taille n (qui s'écrit simplement $map(f, a)$) crée n instances de la fonction f en parallèle. Le compromis temps-espace qui en résulte s'exprime simplement et de façon concise. De plus, le comportement temporel du circuit est prédictible, de même que l'estimation des ressources matérielles avant optimisation par l'outil de synthèse [1].

Ce style de programmation est directement disponible en Eclat. Par exemple, à la section 7.7.2, nous avons implanté en Eclat le *Jeu de la Vie* [62] sous la forme d'une fonction instan-

1. Cette approche peut même être mise en œuvre au *niveau portes logiques* pour engendrer directement une *net-list* sans même utiliser un outil de synthèse. C'était d'ailleurs une caractéristique essentielle des travaux précurseurs en la matière comme μFP [196].

tanée mettant à jour la totalité de la matrice d'entrée à chaque cycle. Cela a induit un facteur d'accélération très important, allant jusqu'à 15 000, par rapport à une exécution séquentielle sur CPU (suivant la méthodologie expérimentale décrite section 7.1).

Cependant, la description de systèmes sous forme de fonctions instantanées demeure limitée en terme d'accélération. Par exemple, et comme nous l'avons montré au chapitre 7, l'implantation du Jeu de la vie mentionnée ci-dessus ne passe pas à l'échelle : au delà d'une taille de matrice relativement petite, la description de circuit n'est plus synthétisable car elle requiert trop d'éléments logiques pour être effectivement réalisable sur le circuit FPGA. Le passage à l'échelle requiert une utilisation suffisante des blocs RAM (BRAM) [225], d'où l'introduction d'une mémoire partagée en Eclat au chapitre 6.

Or, quelle que soit la cible FPGA, les accès en mémoire RAM ne sont *pas* instantanés ; cela rend ces accès difficilement exprimables et surtout composables dans les langages de description de matériel fonctionnels existants. En effet, dans ces langages, toute fonction est instantanée *par construction*. C'est une différence notable avec Eclat où l'instantanéité d'une expression est un propriété temporelle, inférée par typage statique : les fonctions instantanées et les calculs sont programmés en Eclat, avec les mêmes constructions de programmation, dans un style fonctionnel-impératif à la OCaml.

8.2 Synthèse de haut niveau (HLS)

La HLS est une approche populaire pour le prototypage rapide d'applications sur FPGA. Elle permet aux programmeuses et programmeurs d'utiliser un langage algorithmique séquentiel (généralement un sous ensemble de *C*, comme expliqué à la section 1.3.1), et de le *synthétiser* à l'aide d'un *outil de synthèse de haut niveau*, c'est-à-dire un compilateur ciblant le niveau RT.

L'avantage de cette approche est qu'elle ne requiert pas de connaissances en conception de matériel de la part des utilisatrices et utilisateurs. Eclat repose aussi sur cette philosophie qui vise à programmer et surtout composer des calculs en utilisant des traits de programmation orientés flot de contrôle.

En termes d'accélération matérielle, l'intérêt d'utiliser un FPGA réside donc dans la capacité du compilateur à exploiter le parallélisme disponible au niveau circuit. En outre, les outils HLS appliquent des optimisations agressives dont l'efficacité dépend fortement de la structuration du programme source et de l'utilisation de *pragmas*.

Les auteurs du langage Allo [49], qui sera présenté à la section 8.4, rapportent par exemple un cas symptomatique dans l'outil Vitis HLS. Vitis est l'un des outils HLS les plus populaires, y compris dans l'industrie [60] ; il est capable d'optimiser très largement des programmes C en entrée, même lorsqu'ils ceux-ci ne sont pas annotés avec des *pragmas*.

> Etant donnée une fonction f écrite en C et calculant un produit de matrices avec une latence observée de 112 cycles, on s'attendrait à ce que deux appels successifs à f réutilisent la même instance de f. En effet, ces deux appels ne sont pas exécutés simultanément et ne requièrent donc pas de duplication. On s'attendrait aussi à un doublement de la latence, soit $2 \times 112 = 224$ cycles, puisque la fonction est

appelée séquentiellement à deux reprises. Or, Vitis engendre une circuit avec une latence de 280 cycles et un doublement de l'utilisation en ressources. Une analyse minutieuse du code engendré montre que l'outil a créé deux instances de f et que la communication entre ces deux instances induit $280 - 224 = 56$ cycles perdus. Pour obtenir une implantation plus efficace, il faut ajouter l'annotation

```
#pragma allocation instances=f limit=1
```

pour que Vitis crée une instance unique de f. La latence observée sur le code engendré est alors bien de 224 cycles et l'instance de f n'est plus dupliquée. Cela correspond bien au circuit attendu.

Eclat utilise aussi ce type d'optimisation, mais de façon plus transparente et conformément à la sémantique temporelle du langage : si f est une fonction récursive terminale (déclarée avec le mot clé `rec`), alors une seule instance de la fonction est créée et partagée entre les appels non-simultanés à cette fonction. Il y a cependant des exceptions, en particulier en présence d'ordre supérieur ou de passages de tableaux en arguments, qui requièrent malgré tout une duplication car les fonctions et les tableaux ne sont pas des valeurs dans le code engendrés comme expliqué au chapitre 5 et 6.

Le comportement temporel des programmes HLS est non-prédictible. Cette source d'imprédictibilité et de non-déterminisme empêche le contrôle fin du parallélisme exploitable sur le circuit FPGA. C'est une différence fondamentale avec Eclat ainsi que les autres langages de description de matériel fonctionnels, comme discuté à la section 8.1.

L'exploration de l'espace de conception (*Design Space Exploration*) [34] vise à répondre à cette limitations des outils HLS. Cela consiste à instrumenter un outil HLS pour que celui itère le processus de compilation, l'analyse du code engendré et l'annotation du programme source avec des pragmas. Le but est d'obtenir *automatiquement* une solution optimale en fonction d'heuristiques qui se basent sur des critères comme l'utilisation en ressources, la fréquence maximale et la latence du circuit à engendrer.

Une autre voie, suivie par Eclat et par certains travaux connexes, consiste à remarquer que le compilateur HLS, par nature, *doit* découper l'exécution du programme source en une suite de cycles d'horloge pour engendrer une description au niveau RT. Le comportement temporel et le parallélisme est donc connu statiquement par le compilateur [2] et gagne à être reflété dans la syntaxe et la sémantique du programme source.

Eclat pousse cet argument plus loin étant donnée que sa sémantique synchrone vise à faciliter la programmation, suivant des principes simples déjà énoncés dans l'introduction : *toute construction de programmation du noyau Eclat (présenté aux chapitres 3 et 4) est instantanée sauf les appels de fonctions récursives terminales, lesquels prennent un cycle* [3]. L'effort de compilation pour respecter cette sémantique est important et nous tolérons que le code engendré comporte des inefficacités (en termes de fréquence maximale et d'utilisation des

2. À condition que l'ordonnancement soit statique et non dynamique [51].
3. Plus le temps de la récursion non forcément connu.

ressources) du moment qu'il respecte la sémantique temporelle du langage. C'est une similarité avec le langage Handel-C [131].

8.3 Construction de matériel dans des langages généralistes

Des langages dédiés à la conception de matériel au niveau RT ont été embarqués dans des langages hôtes généralistes comme Haskell, OCaml, Scala ou Python.

Cette approche repose sur une idée popularisée par le langage Hydra [153] : *représenter le circuit décrit par une structure de données* (c'est-à-dire, un arbre de syntaxe abstraite). Cette structure de données peut être manipulée dans le langage hôte à des fin de simulation, vérification, optimisation ou génération de code.

Jazz [26] est un langage orienté objet qui a été utilisé pour la synthèse de circuits ; il embarque un langage de *net-list* proche de Lustre. Lava [32] est un environnement de conception de circuits qui utilise Haskell comme langage hôte pour embarquer une variante de μFP [197].

Il existe de nombreuses bibliothèque de description et vérification de circuits suivant cette approche, comme par exemple Chisel [13] en Scala, PyRTL en Python [64], Hardcaml [171] en OCaml et ShakeFlow en Rust [99].

Dans ces langages, la description d'un circuit au niveau RT est le résultat de l'exécution d'un programme généraliste sur un ordinateur classique : celle-ci peut ensuite être synthétisée pour reconfigurer un FPGA.

Le niveau d'abstraction de ces langages de description de matériel correspond exactement au niveau transfert de registres (niveau RT), mais les techniques de méta-programmation offertes par les langages hôtes généralistes permettent de spécifier des circuits de façon plus concise et flexible. Il est possible, par exemple, de définir dans le langage généraliste des transformations de programmes, des interprètes ou des générateurs de code, définis récursivement sur la syntaxe des circuits, puis les appliquer.

La différence entre Eclat et un langage comme Chisel est fondamentale. En Eclat, l'utilisatrice ou l'utilisateur programme des calculs et des circuits réalisés sur un FPGA : la sémantique d'un programme Eclat spécifie le comportement temporel du circuit engendré tel qu'il fonctionne sur une carte FPGA. Avec Chisel, l'utilisatrice ou l'utilisateur décrit en Scala le processus de compilation (exécuté sur un ordinateur classique) pour obtenir l'arbre de syntaxe abstraite d'une description de matériel. À l'inverse, en Eclat, les traits de programmation généralistes (comme la récursion terminale et les accès mémoire) sont implantés directement sur le FPGA.

Il a été montré [88] que Chisel est suffisamment expressif pour encoder des constructions de programmation de SAFL, Bluespec et Handel-C ; il suffit pour cela d'implémenter de nouveaux générateurs de code en Scala ciblant Chisel ; cela permet même de mélanger différents modèles de programmation dans un circuit Chisel. Mais cette approche est limitée dans la mesure où les utilisatrices ou utilisateurs doivent faire un effort important pour réimplanter des compilateurs existants en Scala [4] et sont dès lors responsables des *bugs* dans ces compi-

4. Le compilateur Bluespec [5], par exemple, comporte plus de cent mille lignes de code Haskell ; réécrire

lateurs. À l'inverse, la montée en abstraction, en Eclat, ne repose pas sur la génération de code, mais plutôt par le développement de bibliothèques d'exécution et de machine virtuelle. Cela permet de mélanger les modèles de programmation dans un même programme (comme proposé en Chisel [88])

8.4 Langage de conception d'accélérateurs

Le concept de *langage de conception d'accélérateurs* (*accelerator design language, ADL*) a été introduit récemment [117] pour qualifier des DSL spécifiquement adaptés au développement d'accélérateurs de calculs, dans le but d'augmenter voire remplacer les langages de HLS classiques [49]. Les auteurs de ces nouveaux langages partagent le constat que les langages d'entrées des outils HLS constituent un mélange *adhoc* de logiciel et de matériel qui n'est pas adapté pour la conception d'accélérateurs optimisés [117]. En effet, les utilisatrices et utilisateurs de ces outils doivent reformuler en profondeur leur programme source (typiquement écrit en C), dans un style qui n'est pas idiomatique [72], tout en utilisant des combinaisons de *pragmas*, connues par l'usage et l'expérience de l'outil HLS, pour obtenir une architecture matérielle bien connue dès le départ, un pipeline par exemple.

Les langages de conception d'accélérateurs incorporent des constructions de programmation dédiées, pour exprimer de façon précise les caractéristiques de l'architecture à engendrer. En ce sens, ils suivent une approche similaire à celle des langages de construction de matériel au niveau RT (comme Chisel).

Spatial [117] est un DSL embarqué en Scala, qui intègre des structures de données de niveau architectural, telles que des FIFO, et des composants mémoires de différentes sortes (notamment BRAM et DRAM) explicitement instanciée dans le programme source. Le générateur de code de Spatial engendre une description Chisel.

Allo [49] est un dialecte MLIR (*Multi-Level Intermediate Language*) couplé à un *frontend* Python statiquement typé. Il offre des constructions de programmation et de métaprogrammation pour décrire, non seulement un calcul (dans un langage d'instructions), mais aussi des personnalisations sur ces calculs (comme du réordonnancement d'instructions, du déroulement de boucle et des transformations plus avancée), sur la mémoire (bufferisation) et sur les communications (insertion de FIFO).

Eclat est capable d'exprimer directement des structures de données comme des FIFOs ou des *buffers*, soit en réutilisant des structures de données existantes (n-uplets, vecteurs ou tableaux), soit via le mécanisme d'intéropérabilité entre Eclat et son langage d'implantation VHDL. Comparé à ces langages de conception d'accélérateurs, Eclat a un support bien plus limité pour la métaprogrammation (via les constructions `generate`, `parfor` et `vect_mapi`). Cependant, Eclat permet d'exprimer à la fois des calculs de plus-haut niveau (comme dans les approches HLS) ou de plus bas niveau (comme dans les langages de description de matériel).

ce compilateur dans un nouveau langage (Scala) ne nous paraît pas réaliste.

8.5 Un sous-ensemble d'Haskell pour la programmation au niveau RT

CλASH est un bibliothèque HASKELL [12, 150] qui propose un compilateur HASKELL alternatif ciblant VERILOG et VHDL. Cette approche se distingue des langages de description de matériel embarqués dans des langages généralistes (section 8.3) ; en effet, la description de circuit n'est plus une structure de données (représentant un arbre de syntaxe abstraite) mais plutôt un vrai programme HASKELL dont l'exécution (sur CPU) correspond au comportement du code engendré et synthétisé sur un FPGA. Chaque programme CλASH décrit la fonction de transition d'un circuit : c'est ce que nous appelons une fonction instantanée dans la terminologie d'ECLAT. CλASH supporte les fonctions d'ordre supérieur et la récursion générale à condition que ces constructions soient entièrement déroulées par évaluation partielle à la compilation. En ce sens, le modèle de programmation de CλASH se limite au niveau RT : les calculs non-instantanées doivent être encodés sous forme de machines à états (c'est-à-dire, une fonction combinatoire et un registre), comme on le ferait en μFP. Il est aussi possible d'encoder des calculs de cette manière en ECLAT (en utilisant la construction `reg`), mais ECLAT permet de programmer ces calculs avec des constructions de plus haut niveau comme `let/rec`, `if/then/else` et `(`$e_1 \| e_2$`)`.

ECLAT est plus expressif car il permet à la fois la description de matériel au niveau RT (sous forme de fonctions instantanées comme en CλASH ou LUSTRE) et la programmation de calculs longs compilés au niveau RT, avec une construction de programmation `exec` pour faire la jonction entre les deux mondes. En termes de performance, l'implantation optimisée (sans parallélisation) du problème des n reines en ECLAT a le même comportement temporel que la version CλASH d'origine [121], sur un FPGA de taille similaire et du même fabriquant. Le nombre d'éléments logiques (LE) utilisés par la version ECLAT est 2,6 fois plus petit que la version CλASH :

n	nombre de LE (CλASH)	nombre de LE (ECLAT)
8	2900	1091
12	5812	2238

Cela suggère qu'ECLAT est capable de décrire des circuits aussi précisément et efficacement que CλASH. Or CλASH peut uniquement décrire des circuits sous forme de fonctions instantanées et pas des calculs. Il est possible toutefois que les différences observées ici soient dues en partie à de meilleures optimisation dans le synthétiseur QUARTUS 22 que nous avons utilisé vis-à-vis de la version de QUARTUS (non précisé) utilisée pour CλASH en 2014.

8.6 Prise en compte de la cible FPGA et programmation généraliste

SAFL (*Statically Allocated Parallel Functional Language*) [146] est un langage fonctionnel-impératif à la ML [141] spécifiquement conçu pour être compilé sous forme de circuits. Le langage se limite à la récursion terminale, avec des structures de données statiquement allouées

et des fonctions de premier ordre [6]. Le parallélisme en SAFL est implicite (par exemple, les arguments de fonctions sont évalués en parallèle). Chaque fonction est compilée en un circuit (une ressource partagée) protégé par un arbitre [194] ce qui empêche d'évaluer deux appels à ladite fonction en parallèle ou de décrire des circuits combinatoires. SAFL est utile pour programmer certains types de circuits, orientés "calcul", mais il n'est pas adapté pour la description de matériel. Pour y remédier, une extension de SAFL appelée SAFL+ [193] étend le langage avec des canaux synchrones, ce qui permet à la fois d'encoder des algorithmes concurrents plus fins et d'interagir avec des entrées/sorties, mais pas de manière réactive. SAFL+ offre aussi des tableaux, qui sont implémentés sous forme de blocs de RAM. Finalement, SAFL interopère avec un langage de description de matériel appelé MAGMA [191], ce qui permet par exemple de définir des primitives combinatoires. Une analyse statique, effectuée sur une représentation intermédiaire du compilateur, permet de prédire dans certains cas le temps d'exécution d'une fonction SAFL (en nombre de cycles d'horloge) [192]. Enfin, une extension de SAFL a été proposée pour le traitement de flots (*stream processing*) [82] ; celle-ci s'appuie sur un système de types stratifié en deux niveaux : les types de base et les types de valeurs. La syntaxe des types de SAFL+ est $\sigma ::= T \mid T$ `stream` $\mid \sigma_1 \times \cdots \sigma_n$ où T désigne un type de base ; cela empêche la formation de flots de flots, préservant ainsi les contraintes de statique allocation de SAFL.

8.7 Sémantique temporelle pour un langage généraliste

HANDEL-C est un langage parallèle où la notion de temps joue un rôle crucial [131]. Ce langage est un sous-ensemble de C (sans pointeur ni récursion) et offre la construction `par` s_1 s_2 (inspiré d'OCCAM [155]) qui calcule s_1 et s_2 en parallèle de façon synchronisée (*lockstep*) avec un point de synchronisation sur la terminaison des deux instructions. Comme ECLAT, HANDEL-C est doté d'une sémantique temporelle intuitive [7] qui s'énonce comme suit :

> "*l'affectation et la construction* `delay` *prennent un cycle ;*
> *toutes les autres constructions sont instantanées*"

Par exemple, l'exécution de l'instruction ci-dessous prend deux cycles :

```
par {    // en parallèle : max(1,2) cycles
    {  x = 1; }        // 1 écriture : 1 cycle
    { a = 2; b = 3; }  // 2 écritures : 2 cycles
}
```

Les corps de boucle `for` et `while` doivent contenir au moins un `delay` ou une écriture après chaque itération. Par exemple, l'exécution de l'instruction suivante prend 6 cycles :

6. En SAFL (comme ECLAT), les fonctions ne sont pas des valeurs. Par exemple, elles ne peuvent pas être passées en paramètres d'autres fonctions. En ECLAT, ce sont des valeurs que l'on peut passer à d'autres fonctions mais que l'on ne peut pas retourner.

7. Une formalisation ultérieure, sous forme de sémantique opérationnelle, a été proposée [42].

```
x = 5;          // 1 écriture : 1 cycle

while (x > 0){  // répété 5 fois
  x = x - 1     // 1 écriture : 1 cycle
}
```

Ce choix dans la conception du langage abouti à un style de programmation similaire à celui d'ECLAT pour l'appel de fonctions récursives terminales. Cela rend la sémantique simple et prédictible [8].

Les communications en HANDEL-C se font au moyen de canaux asynchrones. Cela permet à la fois l'interaction *non-instantanée* avec l'environnement extérieur et le partage d'information entre les branches parallèles de l'instruction `par`.

HANDEL-C a toutefois des limitations en lien avec ses choix de conception de langage et d'implantation :

— Les corps des boucles `for` ou `while` doivent contenir au moins une écriture ou un délai. Cela vise à empêcher les boucles combinatoires instantanées, qui ne sont pas implantables matériellement et entrent en contradiction avec l'hypothèse de synchronisme du niveau RT. Le langage synchrone ESTEREL impose la même restriction [25]. ECLAT évite par construction les boucles combinatoires instantanées, dans sa manière d'implanter la récursion, car tout appel récursif introduit un délai. L'implantation de HANDEL-C autorise tout de même des programmes contenant des boucles combinatoires instantanées dans les boucles `for` et `while` : il insère alors automatiquement le délai (pour éliminer les dites boules) et produit un *warning* [38].

— Les écritures simultanées (*resp.* les envois de messages simultanés) d'une même variable (*resp.* sur un même canal) sont illégaux. Le manuel de référence du langage indique que les écritures et les envois de messages ne devraient jamais être concurrents, mais que si c'est le cas, les utilisatrices et utilisateurs doivent s'assurer que ces accès ou envois de messages ne se produisent jamais dans le même cycle [42]. Cela nous semble difficile à vérifier (sans rejeter trop de programmes, de par la présence de boucle `while` et de parallélisme d'instructions), ce qui pose un problème de sûreté du langage. ECLAT n'a pas ces problèmes, de pas sa nature déclarative, sa manière de gérer l'interaction avec une fonction instantanée et le mécanisme de verrou pour protéger les accès à la mémoire partagée.

— De par la sémantique d'HANDEL-C, l'affectation `x[2] = x[3] + 4` avec `x` une bloc `ram` requiert que la lecture `x[3]` soit effectuée dans le même cycle que l'affectation de `x[2]`. Or l'accès en lecture et en écriture dans un même cycle n'est pas implantable avec un tableau représenté dans un bloc RAM (à un seul port) car de façon générale, chaque accès à ce type de composant mémoire prend un cycle et doit être séquentialisé. Cette

8. Comme en ECLAT, le temps d'exécution des programmes n'est pas toujours décidable, mais la manière dont les calculs se déroulent dans le temps est prédictible.

instruction HANDEL-C est donc considérée comme illégale et un message d'erreur est lancé par le compilateur HANDEL-C. ECLAT choisit plutôt d'insérer une pause à chaque accès mémoire et de protéger chaque tableau avec un verrou, ce qui est plus expressif en terme de concurrence et évite de rejeter *trop* de programme après typage.

Ces limitations suggèrent que la sémantique HANDEL-C, bien qu'appréciable pour sa simplicité et son caractère systématique, rend de nombreux programmes illégaux pour des questions d'implantation. ECLAT au contraire, propose un équilibre entre simplicité de la sémantique et difficulté d'implantation. Cet équilibre vient essentiellement du caractère déclaratif du langage, sa gestion de l'interaction par une fonction instantanée, la retard de un cycle induit par chaque accès mémoire et le verrou séquentialisant les accès à chaque tableau.

8.8 Implantation de langages synchrones sur FPGA

Les langages synchrones partagent des techniques issues des premiers HDL fonctionnels [162] et ont constitué une alternative à la synthèse de haut niveau [71].

Des langages synchrones comme ESTEREL [25], LUSTRE [175] et SIGNAL [5] ont été implantés sous forme de circuits synchrones pour reconfigurer des FPGA. La principale caractéristique de ces implantations, partagée avec ECLAT, est la correspondance faite entre temps logique du modèle synchrone et *tick* d'horloge du FPGA cible. Cela rend les langages synchrones suffisamment précis pour décrire des circuits.

La compilation la plus directe d'un langage synchrone en matériel est celle de LUSTRE, qui produit une *net-list* en associant à chaque opérateur logique du langage une porte logique et à chaque opérateur `pre` de LUSTRE un registre [175].

Les langages synchrones se modélisent classiquement comme des machines de Mealy [35] qui produisent des sorties instantanément en fonction des entrées et ont un état interne mis à jour à chaque cycle.

Les programmes synchrones flot de données (à la LUSTRE [93]) s'apparentent à des *netlists* ; ils se compilent naturellement en circuits synchrones, sur FPGA, en faisant correspondre le temps logique du langage synchrone avec les *ticks* d'horloge du FPGA cible.

Les langages synchrones orientés contrôle (comme ESTEREL [25] et QUARTZ [182]) sont plus difficiles à compiler en circuit mais sont très expressifs, comparés à des langages de description de matériel [94].

Le langage SCADE 6 [58] est un dialecte de LUSTRE pour la conception de logiciel certifié. Il intègre des structures de contrôle dans un style flot de données [98], et en particulier des automates de modes inspirés du langage ARGOS [135]. SCADE 6 a été implanté sur FPGA dans le cadre du projet GENCOD [156] : une première approche a consisté à compiler en VHDL le code C certifié produit par le générateur de code KCG à partir de programme SCADE 6 ; une seconde approche a consisté à compiler le langage HEPTAGON, proche de SCADE 6, directement en VHDL [90].

Un dialecte d'ESTEREL appelée ESTEREL LIGHT a été récemment utilisé pour implanter des

architectures de réseaux neuronaux à impulsions (*Spiking Neural Networks* sur FPGA, par génération de code VHDL [170]. L'approche synchrone a été motivée par la proximité entre temps logique et évolution des réseaux neuronaux à impulsions en fonction de leur environnement.

Une variante de cette approche a été récemment proposée en vue du développement d'applications embarquées spatiales [222]. Elle consiste à compiler un sous-ensemble de LUSTRE vers l'outil RETICLE [217]. RETICLE est un langage d'assemblage qui décrit la structure de la *net-list* par une suite d'équations (appelées *affectations*). Le langage est présenté comme une *machine virtuelle*, car il permet de représenter de façon portable les blocs DSP et les blocs RAM (BRAM), voire de nouveaux types de blocs qui apparaissent dans les FPGA modernes mais sont difficilement exploitables de façon portable dans les HDL classiques. En termes de description structurelle de circuit, la précision est la même que dans les HDL fonctionnels. L'expressivité de RETICLE est très limitée, mais l'outil à l'avantage de contourner le synthétiseur logique en s'interfaçant directement avec les outils de *placement/routage*. Cela améliore la traçabilité du code, qui est requise par les autorités de certification dans l'industrie aéronautique [222].

8.9 Conception et implantation de squelettes de parallélisme

Dans de nombreux *frameworks* de calcul parallèle, le comportement temporel des programmes est spécifié de façon informelle [9] ; Dans [4], Aldinucci et Danelutto on proposé une sémantique opérationnelle pour modéliser formellement le comportement parallèle et fonctionnel des bibliothèques de squelettes de parallélisme. Le cadre sémantique proposé est basé sur un système de transitions étiquetées qui peut être utilisé pour prouver des propriétés sur les programmes.

La sémantique opérationnelle d'ECLAT est d'avantage orientée vers la programmation : elle vise à expliciter le passage du temps dans les programmes concurrents dans le but d'améliorer la prédictibilité des résultats et des performances. Le modèle synchrone sur lequel repose ECLAT, réconciliant parallélisme et déterminisme [86], nous a permis de développer des squelettes de parallélisme avec mémoire partagée (section 7.8).

Des travaux récents tels que FPX [154] se sont attaqués à la difficulté de produire un code efficace en termes de temps et d'espace pour les architectures FPGA impliquant un processeur hôte (*hardcore*) à partir de code logiciel parallèle. FPX est un *framework* permettant d'accélérer des traitements flot de données (Data Stream Processing, *DSP*) généralistes. Il fournit un DSL paramétrable, intégré à PYTHON, dans lequel le programmeur d'application peut définir des opérateurs flot de données sous forme de programmes séquentiels, et les composer pour former l'application DSP (c'est-à-dire un graphe d'opérateurs de flot de données). Le programmeur n'est plus responsable des détails bas niveau de l'implémentation, laquelle est obtenue par génération de code (en utilisant OPENCL comme représentation intermédiaire) combinée à

9. Voici un exemple de description informelle pour une *framework* donné : "Le squelette `farm` utilise un ensemble d'éléments de calcul indépendants pour calculer les tâches d'entrée. Chaque fois qu'une tâche en entrée est disponible, une ressource est sélectionnée pour l'exécution de cette tâche, en utilisant telle ou telle politique d'ordonnancement" [4].

une bibliothèque d'exécution qui fournit des opérateurs flot de données efficaces.

Les *frameworks* de parallélisme de haut niveau comme WindFlow [140], sont généralement basés sur une architecture hétérogène comportant un processeur hôte (*hardcore*) et des *noyaux de calculs* distribués sur des unités de calcul telles que des GPU, des cœurs de processeurs ou des FPGA. Dans ce contexte, Eclat pourrait être utilisé à la place d'un outils HLS (comme Vitis HLS) pour exprimer et composer des *noyaux de calculs* avec une latence prédictible. La version actuelle d'Eclat a cependant deux limitations qui freinent son usage pour la conception d'accélérateurs de calcul : l'absence de support primitif pour le calcul flottant et pour l'accès à la mémoire externe (SDRAM) depuis le FPGA cible.

8.10 Synthèse des travaux connexes

Comme présenté aux sections 1.3.1 et 1.3.5, les HDL fonctionnels d'une part et les outils HLS d'autre part ont été proposés pour répondre aux difficultés de programmation au niveau RT. Ces deux approches, bien distinctes, ont des avantages et inconvénients qui ont été discutés aux sections 8.1 et 8.2.

Les HDL fonctionnels [48] décrivent les circuits sous forme de *fonctions instantanées* exploitant le parallélisme à grain fin disponible sur le FPGA cible. Les outils HLS [147] servent à programmer des *calculs généralistes* et à les compiler au niveau RT. Eclat unifie ces deux approches complémentaires dans un modèle synchrone.

Les travaux connexes qui ont été présentés dans ce chapitre proposent d'autres manières de concilier programmation généraliste et description de circuits.

Les DSL de description de matériel, comme Chisel [13] et Lava [32], sont embarqués dans un langage généraliste comme Scala ou Haskell : l'exécution d'un programme sur CPU crée une structure de données représentant un circuit. Cela rend possible diverses formes de métaprogrammation et de génération de code, y compris pour la simulation et la vérification de ce circuit.

Les DSL de conception d'accélérateurs comme Allo [49] et Spatial [117] mettent en œuvre des techniques similaires pour la programmation de calculs. Ils explicitent précisément, dans les programmes sources, certains détails architecturaux (comme les *buffers et les FIFO*) voire même des transformations de programmes à appliquer (comme par exemple le déroulage de boucles et le réordonnancement d'instructions) dans le but d'engendrer une réalisation matérielle efficace.

Le langage Cλash [12] utilise un sous-ensemble d'un langage généraliste (Haskell) compilé au niveau RT pour décrire des circuits ; cette solution est similaire à celle proposée par Eclat, mais se limite à l'expression de fonctions instantanées dans la lignée des HDL fonctionnels. Eclat, au contraire, sert à décrire à la fois des fonctions instantanées et des calculs.

SAFL [88] est un langage de programmation fonctionnel parallèle compilé au niveau RT ; il se base sur l'idée qu'il est nécessaire de restreindre l'expressivité du langage pour que le code engendré soit efficace et prédictible. Nous avons repris cette philosophie pour la conception et l'implantation d'Eclat, quitte à ensuite développer des bibliothèques d'exécution, directement

en ECLAT, pour supporter des traits de programmation plus avancés, comme dans bibliothèque d'exécution OCAML écrite en ECLAT (section 7.9).

HANDEL-C [131] est un langage de programmation parallèle-impératif avec une sémantique précise au niveau cycle (*cycle accurate*). Cette sémantique présente des similarités avec celle d'ECLAT ; cependant, le style impératif de HANDEL-C limite la composabilité des circuits qu'il engendre et cause des incompatibilités entre la sémantique du langage et les contraintes d'implantation. ECLAT est plus expressif que HANDEL-C ou SAFL car il ne requiert pas de mécanisme asynchrone pour communiquer avec l'environment.

La syntaxe et la sémantique d'ECLAT propose un mélange entre la programmation fonctionnelle généraliste *à la SAFL* (appel par valeur avec parallélisme explicite) et la programmation synchrone à la LUSTRE [162]. C'est ce mélange qui permet de décrire à la fois des calculs et l'interaction avec l'environnement extérieur.

Les langages synchrones [21] ont été implantés sur des FPGA dans les années 1990, mais ces travaux, à notre connaissance, ont été peu suivis. On peut toutefois constater, récemment, un regain d'intérêt dans cette thématique pour le développement d'applications médicales [170] et spatiales [222].

ECLAT promeut une approche synchrone pour la programmation généraliste de FPGA : le mélange de calcul et d'interaction en ECLAT constitue une réponse aux besoins des applications embarquées récentes, réalisant des calculs sur de grosses données et interagissant avec des capteurs et des actionneurs voire avec d'autres applications.

9 Conclusion et travaux futurs

Le travail présenté dans ce manuscrit vise à montrer qu'il est possible d'offrir des abstractions de haut niveau – sûres, expressives et efficaces – pour la programmation de circuits FPGA.

Les FPGA interagissent naturellement avec leur environnement physique : ils prennent en compte des entrées et produisent des sorties à chaque cycle d'horloge à des fréquences de l'ordre de 100 MHz (soit 10 nanosecondes par cycle).

Les langages de description de matériel servent à concevoir des circuits au *niveau transfert de registres* (niveau RT) : un modèle de programmation qui spécifie précisément les transferts de données entre registres, au sein des circuits, à chaque *tick* d'horloge physique. Cependant, la programmation d'algorithmes généralistes (ou "calculs") dans les langages de description de matériel classiques nécessite des encodages fastidieux et source d'erreurs.

Pour répondre à cette tension entre programmation de haut niveau et description précise au niveau RT, nous avons développé Eclat, un langage de programmation parallèle synchrone. Eclat unifie deux approches de l'état de l'art en programmation de FPGA : la description de matériel en style fonctionnel (orientée flot de données) et la synthèse de haut niveau (orientée flot de contrôle), cela dans le but de programmer à la fois l'*interaction* avec l'environnement physique et des *calculs*.

Eclat offre des constructions de programmation généralistes *à la OCaml* avec du parallélisme réel, de la mémoire partagée, de la concurrence déterministe équitable et des performances souvent prédictibles. Chaque programme Eclat est une suite de déclarations de fonctions. Chaque fonction est un "calcul" qui, étant donnée une entrée, retourne une sortie en zéro, un ou plusieurs cycles d'horloge. Le temps d'exécution d'une fonction est déterministe mais dépend en général de la valeur de son argument (l'entrée) et parfois aussi du contexte [1] d'appel. Les fonctions qui retournent toujours une sortie en zéro cycle (quels que soient leur entrée ou leur contexte d'appel) sont dites *instantanées* ; le caractère instantané d'une fonction est vérifié par typage statique : un programme Eclat bien typé est un *programme réactif* défini par une fonction instantanée, laquelle s'applique point à point sur un flot d'entrées et retournent instantanément un flot de sorties. La construction `reg` d'Eclat permet de définir des fonctions instantanées avec état : la sortie courante et l'état suivant dépendent de l'entrée et de l'état courants (c'est une machine de Mealy). La construction `exec` d'Eclat sert à plonger

1. Cela arrive en présence de ressources partagées.

des calculs arbitrairement longs dans le programme réactif en déroulant dans le temps leurs exécutions.

Les calculs en ECLAT peuvent exploiter le parallélisme à grain fin disponible au niveau circuit, y compris en présence de mémoire partagée. Cela compense le fait que les fréquences d'horloge des FPGA sont de l'ordre de 50 fois plus faibles que celles des processeurs modernes. Ainsi, ECLAT est adapté pour programmer des accélérateurs matériels.

La sémantique synchrone d'ECLAT est inspirée, d'une part, de la sémantique comportementale d'ESTEREL, et d'autre part, des sémantiques à réductions pour les langages fonctionnels impératifs. Le modèle de programmation résultant s'apparente au modèle synchrone flot de données avec un temps logique dérivé de l'horloge physique. Le système de types d'ECLAT garantit la réactivité des programmes. Il vise de plus à être *sûr* vis-à-vis de la sémantique :

— l'évaluation des programmes préserve les types (*subject reduction*)

— et l'évaluation des programmes progresse toujours jusqu'à retourner une valeur (*progrès*).

La sûreté du typage reste à démontrer mais a été pensée, dès le départ, comme une propriété attendue du langage. On peut noter qu'ECLAT repose sur un noyau de langage bien connu, avec peu de constructions de programmation, ce qui facilite la formalisation. Les points difficiles restant, du point de vue de la sûreté de fonctionnement, viennent d'extensions propres à ECLAT :

— le polymorphisme paramétrique sur les tailles des données (entiers, vecteurs et tableaux),

— le langage de types à deux niveaux (*types* et *types de base*) autorisant une forme limitée d'ordre supérieur

— et enfin une fonction de substitution non-standard, est mise en œuvre dans la sémantique pour modéliser l'instanciation des blocs `reg` et `exec` à l'exécution (section 4.3).

Le compilateur ECLAT propose une chaîne complète et automatisée de bout en bout, allant du programme source ECLAT au fichier de configuration d'un FPGA, avec aussi une infrastructure de simulation au niveau RT utilisant le code VHDL engendré.

Les choix de conception d'ECLAT sont guidés par des choix d'implantation, qui sont fondamentaux pour que les circuits engendrés soient précis, efficaces et prédictibles. Le modèle de programmation d'ECLAT simplifie les raisonnements sur le comportement temporel des programmes, en contrepartie de difficultés d'implantation laissées à l'implémenteur.

Cela a donné lieu à la définition de MINI-HDL, qui est un langage intermédiaire à mi-chemin entre ECLAT et VHDL. MINI-HDL est un langage de description de matériel impératif (comme VHDL), flot de données et avec des automates. Il a été spécialement conçu pour faciliter la conception de circuits et la génération de code ECLAT au niveau RT.

Nous avons proposé une technique de compilation pour traduire un langage parallèle généraliste (ECLAT) en un automate hiérarchique (*i.e.,* un programme exprimé en MINI-HDL) avec partage des appels de fonctions, synchronisation instantanée des calculs et implantation efficace du parallélisme (par juxtaposition de circuits en style flot de données). Le programme MINI-HDL est alors compilé en VHDL.

Le chaîne de compilation d'Eclat cible principalement les FPGA du fabricant Intel (ex-Altera) ; des portages sont en cours vers des FPGA Xilinx (via le synthétiseur Vivado) et du matériel libre (via la chaîne d'outil Yosys/next-prn [190]).

Nous avons montré au chapitre 7 que le modèle de programmation d'Eclat et sa chaîne de compilation vers VHDL s'avèrent bien adaptés pour le développement d'abstractions de programmation (comme des squelettes de parallélisme et des machines virtuelles). Cela a permis de programmer *facilement* des applications réalistes connectées aux entrées/sorties physiques du FPGA cible.

Enseignement

Le travail sur Eclat a été utilisé dans le cadre d'un cours de compilation donné aux étudiants de première année de Master *Science et Technologie du logiciel (STL)* à Sorbonne Université. Le langage Mini-HDL (présenté section 5.5) s'est avéré être un bon langage support pour introduire rapidement, par l'exemple, des notions de conception de matériel telles que la différence entre *partage* et *duplication*, le modèle d'exécution synchrone, le chemin critique et le compromis temps-espace, y compris sur des *exercices* réalistes comme l'implantation et l'optimisation d'un processeur simplifié. Comme les programmes Mini-HDL représentent intrinsèquement des machines de Mealy, nous faisons abstraction des aspects les plus complexes de la conception de circuit au niveau RT. Le langage est du reste beaucoup moins verbeux que VHDL, et nous pensons qu'il est plus accessible que VHDL pour une audience de spécialistes en programmation et en algorithmique, sans prérequis en électronique.

Nous avons ensuite présenté, en cours, la compilation d'Eclat, ce qui a permis d'illustrer des notions de transformations de programmes, de génération de code et d'optimisation.

La dernière partie du cours a porté sur le développement de machines virtuelles en Eclat. Nous avons alors proposé un projet de programmation consistant à programmer en Eclat un interprète de bytecode pour le langage MicroJS [2]. Le temps d'exécution déterministe et le parallélisme explicite d'Eclat ont constitué des critères qualitatifs pour encourager les étudiants à optimiser leurs code et réaliser des analyses expérimentales en utilisant un simulateur au niveau RT (GHDL). Nous comptons poursuivre ces expériences pédagogiques en conception de circuit et en implantation de langages de programmation. Cela pourrait donner lieu à un projet *de la puce au WEB*, s'inspirant du projet *Nand to Tetris* [183], avec l'opportunité de laisser les étudiants manipuler de véritables cartes FPGA, interagissant avec un port ethernet, et sur lesquelles dériver tout un système d'information.

2. MicroJS est un langage *à la Scheme* avec une syntaxe à la Javascript qui a été utilisé dans plusieurs cours de compilation à Sorbonne Université.

Travaux futurs

Nous voyons plusieurs suites possibles à ce travail, lesquelles peuvent être orientées en trois axes de recherche :

1. prouver et tester des propriétés sur la sémantique d'ECLAT ;
2. mettre en œuvre des techniques de vérification en ECLAT ;
3. accompagner le passage à l'échelle et la dissémination du langage.

Sûreté du typage et sémantique exécutable

La distribution d'ECLAT (annexe C) comprend d'une part, un compilateur, et d'autre part, un interprète qui implante la sémantique opérationnelle présentée aux chapitres 4 et 6. Une alternative intéressante serait d'implanter la sémantique d'ECLAT dans l'outil PLT REDEX [76], qui est un DSL reconnu pour le développement de sémantiques de langages de programmation. Ce DSL a notamment été utilisé pour modéliser le langage synchrone ESTEREL [80].

PLT REDEX repose sur un style de sémantique opérationnelle *à réduction avec contextes d'évaluation* [77]. Nous avons développé une sémantique opérationnelle alternative pour ECLAT (sur papier) dans ce style. Une partie de cette sémantique (qui ne traite pas les constructions `exec` et `reg`) a été présentée au symposium HLPP 24 (*High-Level Programming and Applications*) [204]. La version complète de cette sémantique est donnée en Annexe, section B.4.

L'équivalence entre cette sémantique et la sémantique comportementale présentée au chapitre 4 reste à démontrer. L'intérêt d'avoir deux sémantiques équivalentes (l'une comportementale et l'autre par contextes d'évaluation) est de pouvoir garantir des propriétés différentes sur les programmes. En particulier, la sémantique à réductions sera plus adaptée pour prouver *syntaxiquement* la sûreté du typage statique [224], car elle comporte moins de règles d'inférence à considérer et moins de prémisses par règles. En contrepartie, cette sémantique nous semble moins adaptée pour exprimer des propriétés temporelles sur les programmes.

Vérification

Comme la sémantique d'ECLAT spécifie précisément le comportement fonctionnel et temporel des programmes, nous pouvons envisager d'appliquer à ECLAT des techniques de vérifications formelles. Ce serait utile, par exemple, pour des programmes ECLAT réalisant des algorithmes concurrents via du parallélisme explicite et des communications par mémoire partagée (comme les implantations de squelettes de parallélisme présentées à la section 7.8). Rappelons que les programmes ECLATsont déterministes, mais cela n'exclut pas certaines erreurs de programmation telles que des interblocages (*dead lock*) lorsque la mémoire partagée est utilisée pour encoder des synchronisations entre calculs parallèles.

Nous pouvons déjà programmer en ECLAT des observateurs synchrones pour exprimer des propriétés sur les programmes (cf. sous-section 3.5.4 page 50) ; par exemple, exprimer que la durée d'un calcul sous un `exec` (c'est-à-dire la durée entre deux passages à un de la sortie `rdy` associée) est inférieur à n cycles pour un n donné. L'enjeu est alors de vérifier ces propriétés.

Nous pourrions, pour cela, appliquer des techniques existantes comme l'interprétation abstraite [107], la vérification déductive de programmes [78] ou la vérification de modèles (*model checking*) [33], sur du code engendré à partir d'Eclat. Nous développons actuellement un générateur de code OCaml passant par la représentation "circuit" (Mini-HDL) du compilateur Eclat (cf. section C.9 en Annexe). C'est utile pour simuler les programmes Eclat (y compris avec des trais de programmation parallèles et réactifs) sans dépendre d'un environnement de simulation VHDL. Ce code OCaml engendré pourrait être vérifié par interprétation abstraite avec Mopsa, un *framework* d'analyse statique par interprétation abstraite qui est modulaire et propose des domaines abstraits réutilisables pour l'analyse relationnelle multi-langages, y compris pour OCaml [214]. Cela peut apporter une analyse de valeurs, utile pour détecter de possibles erreurs d'exécution (telles que des indices hors des bornes de tableaux) et nous pensons que de telles analyses seraient aussi utilisables pour vérifier les propriétés temporelles (exprimées à nouveau sous forme d'observateurs synchrones).

Une autre voie serait de générer du code WhyML et de les vérifier à l'aide de la plateforme vérification déductive Why3 [79]. Il faudrait pour cela inférer des contrats à partir d'assertions dans les programmes Eclat. De plus, il sera crucial de rendre modulaire le schéma de compilation d'Eclat pour faciliter les raisonnements sur le code engendré ; nous pourrons pour cela nous inspirer de la compilation des appels de fonctions externes d'Eclat (*i.e.*, la génération d'un circuit pour chaque fonction externe définie en Eclat), mais au niveau des fonctions globales d'un même programme, ou par l'introduction d'un système de modules.

Enfin, le langage intermédiaire Mini-HDL utilisé par le compilateur Eclat est synchrone flot de données ; cela pourrait donner lieu à un voie de compilation d'Eclat vers Lustre, pour être en mesure de générer du code C embarqué (sur lequel calculer un WCET) et vérifier des propriétés en utilisant des outils de vérification existants pour Lustre [20] et Scade [101]. Il pourrait s'agir de propriétés temporelles à vérifier avec Lesar [172], qui est un *model checker* pour Lustre. Eclat pourrait aussi utiliser Lutin [173] pour la spécification de scénarios réactifs, ou bien s'en inspirer. Lutin est un langage flot de données avec des structures de contrôle ; il offre des primitives spécialement prévues pour la simulation de programmes, comme par exemple une construction de choix non-déterministe et des générateurs de valeurs aléatoires.

Programmation d'applications et passage à l'échelle

Le prototype de compilateur Eclat [3] est suffisamment stable pour envisager la programmation d'applications plus complexes que celles décrites dans cette thèse. Cela permettrait de montrer l'effectivité de l'approche par rapport à des méthodologies de développement bien établies comme la description de matériel en VHDL ou Verilog, l'usage d'outils de synthèse de haut niveau basés sur C et le calcul parallèle à base d'architectures hétérogènes. Eclat dans ce contexte a principalement deux limitations.

Premièrement, il n'a pas de support prédéfini pour le calcul flottant. Nous pourrions toutefois utiliser le mécanisme de déclaration de fonctions externes d'Eclat, pour appeler des

3. disponible à l'URL `https://github.com/lsylvestre/eclat`

circuits (IP) de calcul flottant. Ces fonctions externes en ECLAT peuvent être multi-cycles et elle peuvent être partagées (via des verrous implicites) pour réduire la taille du circuit engendré.

Deuxièmement, les cartes FPGA ont généralement une mémoire externe (*Synchronous Dynamic Random-Access Memory, SDRAM*) dont la capacité de stockage est bien plus importante que l'ensemble des blocs RAM (BRAM) disponibles à l'intérieur du circuit [4] FPGA. Toutefois ECLAT, n'a pas encore de support prédéfini pour accéder à cette mémoire externe. Nous cherchons actuellement à utiliser (en tant que fonction externe ECLAT) un contrôleur SDRAM existant [125] conçu pour les systèmes embarqués temps réel. Un tel contrôleur devrait pouvoir être encapsulé sous de nouvelles constructions de lecture et d'écriture non-instantanées pour ECLAT, implicitement séquentialisées par un verrou global à la manière du mécanisme introduit au chapitre 6. Cela conduirait à une hiérarchie mémoire à trois niveaux : *vecteurs* (implantés avec des cellules logiques), *tableaux* (en mémoire BRAM) et *mémoire principale* (SDRAM). Une difficulté sera de partager les accès à la mémoire externe, depuis plusieurs programmes ECLAT, contrairement aux tableaux ECLAT qui sont locaux à chaque programme [5].

Notre implantation de la machine virtuelle OCAML pourrait être étendue afin d'offrir un environnement de programmation logicielle prédictible sur FPGA. Il s'agirait d'ajouter un support pour les nombres flottants et la couche objets d'OCAML. Nous travaillons aussi à placer statiquement le bytecode en mémoire Flash pour ne pas devoir resynthétiser la machine à chaque nouveau programme bytecode à exécuter. Nous cherchons aussi à normaliser le plus possible le jeu d'instructions de bytecode (en fusionnant des instructions comme par exemple `MAKE_BLOCK` et `MAKE_CLOSURE`) pour réduire la taille du circuit implantant l'interprète de bytecode. Des modèles de parallélisme, comme des squelettes de parallélisme, pourraient aussi être développés pour manipuler les structures de données dynamiques allouées par la bibliothèque d'exécution OCAML ; le pire temps d'exécution des instructions de bytecode (ou au moins, un sous-ensemble) pourrait être estimé depuis le programme source ECLAT implantant la machine virtuelle en se basant sur la sémantique synchrone du langage et dans la lignée des travaux sur OMICROB [215]. Une évaluation expérimentale approfondie sera toutefois nécessaire pour déterminer si l'approche machine virtuelle est avantageuse et suffisamment efficace pour programmer des applications logicielles embarquées sur FPGA. Il s'agirait de comparer cette approche avec, d'un part, des applications directement programmées en ECLAT, et d'autre part, des approches purement logicielles basées sur des processeurs *softcore*.

Le style de programmation d'ECLAT combine des traits fonctionnels-impératifs d'OCAML et un modèle d'exécution directement inspiré des langages synchrones. Le langage s'adresse donc naturellement aux utilisatrices et utilisateurs de ces deux communautés.

4. Le circuit FPGA et la mémoire externes sont tous deux placés sur la carte FPGA.

5. Les tableaux sont implémentés avec des blocs RAM et dupliqués à chaque instantiation du programme, *i.e.*, à chaque appel du programme en tant que fonction externe depuis un autre programme. Cela n'est pas possible avec la mémoire externe.

À plus long terme, nous pourrions supporter un sous-ensemble plus large d'OCaml dans le langage d'expressions d'Eclat [6] ainsi que des traits de programmation synchrone plus avancés tels que des structures de contrôle [160], des combinateurs de tableaux [59] et des primitives de concurrence orientées flot de contrôle [6]. Cela s'inscrira dans l'approche initiée dans cette thèse pour Eclat : mélanger *calcul* expressif orienté flot de contrôle et *interaction* précise orientée flot de données et ainsi unifier programmation parallèle généraliste d'une part, et programmation réactive d'autre part, avec des applications à l'interface entre le matériel et le logiciel.

6. L'annexe B donne une définition d'Eclat étendu avec des types sommes non-récursifs et une construction de filtrage `match/with`.

A Synthèse d'Eclat sur le FPGA Intel Max10

L'implantation Eclat de la machine virtuelle OCaml et de sa bibliothèque d'exécution (voir section 7.9) est personnalisée, ici :

- des entiers sur 31 bits + 1 bit de marque ;
- l'espace mémoire a une taille de 16 384 mots de 32 bits pour implanter la pile, le tas et le segment des données globales
- avec GC Stop&Copy écrit en Eclat
- le jeu d'instructions de la machine comporte 133 instructions implantées (sur 148), ayant de 0 à 3 arguments :
  - pour exécuter une instruction, nous lisons son code opération (*opcode*) dans le table de *bytecode* ; nous testons si c'est une opération sans argument (premier niveau de `match/with`), si oui, elle est exécutée ;
  - sinon, nous lisons un premier argument dans le tableau de *bytecode* ; nous testons si c'est une opération à un argument (deuxième niveau de `match/with`), si oui, elle est exécutée ;
  - sinon, nous lisons un deuxième argument dans le tableau de *bytecode* ; nous testons si c'est une opération à deux arguments (troisième niveau de `match/with`), si oui, elle est exécutée ;
  - sinon, nous lisons un troisième argument dans le tableau de *bytecode* ; nous testons si c'est une opération à trois arguments (quatrième niveau de `match/with`), si oui, elle est exécutée ;
  - sinon, c'est une erreur (simulée par une boucle infinie dans le circuit)

Voici les caractéristiques du circuit engendré :

- **36 812 éléments logiques (LE) dont 11 951 registres 1-bit.**
- **525 373 bits de mémoire RAM et 8 multiplieurs**
- **Fréquence max : 54 MHz**

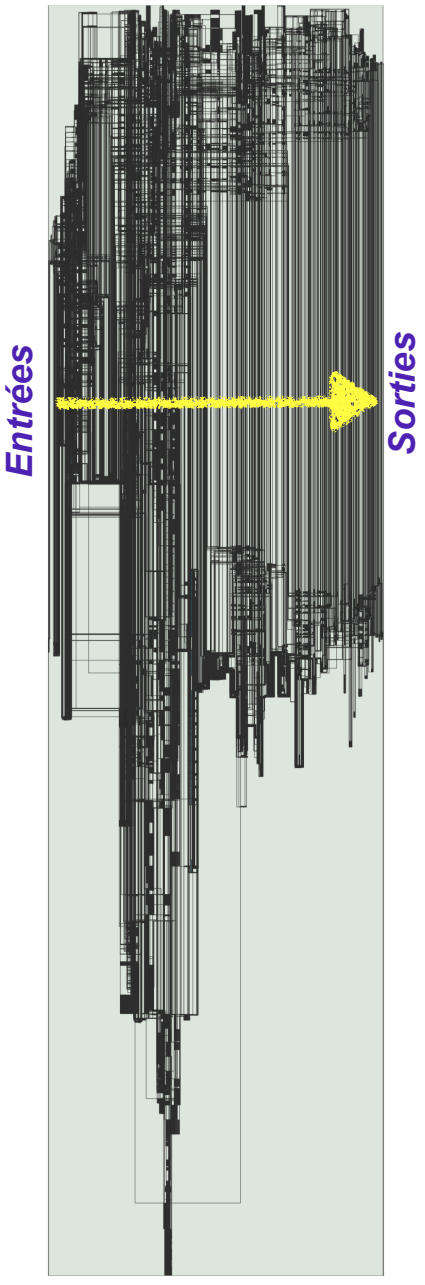


Figure A.1 Implantation d'OCaml sur FPGA avec Eclat

Nous partons du programme source suivant (figure 3.8) :

```
1  let fibonacci n =
2    let rec fib (i, a, b) =
3      if i = 0 then a
4      else fib(i - 1, b, a + b)
5    in fib(n, 0, 1) ;;
6
7  let main (n:int<12>) : int<58> =
8    if n < 0 then 0 else
9    let (o,rdy) = exec
10                   fibonacci(n)
11                 default 42
12   in o ;;
```

Configuration par défaut pour programmer le FPGA Max10 :

— 12 bits connectés aux 10 interrupteurs et 2 boutons ;

— 58 bits connectés aux 10 LED et afficheurs 7 segments.

Voici les caractéristiques du circuit engendré :

— **319 éléments logiques (LE) dont 189 registres d'un bit.**

— **Fréquence max : 143 MHz**

— chaque opérateur ECLAT correspond à une fonction combinatoire dans le circuit. La soustraction est implanté avec un additionneur.

— Pour empêcher la mise à jour des registres lorsque $n < 0$, le synthétiseur utilise des registres avec une entrée spéciale ENA (pour *enable*). C'est un choix d'implantation différent de celui que nous avions fait dans le schéma de la figure 5.6 (qui utilisait un multiplexeur en entrée d'un registre simple) ; mais l'effet est le même.

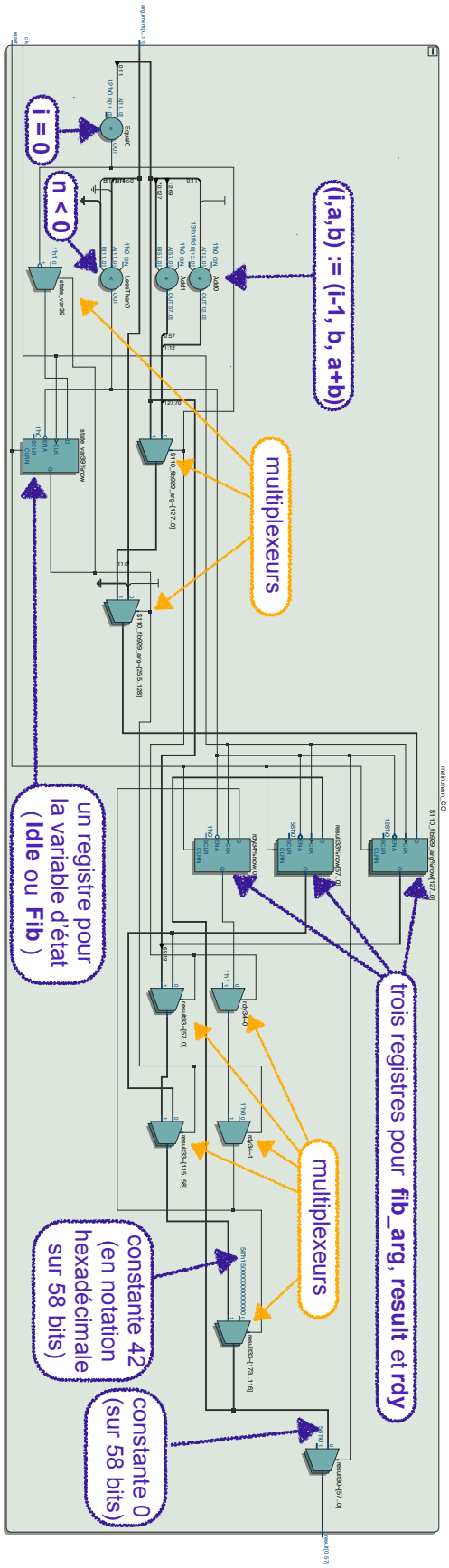


FIGURE A.2 Mélange de calcul et d'interaction

```
1  type t = int<32> ;;
2
3  let fibonacci n =
4    let rec fib ((i, a, b) : t * t * t) =
5      if i = 0 then a
6      else fib(i - 1, b, a + b)
7    in fib(n, 0, 1) ;;
8
9  let main (i:int<12>) : int<58>  =
10   let n = resize_int<32>(i) in
11   let (o,rdy) = exec
12                   let x = fibonacci(n) in
13                   fibonacci(x)
14                 default 42
15   in resize_int<58>(o) ;;
```

Voici les caractéristiques du circuit engendré :

- **200 éléments logiques (LE) dont 133 registres d'un bit.**
- **Fréquence max : 155 MHz**
- les deux appels direct à `fib` sont partagés ; la variable `fib_id` mémorise le numéro (ici 0 ou 1) du site d'appel.

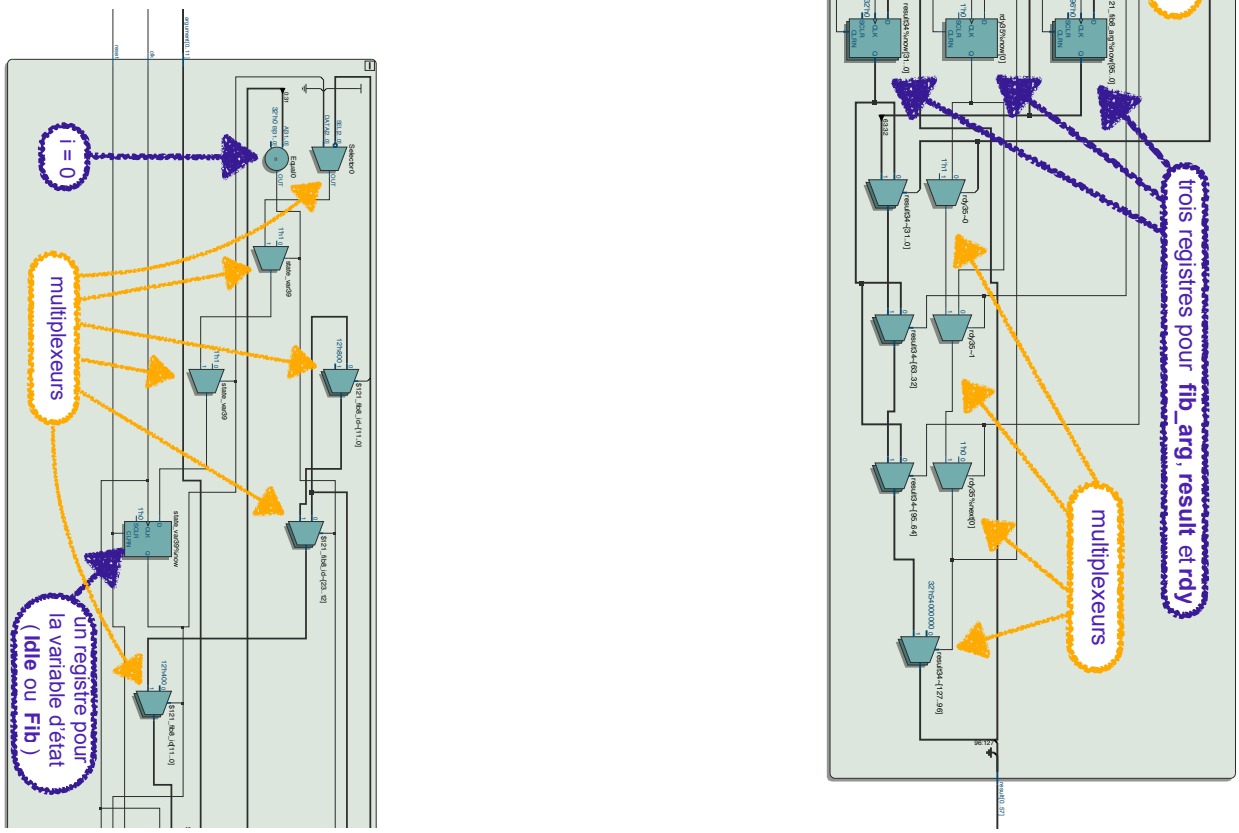


FIGURE A.3 Partage des appels de fonctions récursives

```
1  let main ((i,r):int<11> * bool) : int<58> =
2    (* suspend si i est négatif *)
3    if i < 0 then 0 else (
4      let n = resize_int<32>(i) in
5      let ((x0,x1,x2,x3,x4,x5,x6,x7),rdy) = exec
6          (* 8 calculs en parallèle *)
7          ( fibonacci(n)   ||
8            fibonacci(n+1) ||
9            fibonacci(n+2) ||
10           fibonacci(n+3) ||
11           fibonacci(n+4) ||
12           fibonacci(n+5) ||
13           fibonacci(n+6) ||
14           fibonacci(n+7) )
15       default (n,n,n,n,n,n,n,n)
16       reset r   (* réinitialisation *)
17     in
18      (* séquence d'additions *)
19     let y = x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7
20     in
21     resize_int<58>(y)
22   ) ;;
```

Voici les caractéristiques du circuit engendré :

- **1993 éléments logiques (LE) dont 1299 registres d'un bit.**
  ⇒ environ 8 fois plus gros que l'exemple précédent (figure A.3).
- **Fréquence max : 86 MHz**
- le parallèle d'ECLAT juxtapose les circuits.

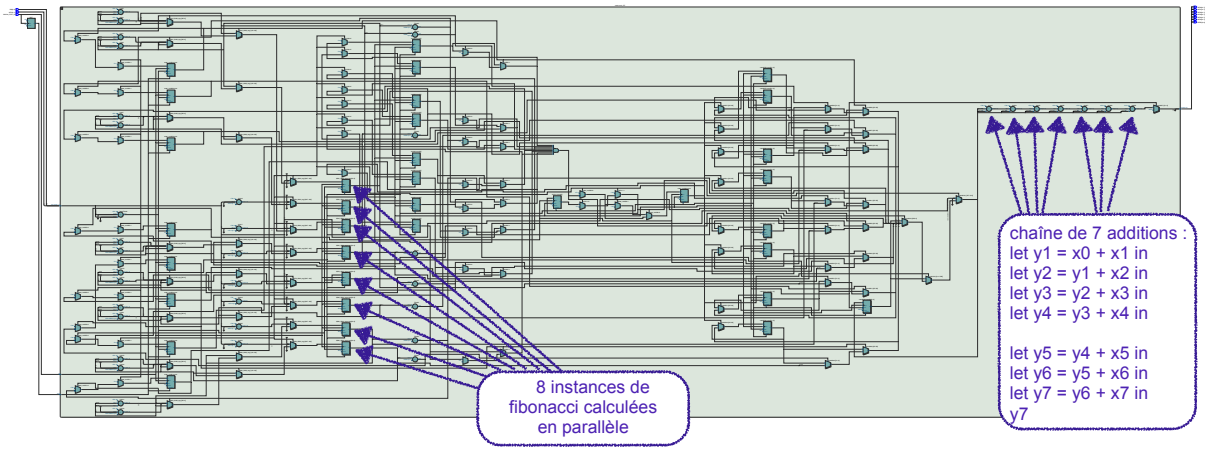


FIGURE A.4 Parallélisme au niveau RT

```
1  let rec delay n =
2    if n = 0 then () else delay (n-1) ;;
3
4  let main ((i,j) : int<6>*int<6>)
5      : (bool * bool * int<56>) =
6    let (b,rdy) =
7      exec
8       let a = create 1 in
9       set(a,0,false);
10      let (b,()) = ( (delay i; get(a,0))
11                   || (delay j; set(a,0,true)) )
12      in b
13     default false
14   in
15    (b,rdy,0) ;;
```

Ce programme déclare une variable partagée `a` (un tableau de une case) initialisée à `false`. La fonction `delay(`n`)` attend n cycles, puis termine. Le programme fait deux accès concurrents imprédictibles :

- la première branche exécute `delay(i)` (donc attend `i` cycles) puis lit `a` ;
- la deuxième branche exécute `delay(j)` (donc attend `j` cycles) puis modifie `b`.

Les valeurs `i` et `j` sont données par l'environnement et évoluent au cours du temps. Statiquement, on ne sait pas quand `i < j`, et donc on ne connaît pas le résultat, mais cela reste purement synchrone et déterministe.

Voici les caractéristiques du circuit engendré :

- **70 éléments logiques (LE) dont 29 registres d'un bit.**
- **Fréquence max : 215 MHz**
- 0 bit de mémoire RAM ;
  la mémoire partagée est implantée avec des LE ici.

Note : La nouvelle version du compilateur évite les *latchs* dans les circuits, contrairement à ceux que l'on trouve sur cette figure en entrée de la cellule de mémoire partagée.

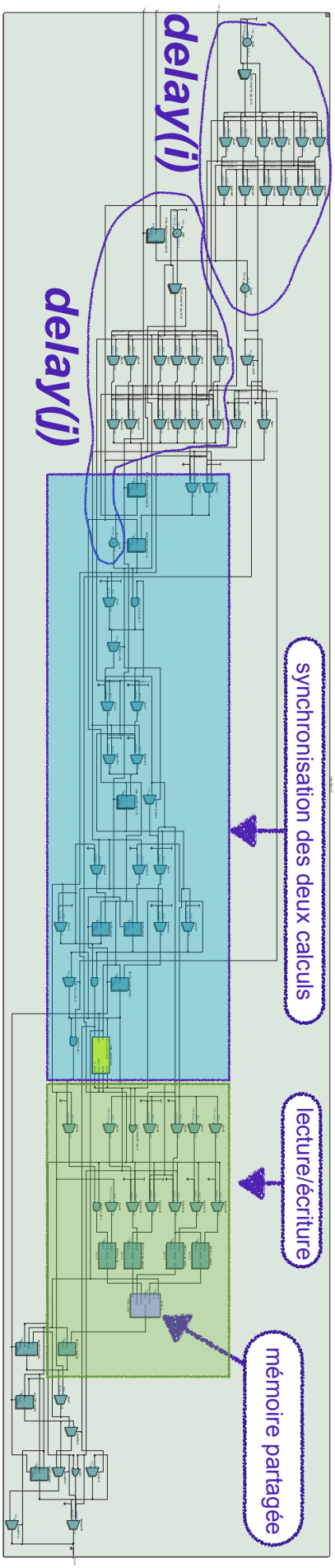


FIGURE A.5 Concurrence au niveau RT

⇒ *voir le code source : figure 3.21 page 50*

`abcro(a,b,c,r)` `flat_abcro(a,b,c,r)`

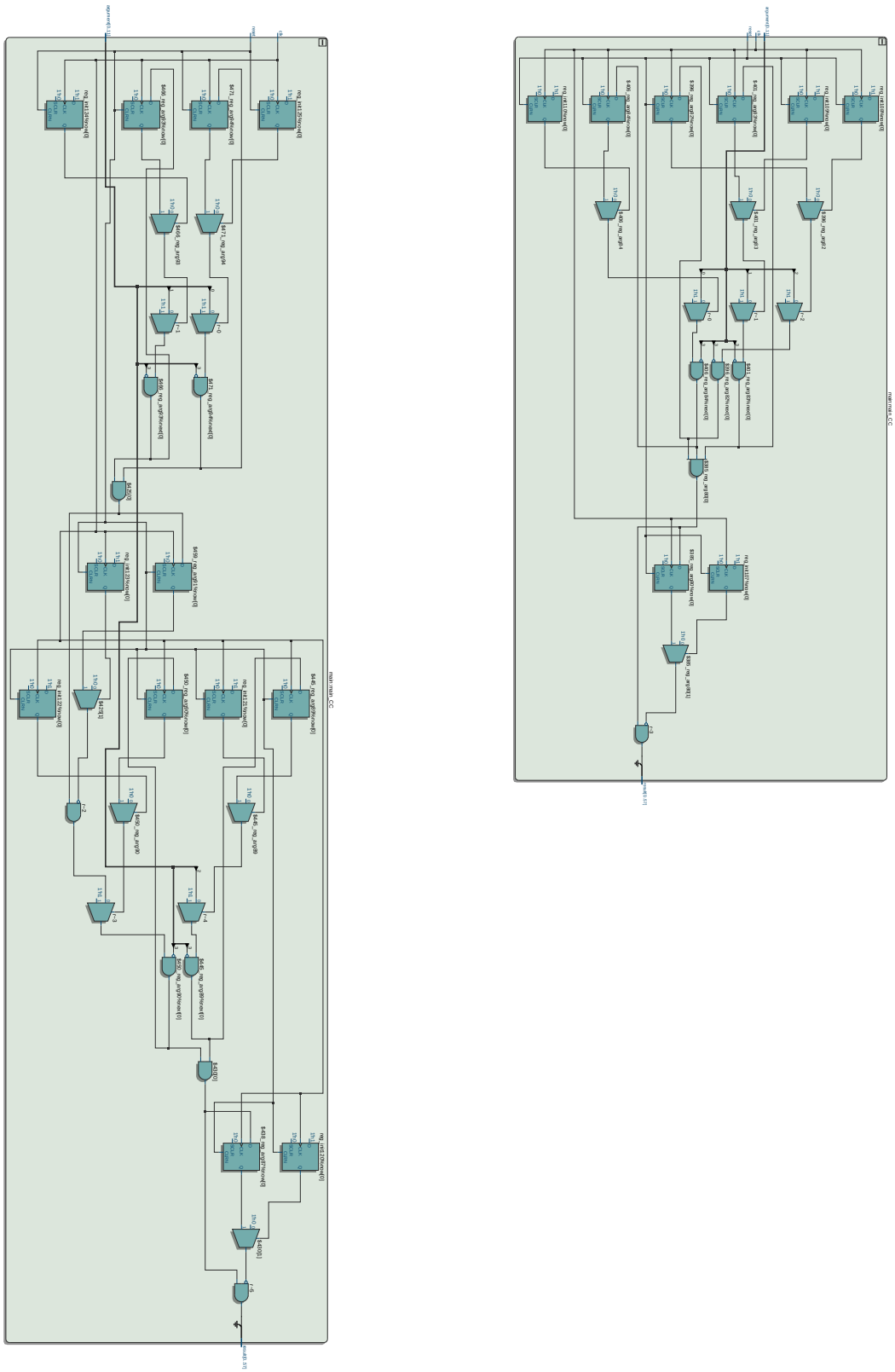


FIGURE A.6 Modularité

B Définition d'ECLAT

Ce chapitre donne une présentation compacte de la syntaxe et de la sémantique d'ECLAT. Cela correspond à la version implantée dans le compilateur ECLAT (voir annexe C), qui se base sur le noyau ECLAT présenté dans le manuscrit (chapitre 4) avec mémoire partagée (chapitre 6). Elle comprend aussi plusieurs extensions :

— une construction **match/with** analogue au **case/with** de MINI-HDL (voir section 5.5) ;

— des déclarations de types sommes (monomorphes et non récursifs) : ce sont des types de base de la forme (**X**$_1$ **of** $\mathring{\tau}_1$ | $\cdots$ **X**$_n$ **of** $\mathring{\tau}_n$) avec des constructeurs de valeurs **X**$_i$ (paramétrés par une constante c_i de type de base $\mathring{\tau}_i$) et du filtrage avec **match/with** ;

— des types abstraits de la forme ($\mathring{\tau}_1, \cdots \mathring{\tau}_m$) **T**<$\theta_1, \cdots \theta_n$> avec m et n possiblement nuls (par exemple, **unsigned_int**<θ>) ; les types abstraits se manipulent via des fonctions externes ;

— des primitives d'affichage et d'assertions pour la simulation de circuits au niveau RT.

Voici un exemple de programme ECLAT écrit en utilisant certaines de ces extensions. Il modélise un automate à deux états **On** et **Off**. L'état **Off** est l'état initial. L'état **On** est paramétré [1] par un entier qui augmente de un à chaque fois que l'entrée `event` vaut `true`. Lorsque l'entrée `go` (*resp.* `stop`) vaut `true`, l'automate passe dans l'état **On** (*resp.* **Off**).

```
1  type state = Off of unit | On of int<16> ;;
2
3  let counter (go,stop,event) =
4    let o = reg (fun (s:state) ->
5              match s with
6              | Off() -> if go then On(c) else Off()
7              | On(c) -> assert (c >= 0);
8                         if stop then Off() else
9                         if event then (print_string "bip"; On(c + 1))
10                        else On(c))
11           init Off() in
12    o ;;
```

FIGURE B.1 Programmation d'un automate en ECLAT

1. Ce trait de programmation est inspiré des *machines à états paramétrées* de LUCID SYNCHRONE [56] en se limitant à des transitions dites *faibles* et sans mécanisme de réinitialisation intégré.

B.1 Syntaxe

programme	π	::=
déclaration globale		d ;; π
point d'entrée		| `let` f = `fun` `(`p`:`τ`)` `:` $\mathring{\tau} \to e$
déclaration	d	::=
fonction externe		| `external` $f : \mathring{\tau} \xrightarrow{\delta} \mathring{\tau}'$
type somme		| `type` $\mathbf{T} = \mathbf{X}_1$ `of` $\mathring{\tau}_1$ | $\cdots$ $\mathbf{X}_n$ `of` $\mathring{\tau}_n$
motif	p	::= `()` | x | `(`p`,`p`)`
expression	e	::=
constante		c
variable		| x
opérateur		| op
constructeur non appliqué		| $\mathbf{X}$
définition locale		| `let` p`:`τ = e `in` e
paire séquentielle		| `(`e`,`e`)`
paire parallèle		| `(`$e \| \| e$`)`
conditionnelle		| `if` e `then` e `else` e
application		| e e
fonction		| `fun` `(`p`:`τ`)` `:` $\mathring{\tau} \to e$
fonction récursive		| `fix` f `(fun` `(`p`:`τ`)` `:` $\mathring{\tau} \to e$`)`
exec		| $\texttt{exec}_\ell$ e `default` e `reset` e
registre		| $\texttt{reg}_\ell$ `(fun` `(`p`:`$\mathring{\tau}$`)` `:` $\mathring{\tau} \to e$`)` `init` e
création de tableau		| `make<`θ`>` c | $\texttt{create}_{\mathring{\tau}}$`<`$\theta$`>()`
suspension		| `halt` e
traitement par cas		| `match` e `with` | $c_1 \to e_1$ | $\cdots$ $c_n \to e_n$ | $p \to e$
filtrage		| `match` e `with` | $\mathbf{X}_1$ $p_1 \to e_1$ | $\cdots$ $\mathbf{X}_n \to e_n$
adresse de tableau		| `(`ℓ`:`$\mathring{\tau}$ `array<`θ`>)`

note : les adresses n'apparaissent pas dans les programmes sources

FIGURE B.2 Syntaxe abstraite

type	τ	$::=$
type de base		$\mathring{\tau}$
type de paire		$\mid \tau$ `*` τ
type fonctionnel		$\mid \tau \xrightarrow{\delta} \mathring{\tau}$
type de tableau		$\mid \mathring{\tau}$ `array<`θ`>`
inconnue		$\mid$ *'a*
type de base	$\mathring{\tau}$	$::=$
type de paire		$\mathring{\tau}$ `*` $\mathring{\tau}$
type de base prédéfini		$\mid$ `unit` $\mid$ `bool` $\mid$ `int<`θ`>` $\mid \mathring{\tau}$ `vect<`θ`>`
inconnue		$\mid$ *'B*
type abstrait		$\mid (\mathring{\tau}_1, \cdots \mathring{\tau}_m)$ **T**`<`$\theta_1, \cdots \theta_n$`>`
type somme		$\mid$ `(`$\mathbf{X}_1$ `of` $\mathring{\tau}_1$ `|` $\cdots$ $\mathbf{X}_n$ `of` $\mathring{\tau}_n$`)`
taille	θ	$::=$
littéral		n
inconnue		$\mid$ *'N*
durée pire cas	δ	$::=$
instantanée		$\mathbf{0}$
non-instantanée		$\mid \mathbf{1}$
composition séquentielle		$\mid \delta + \delta$
composition parallèle		$\mid \max(\delta, \delta)$
inconnue		$\mid$ *'d*
schéma de type	σ	$::=$
type		τ
type quantifié		$\mid \forall$ *'v* $\cdot \sigma$
variable quantifiée	*'v*	$::=$ *'a* $\mid$ *'B* $\mid$ *'N* $\mid$ *'d*

FIGURE B.3 Langage de types

$$
\begin{aligned}
\mathbf{0} &\simeq \mathbf{0} \\
\mathbf{1} &\simeq \mathbf{1} \\
\mathbf{0} &\not\simeq \mathbf{1} \\
\mathbf{0} + \delta &\simeq \delta \\
\mathbf{1} + \delta &\simeq \mathbf{1} \\
\delta_1 + \delta_2 &\simeq \delta_2 + \delta_1 \\
\max(\mathbf{0}, \delta) &\simeq \delta \\
\max(\mathbf{1}, \delta) &\simeq \mathbf{1} \\
\max(\delta_1, \delta_2) &\simeq \max(\delta_2, \delta_1)
\end{aligned}
$$

FIGURE B.4 Règles d'unification des durées

constante	c	::=
unité		`()`
booléen		| `true` | `false`
entier signé		| (*l* : `int<`θ`>`)
paire de constantes		| (*c*,*c*)
vecteur		| {$c_1, \cdots c_n$}
constructeur appliqué		| **X** *c*
opérateur	*op*	::=
		`not` | `&` | `or` | `xor`
fonctions		| `fst` | `snd`
combinatoires		| `+` | `-` | `*` | `<` | `>` | `<=` | `>=` | `=`
		| `resize_int<`θ`>`
		| `vect_create<`θ`>`
		| `vect_size`
fonctions instantanées avec		| `/` | `mod`
gestion d'erreurs non instantanée		| `vect_nth` | `vect_copy_with`
accès mémoire		| `length` | `get` | `set`
		note : `length` est instantanée, `get` et `set` ne le sont pas.
opérateurs réservés		| `%acquire` | `%release`
assertions		| `assert`
primitives d'affichage (simulation)		| `print_int` | `print_string<`*str*`>`

FIGURE B.5 Constantes et opérateurs

Renommage de toutes les variables dans un motif p :

$$\begin{array}{lcl}
\textbf{rename}(\texttt{()}) & \stackrel{\text{def}}{=} & \texttt{()} \\
\textbf{rename}(x) & \stackrel{\text{def}}{=} & x' \text{ où } x' \text{ est un nom frais} \\
\textbf{rename}(\texttt{(}p_1 \| p_2\texttt{)}) & \stackrel{\text{def}}{=} & \texttt{(}p_1 \| p_2\texttt{)} \\
 & & \text{où } p_1' = \textbf{rename}(p_1) \\
 & & \text{et } p_2' = \textbf{rename}(p_2)
\end{array}$$

Conversion d'un motif en une expression :

$$\begin{array}{lcl}
\textbf{exp-of-pat}(\texttt{()}) & \stackrel{\text{def}}{=} & \texttt{()} \\
\textbf{exp-of-pat}(x) & \stackrel{\text{def}}{=} & x \\
\textbf{exp-of-pat}(\texttt{(}p_1 \| p_2\texttt{)}) & \stackrel{\text{def}}{=} & \texttt{(}e_1 \| e_2\texttt{)} \\
 & & \text{où } e_1 = \textbf{exp-of-pat}(p_1) \\
 & & \text{et } e_2 = \textbf{exp-of-pat}(p_2)
\end{array}$$

Substitution sans capture d'un motif p par une expression e' dans une expression e :

$$\begin{array}{lcl}
e[p \mapsto e'] & \stackrel{\text{def}}{=} & \textbf{subst-pat}(e, p, e') \\
\textbf{subst-pat}(e, \texttt{()}, e') & \stackrel{\text{def}}{=} & e \\
\textbf{subst-pat}(e, x, e') & \stackrel{\text{def}}{=} & e[x \mapsto e'] \\
\textbf{subst-pat}(\texttt{(}e_1 \| e_2\texttt{)}, \texttt{(}p_1 \| p_2\texttt{)}, e') & \stackrel{\text{def}}{=} & \texttt{(}e_1 \| e_2\texttt{)} \\
 & & \text{où } e_1 = \textbf{subst-pat}(e_1, p_1, e') \\
 & & \text{et } e_2 = \textbf{subst-pat}(e_1, p_2, e') \\
 & & \\
e[x \mapsto e'] & : & \text{fonction de substitution sans captures} \\
 & & \text{des occurrences de } x \text{ par } e' \text{ dans } e \\
 & & \text{(définie figure B.27)}
\end{array}$$

Ensemble des noms introduits dans un motif p :

$$\begin{array}{rcl}
\textbf{vars-of-pat}(\texttt{()}) & \stackrel{\text{def}}{=} & \emptyset \\
\textbf{vars-of-pat}(x) & \stackrel{\text{def}}{=} & \{x\} \\
\textbf{vars-of-pat}(\texttt{(}p\texttt{,}p'\texttt{)}) & \stackrel{\text{def}}{=} & \textbf{vars-of-pat}(p) \cup \textbf{vars-of-pat}(p')
\end{array}$$

FIGURE B.6 Opérations syntaxiques

$$\underbrace{\left(\begin{array}{l}\texttt{let } x_1 : \tau_1 = v_1 \texttt{;;} \\ \ldots \\ \texttt{let } x_n : \tau_n = v_n \texttt{;;} \\ \texttt{let } f : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}' = v \texttt{;;}\end{array}\right)}_{\textbf{syntaxe concrète}} \stackrel{\text{def}}{=} \underbrace{\left(\begin{array}{l}\texttt{let } f = \texttt{fun } (y : \mathring{\tau}) : \mathring{\tau}' \rightarrow \\ \quad \texttt{let } x_1 : \tau_1 = v_1 \texttt{ in} \\ \quad \ldots \\ \quad \texttt{let } x_n : \tau_n = v_n \texttt{ in} \\ \quad v\ y\end{array}\right)}_{\textbf{syntaxe abstraite}} \text{ où } y \text{ est un nom frais.}$$

FIGURE B.7 Abréviations dans la syntaxe des programmes

$$\begin{array}{rcl}
\tau \Rightarrow \mathring{\tau}' & \stackrel{\text{def}}{=} & \tau \xrightarrow{\mathbf{0}} \mathring{\tau}' \\
\tau \rightarrow \mathring{\tau}' & \stackrel{\text{def}}{=} & \tau \xrightarrow{\mathbf{1}} \mathring{\tau}' \\
e_1\ op\ e_2 & \stackrel{\text{def}}{=} & op(e_1, e_2) \\
(e : \tau) & \stackrel{\text{def}}{=} & \texttt{let } x : \tau = e \texttt{ in } x \quad \text{où } x \text{ est nom frais.} \\
\texttt{let } p = e \texttt{ in } e' & \stackrel{\text{def}}{=} & \texttt{let } p : _ = e \texttt{ in } e' \\
\texttt{fun } (p : \tau) \rightarrow e & \stackrel{\text{def}}{=} & \texttt{fun } (p : \tau) : _ \rightarrow e \\
\texttt{fix } f\ (\texttt{fun } (p : \tau) \rightarrow e) & \stackrel{\text{def}}{=} & \texttt{fix } f\ (\texttt{fun } (p : \tau) : _ \rightarrow e) \\
\texttt{fun } p \rightarrow e & \stackrel{\text{def}}{=} & \texttt{fun } (p : _) : _ \rightarrow e \\
\texttt{fix } f\ (\texttt{fun } p \rightarrow e) & \stackrel{\text{def}}{=} & \texttt{fix } f\ (\texttt{fun } (p : _) : _ \rightarrow e) \\
l & \stackrel{\text{def}}{=} & (l : \texttt{int<_>}) \quad \text{pour } l \text{ un littéral entier positif} \\
\texttt{let } f\ p = e \texttt{ in } e' & \stackrel{\text{def}}{=} & \texttt{let } f = \texttt{fun } p \rightarrow e \texttt{ in } e' \\
\texttt{let } f\ p = e \texttt{;;} & \stackrel{\text{def}}{=} & \texttt{let } f = \texttt{fun } p \rightarrow e \texttt{;;} \\
\texttt{if } e_1 \texttt{ then } e_2 & \stackrel{\text{def}}{=} & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else ()} \\
\texttt{reg (fun } p \rightarrow e) \texttt{ init } e_0 & \stackrel{\text{def}}{=} & \texttt{reg (fun } (p : _) : _ \rightarrow e) \texttt{ init } e_0 \\
\texttt{exec } e \texttt{ default } e_0 & \stackrel{\text{def}}{=} & \texttt{exec } e \texttt{ default } e_0 \texttt{ reset false} \\
\\
\texttt{print_string } str & \stackrel{\text{def}}{=} & \texttt{print_string<}str\texttt{> ()}
\end{array}$$

FIGURE B.8 Abréviations dans la syntaxe des expressions

$$
\begin{array}{rcl}
\texttt{let rec } f\ p = e \texttt{ in } e' & \stackrel{\text{def}}{=} & \texttt{let } f = \texttt{fix } f\ (\texttt{fun } p \to e) \texttt{ in } e' \\
e ; e' & \stackrel{\text{def}}{=} & \texttt{let ()} = e \texttt{ in } e' \\
(p_1, p_2, \cdots p_n) & \stackrel{\text{def}}{=} & ((p_1, p_2), \cdots p_n) \\
(e_1, e_2, \cdots e_n) & \stackrel{\text{def}}{=} & ((e_1, e_2), \cdots e_n) \\
(e_1 \| e_2 \| \cdots e_n) & \stackrel{\text{def}}{=} & ((e_1 \| e_2) \| \cdots e_n) \\
\texttt{let } p_1 = e_1 \texttt{ and } \cdots p_n = e_n \texttt{ in } e' & \stackrel{\text{def}}{=} & \texttt{let } (p_1, \cdots p_n) = (e_1 \| \cdots e_n) \texttt{ in } e' \\
\texttt{pause } e & \stackrel{\text{def}}{=} & (\texttt{fix } f\ (\texttt{fun ()} \to e))\texttt{()} \text{ où } f \text{ est un nom frais.}
\end{array}
$$

FIGURE B.9 Constructions dérivées (expressions et motifs)

$$
\begin{array}{lcl}
\textbf{type}(\texttt{true}) & \stackrel{\text{def}}{=} & \texttt{bool} \\
\textbf{type}(\texttt{false}) & \stackrel{\text{def}}{=} & \texttt{bool} \\
\textbf{type}(\texttt{()}) & \stackrel{\text{def}}{=} & \texttt{unit} \\
\textbf{type}((l\texttt{:int<}n\texttt{>})) & \stackrel{\text{def}}{=} & \texttt{int<}n\texttt{>} \\
\textbf{type}(\{c_1, \cdots c_n\}) & \stackrel{\text{def}}{=} & \mathring{\tau}\ \texttt{vect<}n\texttt{>} \quad \text{où } \mathring{\tau} \stackrel{\text{def}}{=} \textbf{type}(c_1) \\
\textbf{type}(\texttt{fun } (p\texttt{:}\tau)\ \texttt{:}\ \mathring{\tau}' \to e) & \stackrel{\text{def}}{=} & \tau \xrightarrow{\texttt{'}d} \mathring{\tau}' \\
 & & \text{où } \texttt{'}d \text{ est une inconnue de durée fraîche} \\
\textbf{type}(\texttt{fix } f\ (\texttt{fun } (p\texttt{:}\tau)\ \texttt{:}\ \mathring{\tau}' \to e)) & \stackrel{\text{def}}{=} & \tau \xrightarrow{\mathbf{1}} \mathring{\tau}' \\
\textbf{type}((\ell\texttt{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})) & \stackrel{\text{def}}{=} & \mathring{\tau}\ \texttt{array<}n\texttt{>} \\
e[\tau \mapsto \tau] & \stackrel{\text{def}}{=} & e \\
e[\mathring{\tau}\ \texttt{array<}\texttt{'}N\texttt{>} \mapsto \mathring{\tau}'\ \texttt{array<}n\texttt{>}] & \stackrel{\text{def}}{=} & e[\texttt{'}N := n][\mathring{\tau} \mapsto \mathring{\tau}'] \\
e[\texttt{int<}\texttt{'}N\texttt{>} \mapsto \texttt{int<}n\texttt{>}] & \stackrel{\text{def}}{=} & e[\texttt{'}N := n] \\
e[\mathring{\tau}\ \texttt{vect<}\texttt{'}N\texttt{>} \mapsto \mathring{\tau}'\ \texttt{vect<}n\texttt{>}] & \stackrel{\text{def}}{=} & e[\texttt{'}N := n][\mathring{\tau} \mapsto \mathring{\tau}'] \\
e[(\tau_1 * \tau_2) \mapsto (\tau_1' * \tau_2')] & \stackrel{\text{def}}{=} & e[\tau_1 \mapsto \tau_1'][\tau_2 \mapsto \tau_2'] \\
e[(\tau_1 \xrightarrow{\delta} \mathring{\tau}_2) \mapsto (\tau_1' \xrightarrow{\delta'} \mathring{\tau}_2')] & \stackrel{\text{def}}{=} & e[\tau_1 \mapsto \tau_1'][\mathring{\tau}_2 \mapsto \mathring{\tau}_2']
\end{array}
$$

FIGURE B.10 Accès dynamique aux types des valeurs et instantiation d'inconnues de tailles

B.2 Systèmes de types

Le système de types d'Eclat est un système d'effet qui calcule une abstraction du temps d'exécution du programme : **0** si le programme est instantané, et **1** sinon.

Il s'énonce en deux jugements :

— Le jugement $\Gamma_0 \vdash \pi : \tau|\mathbf{0}$ signifie "dans l'environnement initial Γ_0, le programme π est instantané et a le type τ". Pour un programme π donné, le système infère à la fois un type et une estimation pire cas du temps d'exécution ; le programme est rejeté (sans être évalué ou compilé) s'il est mal typé ou s'il n'est pas instantané. Ce jugement est défini par les règles Ty-Prog, Ty-External et Ty-SumDecl figure B.12.

— Le jugement $\Gamma \vdash e : \tau|\delta$ signifie "dans l'environnement de typage Γ, l'expression e a le type τ et un temps d'exécution inférieur ou égal à δ cycles".

Un environnement de typage Γ est une fonction partielle des noms (x) ou des constructeurs (**X**) vers des schémas de types. On note $\Gamma[x : \sigma]$ l'extension de l'environnement de typage Γ avec le nom x de schéma de type σ. La fonction $\Gamma(x)$ de recherche dans l'environnement de typage est définie par $\Gamma[x : \sigma](x) = \sigma$ et $\Gamma[x : \sigma](y) = \Gamma(y)$ si $x \neq y$. On note $\Gamma[p : \tau]$ l'extension de l'environnement de typage Γ avec le motif p de type τ, définie par :

$$\begin{array}{lcl} \Gamma[\texttt{()} : \texttt{unit}] & = & \Gamma \\ \Gamma[x : \tau] & = & \Gamma[x : \tau] \\ \Gamma[\texttt{(}p\texttt{,}p'\texttt{)} : \tau \texttt{ * } \tau'] & = & \Gamma[p : \tau][p' : \tau'] \end{array}$$

Construire $\Gamma[p : \tau]$ suppose que p et τ soient *compatibles* structurellement. La compatibilité entre un motif p et un type τ est un relation $p \sim \tau$ définie par :

$$\begin{array}{rcll} \texttt{()} & \sim & \texttt{unit} & \\ x & \sim & \tau & \\ \texttt{(}p\texttt{,}p'\texttt{)} & \sim & \tau \texttt{ * } \tau' & \text{si } p \sim \tau \text{ et } p' \sim \tau' \end{array}$$

Figure B.11 Environnement de typage

Le type de certaines valeurs liées à un nom x par un `let` est généralisé, formant un schéma de type $\sigma = \mathrm{Gen}(\tau, \Gamma)$ associé à x dans un nouvel environnement de typage $\Gamma[x : \sigma]$.

On note $\mathcal{FV}(\tau)$ (*resp.* $\mathcal{FV}(\Gamma)$) l'ensemble des variables `'v` qui apparaissent libres dans le type τ (*resp.* l'environnement Γ).

La généralisation $\mathrm{Gen}(\tau, \Gamma)$ construit un schéma de type en généralisant, dans τ, toutes les variables `'v` (de types, de types de base, de taille ou de durée) qui n'apparaissent pas libres dans l'environnement de typage Γ :

$$\mathrm{Gen}(\tau,\Gamma) \stackrel{\text{def}}{=} \forall\ 'v_1\ \cdots\ 'v_n\ \cdot\tau \text{ où } \{\ 'v_1\ ,\cdots\ \ 'v_n\ \} \stackrel{\text{def}}{=} \mathcal{FV}(\tau)\setminus\mathcal{FV}(\Gamma)$$

La relation $\tau \leq \sigma$, qui se lit "τ est une instance du schéma de type σ" est définie par :

$$\begin{array}{lcll}
\tau & \leq & \tau & \\
\tau & \leq & \forall\ 'a\ \cdot\sigma & \text{s'il existe } \tau' \text{ tel que } \tau \leq \sigma[\ 'a\ \mapsto \tau'] \\
\tau & \leq & \forall\ 'B\ \cdot\sigma & \text{s'il existe } \mathring{\tau}' \text{ tel que } \tau \leq \sigma[\ 'B\ \mapsto \mathring{\tau}'] \\
\tau & \leq & \forall\ 'd\ \cdot\sigma & \text{s'il existe } \delta \text{ tel que } \tau \leq \sigma[\ 'd\ \mapsto \delta] \\
\tau & \leq & \forall\ 'N\ \cdot\sigma & \text{s'il existe } \theta \text{ tel que } \tau \leq \sigma[\ 'N\ \mapsto \theta]
\end{array}$$

Un critère syntaxique poly(e) est utilisé pour limiter la généralisation. Cela permet d'éviter, par exemple, que l'expression `(42:int<` $'N_0$ `>)` soit associée au schéma de type $\forall\ 'N\ \cdot$ `int<` $'N$ `>`. Le critère poly(e) est vrai dans les cas suivants (et faux pour tout autre forme d'expression) :

$$\begin{array}{l}
\mathrm{poly}(\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e) \\
\mathrm{poly}(\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e))
\end{array}$$

La règle Ty-External vérifie que la fonction externe déclarée est présente dans l'environnement de typage initial avant de typer le programme. tandis que la Ty-SumDecl type le programme où le nom x du type somme $\mathring{\tau}$ déclaré est remplacé pas sa définition, dans un environnement de typage enrichi avec chaque constructeur de valeur $\mathbf{X}_i$ de type $\mathring{\tau}_i \to \mathring{\tau}$ défini dans x.

Ty-External
$$\frac{\Gamma(f) = \mathring{\tau}_1 \xrightarrow{\delta} \mathring{\tau}_2 \qquad \Gamma \vdash \pi : \tau|\mathbf{0}}{\Gamma \vdash \texttt{external}\ f : \mathring{\tau}_1 \xrightarrow{\mathbf{0}} \mathring{\tau}_2\ \texttt{;;}\ \pi : \tau|\mathbf{0}}$$

Ty-Prog
$$\frac{\Gamma_0 \vdash \texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau}' \to e : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'|\mathbf{0}}{\Gamma_0 \vdash \texttt{let}\ f = \texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau}' \to e : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'|\mathbf{0}}$$

Ty-SumDecl
$$\frac{\Gamma[\mathbf{X}_1 \mapsto \mathring{\tau_x}]\cdots[\mathbf{X}_n \mapsto \mathring{\tau_x}] \vdash \pi[\mathbf{T} \coloneqq \mathring{\tau_x}] : \tau|\mathbf{0}}{\Gamma \vdash \texttt{type}\ \mathbf{T} = \underbrace{(\mathbf{X}_1\ \texttt{of}\ \mathring{\tau}_1\ |\cdots\ \mathbf{X}_n\ \texttt{of}\ \mathring{\tau}_n)}_{\mathring{\tau_x}}\ \texttt{;;}\ \pi : \tau|\mathbf{0}}$$

où $\pi[\mathbf{T} \coloneqq \tau]$ est la substitution syntaxique du nom de type $\mathbf{T}$ par sa définition τ dans le programme π. On suppose que les déclarations globales de types sommes ont des noms différents et que deux types sommes $\mathbf{T}_1$ et $\mathbf{T}_2$ n'ont jamais de constructeurs $\mathbf{X}_i$ en communs.

FIGURE B.12 Typage des programmes

Dans les règles Ty-Fun et Ty-Fix,
toute inconnue de taille qui apparaît libre dans le corps de la fonction considérée
mais n'apparaît pas libre dans le type de l'argument est unifiée
avec une taille unique spécifiable pour le programme entier (32 par défaut)

Ty-Let-poly

$$\frac{\begin{array}{c}\text{poly}(v) \text{ est vrai} \\ \Gamma \vdash v : \tau|\delta \qquad \sigma = \text{Gen}(\tau, \Gamma) \\ x \qquad \Gamma[x : \sigma] \vdash e' : \tau'|\delta'\end{array}}{\Gamma \vdash \texttt{let}\ x : \tau = v\ \texttt{in}\ e' : \tau'|\delta + \delta'}$$

Ty-Let-Mono

$$\frac{\begin{array}{c}\text{poly}(e) \text{ est faux ou } p \text{ n'est pas une variable} \\ \Gamma \vdash e : \tau|\delta \qquad p \sim \tau \qquad \Gamma[p : \tau] \vdash e' : \tau'|\delta'\end{array}}{\Gamma \vdash \texttt{let}\ p : \tau = e\ \texttt{in}\ e' : \tau'|\delta + \delta'}$$

Ty-Var

$$\frac{\Gamma(x) = \sigma \qquad \tau \leq \sigma}{\Gamma \vdash x : \tau|\mathbf{0}}$$

Ty-Fun

$$\frac{p \sim \tau \qquad \Gamma[p : \tau] \vdash e : \mathring{\tau}'|\delta}{\Gamma \vdash \texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e : \tau \xrightarrow{\delta} \mathring{\tau}'|\mathbf{0}}$$

Ty-Fix

$$\frac{p \sim \tau \qquad \Gamma[f : \tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'][p : \tau] \vdash e : \mathring{\tau}'|\delta}{\Gamma \vdash \texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e) : \tau \xrightarrow{\mathbf{1}+\delta} \mathring{\tau}'|\mathbf{0}}$$

Ty-Pair

$$\frac{\Gamma \vdash e_1 : \tau_i|\delta_i \qquad i \in \{1, 2\}}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2|\delta_1 + \delta_2}$$

Ty-Par

$$\frac{\Gamma \vdash e_1 : \tau_i|\delta_i \qquad i \in \{1, 2\}}{\Gamma \vdash (e_1 \| e_2) : \tau_1 * \tau_2|\max(\delta_1, \delta_2)}$$

Ty-App

$$\frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\delta} \mathring{\tau}|\mathbf{0} \qquad \Gamma \vdash e_2 : \tau_2|\delta_2}{\Gamma \vdash e_1\ e_2 : \mathring{\tau}|(\mathbf{0} + \delta_2) + \delta}$$

Ty-Operator

$$\frac{\Delta(op) = \sigma \qquad \tau \leq \sigma}{\Gamma \vdash op : \tau|\mathbf{0}}$$

Figure B.13 Typage des expressions (première partie)

$$\textsc{Ty-If}\quad \frac{\Gamma \vdash e_1 : \texttt{bool}|\delta_1 \qquad \Gamma \vdash e_2 : \mathring{\tau}|\delta_2 \qquad \Gamma \vdash e_3 : \mathring{\tau}|\delta_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathring{\tau}|\delta_1 + \max(\delta_2, \delta_3)}$$

$$\textsc{Ty-True}\quad \Gamma \vdash \texttt{true} : \texttt{bool}|\mathbf{0}$$

$$\textsc{Ty-Unit}\quad \Gamma \vdash \texttt{()} : \texttt{unit}|\mathbf{0}$$

$$\textsc{Ty-Int}\quad \Gamma \vdash \texttt{(}l\texttt{:int<}\theta\texttt{>)} : \texttt{int<}\theta\texttt{>}|\mathbf{0}$$

$$\textsc{Ty-False}\quad \Gamma \vdash \texttt{false} : \texttt{bool}|\mathbf{0}$$

$$\textsc{Ty-Vect}\quad \frac{\Gamma \vdash c_i : \mathring{\tau} \qquad i \in \{1, \cdots\ n\}}{\Gamma \vdash \{c_1\,, \cdots\ c_n\} : \mathring{\tau}\ \texttt{vect<}\theta\texttt{>}|\mathbf{0}}$$

$$\textsc{Ty-Reg}\quad \frac{\Gamma \vdash \texttt{fun}\ (p\texttt{:}\mathring{\tau})\ \texttt{:}\ \mathring{\tau} \to e : \mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}|\mathbf{0} \qquad \Gamma \vdash e_0 : \mathring{\tau}|\mathbf{0}}{\Gamma \vdash \texttt{reg}\ (\texttt{fun}\ (p\texttt{:}\mathring{\tau})\ \texttt{:}\ \mathring{\tau}' \to e)\ \texttt{init}\ e_0 : \mathring{\tau}|\mathbf{0}}$$

$$\textsc{Ty-Exec}\quad \frac{\Gamma \vdash e : \mathring{\tau}|\delta \qquad \Gamma \vdash e_0 : \mathring{\tau}|\mathbf{0} \qquad \Gamma \vdash e_1 : \texttt{bool}|\mathbf{0}}{\Gamma \vdash \texttt{exec}\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1 : \mathring{\tau}\ \texttt{*}\ \texttt{bool}|\mathbf{0}}$$

$$\textsc{Ty-Halt}\quad \frac{\Gamma \vdash e : \tau|\delta}{\Gamma \vdash \texttt{halt}\ e : \texttt{unit}|\max(\mathbf{1}, \delta)}$$

$$\textsc{Ty-Array}\quad \Gamma \vdash \texttt{(}\ell\texttt{:}\mathring{\tau}\ \texttt{array<}\theta\texttt{>)} : \mathring{\tau}\ \texttt{array<}\theta\texttt{>}|\mathbf{0}$$

$$\textsc{Ty-Make}\quad \frac{\Gamma \vdash e : \mathring{\tau}|\mathbf{0}}{\Gamma \vdash \texttt{make<}\theta\texttt{>}\ e : \mathring{\tau}\ \texttt{array<}\theta\texttt{>}|\mathbf{1}}$$

$$\textsc{Ty-Create}\quad \Gamma \vdash \texttt{create}_{\mathring{\tau}}\texttt{<}\theta\texttt{>}\ \texttt{()} : \mathring{\tau}\ \texttt{array<}\theta\texttt{>}|\mathbf{0}$$

FIGURE B.14 Typage des expressions (deuxième partie)

$$\textsc{Ty-Ctor}\quad \frac{\Gamma(x) = \mathring{\tau} \qquad \mathring{\tau} = (\mathbf{X}_1\ \texttt{of}\ \mathring{\tau}_1\ \texttt{|} \cdots\ \mathbf{X}_n\ \texttt{of}\ \mathring{\tau}_n) \qquad \mathbf{X}_i \in \{\mathbf{X}_1, \cdots \mathbf{X}_n\}}{\Gamma \vdash \mathbf{X}_i : \tau_i \xrightarrow{\mathbf{0}} \mathring{\tau}|\mathbf{0}}$$

$$\textsc{Ty-Match-Const}\quad \frac{\Gamma \vdash e : \mathring{\tau}|\delta \qquad i \in \{1, \cdots n\} \qquad \Gamma \vdash e_i : \mathring{\tau}|\delta_i \qquad \Gamma \vdash e' : \mathring{\tau}|\delta'}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ c_1 \to e_1\ \texttt{|}\ \cdots\ c_n \to e_n\ \texttt{|}\ p \to e' : \mathring{\tau}|\delta + \max_{i=1}^{n}(\delta_i) + \delta'}$$

$$\textsc{Ty-Match-Ctor}\quad \frac{\Gamma \vdash e : \mathring{\tau}|\delta \qquad i \in \{1, \cdots n\} \qquad \Gamma \vdash e_i : \mathring{\tau}|\delta_i}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ \mathbf{X}_1\ p_1 \to e_1\ \texttt{|}\ \cdots \mathbf{X}_n \to e_n : \mathring{\tau}|\delta + \max_{i=1}^{n}(\delta_i)}$$

FIGURE B.15 Typage des expressions (nouvelles constructions)

Δ(`not`)	:		`bool` $\xrightarrow{0}$ `bool`
Δ(*op*) où *op* $\in$ {`&`, `or`, `xor`}	:		(`bool * bool`) $\xrightarrow{0}$ `bool`
Δ(`fst`)	:	$\forall$ *'a* $\cdot \forall$ *'B* $\cdot$	(*'B* `*` *'a*) $\xrightarrow{0}$ *'B*
Δ(`snd`)	:	$\forall$ *'a* $\cdot \forall$ *'B* $\cdot$	(*'a* `*` *'B*) $\xrightarrow{0}$ *'B*
Δ(*op*) où *op* $\in$ {`+`, `-`, `*`}	:	$\forall$ *'N* $\cdot$	(`int<` *'N* `> * int<` *'N* `>`) $\xrightarrow{0}$ `int<` *'N* `>`
Δ(*op*) où *op* $\in$ {`/`, `mod`}	:	$\forall$ *'N* $\cdot$	(`int<` *'N* `> * int<` *'N* `>`) $\xrightarrow{1}$ `int<` *'N* `>` **note** : ces opérations sont instantanées sauf en cas d'erreur (division par zéro), auquel cas l'exécution est suspendue (boucle infinie) jusqu'à la prochaine réinitialisation du bloc **`exec`** englobant.
Δ(*op*) où *op* $\in$ {`<`, `<=`, `>`, `=>`, `=`}	:	$\forall$ *'N* $\cdot$	(`int<` *'N* `> * int<` *'N* `>`) $\xrightarrow{0}$ `bool`
Δ(`resize_int<`θ`>`)	:	$\forall$ *'N* $\cdot$	`int<` *'N* `>` $\xrightarrow{0}$ `int<`θ`>`
Δ(`vect_create<`θ`>`)	:	$\forall$ *'B* $\cdot$	*'B* $\xrightarrow{0}$ *'B* `vect<`θ`>`
Δ(`vect_size`)	:	$\forall$ *'N* $\cdot \forall$ *'B* $\cdot$	*'B* `vect<` *'N* `>` $\xrightarrow{0}$ `int<16>`
Δ(`vect_nth`)	:	$\forall$ *'N*$_1$ $\cdot \forall$ *'N*$_2$ $\cdot \forall$ *'B* $\cdot$	(*'B* `vect<` *'N*$_1$ `> * int<` *'N*$_2$ `>` $\xrightarrow{1}$ *'B*)
Δ(`vect_copy_with`)	:	$\forall$ *'N*$_1$ $\cdot \forall$ *'N*$_2$ $\cdot \forall$ *'B* $\cdot$	(*'B* `vect<` *'N*$_1$ `> * int<` *'N*$_2$ `>) *` *'B* $\xrightarrow{1}$ *'B* `vect<` *'N* `>` **note** : voir note précédente sur les opérations instantanées avec erreurs potentielles (indice hors des bornes)
Δ(`length`)	:	$\forall$ *'N* $\cdot \forall$ *'B* $\cdot$	*'B* `array<` *'N* `>` $\xrightarrow{0}$ `int<16>`
Δ(`get`)	:	$\forall$ *'N*$_1$ $\cdot \forall$ *'N*$_2$ $\cdot \forall$ *'B* $\cdot$	(*'B* `array<` *'N*$_1$ `> * int<` *'N*$_2$ `>` $\xrightarrow{1}$ *'B*
Δ(`set`)	:	$\forall$ *'N*$_1$ $\cdot \forall$ *'N*$_2$ $\cdot \forall$ *'B* $\cdot$	(*'B* `array<` *'N*$_1$ `> * int<` *'N*$_2$ `> *` *'B*) $\xrightarrow{1}$ `unit`
Δ(`%acquire`$_\ell$)	:		`unit` $\xrightarrow{0}$ `unit`
Δ(`%release`$_\ell$)	:		`unit` $\xrightarrow{0}$ `unit`
Δ(`assert`)	:		`bool` $\xrightarrow{0}$ `unit`
Δ(`print_int`)	:	$\forall$ *'N* $\cdot$	`int<` *'N* `>` $\xrightarrow{0}$ `unit`
Δ(`print_string<`*str*`>`)	:		`unit` $\xrightarrow{0}$ `unit`

FIGURE B.16 Typage des opérateurs (version étendue)

B.3 Sémantique comportementale

On appelle flot de simulation un suite S construite en plaçant des éléments m en tête avec la notation $(m \bullet S)$. Un élément m du flot de simulation est un entier $\mathsf{INT}(n)$, une chaîne de caractères $\mathsf{STRING}(str)$ ou une alarme ALARM.

La sémantique comportementale d'ECLAT repose sur trois jugements :

— Le jugement $\mu, S \vdash \pi\ vs \Downarrow vs', S'$ signifie "dans l'état mémoire μ et le flux de simulation S, le programme π avec la mémoire μ transforme le flot d'entrées vs en un flot de sorties vs' et retourne un flux de simulation S'". Un flot vs est une suite infinie de valeurs $(v \bullet vs')$ où v est l'élément en tête du flot vs et vs' est un flot. Ce jugement est défini figure B.19 par les règles d'inférence EVAL-PROG, EVAL-EXTERNAL-DECL et EVAL-SUM-DECL. La règle EVAL-EXTERNAL-DECL fait intervenir un environnement global de fonctions externes **EXTERNAL**(f). C'est une fonction partielle des nom de fonctions externes vers leur définition $(\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)$.

— Le jugement $\mu, S \vdash \pi(v) \Rightarrow v', \mu', S'$ signifie "dans l'état mémoire μ et le flux de simulation S, le programme π avec l'entrée v retourne instantanément la sortie v' dans l'état mémoire μ' et le flux de simulation S'" Ce jugement est défini par la règle INTERACTION sur la figure B.19.

— Le jugement $e/\mu, S \hookrightarrow e'/\mu', S'$ signifie "la configuration $e/\mu, S$ se réduit en $e'/\mu', S'$ instantanément". Ce jugement est défini par un ensemble de règles d'inférence sur les figures B.18, B.20 et B.21.

On note $\mu[x \mapsto v]$ la modification du nom x de la mémoire μ avec la valeur v, telle que $\mu[x \mapsto v](x) = v$ et $\mu[y \mapsto v](x) = \mu[x \mapsto v]$ si $x \neq y$.

valeur	v	::=
constante		c
paire		$\mid (v,v)$
fonction		$\mid \texttt{fun}\ (p{:}\tau){:}\mathring{\tau} \to e$
fonction récursive		$\mid \texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau){:}\mathring{\tau} \to e)$
opérateur		$\mid op$
adresse mémoire		$\mid \ell_\theta$
constructeur non appliqué		$\mid \mathbf{X}$
constante (figure B.5)	c	::= $\texttt{()} \mid \texttt{true} \mid \texttt{false}$
		$\mid (l : \texttt{int<}\theta\texttt{>})$
		$\mid (c,c) \mid \{c_1, \cdots c_n\}$
constructeur appliqué		$\mid \mathbf{X}\ c$

FIGURE B.17 Langage de valeurs (version étendue)

REDUCE-VAL

$$v/\mu, S \hookrightarrow v/\mu, S$$

REDUCE-LET-VAL

(**note** : $e[p \mapsto v]$ est définie figure B.6)

$$\frac{e_1/\mu, S \hookrightarrow v/\mu', S' \qquad e_2'[p \mapsto v]/\mu', S' \hookrightarrow e'/\mu'', S''}{\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2/\mu, S \hookrightarrow e'/\mu'', S''}$$

REDUCE-LET-PAUSE

$$\frac{e_1/\mu, S \hookrightarrow e_1'/\mu', S' \qquad e_1' \notin \mathbb{V}}{\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2/\mu, S \hookrightarrow \texttt{let}\ p{:}\tau = e_1'\ \texttt{in}\ e_2/\mu', S'}$$

REDUCE-PAIR-VAL

$$\frac{e_1/\mu, S \hookrightarrow v_1/\mu', S' \qquad e_2/\mu', S' \hookrightarrow e_2'/\mu'', S''}{(e_1\,,e_2)/\mu, S \hookrightarrow (v_1\,,e_2')/\mu'', S''}$$

REDUCE-PAIR-PAUSE

$$\frac{e_1/\mu, S \hookrightarrow e_1'/\mu', S' \qquad e_1' \notin \mathbb{V}}{(e_1\,,e_2)/\mu, S \hookrightarrow (e_1'\,,e_2)/\mu', S'}$$

REDUCE-PAR-VAL

$$\frac{e_1/\mu, S \hookrightarrow v_1/\mu', S' \qquad e_2/\mu', S' \hookrightarrow v_2/\mu'', S''}{(e_1 \| e_2)/\mu, S \hookrightarrow (v_1\,,v_2)/\mu'', S''}$$

REDUCE-PAR-PAUSE

$$\frac{e_1/\mu, S \hookrightarrow e_1'/\mu', S' \qquad e_2/\mu', S' \hookrightarrow e_2'/\mu'', S'' \qquad e_1' \notin \mathbb{V} \text{ ou } e_2' \notin \mathbb{V}}{(e_1 \| e_2)/\mu, S \hookrightarrow (e_1' \| e_2')/\mu'', S''}$$

REDUCE-IF-TRUE

$$\frac{e/\mu, S \hookrightarrow \texttt{true}/\mu', S' \qquad e_1/\mu', S' \hookrightarrow e_1'/\mu'', S''}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2/\mu, S \hookrightarrow e_1'/\mu'', S''}$$

REDUCE-IF-FRUE

$$\frac{e/\mu, S \hookrightarrow \texttt{false}/\mu', S' \qquad e_2/\mu', S' \hookrightarrow e_2'/\mu'', S''}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2/\mu, S \hookrightarrow e_2'/\mu'', S''}$$

REDUCE-IF-PAUSE

$$\frac{e/\mu, S \hookrightarrow e'/\mu', S' \qquad e_1' \notin \mathbb{V}}{\texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2/\mu, S \hookrightarrow \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2/\mu', S'}$$

REDUCE-APP-FUN

$$\frac{e_1/\mu, S \hookrightarrow (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)/\mu', S' \qquad e_2/\mu', S' \hookrightarrow v/\mu'', S'' \qquad e[\tau \mapsto \mathbf{type}(v)][p \mapsto v]/\mu'', S'' \hookrightarrow e'/\mu_3, S_3}{e_1\ e_2/\mu, S \hookrightarrow e'/\mu_3, S_3}$$

REDUCE-APP-OP

$$\frac{e_1/\mu, S \hookrightarrow op/\mu', S' \qquad e_2/\mu', S' \hookrightarrow v/\mu'', S'' \qquad \vartheta(op, v, \mu'', S'') \hookrightarrow e'/\mu_3, S_3}{e_1\ e_2/\mu, S \hookrightarrow e'/\mu_3, S_3}$$

REDUCE-APP-FIX

$$\frac{e_1/\mu, S \hookrightarrow \overbrace{(\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e))}^{\phi}/\mu', S' \qquad e_2/\mu', S' \hookrightarrow v/\mu'', S'' \qquad e' \stackrel{\text{def}}{=} e[\tau \mapsto \mathbf{type}(v)][f \mapsto \phi][v \mapsto p]}{e_1\ e_2/\mu, S \hookrightarrow \widehat{e'}/\mu'', S''}$$

REDUCE-APP-PAUSE

$$\frac{e_1/\mu, S \hookrightarrow v/\mu', S' \qquad e_2/\mu', S' \hookrightarrow e_2'/\mu'', S'' \qquad e_2' \notin \mathbb{V}}{e_1\ e_2/\mu, S \hookrightarrow v\ e_2'/\mu'', S''}$$

REDUCE-HAT

$$\frac{e/\mu, S \hookrightarrow e'/\mu', S'}{\widehat{e}\,/\mu, S \hookrightarrow e'/\mu', S'}$$

FIGURE B.18 Sémantique comportementale (noyau fonctionnel parallèle)

$$\textsc{Eval-Prog}\ \frac{(\texttt{fun}\ (p{:}\mathring{\tau})\to e)\ v/\mu, S \hookrightarrow v'/\mu', S'}{\mu, S \vdash (\texttt{let}\ f{:}\mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}' = \texttt{fun}\ p \to e)\,(v) \Rightarrow v', \mu', S'}$$

$$\textsc{Eval-External-Decl}\ \frac{\mathbf{EXTERNAL}(f) = v_f \qquad \mu, S \vdash \pi[f \mapsto v_f]\,(v) \Rightarrow v', \mu', S'}{\mu, S \vdash (\texttt{external}\ f : \mathring{\tau} \xrightarrow{\delta} \mathring{\tau}'\,;;\pi)\,(v) \Rightarrow v', \mu', S'}$$

$$\textsc{Eval-Sum-Decl}\ \frac{\mu, S \vdash \pi(v) \Rightarrow v', \mu', S'}{\mu, S \vdash (\texttt{type}\ \mathbf{T} = \mathbf{X}_1\ \texttt{of}\ \mathring{\tau}_1\ \texttt{|} \cdots\ \mathbf{X}_n\ \texttt{of}\ \mathring{\tau}_n\,;;\pi)\,(v) \Rightarrow v', \mu', S'}$$

$$\textsc{Interaction}\ \frac{\mu, S \vdash \pi(v) \Rightarrow v', \mu', S' \qquad \mu', S' \vdash \pi\ vs \Downarrow vs', S''}{\mu, S \vdash \pi\ (v\ \bullet vs) \Downarrow (v' \bullet vs'), S''}$$

$$\textsc{Reduce-Exec-Init}\ \frac{\ell \notin \mathsf{dom}(\mu)\ \text{ou}\ \mu(\ell) = \mathsf{none} \qquad \texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu'[\ell \mapsto (\texttt{fun}\ (()\texttt{:unit}) \to e)], S \hookrightarrow v/\mu', S'}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu, S \hookrightarrow v/\mu', S'}$$

$$\textsc{Reduce-Exec-Val}\ \frac{\mu(\ell) = (\texttt{fun}\ (()\texttt{:unit}) \to e_\ell) \qquad e_1/\mu, S \hookrightarrow \texttt{false}/\mu', S' \qquad e_\ell/\mu', S' \hookrightarrow v_\ell/\mu'', S''}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu, S \hookrightarrow (v_\ell\texttt{,true})/\mu''[\ell \mapsto \mathsf{none}], S''}$$

$$\textsc{Reduce-Exec-Default}\ \frac{\mu(\ell) = (\texttt{fun}\ (()\texttt{:unit}) \to e_\ell) \qquad e_1/\mu, S \hookrightarrow \texttt{false}/\mu', S' \qquad e_\ell/\mu', S' \hookrightarrow e'/\mu'', S'' \qquad e' \notin \mathbb{V} \qquad e_0/\mu'', S'' \hookrightarrow v_0/\mu_3, S_3}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu, S \hookrightarrow (v_0\texttt{,false})/\mu_3[\ell \mapsto (\texttt{fun}\ (()\texttt{:unit}) \to e')], S_3}$$

$$\textsc{Reduce-Exec-Reset}\ \frac{e_1/\mu, S \hookrightarrow \texttt{true}/\mu', S' \qquad \texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ \texttt{false}/\mu'[\ell \mapsto (\texttt{fun}\ (()\texttt{:unit}) \to e)], S' \hookrightarrow e_3/\mu'', S''}{\texttt{exec}_\ell\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_1/\mu, S \hookrightarrow e_3/\mu'', S''}$$

$$\textsc{Reduce-Reg-Step}\ \frac{\mu(\ell) = v \qquad (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e)\ v/\mu, S \hookrightarrow v'/\mu', S'}{\texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e)\ \texttt{init}\ e_0/\mu, S \hookrightarrow v'/\mu'[\ell \mapsto v'], S'}$$

$$\textsc{Reduce-Reg-Init}\ \frac{\ell \notin \mathsf{dom}(\mu) \qquad e_0/\mu, S \hookrightarrow v/\mu', S' \qquad \texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e)\ \texttt{init}\ e_0/\mu'[\ell \mapsto v], S' \hookrightarrow v'/\mu'', S''}{\texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e)\ \texttt{init}\ e_0/\mu, S \hookrightarrow v'/\mu'', S''}$$

$$\textsc{Reduce-Halt}\quad \texttt{halt}\ ()/\mu, S \hookrightarrow \texttt{halt}\ ()/\mu, S$$

FIGURE B.19 Sémantique comportementale (noyau réactif)

REDUCE-MAKE

$$\frac{e/\mu, S \hookrightarrow c/\mu', S' \qquad \ell \text{ est une adresse fraîche} \qquad \mathring{\tau} \stackrel{\text{def}}{=} \mathbf{type}(c)}{\texttt{make<}n\texttt{>}\ e/\mu, S \hookrightarrow \underbrace{\texttt{pause(); }\cdots\texttt{pause(); }}_{n+1 \text{ fois } \texttt{pause}}\ (\ell\texttt{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})/\mu'[\ell \mapsto \texttt{(false, }\underbrace{(c, \cdots\ c)}_{n \text{ fois}}\texttt{)}], S'}$$

REDUCE-CREATE

$$\frac{\ell \text{ est une adresse fraîche} \qquad \text{les } d_i(\mathring{\tau}) \text{ sont des valeurs non spécifiées de type } \mathring{\tau}}{\texttt{create}_{\mathring{\tau}}\texttt{()<}n\texttt{>}/\mu, S \hookrightarrow\ (\ell\texttt{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})/\mu[\ell \mapsto \texttt{(false, }\underbrace{(d_1(\mathring{\tau}), \cdots\ d_n(\mathring{\tau}))}_{n \text{ fois}}\texttt{)}], S}$$

FIGURE B.20 Sémantique comportementale (création de tableau)

REDUCE-MATCH-CONST

$$\frac{e/\mu, S \hookrightarrow c_i/\mu', S' \qquad c_i \in \{c_1, \cdots c_n\} \qquad e_i/\mu', S' \hookrightarrow e_i'/\mu'', S''}{(\texttt{match}\ e\ \texttt{with} \mid c_1 \to e_1 \mid c_n \to e_n \mid p \to e')/\mu, S \hookrightarrow e_i'/\mu'', S''}$$

REDUCE-MATCH-OTHERWISE

$$\frac{e/\mu, S \hookrightarrow c_i/\mu', S' \qquad c_i \notin \{c_1, \cdots c_n\} \qquad e_{\mathsf{otherwise}}[p \mapsto c_i]/\mu', S' \hookrightarrow e'_{\mathsf{otherwise}}/\mu'', S''}{(\texttt{match}\ e\ \texttt{with} \mid c_1 \to e_1 \mid c_n \to e_n \mid p \to e_{\mathsf{otherwise}})/\mu, S \hookrightarrow e'_{\mathsf{otherwise}}/\mu'', S''}$$

REDUCE-MATCH-CONST-PAUSE

$$\frac{e/\mu, S \hookrightarrow e'/\mu', S' \qquad e' \notin \mathbb{V}}{\begin{array}{c}(\texttt{match}\ e\ \texttt{with} \mid c_1 \to e_1 \mid c_n \to e_n \mid p \to e_{\mathsf{otherwise}})/\mu, S \\ \hookrightarrow (\texttt{match}\ e'\ \texttt{with} \mid c_1 \to e_1 \mid c_n \to e_n \mid p \to e_{\mathsf{otherwise}})/\mu', S'\end{array}}$$

REDUCE-MATCH-CTOR

$$\frac{e/\mu, S \hookrightarrow \mathbf{X}_i\ v/\mu', S' \qquad i \in \{1, \cdots n\} \qquad e_i[p_i \mapsto v]/\mu', S' \hookrightarrow e_i'/\mu'', S''}{(\texttt{match}\ e\ \texttt{with}\ \mathbf{X}_1\ p_1 \to e_1 \cdots \mid \mathbf{X}_n\ p_n \to e_n)/\mu, S \hookrightarrow e_i'/\mu'', S''}$$

REDUCE-MATCH-CTOR-PAUSE

$$\frac{e/\mu, S \hookrightarrow e'/\mu', S' \qquad e' \notin \mathbb{V}}{\begin{array}{c}(\texttt{match}\ e\ \texttt{with}\ \mathbf{X}_1\ p_1 \to e_1 \cdots \mid \mathbf{X}_n\ p_n \to e_n)/\mu, S \\ \hookrightarrow (\texttt{match}\ e'\ \texttt{with}\ \mathbf{X}_1\ p_1 \to e_1 \cdots \mid \mathbf{X}_n\ p_n \to e_n)/\mu', S'\end{array}}$$

FIGURE B.21 Sémantique comportementale (extensions)

$$
\begin{array}{rcl}
\vartheta(\texttt{not}, \texttt{true},\ \mu, S) & \stackrel{\text{def}}{=} & \texttt{false}/\mu, S \\
\vartheta(\texttt{not}, \texttt{false}, \mu, S) & \stackrel{\text{def}}{=} & \texttt{true}/\mu, S \\
\hline
\vartheta(\texttt{or}, (\texttt{false}, v), \mu, S) & \stackrel{\text{def}}{=} & v/\mu, S \\
\vartheta(\texttt{or}, (\texttt{true},\ v), \mu, S) & \stackrel{\text{def}}{=} & \texttt{true}/\mu, S \\
\hline
\vartheta(\texttt{\&}, (\texttt{false}, v), \mu, S) & \stackrel{\text{def}}{=} & \texttt{false}/\mu, S \\
\vartheta(\texttt{\&}, (\texttt{true},\ v), \mu, S) & \stackrel{\text{def}}{=} & v/\mu, S \\
\hline
\vartheta(\texttt{xor}, (\texttt{false}, v), \mu, S) & \stackrel{\text{def}}{=} & v/\mu, S \\
\vartheta(\texttt{xor}, (\texttt{true},\ v), \mu, S) & \stackrel{\text{def}}{=} & \texttt{not}(v)/\mu, S \\
\hline
\vartheta(\texttt{fst}, (v_1, v_2), \mu, S) & \stackrel{\text{def}}{=} & v_1/\mu, S \\
\vartheta(\texttt{snd}, (v_1, v_2), \mu, S) & \stackrel{\text{def}}{=} & v_2/\mu, S \\
\hline
\vartheta(\texttt{+}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & (l : \texttt{int<}n\texttt{>})/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} (l_1 + l_2) \mod 2^n \\
 & & \textbf{note} : \text{les entiers (de type } \texttt{int<}n\texttt{>}) \text{ sont} \\
 & & \text{signés avec une représentation } n \text{ bits} \\
 & & \text{en complément à 2 dont un bit de signe} \\
 & & \text{qui fait partie du littéral } l. \\
\vartheta(\texttt{-}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & (l : \texttt{int<}n\texttt{>})/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} (l_1 - l_2) \mod 2^n \\
\vartheta(\texttt{*}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & (l : \texttt{int<}n\texttt{>})/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} (l_1 \times l_2) \mod 2^n \\
\vartheta(\texttt{/}, ((\ell_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & (l : \texttt{int<}n\texttt{>})/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} (l_1 \div l_2) \mod 2^n \\
 & & \text{si } l_2 \neq 0 \\
\vartheta(\texttt{mod}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & (l : \texttt{int<}n\texttt{>})/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} (l_1 \,\%\, l_2) \mod 2^n \\
 & & \text{si } l_2 \neq 0 \\
\hline
\vartheta(\texttt{<}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & l/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 < l_2 \text{ sinon } \texttt{false} \\
\vartheta(\texttt{>}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & l/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 > l_2 \text{ sinon } \texttt{false} \\
\vartheta(\texttt{=}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & l/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 = l_2 \text{ sinon } \texttt{false} \\
\vartheta(\texttt{<=}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), \mu, S) & \stackrel{\text{def}}{=} & l/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 \leq l_2 \text{ sinon } \texttt{false} \\
\vartheta(\texttt{>=}, ((l_1 : \texttt{int<}n\texttt{>}), (l_2 : \texttt{int<}n\texttt{>})), S) & \stackrel{\text{def}}{=} & l/\mu, S \quad \text{où } l \stackrel{\text{def}}{=} \texttt{true si } l_1 \geq l_2 \text{ sinon } \texttt{false} \\
\hline
\vartheta(\texttt{resize_int<}n\texttt{>}, (l : \texttt{int<}m\texttt{>}), \mu, S) & \stackrel{\text{def}}{=} & (l' : \texttt{int<}n\texttt{>}), \mu, S \text{ où } l' \stackrel{\text{def}}{=} l \mod 2^n \\
\hline
\vartheta(\texttt{vect_create<}n\texttt{>}, c, \mu, S) & \stackrel{\text{def}}{=} & \{\underbrace{c, \cdots c}_{n \text{ fois}}\}/\mu, S \\
 & & \textbf{note} : n > 0 \text{ (à vérifier statiquement)} \\
\vartheta(\texttt{vect_size}, \{v_1, \cdots v_n\}, \mu, S) & \stackrel{\text{def}}{=} & (n : \texttt{int<16>})/\mu, S \\
\vartheta(\texttt{vect_nth}, (\{v_0, \cdots v_{n-1}\}, i), \mu, S) & \stackrel{\text{def}}{=} & v_i/\mu, S \qquad \text{si } i \in \{0, \cdots n-1\} \\
\vartheta(\texttt{vect_copy_with}, (v, i, c), \mu, S) & \stackrel{\text{def}}{=} & v'/\mu, S \\
 & & \text{où } v \stackrel{\text{def}}{=} \{v_0, \cdots v_{i-1}, v_i, v_{i+1} \cdots v_{n-1}\} \\
 & & \text{et } v' \stackrel{\text{def}}{=} \{v_0, \cdots v_{i-1}, c, v_{i+1} \cdots v_{n-1}\} \\
 & & \text{si } i \in \{0, \cdots n-1\}
\end{array}
$$

FIGURE B.22 Sémantique des opérateurs combinatoires

$$
\begin{array}{rcl}
\vartheta(\texttt{length}, (\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), \mu, S) & \stackrel{\text{def}}{=} & (n{:}\texttt{int<16>})/\mu, S \\
\vartheta(\texttt{get}, ((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), i), \mu, S) & \stackrel{\text{def}}{=} & \%\texttt{acquire}_\ell\ (); \texttt{pause}\ (\%\texttt{release}_\ell(); v_i)/\mu, S \\
 & & \quad \text{si } \mu(\ell) = (\texttt{false}, (v_0, \cdots v_i, \cdots v_{n-1})) \\
 & & \quad \text{et } i \in \{0, \cdots n-1\} \\
\vartheta(\texttt{set}, ((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), i, c), \mu, S) & \stackrel{\text{def}}{=} & \%\texttt{acquire}_\ell(); \texttt{pause}\ (\%\texttt{release}_\ell())/\mu', S \\
 & & \text{où } \mu' \stackrel{\text{def}}{=} \mu[\ell \cdot i \mapsto (\texttt{true}, (v_0, \cdots\ v_{i-1}, c, v_{i+1}, \cdots v_{n-1}))] \\
 & & \quad \text{si } \mu(\ell) = (\texttt{false}, (v_0, \cdots\ v_{i-1}, v_i, v_{i+1}, \cdots v_{n-1})) \\
 & & \quad \text{et } i \in \{0, \cdots n-1\} \\
\hline
\vartheta(\texttt{get}, ((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), i), \mu, S) & \stackrel{\text{def}}{=} & \texttt{pause}\ (\texttt{get}((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), i))/\mu, S \\
 & & \quad \text{si } \mu(\ell) = (\texttt{true}, v) \\
\vartheta(\texttt{set}, ((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), i, c), \mu, S) & \stackrel{\text{def}}{=} & \texttt{pause}\ (\texttt{set}((\ell{:}\mathring{\tau}\ \texttt{array<}n\texttt{>}), i, c))/\mu, S \\
 & & \quad \text{si } \mu(\ell) = (\texttt{false}, v) \\
\hline
\vartheta(\%\texttt{acquire}_\ell, (), \mu, S) & \stackrel{\text{def}}{=} & ()/\mu[\ell \mapsto (\texttt{true}, v)], S \quad \text{si } \mu(\ell) = (\texttt{false}, v) \\
\vartheta(\%\texttt{release}_\ell, (), \mu, S) & \stackrel{\text{def}}{=} & ()/\mu[\ell \mapsto (\texttt{false}, v)], S \quad \text{si } \mu(\ell) = (\texttt{true}, v)
\end{array}
$$

FIGURE B.23 Sémantique des accès mémoire

$$
\begin{array}{rcl}
\vartheta(\texttt{assert}, \texttt{true}, \mu, S) & \stackrel{\text{def}}{=} & ()/\mu, S \\
\vartheta(\texttt{assert}, \texttt{false}, \mu, S) & \stackrel{\text{def}}{=} & ()/\mu, (\mathsf{ALARM} \bullet S)] \\
 & & \quad \textbf{note} : (\texttt{assert false})\ \text{ne suspend pas l'exécution} \\
 & & \quad \text{du programme (contrairement à la construction } \texttt{halt}\text{). Ce} \\
 & & \quad \text{choix de conception garantit qu'}\texttt{assert}\text{ termine toujours} \\
 & & \quad \text{instantanément (comme l'indique son type } \texttt{bool} \xrightarrow{\mathbf{0}} \texttt{unit}\text{).} \\
 & & \quad \text{Cela vise à ce que l'on puisse utiliser } \texttt{assert}\text{ dans les} \\
 & & \quad \text{parties interactives (comme sur la figure B.1) sans} \\
 & & \quad \text{contrevenir à la } \textit{subject reduction}\text{ en cas d'assertion erronée.} \\
\vartheta(\texttt{print_int}, n, \mu, S) & \stackrel{\text{def}}{=} & ()/\mu, (\mathsf{INT}(n) \bullet S)] \\
\vartheta(\texttt{print_string<}\textit{str}\texttt{>}, (), \mu, S) & \stackrel{\text{def}}{=} & ()/\mu, (\mathsf{STRING}(\textit{str}) \bullet S)]
\end{array}
$$

FIGURE B.24 Sémantique des primitives de simulation

Division par zéro

$$\vartheta(op, \underbrace{(v\texttt{,}((\texttt{0:int<}n\texttt{>})))}_{c}, \mu, S) \stackrel{\text{def}}{=} \texttt{halt}(op\ c)/\mu, S \quad \text{où } op \in \{\texttt{/}, \texttt{mod}\}$$

Indice hors des bornes

$$\vartheta(\texttt{vect_nth}, \overbrace{(\{v_0\texttt{,}\cdots\ v_{n-1}\texttt{}\texttt{,}i)}^{args}, \mu, S) \stackrel{\text{def}}{=} \texttt{halt (vect_nth } args\texttt{)}/\mu, S$$
si $i \notin \{0, \cdots n-1\}$

$$\vartheta(\texttt{vect_copy_with}, (v\texttt{,}i\texttt{,}c), \mu, S) \stackrel{\text{def}}{=} \texttt{halt (vect_copy_with (}v\texttt{,}i\texttt{,}c\texttt{))}/\mu, S$$
où $v \stackrel{\text{def}}{=} \{v_0\texttt{,}\cdots\ v_{n-1}\texttt{}\}$
si $i \notin \{0, \cdots n-1\}$

$$\vartheta(\texttt{get}, \overbrace{((\ell\texttt{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})\texttt{,}i)}^{args}, \mu, S) \stackrel{\text{def}}{=} \texttt{halt (get } args\texttt{)}/\mu, S$$
si $i \notin \{0, \cdots n-1\}$

$$\vartheta(\texttt{set}, \overbrace{((\ell\texttt{:}\mathring{\tau}\ \texttt{array<}n\texttt{>})\texttt{,}i\texttt{,}c)}^{args}, \mu, S) \stackrel{\text{def}}{=} \texttt{halt (set } args\texttt{)}/\mu, S$$
si $i \notin \{0, \cdots n-1\}$

FIGURE B.25 Gestion des erreurs d'exécution

$$
\begin{array}{rcl}
\mathsf{prefix}(\ell, c) & \stackrel{\text{def}}{=} & c \\
\mathsf{prefix}(\ell, \mathbf{X}) & \stackrel{\text{def}}{=} & \mathbf{X} \\
\mathsf{prefix}(\ell, x) & \stackrel{\text{def}}{=} & x \\
\mathsf{prefix}(\ell, (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)) & \stackrel{\text{def}}{=} & \texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to \mathsf{prefix}(\ell, e) \\
\mathsf{prefix}(\ell, (\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e))) & \stackrel{\text{def}}{=} & \texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to \mathsf{prefix}(\ell, e)) \\
\mathsf{prefix}(\ell, (\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2)) & \stackrel{\text{def}}{=} & \texttt{let}\ p{:}\tau = \mathsf{prefix}(\ell, e_1)\ \texttt{in}\ \mathsf{prefix}(\ell, e_2) \\
\mathsf{prefix}(\ell, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)) & \stackrel{\text{def}}{=} & \texttt{if}\ \mathsf{prefix}(\ell, e_1)\ \texttt{then}\ \mathsf{prefix}(\ell, e_2)\ \texttt{else}\ \mathsf{prefix}(\ell, e_3) \\
\mathsf{prefix}(\ell, \left(\begin{array}{l}\texttt{match}\ e_0\ \texttt{with} \\ \mid\ c_1 \to e_1\ \mid\ \cdots \\ \mid\ c_n \to e_n \\ \mid\ p \to e'\end{array}\right)) & \stackrel{\text{def}}{=} & \left(\begin{array}{l}\texttt{match}\ \mathsf{prefix}(\ell, e_0)\ \texttt{with} \\ \mid\ c_1 \to \mathsf{prefix}(\ell, e_1)\ \mid\ \cdots \\ \mid\ c_n \to \mathsf{prefix}(\ell, e_n) \\ \mid\ p \to \mathsf{prefix}(\ell, e')\end{array}\right) \\
\mathsf{prefix}(\ell, \left(\begin{array}{l}\texttt{match}\ e_0\ \texttt{with} \\ \mid\ \mathbf{X}_1\ p_1 \to e_1\ \mid\ \cdots \\ \mid\ \mathbf{X}_n\ p_n \to e_n\end{array}\right)) & \stackrel{\text{def}}{=} & \left(\begin{array}{l}\texttt{match}\ \mathsf{prefix}(\ell, e_0)\ \texttt{with} \\ \mid\ \mathbf{X}_1\ p_1 \to \mathsf{prefix}(\ell, e_1)\ \mid\ \cdots \\ \mid\ \mathbf{X}_n\ p_n \to \mathsf{prefix}(\ell, e_n)\end{array}\right) \\
\mathsf{prefix}(\ell, (e_1\ e_2)) & \stackrel{\text{def}}{=} & (\mathsf{prefix}(\ell, e_1)\ \mathsf{prefix}(\ell, e_2)) \\
\mathsf{prefix}(\ell, (e_1, e_2)) & \stackrel{\text{def}}{=} & (\mathsf{prefix}(\ell, e_1), \mathsf{prefix}(\ell, e_2)) \\
\mathsf{prefix}(\ell, (e_1 \| e_2)) & \stackrel{\text{def}}{=} & (\mathsf{prefix}(\ell, e_1) \| \mathsf{prefix}(\ell, e_2)) \\
\mathsf{prefix}(\ell, \left[\begin{array}{l}\texttt{reg}_{\ell'}\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e) \\ \texttt{init}\ e_0\end{array}\right]) & \stackrel{\text{def}}{=} & \left[\begin{array}{l}\texttt{reg}_{\ell \cdot \ell'}\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to \mathsf{prefix}(\ell, e)) \\ \texttt{init}\ \mathsf{prefix}(\ell, e_0)\end{array}\right] \\
\mathsf{prefix}(\ell, \texttt{exec}_{\ell'}\ e\ \texttt{default}\ e_0\ \texttt{reset}\ e_2) & \stackrel{\text{def}}{=} & \left[\begin{array}{l}\texttt{exec}_{\ell \cdot \ell'}\ \mathsf{prefix}(\ell, e)\ \texttt{default}\ \mathsf{prefix}(\ell, e_0) \\ \texttt{reset}\ \mathsf{prefix}(\ell, e_2)\end{array}\right] \\
\mathsf{prefix}(\ell, \texttt{halt}\ e) & \stackrel{\text{def}}{=} & \texttt{halt}\ \mathsf{prefix}(\ell, e)
\end{array}
$$

FIGURE B.26 Renommage préfixe pour la substitution sans-capture non-standard d'ECLAT

$$
\begin{array}{rcl}
e[x \mapsto v] & \stackrel{\text{def}}{=} & e[x \overset{\varepsilon}{\mapsto} v] \qquad \textbf{note} : \varepsilon \text{ est le préfixe vide} \\
\\
x[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \mathsf{prefix}(\ell, v) \\
y[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & y \text{ si } y \neq x \\
c[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & c \\
\mathbf{X}[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \mathbf{X} \\
 & & \textbf{note} : \mathbf{X} \text{ et } x \text{ ne sont pas} \\
 & & \text{dans le même espace de noms} \\
(\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e \quad \text{si } x \in \textbf{vars-of-pat}(p) \\
(\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e[x \overset{(\ell\cdot 0)}{\longmapsto} v] \\
 & & \quad \text{si } x \notin \textbf{vars-of-pat}(p) \\
(\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e))[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e) \\
 & & \quad \text{si } x \in (\{f\} \cup \textbf{vars-of-pat}(p)) \\
(\texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e))[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{fix}\ f\ (\texttt{fun}\ (p{:}\tau)\ :\ \mathring{\tau}' \to e[x \overset{(\ell\cdot 0)}{\longmapsto} v]) \\
 & & \quad \text{si } x \notin (\{f\} \cup \textbf{vars-of-pat}(p)) \\
(\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{let}\ p{:}\tau = e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v]\ \texttt{in}\ e_2 \quad \text{si } x \in \textbf{vars-of-pat}(p) \\
(\texttt{let}\ p{:}\tau = e_1\ \texttt{in}\ e_2)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{let}\ p{:}\tau = e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v]\ \texttt{in}\ e_2[x \overset{(\ell\cdot 1)}{\longmapsto} v] \\
 & & \quad \text{si } x \notin \textbf{vars-of-pat}(p) \\
(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \left[\begin{array}{l} \texttt{if}\ e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v]\ \texttt{then}\ e_2[x \overset{(\ell\cdot 1)}{\longmapsto} v] \\ \texttt{else}\ e_1[x \overset{(\ell\cdot 2)}{\longmapsto} v] \end{array}\right. \\
\left(\begin{array}{l} \texttt{match}\ e_0\ \texttt{with} \\ \mid \mathbf{X}_1\ p_1 \to e_1\ \mid\ \cdots \\ \mid \mathbf{X}_n\ p_n \to e_n \end{array}\right)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \texttt{match}\ e_0[x \overset{(\ell\cdot 0)}{\longmapsto} v]\ \texttt{with} \\ \mid \mathbf{X}_1\ p_1 \to e_1[x \overset{(\ell\cdot 1)}{\longmapsto} v]\ \mid\ \cdots \\ \mid \mathbf{X}_n\ p_n \to e_n[x \overset{(\ell\cdot n)}{\longmapsto} v] \end{array}\right) \\
\left(\begin{array}{l} \texttt{match}\ e_0\ \texttt{with} \\ \mid c_1 \to e_1\ \mid\ \cdots \\ \mid c_n \to e_n \\ \mid p \to e' \end{array}\right)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \texttt{match}\ e_0[x \overset{(\ell\cdot 0)}{\longmapsto} v]\ \texttt{with} \\ \mid c_1 \to e_1[x \overset{(\ell\cdot 1)}{\longmapsto} v]\ \mid\ \cdots \\ \mid c_n \to e_n[x \overset{(\ell\cdot n)}{\longmapsto\!\!\mapsto} v] \\ \mid p \to e'[x \overset{(\ell\cdot n+1)}{\longmapsto} v] \end{array}\right) \\
(e_1\ e_2)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & (e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v])\ (e_2[x \overset{(\ell\cdot 1)}{\longmapsto} v]) \\
(e_1, e_2)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & (e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v],\ e_2[x \overset{(\ell\cdot 1)}{\longmapsto} v]) \\
(e_1 \| e_2)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & (e_1[x \overset{(\ell\cdot 0)}{\longmapsto} v] \| e_2[x \overset{(\ell\cdot 1)}{\longmapsto} v]) \\
\left(\begin{array}{l} \texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e_0) \\ \texttt{init}\ e_1 \end{array}\right)[x \overset{\ell'}{\mapsto} v] & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e_0[x \overset{(\ell'\cdot 0)}{\longmapsto} v]) \\ \texttt{init}\ e_1[x \overset{(\ell'\cdot 1)}{\longmapsto} v] \end{array}\right) \\
 & & \quad \text{si } x \notin \textbf{vars-of-pat}(p) \\
\left(\begin{array}{l} \texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e_0) \\ \texttt{init}\ e_1 \end{array}\right)[x \overset{\ell'}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{reg}_\ell\ (\texttt{fun}\ (p{:}\mathring{\tau})\ :\ \mathring{\tau} \to e_0)\ \texttt{init}\ e_1[x \overset{(\ell'\cdot 1)}{\longmapsto} v] \\
 & & \quad \text{si } x \in \textbf{vars-of-pat}(p) \\
\left(\begin{array}{l} \texttt{exec}_\ell\ e_1\ \texttt{default}\ e_2 \\ \texttt{reset}\ e_3 \end{array}\right)[x \overset{\ell'}{\mapsto} v] & \stackrel{\text{def}}{=} & \left[\begin{array}{l} \texttt{exec}_\ell\ e_1[x \overset{(\ell'\cdot 0)}{\longmapsto} v]\ \texttt{default}\ e_2[x \overset{(\ell'\cdot 1)}{\longmapsto} v] \\ \texttt{reset}\ e_3[x \overset{(\ell'\cdot 2)}{\longmapsto} v] \end{array}\right. \\
(\texttt{halt}\ e)[x \overset{\ell}{\mapsto} v] & \stackrel{\text{def}}{=} & \texttt{halt}\ (e[x \overset{(\ell\cdot 0)}{\longmapsto} v])
\end{array}
$$

FIGURE B.27 Substitution sans-capture non-standard d'ECLAT

B.4 Sémantique à réductions

Cette section redéfinit le jugement de réduction $\hookrightarrow$ de la section précédente (ou *sémantique comportementale*) en utilisant un autre style de sémantique, ou *sémantique à réductions*[2]. Le nouveau jugement de réduction est noté $e/\mu, S \rightsquigarrow e'/\mu', S'$. Nous conjecturons que la sémantique comportementale est équivalente à la sémantique à réductions : pour toute mémoire μ, tout environnement simulation S et tout expression e, il existe une expression e', une mémoire μ' et un environnement de simulation S tels que $(e/\mu, S \hookrightarrow e'/\mu', S') \equiv (e/\mu, S \rightsquigarrow e'/\mu', S')$.

La construction `pause` n'est plus une construction dérivée : elle marque la fin de chaque cycle dans le processus d'évaluation.

$$e ::= \cdots \mid \texttt{pause}\ e$$

La relation $e/\mu, S \longrightarrow e'/\mu', S'$ se lit : "l'expression e avec la mémoire μ et l'environnement de simulation S se réduit instantanément en e' dans la mémoire μ' et l'environnement de simulation S'". On note $\longrightarrow^\star$ la fermeture réflexive et transitive de la relation $\rightarrow$: $e/\mu, S \longrightarrow^\star e'/\mu', S'$ si seulement s'il existe $n \geq 0$ et $n+1$ configurations $e_/\mu_i, S_{i_{i\in\{0,\cdots n\}}}$ telles que $e/\mu, S = e_0/\mu_0, S_0 \rightarrow e_1/\mu_1, S_1 \rightarrow \cdots e_n/\mu_n, S_n = e'/\mu', S$.

$$\textsc{Tick-Pause}\ \frac{e/\mu, S \longrightarrow^\star \texttt{pause}\ e'/\mu', S'}{e/\mu, S \rightsquigarrow e'/\mu', S'} \qquad \textsc{Tick-Val}\ \frac{e/\mu, S \longrightarrow^\star v/\mu', S'}{e/\mu, S \rightsquigarrow v/\mu', S'}$$

Un contexte d'évaluation F est une expression avec un trou $\square$ dans lequel insérer une expression. L'insertion de l'expression e dans le trou du contexte F est notée $F[e]$. Les contextes d'évaluation d'Eclat sont définis sur la figure B.28.

contexte d'évaluation

$$\begin{aligned} F ::= {} & (\square, e) \mid (v, \square) \mid \texttt{let}\ p = \square\ \texttt{in}\ e \\ & \mid \square\ e \mid v\ \square \mid \texttt{if}\ \square\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid \texttt{match}\ \square\ \texttt{with} \mid c_1 \rightarrow e_1 \cdots \mid c_n \rightarrow e_n \mid _ \rightarrow e \\ & \mid \texttt{match}\ \square\ \texttt{with} \mid \mathbf{X}_1\ p_1 \rightarrow e_1 \cdots \mid \mathbf{X}_n\ p_n \rightarrow e_n \\ & \mid (\square \| e) \mid (v \| \square) \mid (\texttt{pause}\ e \| \square) \\ & \mid \texttt{exec}_\ell\ e\ \texttt{default}\ e\ \texttt{reset}\ \square \\ & \mid \texttt{make<}\theta\texttt{>}\ \square \end{aligned}$$

FIGURE B.28 Sémantique à réductions (contextes d'évaluation)

2. Cette sémantique a été présentée au symposium HLPP 24 en se focalisant sur le parallélisme avec mémoire partagée ; les constructions `reg`, `exec` et `match/with` n'avaient pas été présentées.

La relation $\rightarrow$ est définie par un ensemble de règles de réduction données sur les figures B.29 et B.30. Nous distinguons les règles d'inférence B.29 (sur la partie haute de la figure B.29) et les règles de réécriture ou *réduction de tête* (sur la partie basse de la figure B.29 et sur la figure B.30). Les sémantiques à réductions sont classiquement définies avec une seule règle d'inférence : la *réduction sous un contexte d'évaluation* (CONTEXT). Nous avons toutefois ajouté trois règles d'inférence additionnelles qui visent à modéliser le plus simplement possible les constructions `reg` et `exec`.

$$\textsc{Context}\quad \frac{e/\mu, S \longrightarrow e'/\mu', S'}{F[e]/\mu, S \longrightarrow F[e']/\mu', S'}$$

$$\textsc{Reg-val}\quad \frac{\mu(\ell) = e' \qquad e'/\mu, S \longrightarrow^\star v/\mu', S'}{\texttt{reg}_\ell\ \texttt{(fun (}p\texttt{:}\mathring{\tau}\texttt{) : }\mathring{\tau} \rightarrow e\texttt{) init }e_0/\mu, S \longrightarrow v/\mu'[\ell \mapsto v], S'}$$

$$\textsc{Exec-val}\quad \frac{\mu(\ell) = e \qquad e/\mu, S \longrightarrow^\star v/\mu', S'}{\texttt{exec}_\ell\ e_1\ \texttt{default}\ e_0\ \texttt{reset false}/\mu, S \longrightarrow \texttt{(}v\texttt{,true)}/\mu'[\ell \mapsto \texttt{none}], S'}$$

$$\textsc{Exec-default}\quad \frac{\mu(\ell) = e \qquad e/\mu, S \longrightarrow^\star \texttt{pause }e'/\mu', S' \qquad e' \notin \mathbb{V}}{\texttt{exec}_\ell\ e_1\ \texttt{default}\ e_0\ \texttt{reset false}/\mu, S \longrightarrow \texttt{(}e_0\texttt{,false)}/\mu'[\ell \mapsto e'], S'}$$

LET	$\texttt{let }p\texttt{:}\tau = v\ \texttt{in }e/\mu, S \longrightarrow e[p \mapsto v]/\mu, S$
FUN-APP	$\texttt{(fun (}p\texttt{:}\tau\texttt{) } \rightarrow e\texttt{) }v/\mu, S \longrightarrow e[\tau \mapsto \mathbf{type}(v)][p \mapsto v]/\mu, S$
FIX-APP	$\overbrace{\texttt{(fix }f\texttt{ (fun (}p\texttt{:}\tau\texttt{) } \rightarrow e\texttt{))}}^{\phi}\ v/\mu, S$ $\longrightarrow \texttt{pause (}e[\tau \mapsto \mathbf{type}(v)][f \mapsto \phi][p \mapsto v]\texttt{)}/\mu, S$
OP-APP	$op\ v/\mu, S \longrightarrow \vartheta(op, v, \mu, S)$
IF-TRUE	$\texttt{if true then }e_1\ \texttt{else }e_2/\mu, S \longrightarrow e_1/\mu, S$
IF-FALSE	$\texttt{if false then }e_1\ \texttt{else }e_2/\mu, S \longrightarrow e_2/\mu, S$
PAUSE	$F[\texttt{pause }e]/\mu, S \longrightarrow \texttt{pause }F[e]/\mu, S \quad \text{si} \begin{cases} F \not\equiv \texttt{(}\square \| e'\texttt{)} \\ F \not\equiv \texttt{(pause }e' \| \square\texttt{)} \end{cases}$
PAUSE-PAR1	$\texttt{(pause }e \parallel v\texttt{)}/\mu, S \longrightarrow \texttt{pause (}e \| v\texttt{)}/\mu, S$
PAUSE-PAR2	$\texttt{(pause }e \parallel \texttt{pause }e'\texttt{)}/\mu, S \longrightarrow \texttt{pause (}e \| e'\texttt{)}/\mu, S$
PAR	$\texttt{(}v \| v'\texttt{)}/\mu, S \longrightarrow \texttt{(}v\texttt{,}v'\texttt{)}/\mu, S$

FIGURE B.29 Sémantique à réductions (première partie)

REG-INIT $\texttt{reg}_\ell\ \texttt{(fun (}p\texttt{:}\mathring{\tau}\texttt{) : }\mathring{\tau} \to e\texttt{) init }e_0/\mu, S$
$\longrightarrow \texttt{reg}_\ell\ \texttt{(fun }p\texttt{:}\mathring{\tau} \to e\texttt{) init }e_0/\mu[\ell \mapsto e], S$ si $\ell \notin \mathsf{dom}(\mu)$

EXEC-INIT $\texttt{exec}_\ell\ e_1\ \texttt{default}\ e_0\ \texttt{reset false}/\mu, S$
$\longrightarrow \texttt{exec}_\ell\ e_1\ \texttt{default}\ e_0\ \texttt{reset false}/\mu[\ell \mapsto e_1], S$
si $\ell \notin \mathsf{dom}(\mu)$ ou $\mu(\ell) = \mathsf{none}$

EXEC-RESET $\texttt{exec}_\ell\ e_1\ \texttt{default}\ e_0\ \texttt{reset true}/\mu, S$
$\longrightarrow \texttt{exec}_\ell\ e_1\ \texttt{default}\ e_0\ \texttt{reset false}/\mu[\ell \mapsto \mathsf{none}], S$

HALT $\texttt{halt}\ e/\mu, S \longrightarrow \texttt{pause (halt}\ e\texttt{)}/\mu, S$

MAKE $\texttt{make<}n\texttt{>}\ c/\mu, S \longrightarrow \underbrace{\texttt{pause(); }\cdots\texttt{pause(); }}_{n+1 \text{ fois } \texttt{pause}}\ \ell/\mu[\ell \mapsto \texttt{(false,(}\underbrace{c,\cdots c}_{n \text{ fois}}\texttt{))}], S$

CREATE $\texttt{create<}n\texttt{>()}/\mu, S \longrightarrow \texttt{(}d_1(\mathring{\tau})\texttt{,}\cdots\ d_n(\mathring{\tau})\texttt{)}$
où ℓ est une adresse fraîche
et les $d_i(\mathring{\tau})$ sont des valeurs non spécifiées de type $\mathring{\tau}$

MATCH-CONST $\texttt{match }c_i\ \texttt{with | }c_1 \to e_1\texttt{ | }\cdots\ c_n \to e_n\texttt{ | }p \to e/\mu, S \longrightarrow e_i/\mu, S$
si $i \in \{1, \cdots n\}$

MATCH-OTHERWISE $\texttt{(match }c\ \texttt{with | }c_1 \to e_1\texttt{ | }\cdots\ c_n \to e_n\texttt{ | }p \to e/\mu, S)$
$\longrightarrow e[p \mapsto c_i]/\mu, S$ si $c \notin \{c_1, \cdots c_n\}$

MATCH-CTOR $\texttt{match }\mathbf{X}_i\ v\ \texttt{with | }\mathbf{X}_1\ p_1 \to e_1\texttt{ | }\cdots\ \mathbf{X}_n\ p_n \to e_n/\mu, S$
$\longrightarrow e_i[p_i \mapsto v]/\mu, S$
note : $\mathbf{X}_i \in \{\ \mathbf{X}_1, \cdots\ \mathbf{X}_n\}$
car le programme est supposé bien typé.

FIGURE B.30 Sémantique à réductions (deuxième partie)

C Prise en main du compilateur ECLAT

Cette annexe présente les principales options et fonctionnalités du compilateur.

C.1 Appeler le compilateur

Le prototype ECLAT comprend un dossier `eclat-compiler` contenant le code source du compilateur et un dossier `target` contenant le code VHDL engendré.

Un *makefile* se situe dans le dossier `eclat-compiler`. L'exécution du *makefile* créé un exécutable `eclat`. Il prend en paramètres des noms de fichiers à compiler. Ces fichiers doivent être donnés par ordre d'inclusion : ils sont concaténés avant d'être compilés. Différentes options de compilation (commençant par un tiret) sont disponibles. Par exemple, l'option `-main=`*f* indique le nom *f* du point d'entrée du programme.

```
$ cd eclat-compiler
$ make
$ ./eclat -help
Usage:
  ./eclat file
  -main entry point (a function name)
  ...

$ ./eclat ../examples/counter.ecl -main=counter

vhdl code generated in ../target/counter.vhdl
testbench generated in ../target/tb_counter.vhdl for software RTL simulation using GHDL.
$
```

C.2 Mode intéractif

Si aucun fichier n'est précisé, le compilateur est en mode interactif. Nous pouvons alors entrer des déclarations ; le compilateur vérifie les types des déclarations au fur et à mesure

qu'elles sont définies. Le symbole `#q.` quitte le mode interactif et compile en VHDL les déclarations déjà saisies.

```
$ ./eclat
=== eclat toploop ===.
Enter phrases (separated by ';;') then compile (or run) with "#q."

> let sum x = reg (fun s -> s + x) init 0 ;;
val sum : forall '140  . (int<~z140> -{0}-> int<~z140>) | 0

> let main (i:int<16>) = sum i;;
val main : (int<16> -{0}-> int<16>) | 0

> #q.

vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$
```

C.3 Typage statique

```
$ ./eclat
> let f (x:bool) = x;;
val f : (bool -{0}-> bool)

> #q.
entry point *main* not set

> let main x = f x ;;
val main : (bool -{0}-> bool)

> #q.

vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$
```

C.4 Analyse de réactivité par typage statique

Le type de la fonction `main` doit être de la forme $\mathring{\tau} \xrightarrow{\mathbf{0}} \mathring{\tau}'$. Si ce n'est pas le cas, le programme n'est pas instantané et ne peut donc pas être utilisé pour modéliser un système réactif.

```
$ ./eclat
> let rec f x = x ;;
val f : forall '68  . ('b68 -{1}-> 'b68) | 0

> let main = f;;
val main : ('b200 -{1}-> 'b200) | 0

> #q.
Error: This program has type ('b212 -{1}-> 'b212). It is not reactive.
$
```

Pour accepter ce programme (qui n'est pas réactif), il faut ajouter l'option `-relax`.

```
$ ./eclat -relax
> let rec f x = x ;;
val f : forall '68  . ('b68 -{1}-> 'b68) | 0

> let main = f;;
val main : ('b200 -{1}-> 'b200) | 0

> #q.

vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$
```

Cependant, en pratique, cette option n'est pas utilisée : il suffit de placer systématiquement les calculs non instantanés sous un bloc **exec**.

```
$ ./eclat
> let rec f x = x ;;
val f : forall '68  . ('b68 -{1}-> 'b68) | 0

> let main x =
   let (o,rdy) = exec f x default 0 in
   o ;;
val main : forall '136  . (int<~z136> -{0}-> int<~z136>) | 0

> #q.

vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$
```

C.5 Simulation et visualisation de formes d'ondes

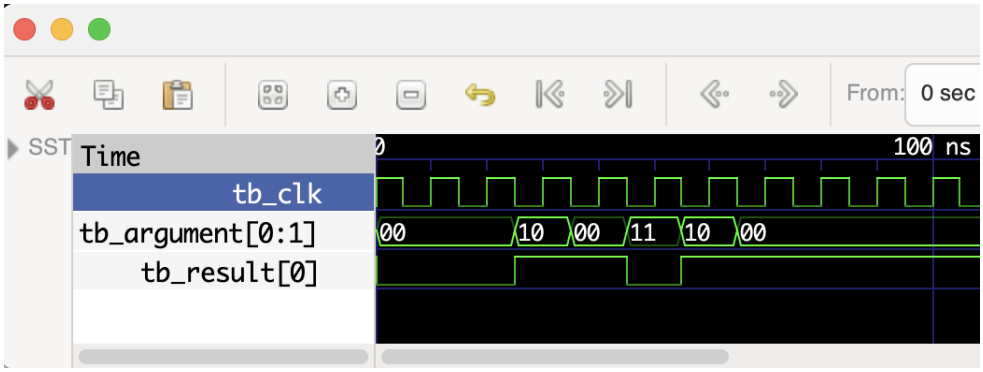
En compilant avec GHDL le code VHDL engendré, il est possible de visualiser précisément le comportement temporel du circuit en utilisant l'outil GTKWave.

```
$ ./eclat -arg="(false,false); (false,false); (true,false); \
                (false,false); (true,true); (true,false) ;(false,false)"
> let await (i,r) =
    reg (fun s -> (s or i) & (not r)) init false;;
val await : ((bool * bool) -{0}-> bool) | 0

> let main = await ;;
val main : ((bool * bool) -{0}-> bool) | 0

> #q.

vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$ make simul
cd ../target; make NS=2000
ghdl -a  runtime.vhdl
ghdl -a  main.vhdl
ghdl -a  tb_main.vhdl
ghdl -e  tb_main
ghdl -r  tb_main --vcd=tb.vcd --stop-time=2000ns
ghdl:info: simulation stopped by --stop-time @2us
$ gtkwave ../target/tb.vcd
GTKWave Analyzer v3.4.0 (w)1999-2022 BSI
```



C.6 Primitives de simulation

L'environnement d'exécution d'Eclat comporte des primitives de simulation (`print_int`, `print_string`, `print_newline`, `assert`) Pour faciliter la mise au point. Elles produisent des effets de bords au cours de la simulation (avec GHDL). Ces primitives ne sont pas synthétisables, mais les options `-noprint` et `-noassert` les retirent automatiquement du code engendré au cours de la compilation.

```
$ ./eclat -arg="()"
> let counter () =
    let o = reg (fun c -> c + 1) init 0 in print_int o; print_string ";"; o ;;
val counter : forall '68  . (unit -{0}-> int<~z68>) | 0

> let main = counter ;;
val main : (unit -{0}-> int<~z232>) | 0

> #q.
vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$ make simul NS=600
cd ../target; make NS=600
ghdl -a  runtime.vhdl
ghdl -a  main.vhdl
ghdl -a  tb_main.vhdl
ghdl -e  tb_main
ghdl -r  tb_main --vcd=tb.vcd --stop-time=600ns
1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;21;22;23;24;25;26;27; \
28;29;30;31;32;33;34;35;36;37;38;39;40;41;42;43;44;45;46;47;48;49;50;51; \
52;53;54;55;56;57;58;59;60; \
ghdl:info: simulation stopped by --stop-time @600ns
$
```

C.7 Interpréteur

L'option `-interp` indique que le programme source doit être interprété (au lieu d'être compilé en VHDL). C'est un outil expérimental pour "jouer" avec la sémantique d'Eclat.

```
$ ./eclat -arg="(false,false); (false,false); (true,false); (false,false);  \
                  (true,true); (true,false) ;(false,false)" -interp
> let await (i,r) =
    reg (fun s -> (s or i) & (not r)) init false ;;
val await : ((bool * bool) -{0}-> bool) | 0
```

```
> let main = await ;;
val main : ((bool * bool) -{0}-> bool) | 0

> #q.
cycle 0 : (false, false) --> false
cycle 1 : (false, false) --> false
cycle 2 : (true, false) --> true
cycle 3 : (false, false) --> true
cycle 4 : (true, true) --> false
cycle 5 : (true, false) --> true
cycle 6 : (false, false) --> true
cycle 7 : (false, false) --> true
cycle 8 : (false, false) --> true
cycle 9 : (false, false) --> true
cycle 10 : (false, false) --> true
$
```

L'option `-interp` peut être combinée avec l'option `-relax` ; cela signale alors les entrées perdues (non traitées).

```
$ ./eclat -relax -arg "0;3;1;2;6;4;1;10;11;12;-3;2" -interp -main=fibonacci -nb-it 15
> let fibonacci (n:int<32>) : int<32> =
  let rec fib (i, (a, b)) =
    if i = 0 then a
    else fib(i - 1, (b, a + b))
  in fib(n, (0, 1)) ;;
val fibonacci : (int<32> -{1}-> int<32>) | 0
> #q
cycle 0 : (0 : int<32>) (running)
cycle 1 : (3 : int<32>) --> (0 : int<32>)
cycle 2 : (1 : int<32>) (running)
cycle 3 : (2 : int<32>) (running)
cycle 4  (6 : int<32>) --> (1 : int<32>)
cycle 5 : (4 : int<32>) (running)
cycle 6 : (1 : int<32>) (running)
cycle 7 : (10 : int<32>) (running)
cycle 8 : (11 : int<32>) (running)
cycle 9 : (12 : int<32>) (running)
cycle 10 : (Int.neg) (3 : int<~z124>) --> (3 : int<32>)
$
```

C.8 Lien avec les entrées/sorties physiques

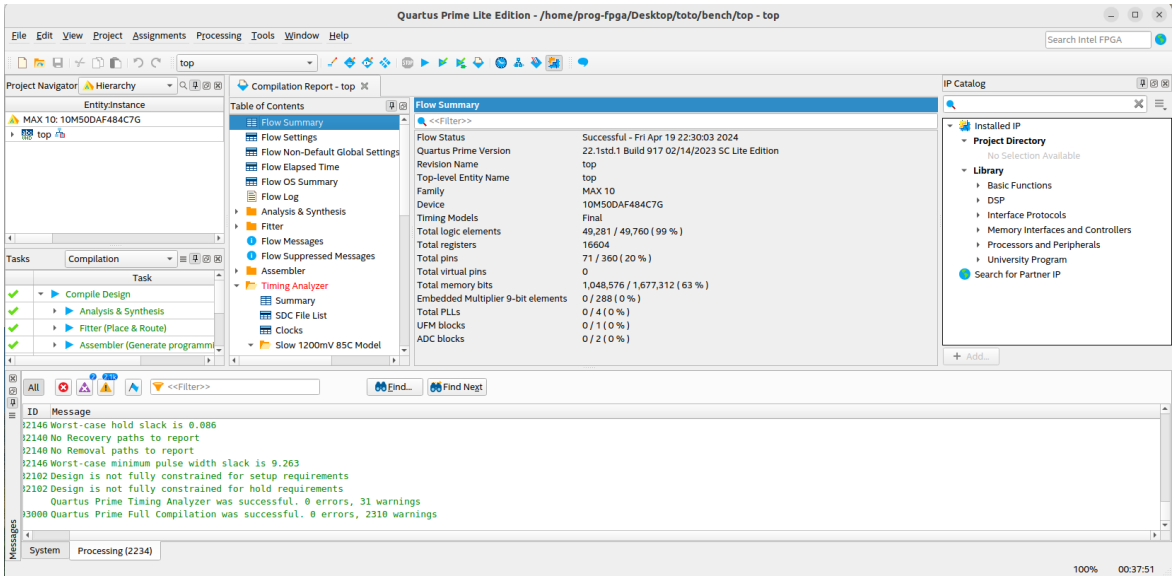
Le code VHDL engendré décrit un circuit à une entrée unique appelée `argument` et une sortie unique appelée `result` ; toutes deux sont réalisées sous forme de vecteurs de bits. Pour faire le lien avec les entrées/sorties physiques, il faut écrire un module principal appelé `top`. Ce module `top` est engendré par le compilateur en utilisant l'option `-top`, laquelle prend en argument une liste de la forme `"<entrées>|<sorties>"` où `<entrées>` et `<sorties>` sont des listes de déclarations *nom* :*nombre de bits*. L'option `-clk-top` définit le nom de l'horloge physique du FPGA cible.

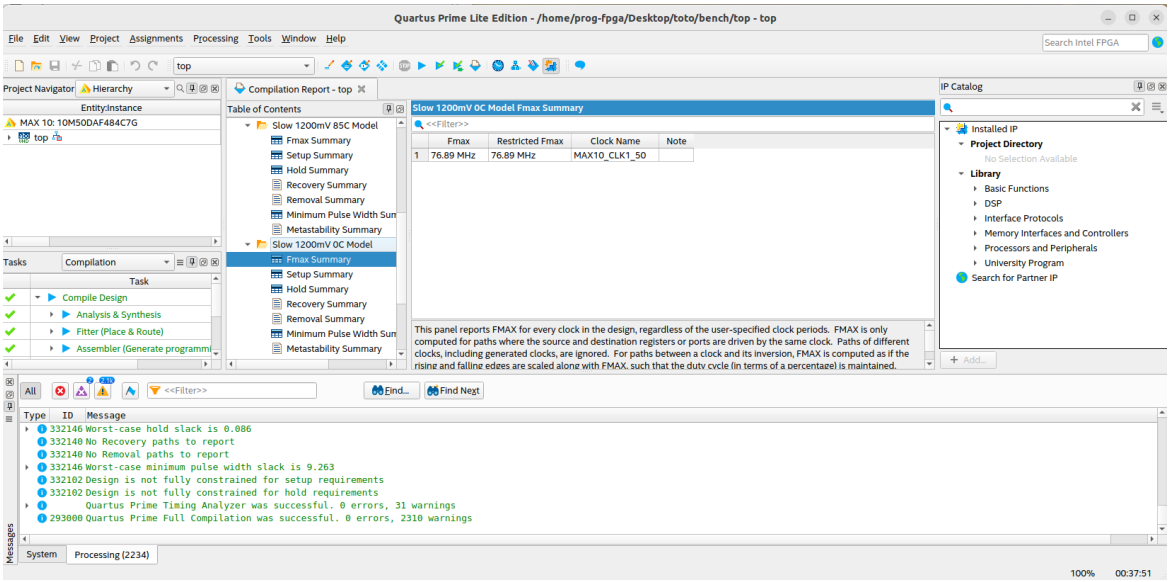
Des raccourcis sont fournis pour certaines cartes FPGA. Par exemple, l'option `-intel-max10` précise une configuration de base pour la carte *DE10 lite* :

```
-clk-top=MAX10_CLK1_50
-top="SW:10,KEY:2|LEDR:10,HEX0:8,HEX1:8,HEX2:8,HEX3:8,HEX4:8,HEX5:8"
```

L'option `-xilinx-zybo` précise une configuration pour la carte Zybo de XILINX. L'option `-yosys-ecp5` précise une configuration pour le FPGA ECP5 (*open hardware*). Des fichiers de configuration additionnels sont situés dans le dossier `target/synth`.

Par exemple, pour cibler la carte *DE10 lite*, il faut ouvrir le logiciel QUARTUS d'Intel, charger les fichiers VHDL engendrés par le compilateur ECLAT et le fichier de configuration `target/synth/intel-max10/top.sdc`. Les captures d'écran ci-dessous montrent les résultats de synthèse de la troisième version du jeu de la vie, présentée à la section 7.7.3 avec une grille de 512 lignes de 2048 cellules réalisée par un tableau de vecteur de booléens. Le circuit exploite autant que possible le compromis temps-espace : 99% des éléments logiques disponibles sur le FPGA sont utilisés ainsi que 63% des blocs de mémoire RAM. La fréquence maximale du circuit est systématiquement calculée par QUARTUS. Sur cet exemple, QUARTUS indique que la fréquence maximale est de 76 MHz ; le circuit peut donc être synchronisé de façon sûre avec l'horloge physique de 50 MHz de la carte FPGA.





C.9 Simulation en OCaml

Les circuits engendrés par le compilateur ECLAT peuvent aussi être traduit en OCAML (à partir du langage intermédiaire MINI-HDL), ce qui facilite le développement d'applications ECLAT. La traduction actuelle supporte une large partie du langage, y compris l'opérateur parallèle, les constructions **reg** et **exec** ou encore les appels de fonctions externes.

```
$ ./eclat -ocaml -arg="false;false;false;false;false;true;false;false;  \
          true;false;false;false;false;false;false;false;true;false"
>  let fibonacci (n:int<32>) : int<32> =
  let rec fib (i, (a, b)) =
    if i = 0 then a
    else fib(i - 1, (b, a + b))
  in fib(n, (0, 1)) ;;
val fibonacci : (int<32> -{1}-> int<32>) | 0
> let main (suspend) =
    if suspend then print_string "<>" else
    let (o,rdy) = exec fibonacci 3 default 0 in
    print_int o ;;
val main : (bool -{0}-> unit) | 0
> #q.
ocaml code generated in ../target/ml/main_step.ml
$ cd ../target/ml
$ ocamlopt runtime.ml main_step.ml
$ ./a.out -run 30
00002<>00<>00200002<>000020000200
$
```

Bibliographie

[1] Abhinav Agarwal *et al.* : *Comparison of high level design methodologies for algorithmic IPs : Bluespec and C-based synthesis.* Thèse de doctorat, Massachusetts Institute of Technology, 2009. 4

[2] Vincent A Akpan : Hard and soft embedded FPGA processor systems design : Design considerations and performance comparisons. *International Journal of Engineering and Technology (IJET)*, 3[11]:1000–1020, 2013. 5

[3] Abdallah Al Zain, Wim Vanderbauwhede et Greg Michaelson : Hume on FPGA. *In Trends in Functional Programming (TFP '10)*, 2010. 5

[4] Marco Aldinucci et Marco Danelutto : Skeleton-based parallel programming : Functional and parallel semantics in a single shot. *Computer Languages, Systems & Structures (CLSS)*, 33[3-4]:179–192, 2007. 192

[5] Michel Allemand, François Bodin, Apostolos Kountouris, Paul Le Guernic, Jean-Christophe Le Lann, André Seznec et Christophe Wolinski : A synchronous approach for hardware design. Rapport technique, Institut de recherche en informatique et systèmes aléatoires (IRISA), 1994. Publication interne n°1131 Octobre 1994 – Projet CAPS EP-ATR. 191

[6] Roberto M Amadio et Frédéric Dabrowski : Feasible reactivity in a synchronous π-calculus. *In Principles and Practice of Declarative Programming (PPDP '07)*, pages 221–230. ACM, 2007. 201

[7] Hafiz Muhammad Amjad, Jianwei Niu, Kai Hu, Naveed Akram et Loïc Besnard : Verilog code generation scheme from Signal language. *In International Bhurban Conference on Applied Sciences and Technology (IBCAST '19)*, pages 457–462. IEEE, 2019. 9

[8] David Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge Ortiz, Ed Komp et Peter Ashenden : Programming models for hybrid FPGA-CPU computational components : a missing link. *IEEE micro*, 24[04]:42–53, 2004. 5, 15

[9] Laurent Arditi, Gerard Berry et Michael Kishinevsky : Late Design Changes (ECOs) for Sequentially Optimized High-Level Esterel Designs. *Designing Correct Circuits (DCC '04)*, 2004. 9

[10] G Arnout : C for system level design. *In Design, Automation and Test in Europe (DATE '99)*, pages 384–386. IEEE, 1999. 1

[11] Edward A. Ashcroft et William W. Wadge : Lucid, a nonprocedural language with iteration. *Communications of the ACM (CACM)*, 20[7]:519–526, 1977. 7

[12] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink et Marco Gerards : Cλash : Structural descriptions of synchronous hardware using Haskell. *In Euromicro Conference on Digital System Design (DSD) : Architectures, Methods and Tools*, pages 714–721. IEEE, 2010. 6, 10, 12, 91, 188, 193

[13] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek et Krste Asanović : Chisel : constructing hardware in a Scala embedded language. *In Design Automation Conference (DAC '12)*, pages 1212–1221. IEEE, 2012. 12, 186, 193

[14] John Backus : Can programming be liberated from the von Neumann style ? A functional style and its algebra of programs. *Communications of the ACM (CACM)*, 21[8]:613–641, 1978. 7

[15] Rabia Bakhteri, Julian Cheng et Alex Semmelhack : Design and implementation of cellular automata on FPGA for hardware acceleration. *Procedia Computer Science*, 171:1999–2007, 2020. 165, 166, 168

[16] Mario R Barbacci : A comparison of register transfer languages for describing computers and digital systems. *Transactions on Computers*, 100[2]:137–150, 1975. 2

[17] Martha Barker, Stephen A Edwards et Martha A Kim : Synthesized in-bramgarbage collection for accelerators with immutable memory. *In Field-Programmable Logic and Applications (FPL '22)*, pages 47–53. IEEE, 2022. 5

[18] Mohammed Belhadj : *Conception d'architectures en utilisant SIGNAL et VHDL*. Thèse de doctorat, Rennes 1, 1994. 9

[19] Bruce Belson, Jason Holdsworth, Wei Xiang et Bronson Philippa : A survey of asynchronous programming using coroutines in the internet of things and embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18[3]:1–21, 2019. 28

[20] Imed Eddine Bennour : Formal verification of timed synchronous dataflow graphs using lustre. *Journal of Logical and Algebraic Methods in Programming*, 121:100678, 2021. 199

[21] 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. 8, 194

[22] Etienne Bergeron, Xavier Saint-Mleux, Marc Feeley et Jean-Pierre David : High level synthesis for data-driven applications. *In Rapid System Prototyping (RSP '05)*, pages 54–60. IEEE, 2005. 5

[23] Cinzia Bernardeschi, Luca Cassano et Andrea Domenici : SRAM-based FPGA systems for safety-critical applications : A survey on design standards and proposed methodologies. *Journal of Computer Science and Technology (JCS&T)*, 30:373–390, 2015. 1

[24] Gérard Berry : Real time programming : Special purpose or general purpose languages. Rapport technique n° 1065, INRIA, 1989. 9

[25] Gérard Berry : A hardware implementation of Pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Science*, 17[1]:95–130, 1992. 9, 190, 191

[26] Gérard Berry : Esterel and Jazz : Two synchronous languages for circuit design. *In Correct Hardware Design and Verification Methods (CHARME '99)*, page 1, 1999. 186

[27] Gérard Berry : *Penser, modéliser et maîtriser le calcul informatique.* Collège de France : Fayard, 2009. 2

[28] Gérard Berry : *Esterel v7 for the Hardware Designer, Draft 1.1*, 2018. `https://www.college-de-france.fr/sites/default/files/documents/gerard-berry/UPL8975876032420233341_Berry_Esterelv7_Draft_Book.pdf` (date d'accès : 30/08/2024). 50

[29] Gérard Berry et Georges Gonthier : The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19[2]:87–152, 1992. 3, 8, 61

[30] Gérard Berry, S Ramesh et RK Shyamasundar : Communicating reactive processes. *In Principles of Programming Languages (POPL 93')*, pages 85–98. ACM, 1993. 9, 28

[31] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni et Giorgio Buttazzo : A framework for supporting real-time applications on dynamic reconfigurable FPGAs. *In Real-Time Systems Symposium (RTSS '16)*, pages 1–12. IEEE, 2016. 14

[32] Per Bjesse, Koen Claessen, Mary Sheeran et Satnam Singh : Lava : hardware design in Haskell. *Acm Sigplan Notices*, 34[1]:174–184, 1998. 186, 193

[33] Jörg Bormann, Jörg Lohse, Michael Payer et Gerd Venzl : Model checking in industrial hardware design. *In Design Automation Conference (DAC '95)*, pages 298–303, 1995. 199

[34] Lilian Bossuet : *Exploration de l'espace de conception des architectures reconfigurables.* Thèse de doctorat, Université de Bretagne Sud, 2004. 4, 185

[35] Timothy Bourke et Arcot Sowmya : A timing model for synchronous language implementations in Simulink. *In Embedded software (EMSOFT '06)*, pages 93–101. ACM, 2006. 8, 191

[36] Frédéric Boussinot : Reactive C : An extension of C to program reactive systems. *Software : Practice and Experience (SPE)*, 21[4]:401–428, 1991. 143

[37] Andrew Boutros et Vaughn Betz : FPGA architecture : Principles and progression. *IEEE Circuits and Systems Magazine*, 21[2]:4–29, 2021. 1

[38] Matthew Bowen *et al.* : *Handel-C Language Reference Manual Version 2.1*, 2001. 190

[39] Edwin C Brady, James McKinna et Kevin Hammond : Constructing correct circuits : verification of functional aspects of hardware specifications with dependent types. *Trends in Functional Programming (TFP '07)*, 8:159–176, 2007. 58

[40] William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow et Mae Milano : Better defunctionalization through lambda set specialization. *In Programming Language Design and Implementation (PLDI '23)*, pages 977–1000, 2023. 124

[41] Jens Brandt, Klaus Schneider et Adrian Willenbücher : Using IP cores in synchronous languages. *In* Carsten Gremzow et Nico Moser, éditeurs : *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 97–106. Universitätsbibliothek Berlin, Germany, 2009. 9, 91

[42] Andrew Butterfield : *A denotational semantics for Handel-C*, pages 45–66. Springer, 2007. 189, 190

[43] Andrew Butterfield et Jim Woodcock : A "hardware compiler" semantics for Handel-C. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 161:73–90, 2006. 12

[44] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown et Tomasz Czajkowski : LegUp : high-level synthesis for FPGA-based processor/accelerator systems. *In Field-Programmable Gate Arrays (FPGA '11)*, pages 33–36. ACM SIGDA, 2011. 5

[45] Paul Caspi et Marc Pouzet : Synchronous kahn networks. *ACM SIGPLAN Notices*, 31[6]:226–238, 1996. 61

[46] David Castells-Rufas, Vinh Ngo, Juan Borrego-Carazo, Marc Codina, Carles Sanchez, Debora Gil et Jordi Carrabina : A survey of FPGA-based vision systems for autonomous cars. *IEEE Access*, 10:132525–132563, 2022. 1

[47] Katharina Ceesay-Seitz, Hamza Boukabache et Daniel Perrin : A functional verification methodology for highly parametrizable, continuously operating safety-critical FPGA designs : Applied to the CERN radiation monitoring electronics (CROME). *In Computer Safety, Reliability, and Security (SAFECOMP '20)*, pages 67–81. Springer, 2020. 1

[48] Gang Chen : A short historical survey of functional hardware languages. *International Scholarly Research Notices (ISRN)*, 2012[1]:271836, 2012. 3, 87, 183, 193

[49] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai et Zhiru Zhang : Allo : a programming model for composable accelerator design. *In Programming Language Design and Implementation (PLDI '24)*, pages 593–620. ACM SIGPLAN, 2024. 184, 187, 193

[50] Chris J Cheney : A nonrecursive list compacting algorithm. *Communications of the ACM (CACM)*, 13[11]:677–678, 1970. 177

[51] Jianyi Cheng, Lana Josipovic, George A Constantinides, Paolo Ienne et John Wickerson : Combining dynamic & static scheduling in high-level synthesis. *In Field-Programmable Gate Arrays (FPGA '20)*, pages 288–298. ACM SIGDA, 2020. 185

[52] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang et Jason Cong : Extending high-level synthesis for task-parallel programs. *In Field-Programmable Custom Computing Machines (FCCM '21)*, pages 204–213. IEEE, 2021. 3

[53] Young-kyu Choi, Peng Zhang, Peng Li et Jason Cong : HLScope+ : Fast and accurate performance estimation for FPGA HLS. *In International Conference on Computer-Aided Design (ICCAD '17)*, pages 691–698. IEEE, 2017. 4

[54] Radosław Cieszewski, Maciej Linczuk, Krzysztof Pozniak et Ryszard Romaniuk : Review of parallel computing methods and tools for FPGA technology. *In Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2013*, volume 8903, pages 596–608. SPIE, 2013. 1

[55] Albert Cohen, Léonard Gérard et Marc Pouzet : Programming parallelism with futures in Lustre. *In Embedded Software (EMSOFT '12)*, pages 197–206, 2012. 9, 28

[56] Jean-Louis Colaço, Grégoire Hamon et Marc Pouzet : Mixing Signals and Modes in Synchronous Data-flow Systems. *In Embedded Software (EMSOFT '6)*, pages 73–82. ACM, 2006. 209

[57] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet : A conservative extension of synchronous data-flow with state machines. *In Embedded Software (EMSOFT '05)*, pages 173–182. ACM, 2005. 9

[58] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet : Scade 6 : a formal language for embedded critical software development. *In Theoretical Aspects of Software Engineering (TASE '17)*, pages 1–11. IEEE, 2017. 9, 33, 191

[59] Jean-Louis Colaço, Baptiste Pauget et Marc Pouzet : Polymorphic types with polynomial sizes. *In Libraries, Languages and Compilers for Array Programming (ARRAY '23)*, pages 36–49, 2023. 75, 201

[60] Jason Cong : Scheduling and physical design. *In International Symposium on Physical Design (ISPD '24)*, pages 219–225. ACM, 2024. 184

[61] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers et Zhiru Zhang : FPGA HLS today : successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15[4]:1–42, 2022. 3

[62] John Conway *et al.* : The game of life. *Scientific American*, 223[4]:4, 1970. 162, 183

[63] Luis Damas et Robin Milner : Principal type-schemes for functional programs. *In Principles of Programming Languages (POPL '82)*, pages 207–212. ACM SIGPLAN-SIGACT, 1982. 28, 31

[64] Deeksha Dangwal, Georgios Tzimpragos et Timothy Sherwood : Agile hardware development and instrumentation with PyRTL. *IEEE Micro*, 40[4]:76–84, 2020. 186

[65] Olivier Danvy et Ulrik P Schultz : Lambda-lifting in quadratic time. *In Functional and Logic Programming (FLOPS '02)*, pages 134–151. Springer, 2002. 121

[66] Florent De Dinechin et Bogdan Pasca : Large multipliers with less DSP blocks. *In Field-Programmable Logic and Applications (FPL '09)*. IEEE, 2009. 13

[67] Al Dewey : VHSIC hardware description (VHDL) development program. *In Design Automation Conference (DAC '83)*, pages 625–628. IEEE, 1983. 1

[68] Rainer Dömer, Andreas Gerstlauer et Dongwan Shin : Cycle-accurate RTL modeling with multi-cycled and pipelined components. Rapport technique cecs-04-19, Center for Embedded Computer Systems, University of California, Irvine, 2004. 2

[69] Adam Donlin : Transaction level modeling : flows and use models. *In Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 75–80. IEEE/ACM/IFIP, 2004. 3

[70] Eleonora D'Arnese, Davide Conficconi, Marco D Santambrogio et Donatella Sciuto : *Reconfigurable architectures : The Shift From General Systems to Domain Specific Solutions*, pages 435–456. Springer, 2022. 3

[71] Stephen A Edwards : High-Level Synthesis from the synchronous language Esterel. *In International Workshop on Logic & Synthesis (IWLS '02)*, pages 401–406, 2002. 191

[72] Stephen A Edwards : The challenges of hardware synthesis from C-like languages. *In Design, Automation and Test in Europe (DATE '05)*, pages 66–67. IEEE, 2005. 3, 187

[73] Stephen A Edwards : Design and verification languages. *In EDA for IC system design, verification, and testing*, pages 1–28. CRC Press, 2018. 1

[74] Stephen A Edwards, Martha A Kim, Richard Townsend, Kuangya Zhai et Lianne Lairmore : The FHW Project : High-level hardware synthesis from Haskell programs. Rapport technique cucs–003–19, Department of Computer Science, Columbia University, 2019. 5

[75] John P Elliott : *Understanding behavioral synthesis : a practical guide to high-level design*. Springer Science & Business Media, 1999. 3

[76] Matthias Felleisen, Robert Bruce Findler et Matthew Flatt : *Semantics engineering with PLT Redex*. Mit Press, 2009. 198

[77] Matthias Felleisen et Daniel P Friedman : A syntactic theory of sequential state. *Theoretical Computer Science (TCS)*, 69[3]:243–287, 1989. 198

[78] Jean-Christophe Filliâtre : Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13:397–403, 2011. 199

[79] Jean-Christophe Filliâtre et Andrei Paskevich : Why3—where programs meet provers. *In European Symposium on Programming Languages and Systems (ESOP '13)*, pages 125–128. Springer, 2013. 199

[80] Spencer P Florence, Shu-Hung You, Jesse A Tov et Robert Bruce Findler : A calculus for Esterel : if can, can. if no can, no can. *In Principles of Programming Languages (POPL '19)*, pages 1–29. ACM, 2019. 198

[81] Fr-Xavier Fornari : *Optimisation du contrôle et implantation en circuits de programmes Esterel*. Thèse de doctorat, Ecole des Mines de Paris, 1995. 9

[82] Simon Frankau et Alan Mycroft : Stream processing hardware from functional language specifications. *In Hawaii International Conference on System Sciences (HICSS '03)*, pages 10–pp. IEEE, 2003. 189

[83] Max Fuchs et Michael Mendler : A functional semantics for delta-delay VHDL based on focus. *In Formal Semantics for VHDL*, pages 9–42. Springer, 1995. 101, 125

[84] Matthias Függer : *Computing at the border of abstractions : the power of timed, non-binary, distributed circuits*. Habilitation à diriger les recherches, ENS Paris-Saclay, France, 2022. 2

[85] Peter Gammie : Synchronous digital circuits as functional programs. *Computing Surveys (CSUR)*, 46[2]:1–27, 2013. 7

[86] Laure Gonnord, Ludovic Henrio, Lionel Morel et Gabriel Radanne : A survey on parallelism and determinism. *Computing Surveys (CSUR)*, 55[10]:1–28, 2023. 9, 192

[87] Sven Goossens, Jasper Kuijsten, Benny Akesson et Kees Goossens : A reconfigurable real-time SDRAM controller for mixed time-criticality systems. *In Hardware/software codesign and system synthesis (CODES+ ISSS '13)*, pages 1–10. IEEE, 2013. 15

[88] David J Greaves : Layering RTL, SAFL, Handel-C and Bluespec constructs on Chisel HCL. *In Formal Methods and Models for Codesign (MEMOCODE '15)*, pages 108–117. IEEE, 2015. 186, 187, 193

[89] Kim Grüttner, Henning Kleen, Frank Oppenheimer, Achim Rettberg et Wolfgang Nebel : Towards a synthesis semantics for systemc channels. *In Hardware/Software Codesign and System Synthesis (CODES+ISSS '10)*, pages 163–172, 2010. 4

[90] Adrien Guatto : *Projet GENCOD – Rapport d'étude sur la traduction de SCADE/-Lustre vers VHDL*, 2010. `https://www.irif.fr/~guatto/notes/rapport_gencod.pdf` (date d'accès : 30/08/2024). 191

[91] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal et Derek Dreyer : Melocoton : A program logic for verified interoperability between OCaml and C. *In Object Oriented Programming Systems Languages and Applications (OOPSLA '23)*, pages 716–744. ACM, 2023. 6

[92] Hipólito Guzman-Miranda, Luca Sterpone, Massimo Violante, Miguel A Aguirre et Manuel Gutierrez-Rizo : Coping with the obsolescence of safety-or mission-critical embedded systems using FPGAs. *Transactions on Industrial Electronics (TIE)*, 58[3]: 814–821, 2010. 4

[93] Nicholas Halbwachs, Paul Caspi, Pascal Raymond et Daniel Pilaud : The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79[9]: 1305–1320, 1991. 191

[94] Nicolas Halbwachs : Synchronous programming of reactive systems : A tutorial and commented bibliography. *In Computer Aided Verification (CAV '98)*, pages 1–16. Springer, 1998. 191

[95] Nicolas Halbwachs, Fabienne Lagnier et Pascal Raymond : Synchronous observers and the verification of reactive systems. *In Algebraic Methodology and Software Technology (AMAST '93)*, pages 83–96. Springer, 1994. 50

[96] Jerker Hammarberg et Simin Nadjm-Tehrani : Development of safety-critical reconfigurable hardware with Esterel. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 80:219–234, 2003. 3

[97] Kevin Hammond et Greg Michaelson : Hume : a domain-specific language for real-time embedded systems. *In Generative Programming and Component Engineering (GPCE '03)*, pages 37–56. Springer, 2003. 5

[98] Grégoire Hamon : *Calcul d'horloges et structures de contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* Thèse de doctorat, Paris 6, 2002. 191

[99] Sungsoo Han, Minseong Jang et Jeehoon Kang : ShakeFlow : Functional Hardware Description with Latency-Insensitive Interface Combinators. *In Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*, pages 702–717, 2023. 183, 186

[100] David Harel et Amir Pnueli : On the development of reactive systems. *In Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1984. 2

[101] Sebastien Heim, Xavier Dumas, Eric Bonnafous, Philippe Dhaussy, Ciprian Teodorov et Lise Leroux : Model checking of scade designed systems. *In Embedded Real Time Software and Systems (ERTS '16)*, 2016. 199

[102] Steven F Hoover : Timing-abstract circuit design in transaction-level Verilog. *In International Conference on Computer Design (ICCD '17)*, pages 525–532. IEEE, 2017. 4

[103] Sara Sadat Hoseininasab, Caroline Collange et Steven Derrien : Rapid prototyping of complex micro-architectures through high-level synthesis. *In Applied Reconfigurable Computing (ARC '23)*, pages 19–34. Springer, 2023. 4

[104] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen et Wen-Mei Hwu : Pylog : An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Transactions on Computers*, 70[12]:2015–2028, 2021. 5

[105] Paul Hudak et Joseph H Fasel : A gentle introduction to Haskell. *ACM Sigplan Notices*, 27[5]:1–52, 1992. 5

[106] R John M Hughes : Super-combinators a new implementation method for applicative languages. *In LISP and Functional Programming (LFP '82)*, pages 1–10, 1982. 6

[107] Charles Hymans : *Vérification de descriptions VHDL par interprétation abstraite.* Thèse de doctorat, Ecole Polytechnique X, 2004. 199

[108] Yasuaki Ito et Koji Nakano : A hardware-software cooperative approach for the exhaustive verification of the Collatz conjecture. *In International Symposium on Parallel and Distributed Processing with Applications (ISPA '09)*, pages 63–70. IEEE, 2009. 43

[109] Axel Jantsch et Ingo Sander : Models of computation and languages for embedded system design. *IEE Proceedings-Computers and Digital Techniques*, 152[2]:114–129, 2005. 2

[110] Steven D Johnson : *Synthesis of digital designs from recursion equations.* Thèse de doctorat, Indiana University, 1983. 7

[111] Steven D Johnson : Applicative programming and digital design. *In Principles of Programming Languages (POPL '84)*, pages 218–227. ACM, 1984. 7

[112] Steven D Johnson : Daisy, DSI and LiMP : Issues in architecture for suspending construction. Rapport technique n° 288, Computer Science Department, Indiana University, 1989. 7

[113] Thomas Johnsson : Lambda lifting : Transforming programs to recursive equations. *In Functional Programming Languages and Computer Architecture (FPCA '85)*, page 190–203. Springer, 1985. 121

[114] Lana Josipovic, Andrea Guerrieri et Paolo Ienne : Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 21[2]:97–118, 2021. 78

[115] Nachiket Kapre et Samuel Bayliss : Survey of domain-specific languages for FPGA computing. *In Field-Programmable Logic and Applications (FPL '16)*, pages 1–12. IEEE, 2016. 4

[116] Andrew Kennedy : Compiling with continuations, continued. *In International Conference on Functional programming (ICFP '07)*, pages 177–190, 2007. 121

[117] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis et Kunle Olukotun : Spatial : A language and compiler for application accelerators. *In Programming Language Design and Implementation (PLDI '18)*, pages 296–311. ACM, 2018. 187, 193

[118] J Kokila, N Ramasubramanian et S Indrajeet : A survey of hardware and software co-design issues for system on chip design. *In Advanced Computing and Communication Technologies (ICACCT '15)*, pages 41–49. Springer, 2016. 4

[119] Jean-Louis Krivine : A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation (HOSC)*, 20:199–207, 2007. 175

[120] Daniel Kroening et Sanjit A Seshia : Formal verification at higher levels of abstraction. *In International Conference on Computer-Aided Design (ICCAD '07)*, pages 572–578. IEEE, 2007. 2

[121] Jan Kuper et Rinse Wester : *N* queens on an FPGA : mathematics, programming, or both ? *In Communicating Process Architectures (CPA '14)*, pages 181–203. Open Channel Publishing Ltd, 2014. 155, 188

[122] Kimmo Kuusilinna, Timo Hämäläinen et Jukka Saarinen : Practical VHDL optimization for timing critical FPGA applications. *Microprocessors and Microsystems*, 23[8-9]:459–469, 1999. 3

[123] Sakari Lahti, Matti Rintala et Timo D Hämäläinen : Leveraging modern C++ in high-level synthesis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42[4]:1123–1132, 2022. 3

[124] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong et Zhiru Zhang : Programming and Synthesis for Software-defined FPGA Acceleration : Status and Future Prospects. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 14[4]:1–39, 2021. 1

[125] Edgar Lakis et Martin Schoeberl : An SDRAM controller for real-time systems. *In Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC '13)*, pages 1–8. IEEE, 2013. 200

[126] Paul Le Guernic, Albert Benveniste et Thierry Gautier : Signal : Un langage pour le traitement du signal. Rapport technique n° 206, INRIA, 1983. 8

[127] Edward A Lee et Thomas M Parks : Dataflow process networks. *Proceedings of the IEEE*, 83[5]:773–801, 1995. 4

[128] Xavier Leroy : The ZINC experiment : an economical implementation of the ML language. Rapport technique 117, INRIA, 1990. 175

[129] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan et Jérôme Vouillon : *The OCaml system release 5.1 : Documentation and user's manual*. Inria, 2023. 6

[130] Menachem Leuchter, Shmuel Tyszberowicz et Yishai A Feldman : Reusing Verilog Designs in the Synchronous Language Esterel. *Journal of Automation Mobile Robotics and Intelligent Systems (JAMRIS)*, 3:25–29, 2009. 9

[131] SM Loo, B Earl Wells, N Freije et J Kulick : Handel-C for rapid prototyping of VLSI coprocessors for real time systems. *In Southeastern Symposium on System Theory (SSST '02)*, pages 6–10. IEEE, 2002. 186, 189, 194

[132] Jason Luu, Conor McCullough, Sen Wang, Safeen Huda, Bo Yan, Charles Chiasson, Kenneth B Kent, Jason Anderson, Jonathan Rose et Vaughn Betz : On hard adders and carry chains in FPGAs. *In Field-Programmable Custom Computing Machines (FCCM '14)*, pages 52–59. IEEE, 2014. 14

[133] Martin Maas, Krste Asanović et John Kubiatowicz : A hardware accelerator for tracing garbage collection. *In International Symposium on Computer Architecture (ISCA'18)*, pages 138–151. IEEE, 2018. 5

[134] Mariem Makni, Mouna Baklouti, Smail Niar, Mohamed Wassim Jmal et Mohamed Abid : A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems. *In International Design & Test Symposium (IDT '16)*, pages 154–159. IEEE, 2016. 4

[135] Florence Maraninchi et Yann Rémond : Argos : an automaton-based synchronous language. *Computer languages*, 27[1-3]:61–92, 2001. 9, 191

[136] Peter Marwedel : *Embedded system design : embedded systems foundations of cyber-physical systems, and the internet of things*. Springer Nature, 2021. 2

[137] Vladimir Matyukha, Sergey Voloshchuk et Sergey Mosin : A configurable IP Core for calculating the integer square root for serial and parallel implementations in FPGA. *Electronics*, 11[15]:2335, 2022. 2

[138] George H Mealy : A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34[5]:1045–1079, 1955. 22

[139] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon et Edward A. Lee : High-performance deterministic concurrency using Lingua Franca. *ACM Transactions on Architecture and Code Optimization (TACO)*, 20[4]:1–29, 2023. 9

[140] Gabriele Mencagli, Massimo Torquati, Dalvan Griebler, Alessandra Fais et Marco Danelutto : General-purpose data stream processing on heterogeneous architectures with windflow. *Journal of Parallel and Distributed Computing*, 184:104782, 2024. 193

[141] John C Mitchell et Robert Harper : The essence of ML. *In Principles of Programming Languages (POPL '88)*, pages 28–46. ACM SIGPLAN-SIGACT, 1988. 6, 53, 188

[142] Romina Soledad Molina, Veronica Gil-Costa, María Liz Crespo et Giovanni Ramponi : High-level synthesis hardware design for FPGA-based accelerators : Models, methodologies, and frameworks. *IEEE Access*, 10:90429–90455, 2022. 3

[143] Edward F Moore : Gedanken-experiments on sequential machines. *Automata Studies : Annals of Mathematics Studies*, [34]:129, 1956. 22

[144] Greg Morrisett, Amal Ahmed et Matthew Fluet : L3 : A linear language with locations. *In Typed Lambda Calculi and Applications (TLCA '05)*, pages 293–307. Springer, 2005. 75

[145] Greg Morrisett, David Walker, Karl Crary et Neal Glew : From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21[3]:527–568, 1999. 75

[146] Alan Mycroft et Richard Sharp : A Statically Allocated Parallel Functional Language. *In International Colloquium on Automata, Languages, and Programming (ICALP 2000)*, pages 37–48. Springer, 2000. 12, 188

[147] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi *et al.* : A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35[10]:1591–1604, 2015. 193

[148] Matthew Naylor et Colin Runciman : The Reduceron : Widening the von Neumann bottleneck for graph reduction using an FPGA. *In Implementation and Application of Functional Languages (IFL '07)*, pages 129–146. Springer, 2007. 6

[149] Xavier Nicollin, Joseph Sifakis et Sergio Yovine : Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18[9]:794–804, 1992. 78

[150] Anja Niedermeier, Rinse Wester, Christiaan Baaij, Jan Kuper et Gerardus Johannes Maria Smit : Comparing CλaSH and VHDL by implementing a dataflow processor. *In Program for Research on Embedded Systems and Software (PROGRESS '10)*, pages 216–221. STW, 2010. 188

[151] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson et Zhiru Zhang : Predictable accelerator design with time-sensitive affine types. *In Programming Language Design and Implementation (PLDI '20)*, pages 393–407. ACM SIGPLAN, 2020. 4

[152] Rishiyur S Nikhil et Arvind : What is Bluespec ? *ACM SIGDA Newsletter*, 38[23]:1–1, 2008. 4

[153] John T O'Donnell : Overview of Hydra : a concurrent language for synchronous digital circuit design. *International Journal of Information*, 9[2]:249–264, 2006. 186

[154] Alberto Ottimo, Gabriele Mencagli et Marco Danelutto : Boosting general-purpose stream processing with reconfigurable hardware. *The Journal of Supercomputing*, pages 1–31, 2024. 192

[155] Ian Page et Wayne Luk : Compiling Occam into field-programmable gate arrays. *In Field-Programmable Logic and Applications (FPL '91)*, volume 15, pages 271–283. Abingdon EE&CS Books 15 Harcourt Way, Abingdon OX14 1NV, UK, 1991. 189

[156] Pascal Pampagnin et Ludovic Letellier : The GENCOD project : Automated generation of Hardware code for safety critical applications on FPGA targets. *In Embedded Real Time Software & Systems (ERTS2 '10)*, 2010. 191

[157] Preeti Ranjan Panda : SystemC : a modeling platform supporting multiple design abstractions. *In International Symposium on Systems Synthesis (ISSS '01)*, pages 75–80, 2001. 4

[158] Daniele Passaretti, Jan Moritz Joseph et Thilo Pionteck : Survey on FPGAs in medical radiology applications : Challenges, architectures and programming models. *In International Conference on Field-Programmable Technology (ICFPT '19)*, pages 279–282. IEEE, 2019. 1

[159] Artem Perepelitsyn, Vitaliy Kulanov et Inna Zarizenko : Method of QoS evaluation of FPGA as a service. *Radioelectronic and Computer Systems*, [4]:153–160, 2022. 2

[160] Basile Pesin : *Verified Compilation of a Synchronous Dataflow Language with State Machines*. Thèse de doctorat, Ecole Normale Supérieure de Paris, 2023. 9, 201

[161] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo et Luca P Carloni : Cosmos : Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 16[5s]:1–22, 2017. 4

[162] Daniel Pilaud, N Halbwachs et JA Plaice : Lustre : A declarative language for programming synchronous systems. *In Principles of Programming Languages (POPL '87)*, volume 178, page 188. ACM, 1987. 8, 10, 191, 194

[163] Hugo Pompougnac, Ulysse Beaugnon, Albert Cohen et Dumitru Potop-Butucaru : From SSA to synchronous concurrency and back. Rapport technique n° 9380, INRIA Sophia Antipolis-Méditerranée (France), 2020. 108

[164] Maxime Popoff, Romain Michon, Tanguy Risset, Yann Orlarey et Stéphane Letz : Towards an fpga-based compilation flow for ultra-low latency audio signal processing. *In SMC-22-Sound and Music Computing*, 2022. 1

[165] Dumitru Potop-Butucaru, Robert De Simone et Jean-Pierre Talpin : The synchronous hypothesis and synchronous languages. *The embedded systems handbook*, pages 1–21, 2005. 3, 8, 27

[166] Thomas B Preusser et Matthias R Engelhardt : Putting queens in carry chains, n° 27. *Journal of Signal Processing Systems*, 88:185–201, 2017. 154

[167] Christian Queinnec : *Lisp in small pieces*. Cambridge University Press, 2003. 34

[168] Craig Ramsay et Robert Stewart : Heron : Modern hardware graph reduction. *In Implementation and Application of Functional Languages (IFL '23)*, pages 1–12, 2023. 6

[169] Craig Ramsay et Robert James Stewart : Cloaca : A concurrent hardware garbage collector for non-strict functional languages. *In Haskell Symposium 2024*. ACM, 2024. 6

[170] Marino Rasamuel : *Modeling and design of neural network architectures for neural artificial-biological hybridization based on synchronous approach.* Thèse de doctorat, Université Côte d'Azur, 2023. 9, 192, 194

[171] Andy Ray, Benjamin Devlin, Fu Yong Quah et Rahul Yesantharao : Hardcaml MSM : A High-Performance Split CPU-FPGA Multi-Scalar Multiplication Engine. *In Field-Programmable Gate Arrays (FPGA '24)*, pages 33–39. ACM SIGDA, 2024. 186

[172] Pascal Raymond : Synchronous program verification with Lustre/Lesar. *Modeling and Verification of Real-Time Systems : Formalisms and Software Tools*, pages 171–206, 2008. 199

[173] Pascal Raymond, Yvan Roux et Erwan Jahier : Specifying and executing reactive scenarios with lutin. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 203[4]: 19–34, 2008. 199

[174] Francesco Restuccia et Alessandro Biondi : Time-predictable acceleration of deep neural networks on FPGA SoC platforms. *In Real-Time Systems Symposium (RTSS '21)*, pages 441–454. IEEE, 2021. 2

[175] Frédéric Rocheteau et Nicolas Halbwachs : Implementing reactive programs on circuits a hardware implementation of Lustre. *In Real-Time : Theory in Practice : REX Workshop Mook*, pages 195–208. Springer, 1992. 91, 191

[176] Fréderic Rocheteau et Nicolas Halbwachs : Pollux : A Lustre based hardware design environment. *In Algorithms and parallel VLSI architecture II*, page 335–346. Elsevier, 1992. 9

[177] Gabriel Rodriguez-Canal, Nick Brown, Tim Dykes, Jess Jones et Utz-Uwe Haus : Fortran high-level synthesis : Reducing the barriers to accelerating HPC codes on FPGAs. *In Field-Programmable Logic and Applications (FPL '23)*, pages 10–18. IEEE, 2023. 3

[178] Kaushik Roy et Jacob A Abraham : A novel approach to accurate timing verification using RTL descriptions. *In Design Automation Conference (DAC '89)*, pages 638–641. ACM/IEEE, 1989. 2

[179] Richard Rudell : Tutorial : Design of a logic synthesis system. *In Design Automation Conference (DAC '96)*, pages 191–196. IEEE, 1996. 1

[180] Xavier Saint-Mleux, Marc Feeley et Jean-Pierre David : SHard : A Scheme to Hardware Compiler. *In Scheme and Functional Programming (Scheme '06)*, 2006. 5, 10

[181] Salahuddin Savugathali, Muslim Mustapa, Mohammed Sharazel Razali et Fazrul Faiz Zakaria : Timing violation reduction in the fpga prototyped design using failed path fixes and time borrowing techniques. *Indonesian Journal of Electrical Engineering and Computer Science (IJEECS)*, 14[2]:628–636, 2019. 24

[182] Klaus Schneider : The synchronous programming language Quartz. Rapport technique n° 375, Department of Computer Science, University of Kaiserslautern, Germany, 2009. 191

[183] Shimon Schocken : Nand to Tetris : Building a modern computer system from first principles. *Communications of the ACM*, 67[5]:76–85, 2024. 197

[184] Alexander Schulz-Rosengarten, Steven Smyth, Reinhard von Hanxleden et Michael Mendler : On reconciling concurrency, sequentiality and determinacy for reactive systems—A sequentially constructive circuit semantics for Esterel. *In Application of Concurrency to System Design (ACSD '18)*, pages 95–104. IEEE, 2018. 9

[185] Luc Séméria et Giovanni De Micheli : SpC : synthesis of pointers in C : application of pointer analysis to the behavioral synthesis from C. *In International Conference on Computer-Aided Design (ICCAD '98)*, pages 340–346, 1998. 4

[186] Jocelyn Sérot : *Les circuits FPGA et le langage VHDL, une introduction pour les programmeurs et par l'exemple.* Editions Ellipses, 2019. 78

[187] Jocelyn Sérot, François Berry et Cédric Bourrasset : High-level dataflow programming for real-time image processing on smart cameras. *Journal of Real-Time Image Processing (JRTIP)*, 12[4]:635–647, 2016. 4

[188] Jocelyn Sérot et Emmanuel Chailloux : OCaml sur circuit FPGA. *In Journées Francophones des Langages Applicatifs (JFLA '21)*, page 72, 2021. 6

[189] Jocelyn Sérot et Greg Michaelson : Harnessing parallelism in FPGAs using the Hume language. *In Functional High-performance Computing (FHPNC '12)*, pages 27–36, 2012. 5

[190] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist et Miodrag Milanovic : Yosys+ nextpnr : an open source framework from Verilog to bitstream for commercial FPGAs. *In Field-Programmable Custom Computing Machines (FCCM '19)*, pages 1–4. IEEE, 2019. 197

[191] Richard Sharp : Functional design using behavioural and structural components. *In Formal Methods in Computer-Aided Design (FMCAD '02)*, pages 324–341. Springer, 2002. 92, 189

[192] Richard Sharp : *Higher-level hardware synthesis*, volume 2963. Springer, 2004. 189

[193] Richard Sharp et Alan Mycroft : A higher-level language for hardware synthesis. *In Correct Hardware Design and Verification Methods (CHARME '01)*, pages 228–243. Springer, 2001. 189

[194] Richard Sharp et Alan Mycroft : Soft scheduling for hardware. *In Static Analysis (SAS '01)*, pages 57–72. Springer, 2001. 189

[195] Mary Sheeran : *μFP, an algebraic VLSI design language*. Thèse de doctorat, Oxford University, 1983. 7, 28

[196] Mary Sheeran : μFP, a language for VLSI design. *In LISP and Functional Programming (LFP '84)*, pages 104–112, 1984. 7, 91, 183

[197] Mary Sheeran : Hardware design and functional programming. *Journal of Universal Computer Science (J.UCS)*, 2[7]:1135–1158, 2005. 8, 186

[198] Michael Simkin : The number of n-queens configurations. *Advances in Mathematics*, 427:109127, 2023. 154

[199] Fabien Siron, Dumitru Potop-Butucaru, Robert De Simone, Damien Chabrol et Amira Methni : The synchronous logical execution time paradigm. *In Embedded Real Time Systems (ERTS '22)*, 2022. 28

[200] Jian Sun : Pulse-width modulation. *In Dynamics and Control of Switched Electronic Systems : Advanced Perspectives for Modeling, Simulation and Control of Power Converters*, pages 25–61. Springer, 2012. 149

[201] Loïc Sylvestre, Emmanuel Chailloux et Jocelyn Sérot : Accelerating OCaml programs on FPGA. *International Journal of Parallel Programming (IJPP)*, 51[2]:186–207, 2023. 6, 10, 11, 175, 181

[202] Loïc Sylvestre, Jocelyn Sérot et Emmanuel Chailloux : Macle : un langage dédié à l'accélération de programmes OCaml sur circuits FPGA. *In Journées Francophones des Langages Applicatifs (JFLA '22)*, 2022. 6

[203] Loïc Sylvestre, Jocelyn Sérot et Emmanuel Chailloux : Hardware implementation of OCaml using a synchronous functional language. *In Practical Aspects of Declarative Languages (PADL '24)*, pages 151–168. Springer, 2024. 178, 181

[204] Loïc Sylvestre, Jocelyn Sérot et Emmanuel Chailloux : Programming parallelism on FPGAs with Eclat. *In High-level Parallel Programming and Applications (HLPP '24)*, 2024. 198

[205] Jean-Pierre Talpin et Pierre Jouvelot : The type and effect discipline. *Information and computation*, 111[2]:245–296, 1994. 31, 53

[206] Olivier Tardieu : *Loops in Esterel : from operational semantics to formally specified compilers*. Thèse de doctorat, École Nationale Supérieure des Mines de Paris, 2004. 27

[207] Olivier Tardieu : Goto and concurrency introducing safe jumps in Esterel. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153[4]:55–70, 2006. 65

[208] David B Thomas : Synthesisable recursion for C++ HLS tools. *In Application-specific Systems, Architectures and Processors (ASAP '16)*, pages 91–98. IEEE, 2016. 4

[209] Donald Thomas et Philip Moorby : *The Verilog® hardware description language*. Springer, 2008. 1

[210] Mattia Tibaldi et Christian Pilato : A survey of FPGA optimization methods for data center energy efficiency. *IEEE Transactions on Sustainable Computing (T-SUSC)*, 8[3]:343–362, 2023. 1

[211] Lenny Truong et Pat Hanrahan : A golden age of hardware description languages : Applying programming language techniques to improve design productivity. *In Summit on Advances in Programming Languages (SNAPL '19)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019. 1

[212] Chun-Jen Tsai, Han-Wen Kuo, Zigang Lin, Zi-Jing Guo et Jun-Fu Wang : A Java processor IP design for embedded SoC. *ACM Transactions on Embedded Computing Systems (TECS)*, 14[2]:1–25, 2015. 5

[213] David A Turner : *Recursion equations as a programming language*. Springer, 2016. 155

[214] Milla Valnet, Raphaël Monat et Antoine Miné : Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs. *In Journées Francophones des Langages Applicatifs (JFLA '23)*, pages 211–242, 2023. 199

[215] Steven Varoumas et Tristan Crolard : WCET of OCaml bytecode on microcontrollers : An automated method and its formalisation. *In Worst-Case Execution Time Analysis (WCET '19)*, volume 72, pages 5–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. 200

[216] Steven Varoumas, Basile Pesin, Benoît Vaugon et Emmanuel Chailloux : Programming microcontrollers through high-level abstractions : The OMicroB project. *Journal of Computer Languages (COLA)*, 77:101228, 2023. 6

[217] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman et Luis Ceze : Reticle : A virtual machine for programming modern FPGAs. *In Programming Language Design and Implementation (PLDI '21)*, pages 756–771, 2021. 192

[218] Paolo Visconti, Ramiro Velázquez, Carolina del Valle-Soto et Roberto de Fazio : FPGA based technical solutions for high throughput data processing and encryption for 5G communication : A review. *Telecommunication Computing Electronics and Control (TELKOMNIKA)*, 19[4]:1291–1306, 2021. 1

[219] Hasitha Muthumala Waidyasooriya, Masanori Hariyama et Kunio Uchiyama : *Design of FPGA-based computing systems with OpenCL*, volume 6. Springer, 2018. 3

[220] Jih-Jong Wang, Richard Katz, JS Sun, Brian Cronquist, John McCollum, TM Speers et WC Plants : SRAM based re-programmable FPGA for space applications. *IEEE Transactions on Nuclear Science (TNS)*, 46[6]:1728–1735, 1999. 14

[221] Markus Weinhardt et Wayne Luk : Memory access optimization and ram inference for pipeline vectorization. *In Field-Programmable Logic and Applications (FPL '99)*, pages 61–70. Springer, 1999. 128

[222] Inès Winandy, Arnaud Dion, Pierre-Loïc Garoche et Florent Manni : A Reactive System-specific Compilation Chain from Synchronous Dataflow Models to FPGA Netlist. *In Space Mission Challenges for Information Technology (SMC-IT/SCC '24)*, 2024. 9, 192, 194

[223] Felix Winterstein, Samuel Bayliss et George A Constantinides : High-level synthesis of dynamic data structures : A case study using Vivado HLS. *In Field-Programmable Technology (FPT '13)*, pages 362–365. IEEE, 2013. 4

[224] Andrew K Wright et Matthias Felleisen : A syntactic approach to type soundness. *Information and computation*, 115[1]:38–94, 1994. 198

[225] Sadegh Yazdanshenas, Kosuke Tatsumura et Vaughn Betz : Don't forget the memory : Automatic block RAM modelling, optimization, and architecture exploration. *In Field-Programmable Gate Arrays (FPGA '17)*, pages 115–124. ACM SIGDA, 2017. 13, 184

[226] Eugene Yip, Alain Girault, Partha S Roop et Morteza Biglari-Abhari : Synchronous deterministic parallel programming for multi-cores with ForeC. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 45[2]:1–74, 2023. 9

[227] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A Kim et Stephen A Edwards : Hardware synthesis from a recursive functional language. *In Hardware/Software Codesign and System Synthesis (CODES+ISSS '15)*, pages 83–93. IEEE, 2015. 5

[228] Dimitris Ziouzios, Pavlos Kilintzis, Nikolaos Baras et Minas Dasygenis : A survey of FPGA robotics applications in the period 2010–2019. *Advances in Science, Technology and Engineering Systems Journal (ASTESJ)*, 6:385–408, 2021. 1

Table des figures

1.1 Chaîne de compilation de Macle et OCaml sur FPGA 6
1.2 Construction d'une fonction avec état en μFP 7
1.3 Implantation de la récursion terminale en Daisy 8

2.1 Architecture d'un FPGA 13
2.2 Structure d'un élément logique avec additionneur intégré 14
2.3 Une carte FPGA (DE10-lite) embarquant le circuit FPGA MAX10 d'Intel . . . 15
2.4 Description d'un circuit en VHDL 16
2.5 *net-list* obtenue par synthèse du module VHDL `top` de la figure 2.4 16
2.6 Réalisation du circuit `top` de la figure 2.5 sur le FPGA MAX 10 17
2.7 Inverseur CMOS 18
2.8 Demi-additionneur 1-bit 19
2.9 Additionneur complet 1-bit 19
2.10 Verrou 20
2.11 Comportement temporel d'un verrou (ou *latch*) 20
2.12 Bascule D 21
2.13 Comportement temporel d'une bascule D 21
2.14 Schéma simplifié d'un circuit séquentiel 22
2.15 Structure des machines de Moore et Mealy 22
2.16 Comportement temporelle d'une machine de Mealy 24
2.17 Deux réalisations d'un circuit d'accumulation 25
2.18 Comportement temporel de machines de Moore et de Mealy 25
2.19 Compositions de machines à états 26

3.1 Syntaxe d'Eclat 29
3.2 Règles d'unification des durées 32
3.3 Typage des opérateurs 33
3.4 Abréviations dans la syntaxe concrète d'Eclat 35
3.5 Constructions dérivées disponibles en Eclat 35
3.6 Un calcul en Eclat 37
3.7 Exécution d'un programme Eclat 37
3.8 Un programme instantané exécutant un calcul long 38

3.9 Exécution pas à pas suivant un temps logique . . . 39
3.10 Un compteur en ECLAT . . . 40
3.11 Un programme ECLAT mixant calcul et interaction . . . 41
3.12 Compositions séquentielles et parallèles en ECLAT . . . 42
3.13 Un calcul long irrégulier en ECLAT . . . 44
3.14 Réinitialisation d'un calcul sous un `exec` . . . 45
3.15 Composition séquentielle de blocs `exec` . . . 45
3.16 Définition d'un type vecteur et parcours de vecteur en ECLAT . . . 46
3.17 Mise à jour d'un registre par un calcul long . . . 47
3.18 Retarder un flot en ECLAT . . . 48
3.19 L'automate d'ABRO *(source Gérard Berry)* . . . 49
3.20 ABRO en ECLAT : "attendre `A` et `B` puis émettre une fois `O` sauf si `R`" . . . 49
3.21 Deux implantations possible du circuit ABCRO . . . 50
3.22 Observateur synchrone en ECLAT . . . 51
3.23 Modéliser un environnement physique de façon synchrone . . . 52

4.1 Exemple de polymorphisme de taille en ECLAT . . . 59
4.2 Accès dynamique aux types des valeurs et instantiation d'inconnues de tailles . . . 60
4.3 Ensemble des valeurs ECLAT . . . 61
4.4 Sémantique des opérateurs . . . 68
4.5 Substituer chaque occurrence de variable par une instance d'une valeur . . . 73
4.6 Exemple d'instanciation au niveau matériel . . . 74
4.7 Substitution sans-capture non-standard d'ECLAT (extrait) . . . 75

5.1 Définition du langage de transitions utilisé . . . 78
5.2 Un calcul ECLAT à compiler . . . 79
5.3 Encodage du programme de la figure 5.2 sous forme d'automate . . . 79
5.4 Implantation du programme de la figure 5.2 sous forme de machine de Mealy . . . 80
5.5 Implantation inefficace du programme ECLAT de la figure 3.8 . . . 82
5.6 Implantation efficace du programme ECLAT de la figure 3.8 . . . 82
5.7 Automate de calcul de $(a + b) + c$ utilisant un seul additionneur . . . 84
5.8 Calcul de $(a + b) + c$ en ECLAT utilisant un seul additionneur . . . 85
5.9 Règles de partage et de duplication des fonctions ECLAT . . . 87
5.10 Chaîne de compilation d'ECLAT sur FPGA . . . 88
5.11 Appels de fonctions externes depuis ECLAT . . . 90
5.12 Définition d'un circuit externe en ECLAT . . . 91
5.13 Définition d'un calcul externe en ECLAT . . . 91
5.14 Syntaxe de MINI-HDL . . . 92
5.15 Valeurs par défaut d'une variable MINI-HDL en fonction de son type . . . 93
5.16 Encodage du circuit de la figure 5.6 . . . 94
5.17 Constructions dérivées de MINI-HDL . . . 95

5.18 Encodage du circuit de la figure 5.6 avec constructions dérivées 96
5.19 Imbrication d'automates en Mini-HDL 97
5.20 Composition parallèle d'automates en Mini-HDL 98
5.21 Typage des instructions Mini-HDL 99
5.22 Sémantique comportementale de Mini-HDL 100
5.23 Traduction des types et constantes Mini-HDL vers VHDL 102
5.24 Traduction des atomes Mini-HDL vers VHDL 102
5.25 Collecte de l'ensemble des variables déclarées dans une instruction Mini-HDL . 103
5.26 Traduction des instructions Mini-HDL vers VHDL 104
5.27 Traduction d'un programme Mini-HDL en VHDL 105
5.28 Syntaxe de Core-Eclat 107
5.29 Compilation d'un appel récursif terminal 111
5.30 Compilation de l'appel direct à une fonction récursive 112
5.31 Compilation d'un appel de fonction externe 112
5.32 Compilation de `reg` 114
5.33 Compilation d'`exec` 115
5.34 Compilation des paires parallèles 116
5.35 Insertion de `case/with` pour l'implantation du retour de fonction 120
5.36 Normalisation d'Eclat 120
5.37 Globalisation par λ-lifting 122
5.38 Globalisation puis intégration des appels de fonctions non-récursives 123
5.39 Monomorphisation 124

6.1 Description d'une mémoire RAM de 1024 mots en VHDL 128
6.2 Syntaxe et sémantique d'Eclat avec mémoire partagée 129
6.3 Typage d'Eclat avec mémoire partagée 131
6.4 Sémantique d'Eclat avec mémoire partagée 132
6.5 Accès simultanés à deux tableaux différents 133
6.6 Section critique en Eclat 134
6.7 Entrelacement équitable des accès mémoire dans un programme Eclat 136
6.8 Extension de la syntaxe de Core-Eclat avec mémoire partagée 136
6.9 Syntaxe de Mini-HDL étendue avec le mémoire RAM 138
6.10 Sémantique comportementale de Mini-HDL avec mémoire partagée 139

7.1 Modulateur de largeur d'impulsions en Eclat 150
7.2 Faire varier le rapport cyclique avec deux boutons 151
7.3 Point d'entrée qui instancie dix PWM 151
7.4 Variations de l'intensité des LED de la carte DE10-lite par PWM 152
7.5 Jeu de pong mis en pause 153
7.6 Insertion de pauses dans la fonction de déplacement des objets du jeu de Pong 155
7.7 Représentation d'une solution au problème des n reines pour $n = 5$ 156

7.8 Recherche exhaustive du nombre de solutions au problème des n reines 157
7.9 Temps d'exécution de `queens(`n`)` en fonction de n 157
7.10 Accélération du problème des n reines par déroulement de la fonction `safe_all` 158
7.11 Décomposition du problème des n reines en n sous problèmes parallélisés . . . 159
7.12 Compromis temps-espace dans la parallélisation des n reines 160
7.13 Facteur d'accélération des n reines en Eclat par rapport à OCaml sur CPU . 161
7.14 Jeu de la vie, version 1 164
7.15 Jeu de la vie, version 2 165
7.16 Compromis temps espace pour le Jeu de la vie sans mémoire partagée 166
7.17 Jeu de la vie, version 3 167
7.18 Compromis temps-espace pour différentes implantations du Jeu de la vie 168
7.19 Une implantation parallèle de `map` 169
7.20 Implantation parallèle de `map` utilisant une ferme de travailleurs 170
7.21 Implantation parallèle de `reduce` utilisant une ferme de travailleurs 171
7.22 Implantation d'un pipeline à trois étages en Eclat 171
7.23 Amorçage de `map_par` avec $T(\mathtt{f}_i) = T_{\mathtt{f}} = \mathtt{p} - 2$ 173
7.24 Accélération et temps d'exécution pour `map_par` en fonction de `p` et $T_{\mathtt{f}}$ 173
7.25 Parallélisation des n reines (adaptée de `queens_par`) avec des squelettes 174
7.26 Contrôle du degré de parallélisme dans le calcul des n reines à base de squelettes 174
7.27 Implantation d'OCaml sur FPGA en utilisant Eclat 175
7.28 Taille de mot paramétrable et partitionnement de la mémoire pour OCaml . . 177
7.29 Perte de vitesse du calcul des n reines en bytecode OCaml sur FPGA vs Eclat 177
7.30 fonction Eclat calculant la longueur d'une liste OCaml 179
7.31 Une application réactive écrite en Eclat et embarquant du code OCaml . . . 180

A.1 Implantation d'OCaml sur FPGA avec Eclat 203
A.2 Mélange de calcul et d'interaction 204
A.3 Partage des appels de fonctions récursives 205
A.4 Parallélisme au niveau RT 206
A.5 Concurrence au niveau RT 207
A.6 Modularité 208

B.1 Programmation d'un automate en Eclat 209
B.2 Syntaxe abstraite 210
B.3 Langage de types 211
B.4 Règles d'unification des durées 212
B.5 Constantes et opérateurs 212
B.6 Opérations syntaxiques 213
B.7 Abréviations dans la syntaxe des programmes 214
B.8 Abréviations dans la syntaxe des expressions 214
B.9 Constructions dérivées (expressions et motifs) 215

B.10 Accès dynamique aux types des valeurs et instantiation d'inconnues de tailles . 215
B.11 Environnement de typage 216
B.12 Typage des programmes 217
B.13 Typage des expressions (première partie) 218
B.14 Typage des expressions (deuxième partie) 219
B.15 Typage des expressions (nouvelles constructions) 219
B.16 Typage des opérateurs (version étendue) 220
B.17 Langage de valeurs (version étendue) 221
B.18 Sémantique comportementale (noyau fonctionnel parallèle) 222
B.19 Sémantique comportementale (noyau réactif) 223
B.20 Sémantique comportementale (création de tableau) 224
B.21 Sémantique comportementale (extensions) 224
B.22 Sémantique des opérateurs combinatoires 225
B.23 Sémantique des accès mémoire 226
B.24 Sémantique des primitives de simulation 226
B.25 Gestion des erreurs d'exécution 227
B.26 Renommage préfixe pour la substitution sans-capture non-standard d'Eclat . 228
B.27 Substitution sans-capture non-standard d'Eclat 229
B.28 Sémantique à réductions (contextes d'évaluation) 230
B.29 Sémantique à réductions (première partie) 231
B.30 Sémantique à réductions (deuxième partie) 232

Table des matières

1 Introduction **1**

1.1 Contexte . . . 1
1.2 Problématique . . . 2
1.3 État de l'art . . . 3
1.3.1 Synthèse de haut niveau . . . 3
1.3.2 Langages dédiés . . . 4
1.3.3 Langages de modélisation au niveau système . . . 4
1.3.4 Implantation de langages de programmation existants sur FPGA . . . 4
1.3.5 Description de matériel en style fonctionnel . . . 7
1.3.6 Langages synchrones . . . 8
1.4 Synthèse : vers un langage synchrone généraliste . . . 9
1.5 Structure du manuscrit . . . 11

2 Conception et implantation de circuits sur FPGA **13**

2.1 Architecture . . . 13
2.2 Interagir avec l'environnement extérieur . . . 14
2.3 Circuits synchrones . . . 18
2.3.1 Fonctions combinatoires . . . 18
2.3.2 Fonctions séquentielles . . . 20
2.4 Modèles d'implantation des circuits synchrones . . . 22
2.5 Hypothèse de synchronisme au niveau RT . . . 23
2.6 Composition de machines à états . . . 25

3 Mixer calcul et interaction : le langage Eclat **27**

3.1 Présentation générale . . . 27
3.2 Syntaxe . . . 28
3.2.1 Langage d'expressions . . . 30
3.2.2 Langage de types . . . 31
3.2.3 Langage de durées . . . 31
3.2.4 Constantes et opérateurs prédéfinis . . . 32
3.2.5 Syntaxe concrète . . . 33
3.2.6 Extension de l'environnement d'exécution ECLAT . . . 36

3.3 Principes de programmation en Eclat . . . 36
3.3.1 Récursion terminale pour le calcul . . . 36
3.3.2 Exécution d'un calcul pas à pas . . . 38
3.3.3 Circuits séquentiels pour l'interaction . . . 40
3.3.4 Mélange de calcul et d'interaction . . . 40
3.3.5 Composition de calculs . . . 41
3.4 Expressivité du langage . . . 42
3.4.1 Calcul irrégulier . . . 43
3.4.2 Réinitialisation . . . 44
3.4.3 Désynchroniser les calculs parallèles . . . 44
3.4.4 Traits de programmation applicatifs . . . 45
3.5 Programmation synchrone . . . 47
3.5.1 Mise à jour de registres en plusieurs cycles . . . 47
3.5.2 Retarder un flot . . . 48
3.5.3 Encoder le contrôle en style flot de données . . . 48
3.5.4 Observateur synchrone . . . 50
3.5.5 Modéliser un environnement . . . 51
3.6 Conclusion . . . 52

4 Sémantique d'Eclat 53
4.1 Sémantique statique : typage et analyse de réactivité . . . 53
4.2 Sémantique synchrone . . . 61
4.2.1 Ensemble des valeurs . . . 61
4.2.2 Jugements d'évaluation et de réduction . . . 62
4.2.3 Notations . . . 62
4.2.4 Réduction des programmes Eclat . . . 63
4.2.5 Noyau applicatif . . . 63
4.2.6 Réduction des appels d'opérateurs . . . 67
4.2.7 Noyau réactif . . . 69
4.2.8 Fonctions extérieures . . . 72
4.3 Substitution sans capture et mémoire locale . . . 72
4.4 Conclusion . . . 76

5 Implantation d'Eclat sur FPGA 77
5.1 Interprétation d'Eclat en termes de circuits . . . 77
5.2 Principes d'implantation . . . 78
5.2.1 Encodage des calculs sous forme d'automates . . . 78
5.2.2 Combiner flot de données et flot de contrôle . . . 81
5.2.3 De la nécessité de dupliquer les circuits . . . 83
5.2.4 Duplication et partage . . . 83
5.2.5 Des synchronisations instantanées . . . 85

5.2.6 Traits applicatifs et imbrications : un partage limité 86
5.2.7 Synthèse 87
5.3 Chaîne de compilation 88
5.4 Interopérabilité 89
5.5 Abstraction de la cible : Mini-HDL 91
5.5.1 Syntaxe et sémantique intuitive 92
5.5.2 Proximité entre Eclat et Mini-HDL 94
5.5.3 Extension de Mini-HDL avec des automates 95
5.5.4 Sémantique 98
5.6 Traduction de Mini-HDL en VHDL 102
5.6.1 Représentation des valeurs 102
5.6.2 Traduction des expressions combinatoires 102
5.6.3 Traduction des instructions 103
5.6.4 Traduction du programme entier 104
5.6.5 Instanciation des circuits externes 106
5.7 Core-Eclat : un sous-ensemble normalisé d'Eclat 107
5.8 Compilation de Core-Eclat vers Mini-HDL 108
5.8.1 Compilation des programmes 109
5.8.2 Compilation des automates 110
5.8.3 Compilation des expressions 111
5.8.4 Retour instantané des appels de fonctions 119
5.9 Normalisation d'Eclat vers Core-Eclat 119
5.9.1 Mise en forme ANF (*Administrative Normal Form*) 120
5.9.2 Spécialisation I 121
5.9.3 Globalisation (par λ-lifting) 121
5.9.4 Spécialisation II 121
5.9.5 Intégration des appels de fonctions non récursives 123
5.9.6 Déconstruction des motifs 123
5.9.7 Instanciation 123
5.9.8 Monomorphisation 124
5.10 Conclusion 125

6 Mixer parallélisme synchrone et mémoire partagée 127
6.1 Mémoire accessible dans un circuit FPGA 128
6.2 Extension d'Eclat avec mémoire partagée 129
6.2.1 Opérations sur les tableaux 129
6.2.2 Programmation concurrente 133
6.2.3 Equité 135
6.3 Compilation d'Eclat avec mémoire partagée 136
6.3.1 Adapter Core-Eclat 136
6.3.2 Adapter la normalisation d'Eclat vers Core-Eclat 136

6.3.3 Adapter Mini-HDL . . . 137
6.3.4 Adapter la compilation de Core-Eclat vers Mini-HDL . . . 139
6.3.5 Adapter la traduction de Mini-HDL vers VHDL . . . 141
6.4 Discussion . . . 142

7 Développement d'applications sur FPGA avec Eclat 145
7.1 Méthodologie expérimentale . . . 146
7.1.1 Flot de conception de circuit . . . 146
7.1.2 Mesure des temps d'exécution . . . 147
7.1.3 Comparer Eclat sur FPGA et OCaml sur CPU . . . 148
7.2 Macro-constructions dans le compilateur Eclat . . . 148
7.3 Faire varier l'intensité d'une rangée de LEDs . . . 149
7.4 Jeu de Pong . . . 152
7.5 Un algorithme généraliste : les n reines . . . 154
7.6 Compromis temps-espace . . . 158
7.7 Mémoire partagée : le Jeu de la vie . . . 162
7.7.1 Version 1 : représenter la grille par un tableau . . . 163
7.7.2 Version 2 : représenter la grille par un vecteur . . . 165
7.7.3 Version 3 : représenter la grille par un tableau de vecteurs . . . 166
7.8 Développement de squelettes de parallélisme en Eclat . . . 168
7.8.1 Implantation de `map` . . . 168
7.8.2 Fermes de travailleurs . . . 169
7.8.3 Implantation de `reduce` . . . 170
7.8.4 Implantation de `pipe` . . . 170
7.8.5 Prédiction des performances . . . 172
7.8.6 Les n reines avec des squelettes . . . 173
7.9 Implantation matérielle de la machine virtuelle OCaml . . . 175
7.9.1 La machine virtuelle OCaml . . . 175
7.9.2 Chaîne de compilation pour OCaml sur FPGA . . . 175
7.9.3 Un environnement d'exécution paramétrable . . . 176
7.9.4 Efficacité . . . 177
7.9.5 Interopérabilité entre Eclat et OCaml . . . 178
7.9.6 Appel de fonctions Eclat depuis les programmes OCaml . . . 178
7.9.7 Accéder à des structures de données OCaml depuis Eclat . . . 179
7.9.8 Embarquer du code OCaml dans des applications réactives . . . 180

8 Travaux connexes 183
8.1 Description de matériel en style fonctionnel . . . 183
8.2 Synthèse de haut niveau (HLS) . . . 184
8.3 Construction de matériel dans des langages généralistes . . . 186
8.4 Langage de conception d'accélérateurs . . . 187

8.5 Un sous-ensemble d'Haskell pour la programmation au niveau RT 188
8.6 Prise en compte de la cible FPGA et programmation généraliste 188
8.7 Sémantique temporelle pour un langage généraliste 189
8.8 Implantation de langages synchrones sur FPGA 191
8.9 Conception et implantation de squelettes de parallélisme 192
8.10 Synthèse des travaux connexes 193

9 Conclusion et travaux futurs **195**

A Synthèse d'Eclat sur le FPGA Intel Max10 **203**

B Définition d'Eclat **209**
B.1 Syntaxe 210
B.2 Systèmes de types 216
B.3 Sémantique comportementale 221
B.4 Sémantique à réductions 230

C Prise en main du compilateur Eclat **233**
C.1 Appeler le compilateur 233
C.2 Mode intéractif 233
C.3 Typage statique 234
C.4 Analyse de réactivité par typage statique 234
C.5 Simulation et visualisation de formes d'ondes 236
C.6 Primitives de simulation 237
C.7 Interpréteur 237
C.8 Lien avec les entrées/sorties physiques 239
C.9 Simulation en OCaml 240

Bibliographie **241**

Table des figures **261**

Table des matières **267**

Résumé

Les FPGA (*Field-Programmable Gate Arrays*) sont des circuits numériques reconfigurables : leur comportement est personnalisable par synthèse logique de spécifications au niveau transfert de registres (niveau RT), dans des langages de description de matériel comme VHDL ou Verilog. Les FPGA sont bien adaptés pour réaliser des systèmes réactifs, directement sous forme de circuits synchrones interagissant avec l'environnement extérieur via des broches d'entrées/sorties – le synthétiseur logique garantissant que les contraintes temporelles du système sont respectées compte tenu de la fréquence d'horloge du FPGA cible. Les FPGA sont par ailleurs utilisés pour implanter des accélérateurs matériels. Cependant, les descriptions au niveau RT de systèmes transformationnels (ou « calculs ») – ayant des latences de plusieurs cycles d'horloge – sont difficiles à mettre au point, maintenir et optimiser manuellement. La synthèse haut niveau (*High-Level Synthesis, HLS*) constitue alors une solution pour exprimer plus simplement les calculs, dans un langage de programmation compilé au niveau RT. L'avantage de cette approche est de cacher aux programmeuses et programmeurs les détails d'implantation, en laissant au compilateur le soin d'ordonnancer les calculs au cours du temps. Cela conduit toutefois à une perte de contrôle sur le comportement temporel et donc la sûreté et l'efficacité des circuits engendrés. À l'heure où les systèmes embarqués, notamment à base de FPGA, doivent exécuter toujours plus de calculs, tout en interagissant avec leur environnement, cette thèse propose un modèle de programmation pour combiner description de matériel (orientée flot de données) et calcul parallèle généraliste (orienté flot de contrôle) en suivant une approche synchrone. Cela donne lieu à la conception et l'implantation d'Eclat, un langage de programmation fonctionnel-impératif, parallèle et synchrone, compilé en VHDL. Eclat est suffisamment précis pour décrire des circuits synchrones au niveau RT. Il permet de programmer simplement des accélérateurs matériels, avec une sémantique temporelle claire et prédictible par laquelle exploiter des compromis temps-espace. Tout programme Eclat est réactif, avec un mécanisme pour plonger les calculs dans les programmes et ainsi mélanger calcul et interaction. Eclat offre aussi de la mémoire partagée (réalisée sous forme de blocs RAM), conciliant concurrence et déterminisme. Il sert à développer des abstractions de programmation telles que des squelettes algorithmiques et des implantations de machines virtuelles pour des langages de haut niveau. Cela répond, à différents niveaux, au besoin d'intégrer des algorithmes généralistes à l'intérieur des applications embarquées réactives à base de FPGA.