

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Micro et Nano Électronique**

Arrêté ministériel : 7 août 2006

Présentée par

Tomasz TOCZEK

Thèse dirigée par **Dominique HOUZET** et
codirigée par **Stéphane MANCINI**

préparée au sein du **Laboratoire Gipsa-lab**
dans l'**École Doctorale E.E.A.T.S.**

Une approche fonctionnelle pour la conception et l'exploration architecturale de systèmes numériques

Thèse soutenue publiquement le **15 juin 2011**,
devant le jury composé de :

Pr. Jocelyn SEROT

Professeur à l'université Blaise Pascal (Clermont-Ferrand), Rapporteur et
Président du jury

Pr. Marc POUZET

Professeur à l'université Pierre et Marie Curie (Paris), Rapporteur

Pr. Dominique HOUZET

Professeur à Grenoble INP, Membre du jury

Dr. Stéphane MANCINI

Maître de conférences à Grenoble INP, Membre du jury

M. Laurent HILI

Ingénieur à l'ESTEC (Pays-Bas), Membre du jury



Résumé : Ce manuscrit présente une méthode de conception au niveau système reposant sur la programmation fonctionnelle typée et visant à atténuer certains des problèmes complexifiant le développement des systèmes numériques modernes, tels que leurs tailles importantes ou la grande variété des blocs les constituant. Nous proposons un ensemble de mécanismes permettant de mélanger au sein d'un même design plusieurs formalismes de description distincts («modèles de calcul») se situant potentiellement à des niveaux d'abstraction différents. De plus, nous offrons au concepteur la possibilité d'explicitier directement les paramètres explorables de chaque sous-partie du design, puis d'en déterminer des valeurs acceptables via une étape d'exploration partiellement ou totalement automatisée réalisée à l'échelle du système. Les gains qu'apportent ces stratégies nouvelles sont illustrés sur plusieurs exemples.

Mots clés : Conception au niveau système, Exploration architecturale, Modèles de calcul, Programmation fonctionnelle, Monades, Haskell

Abstract: This work presents a novel system-level design method based on typed functional programming and aiming at mitigating some of the issues making the development of modern digital systems complex, such as their increasing sizes and the variety of their subcomponents. We propose a range of mechanisms allowing to mix within a single design several description formalisms (“models of computation”), possibly at different abstraction levels. Moreover, the designer is provided with means to directly express the explorable parameters of each part of their design, and to find acceptable values for them through a partially or totally automatic system-wide architectural exploration step. The advantages brought by those new strategies are illustrated on several examples.

Keywords: System-Level Design, Design Space Exploration, Models of Computation, Functional Programming, Monads, Haskell

Remerciements

Avant de poursuivre, je tenais à remercier tout particulièrement mes directeurs de thèse, Dominique Houzet et Stéphane Mancini, pour leurs éclairages précieux et la liberté qu'ils m'ont laissée dans la réalisation de mes travaux de recherche. Je remercie également les professeurs Jocelyn Serot et Marc Pouzet en leur qualité de rapporteurs, et M. Laurent Hili pour sa participation au jury. Enfin, je voudrais témoigner de ma reconnaissance à ma famille pour m'avoir soutenu au cours de ces quelques dernières années, ainsi qu'à l'intégralité des personnels de Gipsa-lab, de Grenoble INP et de l'école doctorale EEATS.

Table des matières

Introduction	13
1 État de l'art	15
1.1 Challenges pour le design niveau système	15
1.2 Instanciation de composants	17
1.2.1 Composants génériques	17
1.2.2 Fichiers annexes, annotations du source, génération	21
1.3 Description de matériel et programmation fonctionnelle	22
1.3.1 Description de matériel	23
1.3.1.1 Bluespec SystemVerilog	23
1.3.1.2 Lava	23
1.3.1.3 ForSyDe	24
1.3.1.4 SML-Sys	24
1.3.1.5 CλaSH	24
1.3.2 Vérification formelle	24
1.3.2.1 Méthodes classiques	24
1.3.2.2 Méthodes basées sur le système de types	25
1.4 Exploration architecturale	25
2 Méthodologie proposée	27
2.1 Objectifs	27
2.1.1 Instanciation et explorabilité	27
2.1.2 Multiplicité des niveaux d'abstraction	28
2.1.3 Utilisation d'un seul langage	28
2.1.4 Facilité d'emploi	29
2.2 Dichotomie MoC/Moteur	29
2.2.1 Monade d'état explorable	30
2.2.2 Modèles de Calcul	34
2.2.3 Moteurs	35
2.2.4 Règles d'implémentation	36
2.3 Expression des méthodes d'exploration	36
2.3.1 Méthodes simples	36
2.3.1.1 Choix par défaut	36
2.3.1.2 Parcours aléatoire	37
2.3.2 Méthodes avancées	38
2.3.2.1 Types de données	38
2.3.2.2 Recuit simulé	40
2.3.2.3 Programmation génétique	42
2.3.2.4 Évaluation du score dans un contexte monadique	45
2.4 Exemple et comparaison	46
2.4.1 Description et pertinence	46
2.4.2 Implémentation	46

2.4.3	Vitesses de convergence	49
2.5	Conclusion	52
3	Modélisations sans horloge et niveau RTL	53
3.1	Modélisation sans horloge	53
3.1.1	Objectifs	53
3.1.2	Modèle de calcul	53
3.1.3	Moteur	55
3.1.4	Modèles de caches	57
3.1.4.1	Cache unidimensionnel	57
3.1.4.2	Cache multidimensionnel	59
3.1.5	Design	60
3.1.6	Résultats	62
3.1.7	Conclusions	64
3.2	Modélisation au niveau RTL	64
3.2.1	Objectifs	64
3.2.2	Modèle de calcul	65
3.2.2.1	Transfert	65
3.2.2.2	Hierarchie de classes	67
3.2.3	Moteur	70
3.2.3.1	Netlist	70
3.2.3.2	Simulation	73
3.2.3.3	Évaluation de la complexité	74
3.2.4	Modèles de caches	74
3.2.4.1	Interface mémoire	74
3.2.4.2	Cache unidimensionnel	75
3.2.4.3	Cache multidimensionnel	77
3.2.5	Design	78
3.2.6	Résultats	80
3.2.7	Conclusions	83
4	Résolution de divers problèmes architecturaux	85
4.1	Découpage des pipelines en étages	85
4.1.1	Contexte et principe	85
4.1.2	Extensions au modèle de calcul	85
4.1.2.1	Découpage automatique déterministe	85
4.1.2.2	Découpage automatique probabiliste	87
4.1.3	Étude de cas : pipeline de ray casting dans des grilles hiérarchiques	87
4.1.3.1	Grilles hiérarchiques	87
4.1.3.2	Principe de la traversée	88
4.1.3.3	Organisation matérielle	89
4.1.3.4	Résultats	89
4.2	Allocation de ressources sur GPU	93
4.2.1	Programmation sur GPU	93
4.2.1.1	Historique	93
4.2.1.2	Architecture	95
4.2.1.3	Problèmes	97
4.2.2	Modèles de calcul appropriés	98
4.2.2.1	Direct	98
4.2.2.2	Avec ordonnanceur	100
4.2.3	Gains apportés	102
4.3	Instanciation de réseaux d'interconnexions	103
4.4	Conclusion	105

Conclusion générale	107
A Syntaxe de Haskell	109
A.1 Bases	109
A.2 Filtrage par motif	110
A.3 Polymorphisme	112
A.4 Monades	113
A.5 Prélude	114
A.5.1 Arithmétique	114
A.5.2 Listes et chaînes de caractères	114
A.5.3 Monades	116
A.5.4 Divers	117
B Reconstruction tomographique	119
B.1 Principe de la reconstruction tomographique	119
B.2 Rétroprojection en tomographie PET	119
C Coprocesseur pour la traversée de grilles hiérarchiques	121
C.1 Original (VHDL)	121
C.2 MoC RTL, calqué sur la version VHDL	133
C.3 MoC RTL avec découpage automatique du pipeline	143

TABLE DES MATIÈRES

Table des figures

1.1	Quantité prévisionnelle de logique présente dans les SoCs	16
1.2	Quantité prévisionnelle de logique réutilisée dans les SoCs	16
1.3	Productivité prévisionnelle des concepteurs de SoCs	16
1.4	<code>my_noc.vhd</code>	18
1.5	<code>switch.vhd</code>	18
1.6	<code>my_noc.vh</code>	19
1.7	<code>priorities.v</code>	19
1.8	<code>switch.v</code>	19
2.1	Comportement de <code>pickTwo</code>	44
2.2	Exemple de parcours aléatoire de l'espace de design	50
2.3	Exemple de recuit simulé	50
2.4	Exemple d'exploration par programmation génétique	51
2.5	Comparaison de exemples précédents	51
3.1	Comparaison des politiques d'adressage implémentées	60
3.2	Allure typique des accès mémoire pour le calcul d'un voisinage de voxels	63
3.3	Hierarchie de classes définissant les MoCs RTL et impératifs (listing)	68
3.4	Hierarchie de classes définissant les MoCs RTL et impératifs (graphe)	69
3.5	Représentation d'une netlist	71
3.6	Modèle RTL de mémoire à latence élevée	76
3.7	Adressage de tableaux multidimensionnels	79
3.8	Schéma global du réseau d'unités de reconstruction tomographique	80
3.9	Unité de reconstruction tomographique modélisée au niveau RTL	81
3.10	Réseau d'unités de reconstruction tomographiques modélisé au niveau RTL	82
3.11	Nombre moyen d'éléments lus par cycle en fonction de l'itération	83
4.1	Exemple de grille hiérarchique $4 \times 4 \times 4$	88
4.2	Exemple de traversée de grille uniforme 2D via l'algorithme DDA	88
4.3	Variables caractérisant un rayon à un moment de la traversée	90
4.4	Algorithme de détermination de la prochaine cellule	90
4.5	Schéma d'ensemble du SoPC de rendu par lancer de rayons	91
4.6	Organisation de l'unité de traversée	91
4.7	Organisation de l'unité de recherche de voisins	92
4.8	Comparaison entre trois implémentations de l'unité de recherche de voisins (en terme de code source)	92
4.9	Schéma architectural d'un GPU de nVidia (figure empruntée à [69])	95
4.10	Grilles et blocs de threads sous CUDA (figure empruntée à [69])	96
4.11	MoC PTX	99
4.12	Programme exprimé dans le MoC PTX	100
4.13	Modèles de calcul pour la programmation impérative sur GPU	101
4.14	Exemple de programme dans ce modèle de calcul	101

TABLE DES FIGURES

4.15	Modèle de calcul pour la programmation sur GPU avec primitives de communication entre grilles et ordonnancement	102
4.16	Recuit simulé d'un programme sur GPU	104
4.17	Organisation possible de classes pour la modélisation de NoCs	104
A.1	Hierarchie des classes dédiées à l'arithmétique et leurs instances	115

Introduction

Le design niveau système est un domaine difficile pour un grand nombre de raisons. Les systèmes à concevoir sont de plus en plus complexes, et comportent des blocs hétérogènes aux fonctionnalités et comportements extrêmement variés qui pourtant doivent communiquer efficacement. La réutilisation de composants devient une nécessité, et il est de plus en plus ardu pour le concepteur de connaître le design dans son intégralité et donc de comprendre les motifs qui en dictent les performances. De cet état de fait résulte la nécessité croissante d'automatiser les choix architecturaux, ou tout du moins de guider le concepteur dans leur réalisation. Parallèlement, la taille même des designs actuels et les besoins en terme de réutilisation appellent à une flexibilité toujours plus grande dans les mécanismes d'instanciation des composants individuels.

Nous proposons dans ce manuscrit une approche basée sur la programmation fonctionnelle typée pour le design niveau système. L'intérêt de l'utilisation de langages fonctionnels est multiple. D'ores et déjà, l'expressivité de ces langages permet de remédier à une partie significative des problèmes que le designer peut rencontrer dans l'organisation de son système. L'utilisation de fonctions d'ordre supérieur et du polymorphisme peut favorablement remplacer des mécanismes telles que la programmation générique et constitue une excellente voie pour favoriser la réutilisation de code source. Par ailleurs, les paradigmes fonctionnels trouvent leur applications dans des domaines appelés à prendre une importance croissante à l'avenir, tels que vérification formelle ou l'extraction de parallélisme. On peut même affirmer qu'attaquer ces deux problèmes requiert inévitablement une analyse fonctionnelle du programme à un moment ou à un autre.

Un des aspects centraux de la méthode proposée est de permettre au concepteur de spécifier les différentes parties de son design via différents jeux de primitives que nous appelons modèles de calcul. Procéder ainsi permet à chaque élément du design d'être décrit dans le paradigme le plus adapté à son niveau d'abstraction et à ses autres propriétés intrinsèques. Chaque modèle de calcul constitue ce que l'on qualifie souvent de langage embarqué spécifique à un domaine : il s'agit techniquement d'une bibliothèque de constructions qu'il est possible d'agencer de sorte à dégager des éléments syntaxiques et sémantiques non-présents dans le langage hôte. Nous proposons un ensemble de règles à suivre dans la construction de modèles de calcul, leur permettant de communiquer entre eux, et garantissant la possibilité d'explorer architecturalement les designs exprimés par leur biais. Il est intéressant de noter que cantonner les différents éléments d'un design à des modèles de calcul précis est en soi désirable dans la mesure où cela permet de bénéficier des invariants propres à chacun. Par contraste, l'utilisation d'un langage à usage plus général ne permet pas de tirer parti aussi facilement et systématiquement des propriétés résultant (éventuellement) de la discipline d'implémentation utilisée. Le parallélisme en est un exemple : difficile à déduire dans le contexte d'un langage à usage général, il peut néanmoins être extrait naturellement dans le cadre de toute une variété de modèles de calcul qui s'y prêtent.

La suite de ce manuscrit comporte en tout quatre chapitre et trois annexes.

Le premier chapitre fait office d'introduction étendue et recense quelques approches comparables à celle que nous proposons. Plus spécifiquement, on y trouve quelques prévisions sur la productivité dans le domaine de la conception électronique au cours des prochaines années, ainsi qu'une discussion des limitations des approches classiques pour le design niveau système. La fin du chapitre énumère des méthodes plus modernes permettant de palier à ces limitations. Elles sont divisées en trois catégories : les approches niveau système utilisant des annotations ou des fichiers annexes en conjonction des méthodes classiques, les approches fonctionnelles, et enfin celles se

focalisant sur l'exploration architecturale.

Le second chapitre présente les objectifs que nous nous sommes fixés et les principes de base de la méthode que nous proposons pour les atteindre. On y voit en détail les mécanismes d'exploration architecturale et leur fondements fonctionnels. Le chapitre donne et commente les implémentations de trois algorithmes d'exploration stochastiques : le parcours aléatoire, le recuit simulé et une méthode basée sur la programmation génétique. On discute également les moyens à mettre en œuvre pour permettre l'utilisation d'entrées/sorties durant l'étape d'exploration tout en restant dans un contexte fonctionnel pur. Enfin, les trois algorithmes proposés sont comparés sur un exemple très simple mais avec une taille d'espace de design volontairement très importante.

Le troisième chapitre constitue une étude de cas permettant de se faire une idée des apports de la méthodologie proposée dans un contexte plus concret. L'application cible choisie relève de l'imagerie médicale, et plus spécifiquement de la tomographie à émission de positrons. On propose deux implémentations matérielles distinctes d'unités de rétroprojection tomographique. La rétroprojection est une étape de la reconstruction tomographique coûteuse à la fois en calcul et en accès mémoire, et les choix liés aux détails algorithmiques et architecturaux sont particulièrement importants. Il s'agit donc d'un bon moyen d'évaluer les possibilités en matière d'exploration automatisée de l'espace de design sur un exemple réaliste. Chaque implémentation présente et utilise un modèle de calcul différent : modélisation sans horloge pour la première, et niveau RTL pour la seconde.

Le quatrième chapitre présente plusieurs aspects difficiles du design niveau système ou orienté plateforme, et donne des pistes permettant de les résoudre se basant sur les propositions élaborées au second chapitre. On y privilégie les problèmes qu'il est ardu d'attaquer sans l'aide d'outils supplémentaires dans les approches conventionnellement utilisées en conception. On y abordera le découpage automatisé en pipelines de chemins combinatoires, la programmation et l'allocation de ressources sur processeurs graphiques programmables, ainsi que l'instanciation de réseaux d'interconnexions.

La première annexe est une introduction à Haskell. Les chapitres deux à quatre contenant de nombreux extraits de code dans ce langage, il est recommandé que le lecteur ne connaissant pas encore ce langage commence par cette annexe. La seconde annexe regroupe les portions de code trop longues pour être intégrés à l'endroit du manuscrit où elles sont discutées. La troisième présente très succinctement les équations relatives à l'étape de rétroprojection en topographie PET qui a été prise comme exemple au chapitre 3.

Chapitre 1

État de l'art

1.1 Challenges pour le design niveau système

L'évolution des technologies dans le domaine de la micro-électronique a toujours été source d'une forte activité de recherche dans le domaine de la conception assistée par ordinateur électronique (EDA). Gordon Moore est considéré comme le premier à avoir clairement mis en évidence le caractère empiriquement exponentiel de l'évolution de la complexité des circuits numériques intégré avec le temps [64]. Sa prédiction de 1975, stipulant que le nombre de transistors par *die* doublerait tous les deux ans [63], s'est avérée remarquablement exacte jusqu'à aujourd'hui, et devrait le rester pendant au moins les 5 prochaines années [4]. Cette croissance rapide implique que le designer augmente régulièrement sa productivité s'il veut être capable de concevoir des circuits aussi complexes que la technologie lui permet (voir figures 1.1, 1.2 et 1.3). Historiquement, de cette nécessité déjà a découlé un certain nombre de révolutions dans la conception électronique. La plus significative a sans doute été celle de l'arrivée des langages de modélisation et design de matériel, pour remplacer, là où c'était possible, le design niveau porte. Bien qu'ayant augmenté considérablement l'efficacité du designer, la conception de puces entières au niveau RTL par le biais de ces langages est depuis déjà plusieurs années devenue très difficile à un coût raisonnable. L'avènement des Systems on Chip (SoCs), conséquence de l'explosion de la complexité des circuits, a à la fois permis de mitiger le fardeau du designer en se basant davantage sur la réutilisation de microprocesseurs, d'architectures d'interconnexions et d'autres blocs matériels (IP), mais aussi apporté son lot de challenges spécifiques :

1. problèmes de partitionnement logiciel/matériel : la théorie veut que les tâches de haut niveau soient faites en soft, avec des coprocesseurs hardware dans le rôle d'accélérateurs de traitements répétitifs de plus bas niveau ; hélas, dans les cas non triviaux, le choix entre matériel et logiciel pour une partie d'un traitement donnée est à la fois difficile et crucial du point de vue des performances obtenues après optimisation
2. problèmes de co-design logiciel/matériel : une fois le partitionnement effectué, le fait qu'un traitement donné comporte à la fois une partie soft et une partie hard le rend moins trivial à implémenter et à mettre au point
3. problèmes d'interconnexions : de nombreux composants vont devoir communiquer entre eux, ce qui va se traduire par des motifs de trafic dans les interconnexions souvent complexes et difficiles à anticiper ; dans beaucoup de cas, le nombre d'IP est trop grand pour qu'un bus unique suffise — les hiérarchies de bus et les Networks-on-Chip (NoCs) permettent d'y remédier mais amènent leur propre lot de paramètres à configurer : position des blocs IP, tailles des buffers des switches/bridges, problèmes de contention/deadlocks, de délais variables dans la transmission des messages, etc.
4. problèmes d'organisation des hiérarchies mémoire : pour faire face à la latence importante des DRAMs off-chip, une surface de plus en plus importante du die est consacrée aux buffers

1.1. CHALLENGES POUR LE DESIGN NIVEAU SYSTÈME

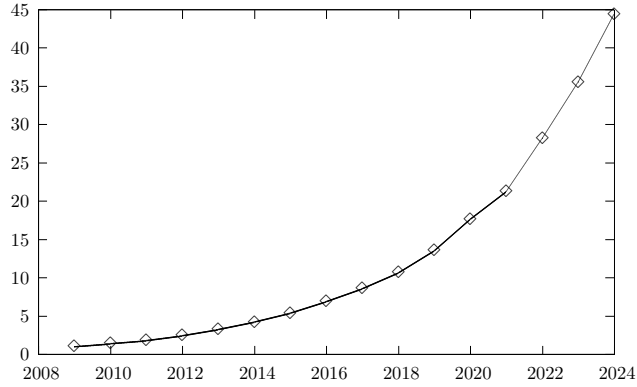


FIGURE 1.1 – Quantité de logique présente dans un SoC utilisé dans un appareil électronique grand public dans les années qui viennent, normalisée par rapport à 2009 (prévisions de l'ITRS)

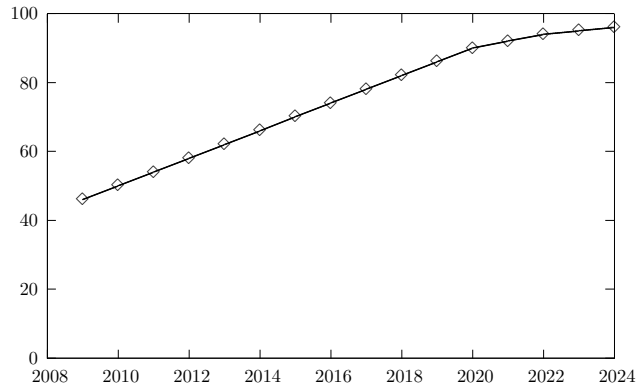


FIGURE 1.2 – Quantité de logique réutilisée dans un tel SoC, en pour cent (prévisions de l'ITRS)

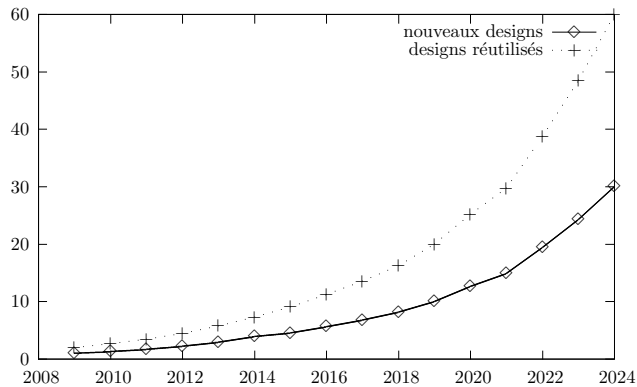


FIGURE 1.3 – Productivité par designer nécessaire pour la conception d'un tel SoC, normalisée par rapport à la productivité pour de nouveaux designs en 2009 (prévisions de l'ITRS)

on-chip ; il s'agit de caches ou de scratchpads dont il va falloir gérer le partage des données, la cohérence mutuelle et le write-back, les garanties de temps de réponse, etc.

5. vérification et profiling : la simulation d'un circuit niveau porte a un coût prohibitif ; l'utilisation d'émulateurs de jeux d'instructions permet de le réduire dans une certaine mesure (souvent insuffisante), tandis que la simulation à des niveaux d'abstraction plus élevés est nettement plus rapide, au détriment de la précision des résultats ; le prototypage sur FPGA offre d'excellentes performances, mais peut s'avérer compliqué pour les très gros designs
6. mise au point : un bug non descellé avant que la totalité du système ne soit en place va souvent être difficile à trouver et à résoudre
7. lorsque l'on s'intéresse aux systèmes embarqués, des contraintes supplémentaires liées à l'environnement dans lequel le système se trouve (ex : consommation) viennent encore noircir le tableau
8. multiplicité des outils et modèles : le caractère disparate des logiciels permettant de contribuer à résoudre certains de ces problèmes rend la gestion de projet difficile ; le flot de conception peut devenir très compliqué du fait des dépendances et/ou incompatibilités entre les outils utilisés

La présente thèse propose une méthodologie qui se veut flexible et adaptée pour attaquer les challenges évoqués plus haut. En particulier, elle permet l'utilisation de diverses méthodes d'exploration de l'espace de design, tout en laissant le designer maître de la *taille* de l'espace de design à explorer, et du niveau d'abstraction dans lequel les différentes parties de son design seront exprimées. Elle se veut unificatrice : il doit être possible d'incorporer au sein d'un même design divers modèles de calcul (MoCs), modèles de performances, traitements heuristiques, etc. de façon aussi simple et élégante que possible.

1.2 Instanciation de composants

La conception de SoCs demande au designer de mélanger un grand nombre de blocs IP au sein d'un même design. Il s'agit d'une tâche pour laquelle les langages de description de matériel classiques et encore largement utilisés, tels que le VHDL [41] ou Verilog [42], commencent à montrer leurs limites. En effet, bien que ces langages intègrent la possibilité de définir des composants et de les instancier hiérarchiquement, souvent le câblage des interfaces est assez fastidieux et verbeux. Similairement, les mécanismes de généricité qu'offrent les HDL bas niveau, bien que très efficaces pour certains paramètres tels que les tailles des entrées et des sorties, peuvent s'avérer maladroits, comme nous le verrons, lorsqu'ils concernent des aspects plus subtiles du comportement du bloc matériel. Pour toutes ces raisons, un nombre conséquent de techniques plus adaptées à l'instanciation massive de blocs matériels paramétrables ont été développées.

Dans les sections qui suivent, les plus significatives d'entre elles seront analysées un peu plus en détail, exemples à l'appui lorsque ce sera possible.

1.2.1 Composants génériques

Avant de détailler les techniques alternatives, il convient de faire la lumière sur ce qu'il est possible et n'est pas possible de faire avec du VHDL ou du Verilog standard. Il s'agit en effet des langages qui dominent l'industrie, et qui ont une longue histoire durant laquelle ils ont fait l'objet d'attention de la part d'un grand nombre d'acteurs du marché. On peut donc les considérer comme matures. Les outils d'EDA implémentent ces deux langages de façons souvent assez robuste, malgré des différences sur la sémantique exacte de certaines constructions, notamment en synthèse à partir du VHDL [96].

Nous avons choisi un exemple de bloc IP pour illustrer les problématiques souvent rencontrées au niveau de l'instanciation. Il s'agit d'un switch tel qu'on peut en trouver dans les NoCs matriciels. Notre cahier des charges est le suivant :

1. le switch comporte quatre ports

1.2. INSTANCIATION DE COMPOSANTS

```
1 package my_noc is
2   type priority_policy_t is
3     (priority_policy_alpha , priority_policy_beta , priority_policy_gamma);
4   type switch_interface_t is
5     (switch_interface_a , switch_interface_b);
6
7   type packet_header_t is record — ...
8   end record;
9   type packet_header_array_t is array(natural range<>) of packet_header_t;
10
11  function compute_priority_alpha(hdrs: packet_header_array_t) return natural;
12  function compute_priority_beta(hdrs: packet_header_array_t) return natural;
13  function compute_priority_gamma(hdrs: packet_header_array_t) return natural;
14  function compute_priority(
15    policy: priority_policy_t;
16    hdrs: packet_header_array_t
17  ) return natural;
18
19  type switch_in_a_t is record — ...
20  end record;
21  type switch_out_a_t is record — ...
22  end record;
23  type switch_in_b_t is record — ...
24  end record;
25  type switch_out_b_t is record — ...
26  end record;
27 end package; — my_noc
28
29 package body my_noc is
30   function compute_priority(
31     policy: priority_policy_t;
32     hdrs: packet_header_array_t
33   ) return natural is
34   begin
35     case policy is
36       when priority_policy_alpha => return compute_priority_alpha(hdrs);
37       when priority_policy_beta => return compute_priority_beta(hdrs);
38       when priority_policy_gamma => return compute_priority_gamma(hdrs);
39     end case;
40   end;
41   — ...
42 end package body; — my_noc
```

FIGURE 1.4 – my_noc.vhd

```
1 entity switch is
2   generic(buffer_size: natural := 8;
3     priority_policy: priority_policy_t := priority_policy_alpha;
4     switch_interfaces: array (3 downto 0) of switch_interface_t;
5   );
6   port( clk: in std_ulogic;
7     reset: in std_ulogic;
8     switch_ins_a: in array(3 downto 0) of switch_in_a_t;
9     switch_outs_a: out array (3 downto 0) of switch_out_a_t;
10    switch_ins_b: in array (3 downto 0) of switch_in_b_t;
11    switch_outs_b: out array (3 downto 0) of switch_out_b_t;
12  );
13 end entity; — switch
14
15 architecture rtl of switch is — ...
16 begin — ...
17 end; —architecture rtl of switch
```

FIGURE 1.5 – switch.vhd

```

1  'ifndef MY.NOC.VH
2  'define MY.NOC.VH
3
4  'define SWITCHINTERFACE_A          0
5  'define SWITCHINTERFACE_B          1
6
7  'define SWITCHINTERFACE_A.IN_LENGTH 48 // example
8  'define SWITCHINTERFACE_A.OUT_LENGTH 33 // example
9  'define SWITCHINTERFACE_B.IN_LENGTH 36 // example
10 'define SWITCHINTERFACE_B.OUT_LENGTH 33 // example
11
12 'endif // !MY.NOC.VH

```

FIGURE 1.6 – my_noc.vh

```

1  module priorities ();
2
3  function compute_priority_alpha; // ...
4  endfunction;
5
6  function compute_priority_beta; // ...
7  endfunction;
8
9  function compute_proirity_gamma; // ...
10 endfunction;
11
12 end module;

```

FIGURE 1.7 – priorities.v

```

1  'include "my_noc.vh"
2
3  module switch(
4      input clk,
5      input reset,
6      input [4*SWITCHINTERFACE_A.IN_LENGTH-1:0] switch_ins_a,
7      output [4*SWITCHINTERFACE_A.OUT_LENGTH-1:0] switch_outs_a,
8      input [4*SWITCHINTERFACE_B.IN_LENGTH-1:0] switch_ins_b,
9      output [4*SWITCHINTERFACE_B.OUT_LENGTH-1:0] switch_outs_b
10 );
11
12 parameter INTERFACE1 = SWITCHINTERFACE_A;
13 parameter INTERFACE2 = SWITCHINTERFACE_A;
14 parameter INTERFACE3 = SWITCHINTERFACE_A;
15 parameter INTERFACE4 = SWITCHINTERFACE_A;
16 parameter BUFFER_SIZE = 8;
17
18 'ifndef COMPUTE_PRIORITY
19 'include "priorities.v"
20 'define COMPUTE_PRIORITY          compute_priority_alpha
21 'endif
22
23 // ...
24
25 'undef COMPUTE_PRIORITY
26 endmodule

```

FIGURE 1.8 – switch.v

2. chaque port doit pouvoir être câblé de deux façons¹, et il doit être possible de choisir individuellement le câblage de chaque port, indépendamment des autres
3. on souhaite pouvoir spécifier une politique de priorité des paquets, idéalement via une fonction définie par le designer qui instancie le bloc
4. enfin, la taille du buffer stockant les paquets en attente d'être traités doit pouvoir être spécifiée

Les listings des figures 1.4 et 1.5 permettent de se faire une idée de ce que demande l'implémentation d'un tel bloc en VHDL. Le caractère fortement typé du VHDL se traduit par une certaine verbosité, souvent rentabilisée par un gain en clarté. Il y a essentiellement deux façons de procéder pour gérer les politiques de priorités :

- La première est de se limiter à un ensemble de politiques, et de spécifier celle utilisée en passant un paramètre générique. C'est ce cas qui a été illustré sur les figures 1.4 et 1.5. On définit deux types énumérés permettant le paramétrage de la politique de priorité et des interfaces du switch, un type correspondant à l'information contenue dans un en-tête de paquet, et trois fonctions correspondant à trois politiques de priorités prédéfinies. La fonction `compute_priority`, qui est celle qui sera utilisée dans le code du switch, appellera de façon adéquate l'une de ces fonctions suivant le paramètre générique que l'utilisateur aura passé à l'instanciation du bloc. En conséquence, la gestion de multiples politiques de priorités ne pose pas de problème, à condition de se cantonner à celles prévues par le designer du switch. En effet, en VHDL'00, il n'est pas possible de passer une fonction ou un composant en argument générique lors de l'instanciation d'un bloc.
- Dans les cas où il est impératif que l'utilisateur puisse spécifier un comportement non prédéfini, on aurait tendance à laisser le designer créer un bloc séparé effectuant le calcul en question, puis à le raccorder au bloc à paramétrer par des ports dédiés, ou bien utiliser un sous-composant ordinaire et laisser l'utilisateur en définir une nouvelle architecture, puis choisir celle-ci par le biais d'une configuration. Ces approches posent néanmoins souvent problème. Si on utilise un bloc séparé, alors le bloc à paramétrer ne choisit plus combien de blocs auxiliaires instancier, et les mécanismes d'optimisation de l'outil de synthèse sont potentiellement moins efficaces. De plus, la complexité induite par ce nouveau câblage peut être significative, et constitue éventuellement une source d'erreur supplémentaire pour le designer. Si on utilise la méthode basée sur les configurations, on s'expose potentiellement à des lourdeurs si le bloc à paramétrer est complexe. L'utilisateur peut en effet alors devoir spécifier la configuration pour toute la hiérarchie de composants sous-jacents, ce qui encore une fois peut être source d'erreur, en plus d'exposer des détails d'implémentation qu'il aurait été préférable de cacher. De plus, le paramétrage est séparé de l'instanciation dans le code source, ce qui n'est pas particulièrement logique ; cela a pour effet que l'utilisateur peut penser que le paramétrage n'est pas possible s'il se contente de regarder les ports et les paramètres génériques du composant plutôt que de se référer à sa documentation (si tant est qu'elle existe).

Les interfaces entre le switch et ses voisins sont définies par le biais d'enregistrements. Il s'agit d'un motif de conception fréquent dans les modèles VHDL, dont l'utilité principale est de réduire la verbosité de l'instanciation d'un composant. Étant donné le caractère général du langage, l'implémentation d'un bloc synthétisable implique de se borner à un sous-ensemble du VHDL. Généralement, on utilise pour cela deux processus, l'un dit synchrone sensible aux fronts d'horloge, et l'autre dit combinatoire sensible aux sorties des bascules [26]. Alternativement, certains outils de synthèse [75] acceptent en entrée un design contenant des processus parsemés d'instructions du type `wait until rising_edge(clk)`, la création de registres correspondant aux variables et la construction des machines à état sous-jacentes aux différents processus devenant implicites. Dans tous les cas, il est important (et souvent non trivial) de savoir où commence et où s'arrête le

1. on peut par exemple supposer que l'on souhaite pouvoir connecter les switches en point-à-point, mais également les relier à un bus standard tel que AMBA [1] ; si les interfaces sont suffisamment différentes, il n'est effectivement pas rentable du point de vue des performances d'utiliser un simple bloc convertisseur, comme on le fait souvent pour justement éviter les problèmes d'instanciation liés aux composants aux interfaces paramétrables

sous-ensemble synthétisable par l'outil. Malgré l'omniprésence du VHDL dans le monde de l'EDA, un certain nombre de pratiques sous-optimales sont courantes, comme l'usage des types résolus ou du reset asynchrone [46].

Les listings des figures 1.6 à 1.8 illustrent le même bloc mais cette fois modélisé en Verilog. Contrairement à VHDL, Verilog a été en premier lieu pensé comme un langage de simulation plutôt que de formalisation de cahiers des charges. En conséquence, il est plus concis et emprunte beaucoup aux langages d'ingénierie logicielle tels que le C. Le système de typage est par contre minimaliste pour ne pas dire pauvre, puisqu'il se limite à de la logique 4-états, sans possibilité pour l'utilisateur de définir de nouveaux types ou sous-types de données.

Comme on peut le voir à la figure 1.8, en l'absence d'enregistrements, on passe de longues chaînes de bits qu'on va «désérialiser» à l'intérieur du bloc pour limiter le nombre de ports. Verilog est doté d'un mécanisme de paramètres qu'il est possible de surcharger, fonctionnant comme l'équivalent des paramètres génériques avec valeur par défaut du VHDL. Comme il n'est pas non-plus possible de passer de tâche, fonction ou module en paramètre par cette voie là, nous allons avoir recours au préprocesseur. Verilog ne prévoit aucun mécanisme de compilation séparée ; le moyen adopté pour réutiliser du code est justement son inclusion via des directives du préprocesseur. Ce préprocesseur est très similaire à celui du C ANSI, en un peu plus rudimentaire (de par la non-possibilité de définir des macros acceptant des arguments, ou l'absence de certaines directives telles que `#warning` ou `#error`). Les pratiques recommandées dans son utilisation [90] empruntent partiellement au C, à ceci près que l'inclusion en milieu de fichier est nettement plus courante. Dans notre exemple, nous avons utilisé une technique inspirée des *x-macros* du C (parfois appelées supermacros [14]) pour paramétrer *de facto* notre bloc par une politique de priorités. Avant d'inclure le fichier «switch.v», l'utilisateur a la possibilité de spécifier sa propre fonction de priorité en associant son nom à la macro `COMPUTE_PRIORITY`. Si cette macro n'est pas définie à l'inclusion de «switch.v», elle prend comme valeur par défaut l'une des fonction de priorité prédéfinies.

Mentionnons au passage l'existence de System Verilog [6], qui dispose d'une syntaxe proche de celle de Verilog mais offre la possibilité de définir des types énumérés et des enregistrements, dispose d'un système de classes, et d'un assez grand nombre d'autres ajouts, principalement destinés à la simulation. Si nous avons voulu implémenter notre switch en System Verilog, la façon la plus propre de faire aurait été de définir les politiques de priorité comme descendantes d'une classe abstraite. Le design résultant n'aurait néanmoins pas été synthétisable.

Enfin, le lecteur attentif aura remarqué que les standards VHDL et Verilog cités plus haut sont ceux de 2000 et 2001, respectivement. Ce choix est dû au fait que l'essentiel du code écrit de nos jours dans ces langages se conforme à ces versions des standards. Cela dit, certaines des faiblesses mentionnées dans cette section ont été corrigées en totalité ou en partie dans les versions subséquentes [43, 44], qui ne sont hélas pas encore supportées par la vaste majorité des outils d'EDA répandus. Notamment, VHDL'08 autorise le passage de fonctions et de types par le biais de paramètres génériques. Ces évolutions semblent s'inscrire dans une tendance plus générale à l'ajout d'éléments fonctionnels dans les versions récentes ou à venir des langages de programmation, illustrée par exemple par l'incorporation d'inférence de types ou de *lambda expressions* dans les langages Java [30] et C++ [15, 47].

1.2.2 Fichiers annexes, annotations du source, génération

Une façon de faciliter le travail d'instanciation de quantités significatives de composants est d'avoir recours à des informations fournies par le designer sous une forme alternative ou additionnelle plus appropriée que celle des HDL classiques. Cela peut se faire soit par l'utilisation de fichiers annexes (souvent dans des formats dont l'analyse syntaxique est simple et capables d'organiser les données sous forme de hiérarchies ou des graphes, comme par exemple XML ou UML ; MARTE [9, 22] est un exemple particulièrement notable de telle approche pour la modélisation niveau système), soit par le biais d'annotations insérées dans le code source par le biais de directives (par exemple, `#pragma` en C) ou de commentaires spécialement formatés. Ces informations seront ensuite utilisées pour produire des netlists, du code HDL intermédiaire, etc.

Un exemple très caractéristique d'une telle approche est Xilinx Platform Studio (XPS) et les outils associés. Les composants constituant le système que le designer décrit sont encapsulés puis instanciés via un fichier MHS (Microprocessor Hardware Specification) qui sera ensuite utilisé par Platgen (le générateur de plateformes) au cours de la création du bitstream. Il est possible de paramétrer de cette façon chaque bloc instancié, y compris dans la façon dont il est connecté au reste du monde. Il en résulte une spécification des composants instanciés plus concise et plus sûre (un câblage incohérent pouvant être détecté par l'outil). Platgen permet également de gérer de façon cohérente à l'échelle du système l'espace d'adressage tel qu'il sera vu par les microprocesseurs (et donc les drivers). En revanche, il est toujours impossible de paramétrer un bloc par une fonction, un autre bloc, etc.

La génération de code intermédiaire à partir d'information sur la topologie des interconnexions est également particulièrement utilisée dans le cas des NoCs. C'est par exemple le cas du NoC Arteris. Le terme Arteris désigne un tout comportant à la fois la bibliothèque de composants permettant de construire le réseau sur puce à proprement parler, le protocole que ces composants vont utiliser pour communiquer, et l'outil permettant l'instanciation du réseau sur puce. À partir de la liste des composants à interconnecter et de leurs interfaces (potentiellement hétérogènes), cet outil produit une représentation du NoC sous forme de fichiers IP-XACT [45, 57], un format de description de design basé sur l'XML et destiné à être utilisé directement par les outils d'EDA.

Il ne faut également pas négliger les approches qui «détournent» de leur usage des langages de programmation logicielle pour décrire du matériel. Il y a dans ces cas deux façons possibles de procéder : écrire puis utiliser un compilateur capable de traduire le langage en question sous une forme exploitable (par exemple, VHDL intermédiaire) ou faire en sorte d'obtenir le résultat souhaité directement en exécutant le programme compilé via un compilateur classique. Dans la première catégorie, on peut notamment citer un certain nombre d'outils dont l'optique est d'utiliser le C comme un HDL : ImpulseC de Impulse Accelerated Technologies, PICO de Synfora, ou encore Catapult C de Mentor Graphics en sont trois exemples. L'utilisation du C dans le domaine de l'EDA est motivée par son caractère relativement bas niveau d'une part et l'omniprésence de ce langage dans l'ingénierie logicielle d'autre part. Ultiment, si un même langage peut être utilisé pour décrire le matériel comme le logiciel, on peut également espérer automatiser le partitionnement matériel/logiciel, puisque le même code pourra être utilisé pour décrire les deux. Il existe également des exemples d'outils prenant en entrée d'autres langages, tels que Matlab.

Parmi les méthodes ne nécessitant pas de compilateur particulier, en dehors de celles basées sur les langages fonctionnels et qui seront détaillées dans la section suivante, on citera (sans être exhaustif) SystemC et Ptolemy II. SystemC [5] est un cas un peu particulier, puisqu'il existe des outils de synthèse analysant lexicalement et syntaxiquement du code SystemC en vue de produire une netlist ou des binaires utilisables par un simulateur. Cependant, l'un des buts premiers du SystemC a été d'offrir un environnement de simulation sous forme de bibliothèque et runtime C++ classique, de sorte à ce que les modèles comportementaux de composants puissent incorporer du code C++ ordinaire ou faire appel à du code objet existant. En termes de niveau d'abstraction, SystemC laisse dans une certaine mesure le choix au designer qui peut opter pour des modèles clockés, du TLM, etc. Des approches cherchant à doter SystemC de MoCs aux propriétés plus spécifiques peuvent également être dénombrées [37, 74]. Ptolemy II [84] est, quant à lui, basé sur Java. Il se compose d'une riche bibliothèque de classes correspondant entre autres à un jeu de composants se comportant comme de boîtes noires, et à un ensemble diversifié de modèles de calcul. Le programmeur peut utiliser Java pour spécifier son design, en raccordant les composants entre eux, voire en en définissant de nouveaux si nécessaire. Il est également possible de définir le design par le biais d'un front-end graphique, ainsi que du langage MoML, basé sur XML et propre à Ptolemy II, ce qui le rapproche un peu des méthodes évoquées dans les paragraphes précédents.

1.3 Description de matériel et programmation fonctionnelle

Dans cette section, nous allons passer en revue quelques unes des différentes approches existantes pour la spécification et la vérification de matériel utilisant dans langages fonction-

nels. Si nous insistons plus particulièrement sur cette classe de méthodes, c'est parce que notre méthodologie, qui sera exposée dans les chapitres subséquents de ce manuscrit, s'inscrit dedans (comme nous le verrons un peu plus loin).

D'une façon générale, outre leur adéquation pour la description de constructions parallèles, l'intérêt de l'utilisation de langages fonctionnels typés pour la description de matériel [61, 88, 89] réside dans leur adéquation pour manipuler le genre de représentations de données successible d'apparaître dans ce domaine. Le *pattern matching* rend l'expression de transformations de graphes aisée, et le lambda-calcul permet la construction de bibliothèques de combinateurs efficaces et flexibles. De plus, les techniques telles que la compilation dynamique, la construction de langages embarqués et la *quasi-quotation* (qui permet à l'utilisateur de définir sa propre syntaxe pour certaines parties du programme) trouvent leur utilisation naturellement dans la description de matériel ou dans les domaines connexes tels que la simulation. Enfin, la possibilité de construire facilement des langages embarqués autorise également le programmeur à se définir ses propres boîtes à outils adaptées à la résolution de certaines classes de problèmes.

1.3.1 Description de matériel

1.3.1.1 Bluespec SystemVerilog

Bluespec SystemVerilog (BSV) [2, 10] est un langage de description de matériel fonctionnel très proche de Haskell, mais doté de constructions et mots clés propres à la représentation de circuits sous forme RTL. Il s'agit probablement de l'approche fonctionnelle la plus connue dans le domaine de l'EDA. Le compilateur Bluespec est un outil commercial, et accepte en entrée SystemVerilog en plus du langage BSV. Il est important de noter que, malgré son nom, le langage BSV ne partage avec SystemVerilog qu'une similarité syntaxique superficielle, et n'est absolument pas compatible avec lui au niveau du code source (il est cependant possible de mélanger des unités de compilation BSV et SystemVerilog dans un même design).

BSV constitue un bon exemple permettant de comprendre l'intérêt de l'utilisation de combineteurs dans le domaine du design matériel. Généralement, les descriptions BSV sont plus concises et moins propices aux erreurs que leurs équivalents dans les approches impératives ou déclaratives classiques. Cela est dû à un ensemble de facteurs. Le comportement de circuits est déterminé par un ensemble de *règles* (rangées par *modules*) qu'il est possible de composer et de traiter aisément. BSV est accompagné d'une bibliothèque de types/composants utilisant judicieusement les possibilités du langage et aidant à réduire au maximum la quantité de logique de contrôle exprimée par l'utilisateur, lui permettant de se focaliser sur la fonctionnalité du circuit. Notons en outre que BSV est apte à répondre au cahier des charges que nous nous sommes fixés pour le design du switch de la section 1.2.1. En particulier, la politique de gestion des priorités peut être passé directement à l'instanciation du switch soit par le biais du type `Action`, soit par le biais du type `Rules`, tous deux définis par le langage et utilisables au même titre que les plus classiques `Bit#(n)`, `Int#(n)`, etc.

1.3.1.2 Lava

Lava [17] est une autre approche fonctionnelle au niveau RTL. Il s'agit cette fois d'une simple bibliothèque Haskell définissant des types adéquats pour la représentation de portes logiques et des fonctions permettant leur manipulation. Il est ainsi possible de simuler, de vérifier et de générer du code intermédiaire à partir des designs. Comme Lava peut être présenté comme un langage dans un langage, on parle d'EDSL (*Embedded Domain Specific Language*). La spécification des designs eux mêmes se fait via l'utilisation d'un petit nombre de blocs élémentaires type portes logiques (`and2`, `or2`, `inv`, etc.) raccordés par le biais d'opérateurs ou de fonctions (souvent appelées de façon infixée en pratique), tels que `>>`, permettant de sérialiser deux blocs, ou `par2`, les mettant en parallèle. Ce genre de description niveau «porte» peut sembler excessivement proche du matériel, mais une utilisation judicieuse de fonctions d'ordre supérieur, aussi bien celles offertes par Lava que celles du préluce Haskell (c.f. section A.5) ou celles définies par l'utilisateur, lui enlève beaucoup de

sa lourdeur. Le fait de passer des blocs (ou des fonctions retournant des blocs) en arguments à d'autres est un des aspects centraux et idiomatiques de la philosophie du langage (donc, là aussi, nous n'aurions pas eu de problèmes pour instancier notre switch).

Parmi les variantes de Lava existantes, certaines, dont celle de Xilinx, font usage du polymorphisme via le système de classes de types de Haskell afin d'interpréter différemment chaque design en fonction de la tâche à réaliser. Les primitives (portes, fonction de sérialisation, etc.) sont définies dans des classes dont les instances vont correspondre aux types de données nécessaires pour simuler, vérifier, ou générer du code intermédiaire à partir du design. Ces classes sont monadiques², notamment parce que les monades sont un excellent moyen d'abstraire les détails de fonctionnement interne d'un langage embarqué dont les constructions peuvent présenter des effets de bord [97].

1.3.1.3 ForSyDe

ForSyDe (*Formal System Design*) [82] est, tout comme Lava, un EDSL basé sur Haskell. Contrairement aux deux approches précédentes, ForSyDe offre plusieurs modèles de calcul définissant les modalités et contraintes de communication entre processus. On dénombre ainsi un MoC synchrone, un MoC sans horloge, et un MoC dédié aux signaux continus (par opposition aux signaux binaires). Il est possible de simuler ou de générer du code VHDL à partir des designs.

La particularité de ForSyDe est de reposer sur un certain nombre d'extensions de Haskell supportées par le compilateur GHC (Glasgow Haskell Compiler), à savoir les classes de types à arité supérieure à un, la surcharge d'instances, mais aussi et surtout *Template Haskell* (TH). Les deux premières sont *de facto* acceptées au sein de la communauté de Haskell (elles sont en particulier implémentées par Hugs, l'autre compilateur Haskell majeur); la dernière, TH, est spécifique à GHC version 6 et subséquentes, et permet la réflexion et la compilation dynamique. Le design spécifié par l'utilisateur n'est pas évalué, comme en Lava, mais est immédiatement converti en arbre syntaxique abstrait qui va ensuite être transformé vers ce que l'on cherche à en obtenir : soit une fonction de simulation compilée dynamiquement, soit du code VHDL intermédiaire.

1.3.1.4 SML-Sys

SML-Sys [61] est une approche orientée vers la simulation et implémentée dans le langage Standard ML, un langage fonctionnel impur [62]. Contrairement aux approches précédentes, il n'est pas possible de générer du code à partir de designs exprimés via SML-Sys. L'atout majeur de l'approche réside dans sa capacité à gérer des designs spécifiés dans une grande variété de modèles de calcul pouvant communiquer entre eux. L'intérêt du paradigme fonctionnel dans ce contexte réside essentiellement dans la très claire séparation entre la partie communication et calcul du design.

1.3.1.5 ClaSH

ClaSH [13] est une méthode de conception de circuits synchrones basée sur le langage Haskell. Contrairement à Lava ou à ForSyDe, qui sont des langages embarqués ayant vocation à être utilisés en conjonction avec des compilateurs ou interpréteurs classiques tels que GHC ou Hugs, ClaSH utilise un compilateur séparé spécifique. Cela lui permet de synthétiser les «structures de contrôle» natives de Haskell telles que les clauses if-then-else ou le pattern matching. Le résultat est un code très concis et peu propice aux erreurs.

1.3.2 Vérification formelle

1.3.2.1 Méthodes classiques

D'ores et déjà, il convient de mentionner l'existence ou la possibilité d'utiliser des procédés de vérification formelle «classiques» dans les paradigmes fonctionnels. C'est en particulier le cas dans Lava, qui permet la vérification de propriétés de circuits combinatoires en testant toutes

2. Le principe des monades est rappelé section A.4.

les combinaisons d'entrées possibles. Ce genre de procédé exhaustif est bien sûr limité aux circuits de faibles tailles. Il existe d'autres algorithmes permettant d'effectuer de la vérification ou du test d'équivalence au niveau netlist et capables de prendre en compte l'existence d'éléments mémorisants (par exemple, [49]) qui pourraient être aisément mis en œuvre dans le contexte de la programmation fonctionnelle. Par ailleurs, les outils d'EDA existants peuvent être utilisés si une approche fonctionnelle donnée est capable de produire du code intermédiaire.

1.3.2.2 Méthodes basées sur le système de types

La correspondance de Curry-Howard est le constat qu'en lambda-calcul typé, un type et une fonction s'y conformant se comportent exactement comme une formule et sa preuve [39]. Il en résulte qu'un système de types suffisamment expressif peut servir à définir et à prouver toutes les propriétés exprimables dans une large variété de systèmes logiques.

Ce constat a motivé le développement de langages sensiblement plus richement typés que ceux auxquels on est habitués. Une raison pour laquelle les langages conventionnels se cantonnent aux types algébriques et à leurs variantes est que les algorithmes de vérification de types s'exécutent alors en temps bornés (et terminent forcément). De plus, les algorithmes d'inférence de types, tels que celui de Hindley-Minler (très proche de celui utilisé par Haskell), ne fonctionnent que sur des types algébriques ou certaines de leurs généralisations qui restent simples. Les langages construits autour de types plus complexes partent du postulat que les gains offerts par ces types excèdent les désavantages précédemment mentionnés.

Il existe des langages fonctionnels spécialisés dans la preuve de théorèmes, tels que Agda [68] ou encore Coq [94]. Ces langages sont dotés de bibliothèques standards dont l'utilité première est d'aider à la rédaction de preuves. On peut conjecturer que des bibliothèques de ce genre de constructions pourraient être implémentées dans tous les langages disposant d'un système de types suffisamment riche.

Cayenne [12] est un exemple de langage de programmation à usage général permettant le typage dépendant (c.-à-d. autorisant le paramétrage des types par des valeurs, et non plus seulement par d'autres types). Sa syntaxe emprunte des éléments aux langages fonctionnels les plus répandus (Haskell, ou dans une moindre mesure Standard ML) comme aux langages impératifs et objets (C/C++, Java). Les types dépendants accroissent considérablement l'expressivité d'un système de types dans la mesure où ils équivalent à l'usage de propositions quantifiées dans les formules que les types expriment, mais aussi se traduisent plus simplement par une plus grande liberté offerte au programmeur dans l'organisation de son code tout en lui garantissant une sécurité de typage maximale, un avantage qu'il ne faut pas négliger.

Enfin, citons Ω mega [86], un exemple de langage fonctionnel richement typé qui peut être utilisé pour la description de matériel [85, 87]. Ω mega permet de définir des fonctions dans l'espace des types comme dans celui des données. Bien qu'il ne s'agisse pas techniquement de typage dépendant, les possibilités de Ω mega en sont quasi-identiques dans la mesure où il suffit de traiter les types comme des valeurs pour arriver à des fins similaires. La catégorisation est hiérarchique (autrement dit, il est possible de définir des «types de types», par exemple). Une grande partie de la syntaxe du langage lui-même et nombre de ses caractéristiques proviennent de Haskell. En particulier, Ω mega dispose des mêmes raccourcis syntaxiques que Haskell pour faciliter l'écriture de monades, et d'une syntaxe de métaprogrammation calquée sur celle de Template Haskell. En plus de prouver la correction des différentes parties du design, les types dans Ω mega peuvent en plus servir à garantir d'autres propriétés, telles que la conformité du design à des contraintes de taille ou de consommation, par exemple.

1.4 Exploration architecturale

La recherche en exploration architecturale va essentiellement dans deux directions distinctes mais complémentaires : le parcours de l'espace de design et l'évaluation des performances. L'espace de design peut être parcouru manuellement, automatiquement ou suivant une approche mêlant

les deux. L'évaluation de performances doit être aussi rapide et précise que possible. Elle peut être réalisée en simulant ou en émulant le design (auquel cas les performances mesurées sont exactes), ou bien extraite à partir d'un modèle simplifié. Il est important de noter que, pour une efficacité maximale, l'exploration architecturale doit concerner aussi bien la composante matérielle que logicielle d'un système, si tant est qu'on puisse influencer sur les deux [78].

Le parcours de l'espace de design, lorsqu'il est automatisé, peut être exhaustif ou localement exhaustif dans les cas où le temps nécessaire pour évaluer les performances est suffisamment court. Un exemple de telle approche est [79], présentant une méthode de recherche de topologie optimale de réseau sur puce qui teste toutes les combinaisons possibles et utilise un modèle d'évaluation des performances très agressif permettant de passer à peine quelques secondes sur une configuration donnée. [27], quant à lui, présente une approche pour configurer les SoCs paramétrés, en les partitionnant à l'aide d'un graphe de dépendances préalablement construit puis en effectuant une recherche exhaustive sur chaque partie.

Lorsqu'un parcours exhaustif est trop coûteux, ou lorsque l'on souhaite tout simplement réduire la durée de l'exploration, on a tendance à avoir recours à des méthodes stochastiques. Les algorithmes génétiques semblent être le plus fréquemment utilisés dans ces cas là. On pourra notamment citer [11] pour la configuration de SoCs, [93] pour celle des hiérarchies mémoires, [18] qui concerne plus particulièrement la synthèse des DSP, etc. Les algorithmes génétiques (tout comme d'autres méthodes stochastiques telles que le recuit simulé) interviennent aussi au moment du *mapping* sur la technologie cible ; [53] résume certaines de ces techniques dans le contexte des FPGA. Enfin, il est intéressant de noter que ce genre de méthodes peut également servir à déterminer les paramètres et options optimales des outils de synthèse et des compilateurs utilisés, en fonction des critères recherchés.

En terme d'évaluation des performances, en plus des simulations à l'échelle du système, on peut énumérer des approches aussi diverses que l'extraction des paramètres des composants individuels (processeurs, bus, etc.) via simulation *cycle accurate* dans le but d'en extraire des modèles plus simples, ensuite utilisés pour simuler le SoC entier [27], ou bien celles permettant de **mapper** un design de taille importante sur un ou plusieurs FPGAs [98].

Pour pouvoir mener l'exploration à son terme, il faut avoir un idée des aspects que l'on souhaite optimiser. À chaque design correspond un compromis entre fonctionnalité, quantité de logique nécessaire, puissance consommée, etc. Ce compromis peut être trouvé en informant l'outil des contraintes de place, de consommation, etc., ou bien en faisant en sorte que l'outil se contente de générer de logs permettant au designer de réaliser lui-même les choix architecturaux.

Chapitre 2

Méthodologie proposée

2.1 Objectifs

Afin de répondre aux besoins spécifiques qu'impose la conception de SoCs complexes, nous proposons une méthodologie de design originale. Nous nous sommes pour cela fixés un certain nombre d'objectifs à atteindre. À notre connaissance, aucune des méthodes de design actuelles ne satisfait simultanément et de façon acceptable à tous ces objectifs.

2.1.1 Instanciation et explorabilité

Un des problèmes majeurs du design d'un SoC, comme nous l'avons vu au chapitre précédant, est le paramétrage des blocs matériels hétérogènes qu'il comporte. Nous partons du principe qu'il est de moins en moins acceptable de demander au concepteur de comprendre l'implication qu'aura le paramétrage d'un bloc en particulier sur le reste du système. Face à ce constat, nous avons pris le parti de :

- intégrer la possibilité pour le designer d'exprimer les différentes implémentations alternatives d'un bloc IP via des primitives prévues à cet effet
- fournir à l'utilisateur un certain nombre de méthodes d'exploration architecturale afin de parcourir l'espace de design ainsi obtenu

Les méthodes d'exploration en question peuvent être automatiques ou manuelles, comme on le verra plus loin. Afin d'accélérer l'étape d'exploration de l'espace de design, le designer pourra pondérer les alternatives en fonction de la probabilité qu'il estime que chacune d'entre elle a d'être le choix optimal. Il s'agit d'une mesure heuristique qu'il peut s'il le désire ignorer, en affectant un poids identique à toutes.

Un corollaire de tout cela est que l'utilisateur peut utiliser des blocs matériels comme des boîtes noires et espérer que l'algorithme d'exploration stochastique qu'il aura choisi d'utiliser les paramètre correctement. Un autre corollaire, et non des moindres, est qu'il sera facile de faire en sorte que la topologie des interconnexions soit explorée en même temps que les IPs sont paramétrées, de sorte à ce qu'elle soit adaptée au trafic que vont générer les blocs une fois configurés. C'est un élément fondamental permettant une véritable intégration des communications au sein de la conception système. Procéder autrement a tendance à produire des goulots d'étranglement réduisant sévèrement les performances globales du système.

L'utilisation d'algorithmes d'exploration automatique implique bien sûr que l'utilisateur fournisse une fonction de coût à maximiser¹.

1. On pourrait aussi envisager une optimisation multiobjectif produisant un ensemble de paramétrages appartenant à la frontière de Pareto (ce que nous ne ferons pas pour rester dans des cas simples) ; en tout cas, il faut que l'utilisateur donne un moyen à l'algorithme pour évaluer un paramétrage donné.

2.1.2 Multiplicité des niveaux d'abstraction

Il est intéressant que la totalité du système soit exprimée dans le formalisme proposé pour que la phase d'exploration puisse se faire via l'estimation des performances globales. Cela implique par exemple que le designer exprime non seulement le matériel, mais également la partie logicielle de son SoC. Cela comporte de nombreux avantages, comme le fait de pouvoir déléguer le partitionnement matériel/logiciel à l'outil, ou encore de le laisser explorer la partie software en plus de la partie hardware.

Une façon de rendre ceci possible est de permettre de définir de multiples MoCs, chacun doté de primitives de modélisation spécifiques. Ainsi, des modèles de calcul bas niveau spécialisés dans la description fine de matériel pourront être utilisés pour spécifier les parties du système critiques au niveau des performances, tandis que les tâches moins décisives seraient exprimés dans des MoCs plus souples, adaptés à la fois à la description de matériel et de logiciel, la segmentation pouvant alors par exemple être laissée à la charge de l'outil. Pour la réalisation de SoCs comportant un nombre significatif de blocs interconnectés, des MoCs spécialisés dans la communication et garantissant des propriétés utiles sur la latence des message transmis, l'ordre d'arrivée, etc., pourraient être utilisés pour la description à l'échelle du système, tandis que les blocs individuels seraient, eux, potentiellement spécifiés dans des MoCs différents.

En fait, permettre à l'utilisateur de définir ses propres modèles de calcul a des applications bien plus nombreuses que celles données en exemple dans le paragraphe précédent. Les définitions exactes du terme MoC varient, mais au sens où nous l'entendons, à chaque fois qu'un outil propose de définir tout ou partie d'un système via un fichier annexe (texte, XML ou autre), on peut le voir comme l'adjonction d'un nouveau modèle de calcul ou l'extension d'un modèle de calcul existant. Le fait d'exprimer tous ces MoCs dans un même langage tend à simplifier la gestion d'un projet. En effet, l'utilisation de fichiers annexes ou d'annotations non standard dans le code source implique l'ajout de nouvelles étapes et d'outils spécifiques au flot de design. Le développement de tels outils peut être coûteux, et l'utilisation d'outils préexistants peut causer des soucis d'incompatibilité mutuelle constituant une barrière aux progrès de la productivité. La définition de MoCs dans notre formalisme est comparativement aisée, tout comme la construction de primitives de communication entre MoCs. De plus, comme on le verra par la suite, tout modèle de calcul défini dans notre formalisme acquière automatiquement des propriétés intéressantes, au nombre desquelles le caractère explorable du design exprimé par son biais.

En pratique, un MoC dans notre approche se présentera comme un petit langage embarqué spécifique à un niveau d'abstraction ou à un formalisme donné. Le détail des principes de construction des MoCs est présenté dans la suite de ce chapitre, et nous étudierons deux MoCs en particulier au cours du chapitre 3 : la modélisation *bit accurate* sans horloge et la modélisation au niveau RTL avec un seul domaine d'horloge. Les autres MoCs envisageables incluent la modélisation procédurale ou impérative, des approches déclaratives (qui incluent par exemple des jeux de constructions spécialisés dans l'instanciation de bus ou de NoCs, ou ayant vocation à exploiter les ressources spécifiques à une plateforme donnée), ou des formalismes de modélisation plus abstraits mais disposant de propriétés formelles utiles (KPN, etc.). L'explicitation d'aspects tels que le parallélisme peut également faire partie du modèle de calcul. En somme, toutes les constructions liées à la spécification de la fonctionnalité, des communications et éventuellement des détails d'implémentation ont potentiellement leur place dans les MoCs au sens où nous l'entendons.

2.1.3 Utilisation d'un seul langage

L'utilisation d'un seul langage est souhaitable pour à la fois décrire le SoC de l'utilisateur, les MoCs et les méthodes d'exploration. Des MoCs et méthodes d'exploration seront fournies d'office sous forme de bibliothèques, mais l'utilisateur pourra en spécifier de nouveaux à sa guise. Ceci permet de garantir une extensibilité maximale de l'approche, et a vocation à rendre superflus les artifices couramment utilisés sur des projets complexes, tels que l'utilisation de scripts de paramétrage ou de préprocesseurs. Le but est de faire en sorte que l'essentiel des outils couramment utilisés au sein d'un projet puissent être exprimés par le biais de bibliothèques plutôt que de

programmes indépendants les uns des autres. Ce faisant, le développement et l'utilisation de ces outils devraient être facilités, et le risque d'incompatibilité mutuelle entre outils amoindri. Par exemple, un outil qui prenait du VHDL annoté en entrée n'aurait plus à disposer d'un analyseur lexical et sémantique, puisqu'il partirait dorénavant directement l'équivalent de l'arbre sémantique, et deux outils qui reposaient sur des annotations mutuellement incompatibles pourraient être «réconciliés» une fois utilisable sous forme de fonctions bibliothèques. Enfin, comme l'essentiel du flot de développement serait concentré dans un seul et même programme (qui aurait vocation à à la fois décrire le design et le processus le transformant vers une netlist, un modèle simulable, etc.), la gestion de projet en serait simplifiée.

Nous avons choisi d'utiliser un langage fonctionnel pur typé. Les raisons de ce choix ont été détaillées dans le chapitre précédant. Il s'agit de langages adaptés à la représentation et à la transformation de graphes, à l'extension de leur syntaxe par de nouvelles constructions et opérateurs, et offrant souvent des systèmes de typage statiques très intéressants, se traduisant par des notions telles que le polymorphisme ou l'évaluation partielle.

2.1.4 Facilité d'emploi

La bibliothèque de MoCs, composants et primitives fournie d'office à l'utilisateur se veut aussi intuitive à utiliser que possible. Cela implique d'utiliser des notions existantes et bien connues, telles que par exemple les concepts d'architectures et de processus similaires à ceux du VHDL pour le design au niveau RTL. Des exemples seront fournis dans les chapitres suivants.

En pratique, obtenir ce genre de syntaxe proche du style impératif est possible dans un langage purement fonctionnel par l'utilisation de monades.

Il est à noter que concevoir une bibliothèque de modèles de calcul exhaustive et bien conçue (notamment du point de vue des interactions possibles entre les modèles de calcul) est une tâche difficile que nous n'avons hélas pas pu mener de façon complète au cours de cette thèse, faute de temps. Ce manuscrit présente donc quelques exemples de modèles de calcul significatifs ainsi que les principes à suivre pour en créer de nouveaux efficacement, mais ne prétend pas constituer un panorama des MoCs qu'il est souhaitable de prédéfinir. La construction d'une telle bibliothèque s'inscrit dans les perspectives du travail présentement réalisé.

2.2 Dichotomie MoC/Moteur

Au vu des objectifs fixés, il nous a fallu choisir un langage adapté à la programmation monadique. Assez naturellement, nous avons choisi Haskell, le langage fonctionnel typé qui a généralisé l'usage des monades. Haskell présente l'avantage d'être l'un des langages fonctionnels les plus largement utilisés et les plus fédérateurs, bénéficiant d'un grand nombre de bibliothèques déjà écrites. L'aisance de définition d'EDSLs est également l'un des ses points forts. Enfin, la philosophie de design de Haskell et de ses bibliothèques est de mettre en avant le polymorphisme autant que possible (ce qui apporte le même genre de bénéfices que la programmation générique, à savoir de favoriser l'interaction entre composants écrits indépendamment). Le reste de ce manuscrit contiendra beaucoup d'extraits de code écrits en Haskell, et le lecteur est donc invité à prendre connaissance de l'annexe A s'il n'a pas déjà quelques bases dans ce langage.

Les types polymorphes dont le système de classes de Haskell permet la définition sont un très bon moyen de permettre la multiplicité des modèles de calcul. On peut alors définir une classe (au sens Haskell de ce terme) correspondant aux primitives d'un MoC, et y faire correspondre ensuite plusieurs types de données que nous appelons moteurs. Par exemple, un MoC pourra correspondre à une spécification au niveau RTL du design dans un domaine d'horloge donné, et les moteurs associés à ce MoC pourront servir à simuler, émuler, ou convertir dans un autre langage le design. Cette dichotomie est un aspect fondamental de notre méthodologie qui sera détaillé dans les sous-sections qui vont suivre.

Tous les modèles de calcul partagent comme point commun d'être monadiques et explorables. La section 2.2.1 présente les classes et structures de données qui rendent cela possible. Les principes

de construction des MoCs et des moteurs associées sont données dans les sections 2.2.2 et 2.2.3. Les règles à suivre pour l'implémentation d'une bibliothèque efficace de MoCs et moteurs associés sont données dans la section 2.2.4.

2.2.1 Monade d'état explorable

La base du mécanisme d'exploration architecturale proposé est ce que l'on appelle la monade d'état explorable. Il s'agit d'une extension de la monade d'état classique, cette dernière étant définie habituellement de cette façon :

```

1 data State s a = StateC { runState :: s -> (a, s) }
2
3 instance Monad (State s) where
4   — Placer x dans la monade sans altérer l'état :
5   return x = StateC (\t -> (x, t))
6
7   — Altérer l'état via m, puis via k a (a : valeur contenue dans la monade m)
8   m >>= k = StateC (\t -> let (a, t') = runState m t in runState (k a) t')
```

On pourrait tenter d'écrire la même chose en utilisant le lambda calcul typé polymorphe. En empruntant les notations de [81] et en définissant **State** comme $\Delta s. \Delta a. s \rightarrow (a, s)$, on aurait :

$$\begin{aligned} \text{return}[\text{State}[s][a]] &= \lambda x_a. \lambda t_s. (x, t) \\ \text{bind}[\text{State}[s][a]] &= \Lambda a''. \lambda m_{\text{State}[s][a'']}. \lambda k_{a'' \rightarrow \text{State}[s][a]}. \lambda t_s. (k(x)(t') \text{ où } (x, t') = m(t)) \end{aligned}$$

Dans tous les cas, le type **State** est une simple encapsulation d'une fonction exprimant la transformation d'un état de type **s** et renvoyant au passage une valeur de type **a**. Sémantiquement, un *return* spécifie une nouvelle valeur renvoyée sans altérer l'état, tandis qu'un *bind* (>>=) a pour effet, schématiquement, de «composer» ses deux arguments. En d'autres termes, une suite d'instructions dans un «do» va très intuitivement correspondre à une succession d'altérations de l'état **s** qui pourront ensuite être appliquées toutes d'un seul bloc par le biais de la fonction *runState*.

L'intérêt de la monade d'état est d'émuler les «effets de bord». Chaque action monadique y ressemble assez à un appel de procédure ou de fonction impure dans un langage impératif, pouvant altérer tout ou partie de l'état. Dans un langage fonctionnel pur tel que Haskell, qui ne permet pas à l'utilisateur de manipuler d'objets mutables², il s'agit d'un des seuls moyens d'exprimer des constructions altérant leur environnement, quant bien même celui-ci ait une portée limitée et clairement définie (l'état en l'occurrence).

Nous sommes partis du constat que l'utilisation de monades d'état paraît être un bon moyen de représenter différentes parties d'un système électronique. En effet, la possibilité d'utiliser divers types d'états, avec des primitives propres à chacun, permet de définir de diverses façons diverses parties du design, en fonction de leur nature, de leur niveau d'abstraction, et de leurs propriétés théoriques souhaitables (ce que nous englobons sous le terme modèle de calcul³). L'enchaînement de primitives spécifiques à chaque état exprimé par le biais d'un «do» est de plus facile et naturel à lire, et ne dépaysera pas un habitué des langages déclaratifs ou impératifs. Il devient alors intéressant de voir s'il est possible et judicieux d'ajouter de nouvelles constructions à la monade d'état, ce qui permettrait à tous les modèles de calcul d'hériter des propriétés qu'elles définiraient. Une telle propriété est l'explorabilité : si tous les modèles de calcul partagent des primitives d'exploration identiques, tout le design devient naturellement explorable. De plus, l'implémentation de méthodes d'exploration spécifiques devient indépendante de la construction des modèles de calcul/moteurs, rendant ces deux tâches plus faciles, au même titre que le maintien de la bibliothèque

2. Sans tricher tout du moins (l'intégration d'appels à du code C pourrait par exemple permettre de se défaire de cette limitation ; c'est d'ailleurs ainsi que sont implémentées les constructions des monades IO ou ST capables de manipuler des références)

3. Plus exactement, comme déjà spécifié, un MoC sera une *classe de type* contenant des primitives propres à définir une partie du design caractérisée par sa nature, son niveau d'abstraction et ses propriétés. Cela permet de plusieurs type d'états d'implémenter un même MoC, comme nous le verrons plus loin.

Haskell résultante. On se propose donc d'étendre le mécanisme de la monade d'état pour permettre à l'utilisateur d'exprimer s'il le souhaite plusieurs transformations équivalentes à ses yeux sur le plan fonctionnel :

```

1 data ExpState s a = ExpStateC {
2     runExpState :: ExpInfo info => String -> info -> s -> (info, a, s)
3 }
4
5 instance Monad (ExpState s) where
6     — Placer x dans la monade sans altérer l'état ou l'info. d'exploration
7     return x = ExpStateC (\prefix info s -> (info, x, s))
8
9     — Altérer l'état (et l'info. d'exploration) via m puis via k a
10    m >>= k =
11        let f prefix info s =
12            let (info', a, s') = (runExpState m) prefix info s
13                in (runExpState (k a)) prefix info' s'
14        in ExpStateC f

```

Soit encore, en utilisant les mêmes notations que précédemment, on poserait cette fois **ExpState** comme $\Delta s.\Delta a.\Delta i_{\text{ExpInfo}}.\text{String} \rightarrow i \rightarrow s \rightarrow (i, a, s)$ et on aurait :

$$\begin{aligned} \text{return}[\text{ExpState}[s][a]] &= \lambda x_a. \\ &\quad \Lambda i_{\text{ExpInfo}}.\lambda p_{\text{String}}.\lambda z_i.\lambda t_s.(p, x, t) \\ \text{bind}[\text{State}[s][a]] &= \Lambda a''.\lambda m_{\text{ExtState}[s][a'']}. \lambda k_{a'' \rightarrow \text{ExtState}[s][a]}. \\ &\quad \Lambda i_{\text{ExpInfo}}. \\ &\quad \lambda p_{\text{String}}.\lambda z_i.\lambda t_s.(k(x)(p)(z')(t') \text{ où } (z', x, t') = m(p)(z)(t)) \end{aligned}$$

Dans cette dernière formulation, l'écriture i_{ExpInfo} traduit l'appartenance de la variable de type i à la classe de types **ExpInfo** définie un peu plus loin. Cela signifie que les fonctions contenues dans cette classe de types disposent d'une définition pour le type i . En d'autres termes, **eiCall**[i], **eiAlternatives**[i], **eiNested**[i] et **eiNested2**[i] sont toutes définies. La signification exacte de ces fonctions est discutée à partir du paragraphe suivant. (Le besoin d'exprimer cette contrainte souligne d'ailleurs l'utilité centrale du système de classes de types dans l'approche proposée. Dans toute la suite du manuscrit, on utilisera la syntaxe de Haskell plutôt que le lambda calcul typé polymorphe pour discuter notre approche. En effet, la syntaxe de ce langage est à la fois légère et adaptée pour discuter les problématiques étroitement liées aux classes de types.)

La fonction **runExpState**, qui est l'équivalent de la fonction **runState** de la monade d'état ordinaire, doit cette fois choisir parmi les alternatives exprimées par l'utilisateur pour aboutir à l'état définitif. Pour cela, elle devra suivre une politique dépendante de la façon dont l'étape d'exploration architecturale doit être effectuée. C'est pour cela que la classe de types **ExpInfo** est introduite. Elle est peuplée par des méthodes exprimant justement cette politique. Les objets appartenant à un type instanciant la classe **ExpInfo** contiendront, comme leur nom l'indique, l'information d'exploration architecturale nécessaire pour effectuer les choix survenant pendant l'exécution de la fonction **runExpState**. Nous verrons comment construire de tels types dans la section 2.3. Quoi qu'il en soit, un objet appartenant à un tel type est «contenu» dans la monade d'état explorable, tout comme l'état lui-même et la valeur retournée. Un *return* laisse l'information d'exploration architecturale inchangée (au même titre que l'état), tandis qu'un *bind* «propage» les modifications de cette information (au même titre qu'il propage les modifications de l'état). Comme on peut le voir, les similitudes entre le traitement de cette information et de l'état sont nombreuses. Cela dit, une distinction essentielle est le fait que la monade d'état explorable est paramétrée par le type de l'état, mais la fonction qu'elle encapsule est polymorphe vis à vis du type utilisé pour stocker l'information d'exploration. On le voit en particulier dans les définitions de **return**[**ExpState**[s][a]] et de **bind**[**ExpState**[s][a]] par la présence du terme $\Lambda i_{\text{ExpInfo}}$.

Pour l'instant, attardons nous sur l'expression de la classe **ExpInfo**, puisque c'est elle qui va conditionner quelles sont les primitives d'exploration architecturale disponibles. On pose :

```

1  class ExpInfo info where
2      eiCall  ::
3          (s -> (a, s))
4              -> String -> info -> s -> (info, a, s)
5      eiAlternatives  ::
6          String
7              -> [(Float, ExpState s a)]
8              -> String -> info -> s -> (info, a, s)
9      eiNested  ::
10         String
11         -> s1
12         -> ((a1, s1) -> ExpState s a)
13         -> ExpState s1 a1
14         -> String -> info -> s -> (info, a, s)
15     eiNested2  ::
16         String
17         -> s1 -> s2
18         -> ((a1, s1) -> (a2, s2) -> ExpState s a)
19         -> ExpState s1 a1 -> ExpState s2 a2
20         -> String -> info -> s -> (info, a, s)
21
22     _____
23     — Utilisable depuis un do: —
24     _____
25
26     call  :: (s -> (a, s)) -> ExpState s a
27     call f = ExpStateC (eiCall f)
28
29     modify  :: (s -> s) -> ExpState s ()
30     modify f = call (\s -> ((), f s))
31
32     inquire  :: (s -> a) -> ExpState s a
33     inquire f = call (\s -> (f s, s))

```

La première chose qui saute aux yeux est le type des fonctions `eiCall`, `eiAlternatives`, `eiNested` et similaires. En effet, leur signature se termine systématiquement par «`String -> info -> s -> (info, a, s)`», c'est à dire le type de `runExpState` (en tant que membre de `ExpState`). Ces fonctions sont en effet conçues pour être évaluées partiellement, les clôtures ainsi engendrées devenant candidates à l'encapsulation dans un objet de type `ExpState`. Les fonctions `call`, `modify` et `inquire` permettent justement cette encapsulation, et peuvent être utilisés directement dans un bloc «`do`» exprimant une monade de type `ExpState`. La fonction `call` effectue un appel modifiant l'état et plaçant une valeur dans la monade. La fonction `modify` altère l'état et place () dans la monade (c'est équivalent à «ne rien renvoyer»), tandis que `inquire`, au contraire, injecte dans la monade le résultat de l'évaluation de son argument sur l'état sans altérer ce dernier. Ces deux dernières fonctions constituent deux cas particuliers d'utilisation de `call`, et sont implémentées via l'utilisation de celle-ci.

Il y a en tout et pour tout trois types de primitives disponibles par le biais de la classe `ExpInfo` :

- `eiCall`, la primitive d'appel : correspond à une simple modification de l'état ou de la valeur retournée; elle ne doit pas, en principe, modifier l'information d'exploration
- `eiAlternatives`, l'expression d'implémentations alternatives : elle admet comme argument un identifiant et une liste pondérée de monades d'état explorables, et signifie que l'outil doit choisir l'une d'elles et une seule, idéalement en suivant la pondération fournie dans l'éventualité où l'exploration se ferait par le biais d'un algorithme stochastique
- `eiNested` et variantes, les primitives d'encapsulation : elles permettent d'encapsuler une ou plusieurs monades d'état explorables dans une autre, dont le type de l'état peut différer; il s'agit du mécanisme autorisant la cohabitation de plusieurs modèles de calculs différents au sein d'un même design

Les fonctions `eiNested` et similaires, pour pouvoir fonctionner, contiennent en pratique des appels à `runExpState`, au nombre de un par monade d'état explorable à imbriquer. Elles admettent comme arguments dans l'ordre :

- un identifiant, une sorte d'étiquette pour une encapsulation donnée

- les états initiaux pour chaque bloc monadique imbriqué, qui seront en pratique passées à `runExpState` lorsqu'elle sera appelée en interne
- une fonction qui, à partir des états finaux et valeurs retournées obtenues via `runExpState`, va générer un objet de type `ExpState s a`, c'est à dire du type de la monade dans laquelle on cherche à faire l'imbrication
- enfin, les constructions monadiques à imbriquer

Ici, on n'a défini que les fonctions `eiNested` et `eiNested2`, correspondant aux cas où l'on souhaite respectivement imbriquer un ou deux MoCs dans un autre. On pourrait se demander s'il n'est pas gênant de devoir définir une nouvelle fonction si on souhaite inclure un nombre de MoCs plus élevé en un seul appel. En pratique, on a très rarement besoin d'imbriquer plus de 2 blocs monadiques simultanément, et c'est la raison pour laquelle cet aspect ne nous paraît pas limitant.

Enfin, quelques mots sur les identifiants. Nous avons vu que les fonctions `eiAlternatives` et `eiNested` (et variantes) prenaient en argument une étiquette. De plus, la fonction `runExpState` prend elle aussi en argument une chaîne de caractères, alors que la fonction `runState` n'avait pas d'argument similaire. Ces identifiants ont un double rôle. D'une part, ils permettent à l'utilisateur de lire plus facilement les logs que les différentes méthodes d'exploration peuvent générer. D'autre part, ils permettent d'organiser hiérarchiquement les alternatives et imbrications. En fait, la chaîne fournie en guise d'argument à `runExpState` est un *préfixe*, qui va ensuite être utilisé pour construire une chaîne de caractères désignant le «chemin» d'une imbrication ou d'une expression d'alternatives dans le design. L'identifiant passé à `eiAlternatives` ou `eiNested` va être simplement concaténé à ce préfixe (à la suite d'un «`:`» ou «`/`» utilisé en guise de séparateur) pour les appels à `runExpState` contenus dans ces fonctions. Ainsi, si on a une imbrication appelée «`imb`» à l'intérieur d'une expression d'alternatives appelée «`alt`», le préfixe qui sera utilisé lors de l'appel à `runExpState` correspondant sera «`:alt:imb`». La hiérarchie ainsi obtenue pourra ensuite être utilisée par les méthodes d'exploration automatiques (voir par exemple section 2.3.2.3). D'une façon plus générale, la section 2.3 reviendra plus en détails sur l'usage de ces identifiants.

Avec tout ce que l'on a vu jusqu'ici, l'utilisateur peut utiliser le type de données `ExpState`, en conjonction avec de nouvelles instances de `ExpInfo`, pour implémenter diverses stratégies d'exploration architecturale. Cependant, nous n'avons pas encore défini de moyen d'exprimer les alternatives et imbrications dans les modèles de calcul eux-mêmes, qui sont de simples classes. Il convient pour cela de leur définir une classe parent abstrayant les primitives décrites jusqu'ici, le type `ExpState` instanciant cette classe. Nous appelons cette classe `Explorable`. On pose :

```

1 class Monad m => Explorable m where
2     namedAlternatives
3         :: String -> [(Float, m a)] -> m a
4     namedNest
5         :: String -> s1 -> ((a1, s1) -> m a) -> ExpState s1 a1 -> m a
6     namedNest2
7         :: String -> s1 -> s2 -> ((a1, s1) -> (a2, s2) -> m a)
8         -> ExpState s1 a1 -> ExpState s2 a2 -> m a
9
10 instance Explorable (ExpState s) where
11     namedAlternatives name alt = ExpStateC (eiAlternatives name alt)
12     namedNest name ini tr s = ExpStateC (eiNested name ini tr s)
13     namedNest2 name ini1 ini2 tr s1 s2
14         = ExpStateC (eiNested2 name ini1 ini2 tr s1 s2)
15
16 namedEquiprobable :: Explorable m => String -> [m a] -> m a
17 namedEquiprobable name alt =
18     let alt' = map (\x -> (1.0, x)) alt in namedAlternatives name alt'
19
20 namedOneOf name xs = namedEquiprobable name (map return xs)

```

La classe `Explorable` est celle dont devra dériver tout MoC. Elle offre au designer les fonctions `namedAlternatives`, `namedNest` et `namedNest2`, qui sont de simples équivalents des fonctions vues précédemment. `ExpState` est une instance de `Explorable`, et, sans surprises, se contente d'encapsuler les clôtures générées à partir des membres de `ExpInfo` correspondants. Pour des

raisons de commodité, on met à disposition du designer les fonctions `namedEquiprobable` et `namedOneOf`, lui permettant d'exprimer des alternatives équiprobables avec une syntaxe allégée, un peu à la façon des fonctions `call`, `modify` et `inquire` vues précédemment.

Un aspect qui pourrait sembler de prime abord étonnant est la présence du type `ExpState` dans la signature de `namedNest/namedNest2`. On pourrait se dire qu'il serait préférable d'utiliser n'importe quel type instanciant `Explorable` à la place. En fait, ceci n'est pas possible en pratique à cause des appels à `runExpState` cachés dans les fonctions d'imbrication. Un corollaire de ceci est qu'il n'est pas vraiment concevable de construire une autre instance d'`Explorable` qu'`ExpState`. Cela n'est cependant pas gênant, puisque le programmeur a toute la latitude nécessaire pour implémenter de nouvelles instances de `ExpInfo`. La généralité est simplement transférée aux instances d'`ExpInfo`, elle n'est pas perdue. Le fait de définir la classe `Explorable` reste une nécessité malgré tout puisque les modèles de calcul eux-mêmes sont des classes. Il n'y a donc pas d'autres moyen de s'assurer de leur explorabilité que de les dériver de `Explorable`.

Dans la section suivante, nous allons nous intéresser justement à la spécification de modèles de calcul. La spécification de méthodes d'exploration, qui implique la construction d'instances d'`ExpInfo`, sera traitée plus loin, dans la section 2.3.

2.2.2 Modèles de Calcul

Dans notre terminologie, un MoC est une classe Haskell dérivant de `Explorable`, un moteur est une instance d'un MoC, et un design est un objet polymorphe affectable à n'importe quel moteur d'un MoC donné, ou une fonction retournant un tel objet. En somme, si on a un MoC appelé MoC, un design est un objet ayant pour type MoC `em => em a`, ou une fonction renvoyant ce type d'objet. (Souvent, on prendra `()` pour `a`; ainsi le design aura pour type MoC `em => em ()`.)

Voici un exemple de deux modèles de calcul et d'un design associé :

```

1 data Obj1Id = ...
2 data Obj2Id = ...
3
4 class Explorable em => MoC1 em where
5     newObj1 :: String -> em Obj1Id  — "renvoie" Obj1Id
6     primitive1 :: Obj1Id -> em ()   — ne "renvoie" rien ("void")
7
8 class Explorable em => MoC2 em where
9     newObj2 :: String -> em Obj2Id  — "renvoie" Obj2Id
10    includeMoC1 :: Obj2Id -> (forall em'. MoC1 em' => em' ()) -> em () — void
11
12 design :: forall em. MoC2 em => em () — void
13 design = do
14     obj <- newObj2 "... "
15     obj 'includeMoC1' do — notation infixe; le do est le deuxième argument
16         obj' <- newObj1 "... "
17         primitive1 obj'
```

On a défini deux modèles de calcul, `MoC1` et `MoC2`, qui comportent chacun quelques primitives et quelques «constructeurs⁴». Les constructeurs renvoient souvent, comme dans cet exemple, des types concrets. On verra au chapitre 3 des constructeurs au type de retour polymorphe et l'intérêt qu'ils peuvent avoir dans certains cas bien précis.

La primitive `includeMoC1` permet d'encapsuler `MoC1` dans `MoC2`. (On peut vouloir encapsuler des MoCs pour une multitude de raisons : créer des sous-composants, des processus, spécifier une partie critique d'un système à un niveau d'abstraction plus faible, etc.) On peut voir que le deuxième argument de `includeMoC1`, correspondant à une partie du design exprimée dans le MoC 1, est bien polymorphe comme tout design doit l'être. Son type est identique à celui de l'objet

4. par analogie avec le sens que ce mot revêt dans le contexte de la programmation par objets, nous appelons constructeurs des fonctions prévues pour être utilisées avec l'opérateur monadique `<-`, créant un nouvel identifiant dans un «do»; à ne pas confondre avec les constructeurs de données ou de types, au sens Haskell du terme

`design` (ligne 12), dont la signature est explicite⁵. En pratique, l'implémentation de la fonction `includeMoC1` va utiliser la fonction `namedNest` vue dans la section précédente. Il en résulte que le sous-design passé en argument va forcément être affecté à un moteur de MoC1. Garder cet argument polymorphe est cependant préférable, puisque les différents moteurs de MoC2 peuvent vouloir passer par différents moteurs de MoC1 dans leur implémentation de `includeMoC1`.

Le design lui-même est exprimé en utilisant la notation «do», devrait pouvoir être compris assez naturellement par tout habitué des paradigmes impératifs ou déclaratifs des langages de description de matériel classiques.

D'ordinaire, un modèle de calcul est spécifié dans un module (c.-à-d. unité de compilation) séparé de celui du design, pour des raisons purement pratiques. Cela permet de le réutiliser facilement entre plusieurs designs, d'ajouter dans l'espace de noms du module du MoC diverses fonctions bibliothèques liés à son emploi, etc. Cela dit, les MoCs suffisamment simples ou dont la réutilisation présente un intérêt réduit peuvent parfaitement être définis au même endroit que le design.

2.2.3 Moteurs

Comme les instances des MoCs doivent être explorables, il s'agit en pratique de types obtenus en passant *un seul argument* au constructeur de type `ExpState`. La définition d'un moteur passe donc avant tout par la définition de l'état qui sera ensuite utilisé dans la monade d'état explorable correspondante.

Pour permettre l'exploration architecturale automatique, des fonctions de coût ou des aides pour en définir accompagnent la plupart des moteurs. De plus, une fois l'étape d'exploration effectuée, l'utilisateur doit pouvoir faire quelque chose d'utile avec l'état final obtenu. Pour cela, un moteur doit être accompagné de fonctions permettant de générer des logs, du code, etc., à partir cet état final.

Voici un exemple illustrant le genre de syntaxe qu'implique la définition d'un moteur pour le design défini plus haut, et son utilisation du point de vue de l'utilisateur :

```

1 data MoC2State = ... — état interne utilisé par le moteur
2
3 instance MoC2 (ExpState MoC2State) where
4     newObj2 params = ...
5     includeMoC1 obj2 moc1 = ...
6
7 moc2Empty :: MoC2State — design vide (utilisé comme état initial)
8 moc2Empty = ...
9
10 moc2Score :: (a, MoC2State) -> Int — évalue la qualité d'un design
11 moc2Score (_, moc2state) = ...
12
13 generateUsefulFiles :: MoC2State -> IO () — écrit le résultat sur le disque
14 generateUsefulFiles moc2state = ...
15
16 main = do
17     let iterations = simulateAnnealing (\_ -> 1.0) moc2Score design moc2Empty
18         let engineState = pickBest (take 500 iterations)
19             generateUsefulFiles engineState

```

Le détail des types de données utilisés ainsi que les corps de toutes les fonctions à l'exception de `main` est pour l'instant passé sous silence. On peut néanmoins constater la présence d'une fonction de coût à maximiser, et d'une fonction permettant de générer des fichiers utiles à partir de l'état final obtenu (par exemple, du code intermédiaire, des logs, etc.). L'utilisateur n'a plus qu'à écrire une fonction principale dans laquelle il va invoquer un algorithme d'exploration (ici, un recuit simulé – expliqué en détail plus loin) sur son design avant de produire les fichiers en question.

5. les signatures des objets polymorphes sont obligatoirement explicites en Haskell

2.2.4 Règles d'implémentation

De sorte à faciliter au maximum l'implémentation, un certain nombre de principes est à suivre lors de la définition des MoCs et des moteurs associés :

- La classe de types correspondant à un MoC doit comporter aussi peu de membres obligatoires que possible. Le moyen le plus élégant et le plus pratique pour enrichir un MoC avec de nouvelles constructions est de les définir en utilisant les primitives qui y existent déjà. Ainsi, l'implémentation des moteurs reste simple. (Il est possible, également, de spécifier des implémentations par défaut pour certaines des primitives d'un MoC. De cette façon, un moteur est libre de proposer une implémentation plus efficace d'une primitive donnée. Néanmoins, ce manuscrit ne s'appesantira pas sur ce genre de considérations, au profit d'une simplicité accrue des exemples fournis.)
- Un moteur, comme on l'a vu, a au contraire tout intérêt à proposer autant de fonctions associés que possible pour maximiser son utilité et sa versatilité.

2.3 Expression des méthodes d'exploration

On distingue deux types de méthodes d'exploration : celles que l'on appelle simples et qui génèrent de manière *ad hoc* les choix aux alternatives présentes dans le design, et celles dites avancées qui reposent sur l'extraction de l'ensemble des alternatives que le design présente dans une configuration donnée, et qui améliorent itérativement les choix effectués sur chacune d'entre elles.

2.3.1 Méthodes simples

2.3.1.1 Choix par défaut

Lorsque le design ne comporte pas d'alternatives, ou que l'utilisateur cherche juste à en obtenir une première version, il peut être intéressant de disposer d'une méthode pour transformer une monade d'état explorable en monade d'état simple. On peut y parvenir en définissant la méthode d'exploration suivante :

```

1 data ExpDefault = ExpDefault
2
3 instance ExpInfo ExpDefault where
4   eiCall f prefix info s = let (a, s') = f s in (info, a, s')
5   eiAlternatives name (x:xs) prefix info s =
6     let
7       nm = prefix++":"++name
8       cmp x@(w1, _) y@(w2, _) = if (w1 > w2) then x else y
9       (_, f) = foldr cmp x xs
10      in (runExpState f) nm info s
11   eiNested name ini tr s' prefix info s =
12     let
13       nm sep = prefix++sep++name
14       (info', a1, s1) = (runExpState s') (nm ":") info ini
15       (info'', a'', s'') =
16         runExpState (tr (a1, s1)) (nm "/" ) info' s
17       in (info'', a'', s'')
18   eiNested2 name ini1 ini2 tr s1 s2 prefix info s =
19     let
20       nm clause = prefix++":"++name++" ("++clause++)"
21       nm' = prefix++"/"++name
22       (info', a1', s1') = (runExpState s1) (nm "A") info ini1
23       (info'', a2', s2') = (runExpState s2) (nm "B") info' ini2
24       (info''', a', s') =
25         runExpState (tr (a1', s1')(a2', s2')) nm' info'' s
26       in (info''', a', s')
27
28 pickDefault :: ExpState s a -> State s a

```

```

29 pickDefault m =
30     let aux g = \s -> let (info, a, s') = g ExpDefault s in (a, s')
31     in StateC (aux (runExpState m ""))

```

Le membre `eiAlternatives` se contente de chercher l'alternative à la probabilité la plus élevée et l'évalue. En cas d'égalité, c'est l'alternative donnée en premier qui est évaluée. Les fonctions `eiNested` et `eiNested2` peuvent paraître excessivement compliquées, sachant que le type `ExpDefault`, comme on le voit, ne contient aucune information. En réalité, ces fonctions seront écrites de la même façon pour quasiment toutes les méthodes d'exploration. On y évalue tour à tour chacune des monades à imbriquer, en enrichissant à chaque fois l'information d'exploration des nouveaux renseignements qui peuvent y être récoltés, puis on évalue le résultat de la fonction `tr` sur l'état courant et l'information d'exploration précédemment complétée. On aurait pu définir ceci comme l'implémentation par défaut des fonctions `eiNested` et variantes ; ceci dit, cela aurait pu donner l'impression au programmeur que l'information d'exploration doit *forcément* être recueillie de cette façon là, alors que ce n'est pas du tout le cas. Toute implémentation de `ExpInfo` est libre de construire la partie `info` du triplet `(info, a, s)` de la façon qui lui convient le mieux (idéalement, en respectant malgré tout les conventions de nommage et de séparateurs pour les préfixes).

Ultimement, on définit `pickDefault` permettant de démonter une monade d'état explorable au rang de monade d'état ordinaire, via l'utilisation de `ExpDefault` en guise d'argument de `runExpState`.

2.3.1.2 Parcours aléatoire

Une autre méthode d'exploration simple est celle du parcours aléatoire de l'espace de design. Comme le nom l'indique, chaque alternative est choisie aléatoirement avec une probabilité proportionnelle à son poids, et indépendamment de la valeur qu'elle pouvait avoir lors de l'itération précédente. L'expression de cette méthode peut se faire ainsi :

```

1  pickAlternative :: Float -> [(Float, a)] -> a -- arg. dans [0, somme des poids[
2  pickAlternative p ((p', x):[]) = x
3  pickAlternative p ((p', x):xs) | (p - p') < 0.0 = x
4  pickAlternative p ((p', x):xs) = pickAlternative (p - p') xs
5
6  data ExpRandom = ExpRandom [Float]
7
8  instance ExpInfo ExpRandom where
9      eiCall f prefix info s = let (a, s') = f s in (info, a, s')
10     eiAlternatives name (x:xs) prefix (ExpRandom (r:rs)) s =
11         let
12             nm = prefix++":"++name
13             sum = foldr (+) (fst x) (map fst xs)
14             info = ExpRandom rs
15             r' = r * sum
16         in (runExpState (pickAlternative r' (x:xs))) nm info s
17     eiNested name ini tr s' prefix info s = ...
18     eiNested2 name ini1 ini2 tr s1 s2 prefix info s = ...
19
20 randomWalk' :: [Float] -> ExpState s a -> s -> [(a, s)]
21 randomWalk  :: ExpState s a -> s -> [(a, s)]
22
23 randomWalk' rs es ini =
24     let (ExpRandom rs', a', s') = runExpState es "" (ExpRandom rs) ini
25     in (a', s'):(randomWalk' rs' es ini)
26 randomWalk =
27     let ExpRandom rs = ExpRandom ((randoms (mkStdGen 42)) :: [Float])
28     in randomWalk' rs

```

Cette fois, `ExpRandom`, le type instance de `ExpInfo`, n'est pas vide, mais comporte une liste infinie de flottants pseudo-aléatoires compris entre 0 et 1. On définit la fonction `pickAlternative` qui choisit une alternative parmi une liste à partir d'un nombre flottant compris entre 0 et la

somme des poids des éléments de la liste. La fonction `eiAlternatives` se contente de prendre un nombre aléatoire contenu dans l'information d'exploration `ExpRandom`, le multiplier par la somme des poids de la liste d'alternatives, et appeler `pickAlternative` pour sélectionner l'alternative à évaluer. Les fonctions d'imbrication `eiNested` et `eiNested2` ont exactement le même corps que dans `ExpDefault` (c.f. section 2.3.1.1), que nous n'avons pas redonné ici pour des raisons de concision.

Les fonctions `randomWalk` et `randomWalk'` sont celles que l'utilisateur est destiné à utiliser. Elles prennent en argument le design et l'état initial du moteur, et retournent une liste de paires valeur renvoyée / état final. `randomWalk'` prend également en argument la liste de flottants qui sera placé dans l'objet de type `ExpRandom`, tandis que `randomWalk` génère cette liste à partir d'une graine fixée à 42, via la fonction `mkStdGen` du module `System.Random`, fourni avec les compilateurs Haskell tels que Hugs ou GHC.

2.3.2 Méthodes avancées

2.3.2.1 Types de données

Les méthodes précédentes ont pour elles leur simplicité, mais sont assez limitées du point de vue de leur efficacité. Il n'est pas forcément non plus aisé, pour l'utilisateur, de récupérer la liste des choix effectués au moment de l'exploration architecturale. On se propose de construire une instance de `ExpInfo` réutilisable entre divers algorithmes d'exploration itératifs, et garantissant de surcroît à l'utilisateur une plus grande observabilité du processus d'exploration.

On pourrait songer à plusieurs façons d'organiser ce genre de type de données. Une des façons possibles de faire est d'utiliser un arbre, dont les noeuds correspondraient aux alternatives et imbrications. On stockerait alors le chemin emprunté aux divers embranchements. À chaque itération du processus d'exploration, il suffirait alors de modifier ce chemin. (Il est à noter que c'est précisément dans ce genre de cas que l'allure des fonctions `eiNested` et variantes changerait notablement.)

Nous avons préféré opter pour une organisation à plat qui s'avère remarquablement efficace à l'utilisation. Elle est basée sur le stockage des choix dans une table de hachage indexée par leur nom. Si plusieurs alternatives ou imbrications portent la même étiquette, un système de *name mangling* est là pour leur trouver des étiquettes uniques, proches de celle demandée à l'origine. (Notons que les collisions d'étiquettes peuvent arriver assez fréquemment à qui n'y prend garde; c'est même un argument en faveur de techniques telles que la quasiquote. Nous n'insisterons pas plus avant sur cet aspect pour l'instant.)

```

1 data ExpChoice = ExpChoice {
2     expChoiceName :: String,
3     expChoiceWeights :: [Float], — poids des alternatives
4     expChoiceSum :: Float, — somme des poids
5     expChoiceCurrent :: Float — alternative courrante (dans [0,somme des poids])
6 }
7
8 data ExpChoices = ExpChoices {
9     expChoicesChoices :: Map String ExpChoice, — choix en eux-même
10    expChoicesActive :: [String] — clés des choix actifs
11 }
12
13 mangle :: String -> Map String ExpChoice -> String
14 mangle name choices = case Map.lookup name choices of
15     Nothing -> name
16     otherwise -> mangle (name ++ "'") choices
17
18 expChoicesEmpty :: ExpChoices
19 expChoicesEmpty = ExpChoices (Map.empty) []
20
21 expChoicesLookup :: ExpChoices -> String -> Maybe ExpChoice
22 expChoicesLookup choices name = Map.lookup name (expChoicesChoices choices)
23
24 instance ExpInfo ExpChoices where

```

```

25     eiCall f prefix info s = let (a, s') = f s in (info, a, s')
26     eiAlternatives name alts@(x:xs) prefix info s =
27         let
28             nm = prefix ++ ":" ++ name
29             choice = expChoicesLookup info nm
30             f1 p = pickAlternative p alts
31             choiceIfMissing = ExpChoice
32                 { expChoiceName = nm
33                   , expChoiceWeights = map fst alts
34                   , expChoiceSum = sum (map fst alts)
35                   , expChoiceCurrent = 0.0
36                 }
37             g choice = case choice of
38                 Just (ExpChoice _ _ p) -> (f1 p, info)
39                 Nothing -> (snd x, completed info)
40             mangledNm = mangle nm (expChoicesChoices info)
41             completed info = info
42                 { expChoicesChoices = Map.insert
43                   nm choiceIfMissing
44                   (expChoicesChoices info) }
45             activated info = info
46                 { expChoicesActive = nm:(expChoicesActive info)}
47         in let (f, info') = g choice
48             in (runExpState f) nm (activated info') s
49     eiNested name ini tr s' prefix info s = ...
50     eiNested2 name ini1 ini2 tr s1 s2 prefix info s = ...
51
52 pickBest :: Ord score => [(score, ExpChoices, a, s)] -> s
53 pickBest ((score, -, -, s):xs) =
54     let
55         aux (score, s) [] = s
56         aux (score, s) ((score', -, -, s'):xs) | score < score'
57             = aux (score', s') xs
58         aux (score, s) ((score', -, -, s'):xs)
59             = aux (score, s) xs
60     in aux (score, s) xs
    
```

D'abord, quelques mots sur les types de données eux mêmes. Chaque *choix* (type `ExpChoice`) correspond à une expression d'alternative, et est caractérisé par :

- son étiquette courte (sans les autres préfixes concaténés devant)
- les poids des alternatives, notés $(w_i)_{i \in [1, n]}$
- la somme de ces poids – c'est en effet l'information dont on va se servir le plus souvent
- l'information sur le choix courant c (membre `expChoicesCurrent`), sous la forme d'un nombre flottant compris entre 0 et la somme des poids ; on utilisera l'alternative k si et seulement si $c \in [\sum_{i=1}^{k-1} w_i, \sum_{i=1}^k w_i]$ (on a déjà vu ce comportement avec la fonction `pickAlternative` de la section 2.3.1.2)

L'ensemble des choix recueillis au terme du parcours du design est consigné dans un objet du type `ExpChoices`, qui est une instance de `ExpInfo`. Il contient une table de hachage des choix indexés par leur étiquette complète (avec tous les préfixes), ainsi que la liste des étiquettes actives, c'est à dire celle des alternatives réellement évaluées lors du parcours du design. En effet, la table de hachage sera conservée et progressivement enrichie d'itération en itération au cours du processus d'exploration, et pourra donc contenir des choix non évalués à une itération donnée (se trouvant dans une alternative non sélectionnée). Il est important de ne pas perdre cette information, puisqu'il y a des chances qu'elle serve à nouveau, mais il est tout aussi important de savoir que modifier les choix qui concernent les parties inactives n'aura aucun impact dans l'immédiat. Les algorithmes d'exploration se contenteront donc de «muter» les choix actifs.

L'implémentation de `aiAlternatives` traduit ce comportement de préservation de l'enrichissement de la table de hachage. Si un choix n'existe pas dans la table, il est créé, avec $c = 0$ par défaut. S'il existe, il est préservé tel quel. Dans tous les cas, l'étiquette complète de l'ensemble d'alternatives courant est ajoutée à la liste des étiquettes des ensembles actifs, qui est censée avoir été remise à `[]` avant l'appel à `runExpState`. En toute logique, l'alternative évaluée correspond

au `c` du choix présent ou introduit dans la table de hachage. Les implémentations de `eiCall`, `eiNested` et `eiNested2` pour `ExpChoices` sont les mêmes que celles que nous avons vues jusqu'ici.

On met également à disposition de l'utilisateur la fonction `pickBest`, qui à partir d'une liste de quadruplets `score/choix/valeur renvoyée/état final` va renvoyer l'état final au score le plus élevé. Nous verrons par la suite que les algorithmes d'exploration que nous proposons sont formatés pour produire de telles listes.

2.3.2.2 Recuit simulé

Un premier exemple de méthode d'exploration implémentée par dessus `ExpChoices` est le recuit simulé. C'est un algorithme qui présente l'avantage d'être simple, plutôt robuste, et configurable. Le principe est d'optimiser progressivement le design en vue de maximiser une fonction de coût, en faisant varier le niveau d'altération des choix d'une itération à la suivante suivant un paramètre appelé température. Pour une température élevée, la quasi-totalité des choix sera bouleversée à chaque étape, tandis qu'une température faible induira seulement des changements minimes. L'idée est de faire décroître la température au cours du processus pour peu à peu se rapprocher d'un maximum local (si possible, proche du maximum global).

Est-ce que ce genre d'algorithme a sa place pour l'optimisation architecturale de circuits électroniques? La réponse est sujette à controverse, mais on peut néanmoins pencher pour un oui prudent. Il est clair qu'un grand nombre de paramètres d'un design manifestent des comportements apparemment chaotiques peu propices à l'apparition de grandes «zones» dans l'espace de design où la fonction de coût serait globalement faible ou élevée – condition *sine qua non* pour que le recuit simulé ait de l'intérêt par rapport à un parcours complètement aléatoire. Cependant, d'autres paramètres sont plus prévisibles, du moins sous certaines conditions. (Nous verrons des exemples piochés dans les deux catégories dans les chapitres suivants.) Le recuit simulé est un bon exemple de méthode tirant parti de la régularité, si faible soit-elle, d'un jeu de paramètres pour trouver une solution acceptable. Il marche de façon acceptable dans une grande variété de situations.

Il nous faut dans un premier temps définir un moyen de «muter» un ensemble de choix en fonction de la température. On peut pour cela définir la fonction `mutate` ainsi :

```

1 mutate :: Float -> [Float] -> ExpChoices -> ([Float], ExpChoices)
2 mutate = mutate' corrBrutal
3
4 mutate' ::
5   (Float -> Float -> Float -> Float -> Float -> Float)
6   -> Float -> [Float] -> ExpChoices -> ([Float], ExpChoices)
7 mutate' corr intensity randoms choices =
8   let
9     aux xs [] choices = (xs, choices)
10    aux (x:x':xs) (altNm:altNms) choices =
11      let
12        adj (ExpChoice name w sum c) =
13          ExpChoice name w sum
14            (corr intensity c sum x x')
15        choices' = Map.adjust adj altNm choices
16      in aux xs altNms choices'
17    active = expChoicesActive choices
18    (rd', ch') = aux randoms active (expChoicesChoices choices)
19  in (rd', choices { expChoicesChoices = ch', expChoicesActive = [] })
20
21 corrBrutal intensity c sum x x' | x < intensity = x' * sum
22 corrBrutal _ c _ _ = c
23
24 corrSoft intensity p sum x x' | intensity >= 0.5 = x' * sum
25 corrSoft intensity p sum x x' | p - sum*intensity <= 0.0
26   = x' * sum * 2.0 * intensity
27 corrSoft intensity p sum x x' | p + sum*intensity >= sum
28   = sum - x' * sum * 2.0 * intensity
29 corrSoft intensity p sum x x' = p + (2.0 * x' - 1.0) * intensity * sum

```

`mutate` prend en argument respectivement la température, une liste de nombres pseudo-aléatoires, et les choix à muter ; elle renvoie une paire correspondant au reste des nombres pseudo-aléatoires (ceux qui n'ont pas été utilisés), ainsi que les nouveaux choix. En fait, `mutate` est une application partielle de `mutate'`, qui, elle, prend de plus en argument une fonction de correction. Cette fonction de correction accepte 5 flottants en argument (la température, le c à corriger, la somme des poids de l'ensemble d'alternatives auquel appartient ce c , et deux flottants aléatoires). Elle renvoie une nouvelle valeur de c , «corrigée». L'implémentation de `mutate` rend clair que chaque ensemble d'alternatives actif est considéré indépendamment, et que c'est le rôle de la fonction de correction de faire le travail de mutation effectif, en fonction de la température. Ici, deux fonctions de correction ont été proposées :

- `corrBrutal`, une fonction tout ou rien, qui soit laisse c inchangé, soit en prend une nouvelle valeur de façon complètement aléatoire ; la température correspond à la probabilité que c change de valeur (ainsi, si elle est supérieure à 1.0, tous les choix seront remplacés)
- `corrSoft`, une fonction qui change c systématiquement, mais le fait varier sur un intervalle centré sur sa valeur précédente dont l'étendue est proportionnelle à la température ; notez que cela ne change pas forcément l'alternative qui sera choisie – il faut pour cela que c traverse une «frontière» (un des nombres de l'ensemble suivant : $(\sum_{i=1}^k w_i)_{k \in [1, n-1]}$)

On observe que la correction brutale marche beaucoup mieux en pratique que la douce, sur les designs que nous avons pu tester. Dans l'absolu, on pourrait laisser l'utilisateur choisir celle qu'il préfère, en exigeant de lui un paramètre supplémentaire au lancement de l'étape d'exploration. Sans raison particulière, nous nous sommes abstenus de le faire.

Une fois `mutate` définie, le reste de l'implémentation du recuit simulé devient une simple formalité :

```

1 simulateAnnealing' ::
2   Ord score => [Float] -> (Integer -> Float)
3   -> ((a, s) -> score) -> ExpState s a -> s -> [(score, ExpChoices, a, s)]
4 simulateAnnealing ::
5   Ord score => (Integer -> Float)
6   -> ((a, s) -> score) -> ExpState s a -> s -> [(score, ExpChoices, a, s)]
7
8 simulateAnnealing' randoms f score es ini =
9   let
10      aux best randoms (intensity:intensities) choices =
11        let
12          (randoms', choices')
13            = mutate intensity randoms choices
14          (choices'', a, s)
15            = runExpState es "" choices' ini
16          val = score (a, s)
17          (best', choices''') = case best of
18            Just b | b > val -> (best, choices)
19            otherwise -> (Just val, choices'')
20          rest = aux best' randoms' intensities choices'''
21        in (val, choices'', a, s):rest
22   in aux Nothing randoms (map f [0..]) expChoicesEmpty
23
24 simulateAnnealing f score es ini
25   = simulateAnnealing' (randoms (mkStdGen 42)) f score es ini

```

À chaque itération, on évalue un voisin de la position courante et un seul. Si ce voisin obtient un meilleur score via la fonction de coût que la position courante, ce voisin devient la nouvelle position courante. L'argument que l'utilisateur fournit à `simulateAnnealing` est une fonction qui à un numéro d'itération fait correspondre une température.

Notez qu'à chaque itération, on produit un quadruplet score/choix/valeur/état, comme on l'a expliqué précédemment. Ce quadruplet correspond à l'évaluation du voisin. Il est concaténé à la liste générée (qui est donc infinie au final), que son score soit meilleur que celui de la position actuelle ou pas. Il est important de procéder de la sorte, puisque l'utilisateur va ensuite utiliser une fonction de type `pickBest.(take nIterations)` pour évaluer le meilleur résultat sur un certain

nombre d'itérations. S'il dispose d'une borne supérieure de la complexité de l'évaluation du score sur une itération, il peut en déduire une de l'évaluation de son `pickBest.(take nIterations)`. Si on avait choisi, par exemple, de ne lister que les quadruplets pour lesquels le score est strictement supérieur à celui de la position actuelle, l'utilisateur n'aurait aucune maîtrise quant au nombre d'itérations de recuit simulé séparant deux maillons consécutifs de la liste générée.

2.3.2.3 Programmation génétique

Un autre exemple de famille d'algorithmes stochastiques utilisables à des fins d'optimisation est celle des algorithmes génétiques. Le principe est de faire évoluer une population de designs («individus») afin de maximiser leurs scores par le biais d'un processus inspiré de la sélection naturelle.

Il existe une grande variété de façons possible d'implémenter ce genre d'algorithme. Il faut essentiellement définir comment se déroulent trois processus :

1. La mutation : afin d'augmenter la diversité génétique de la population, les individus doivent pouvoir muter «tout seuls»; cette mutation peut être plus ou moins importante
2. Le croisement : deux individus doivent pouvoir se croiser pour produire un troisième héritant ses caractéristiques de ses deux parents
3. La sélection : comme la population est en générale limitée, il faut définir quels individus sont croisés et quels autres sont effacés pour laisser place au résultat des croisements effectués

Nous avons déjà vu la fonction de mutation `mutate` à la section 2.3.2.2. Nous allons la réutiliser. Cette fonction s'attend à recevoir en paramètre une température, indiquant à quel degré l'individu va être muté. Nous laisserons le choix de cette température à l'utilisateur.

Il nous faut définir une fonction de croisement. Lorsqu'on s'attaque à l'optimisation de programmes et assimilables (c.-à-d. d'expressions ayant un arbre pour représentation intermédiaire), on pense spontanément à la programmation génétique, sous-classe d'algorithmes génétiques pour lesquels une opération de croisement passe par l'échange de nœuds entre les arbres des parents. Nous aurions pu utiliser cette technique si nous avions choisi de préserver la hiérarchie des alternatives/imbrications dans notre objet `ExpChoice`⁶. Nous utilisons au lieu de cela une procédure heuristique basée sur la distance de Levenshtein [55] entre les étiquettes des ensembles d'alternatives/imbrications :

```

1 levenshteinDistance :: String -> String -> Int
2 levenshteinDistance s t =
3   let
4     m = length s
5     n = length t
6     makeElem cS cT nx ny nxy | cS == cT = nxy
7     makeElem cS cT nx ny nxy = (minimum [nx, ny, nxy]) + 1
8     makeNextLine prevLine =
9       let
10        nLine = head prevLine + 1
11        cS = s !! (head prevLine)
12        aux nx (nxy:[]) [] = []
13        aux nx (nxy:ny:ns) (cT:cTs) =
14          let x = makeElem cS cT nx ny nxy
15            in x:(aux x (ny:ns) cTs)
16      in nLine:(aux nLine prevLine t)
17   in last (head (drop m (iterate makeNextLine [0..n])))
18
19 crossover :: [Float] -> ExpChoices -> ExpChoices -> ([Float], ExpChoices)
20 crossover (r1:r2:r3:randoms) choices1 choices2 =
21   let
22     ch1 = expChoicesChoices choices1
23     ch2 = expChoicesChoices choices2

```

6. En fait, on pourrait tout à fait reconstruire cette hiérarchie à partir de la table de hachage des alternatives, et ce à un coût raisonnable; on choisit donc sciemment de procéder différemment

```

24     active1 = Set.fromList (expChoicesActive choices1)
25     active2 = Set.fromList (expChoicesActive choices2)
26     size1 = Map.size ch1
27     size2 = Map.size ch2
28     c1 = floor (r1 * fromIntegral size1)
29     c2 = floor (r2 * fromIntegral size2)
30     (str1, _) = Map.elemAt c1 ch1
31     (str2, _) = Map.elemAt c2 ch2
32     dist = levenshteinDistance
33     merge :: String -> ExpChoice -> ExpChoice -> ExpChoice
34     merge nm choice1 choice2
35         | Set.member nm active1 && Set.notMember nm active2
36         = choice1
37     merge nm choice1 choice2
38         | Set.member nm active2 && Set.notMember nm active1
39         = choice2
40     merge nm choice1 choice2 | dist nm str1 < dist nm str2 = choice1
41     merge nm choice1 choice2 | dist nm str1 > dist nm str2 = choice2
42     merge nm choice1 choice2 | r3 < 0.5 = choice1
43     merge nm choice1 choice2 = choice2
44     in (randoms, ExpChoices (Map.unionWithKey merge ch1 ch2) [])

```

La distance de Levenshtein entre deux chaînes de caractères est le nombre d'insertions, suppressions ou permutations de caractères qu'il faut effectuer pour passer d'une chaîne à l'autre. Comme on peut le voir dans le code, cette opération se fait en $O(l_1 l_2)$ où l_1 et l_2 sont les longueurs des opérandes, et se prête très bien à une implémentation fonctionnelle.

Le processus de croisement lui-même sélectionne au hasard un identifiant chez chaque parent (qu'il fasse partie des ensembles de choix actifs ou pas). On appelle cet identifiant l'identifiant de référence. Pour effectuer le croisement, on fait l'union des tables de hachage des deux parents en respectant les règles suivantes :

- si un ensemble de choix n'est «connu» que d'un seul parent, l'information correspondante chez ce parent est transférée à l'enfant
- si un ensemble de choix est «connu» des deux parents mais n'est actif que chez l'un d'entre eux, c'est le parent actif qui transmet l'information correspondante à l'enfant
- dans tous les autres cas, c'est le parent dont l'identifiant de référence est le plus proche, au sens de la distance de Levenshtein, de celui de l'ensemble de choix considéré qui transmet l'information associée; en cas de distance égale, le parent transmetteur est choisi au hasard

Cette façon de procéder fait que, globalement, un parent a tendance à transmettre des choix qui correspondraient à un sous-arbre, si on avait procédé hiérarchiquement. De plus, les choix aux identifiants proches mais sur des noeuds situés potentiellement à des endroits très différents sont groupés. Sachant cela, le designer peut s'en servir pour accélérer la vitesse de convergence en attribuant des identifiants similaires à des ensembles d'alternatives qu'il pense corrélés. Il s'agit d'une heuristique assez intéressante, d'autant plus qu'on peut penser qu'un programmeur aura tendance à suivre cette règle spontanément, sans même réfléchir aux (ou être au courant des) conséquences que cela va avoir sur le processus de croisement.

En ce qui concerne le processus de sélection, on effectue au plus un croisement par itération (la probabilité d'effectuer un croisement lors de n'importe quelle itération est une constante fournie par l'utilisateur). À chaque itération, tous les individus non issus de croisement sont mutés. Pour réaliser le croisement, s'il a lieu, on procède la manière suivante : parmi notre population de n individus, on en choisit deux parmi les $n - 1$ les mieux classés, pour remplacer celui dont le score est le moins bon. La fonction utilisée pour réaliser ce choix est la suivante :

```

1 pickTwo (p, p') r1 r2 [x1, x2] = (x1, x2)
2 pickTwo (p, p') r1 r2 (x1:x2:xs) | (r1 < p) && (r2 < p) = (x1, x2)
3 pickTwo (p, p') r1 r2 (x1:x2:xs) | (r1 < p) = pickTwo (p, p') r1 (r2/p') (x1:xs)
4 pickTwo (p, p') r1 r2 (x1:x2:xs) = pickTwo (p, p') (r1/p') r2 (x2:xs)

```

Les paramètres p et p' que l'utilisateur fournit définissent la probabilité de sélectionner chaque paire d'individus. Plus il sont proches de 1, plus la probabilité de sélectionner les individus les

2.3. EXPRESSION DES MÉTHODES D'EXPLORATION

	x_1	x_2	x_3	x_4	x_5	x_6
x_1	×	×	×	×	×	×
x_2	49.0 %	×	×	×	×	×
x_3	12.3 %	12.3 %	×	×	×	×
x_4	8.63 %	3.12 %	8.63 %	×	×	×
x_5	0 %	2.17 %	3.69 %	0 %	×	×
x_6	0 %	0 %	0 %	0 %	0 %	×

FIGURE 2.1 – Exemple de probabilités de sélectionner une paire d'individus via la fonction `pickTwo` (0.7, 0.8), parmi une population de 6 individus appelés $\{x_1, \dots, x_6\}$ rangés du meilleur au plus mauvais

mieux classés est élevée. La figure 2.1 illustre un exemple d'ensemble de probabilités pour une paire (p, p') donnée.

Le reste de la procédure de sélection est assez verbeux mais ne présente pas de difficultés :

```

1  breed' ::
2      Ord score => [Float] -> Float -> [Float]
3      -> ((a, s) -> score) -> ExpState s a -> s -> a
4      -> [(score, ExpChoices, a, s)]
5
6  breed' randoms pCrossover temps score es ini a =
7      let
8          n :: Num a => a
9          n = fromIntegral (length temps)
10         cmp (score1, -, -, -) (score2, -, -, -) = compare score1 score2
11         crs randoms
12             papa@(pScore, pChoices, pA, pS)
13             mama@(mScore, mChoices, mA, mS) =
14             let
15                 (randoms', choices')
16                     = crossover randoms pChoices mChoices
17                 (choices'', a, s)
18                     = runExpState es "" choices' ini
19                 val = score (a, s)
20                 evald = (val, choices'', a, s)
21             in (randoms', evald, evald)
22         mut temp randoms bestSoFar@(scorePrec, choices, -, -) =
23         let
24             (randoms', choices')
25                 = mutate temp randoms choices
26             (choices'', a, s)
27                 = runExpState es "" choices' ini
28             val = score (a, s)
29             evald = (val, choices'', a, s)
30             newBest = if (scorePrec > val)
31                 then bestSoFar
32                 else evald
33         in (randoms', newBest, evald)
34     chMut randoms [] [] = (randoms, [], [])
35     chMut randoms (temp:temps) (quad:quads) =
36     let
37         (randoms', quad', e) = mut temp randoms quad
38         (randoms'', rest, es) = chMut randoms' temps quads
39     in (randoms'', quad':rest, e:es)
40     aux individuals randoms =
41     let
42         r1:r2:r3:randoms' = randoms
43         doCrossover = r1 < pCrossover
44         crossoverPos = floor (r2 * n)
45         (worst:rest) = List.sortBy cmp individuals
46         (papa, mama) = pickTwo (0.65, 0.5) r2 r3 rest

```

```

47         temp:temps' = temps
48         (randoms'', restBest, restEvald)
49           = chMut randoms' temps' rest
50         (randoms'', hdBest, hdEvald) = if doCrossover
51           then crs randoms' papa mama
52           else mut temp randoms' worst
53         bests = hdBest:restBest
54         evald = hdEvald:restEvald
55         in evald++(aux bests randoms'')
56         iniScore = score (a, ini)
57         iniQuad = (iniScore, expChoicesEmpty, a, ini)
58     in aux (take (length temps) (repeat iniQuad)) randoms
59
60 breed ::
61     Ord score => Float -> [Float]
62     -> ((a, s) -> score) -> ExpState s a -> s -> a
63     -> [(score, ExpChoices, a, s)]
64
65 breed pCrossover temps score es ini a =
66     breed' (randoms (mkStdGen 42)) pCrossover temps score es ini a

```

Plusieurs arguments de `breed'` portent le même nom et jouent le même rôle que ceux de `simulateAnnealing'` – nous ne les détaillerons pas à nouveau. Comme pour le recuit simulé, le résultat retourné correspond à une liste infinie des designs évalués. À l’invocation, l’utilisateur doit passer une liste de températures, qui vont être celles utilisées lors des appels à la fonction `mutate`, empruntée au recuit simulé. Chaque température de la liste sera utilisée lors de la mutation d’un individu, du plus mauvais au meilleur en fonction du score obtenu. On a donc tout intérêt à placer des températures élevées en début de liste et faibles en fin de liste. Cette façon de procéder a également pour conséquence que le nombre de températures spécifiées est égal à la population à une itération donnée.

Comme d’habitude, nous mettons à disposition de l’utilisateur une version allégée de la fonction `breed'`, appelée `breed`, qui lui épargne la construction de la liste de nombres aléatoires.

2.3.2.4 Évaluation du score dans un contexte monadique

Il y a des cas où l’on peut souhaiter que l’évaluation du score correspondant à un paramétrage donné du design soit effectuée dans une monade. C’est en particulier le cas lorsque cette évaluation nécessite l’utilisation d’outils externes, comme des simulateurs ou des compilateurs, auquel cas il est obligatoire que le score soit évalué dans `IO`, la monade d’entrées/sorties⁷. Dans une moindre mesure, on peut vouloir profiter de l’expertise de l’utilisateur pour évaluer une configuration donnée (ce qui demande également à interagir avec l’environnement) ou optimiser l’évaluation du score par le biais de monades telles que `ST`, permettant de manipuler efficacement des références.

Les modifications à apporter aux fonctions d’exploration telles que vues précédemment sont relativement minimales. Par exemple, la version monadique du recuit simulé donne :

```

1 simulateAnnealingM' ::
2     (Ord score, Monad m) => [Float] -> Int -> (Integer -> Float)
3     -> ((a, s) -> m score) -> ExpState s a -> s -> m [(score, ExpChoices, a, s)]
4 simulateAnnealingM' ::
5     (Ord score, Monad m) => Int -> (Integer -> Float)
6     -> ((a, s) -> m score) -> ExpState s a -> s -> m [(score, ExpChoices, a, s)]
7
8 simulateAnnealingM' randoms n f scoreM es ini =
9     let aux i best randoms (intensity:intensities) choices = do
10         let (randoms', choices')
11             = mutate intensity randoms choices
12         let (choices'', a, s)
13             = runExpState es "" choices' ini

```

7. À moins d’avoir recours à une fonction telle que `performUnsafeIO` (dont la signature est `IO a -> a'`), mais il s’agit d’une méthode propice aux erreurs et pouvant engendrer un comportement indéfini du programme lorsqu’elle est mal utilisée. On considérera donc qu’elle est à proscrire.

```

14     val <- scoreM (a, s)
15     let (best', choices'') = case best of
16         Just b | b > val -> (best, choices)
17         otherwise -> (Just val, choices'')
18     let r' = aux (i+1) best' randoms' intensities choices''
19     rest <- if (i < n) then r' else return []
20     return ((val, choices''), a, s):rest
21 in aux 0 Nothing randoms (map f [0..]) expChoicesEmpty
22
23 simulateAnnealingM n f scoreM es ini
24 = simulateAnnealingM' (randoms (mkStdGen 42)) n f scoreM es ini

```

L'une des principales conséquences de l'évaluation du score dans une monade est qu'il n'est plus possible d'utiliser l'évaluation paresseuse pour produire une liste infinie de designs paramétrés comme nous l'avons fait précédemment. En effet, si la liste encapsulée dans la monade retournée par `simulateAnnealingM` était infinie, il deviendrait impossible de réaliser une action dans cette monade après l'avoir récupérée par le biais de l'opérateur `<-` (car cela entraînerait l'évaluation de tous les éléments de cette liste). Par conséquent, nous avons dû ajouter un argument supplémentaire, `n`, spécifiant la longueur de la liste souhaitée. En conséquence, on pourra noter que `take n (simulateAnnealing score es ini)` est équivalent à `simulateAnnealingM n (return score) es ini`.

Le code ne présente aucune autre difficulté particulière par rapport à la version non-monadique.

2.4 Exemple et comparaison

Dans cette section, nous allons définir un exemple de modèle de calcul et de moteur associé, puis tester dessus les différents algorithmes d'exploration décrits précédemment.

2.4.1 Description et pertinence

Nous avons choisi un exemple simple, à la fois sur le plan conceptuel que sur celui de l'implémentation, au détriment peut-être, dans une certaine mesure, d'une parenté claire avec le design niveau système.

Dans notre exemple, le «design» sera un ensemble de cercles et de rectangles situés sur un canevas carré. Chacune de ces primitives disposera d'une couleur stockée sous la forme de trois composantes (rouge, vert, bleu) et d'un canal d'opacité. Grâce aux primitives d'exploration, nous définirons un espace de design de grande taille, que nous explorerons ensuite en essayant de minimiser l'erreur quadratique moyenne entre le contenu du canevas et une image de référence.

Bien qu'*a priori* éloigné des problématiques de design matériel, cet exemple permet de retrouver des phénomènes assez similaires à bien des égards au niveau de l'exploration de l'espace de design. Notamment, le positionnement de composants dans les NoCs à topologie matricielle, ou encore la détermination de la taille optimale de blocs de threads dans les réseaux de processeurs (tels que les GPUs) sont des exemples de tâches impliquant la manipulation de coordonnées bidimensionnelles au cours de l'exploration architecturale. Ces deux cas de figure seront brièvement traités au chapitre 4. Dans les modèles de calcul «réels», on utilisera des primitives correspondant à des déclarations, instanciations ou structures de contrôles en lieu et place de cercles et de rectangles, mais de façon assez intéressante, cette distinction n'a pas trop d'impact du point de vue du déroulement de l'étape d'exploration.

2.4.2 Implémentation

On définit le MoC simplement de la façon suivante :

```

1 type Color = (Float, Float, Float, Float)
2 type Point = (Float, Float)
3

```

```

4 class Explorable em => Picture em where
5   circle :: Point -> Float -> Color -> em ()
6   rectangle :: Point -> Point -> Color -> em ()

```

Les composantes d'une couleur sont comprises entre 0 et 1. Les coordonnées d'un point également, si on veut qu'il soit dans le canevas. Un cercle est construit à partir de son centre, son rayon et sa couleur, tandis qu'un rectangle a besoin de deux sommets opposés et d'une couleur. Comme annoncé précédemment, la définition de notre modèle de calcul est minimaliste. On peut également remarquer que les fonctions `circle` et `rectangle` ne retournent rien. Nous procédons ainsi du fait que, dans cet exemple, on peut se dispenser de permettre au designer d'altérer les primitives après leur création. Nous verrons à plusieurs occasions (par exemple, aux section 3.1 et 3.2) comment faire lorsqu'on souhaite lui donner la possibilité de manipuler les objets qu'il crée.

Construisons une petite bibliothèque de fonctions utiles par dessus notre MoC :

```

1 anyFloat :: Explorable em => String -> em Float
2 anyPoint :: Explorable em => String -> em Point
3 anyColor :: Explorable em => String -> em Color
4 anyGray :: Explorable em => String -> em Color
5
6 anyFloat name = namedOneOf name (map (\x -> fromIntegral x * 0.01) [0..100])
7 anyPoint name = do
8   x <- anyFloat (name++".x")
9   y <- anyFloat (name++".y")
10  return (x, y)
11 anyColor name = do
12   r <- anyFloat (name++".r")
13   g <- anyFloat (name++".g")
14   b <- anyFloat (name++".b")
15   a <- anyFloat (name++".a")
16  return (r, g, b, a)
17 anyGray name = do
18   l <- anyFloat (name++".l")
19   a <- anyFloat (name++".a")
20  return (l, l, l, a)
21
22 anyCircle :: Picture em => String -> (String -> em Color) -> em ()
23 anyRect :: Picture em => String -> (String -> em Color) -> em ()
24 anyPrim :: Picture em => String -> (String -> em Color) -> em ()
25
26 anyCircle name col = do
27   center <- anyPoint (name++".center")
28   radius <- anyFloat (name++".radius")
29   color <- col (name++".color")
30   circle center (radius/3.0) color
31
32 anyRect name col = do
33   (x1, y1) <- anyPoint (name++".p")
34   (x2, y2) <- anyPoint (name++".q")
35   let corner1 = (min x1 x2, min y1 y2)
36       corner2 = (max x1 x2, max y1 y2)
37   color <- col (name++".color")
38   rectangle corner1 corner2 color
39
40 anyPrim name col =
41   let possibilities = map (\f -> f name col) [anyCircle, anyRect]
42   in namedEquiprobable (name++".kind") possibilities

```

Les fonctions `anyFloat`, `anyPoint`, `anyColor` et `anyGray` peuvent être utilisées de n'importe quel modèle de calcul, tandis qu'`anyCircle`, `anyRect` et `anyPrim` sont spécifiques à `Picture`. En particulier, la fonction `anyFloat` construit un flottant parmi les 101 appartenant à l'ensemble $\{0, 0.01, 0.02, \dots, 0.99, 1\}$. Cela est suffisant car nous utiliserons une image de référence de faible résolution. Pour gérer des ensembles plus grands sans utiliser de listes chaînées excessivement longues, chaque bit pourrait être placé individuellement par un ensemble de deux alternatives qui lui serait propre, par exemple.

2.4. EXEMPLE ET COMPARAISON

Un moteur trivial pour le modèle de calcul `Picture` pourrait être :

```
1 data Picto = Picto
2   { pictoCircles :: [(Point, Float, Color)]
3     , pictoRects  :: [(Point, Point, Color)]
4   } deriving Show
5
6 nPrimitives :: Num a => Picto -> a
7 nPrimitives picto =
8   let n = length (pictoCircles picto) + length (pictoRects picto)
9       in fromIntegral n
10
11 addCircle circle picto =
12   let circles = pictoCircles picto
13       in picto { pictoCircles = circle : circles }
14
15 addRect rect picto =
16   let rects = pictoRects picto
17       in picto { pictoRects = rect : rects }
18
19 instance Picture (ExpState Picto) where
20   circle center radius color = modify (addCircle (center, radius, color))
21   rectangle c1 c2 color = modify (addRect (c1, c2, color))
```

Comme ce moteur se contente de faire la liste des primitives (au lieu, par exemple, de maintenir un tableau de pixels), la partie composition des couleurs sera effectué au moment de l'évaluation du score :

```
1 data Pixel = Pixel
2   { pixelPos :: Point
3     , pixelColor :: Color
4   }
5
6 colorError (r1, g1, b1, a1) (r2, g2, b2, a2) =
7   let (dr, dg, db, da) = (r2 - r1, g2 - g1, b2 - b1, a2 - a1)
8       in dr*dr + dg*dg + db*db + da*da
9
10 composite :: [Color] -> Color
11 composite [] = (1.0, 1.0, 1.0, 1.0)
12 composite ((r, g, b, a):cols) =
13   let
14     (r', g', b', a') = composite cols
15     f compo compo' = compo * a + compo' * (1.0 - a)
16   in (f r r', f g g', f b b', a + (1.0 - a) * a')
17
18 class WithGetPixel a where
19   getPixel :: Point -> a -> Color
20
21 instance WithGetPixel (Point, Float, Color) where
22   getPixel (x, y) ((cx, cy), r, col) =
23     let (dx, dy) = (cx - x, cy - y)
24         in if (dx*dx + dy*dy) < r*r
25           then col
26           else (0.0, 0.0, 0.0, 0.0)
27
28 instance WithGetPixel (Point, Point, Color) where
29   getPixel (x, y) ((x1, y1), (x2, y2), col) =
30     if (x >= x1) && (x < x2) && (y >= y1) && (y < y2)
31       then col
32       else (0.0, 0.0, 0.0, 0.0)
33
34 instance WithGetPixel Picto where
35   getPixel pt (Picto circles rects) =
36     let
37       cols1 = (map (getPixel pt) circles)
38       cols2 = (map (getPixel pt) rects)
```

```

39         in composite (cols1 ++ cols2)
40
41     pictoGetError picto pixels =
42         let error px = colorError (getPixel (pixelPos px) picto) (pixelColor px)
43         in sum (map error pixels) + 200 * max 0 (nPrimitives picto - 30)
44
45     score (_, picto) = negate (pictoGetError picto reference)

```

Pour évaluer l'erreur, l'image de référence sera convertie en une liste de pixels (type `Pixel`) avant d'être passé en deuxième argument à `pictoGetError`. Pour chaque pixel, la valeur au même point du canevas correspondant au design est calculée avec l'aide de la fonction `composite`, qui se contente de suivre les règles habituelles de la composition d'avant en arrière. (Il en résulte que l'ordre des primitives a une importance.) De plus, la fonction `pictoGetError` ajoute à l'erreur quadratique moyenne sur l'ensemble de l'image une pénalité par primitive utilisée au deçà de la trentième. Inclure ce genre de considérations est souvent rentable dans un design (par exemple, pour essayer de limiter les ressources utilisées). Notons enfin que la fonction `score`, que les algorithmes d'exploration essaient de *maximiser*, multiplie par -1 le résultat de `pictoGetError`. Le score le plus élevé est donc 0 (aucune différence avec l'image de référence, moins de 30 primitives utilisées).

En guise de design, spécifions à titre d'exemple un canevas avec 15 primitives (cercles ou rectangles) complètement aléatoires :

```

1 image :: Picture em => em ()
2 image = sequence_ (map (\n -> anyPrim ("prim"++show n) anyColor) [1..15])

```

(La fonction `sequence_` enchaîne les actions monadiques de la liste fournie en argument dans l'ordre où elles y figurent. Elle est brièvement discutée à la section A.5.3.)

Il ne reste plus qu'à l'explorer, par exemple via l'algorithme génétique vu à la section 2.3.2.3 :

```

1 main :: IO ()
2 main = do
3     let iterations = breed
4         score — fonction d'évaluation du design
5         image — design lui-même
6         (Picto [] []) () — état initial
7     let best = pickBest (take 500 iterations)
8     putStr (show best)
9

```

(Rappelons que les `let` situés dans une construction `do` ne sont pas des actions monadiques, et se comportent exactement comme s'ils étaient en dehors. C.f. section A.4.)

2.4.3 Vitesses de convergence

Ce design permet de mettre à l'épreuve les méthodes d'explorations que nous avons exprimées dans la section 2.3. Bien qu'il soit très concis, le nombre de combinaisons possibles est clairement au delà de ce que pourrait permettre d'attaquer une approche exhaustive⁸.

Les figures 2.2, 2.3 et 2.4 illustrent la qualité des designs évalués par les différents algorithmes au cours de leur progression. On constate sans étonnement que le parcours aléatoire ne fournit pas de résultats s'améliorant en moyenne avec le temps, contrairement au recuit simulé. L'optimisation génétique a pour particularité de produire régulièrement des individus de mauvaise qualité en raison des croisements ratés, mais s'avère plus efficace que le recuit simulé au final.

La figure 2.5 permet de comparer la vitesse de convergence des trois algorithmes. La programmation génétique offre les meilleurs résultats sur la durée, tandis que les deux autres méthodes présentent l'avantage d'offrir un design convenable en peu d'itérations.

8. Il s'élève à : $n = (101^{2+1+4} \times 101^{2+2+4})^{15} \approx 3,83 \times 10^{240}$

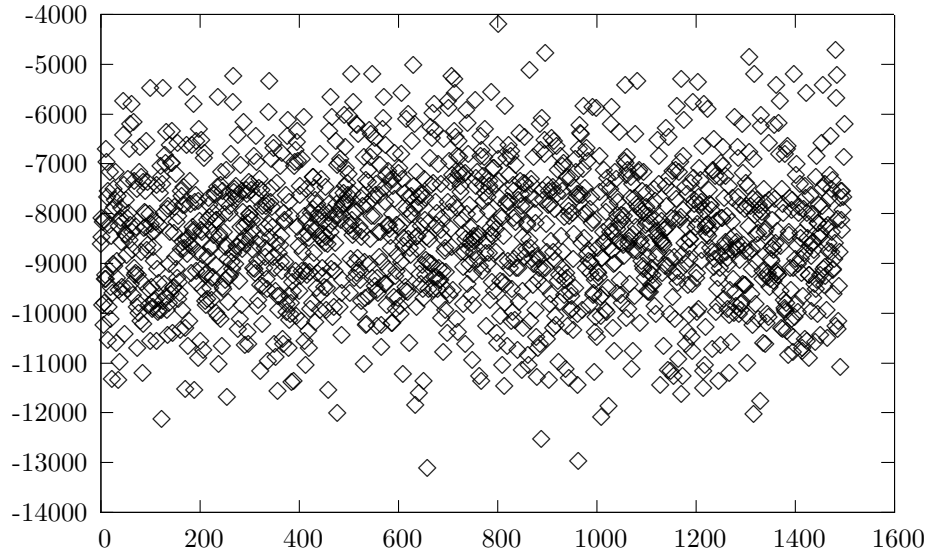


FIGURE 2.2 – Parcours aléatoire de l'espace de design – score des designs évalués en fonction de l'itération

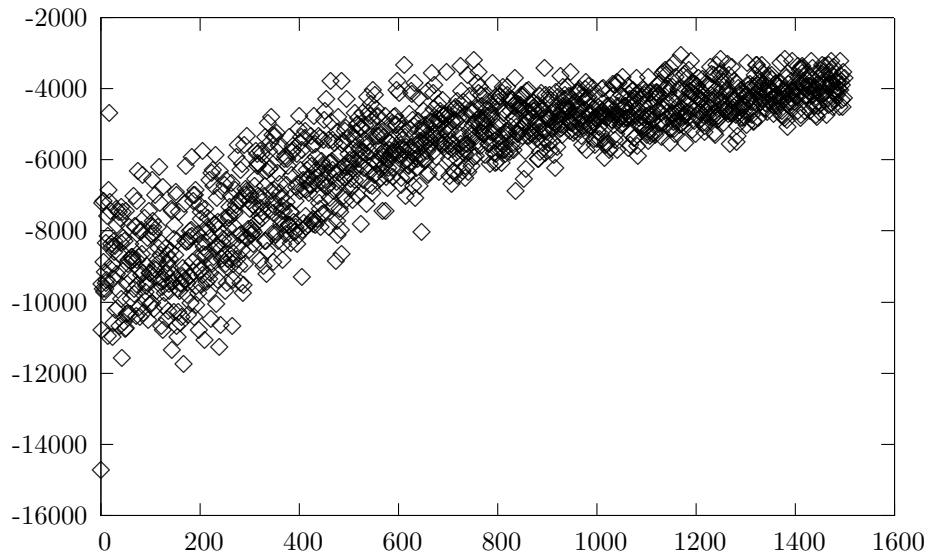


FIGURE 2.3 – Recuit simulé – score des designs évalués en fonction de l'itération

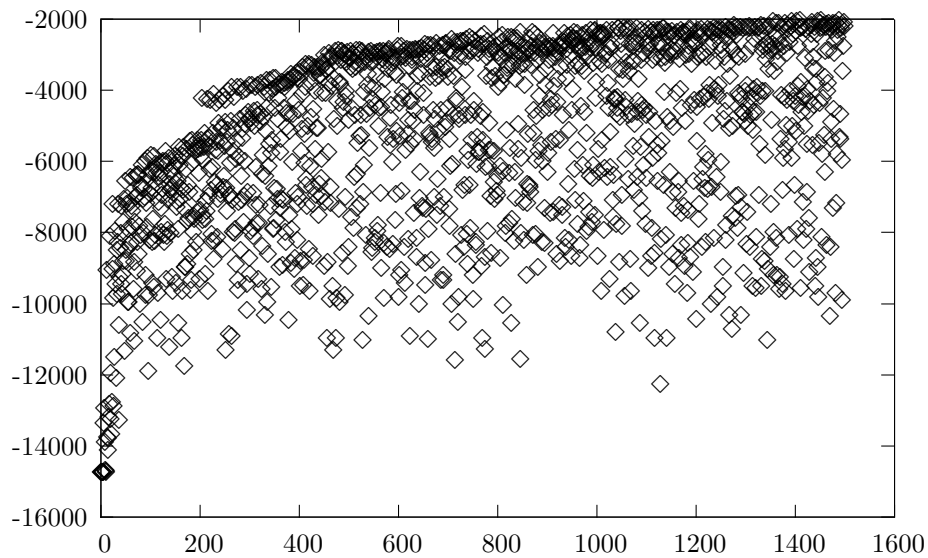


FIGURE 2.4 – Programmation génétique – score des designs en fonction du nombre de designs précédemment évalués

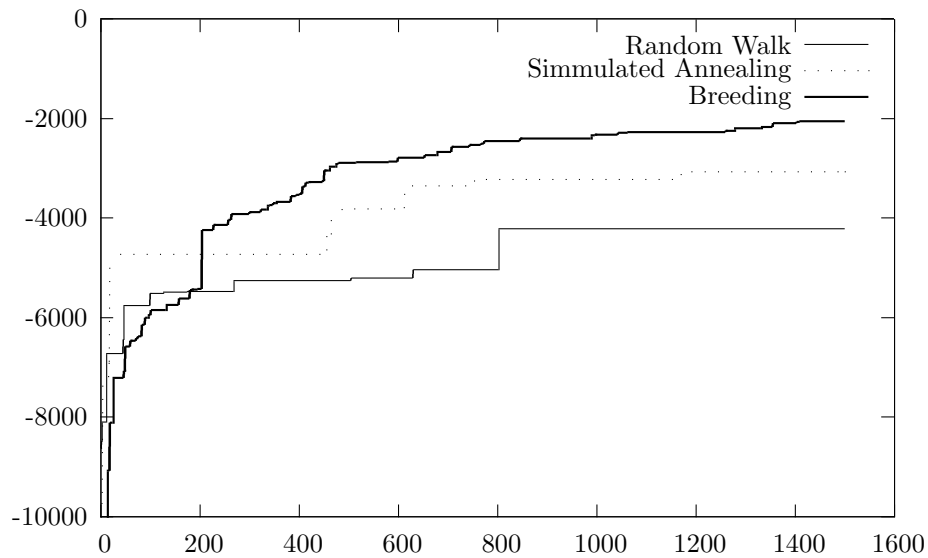


FIGURE 2.5 – Comparaison des meilleurs scores trouvés en fonction du nombre de designs évalués

2.5 Conclusion

Ce chapitre nous a permis de poser les bases de la méthodologie proposée. Nous avons défini la monade d'état explorable, donné le principe de son utilisation pour la description de modèles de calculs ainsi que de moteurs associés (dont l'usage peut varier : simulation, génération de code intermédiaire, etc.), et décrit la façon d'implémenter diverses méthodes d'exploration utilisables en conjonction à eux. Enfin, nous avons illustré le fonctionnement de tous ces éléments combinés sur un exemple très simple mais néanmoins assez illustratif.

Il nous reste à tester notre méthodologie sur des exemples plus concrets utilisant des modèles de calcul plus proches de ceux utilisés dans la réalité. C'est ce que nous allons faire au chapitre suivant.

Chapitre 3

Études de cas : modélisation bit-accurate sans horloge et modélisation niveau RTL

Dans ce chapitre, nous allons décrire deux architectures de reconstruction tomographique à l'aide notre approche, à des niveaux d'abstraction distincts. Elles nous permettront de voir comment mettre en œuvre de deux façons différentes les techniques présentées au chapitre précédent pour résoudre des problèmes rencontrés en pratique dans la conception architecturale. De manière plus spécifique, l'évaluation du design dans notre approche va se traduire soit par sa *simulation*, soit par son *élaboration*, en fonction de la façon dont un moteur est écrit. C'est ces deux cas de figure que nous allons illustrer.

Le principe de la reconstruction en tomographie PET est résumé en annexe B. Le lecteur intéressé pourra y trouver la signification des variables utilisés dans les listings qui seront discutés dans ce chapitre. Nous avons choisi la reconstruction tomographie en guise d'exemple en raison de l'impact fort de l'organisation de la hiérarchie mémoire sur les performances obtenues ; en cela, il s'agit d'un cas d'école très intéressant.

3.1 Modélisation sans horloge

3.1.1 Objectifs

On se propose dans un premier temps de réaliser un modèle sans horloge d'une seule unité de rétroprojection. Cette unité sera raccordée à un modèle de cache, dont la taille, l'associativité et le comportement seront également spécifiés. L'idée est de faire en sorte que, sur cet exemple, l'affectation du design au moteur corresponde à une étape de *simulation*.

3.1.2 Modèle de calcul

On définit un MoC permettant de manipuler des variables entières mutables que nous appellerons accumulateurs, et qui seront selon le cas interprétables comme des vecteurs de bits, ou simplement comme des variables d'instrumentation. Le MoC définira les structures de contrôle que l'on voit habituellement en programmation impérative, et permettra de convertir le contenu d'un accumulateur en `Integer`. Ceci a pour conséquence qu'un moteur de ce MoC devra nécessairement effectuer les calculs au fur et à mesure que les constructions monadiques sont évaluées (en d'autres termes, il devra — comme prévu — simuler le design). En effet, puisqu'il est possible de connaître à tout moment la valeur d'un accumulateur, un simple branchement via un `if` ou via *pattern matching* pourra faire varier les actions monadiques évaluées en fonction de cette valeur. Il s'agit d'une contrainte importante, rendant par exemple pratiquement impossible la génération de code

intermédiaire à partir du design de l'utilisateur (puisque pour cela on aurait eu besoin de récupérer l'équivalent de l'arbre syntaxique abstrait correspondant à chaque «if», etc.). Nous verrons un moyen d'y passer outre à la section 3.2. Revers de la médaille, ce mode d'évaluation offre comme avantage de permettre une utilisation aisée de fonctions existantes sur le contenu des accumulateurs.

En suivant ces principes de design, nous posons donc :

```

1 type family Accu en :: * — famille de types representant un accumulateur
2
3 class Explorable em => Accus en em | em -> en where — em -> en: dépendance fonctionnelle
4   — Obligatoire (gestion des accumulateurs)
5   namedAccu :: String -> em (Accu en)
6   (#=) :: Accu en -> Integer -> em ()
7   accuVal :: Accu en -> em Integer
8   accuFromName :: String -> em (Accu en)
9
10  — Obligatoire (gestion des fichiers)
11  open :: String -> Int -> em Bool — renvoie vrai si le fichier a été créé
12  append :: String -> em ()
13  goToSection :: Int -> em ()
14
15  — Avec implémentations par défaut
16  accu :: em (Accu en)
17  (+=) :: Accu en -> Integer -> em ()
18  (-=) :: Accu en -> Integer -> em ()
19  while :: em Bool -> em () -> em ()
20  whilst :: Accu en -> (Integer -> Bool) -> em () -> em ()
21  setPrettyPrinter :: Accu en -> (Integer -> String) -> em ()
22
23  accu = namedAccu "<anonymous_accu>"
24  x += y = do tmp <- accuVal x; x #= (tmp + y)
25  x -= y = do tmp <- accuVal x; x #= (tmp - y)
26  while cond block = do
27    continue <- cond
28    if continue then do block; while cond block else return ()
29  whilst x cond' block =
30    let cond = do tmp <- accuVal x; return (cond' tmp)
31    in while cond block
32  setPrettyPrinter accu printer = return ()

```

Avant d'entrer plus avant dans le détail, soulignons l'utilisation de familles de types dans cet exemple de modèle de calcul (ligne 1). Les familles de type sont une extension de Haskell offerte par le compilateur GHC. À plus long terme, leur inclusion est prévue dans le standard Haskell'. Ici, nous nous en servons pour permettre d'une façon propre aux implémenteurs des moteurs de définir les types de données qui pourront être «construits» et manipulés par les primitives définies dans le modèle de calcul. L'utilisation de familles de types permet de rajouter une couche de polymorphisme supplémentaire souhaitable bien que non absolument indispensable en pratique. Néanmoins, il est intéressant de comprendre la logique qu'il y a derrière :

- La classe correspondant à un modèle de calcul doit prendre au moins un paramètre, à savoir un type monadique dans lequel va être exprimé le design ; mais elle peut en prendre d'autres, correspondant aux types manipulés par les primitives du MoC – ainsi, le choix des représentations les plus adéquates est laissé à chaque moteur
- Une première solution est de passer directement en argument les types en question ; par exemple, on pourrait envisager de définir `Accus accu em`, avec des fonctions aux signatures similaires à `namedAccu :: String -> em accu` à l'intérieur
- Cependant, s'il est conceptuellement intéressant de grouper un certain nombre de types manipulés entre eux, il vaut mieux passer en argument un ou plusieurs types que nous appellerons «associés», dont on va se servir pour paramétrer des familles de types représentant les objets manipulés ; ainsi, un même type associé pourra être partagé par plusieurs MoCs dans l'hypothèse où cela présenterait un sens (par exemple, entre MoCs partageant une représentation similaire du temps, etc.)

- Pour que la situation reste en pratique gérable, l'utilisation de dépendances fonctionnelles est quasiment indispensable lorsque l'une ou l'autre de ces techniques est utilisée; schématiquement, il faut préciser au compilateur que les types représentant les objets manipulés découlent du type monadique employé (c.f. «`em -> en`» à la ligne 3); pour plus d'information sur les familles de types, le lecteur intéressé pourra se référer à [19]

L'intérêt de l'utilisation de familles de types croît lorsque l'on projette de construire une bibliothèque complète de modèles de calcul coopérant entre eux. La *réutilisation de code* devient alors significative et la communication entre MoCs devient plus simple et plus flexible. Hélas, nous ne pouvons prétendre avoir réalisé un travail d'une exhaustivité suffisante au cours de cette thèse pour que cet intérêt soit flagrant dans les exemples de MoCs auxquels nous allons nous limiter dans ce manuscrit. En conséquence, dans les exemples qui vont suivre, nous utiliserons des types concrets pratiquement systématiquement pour représenter les objets manipulés dans les MoCs. Le présent MoC et le moteur associé permettent malgré tout de se faire une idée de ce qu'implique l'utilisation de familles de type du point de vue de l'implémentation.

La fonction `namedAccu` crée un accumulateur à partir de son nom. L'opérateur `#=` est notre opérateur d'affectation, similaire dans sa sémantique à l'opérateur `:=` présent dans d'autres langages. Ici, il place un entier dans un accumulateur. La fonction `accuVal` renvoie la valeur entière d'un accumulateur. Enfin, dans un souci d'exhaustivité, il est également possible d'obtenir un accumulateur à partir de son nom, via un appel de `accuFromName`. Toutes ces fonctions utilisent le type `Accu en` pour représenter un accumulateur.

L'utilisateur a également la possibilité de créer des logs en utilisant les fonctions `open`, `append` et `goToSection`. Ces fonctions ne permettent que d'écrire des logs, et non pas de lire des fichiers en entrée.

Enfin, un certain nombre de fonctions avec des implémentations par défaut sont là pour faciliter la tâche à l'utilisateur. Pour une grande part (`accu`, `+=`, `--`, `while` et `whilst`), ces fonctions auraient pu être déclarées en dehors de la classe; en les déclarant dedans, on permet au moteur d'en fournir une implémentation plus efficace que celle par défaut. La fonction `setPrettyPrinter`, quant à elle, permet d'accéder à une fonctionnalité optionnelle des moteurs. Elle permet de spécifier une fonction de conversion de la valeur entière d'un accumulateur en une chaîne de caractères facile à lire pour l'utilisateur final¹. Cela implique que le moteur correspondant offre des mécanismes pour afficher le contenu des accumulateurs. Par défaut, la fonction `setPrettyPrinter` ne fait rien.

3.1.3 Moteur

On peut dénombrer deux fonctionnalités assez distinctes de MoC `Accus` :

- la gestion des accumulateurs eux-mêmes (c.-à-d. la manipulation de valeurs pendant la simulation)
- la possibilité de gérer des logs ou autres fichiers

Il est préférable de les implémenter distinctement (la gestion des fichier pouvant être réutilisée ailleurs, par exemple). Commençons par la gestion des fichiers :

```

1 data FilesEngine = Files Int Int [(String, [String])]
2
3 type Files = ExpState FilesEngine
4
5 filesEmpty = Files (-1) (-1) []
6
7 ensureFile :: String -> Int -> FilesEngine -> (Int, FilesEngine)
8 getCurrentFileContents :: FilesEngine -> String
9 appendToCurrentFile :: String -> FilesEngine -> FilesEngine
10 writeFiles :: String -> FilesEngine -> IO ()
11
```

1. Cela peut se justifier dans plusieurs types de cas. Par exemple, si l'accumulateur stocke une chaîne de caractères ou la représentation binaire d'un nombre flottant, on peut avoir intérêt à faire la conversion avant l'affichage. Similairement, s'il contient *de facto* un type énuméré dérivant de `Show` et écrit via la fonction `fromEnum`, il peut être intéressant de prendre `printer x = show ((toEnum x) :: Type)`

3.1. MODÉLISATION SANS HORLOGE

```
12 -----
13 -- Monadic constructs --
14 -----
15
16 open :: String -> Int -> Files Bool
17 append :: String -> Files ()
18 goToSection n = modify (\(Files f s fs) -> Files f n fs)
```

Pour des raisons de concision et pour éviter un excès de banalités, le corps de la plupart des fonctions n'est pas donné. (On prendra régulièrement cette liberté dans la suite de ce manuscrit.)

Le type `FilesEngine` comporte comme information d'indice du fichier courant (-1 si aucun fichier n'a été ouvert), la section courante (on découpe les fichiers en sections pour des raisons pratiques), et une liste chaînée avec les noms et contenus des fichiers; on n'écrit pas directement sur le disque car cela nécessiterait d'utiliser la monade `IO`. La fonction `ensureFile` est utilisée par `open` pour obtenir l'indice d'un fichier à partir de son nom, en le créant si nécessaire. La fonction `writeFiles`, enfin, écrit le contenu des fichiers (stocké sous forme de `String` dans la liste chaînée) sur le disque via l'utilisation de la monade d'entrées/sorties `IO`.

On définit ensuite un moteur pour `Accus`, utilisant `FilesEngine` pour gérer les logs. On pose :

```
1 import Memnoc.Base
2 import Memnoc.Exploration
3 import Memnoc.MoC.Accus
4 import qualified Memnoc.Engine.Files as Files
5
6 data StdAccus = StdAccus {
7     stdAccusAccus :: [(String, Integer, Integer -> String)],
8     stdAccusFiles :: Files.FilesEngine
9 }
10
11 data StdAccu = StdAccu Int — indice de l'accu dans la liste
12
13 data StdAccusAssoc = StdAccusAssoc
14 type instance Accu StdAccusAssoc = StdAccu — type instanciant la *famille* Accu
15
16 stdAccusNewAccu :: String -> StdAccus -> (StdAccu, StdAccus)
17 stdAccusSetAccu :: StdAccu -> Integer -> StdAccus -> StdAccus
18 stdAccusAccuFromName :: String -> StdAccus -> StdAccu
19 stdAccusSetPrettyPrinter :: StdAccu -> (Integer->String) -> StdAccus -> StdAccus
20
21 callOnFiles :: (Files.FilesEngine -> (a, Files.FilesEngine)) -> ExpState StdAccus a
22 modifyFiles :: (Files.FilesEngine -> Files.FilesEngine) -> ExpState StdAccus ()
23
24 callOnFiles f =
25     let g accu =
26         let files = stdAccusFiles accu
27             (rv, files') = f files
28         in (rv, accu { stdAccusFiles = files' })
29     in call g
30 modifyFiles f = let g x = ((), f x) in callOnFiles g
31
32 getNFiles = callOnFiles (\files@(Files.Files _ _ list) -> (length list, files))
33
34 instance Accus StdAccusAssoc (ExpState StdAccus) where
35     namedAccu name = call (stdAccusNewAccu name)
36     x #= y = modify (stdAccusSetAccu x y)
37     accuFromName name = call (\s -> (stdAccusAccuFromName name s, s))
38     accuVal (StdAccu accu) = call (\s -> (snd3 ((stdAccusAccus s)!! accu), s))
39     open fileName nSections = do
40         nFiles <- getNFiles
41         f <- callOnFiles (Files.ensureFile fileName nSections)
42         modifyFiles (\(Files.Files _ _ files) -> Files.Files f 0 files)
43         nFiles' <- getNFiles
44         return (nFiles /= nFiles')
45     append contents = modifyFiles (Files.appendToCurrentFile contents)
46     goToSection n = modifyFiles (\(Files.Files f s fs) -> Files.Files f n fs)
```

```

47     setPrettyPrinter accu printer =
48         modify (stdAccusSetPrettyPrinter accu printer)
49
50 stdAccusEmpty = StdAccus [] Files.filesEmpty
51 getStdAccus :: State StdAccus () -> StdAccus
52 getAccuVal :: StdAccus -> String -> Maybe Integer
53
54 writeFiles directory accus = Files.writeFiles directory (stdAccusFiles accus)

```

Nous importons la partie fichiers du moteur depuis un module séparé, puisque certaines des fonctions qui y sont définies portent le même nom que certains membres de `Accus`. Les types de données définis sont :

- `StdAccus`, l'état qui va être encapsulé dans `ExpState`, contient une liste de triplets donc chaque élément correspond à un accumulateur, ainsi qu'un objet de type `Files.FilesEngine`; pour chaque accumulateur, on stocke le nom, la valeur courante et la fonction d'affichage (qui peut être changée grâce à `setPrettyPrinter`)
- `StdAccu`, le type qui sert à désigner un accumulateur, et qui contient juste son indice dans la liste des accumulateurs
- `StdAccusAssoc`, le type associé qui va servir à paramétrer la famille `Accu`; on définit `StdAccu` comme une instance de `Accu StdAccusAssoc`

Un certain nombre de fonctions destinées à être partiellement évaluées avant d'être passées à `call`, `inquire` ou `modify` sont présentes. Notons également la définition de `callOnFiles` et `modifyFiles`, qui a pour effet de n'altérer que le membre `stdAccusFiles` en laissant la liste des accumulateurs inchangée. Il s'agit de la façon la plus simple d'imbriquer un *moteur* dans un autre.

3.1.4 Modèles de caches

3.1.4.1 Cache unidimensionnel

Revenons au niveau du MoC pour introduire un modèle de cache *set associatif* unidimensionnel :

```

1  data CacheAssociativity
2      = FullyAssociative
3      | DirectMapped
4      | SetAssociative Integer — Slots par set ("n-associatif")
5
6  data Cache = Cache {
7      cacheName :: String,
8      cacheAssociativity :: CacheAssociativity,
9      cacheLineSize :: Integer,
10     cacheNSlots :: Integer
11 }
12
13 data CacheId en = CacheId {
14     cacheParameters :: Cache,
15     cacheNAccesses :: Accu en,
16     cacheNAccessesToRAM :: Accu en,
17     cacheSlotAddresses :: [Accu en], — -1 si non occupé
18     cacheSlotFillingDates :: [Accu en],
19     cacheSlotsRead :: [Accu en] — nSlots champs de lineSize bits
20 }
21
22 cache :: forall en em. Accus en em => Cache -> em (CacheId en)
23 cacheRead :: forall en em. Accus en em =>
24     CacheId en -> Integer -> Integer -> em ()
25
26 cache params@(Cache name associativity lineSize nSlots) =
27     let
28         mkArray :: String -> Integer -> em [Accu en]
29         mkArray attr 0 = return []
30         mkArray attr n =

```

```

31         let elemName = name++":_slot_"++(show n)++"_"++attr
32         in do
33             elem <- namedAccu elemName
34             rest <- mkArray attr (n-1)
35             elem #= (-1)
36             return (elem:rest)
37     in do
38         nAccs <- namedAccu (name ++ " :_#_accesses")
39         nAccsToRAM <- namedAccu (name ++ " :_#_accesses_to_RAM")
40         slotAddr <- mkArray "address" nSlots
41         fillingDates <- mkArray "filling_date" nSlots
42         reads <- mkArray "reads" nSlots
43         plotName <- return ((cacheName params) ++ "_lin_accesses")
44         open (plotName++".dat") 1
45         return (CacheId
46             params nAccs nAccsToRAM slotAddr fillingDates
47             reads actuallyRead loaded)
48
49 cacheRead id x s =
50     let
51         CacheId _ nAccs nAccsToRAM slotAddr fillingDates
52             reads actuallyRead loaded = id
53         findDatum [] [] [] = return Nothing
54         findDatum (addr:addr) (fDate:fDates) (read:reads) = do — ...
55         getDatum :: Maybe (Accu en, Accu en) -> em ()
56         getDatum Nothing = do — ...
57         getDatum (Just (fDate, read)) = do — ...
58     in do
59         datum <- findDatum slotAddr fillingDates reads
60         getDatum datum
61         nAccs += 1
62         open (nm ++ "_lin_accesses.dat") 1
63         nAccsInt <- accuVal nAccs
64         append ((show nAccsInt) ++ "\t" ++ (show x) ++ "\n");

```

Le type `CacheAssociativity` permet de spécifier le nombre de slots par set, et peut prendre les valeurs symboliques `FullyAssociative` et `DirectMapped` comme synonymes de n -associatif et 1-associatif, respectivement. Le type `Cache` est l'ensemble des paramètres que l'utilisateur doit fournir lors de la création d'un nouveau cache par le biais de la fonction `cache` : un nom, l'associativité, la taille d'une ligne de cache et le nombre de slots n . `CacheId`, quant à lui, contient des accumulateurs représentant l'ensemble de l'information sur l'état du modèle de cache : un compteur du nombre total d'accès au cache, du nombre total d'accès vers la RAM externe (en cas de miss), les adresses de base des slots, les dates auxquels chaque slot a été rempli², et une information permettant de savoir quels mots de chaque slot ont été lus (permet d'évaluer la proportion de données chargées inutilement). Les paramètres fournis par l'utilisateur lors de l'instanciation du cache sont également sauvegardés dans `CacheId`. On peut noter que le contenu du cache à proprement parler n'est pas stocké, puisque nous n'en avons pas besoin dans notre exemple : les accès mémoires lors de la rétroprojection en tomographie PET ne dépendent pas des données lues, donc il est possible de se dispenser de faire le calcul de la valeur des voxels lorsque le but est simplement d'étudier le comportement de la hiérarchie mémoire. Rajouter un tableau avec des accumulateurs stockant le contenu des slots ne présenterait pas de difficulté.

La fonction `cache` crée un nouveau cache, renvoyant l'objet `CacheId` correspondant. Sans surprises, elle crée les accumulateurs en question et leur affecte leurs valeurs initiales (par défaut 0). De plus, elle ouvre un fichier journal avec les adresses des accès mémoire effectués.

La fonction `cacheRead` effectue une lecture dans un cache `id`, en demandant un segment de s mots commençant à l'adresse x . C'est elle qui implémente le comportement du cache lui-même, et constitue donc la moëlle substantifique du modèle (90 lignes sans les parties que nous avons abrégées ici). Le cache a pour politique de remplacer le slot qui a été lu le moins récemment (LRU), puisqu'il s'agit d'un choix naturel pour l'algorithme de rétroprojection considéré ; la section 3.1.6

2. En fait, à défaut d'avoir une horloge, chaque «date» est le numéro de l'accès qui a déclenché le chargement du slot

permet de s'en persuader. Implémenter d'autres politiques serait aisément faisable, et souhaitable si le designer n'a pas de raisons de faire un choix *a priori* (auquel cas il a également intérêt à rendre ce paramètre explorable).

3.1.4.2 Cache multidimensionnel

Comme nous l'avons vu à la section B.2, nous aurons besoin de faire d'accéder à un tableau multidimensionnel durant la rétroprojection. Nous allons donc définir un modèle de cache multidimensionnel pour remédier à ce besoin.

```

1  data CacheND
2      = DumbCacheND
3      { cacheNDName :: String
4        , cacheNDBytesPerPixel :: Integer
5        , cacheNDDims :: [Integer]
6        , cacheNDAssociativity :: CacheAssociativity
7        , cacheNDLineSize :: Integer
8        , cacheNDNSlots :: Integer
9        , cacheNDAddr :: CacheND -> [Integer] -> Integer
10     }
11
12 data CacheNDId en
13     = DumbCacheNDId
14     { cacheNDParameters :: CacheND
15       , cacheNDCacheId :: CacheId en
16     }
17
18 cacheND :: forall en em. Accus en em => CacheND -> em (CacheNDId en)
19 cacheNDRead :: Accus en em => CacheNDId en -> [Integer] -> em ()
20
21 cacheND params@(DumbCacheND name c dims associativity lineSize nSlots addr) = do
22   cacheId <- cache (Cache name associativity lineSize nSlots)
23   plotName <- return
24     ((cacheNDName params)++"_"++(show (length dims))++"d_accesses")
25   open (plotName++".dat") 1
26   setGnuplotOutput plotName
27   plotGnuplot (" "++plotName++".dat ")
28   return (DumbCacheNDId params cacheId)
29
30 cacheNDRead cacheNDId@(DumbCacheNDId params cacheId) xs = do
31   c <- return (cacheNDBytesPerPixel params)
32   addr <- return ((cacheNDAddr params) params xs)
33   cacheRead cacheId addr c
34   open ((cacheNDName params)++"_"++(show (length xs))++"d_accesses.dat") 1
35   append (show (head xs))
36   foldr (>>) (return ()) (map (\x -> append ("\t"++(show x))) (tail xs))
37   append "\n"
38
39 addrNDLinear :: CacheND -> [Integer] -> Integer
40 addrNDMortonLSBFfirst :: CacheND -> [Integer] -> Integer
41 addrNDMortonMSBFfirst :: CacheND -> [Integer] -> Integer

```

Par similarité de nomenclature avec le cache ordinaire, les types de données `CacheND` et `CacheNDId` correspondent respectivement aux paramètres d'un cache *nD* et à l'ensemble de son état. Les paramètres incluent le nom du cache, le nombre d'octets par pixel, les dimensions du tableau, l'associativité du cache associatif sous-jacent, la taille de ses lignes de cache, son nombre de slots, et enfin une fonction d'adressage, convertissant une liste de coordonnées en une adresse mémoire. L'état se limite à une copie de ces paramètres et à un cache unidimensionnel encapsulé.

Insistons sur le nom des constructeurs de données `DumbCacheND` et `DumbCacheNDId`, qui diffèrent du nom des types de données associés. Cela est dû au fait qu'on puisse trouver souhaitable d'implémenter un cache multidimensionnel de façon plus subtile que simplement en encapsulant un cache set associatif. Par exemple, si on avait souhaité implémenter un cache adaptatif et prédictif comme [60], on pourrait poser :

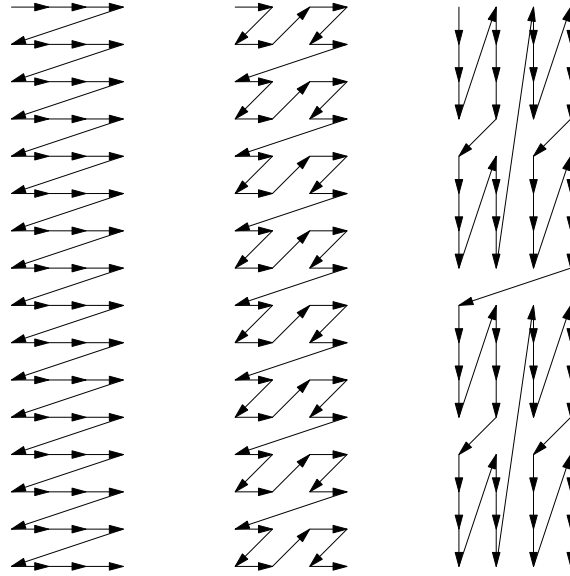


FIGURE 3.1 – De gauche à droite : l’ordre dans lequel sont adressés les éléments d’un tableau 4×16 , en utilisant les fonctions `addrNDLinear`, `addrNDMortonLSBFirst` et `addrNDMortonMSBFirst`

```

1 data CacheND
2   = DumbCacheND { ... }
3   | PredictiveCacheND { ... }
4
5 data CacheNDId en
6   = DumbCacheNDId { ... }
7   | PredictiveCacheNDId { ... }

```

Le pattern matching permettrait alors d’implémenter un comportement prédictif tout en conservant la possibilité d’instancier un cache multidimensionnel via encapsulation d’un cache set associatif sans avoir à altérer le code existant.

Par ailleurs, trois fonctions d’adressage sont prédéfinies :

- `addrNDLinear`, pour un adressage scanline classique, qui concatène les bits de toutes les coordonnées (ex : $(x_4x_3x_2x_1, y_2y_1, z_3z_2z_1) \rightarrow z_3z_2z_1y_2y_1x_4x_3x_2x_1$)
- `addrNDMortonLSBFirst`, l’ordre de Morton [66] en entrelaçant les bits de poids faible en premier (ex : $(x_4x_3x_2x_1, y_2y_1, z_3z_2z_1) \rightarrow x_4z_3x_3z_2y_2x_2z_1y_1x_1$)
- `addrNDMortonMSBFirst`, l’ordre de Morton en entrelaçant les bits de poids fort en premier (ex : $(x_4x_3x_2x_1, y_2y_1, z_3z_2z_1) \rightarrow x_4y_2z_3x_3y_1z_2x_2z_1x_1$)

La figure 3.1 illustre les différences entre les trois méthodes, l’ordre de Morton étant réputé accroître la localité des accès mémoire. Les fonctions `addrNDMortonLSBFirst` et `addrNDMortonMSBFirst` sont équivalentes dans l’hypothèse où toutes les coordonnées du tableau sont égales.

3.1.5 Design

Le design du modèle d’unité de rétroprojection est le suivant :

```

1 printOrder order | order == 0 = "linear"
2 printOrder order | order == 1 = "z-order"
3 printOrder order | order == 2 = "z-order_(MSB_first)"
4 printAs as | as == 1 = "direct_mapped"
5 printAs as = show as ++ "-way"
6

```

```

7 possibleAs nSlots = map (2^) [0..(intlog2 (fromIntegral nSlots))]
8
9 instrument [] = idle
10 instrument ((var, name):xs) = do
11     acc <- namedAccu name
12     acc #≠ var
13     instrument xs
14
15 tomo :: Accus en em => em ()
16 tomo =
17     let
18         (psiMax, uMax, vMax) = (256, 256, 32)
19         dims = [psiMax, uMax, vMax]
20         — (x, y, z) sont dans [-1.0, 1.0]^3
21         r@(xo, yo, zo) = (0.2, 0.3, 0.1)
22         (dx, dy, dz) = (2.0 / 64.0, 2.0 / 64.0, 2.0 / 64.0)
23         (bx, by, bz) = (2, 2, 2) — taille du bloc
24         c = 4
25         delta = 1.0
26         u psi r@(x, y, z) =
27             let rv = (x * cos psi + y * sin psi + 1.0)
28                 in floor (rv * 0.5 * fromIntegral uMax)
29         v psi delta r@(x, y, z) =
30             let ra = 1.0
31                 a = x * sin psi - y * cos psi
32                 rv = delta / (2.0 * ra) * a + z + 2.0
33                 in floor (rv * 0.25 * fromIntegral vMax)
34         psiStep = 2.0 * 3.14159 / (fromIntegral psiMax)
35         getCoords it r =
36             let psi = (fromIntegral it) * psiStep
37                 in [it, u psi r, v psi delta r]
38     in do
39         lineSize <- oneOf [64, 128]
40         nSlots <- oneOf [1, 2, 4, 8, 16]
41         nAs <- oneOf (possibleAs nSlots)
42         as <- return (SetAssociative nAs)
43         order <- oneOf [0, 1, 2]
44         addr <- case order of
45             0 -> return addrNDLinear
46             1 -> return addrNDMortonLSBFirst
47             2 -> return addrNDMortonMSBFirst
48         cache <- cacheND (DumbCacheND "cache" c dims
49             as lineSize nSlots addr)
50         instrument [(nSlots, "#_slots"), (lineSize, "line_size")]
51         nAsAcc <- namedAccu "associativity"
52         orderAcc <- namedAccu "adressing"
53         nAsAcc #≠ nAs
54         orderAcc #≠ order
55         setPrettyPrinter orderAcc printOrder
56         setPrettyPrinter nAsAcc printAs
57         currentPsi <- namedAccu "current_psi"
58         whilst currentPsi (< psiMax)
59             ( do
60                 it <- accuVal currentPsi
61                 [0 .. (bx*by*bz - 1)] 'for' \i -> — boucle 3D
62                     let
63                         xi = i 'rem' bx
64                         yi = (i 'quot' bx) 'rem' by
65                         zi = i 'quot' (bx*by)
66                         x = xo + dx * fromIntegral xi
67                         y = yo + dy * fromIntegral yi
68                         z = zo + dz * fromIntegral zi
69                         r = (x, y, z)
70                     in cacheNDRead cache (getCoords it r)
71                 currentPsi += 1
72             )
73

```

associativity	# slots				
	1	2	4	8	16
direct mapped	1473 à 1922	574 à 1922	574 à 1922	380 à 1922	152 à 1922
2-way		646 à 1754	163 à 1754	163 à 1754	152 à 1754
4-way			174 à 847	152 à 847	152 à 847
8-way				152 à 447	152 à 447
16-way					152 à 371

TABLE 3.1 – Nombre de bursts vers la RAM, en fonction du nombre de slots et de la taille des sets

```

74
75 design :: Accus en em => em ()
76 design = tomo
77 accus = take 50 (map snd (randomWalk design stdAccusEmpty))

```

Le design `tomo` comporte dans la partie `let` des déclarations de fonctions construites à partir de l'équation B.1. En particulier, la fonction `getCoords r it` permet d'obtenir les coordonnées préalablement quantifiées de l'élément du sinogramme lu à la it^{eme} itération pour la détermination du voxel de coordonnées r dans l'espace d'origine³. Dans la partie `do` de `tomo`, on peut voir :

- un énoncé des paramètres explorables (on utilise `oneOf` à la place de `namedOneOf`, qui attribue une étiquette automatiquement au jeu d'alternatives) ; on pourra noter que le nombre de slots par set possible, qui dépend du nombre de slots total, est correctement calculé grâce à la fonction `possibleAs`⁴.
- l'appel à la fonction `instrument`, qui permet de créer des accumulateurs stockant des valeurs constantes (en effet, les objets de type `Accu en` correspondant aux accumulateurs créés sont perdus ; mais les accumulateurs existent néanmoins ; par exemple, il serait possible d'obtenir leur contenu via un appel de `show` sur le moteur)
- la mise en place de fonctions d'affichages pour certains accumulateurs utilisés comme des énumérations
- l'instanciation d'un cache nD tel que décrit précédemment à partir des paramètres explorables
- la boucle principale ; il est à noter que nous procédons à la reconstructions d'un voisinage de $2 \times 2 \times 2$ voxels à tout instant ; il a été démontré qu'il est bien plus efficace de procéder ainsi du point de vue de la localité des accès mémoire que de rétroprojeter chaque voxel individuellement [25]

En guise d'exemple d'utilisation du design, on prend une liste de 50 de ses évaluations via le moteur `stdAccus` en utilisant des paramètres choisis aléatoirement à chaque fois.

3.1.6 Résultats

La boucle principale comprend 2048 accès au sinogramme (`bx*by*bz*psiMax`), dont l'allure générale est donnée en figure 3.2. Il s'agit approximativement d'une sinusoïde tridimensionnelle. Les tables 3.1, 3.2 et 3.3 reflètent les mesures effectuées en parcourant exhaustivement l'espace de design. (Vu sa petite taille dans cet exemple, on peut se le permettre.) Chaque cellule indique les valeurs minimales et maximale obtenues en faisant varier tous les paramètres non fixés par les lignes et colonnes du tableau.

3. Alors, pourquoi trois coordonnées et pas quatre ? En fait, pour complètement calculer la valeur du voxel, il faudrait itérer sur ψ et Δ . Comme Δ est l'une des coordonnées du sinogramme, il n'y aura pas de réutilisation de données entre deux itérations successives sur Δ . On se contente donc d'étudier la boucle sur ψ en prenant Δ constante, sachant que cela n'aura pas d'incidence sur les performances mesurées.

4. La fonction `intlog2 x` s'évaluant à $\lceil \log_2 x \rceil$.

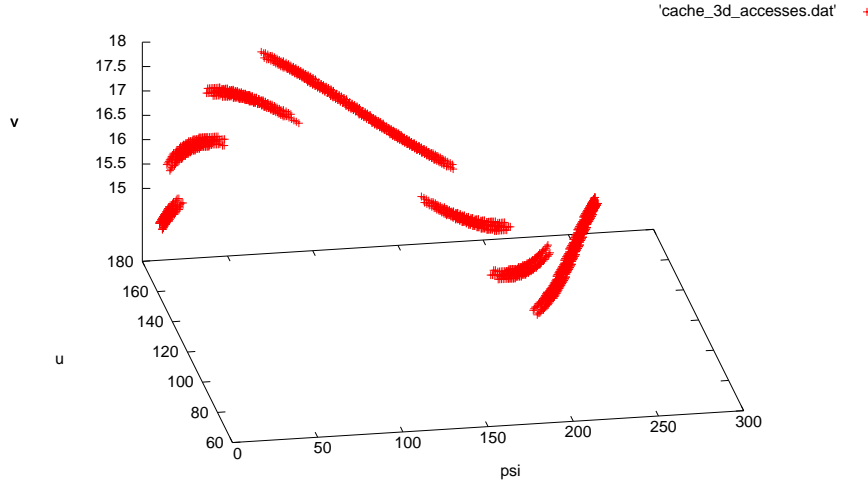


FIGURE 3.2 – À Δ fixé, coordonnées des accès au sinogramme pour la rétroprojection d’un voisinage de $2 \times 2 \times 2$ voxels dans l’espace d’origine

# slots	adressing		
	linear	z-order	z-order (MSB first)
1	1922	1473 à 1779	1494 à 1502
2	1752 à 1922	646 à 1623	574 à 1502
4	839 à 1922	231 à 1606	163 à 725
8	411 à 1922	215 à 1606	152 à 613
16	326 à 1922	215 à 380	152 à 449

TABLE 3.2 – Nombre de bursts vers la RAM, en fonction du nombre de slots et de la méthode d’adressage

# slots	line size	
	64	128
1	1502 à 1922	1473 à 1922
2	725 à 1922	574 à 1922
4	243 à 1922	163 à 1922
8	227 à 1922	152 à 1922
16	227 à 1922	152 à 1784

TABLE 3.3 – Nombre de bursts vers la RAM, en fonction du nombre de slots et de la taille d’une ligne de cache

On peut conclure de cette étude qu'un paramètre essentiel pour les performances de l'application est la politique d'adressage. Dans notre cas, comme les trois dimensions du sinogramme qui importent ont une forme assez écrasée ($256 \times 256 \times 32$), utiliser l'ordre de Morton en entrelaçant en premier lieu les bits de poids fort semble être la meilleure approche. On peut voir également qu'en utilisant l'indiciage *scanline* classique, les performances augmentent au fur et à mesure qu'on ajoute de nouveaux slots au cache, tandis qu'elles atteignent leur maximum très vite lorsqu'on utilise l'ordre de Morton – peu de gains au delà de 4 slots pour un choix adéquat des autres paramètres. L'impacte de l'associativité peut être observé sur la table 3.1. Un cache fortement associatif aide surtout dans les pires cas, typiquement lorsque la politique d'adressage ne préserve pas suffisamment la localité. Si on devait effectuer des choix architecturaux en se basant sur ces résultats, un bon compromis serait de prendre un cache avec 4 slots de 128 octets regroupés en 2 sets de 2 slots, adressé via l'ordre de Morton en entrelaçant les bits les plus significatifs en premier. En raison de la légère sous-optimalité de la politique de remplacement des données, un tel cache avec juste deux sets offre des performances un peu plus élevées qu'un cache *full associatif* identique par ailleurs.

3.1.7 Conclusions

Nous avons vu un exemple de MoC écrit de sorte à ce que les moteurs correspondants soient nécessairement des simulateurs évaluant au fur et à mesure le code de l'utilisateur. Nous appelons ceci un MoC interprété, par opposition aux MoCs déclaratifs donc un exemple sera donné à la section 3.2. Les MoCs interprétés rendent possible l'utilisation des bibliothèques Haskell existantes pour agir directement et de manière efficace sur les types de données manipulés dans le design (que se soit par le biais de code Haskell optimisé, ou par le biais de *bindings* avec des bibliothèques écrites dans d'autres langages). L'intérêt d'utiliser notre approche pour réaliser ce type de modèles, par rapports aux autres langages spécialisés dans la simulation et l'intégration de code performant existant (par exemple, SystemC), réside notamment dans la possibilité d'utiliser les méthodes d'exploration définies au chapitre 2. De plus, il reste possible et assez facile d'imbriquer si nécessaire des parties de design écrites dans des MoCs déclaratifs au sein de MoCs interprétés (l'inverse n'est en général pas vrai).

3.2 Modélisation au niveau RTL

3.2.1 Objectifs

On se propose cette fois de modéliser notre design au niveau RTL. Ce faisant, on aura accès à des informations précises sur son comportement temporel, la possibilité d'évaluer la longueur du chemin critique, etc. Le modèle de calcul que nous proposons est proche de celui du VHDL, avec par exemple la possibilité pour l'utilisateur d'écrire des processus dans un style impératif. Il se limite aux circuits synchrones ne comportant qu'un seul domaine d'horloge⁵. Il est centré autour du concept de *transfert* ; par ce terme, on désigne un circuit combinatoire associant à un ensemble d'entrées un ensemble de sorties. L'objectif est de pouvoir simuler le circuit exprimé par le designer, mais aussi de générer du code HDL intermédiaire en vue d'une simulation ou d'une synthèse par un outil externe (par exemple dans le but de faire de l'émulation sur carte). Cette fois, l'évaluation du design à l'aide d'un moteur va pouvoir correspondre à l'étape d'élaboration, qui sera clairement distincte des étapes de simulation ou de génération de code intermédiaire (désormais devenue possible) qui pourraient suivre.

5. Si nécessaire, il serait possible de définir un MoC englobant, permettant d'instancier et de faire communiquer plusieurs blocs appartenant à différents domaines d'horloge, et ainsi de construire un circuit globalement asynchrone, localement synchrone (GALS). Procéder ainsi non seulement ajouterait à la clarté du code, mais en plus limiterait les sources d'erreurs et permettrait d'aboutir à des simulateurs plus performants (conscients des domaines d'horloge, donc n'ayant pas forcément à utiliser un paradigme événementiel classique).

3.2.2 Modèle de calcul

Le modèle de calcul présenté ici est le plus complet de ceux développés au cours de cette thèse, et le plus représentatif des règles de programmation à suivre pour définir efficacement un MoC dans l'approche proposée. Un effet de bord de cet état de fait est que le code associé est assez long. Les extraits de code qui vont suivre ne comporteront donc que ce qui est essentiel à la compréhension des concepts sous-jacents, et font en général l'impasse sur les détails d'implémentation eux mêmes.

3.2.2.1 Transfert

Nous cherchons à donner un moyen au programmeur d'exprimer de façon simple et puissante les transferts. Un moyen pour y parvenir est l'emploi d'opérateurs afin de permettre la spécification d'expressions arithmétiques et logiques comme cela peut se faire dans les langages traditionnels. Cela dit, le programmeur doit également pouvoir clairement définir des *noeuds* correspondant aux sorties des portes logiques dont la sortie est connectée à plusieurs entrées d'autres portes (c.-à-d. dont le *fanout* est supérieur à une porte). Ce faisant, n'importe quel circuit combinatoire peut être explicité sous forme de transfert.

Le plus simple et le plus efficace pour y parvenir est de définir le concept de transfert sous forme monadique, en donnant à l'enchaînement d'actions une sémantique similaire à celle de l'opérateur \ll, \gg en C.⁶ En utilisant pour cela la monade d'état explorable, on offre de plus au programmeur la possibilité de spécifier des imbrications et alternatives au sein des parties purement combinatoires de son design. On posera donc :

```

1  data UnOp
2      = UnOpPadWithZeros | UnOpPadWithMSB
3          | UnOpNot | UnOpOrReduce | UnOpAndReduce deriving Eq
4
5  data BinOp
6      = BinOpSum | BinOpDiff | BinOpAnd | BinOpOr | BinOpXor
7          | BinOpMult | BinOpDiv | BinOpMod
8          | BinOpEq | BinOpNotEq | BinOpGt | BinOpGe | BinOpLt | BinOpLe
9          | BinOpSftL | BinOpSftR
10         | BinOpCat
11         deriving Eq
12
13 data TerOp = TerOpMux deriving Eq
14
15 data TrTree
16     = TrTreeNotSet
17         | TrTreeObject Int — identifiant
18         | TrTreeNode Int — identifiant
19         | TrTreeConstant Integer
20         | TrTreeUnOp Int UnOp TrTree
21         | TrTreeBinOp Int BinOp TrTree TrTree
22         | TrTreeTerOp Int TerOp TrTree TrTree TrTree
23         deriving Eq
24
25 data TrNode = TrNode String Type TrTree
26
27 data TrNodes = TrNodes Int [TrNode] — Indice noeud de départ, noeuds
28 type Transfer' = ExpState TrNodes
29 type Transfer ty = Transfer' (TTr ty)
30
31 — Évaluation:
32 unOpApply :: UnOp -> Int -> Integer -> Integer
33 binOpApply :: BinOp -> Int -> Integer -> Integer -> Integer
34 terOpApply :: TerOp -> Int -> Integer -> Integer -> Integer -> Integer

```

6. Cet opérateur C (assez rarement utilisé) permet de combiner plusieurs sous-expressions (ayant potentiellement des effets de bord) dans l'évaluation d'une expression. Il est un peu l'équivalent d'un $\ll; \gg$, mais séparant des expressions et non pas des déclarations ou instructions.

Les types `UnOp`, `BinOp` et `TerOp` désignent les opérations unaires, binaires et ternaires de base sur des chaînes de bits. Ici, nous avons défini comme opération de base ce qu'il est possible de réaliser à l'aide d'opérateurs en Verilog, ou en VHDL sur des types simples comme `unsigned` ou `std_ulogic_vector`. Nous aurions pu nous cantonner aux opérations logiques de base et implémenter tout le reste (par exemple, l'arithmétique) par dessus, mais il en aurait résulté un code HDL généré à partir du design fourni par l'utilisateur reimplementant l'arithmétique au lieu d'utiliser les possibilités natives du HDL utilisé. Il est également important de ne *pas* prédéfinir trop d'opérations de base, car cela complique dans une certaine mesure l'implémentation des moteurs. Les fonctions `unOpApply`, `binOpApply` et `terOpApply` sont des implémentations de références des opérations de base, agissant sur des entiers de taille arbitraire. Elles sont utilisées en simulation, ou lors de la génération de code intermédiaire lorsque certaines expressions peuvent être réduites à des constantes.

Le type `TrTree` définit récursivement un arbre combinatoire, pouvant comporter :

- des nœuds combinatoires partagés par plusieurs arbres, caractérisés par un numéro unique (voir plus loin)
- des «objets», dont la sémantique exacte varie en fonction du moteur utilisé (typiquement, il s'agit de registres) ; tout comme les nœuds, ils disposent d'un numéro propre à chacun d'eux
- des chaînes de bits constantes
- des opérations unaires, binaires ou ternaires, caractérisés par la nature de l'opération (parmi celles définies plus haut) et ses opérandes

Enfin, les types `TrNode` et `TrNodes` servent à représenter respectivement un nœud nommé et un ensemble de nœuds nommés. Le type code `TrNodes` comporte deux informations : une liste de nœuds nommés et le numéro identifiant le premier nœud de cette liste. Ainsi, n'importe quel arbre combinatoire pourra référencer ce nœud via l'expression `TrTreeNode numeroEnQuestion`. Le type `Transfer`¹ est un alias pour `ExpState TrNodes`, une monade d'état explorable comportant des nœuds nommés – dont la construction était notre objectif de départ. L'arbre combinatoire finalement retenu comme «valeur» correspondant à un transfert doit être placé dans la monade par la dernière action qui y est effectuée. C'est ce comportement qui est similaire à celui de l'opérateur `<, >` en C, aux suites d'instructions en Caml, etc.

On affine néanmoins encore un peu le concept en proposant un «système de types» pour les chaînes de bits utilisées dans les transferts. Si nous nous étions arrêtés ici, nous disposerions d'une syntaxe proche de celle de Verilog, avec la possibilité de manipuler des chaînes de bits mais pas d'information sur ce qu'elles représentent. En fait, ce ne serait pas vraiment gênant, puisqu'il aurait suffi d'encapsuler ces chaînes de bits dans des types Haskell pour retrouver un système ressemblant à du typage fort. Cela dit, l'information sur les types serait perdue (comprendre : non récupérée par le moteur), et les résultats de simulation ou de génération de HDL ne contiendraient que des opérations sur des chaînes de bits sans davantage de précision. Afin de faciliter le débogage sur le code VHDL produit, et d'accroître la lisibilité des résultats de simulation, on introduit les types `Type` (désignant les «types» des chaînes de bits) et `TTr` (désignant un arbre combinatoire accompagné de son «type», souvent une expression du type `Type`). Puis on définit `Transfer`, paramétré par le «type» `ty`, comme un alias pour `Transfer (TTr ty)`. Ça y est, nous venons de définir un *type monadique explorable représentant un transfert typé*, ce qui était notre objectif. Notons au passage que chacun des nœuds nommés comporte, comme on s'y attend, une information de «type» qui lui est propre, en plus de son nom et de l'arbre combinatoire qui lui est associé. Sans s'appesantir sur la définition exacte de `Type`, précisons juste que l'on met à disposition du designer les fonctions `uint` et `sint`, à la signature `Int -> Type`, qui correspondent aux entiers non-signés et signés de n bits, ainsi que la fonction `rint` (pour *ranged integer*), à la signature `(Int, Int) -> Type`, correspondant à tous les entiers compris entre deux bornes connues. Accessoirement, pour davantage de lisibilité et de concision, on pose aussi `bool = uint 1`, `uint8 = uint 8`, etc.

Une fois ce concept de transfert dûment défini, il convient d'explicitier les opérateurs permettant de le manipuler :

```
1 class WithArithmetics ty where
2     (/+/) :: Transfer ty -> Transfer ty -> Transfer ty
```

```

3      (/ - /) :: Transfer ty -> Transfer ty -> Transfer ty
4      (//) :: Transfer ty -> Transfer ty -> Transfer ty
5      (./) :: Transfer ty -> Transfer ty -> Transfer ty
6      (/%/) :: Transfer ty -> Transfer ty -> Transfer ty
7
8      bwn :: Transfer ty -> Transfer ty
9      (/&/) :: Transfer ty -> Transfer ty -> Transfer ty
10     (/|/) :: Transfer ty -> Transfer ty -> Transfer ty
11     (/^/) :: Transfer ty -> Transfer ty -> Transfer ty
12
13     (</) :: Transfer ty -> Transfer ty -> Transfer ty
14     (==/) :: Transfer ty -> Transfer ty -> Transfer ty
15     (>/) :: Transfer ty -> Transfer ty -> Transfer ty
16     (<=/) :: Transfer ty -> Transfer ty -> Transfer ty
17     (>=/) :: Transfer ty -> Transfer ty -> Transfer ty
18
19     (<</) :: Transfer ty -> Transfer ty -> Transfer ty
20     (>>/) :: Transfer ty -> Transfer ty -> Transfer ty
21
22     (/?) :: Transfer ty -> (Transfer ty, Transfer ty) -> Transfer ty
23
24 instance Num (Transfer Type) where
25     x + y = x /+/ y
26     x * y = x ./ / y
27     — etc...
28
29 performSumOp :: UnOp -> TTr Type -> TTr Type -> Transfer Type
30 performDiffOp :: BinOp -> TTr Type -> TTr Type -> Transfer Type
31 — etc...
32
33 instance WithArithmetics Type where
34     x /+/ y = do x' <- x; y' <- y; return (performSumOp x' y')
35     x /- / y = do x' <- x; y' <- y; return (performDiffOp x' y')
36     — etc...
    
```

Ces opérateurs sont calqués sur ceux de C, préfixés et suffixés d’un $\langle\langle/\rangle\rangle$ pour éviter les conflits avec les opérateurs prédéfinis dans le prélude. Ils tiennent compte de l’information de type accompagnant les chaînes de bits pour gérer correctement les cas un peu spéciaux (par exemple, le bit de signe lors de la multiplication d’entiers signés).

Il est intéressant de noter qu’une telle implémentation des opérateurs arithmétiques et logiques permet au moteur de déduire lorsque nécessaire les tailles optimales des vecteurs de bits représentant les résultats des calculs (par exemple en utilisant de l’arithmétique d’intervalles). Il devient alors envisageable d’avertir l’utilisateur à chaque construction où de l’information peut être implicitement perdue par débordement arithmétique. Il peut s’agit d’un confort non négligeable.

3.2.2.2 Hiérarchie de classes

On définit (figure 3.3) une hiérarchie de classes afin de construire le modèle de calcul RTL dont on a besoin. La figure 3.4 en donne une illustration simplifiée. Cet ensemble de classes forme un mini-écosystème permettant de réaliser un certain nombre de choses. Comme initialement prévu, il est possible de décrire du matériel au niveau RTL (classe `RTL`) mais aussi de façon impérative (classe `Imperative`), pour la spécification de logiciels par exemple. Le MoC `Multithreaded` (dont nous ne nous servirons pas pour l’implémentation RTL que nous allons donner en exemple dans la suite de ce chapitre) permet de faire tourner plusieurs threads logiciels écrits de façon impérative. Cet ensemble de classes peut permettre également à l’utilisateur de facilement construire, à l’aide d’alternatives, un bloc acceptant par exemple une partie du design décrite impérativement, et pouvant soit faire un processus matériel, soit synthétiser un processeur pour l’exécuter sous forme logicielle. Il s’agit d’une façon possible d’automatiser le partitionnement matériel/logiciel via l’étape d’exploration. La section 4.1 donnera un autre exemple de construction intéressante obtenue en utilisant cette hiérarchie.

Analysons plus en détail le rôle de chaque classe :

```

1  class SizedType ty where
2      size :: ty -> Integer
3      typeStandardize :: ty -> Type
4
5  class Explorable em => WithAffectation em where
6      (#=) :: SizedType ty => Transfer ty -> Transfer ty -> em ()
7
8  class Explorable em => WithFifos em where
9      namedFifo :: SizedType ty => ty -> String -> em (Fifo ty)
10     fifoNElems :: SizedType ty => Fifo ty -> em (Transfer Type)
11     fifoPop :: SizedType ty => Fifo ty -> em (Transfer ty)
12     fifoPush :: SizedType ty => Fifo ty -> Transfer ty -> em ()
13     fifoPeek :: SizedType ty => Fifo ty -> em (Transfer ty)
14     fifoPeek fifo = do
15         rv <- fifoPop fifo
16         fifoPush fifo rv
17     return rv
18
19  class WithAffectation em => WithNodes em where
20     namedNode :: SizedType ty => ty -> String -> em (Transfer ty)
21     addTransferNodes :: Transfer ty -> em (TTr ty)
22
23     -----
24     -- Imperative MoC --
25     -----
26  data Fifo ty = Fifo ty Int -- type, identifiant
27
28  class WithNodes em => Imperative em where
29     namedVar :: SizedType ty => ty -> String -> em (Transfer ty)
30     while :: Transfer Type -> em () -> em ()
31     iff :: Transfer Type -> em () -> em ()
32     elsiff :: Transfer Type -> em () -> em ()
33     wait :: em ()
34
35  class (Imperative em, WithFifos em) => ImperativeWithFifos em
36
37     -----
38     -- RTL MoC --
39     -----
40  class WithNodes em => RTL em where
41     namedReg :: SizedType ty => ty -> String -> em (Transfer ty)
42     defaultsTo :: SizedType ty => Transfer ty -> Transfer ty -> em ()
43     process :: Integer -> (forall em'. Imperative em' => em' ()) -> em ()
44
45     -----
46     -- Multithreaded MoC --
47     -----
48  class ImperativeWithFifos em => Multithreaded em where
49     thread :: Integer -> (forall em'. ImperativeWithFifos em' => em' ()) -> em()
50
51     -----
52     -- Transfer' as an engine --
53     -----
54  instance WithAffectation Transfer' where
55     x #= y = -- ...
56
57  instance WithNodes Transfer' where
58     namedNode ty name = -- ...
59     addTransferNodes x = x

```

FIGURE 3.3 – Hiérarchie de classes définissant les MoCs RTL et impératifs (listing)

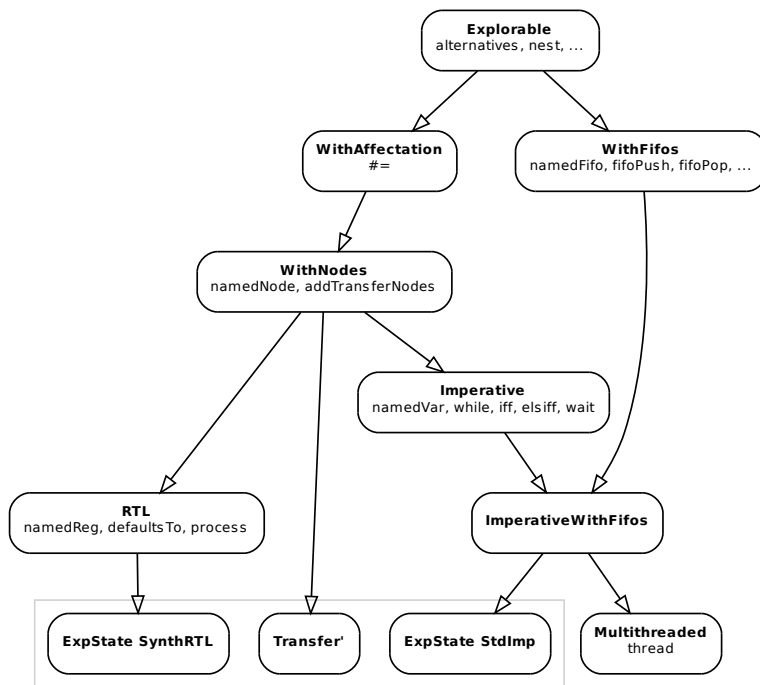


FIGURE 3.4 – Hiérarchie de classes définissant les MoCs RTL et impératifs (graphe). Les instances (moteurs) sont situées dans le rectangle gris clair.

- **SizedType** : correspond à un type de chaîne de bits qui peut être réduit au type **Type** que les moteurs savent manipuler. **Type** est une instance triviale de cette classe.
- **WithAffectation** : correspond aux MoCs pour lesquels il est possible d'affecter des nœuds ou des objets. L'opérateur correspondant est **#=**, et se comportera, comme nous les verrons, comme **:=** ou **<=** en VHDL, suivant ce sur quoi il est utilisé. L'arbre combinatoire correspondant à l'opérande de gauche («lvalue») doit forcément se réduire à **TrTreeObject n** ou **TrTreeNode n**.
- **WithFifos** : permet la manipulation de FIFOs infinies (non synthétisables)
- **WithNodes** : permet la création et l'utilisation de nœuds combinatoires. Le type **Transfer'** en est une instance (on pourrait dire que **WithNodes** est le MoC dont **Transfer'** est le moteur). Comme les nœuds combinatoires ont également un sens dans les MoCs RTL et **Imperative**, ils hériteront de cette classe. Il est alors important de leur donner la possibilité d'importer tous les nœuds combinatoires à partir d'une expression de type **Transfer' qqch**. Ceci est possible grâce à la fonction **addTransferNodes**. Notons également que, comme on s'y attend, **namedNode**, qui permet de créer un nouveau nœud, renvoie un transfert contenant un arbre combinatoire se réduisant à **TrTreeNode n**. Il est donc possible de l'utiliser comme lvalue. La sémantique de **#=** sur un nœud combinatoire est celle d'un **<=** en VHDL (hors processus, ou sur un signal faisant partie de la liste de sensibilité du processus combinatoire dans lequel se déroule l'affectation).
- **Imperative** : MoC impératif, offrant la possibilité de créer des variables, d'utiliser les structures de contrôle classiques, et d'attendre le front montant suivant de l'horloge (**wait**, qui est ignorée dans le contexte de l'écriture de logiciel). Le transfert produit par **namedVar** est une lvalue (dont l'arbre combinatoire se réduit à **TrTreeObject n**). La sémantique de **#=** sur une variable est celle de l'opérateur **:=** en VHDL, Ada, Pascal, etc.
- **ImperativeWithFifos** : comme son nom l'indique...
- **RTL** : MoC dont l'écriture est similaire à celle d'une architecture en VHDL synthétisable, mais simplifiée (pas de listes de sensibilités explicites, gestion automatique et implicite de l'horloge et du reset). Il est possible de créer des registres (**namedReg**, renvoyant une lvalue dont l'arbre combinatoire se réduit à **TrTreeObject n**), de leur attribuer la valeur qu'il prendront suite à un reset (**defaultsTo**), et d'instancier des processus exprimés impérativement (**process**, dont l'écriture se conforme aux principes exposés à la section 2.2.2). Comme à la fois **namedVar** et **namedReg** renvoient des expressions dont l'arbre combinatoire équivaut à **TrTreeObject n**, c'est au moteur d'être capable de distinguer si un objet est une variable ou un registre à l'intérieur d'un processus. Ce n'est pas vraiment un problème difficile, mais il faudra le garder en tête lors de la conception des moteurs. La sémantique de **#=** sur un registre est similaire à celle du VHDL synchrone. Par exemple, écrire **<r #= r /+ / 1>** signifie que le registre **r** est incrémenté à chaque front d'horloge.

3.2.3 Moteur

3.2.3.1 Netlist

Nous avons choisi dans cet exemple d'implémenter un moteur sous la forme d'une monade d'état explorable dont l'état **SynthRTL** est une représentation de la netlist correspondant au design. L'affectation du design au moteur **ExpState SynthRTL** va exprimer une transformation correspondant à la phase d'*élaboration* du design. Une fois le design élaboré, il pourra être *simulé* ou utilisé pour *générer du code HDL intermédiaire*. La figure 3.5 détaille la déclaration de ce type et des instances associées. (La valeur de l'état initial à utiliser pour élaborer le design complet est **synthRTLEmpty**.)

Le type **SynthRTL** comporte comme information :

- **synthRTLNodes** : les noms et les indications de types des nœuds combinatoires explicites présents dans le circuit
- **synthRTLRegs** : les noms et les indications de types des registres présents dans le circuit
- **synthRTLNodeTrs** : les arbres combinatoires donnant la valeur de chaque nœud au *delta cycle* suivant

```

1  data SynthRTL = SynthRTL
2      { synthRTLNods :: [(String, Type)]
3        , synthRTLRegs :: [(String, Type)]
4        , synthRTLNodeTrs :: [TrTree]
5        , synthRTLRegTrs :: [TrTree]
6        , synthRTLRegDefaults :: [Integer] }
7
8  synthRTLAddNamedNode ty name synthRTL = -- ...
9  synthRTLAddNamedReg ty name synthRTL = -- ...
10 synthRTLAffect (TTr ty (TrTreeNode n)) (TTr ty' src) synthRTL = -- ...
11 synthRTLAffect (TTr ty (TrTreeObject n)) (TTr ty' src) synthRTL = -- ...
12 synthRTLSetRegDefault obj val synthRTL = -- ...
13
14 instance WithNodes (ExpState SynthRTL) where
15     namedNode ty name = do
16         n <- call (synthRTLAddNamedNode (typeStandardize ty) name)
17         return (return (TTr ty (TrTreeNode n)))
18     addTransferNodes x =
19         let fusion x' = call$ \synthRTL ->
20             let (rv, TrNodes _ nds) = x'
21                 getNode (TrNode n ty _) = (n, ty)
22                 getNodeTr (TrNode _ _ tr) = tr
23                 newNodes = map getNode nds
24                 newTrs = map getNodeTr nds
25                 nodes = synthRTLNods synthRTL
26                 trs = synthRTLNodeTrs synthRTL
27             in
28                 ( rv
29                   , synthRTL
30                     { synthRTLNods = nodes ++ newNodes
31                       , synthRTLNodeTrs = trs ++ newTrs
32                     } )
33         in do
34             startNode <- inquire (length.synthRTLNods)
35             namedNest "transferNodes" (TrNodes startNode []) fusion x
36
37 instance WithAffectation (ExpState SynthRTL) where
38     x #= y = do
39         x' <- addTransferNodes x
40         y' <- addTransferNodes y
41         modify (synthRTLAffect x' y')
42
43 instance RTL (ExpState SynthRTL) where
44     namedReg ty name = do
45         n <- call (synthRTLAddNamedReg (typeStandardize ty) name)
46         return (return (TTr ty (TrTreeObject n)))
47     defaultsTo x val = do
48         x' <- addTransferNodes x
49         val' <- addTransferNodes val
50         modify (synthRTLSetRegDefault x' val')
51     process cycles body = do
52         nodes <- inquire synthRTLNods
53         regs <- inquire synthRTLRegs
54         namedNest
55             "process"
56             (stdImpEmpty' (length nodes, length regs, 0))
57             (addSynthProcess nodes regs (fromIntegral cycles))
58             body
59
60 synthRTLEmpty :: SynthRTL
61 synthRTLEmpty = SynthRTL [] [] [] [] []
62 getSynthRTL :: State SynthRTL () -> SynthRTL
63 getSynthRTL em = let (_, rv) = runState em synthRTLEmpty in synthRTLSimplify rv
64 synthRTLSimplify synthRTL = -- ...

```

FIGURE 3.5 – Représentation d'une netlist

- `synthRTLRegTrs` : les arbres combinatoires donnant la valeur des entrées de bascules des registres, et déterminant donc leur valeurs au *cycle d'horloge* suivant
- `synthRTLRegDefaults` : les valeurs que prennent les registres suite à un reset synchrone

Les listes `synthRTLNodes etat` et `synthRTLNodeTrs etat` sont indicées de la même façon et ont une longueur identique (égale au nombre de nœuds). Il en va de même pour les listes `synthRTLRegs etat`, `synthRTLRegTrs etat` et `synthRTLRegDefaults` (longueur égale au nombre de registres).

Pour un maximum de clarté, nous avons implémenté la majorité des traitements dans des fonctions agissant sur l'état⁷ et commençant par `synthRTL`. L'implémentation des instances devient alors beaucoup plus simple, avec juste l'utilisation de `call`, `inquire`, `modify` et `namedNest` pour appeler les fonctions précédemment définies. Parmi les choses intéressantes visibles sur cet extrait de code, notons l'utilisation préalable de `addTransferNodes` sur tout transfert passé en argument à une fonction intervenant dans une construction monadique. Le rôle de `addTransferNodes` est d'ajouter aux listes `synthRTLNodes etat` et `synthRTLNodeTrs etat` l'information sur les nœuds combinatoires présents dans le transfert. Pour cela, on va simplement appliquer la transformation dictée par l'expression monadique du transfert en prenant pour état initial `TrNodes startNode []`, où `startNode` est le nombre de nœuds déjà présents dans la netlist. Ainsi, tous les nouveaux nœuds auront des identifiants uniques, n'entrant pas en conflit avec ceux déjà présents. Ceci fait, il suffira d'extraire l'information qui nous intéresse de l'objet `TrNodes` résultant, puis de la concaténer aux listes `synthRTLNodes etat` et `synthRTLNodeTrs etat`. Tout cela est réalisé par un simple appel de la fonction `namedNest` (ce qui est logique, puisque nous avons vu que `Transfer ty` peut être vu comme un MoC et `TrNodes` comme un de ses moteurs).

Un aspect qui peut peut-être intriguer est le «double `return`», aux lignes 17 et 46. C'est parfaitement normal : on place dans une monade (`ExpState SynthRTL`) une autre monade (`Transfer qqch`). On pourrait supposer que ce genre d'encapsulation peut conduire à des lourdeurs, mais en fait ce n'est pas le cas. L'utilisation d'un type monadique pour représenter les transferts s'avère être une bien meilleure solution du point de vue de l'utilisabilité, tant dans la spécification de designs que dans l'implémentation des moteurs, que l'emploi de types algébriques «simples» (non-monadiques) ou d'opérateurs avec arguments implicites, qui sont les deux méthodes que nous avons testées et qui se sont avérées largement déficientes sur bien des aspects que nous ne détaillerons pas ici. (Une présentation satisfaisante de ces méthodes et la discussion de leur impact sur l'écriture des designs serait longue et plutôt inintéressante au final, au vu des très nombreux désavantages qu'elles comportent. Nous aurons par contre l'opportunité de voir, dans les sections qui suivent, un certain nombre des avantages concrets qu'apportent les transferts monadiques explorables.)

Une utilisation importante de `namedNest` intervient dans la fonction `process`. Elle s'appuie sur l'utilisation du moteur `ExpState StdImp`, dont l'implémentation est assez similaire à celle de `ExpState SynthRTL`. Comme précédemment, l'affectation du design au moteur va correspondre à une étape d'*élaboration* et non d'*exécution*. Sans rentrer dans les détails des instances et fonctions associées, analysons juste sur l'information contenue dans l'état `StdImp` :

```

1 type StdImp = StdImpBlock
2
3 data StdImpAction
4     = StdImpAffectNode Int TrTree
5       | StdImpAffectVar Int TrTree
6       | StdImpWait
7       | StdImpWhile TrTree StdImpBlock
8       | StdImpIf [(TrTree, StdImpBlock)]
9     deriving Show
10
11 data StdImpBlock = StdImpBlock
12     { stdImpNodeStart :: Int
13     , stdImpVarStart  :: Int
14     , stdImpNodes    :: [(String, Type)]

```

7. et dont l'état est systématiquement le dernier argument, pour maximiser l'intérêt de l'évaluation partielle ; c'est une façon de programmer habituelle en Haskell [33]

```

15     , stdImpVars :: [(String, Type)]
16     , stdImpRevActions :: [StdImpAction] — actions en ordre inverse
17   } deriving Show

```

Deux concepts sont introduits : celui d'action et de bloc. Un bloc est une suite d'actions, chaque action correspondant à une affectation, une indication d'attente de front d'horloge montant (en cas de synthèse de machine à état implicite) ou une structure de contrôle. Chaque bloc contient en plus l'information caractérisant les nœuds et variables locales, à savoir leur noms, leur types et la façon dont sont organisés les indices les identifiant (cette donnée est représentée sous la forme des indices du premiers nœud et de la première variable locale, qui jouent le même rôle que le champ correspondant de `TrNodes`). Les actions d'un bloc sont ordonnées de la dernière à la première. Ainsi, l'ajout d'une action à la suite des précédentes (donc en tête de liste) se fait en temps constant. Comme zéro, un ou plusieurs ajouts d'actions peuvent survenir à chaque `bind` (`>>=`), il est en pratique beaucoup plus rentable d'inverser cette liste à la fin, une fois le design complètement évalué, que de rajouter des éléments en fin de liste au fur et à mesure.

Il ne reste plus qu'à définir `addSynthProcess`, utilisée à la ligne 57 du listing de la figure 3.5. Cette fonction va prendre en entrée les listes des nœuds et registres présents dans la netlist au moment de la création du processus, un entier (non-utilisé), et finalement la paire valeur retournée (`()`) / état final (`StdImp`) obtenue en évaluant le bloc impératif. Comme le nombre de registres n_{reg} existants est connu, `addSynthProcess` peut faire la différence entre les registres et variables en fonction de leur numéro identifiant (`[0, nreg - 1]` pour les registres, `[nreg, ∞[` pour les variables).

3.2.3.2 Simulation

Le fait que le design soit stocké sous la forme de netlist rend l'écriture d'un simulateur quasi triviale⁸ :

```

1 data SynthRTLEnv = SynthRTLEnv
2   { synthRTLEnvNodes :: Seq Integer
3     , synthRTLEnvRegs :: Seq Integer
4     } deriving (Show, Eq)
5
6 evaluate :: SynthRTLEnv -> TrTree -> Maybe Integer
7 simulate :: SynthRTL -> Int -> [SynthRTLEnv]
8 mkChronogram :: SynthRTL -> [SynthRTLEnv] -> String -> String -> Chronogram

```

On va représenter l'état du circuit à un delta cycle donné sous la forme d'une paire de séquences (`Data.Seq`) correspondant aux valeurs des nœuds combinatoires et des registres. Une séquence [38] est un type de données prédéfini ressemblant à une liste ordinaire du point de vue des opérations qu'il est possible de réaliser dessus, mais offrant de bien meilleures performances asymptotiques dans la plupart des cas. Ainsi, récupérer le i -ème élément d'une séquence de longueur n se fait en $O(\log(\min(i, n - i)))$, et non plus en $O(i)$. L'inconvénient des séquences est que leur évaluation n'est pas paresseuse. Cela n'a cependant aucune importance dans l'écriture du simulateur présenté ici.

La simulation se déroule de façon itérative, en suivant un algorithme très simple :

1. On évalue les valeurs de tous les nœuds à partir des valeurs qu'ils avaient au delta cycle précédant, ainsi que des valeurs des registres
2. Si les valeurs des nœuds sont stables (aucun changement depuis le delta cycle précédant), on passe au cycle d'horloge suivant, ce qui se traduit par l'évaluation des nouvelles valeurs de tous les registres

La fonction `evaluate` sert précisément à évaluer la valeur d'une chaîne de bits en sortie d'un arbre combinatoire, en se servant de l'information sur les valeurs courantes des nœuds et registres. Le type de retour est `Maybe Integer`, la fonction renvoyant `Nothing` dans le cas où la sortie n'est pas définie (par exemple lorsqu'elle n'est pas correctement câblée).

⁸. Notons que nous pourrions faire de l'émulation sur carte en lieu et place de simulation dans la boucle d'exploration.

`simulate` produit une liste infinie de `SynthRTLEnv`, permettant d'observer cycle après cycle les valeurs des nœuds stabilisées et des registres⁹. L'argument entier que prend `simulate` est le nombre maximum de delta cycles consécutifs avant de passer au cycle suivant. Si ce nombre est atteint, on peut penser qu'il reste des instabilités dans le design de l'utilisateur, et `simulate` se termine en produisant une erreur.

Enfin, `mkChronogram` permet de mettre les résultats de simulation sous la forme d'un chronogramme, qu'il sera ensuite possible d'écrire dans un fichier VCD.

3.2.3.3 Évaluation de la complexité

La représentation du design sous forme de netlist permet de se faire une idée de sa taille et de la longueur du chemin critique. Une façon simple d'évaluer ces dernières est de poser `treeEvalSize` et `treeEvalCrit` (évaluant la taille et le chemin critique d'un arbre combinatoire), et de les sommer sur tout le design. Au final, on obtient ce genre de code :

```

1 class WithTreeEvals engine where
2     treeEvalSize :: engine -> TrTree -> Integer
3     treeEvalCrit :: engine -> TrTree -> Integer
4     evalSize :: engine -> Integer
5     evalCrit :: engine -> Integer
6
7 instance WithTreeEvals SynthRTL where
8     treeEvalSize TrTreeNotSet = 0
9     treeEvalSize (TrTreeObject _) = 0
10    — ...
11    treeEvalSize (TrTreeTerOp sz TerOpMux arg1 arg2 arg3)
12        = 4*sz + treeEvalSize arg1 + treeEvalSize arg2 + treeEvalSize arg3
13
14    treeEvalCrit synthRTL TrTreeNotSet = 0
15    treeEvalCrit synthRTL (TrTreeObject _) = 0
16    — ...
17    treeEvalCrit synthRTL (TrTreeTerOp sz TerOpMux arg1 arg2 arg3) =
18        1 + maximum (map (treeEvalCrit synthRTL) [arg1, arg2, arg3])
19
20    evalSize synthRTL =
21        foldr (+) 0 (map treeEvalSize (synthRTLNodeTrs synthRTL))
22        + foldr (+) 0 (map treeEvalSize (synthRTLRegTrs synthRTL))
23        + 3 * length (synthRTLRegTrs synthRTL) — place occupée par les registres
24
25    evalCrit synthRTL = max
26        (maximum (map (treeEvalCrit synthRTL) (synthRTLNodeTrs synthRTL)))
27        (maximum (map (treeEvalCrit synthRTL) (synthRTLRegTrs synthRTL)))

```

Les fonctions `treeEvalSize` et `treeEvalCrit` de l'instance `WithTreeEvals SynthRTL` ont été ici écrites de façon plus ou moins arbitraire; l'idéal aurait été de mesurer le coût de chaque opérateur dans la technologie cible, et de les ajuster en conséquence. Dans tous les cas, ces fonctions ne seront que des approximations, ne tenant pas compte de problématiques de placement et de routage, par exemple¹⁰. Néanmoins, elles permettent de réaliser des choses assez intéressantes en conjonction avec les possibilités d'exploration architecturale de l'approche. Voir section 4.1.

3.2.4 Modèles de caches

3.2.4.1 Interface mémoire

Il nous faudra définir une interface niveau RTL pour communiquer avec une mémoire. On procède comme on l'a fait dans l'exemple de la section 3.1, en définissant une paire de types de données, correspondant aux paramètres d'une mémoire, et aux objets créés lors de son instantiation :

9. Il aurait été possible de garder tous les deltas cycle individuels. Arbitrairement, nous avons au lieu de cela choisi de ne garder que les états stables, survenant juste avant de passer au cycle d'horloge suivant.

10. à moins, bien sûr, de faire en sorte que le moteur lui-même sache gérer ce genre de concept

```

1 data Memory = Memory
2     { memoryName :: String
3       , memoryAddrLength :: Int — en bits
4       , memoryBusWidth :: Int — en bits
5       , memoryLatency :: Int — en cycles
6     }
7
8 data MemoryId = MemoryId
9     { memoryParams :: Memory
10      , memoryBackend :: Maybe MemoryId
11      , memoryNAccesses :: Transfer Type — 64 bits; compteur d'accès
12      , memoryNWordsRead :: Transfer Type — 64 bits; compteur de mots lus
13      , memoryRead :: Transfer Type — 1 bit
14      , memoryReadBase :: Transfer Type
15      , memoryReadLength :: Transfer Type
16      , memoryReadReady :: Transfer Type — 1 bit; lecture prête?
17      , memoryReadOk :: Transfer Type — 1 bit; readData et readAddr sont valides?
18      , memoryReadAddr :: Transfer Type — addrLength bits; adresse courante
19    }

```

Le type `Memory` se comprend assez facilement de lui même. Il contient le nom d'une mémoire (pour accroître la lisibilité des résultats produits), les tailles des bus d'adresses et de données associés, ainsi que la latence, à supposer que cette dernière soit constante ; dans le cas contraire, ce champ doit être ignoré. `MemoryId`, quant à lui, regroupe les paramètres et les ports des composants permettant des lectures en *burst*. En plus des paramètres mentionnés plus haut, il contient :

- Une référence au composant sous-jacent de la hiérarchie mémoire, à supposer qu'il existe
- Des registres comptant le nombre d'accès en burst effectués, et le nombre de mots lus
- Les ports d'entrée `memoryRead` (demande de lecture), `memoryReadBase` (adresse du début du *burst*), `memoryReadLength` (longueur du *burst* en mots)
- Les ports de sortie `memoryReadReady` (composant en attente de requête), `memoryReadAddr` (adresse lue au cycle courant), `memoryReadOk` (indique que `memoryAddr` a une valeur valable au cycle courant)
- On aurait ajouté le port de sortie `memoryReadData` (donnée lue au cycle courant, valide si `memoryOk` vaut 1) à cette liste si on avait géré le contenu effectif des mémoires (inutile dans notre exemple)
- On aurait également pu ajouter les ports permettant les opérations d'écriture (inutile dans notre exemple)

L'écriture d'un modèle de mémoire externe à latence élevée se conformant à l'interface définie par `MemoryId` est assez simple (c.f. listing de la figure 3.6), et permet de se convaincre si besoin était de la ressemblance entre le MoC RTL et le VHDL. Nous avons utilisé le préprocesseur C (comme le permet GHC) pour définir des alias pour `iff`, `elsiff`, etc., pour accentuer ce au maximum parallèle syntaxique, bien que dans la pratique il soit possible d'utiliser les primitives du MoC directement, et ce sans lourdeurs syntaxiques particulières. Le renvoi de l'interface par le biais d'un `return` est pratiquement la seule pratique pouvant étonner un designer VHDL dans cet exemple.

3.2.4.2 Cache unidimensionnel

Comme dans l'exemple précédant, nous allons définir tout d'abord un cache set-associatif unidimensionnel. Voici le modèle que l'on propose :

```

1 data CacheAssociativity
2     = FullyAssociative
3     | DirectMapped
4     | SetAssociative Integer — Slots par set ("n-associatif")
5
6 data Cache = Cache
7     { cacheName :: String
8       , cacheAssociativity :: CacheAssociativity
9       , cacheLineSize :: Integer — en mots de busWidth bits

```

3.2. MODÉLISATION AU NIVEAU RTL

```

1 memory :: RTL em => Memory -> em MemoryId
2
3 memory params = do
4     let name = memoryName params
5         let addrLength = memoryAddrLength params
6         let busWidth = memoryBusWidth params
7         let latency = memoryLatency params
8
9         nAccesses <- reg' (uint 64)
10        nWordsRead <- reg' (uint 64)
11        read <- node' bool
12        readBase <- node' (uint addrLength)
13        readLength <- node' (uint addrLength)
14        readReady <- node' bool
15        readOk <- node' bool
16        readAddr <- node' (uint addrLength)
17
18    PROCESS do
19        ttw <- var' (uint 32) — time to wait
20        reqBase <- var' (uint addrLength)
21        reqLength <- var' (uint addrLength)
22        curBase <- var' (uint addrLength)
23        curLength <- var' (uint addrLength)
24
25        readReady #/= reqLength /==/ 0
26        readOk #/= 0
27        readAddr #/= 0
28        nAccesses #/= nAccesses
29        nWordsRead #/= nWordsRead
30
31        IF (readReady ./ read) do
32            reqBase #/= readBase
33            reqLength #/= readLength
34
35        IF (ttw) do
36            ttw #/= ttw /-/ 1
37        ELSE do
38            IF (curLength /==/ 0) do
39                IF (reqLength) do
40                    curBase #/= reqBase
41                    curLength #/= reqLength
42                    reqBase #/= 0
43                    reqLength #/= 0
44                    ttw #/= fromIntegral latency
45                    nAccesses #/= nAccesses /+/ 1
46                ELSE do
47                    readOk #/= 1
48                    readAddr #/= curBase
49                    curLength #/= curLength /-/ 1
50                    curBase #/= curBase /+/ 1
51                    nWordsRead #/= nWordsRead /+/ 1
52
53    return MemoryId
54        { memoryParams = params
55          , memoryBackend = Nothing
56          , memoryNAccesses = nAccesses
57          , memoryNWordsRead = nWordsRead
58          , memoryRead = read
59          , memoryReadBase = readBase
60          , memoryReadLength = readLength
61          , memoryReadReady = readReady
62          , memoryReadOk = readOk
63          , memoryReadAddr = readAddr
64        }

```

FIGURE 3.6 – Modèle RTL de mémoire à latence élevée

```

10     , cacheNSlots :: Integer
11   }
12
13 data CacheId = CacheId
14   { cacheParams :: Cache
15     , cacheBackend :: Maybe MemoryId
16     , cacheNAccesses :: Transfer Type — 64 bits; compteur d'accès
17     , cacheRead :: Transfer Type — 1 bit
18     , cacheReadAddr :: Transfer Type — mot de addrLength bits
19     , cacheReadOk :: Transfer Type — 1 bit; readData est valide?
20     —, cacheReadData :: Transfer Type — mot de busWidth bits
21   }
22
23 — Args: paramètres, back-end
24 cache :: RTL em => Cache -> MemoryId -> em CacheId
25 cache params memory = ...
26
27 — Arg: taille (en mots de busWidth bits)
28 autocacheParams :: RTL em => Integer -> em (String -> Cache)
29 autocacheParams size = do
30   lineSize <- oneOf (takeWhile (\x -> x <= size) [16, 32, 64])
31   let nSlots = size `quot` lineSize
32       associativity <- oneOf (map (\k -> SetAssociative (2^k)) [0..(intlog nSlots)])
33   let params name = Cache
34       { cacheName = name
35         , cacheAssociativity = associativity
36         , cacheLineSize = lineSize
37         , cacheNSlots = nSlots
38       }
39   return params
40
41 autocache name size memory = do
42   params <- autocacheParams size
43   cache (params name) memory

```

On retrouve la classique dualité `Cache/CacheId`, correspondant à l'information liée aux paramètres et à l'instance d'un cache, respectivement. Les paramètres incluent le nom d'un cache, son associativité, la taille d'une ligne de cache et le nombre de slots. Le type `CacheId` contient les paramètres mentionnés précédemment, la mémoire à latence élevée associée au cache, un registre contenant le nombre d'accès effectués, ainsi que les ports du cache : demande de lecture, adresse à lire, signal de validité de la donnée lue. On aurait ajouté un signal correspondant à la donnée lue elle-même si cela avait importé.

La fonction `cache` permet d'instancier un cache à partir d'un jeu de paramètres et de la mémoire à utiliser comme back-end. Le détail de son implémentation n'apparaît pas ici.

On définit également `autocache` et `autocacheParams`. La fonction `autocacheParams` construit un jeu de paramètres «raisonnables» pour un cache. En fonction de la taille totale demandée, la taille de la ligne de cache et l'associativité sont choisies parmi plusieurs alternatives. Il est possible de se dispenser de l'appel explicite à `autocacheParams` en utilisant directement la fonction `autocache`, qui se comporte un peu comme une boîte noire, permettant à un utilisateur d'instancier un cache en ne connaissant que la taille qu'il souhaite qu'il ait. Les paramètres exacts sont alors déterminés pendant l'étape d'exploration. Il est à noter qu'il peut être souhaitable pour un auteur de bibliothèque de composants d'encourager tout designer l'utilisant à employer ce genre de fonctions. En effet il est alors possible au fur et à mesure que la bibliothèque évolue de rajouter de nouveaux paramètres et implémentations alternatives dont l'utilisateur de la bibliothèque pourra profiter sans avoir à modifier son design.

3.2.4.3 Cache multidimensionnel

Dans l'absolu, la meilleure façon de procéder pour effectuer des requêtes multidimensionnelles est de définir un nouvel composant avec une nouvelle interface, quitte à simplement y encapsuler le cache set-associatif que nous venons de définir ; nous avons déjà procédé de façon similaire à la

section 3.1.4.2. On a alors le genre de bénéfices déjà évoqué à la section précédente : il est simple de modifier le composant pour lui conférer un comportement potentiellement plus adéquat.

Comme ce principe a déjà été illustré, nous allons simplement nous contenter d'aller à l'essentiel, en implémentant juste ce qu'il faut pour permettre l'adressage de tableaux multidimensionnels dans une mémoire adressée linéairement. Le listing correspondant se trouve sur la figure 3.7. On définit les fonctions `addrNDLinear`, `addrNDMortonLSBFirst` et `addrNDMortonMSBFirst`, se comportant exactement comme les fonctions homonymes définies à la section 3.1.4.2 (et illustrées par la figure 3.1). Ces fonctions prennent toujours en argument la liste des dimensions du tableau ainsi que les coordonnées de l'élément auquel on cherche à accéder, et renvoient l'offset de cet élément par rapport au début du tableau. Cette fois, les coordonnées sont passées sous forme de transferts pour être compatibles avec le MoC RTL. Techniquement, les fonctions correspondant à l'ordre de Morton sont implémentées avec l'aide de la fonction `reorderBits`, qui réorganise les bits des coordonnées dans un ordre déterminé.

On adjoint à ces différentes politiques d'adressage la possibilité de permuter les coordonnées. La fonction `permut` fait la permutation elle-même, tandis que `addrNDLinearP`, `addrNDMortonLSBFirstP` et `addrNDMortonMSBFirstP` sont les équivalents des fonctions précédentes mais permutant leurs coordonnées en fonction de la valeur de leur premier argument. Pour un tableau tridimensionnel, il y a 3! (donc 6) permutations possibles.

3.2.5 Design

On modélise au niveau RTL un réseau d'unités de reconstruction tomographique dont la structure est illustrée par la figure 3.8. Chaque unité est raccordée à son propre cache. Les caches, eux sont connectés à un bloc arbitrant leurs accès à la SDRAM.

Le bloc arbitrant les accès des caches à la mémoire externe est instancié par la fonction `multicache`, de signature RTL `em => Integer -> MemoryId -> em [MemoryId]`, qui, à partir du nombre de caches à raccorder et de l'interface de la SDRAM, renvoie une liste d'interfaces similaire à celle d'une mémoire externe ; c'est sur ces interfaces que vont venir se brancher les caches. De petites mémoires on-chip permettent de temporairement stocker les zones demandés par chaque cache. Le bloc a la possibilité de fusionner les bursts se recouvrant pour gagner à la fois en débit et en bande passante. Ce comportement est facultatif, et peut être activé ou désactivé durant l'étape exploration architecturale. Si la fonction de coût à minimiser tient compte de la complexité matérielle, il peut en effet être avantageux de ne pas incorporer la logique permettant la fusion des requêtes, par exemple si l'on constate à la simulation que peu de requêtes se recouvrent et que le gain en devient négligeable.

L'unité de reconstruction tomographique est instanciée à partir d'un jeu de paramètres qui lui est propre et de l'interface du cache qui lui est associée. Elle renvoie un signal indiquant si la tâche qui lui a été demandé a été ou non réalisée. Une unité de reconstruction fait la même chose que le design de la fonction 3.1.5 ; une fois les `psiMax` accès effectués, le signal renvoyé prend la valeur 1. Nous définissons `Tomo` le type `Tomo`, regroupant tous les paramètres de l'unité de reconstruction :

```

1 data Tomo = Tomo
2   { tomoName :: String
3     , tomoSinogramSize :: (Integer, Integer, Integer)
4     , tomoAddressing :: [Integer] -> [Transfer Type] -> Transfer Type
5     , tomoR :: (Float, Float, Float)
6     , tomoD :: (Float, Float, Float)
7     , tomoDelta :: Float
8     , tomoBlockSize :: (Integer, Integer, Integer) }
```

Les champs de ce type correspondent au nom, à la taille du sinogramme, à sa fonction d'adressage (comme décrite précédemment), aux paramètres déterminant la position des éléments du sinogramme auxquels on va accéder durant la simulation, et enfin la taille du bloc de voxels qui va être reconstruit simultanément par l'unité. Tous les paramètres sont similaires à ceux que nous avons déjà pu voir à la section 3.1.5. Le listing de la figure 3.9 donne l'implémentation complète

```

1  -----
2  -- Addressage --
3  -----
4
5  addrNDLinear :: [Integer] -> [Transfer Type] -> Transfer Type
6
7  addrNDLinear [] [] = 0
8  addrNDLinear [] _ = error "addrNDLinear: too much coordinates"
9  addrNDLinear _ [] = error "addrNDLinear: not enough coordinates"
10 addrNDLinear (dim:dims) (x:xs) = x /+ ((fromIntegral dim) ./ addrNDLinear dims xs)
11
12 reorderBits :: Int -> [Transfer Type] -> [Int] -> Transfer Type
13 reorderBits pos trs [] = fromIntegral 0
14 reorderBits pos trs (n:ns) =
15     (((trs!!n)/%(fromIntegral 2)) /<</ (fromIntegral pos))
16     /+ (reorderBits (pos + 1) (mapNth pos (/>>/1) trs) ns)
17
18 addrNDMortonLSBFirst dims xs =
19     let aux [0, 0, 0] = []
20         aux [x, 0, 0] = [0]++(aux [x'quot'2, 0, 0])
21         aux [0, y, 0] = [1]++(aux [0, y'quot'2, 0])
22         aux [0, 0, z] = [2]++(aux [0, 0, z'quot'2])
23         aux [x, y, 0] = [0, 1]++(aux [x'quot'2, y'quot'2, 0])
24         aux [x, 0, z] = [0, 2]++(aux [x'quot'2, 0, z'quot'2])
25         aux [0, y, z] = [1, 2]++(aux [0, y'quot'2, z'quot'2])
26         aux coords = [0, 1, 2]++(aux (map ('quot'2) coords))
27     in reorderBits 0 xs (aux dims)
28
29 addrNDMortonMSBFirst dims xs =
30     let aux [0, 0, 0] = []
31         aux [x, y, z] | x > y && x > z = [0]++(aux [x'quot'2, y, z])
32         aux [x, y, z] | y > x && y > z = [1]++(aux [x, y'quot'2, z])
33         aux [x, y, z] | z > x && z > y = [2]++(aux [x, y, z'quot'2])
34         aux [x, y, z] | x == y && y > z = [0, 1]++(aux [x'quot'2, y'quot'2, z])
35         aux [x, y, z] | x == z && z > y = [0, 2]++(aux [x'quot'2, y, z'quot'2])
36         aux [x, y, z] | y == z && z > x = [1, 2]++(aux [x, y'quot'2, z'quot'2])
37         aux coords = [0, 1, 2]++(aux (map ('quot'2) coords))
38     in reorderBits 0 xs (aux dims)
39
40 -----
41 -- Permutation des coordonnées --
42 -----
43
44 permut :: Integer -> ([Integer], [Transfer Type]) -> ([Integer], [Transfer Type])
45 permut 0 ([a, b, c], [x, y, z]) = ([a, b, c], [x, y, z])
46 permut 1 ([a, b, c], [x, y, z]) = ([a, c, b], [x, z, y])
47 permut 2 ([a, b, c], [x, y, z]) = ([b, a, c], [y, x, z])
48 permut 3 ([a, b, c], [x, y, z]) = ([c, a, b], [z, x, y])
49 permut 4 ([a, b, c], [x, y, z]) = ([b, c, a], [y, z, x])
50 permut 5 ([a, b, c], [x, y, z]) = ([c, b, a], [z, y, x])
51
52 -----
53 -- Addressage avec permutation --
54 -----
55
56 addrWithPermut f p dims xs = let (dims', xs') = permut p (dims, xs) in f dims' xs'
57
58 addrNDLinearP p dims xs = addrWithPermut addrNDLinear p dims xs
59 addrNDMortonLSBFirstP p dims xs = addrWithPermut addrNDMortonLSBFirst p dims xs
60 addrNDMortonMSBFirstP p dims xs = addrWithPermut addrNDMortonMSBFirst p dims xs

```

FIGURE 3.7 – Adressage de tableaux multidimensionnels

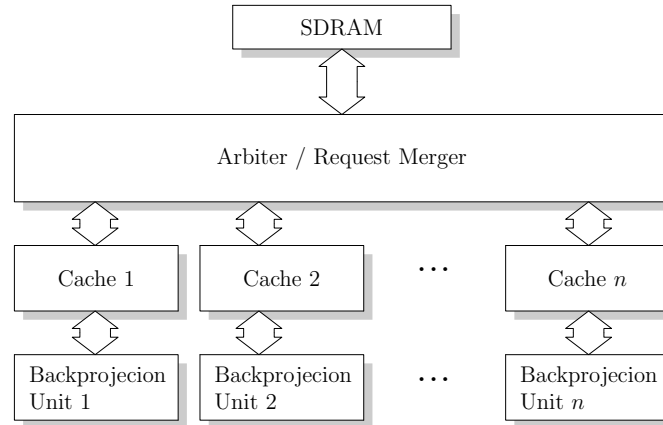


FIGURE 3.8 – Schéma global du réseau d'unités de reconstruction tomographique

de l'unité de reconstruction (qu'il peut être intéressant de comparer avec celui donné à la section 3.1.5, justement).

Le design complet est donné quant à lui par le listing de la figure 3.10. On y instancie les blocs décrits précédemment, à savoir une mémoire SDRAM externe avec 5 cycles de latence, une unité d'arbitrage, et les unités de reconstruction tomographique et leurs caches associés. Le paramétrage de ces dernières est fait de sorte à ce que les éléments auxquels on accède soient à peu près les mêmes que ceux que nous avons vu dans notre exemple sans horloge. Le nombre d'unités de reconstruction instancié ainsi que les tailles des blocs de voxels reconstruits par chaque unité sont choisis de sorte à ce que le nombre de voxels reconstruit par l'ensemble du design soit toujours le même, en l'occurrence 32 ($4 \times 4 \times 2$, voir ligne 11 du listing).

3.2.6 Résultats

Notre espace de design comporte 13248 combinaisons de paramètres valables. Ce n'est pas énorme, mais déjà pratiquement trop pour une exploration exhaustive. En effet, avec un temps de simulation moyen de 1 minute et 48 secondes pour évaluer un design paramétré¹¹, il nous aurait fallu environ 16 jours et demi pour tester toutes les combinaisons.

Nous avons procédé à la phase d'exploration en utilisant deux méthodes différentes :

- un recuit simulé sur 250 itérations
- un parcours aléatoire sur 250 itérations

Nous n'avons pas utilisé l'algorithme génétique, car celui-ci consommait nettement plus de mémoire vive pour des tailles de population acceptables. Le recuit simulé et le parcours aléatoires étaient suffisamment efficaces pour que la machine effectuant la simulation n'ai pas besoin de *swapper* le contenu de la mémoire vive avec le disque dur.

Le résultat est donné sur la figure 3.11. La fonction à maximiser ici correspond au nombre moyen de données lues par cycle pour l'intégrité du système. Nous avons considéré ici que le sinogramme était constitué d'éléments dont la taille est égale à la largeur du bus mémoire, c'est à dire 64 bits¹². Le maximum obtenu par les deux méthodes d'exploration est de 1,152 éléments par cycle. Ce maximum est obtenu au bout de 20 cycles pour le recuit simulé, et 172 cycles pour

11. La machine de test était un PC équipé d'un Intel Core 2 Duo à 1,8 Ghz, mais notre code n'étant pas multi-threadé, seul un cœur était utilisé ; nous n'avons pas implémenté de versions parallèles des algorithmes d'exploration, bien que certains d'entre eux (parcours aléatoire, programmation génétique) s'y prêtent assez bien.

12. Il s'agit d'une dynamique nettement supérieure à celle dont sont capables les scanners IRM actuels. L'intérêt de cette hypothèse réside dans l'interprétation facilitée des résultats. En effet, si on suppose que les données lues dans la SDRAM ne sont jamais réutilisées (à la fois dans le temps et entre unités de reconstruction) et que la latence des accès est complètement recouverte, le nombre d'éléments lus par cycle est exactement égal à un. Cela offre un point de comparaison intéressant.

```

1  tomo :: RTL em => Tomo -> CacheId -> em (Transfer Type)
2  tomo params cache = do
3      let name = tomoName params
4          let Just backend = cacheBackend cache
5              let memory = memoryParams backend
6                  let (psiMax, uMax, vMax) = tomoSinogramSize params
7                      let dims = [psiMax, uMax, vMax]
8                          let r@(xo, yo, zo) = tomoR params
9                              let (dx, dy, dz) = tomoD params
10                                  let (bx, by, bz) = tomoBlockSize params
11                                      let delta = tomoDelta params
12                                          let u psi r@(x, y, z) =
13                                              let rv = (x * cos psi + y * sin psi + 1.0)
14                                                  in fromIntegral (floor (rv * 0.5 * fromInteger uMax))
15      let v psi delta r@(x, y, z) =
16          let ra = 1.0
17              a = x * sin psi - y * cos psi
18              rv = delta / (2.0 * ra) * a + z + 2.0
19              in fromIntegral (floor (rv * 0.25 * fromInteger vMax))
20      let psiStep = 2.0 * 3.14159 / (fromIntegral psiMax)
21      let getAddr' it r =
22          let psi = (fromIntegral it) * psiStep
23              x = fromIntegral it
24              in (tomoAddressing params) dims [u psi r, v psi delta r, x]
25      let getAddr it r =
26          let aux :: Integer -> Integer -> Transfer Type
27              aux (-1) it' = getAddr' it' r
28              aux n it' = let (n', limit) = (n - 1, (it' + (2^n)))
29                          in (it' /</ fromIntegral limit)/?/
30                          (aux n' it', aux n' limit)
31              in aux ((intlog psiMax)+2) 0
32
33      nFetches <- reg (uint 32)
34      done <- node' bool
35      currentPsi <- reg' (rint (0, psiMax))
36      i <- reg' (rint (0, (bx*by*bz - 1)))
37      addr <- node' (uint (memoryAddrLength memory))
38
39      PROCESS do
40          done #= 0
41          i #= i
42          currentPsi #= currentPsi
43          cacheReadAddr cache #= addr
44          cacheRead cache #= 1
45          addr #= 0
46          nFetches #= nFetches
47          IF (currentPsi /</ (fromIntegral psiMax)) do
48              IF (cacheReadOk cache) do
49                  nFetches #= nFetches /+ 1
50                  IF (i /==/ (fromIntegral (bx*by*bz - 1))) do
51                      i #= 0; currentPsi #= currentPsi /+ 1
52                  ELSE do i #= i /+ 1
53              [0..(bx*by*bz - 1)] 'for' \j -> do
54                  let xi = j 'rem' bx
55                      let yi = (j 'quot' bx) 'rem' by
56                          let zi = j 'quot' (bx*by)
57                              let x = xo + dx * fromIntegral xi
58                                  let y = yo + dy * fromIntegral yi
59                                  let z = zo + dz * fromIntegral zi
60                                  let r = (x, y, z)
61                                  IF (i /==/ fromIntegral j) do
62                                      addr #= getAddr currentPsi r
63              ELSE do
64                  done #= 1; cacheRead cache #= 0
65      return done

```

FIGURE 3.9 – Unité de reconstruction tomographique modélisée au niveau RTL

```

1  design = do
2      addrLength <- return 32
3      busWidth <- return 64
4      cacheSize <- oneOf [16, 32, 64, 128, 256, 512]
5      d@(dX, dY, dZ) <- return (2.0 / 64.0, 2.0 / 64.0, 2.0 / 64.0)
6      delta <- return 1.0
7      bsX <- oneOf [2, 4]
8      bsY <- oneOf [2, 4]
9      bsZ <- oneOf [1, 2]
10     blockSize@(bsX, bsY, bsZ) <- return (bsX, bsY, bsZ)
11     let (nX, nY, nZ) = (4 'quot' bsX, 4 'quot' bsY, 2 'quot' bsZ)
12         nUnits <- return (nX*nY*nZ)
13         perm <- oneOf [0..5]
14         addr <- oneOf
15             [ addrNDLinearP perm
16               , addrNDMortonMSBFirstP perm
17               , addrNDMortonLSBFirstP perm ]
18
19     cycle <- reg (uint 64)
20
21     let sdramParams = Memory
22         { memoryName = "sdram"
23           , memoryAddrLength = addrLength
24           , memoryBusWidth = busWidth
25           , memoryLatency = 5 }
26
27     sdram <- memory sdramParams
28     mergerIfces <- multicache nUnits sdram
29
30     let offset bs n = (fromIntegral bs) * (fromIntegral n)
31         let tomoParam n = Tomo
32             { tomoName = "tomo"++(show n)
33               , tomoSinogramSize = (256, 256, 32)
34               , tomoAddressing = addr
35               , tomoR =
36                 ( 0.25 + dX * offset bsX (n 'rem' nX)
37                   , 0.30 + dY * offset bsY ((n 'quot' nX) 'rem' nY)
38                   , 0.42 + dZ * offset bsZ (n 'quot' (nX*nY)) )
39               , tomoD = d
40               , tomoDelta = delta
41               , tomoBlockSize = blockSize
42             }
43
44     let cacheNames = map (("cache"++) . show) [1..nUnits]
45         autocacheParams' <- autocacheParams cacheSize
46         let autocache (name, ifce) = cache (autocacheParams' name) ifce
47             cacheIfces <- sequence (map autocache (zip cacheNames mergerIfces))
48
49     let tomoParams = map tomoParam [0..(nUnits-1)]
50         ready <- sequence (zipWith (\p i -> tomo p i) tomoParams cacheIfces)
51
52     done <- reg bool
53     done #= foldr (/./) 1 ready
54     cycle #= cycle /+/ 1

```

FIGURE 3.10 – Réseau d'unités de reconstruction tomographiques modélisé au niveau RTL

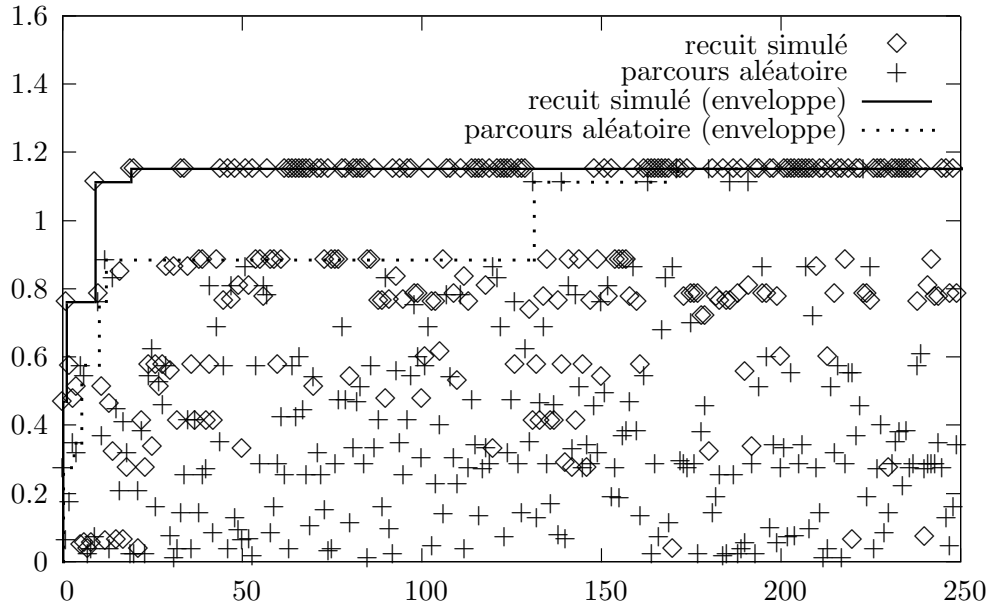


FIGURE 3.11 – Nombre moyen d’éléments lus par cycle en fonction de l’itération

le parcours aléatoire. Il est difficile de dire s’il s’agit d’un maximum global sans avoir testé toutes les combinaisons de paramètres, mais le fait que ce score ne soit pas battu au bout de 250 cycles de recuit simulé peut néanmoins inciter à prudemment avancer que cela peut tout à fait être le cas.

3.2.7 Conclusions

Cette deuxième méthode, dans laquelle l’évaluation de l’expression monadique représentant le design correspond à l’étape d’exploration, est clairement plus souple que la précédente, qui se limitait à la simulation. Son inconvénient réside dans l’impossibilité d’utiliser dans le cas général du code issu de bibliothèques existantes pour implémenter un modèle fonctionnel du design. Ses nombreux avantages résident dans la possibilité d’accéder facilement à la représentation intermédiaire du design, dans le but de générer du code intermédiaire, d’évaluer heuristiquement les performances a priori via des modèles appropriés, de réaliser des conversions entre modèles de calcul, etc. Nous pensons que passer par une étape d’élaboration comme nous venons de le faire est la meilleure façon de tirer un maximum d’intérêt de l’approche proposée, et est à privilégier dans la majorité des cas par rapport à la simulation directe vue à la section 3.1.

Par ailleurs, notre étude sur un exemple d’architecture raisonnablement complexe nous a également permis de nous assurer que l’approche était utilisable dans des cas réels avec des résultats satisfaisants, tant au niveau de l’effort à fournir par le programmeur pour exprimer son design que dans les performances et l’issue de la phase d’exploration.

Chapitre 4

Pistes pour la résolution de divers problèmes architecturaux

Ce chapitre souligne l'intérêt de l'approche proposée pour la résolution de divers problèmes que l'on rencontre en conception architecturale. Il ne s'agit pas d'études de cas complètes comme on a pu en voir dans le chapitre précédent, mais davantage de discussion des moyens à mettre en œuvre pour s'attaquer à un problème donné. Les propos avancés seront souvent étayés par des exemples.

4.1 Découpage des pipelines en étages

4.1.1 Contexte et principe

Le découpage de longs chemins combinatoires en pipelines est une des optimisations les plus courantes dans le domaine de la conception matérielle. Il s'agit d'un excellent moyen de raccourcir le chemin critique, et donc d'accroître la fréquence maximale de fonctionnement d'un circuit électronique.

Le partitionnement d'un pipeline est néanmoins une tâche relativement compliquée pour plusieurs raisons. Pour obtenir un gain en fréquence maximal, on aura en général tout intérêt à équilibrer les étages du pipeline. Cependant, il ne faut pas oublier qu'un pipeline reste un compromis fréquence/surface, et qu'un découpage en étages au temps de propagation parfaitement identiques risque potentiellement d'être sous-optimal en terme de quantité de registres présents entre les étages. Par ailleurs, il se peut qu'une autre partie du circuit devienne critique une fois le pipeline raisonnablement équilibré. Dans ce cas là, il n'apporte rien de plus d'essayer d'homogénéiser les étages davantage, et il faut plutôt essayer de faire en sorte de placer les frontières entre les étages là où elles demanderont le moins de registres.

Enfin, il faut garder à l'esprit la technologie cible visée lorsque l'on procède au découpage en étages. Lorsqu'il s'agit d'ASICs *full custom*, on garde éventuellement la possibilité d'ajuster dans une certaine mesure la longueur d'un étage via le choix du rapport W/L des transistors le constituant. Cette façon de faire n'est plus du tout envisageable lorsque l'on cible les FPGA. Par ailleurs, pour les pipelines aux étages vraiment courts, le type des cellules entrant dans la composition du FPGA cible peut également s'avérer important.

4.1.2 Extensions au modèle de calcul

4.1.2.1 Découpage automatique déterministe

On repart du MoC RTL défini précédemment. Dans le principe, il nous «suffira» de définir une fonction prenant en argument le circuit combinatoire à découper en pipeline et le nombre d'étages souhaités. Le type de retour le plus pratique pour cette fonction est sans doute RTL `em => em`

4.1. DÉCOUPAGE DES PIPELINES EN ÉTAGES

a, où **a** est de type de donné contenu dans la monade passée en argument et correspondant au circuit combinatoire. Ce qui va être retourné correspondra à l'interface du pipeline, contenant les signaux des données en entrée et en sortie, ainsi que les signaux de contrôle. Pour un maximum de clarté, on va procéder en deux étapes :

- définir une fonction découpant un circuit combinatoire représenté par un type de données concret dérivé via un moteur connu (en l'occurrence, ce type sera `TrNodes` dans notre exemple) — nous l'appellerons `partition`
- définir une fonction encapsulant la précédente via un appel à `namedNest`, la fonction d'inclusion entre MoC de la classe `Explorable` ; c'est cette nouvelle fonction qui sera utilisée au final par le designer — nous l'appellerons simplement `pipeline`

Afin de pouvoir proposer un partitionnement sensé du circuit combinatoire à pipeliner, il nous faudra disposer d'un modèle capable d'estimer la latence d'un chemin dans ce circuit. Ce modèle devra être compatible avec le moteur utilisé lors de l'appel à `namedNest` contenu dans `pipeline`. Supposons que ce moteur soit `Transfer'` (c'est à dire `ExpState TrNodes`) ; l'état correspondant est en somme une liste de `TrTree` annotés par leur nom et l'information sur leur type. On peut alors demander à l'utilisateur de fournir une fonction d'estimation de longueur de chemin combinatoire ayant pour signature `TrNodes -> TrTree -> Integer`. Une telle fonction ne tient pas compte des effets du placement-routage (puisque celui-ci n'est pas géré par le moteur), mais à cette limitation près peut évaluer assez finement la longueur du chemin critique en fonction de l'architecture visée. Dans la plupart des cas, on peut supposer qu'une version générique de cette fonction (ne tenant compte que des opérations réalisées et des tailles des opérands, sans faire d'hypothèses sur la technologie sous-jacente) serait suffisamment efficace pour produire un pipeline aux étages de longueurs raisonnablement équilibrées. Nous construirons une instance de la classe `WithTreeEvals` (décrite à la section 3.2.3.3) pour le type `TrNodes`, de sorte à ce que son implémentation de la fonction `treeEvalCrit` satisfasse à ce critère de généralité. Cette instance est très similaire à celle utilisée pour le type `SynthRTL`, à la différence près que `TrNodes` ne contient pas de registres (donc il n'y a plus à en tenir compte). Ainsi, le designer ne souhaitant pas écrire lui même sa fonction d'évaluation de longueur d'un chemin combinatoire pourra simplement passer `treeEvalCrit` à `pipeline`.

Voici une idée de ce à quoi pourrait ressembler le code implémentant toutes ces fonctionnalités :

```
1 instance WithTreeEvals TrNodes where
2   — ... (contenu similaire à celui utilisé pour SynthRTL)
3
4 partition ::
5   TrNodes -> — circuit combinatoire à découper
6   Int -> — nombre d'étages souhaité
7   (TrNodes -> TrTree -> Integer) -> — fonction d'évaluation du chemin critique
8   ( [TrNodes] — retourné: circuits combinatoires correspondant aux étages
9     , [Type] ) — retourné: types des registres situés entre les étages
10  partition circuit evalCrit firstUnusedRegister = — ...
11
12 pipeline :: RTL em => Int — nombre d'étages souhaité
13   -> Transfer a — circuit à découper
14   -> em ( Transfer Type — signal de validité des données consommées (in)
15         , Transfer Type — signal de lecture des données consommées (out)
16         , Transfer Type — signal de validité des données produites (out)
17         , Transfer Type — signal de lecture des données produites (in)
18         , a ) — reste de l'interface
19
20 pipeline (nStages, evalCrit) circuit =
21   let   addPipeline :: RTL em => (a, TrNodes)
22         -> em (Transfer Type, Transfer Type, Transfer Type, Transfer Type, a)
23         addPipeline (rv, circ) =
24           let (stages, regTypes) = partition circ nStages evalCrit
25               in do — ... (ajout des registres, noeuds, signaux de contrôle)
26                 return ( consumedValid, consumedAck
27                       , producedValid, producedAck, rv )
28   in namedNest "pipeline" (TrNodes [] []) addPipeline circuit
```

4.1.2.2 Découpage automatique probabiliste

Nous avons vu à la section précédente un moyen d'effectuer de façon déterministe le découpage en pipeline d'un circuit combinatoire à l'aide d'un modèle de délai combinatoire. Il serait intéressant de palier au caractère approximatif de ce modèle en effectuant un découpage probabiliste, via l'utilisation d'alternatives pondérées.

Les modifications à effectuer par rapport au code précédent sont relativement minimales. On va construire une fonction `partition'`, similaire à `partition` mais renvoyant une liste de découpages possibles pondérés par leur probabilité (le type de retour devient `[(Float, [TrNodes], [Type])]`). On va également lui passer un argument supplémentaire, permettant de spécifier la façon dont on souhaite distribuer les probabilités des différents découpages. Le principe est de faire en sorte que les découpages dont les étages ont des chemins critiques de longueurs approximativement égales aient une probabilité plus élevée d'être sélectionnés que les autres. Le plus simple est de construire un indicateur scalaire du déséquilibre du découpage, et de demander à l'utilisateur de fournir une fonction `f` de `Float` dans `Float`, associant à cet indicateur la probabilité relative d'accepter le découpage correspondant. Si on pose $\{l_i\}_{i \in [1, n]}$, les longueurs des chemins critiques des n étages d'un partitionnement donné, on peut par exemple prendre comme indicateur :

$$d = \frac{\max_{i \in [1, n]} l_i}{\min_{i \in [1, n]} l_i} - 1 \quad (4.1)$$

d ainsi construit est positif, et d'autant plus petit que le découpage est équilibré (selon le modèle utilisé pour évaluer les longueurs de chemins combinatoires). Il peut prendre la valeur 0 dans l'hypothèse où tous les étages ont un chemin critique de longueur exactement égale (toujours d'après le modèle utilisé). La fonction de probabilité relative `f` que doit passer l'utilisateur doit prendre des valeurs plus importantes aux alentours de 0, et converger vers 0 en $+\infty$, plus ou moins vite suivant si l'on souhaite évaluer, lors de la phase d'exploration architecturale, de nombreux partitionnements prédits comme déséquilibrés par le modèle utilisé. On peut par exemple poser `f d = exp (-1.0 * d / a)`, où `a` est un flottant positif reflétant justement la chance accordée aux découpages prédits comme sous-optimaux.

Posons `pipeline'`, la variante probabiliste de `pipeline`. La seule différence à l'utilisation est la présence d'un premier argument correspondant à la fonction `f` décrite plus tôt. Cette fonction sera passée lors de l'appel à `partition'` imbriqué. On définira également `stdPipeline` comme égal à `pipeline' (\x -> exp(-1.0 * d))`, et ayant donc la même signature que `pipeline`.

4.1.3 Étude de cas : pipeline de ray casting dans des grilles hiérarchiques

On illustrera l'utilisation de la génération automatique de pipeline via l'exemple d'une sous-partie d'une unité de lancer de rayons au travers de grilles hiérarchiques.

4.1.3.1 Grilles hiérarchiques

Une grille hiérarchique [95], dont on peut trouver une illustration sur la figure 4.1, est une généralisation de l'octree [28, 34] conservant l'essentiel de ses propriétés intéressantes tout en en proposant de nouvelles.

Plus formellement, une grille hiérarchique $2^n \times 2^n \times 2^n$ est un arbre partitionnant une région cubique d'un espace tridimensionnel. Chaque nœud est un cube, et tout nœud interne comporte 2^{3n} enfants tous de même taille. Le rôle des feuilles dépend de l'application visée. Souvent, en rendu par lancer de rayons, la scène est constituée par un ensemble de primitives [35] ; dans ce cas là, on utilisera la grille hiérarchique comme une structure d'indexation, en annotant chaque feuille par la liste des primitives qui y sont présentes (comme on le fait typiquement avec les *k*d-tree [58] ou leurs dérivés [99] par exemple). Ainsi, on économisera le coût des tests d'intersection entre le rayon et les primitives à proximité desquelles il ne passe pas. De plus, une subdivision adaptative de la grille hiérarchique (plus fine là où la densité des primitives est plus élevée) permet d'accroître la vitesse de traversée des rayons qui passent par des régions relativement vides. Alternativement,

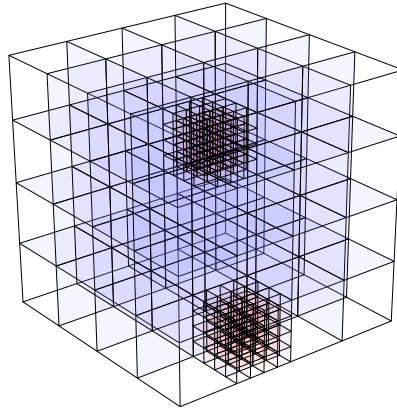


FIGURE 4.1 – Exemple de grille hiérarchique $4 \times 4 \times 4$

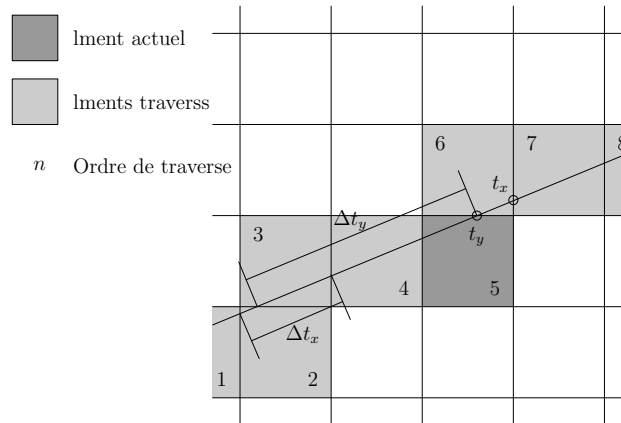


FIGURE 4.2 – Exemple de traversée de grille uniforme 2D via l’algorithme DDA

il est possible d’utiliser les feuilles comme des *voxels*¹, correspondant à des régions cubiques dont la couleur et la densité sont considérées comme homogènes. Ce genre de procédé a notamment sa place en visualisation médicale [23, 32].

L’intérêt des grilles hiérarchiques par rapport aux octrees est d’accroître la localité et la prévisibilité des accès à la mémoire externe. Comme les nœuds internes d’une grille hiérarchique sont typiquement plus gros que ceux d’un octree (qui correspond à une grille hiérarchique $2 \times 2 \times 2$, c.-à-d. la plus petite possible), une quantité d’information pertinente plus importante est récupérée à chaque lecture. De plus, l’organisation des pointeurs vers les enfants dans chaque nœud permet d’observer des gains en performances en cas d’utilisation de caches prédictifs [95] et potentiellement, dans une certaine mesure, de mécanismes de *stride prediction* ou similaires [24].

4.1.3.2 Principe de la traversée

La traversée de grilles hiérarchiques n’est pas beaucoup plus compliquée que celle d’octrees. On va utiliser une méthode paramétrique inspirée du *Digital Differential Analyzer* (DDA) [8], un algorithme utilisé à l’origine pour la traversée de grilles uniformes. Son principe est illustré sur la figure 4.2. Une fois le rayon paramétré par une origine et un vecteur directeur, on réalise la traversée en maintenant une variable correspondant au paramètre du point d’entrée dans la cellule

1. Contraction de *volume pixel*

courante, et 3 variables correspondant aux paramètres points d'intersection entre le rayon et les plans des faces par lesquelles il peut potentiellement sortir de la cellule. La face par laquelle il sort est celle dont le paramètre correspondant est le plus faible. On remarquera que la différence de paramètres entre les points d'intersections et deux faces opposés d'une cellule suivant un axe donné est une constante qu'il suffit de calculer qu'une seule fois par rayon lancé.

L'algorithme utilisé au final pour la traversée de grilles hiérarchiques est exposé sur le listing des figures 4.3 et 4.4. Le principe reste identique, mais il faut en plus gérer l'aspect récursif de la grille. Dans le code donné en exemple, on a supposé que l'on avançait dans le sens des coordonnées croissantes suivant tous les axes. On peut faire cette hypothèse sans perte de généralité, en appliquant la méthode présentée par [80] et qu'il n'est pas primordial de détailler ici.

Nous avons choisi cet algorithme pour diverses raisons pragmatiques, telles que l'utilisation de la géométrie projective lors de la traversée [59] et la minimisation du nombre de multiplicateurs utilisés. Pour plus de détails, veuillez vous référer à [95]. Parmi les autres algorithmes qu'il aurait été envisageable d'utiliser, mentionnons [80], déjà cité, ou encore [91]. Plus généralement, l'adaptation aux grilles récursives $2^n \times 2^n \times 2^n$ d'algorithmes de traversée d'octrees est en effet possible au prix de n itérations effectuées par nœud pour chaque itération qui aurait été effectuée sur un octant².

4.1.3.3 Organisation matérielle

On souhaite réaliser une unité matérielle réalisant le traitement décrit par le listing de la figure 4.4. Elle aura vocation à s'intégrer dans l'unité de traversée d'un coprocesseur de lancer de rayon par faisceaux cohérents. La figure 4.5 présente l'intégralité du système considéré. En plus du coprocesseur de lancer de rayons, il faudra un microprocesseur (pour réaliser certains calculs liés notamment à l'initialisation du faisceau de rayons) et un contrôleur mémoire (pour accéder à la scène stockée dans une RAM externe du fait de sa grande taille potentielle). L'unité de traversée elle-même se compose de trois blocs (figure 4.6). Le bloc de propagation en phase (PPU) maintient l'information concernant l'état de traversée de tous les rayons d'un faisceau cohérent. Elle envoie cette information, rayon par rayon, à l'interface de cache qui accède aux portions de la scène concernées avant de relayer le tout à l'unité de recherche de voisin (NFU). C'est elle qui va déterminer la cellule à laquelle le rayon devra ensuite accéder, en appliquant l'algorithme de la figure 4.4. Elle transmet le résultat à la PPU (qui peut alors mettre à jour l'information du rayon correspondant) ainsi qu'à l'unité de composition dont le but est de générer une image de la scène. C'est au bloc NFU que nous allons nous intéresser.

Son organisation peut être trouvée sur la figure 4.7. Il y a deux types de traitement possibles : le tri et la remontée, qui correspondent respectivement à la clause «alors» et «sinon» du «si» du listing de la figure 4.4. Comme la majorité des cellules traversées seront des feuilles (clause «alors»), la partie tri est pipelinée, tandis que la partie remontée est réalisée de façon séquentielle en n cycles (dans le cas d'une grille hiérarchique $2^n \times 2^n \times 2^n$). Cela permet d'obtenir de bonnes performances tout en économisant la quantité de logique nécessaire. À un moment donné, un seul de ces chemins est actif, du fait qu'ils partagent des ressources communes (ce qui n'apparaît pas sur la figure 4.7 pour ne pas la surcharger). Le pipeline de tri comporte trois étages.

4.1.3.4 Résultats

Le bloc NFU a dans un premier temps été écrit et testé en VHDL. Il a été ensuite réécrit en utilisant le MoC RTL présenté à la section 3.2, en imitant au maximum la structure du code VHDL original. Ces deux versions figurent dans l'annexe C. Afin de rendre les deux aussi comparables que possible, un certain nombre de possibilités offertes par notre hiérarchie de MoC et non détaillées jusqu'ici ont été utilisées. Par exemple, les enregistrements VHDL ont été imités par des listes se servant de l'opérateur `--` défini à cette fin. Généralement, on peut éviter d'y avoir recours dans la mesure où elles peuvent être favorablement remplacées par les types algébriques natifs de Haskell (comme on l'a d'ailleurs fait jusqu'ici).

2. c.-à-d. un nœud d'octree

4.1. DÉCOUPAGE DES PIPELINES EN ÉTAGES

```

1   $\vec{r}$  ← vecteur directeur unitaire du rayon
2  t ← paramètre du point d'entrée dans la cellule courante
3  ( $t_x, t_y, t_z$ ) ← paramètres des points d'intersection avec les bords
4  ( $\Delta t_x, \Delta t_y, \Delta t_z$ ) ← différences de paramètres entre 2 faces successives suivant chaque axe
5  ( $p_x, p_y, p_z$ ) ← position absolue de la cellule courante
6  depth ← profondeur de la cellule courante
7  max_depth ← profondeur maximale de l'arbre
8
9  // L'unité de  $p_x, p_y$  et  $p_z$  est le côté de la plus petite cellule possible, la
10 // taille de cette dernière étant déterminée par max_depth. Par exemple, dans
11 // une grille  $4 \times 4 \times 4$  avec une profondeur maximale de 5,  $p_x, p_y$  et  $p_z$  sont
12 // compris entre 0 et 1023 (car  $4^5 - 1 = 1023$ ).

```

FIGURE 4.3 – Variables caractérisant un rayon à un moment de la traversée

```

1  si (la cellule actuelle est une feuille) alors
2      envoyer les annotations (couleur, etc.) à l'unité de composition
3
4      // Il faut trouver la cellule suivante (à une profondeur égale ou inférieure)
5
6      // Détermination de la direction k dans laquelle aller:
7      pourtout l ∈ {x, y, z}
8          si  $t_l = \min(t_x, t_y, t_z)$  alors k = l;
9
10     // Si on sort de la cellule courante, remonter tant que nécessaire:
11     tantque  $p_k[(h-1)..(h-n)] = '1..1'$  avec  $h = n * (\text{max\_depth} - \text{depth})$ 
12         si depth > 0 alors
13             depth ← depth - 1
14         sinon
15             la traversée est terminée pour le rayon courant
16             pour m dans 0 jusqu'à (n - 1)
17                 h ← n * (max_depth - depth) - (m + 1)
18                 pourtout l ∈ {x, y, z}
19                     si  $p_l[h] = '1'$  alors
20                          $t_l \leftarrow t_l - \Delta t_l$ 
21                          $\Delta t_l \leftarrow 2 * \Delta t_l$ 
22
23             // Enfin, «avancer» dans la nouvelle cellule:
24              $p_k \leftarrow p_k + 2^{(n * (\text{max\_depth} - \text{depth} - 1))}$ 
25              $t_k \leftarrow t_k + \Delta t_k$ 
26     sinon
27         // Il faut continuer à descendre
28         pour m in 0 to (n - 1)
29             h ← 2 * (max_depth - depth) - (m + 1)
30             pourtout l ∈ {x, y, z}
31                  $\Delta t_l \leftarrow \Delta t_l / 2$ 
32                  $t_l \leftarrow t_l - \Delta t_l$ 
33                 si  $t > t_l$  alors
34                      $t_l \leftarrow t_l + \Delta t_l$ 
35                      $p_l[h] \leftarrow '1'$  // hème bit of  $p_l$  passé à 1
36                 sinon
37                      $p_l[h] \leftarrow '0'$  // hème bit of  $p_l$  passé à 0
38             depth ← depth + 1;

```

FIGURE 4.4 – Algorithme de détermination de la prochaine cellule (utilisé à chaque itération de la traversée d'une grille hiérarchique $2^n \times 2^n \times 2^n$)

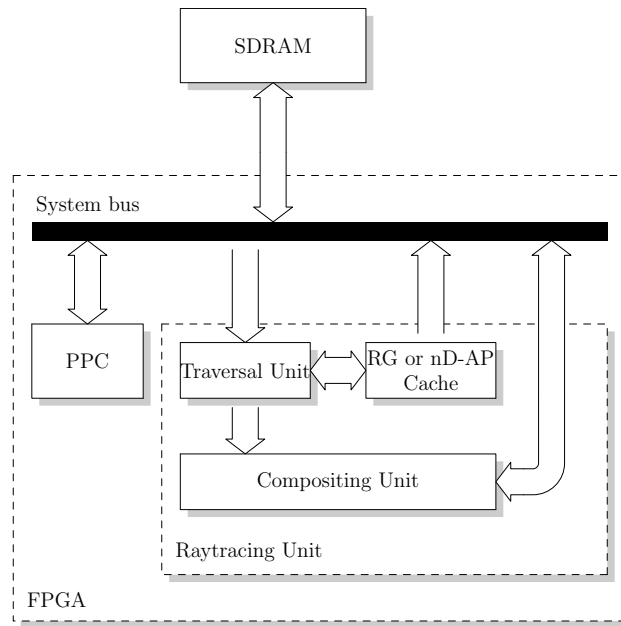


FIGURE 4.5 – Schéma d'ensemble du SoPC de rendu par lancer de rayons considéré dans notre exemple

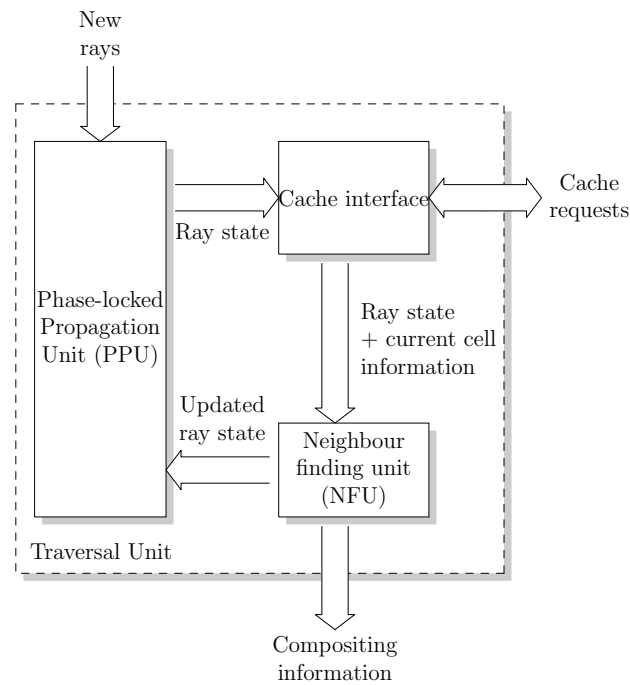


FIGURE 4.6 – Organisation de l'unité de traversée

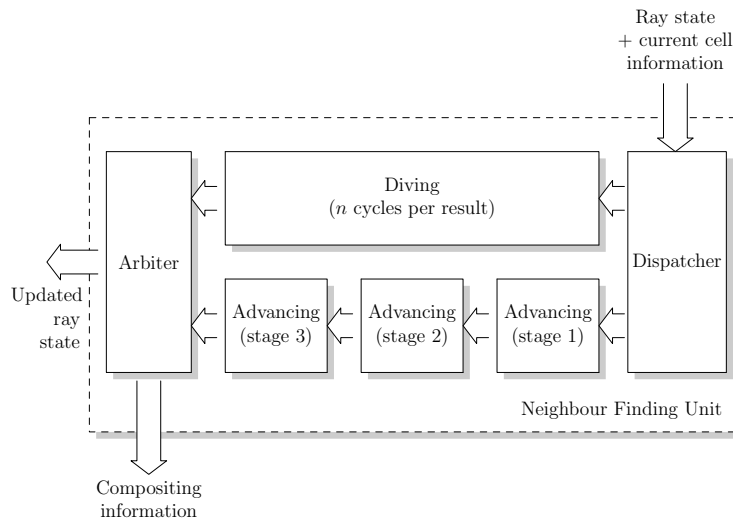


FIGURE 4.7 – Organisation de l'unité de recherche de voisins

Implémentation	lignes de code	taille du code source
VHDL	736	21,3 Ko
Haskell	507	14,9 Ko
Haskell (avec <code>stdPipeline</code>)	442	13,5 Ko

FIGURE 4.8 – Comparaison entre trois implémentations de l'unité de recherche de voisins (en terme de code source)

Enfin, la version Haskell précédente a été modifiée de sorte à ce que le pipeline de tri soit découpé automatiquement par `stdPipeline`. Le code résultant peut être trouvé à la section C.3. Il a fallu exprimer tout le pipeline sous forme de `Transfer`, ce qui a justifié l'introduction de quelques nouvelles constructions pour faciliter cette tâche, comme par exemple `mux`' (se comportant comme `mux`, défini dans le MoC RTL). Normalement, leur compréhension devrait être intuitive.

Une métrique possible pour comparer ces trois implémentations est la taille du code. Elle présente l'avantage d'être objective et facile à mesurer. En revanche, elle ne rend que vaguement compte des critères qui comptent réellement, tels que le temps de développement en heure-personne, le temps de débogage et de profiling, etc. En gardant ces limitations en tête, le compte des lignes pour exprimer le design en utilisant ces trois stratégies est donné sur la figure 4.8. Les lignes vides, ou ne comportant que des espaces ou des commentaires ne sont pas comptabilisées.

D'ores et déjà, remarquons que le code VHDL est nettement plus long (environ 45%) du fait de son organisation en processus combinatoire et synchrone, chacun annoté de sa liste de sensibilité. Le MoC RTL considère qu'il n'y a qu'un seul domaine d'horloge et que le programmeur ne cherche pas à synthétiser de nouveaux registres ou latches via omission de signaux dans la liste de sensibilité. Ceci permet d'accroître la concision du code au détriment de la généralité du modèle de calcul. (Il aurait également été possible de raccourcir le code VHDL de référence en l'organisant de façon à ce que les registres soient justement synthétisés automatiquement. Cette méthode est moins portable, et surtout réduit la visibilité du programmeur sur le matériel réellement synthétisé, le chemin critique, etc.)

La synthèse automatique du pipeline de tri permet de raccourcir encore un peu le code : environ 10% si l'on compare les listings des sections C.2 et C.3. Sachant que les déclarations restent identiques entre ces deux listings, et que le traitement réalisé est le même, cette économie intervient sur la partie contrôle, qui devient par ailleurs plus lisible (il devient beaucoup plus

facile de voir que l'on est en train de réaliser un pipeline, par exemple). La partie traitement bénéficie également d'une lisibilité accrue : l'algorithme de la figure 4.4 est désormais dans une seule fonction (`tri`) au lieu de trois (`triEtage1`, `triEtage2`, `triEtage3`). Enfin, le code est plus facile à maintenir ou à paramétrer, dans la mesure où le nombre d'étages est donné par un simple nombre entier (potentiellement choisi parmi un ensemble d'alternatives).

4.2 Allocation de ressources sur GPU

Un aspect notable de la méthodologie proposée sur lequel nous n'avons pas vraiment insisté jusqu'ici est son adaptation au concept de plateforme [29, 83]. Par le terme «plateforme» on désignera une famille d'architectures plus ou moins fixe et potentiellement composée d'éléments logiciels en plus des composants matériels. L'utilisateur peut avoir la possibilité d'introduire des composants personnalisés type ASIC, ou peut avoir à sa disposition de la logique programmable, mais ce n'est nullement obligatoire et bien des plateformes sont dotés de caractéristiques matérielles entièrement fixées par le fabriquant.

Le fait de pouvoir utiliser les ressources matérielles ou logicielles offertes par une plateforme est un aspect important. Le design intégral d'un ASIC *full custom* dans les technologies modernes est une chose à la portée d'un nombre de plus en plus restreint d'entités. Souvent, on utilisera des bibliothèques de cellules existantes, des IP existantes. Dans bien des cas, partir d'une architecture existante en n'en modifiant que certains blocs sera le choix le plus judicieux. Similairement, dans le cas de la logique programmable, une surface de plus en plus significative est consacrée à des composants spécifiques, tels des multiplicateurs, des DSP, voire des microprocesseurs entiers (dotés de mécanismes de caching et de MMUs), au détriment dans une certaine mesure des cellules programmables. Il est primordial de faire bon usage de ces ressources.

Dans cette section, nous allons aborder la problématique de la programmation parallèle sur GPU. Les GPUs constituent un exemple de plateforme intéressant, du fait de leur caractère massivement parallèle, et des problématiques d'accès aux données qui en résultent. Une carte graphique programmable est en général dotée d'une ou plusieurs puces (GPU, pour *Graphics Processing Unit*) contenant des matrices de processeurs SIMD avec leurs mémoires on-chip associées, ainsi que de mémoire vive externe, reliée aux puces par des bus haut débit. Les puces graphiques sont également dotées d'une partie contrôle, permettant l'échange de données avec le CPU par l'intermédiaire d'un bus (PCI Express, quasi-universellement). En dépit du fait que le programmeur n'ait aucune marge de manœuvre sur le matériel dont il se sert (au delà de la sélection dans un catalogue du ou des modèles de carte graphique qu'il utilisera pour son application), il va se trouver confronté à un certain nombre de problèmes architecturaux bien connus du designer de SoCs : organisation de la hiérarchie mémoire, synchronisation entre un grand nombre d'unités de traitement, etc.

Dans la suite de cette section, nous rappellerons les bases de la programmation sur GPU, puis nous présenterons les intérêts que peut avoir notre approche pour la réalisation de cette tâche.

4.2.1 Programmation sur GPU

4.2.1.1 Historique

L'organisation actuelle des GPU programmables tient beaucoup à l'évolution qu'ont connus les processeurs graphiques au cours des deux dernières décennies.

Comme son nom l'indique, une carte graphique a vocation première à s'occuper de l'affichage. Nous ne nous intéresserons pas aux systèmes d'affichage vectoriels, qui utilisaient des faisceaux d'électrons pour dessiner directement les formes souhaitées à l'écran, à la manière d'un oscilloscope (ce type d'affichage a pratiquement disparu). Ainsi, nous considérerons qu'à la base, une carte graphique contenait au minimum de la RAM dans laquelle était stocké le *framebuffer* (c.-à-d. le bitmap des pixels affichés à l'écran) et une puce permettant de transmettre le signal correspondant à l'écran.

Très tôt, il est devenu tentant d'étendre les possibilités du GPU dans le but de soulager le CPU de certains traitements graphiques. Par exemple, il était possible de créer

un framebuffer de taille importante et de n'en afficher qu'une partie à l'écran afin de créer des effets de défilement ou de *double buffering* gérés en matériel. Similairement, les couleurs pouvaient être gérées par l'intermédiaire d'une LUT («palette»), autorisant certaines animations simples ne consommant que très peu de puissance CPU.

Rapidement, les primitives 2D les plus courantes se sont vues gérées par le GPU (par exemple : remplissage de rectangles par une couleur uniforme, copies «2D» entre deux zones de la mémoire vidéo, etc.). Puis ce fut au tour des primitives 3D. Alors que des contrôleurs DMA légèrement modifiés suffisaient généralement pour l'essentiel des tâches liées à l'accélération 2D, il a fallu mettre en place un pipeline graphique nettement plus complexe afin de gérer les techniques les plus couramment utilisées en rastérisation de scènes tridimensionnelles (par exemple : remplissage de triangles par ombrage de Gouraud [31], texturage par *wv mapping* avec correction de perspective [36], etc.). L'organisation de la mémoire vidéo a également dû évoluer, non seulement pour contenir les textures, mais aussi des tampons additionnels, tels que le *depth buffer* (parfois appelé *z-buffer*, contenant la distance de chaque pixel au plan de la camera) ou le *stencil buffer* (dont il appartient au programmeur de choisir le rôle). Pour chaque pixel considéré lors de la rastérisation d'un triangle, les valeurs correspondantes de ces deux tampons permettront de déterminer si l'on écrira effectivement la couleur calculée par ombrage dans le framebuffer, ou si au contraire on va laisser la valeur du framebuffer inchangée pour ce pixel.

L'étape suivante dans la complexification croissante des GPU a consisté à permettre à l'utilisateur de spécifier ses propres méthodes d'ombrage par le biais de petits programmes exécutés sur des processeurs SIMD minimalistes intégrés dans le GPU. Un tel programme allait typiquement être lancé pour chaque pixel d'un triangle donné («pixel shader» ou «fragment shader», les deux termes étant équivalents), et pouvait servir à obtenir des effets de textures multiples, de *bump mapping*, d'eau, de fourrure, etc. Du fait du caractère généraliste des processeurs servant à évaluer les pixel shaders, ils ont également été mis à contribution pour d'autres tâches, telles que la transformation des positions des sommets des triangles («vertex shaders»³). Au départ, il existait des limitations importantes concernant la taille d'un shader (en nombre d'instructions) et les structures de contrôle qui y étaient présentes, mais progressivement elles se sont assouplies. Comme les shaders devenaient de plus en plus complexes, les GPUs consacraient des surfaces de plus en plus importantes aux processeurs destinés à leur évaluation.

Bien que les GPU modernes contiennent toujours des blocs spécifiques au pipeline de rendu, il est devenu possible de les utiliser dans un mode où ils sont désactivés et où le programmeur a le champ libre pour utiliser les processeurs SIMD à sa guise. C'est ce que nVidia (le premier constructeur à avoir permis ce mode d'utilisation) appelle le mode *compute*. Étant donné les caractéristiques des processeurs de traitement des GPUs, l'utilisation du mode *compute* permet d'obtenir des performances différentes de celle des CPUs classiques. En règle générale, ces performances sont supérieures pour des applications fortement parallèles, exhibant peu d'irrégularités dans le contrôle. Elles peuvent chuter (potentiellement de façon spectaculaire) en cas de parallélisme à trop gros grain, de barrières de synchronisation mal placées, ou d'accès aléatoires à la mémoire vidéo (suivant la façon dont ceux-ci sont effectués — la présence de plusieurs caches et scratchpads permet de mitiger ce problème).

Il est clair que la façon de programmer un GPU varie grandement en fonction de la génération de carte vidéo considérée. D'ores et déjà, certaines applications image peuvent être réalisées astucieusement sur un GPU sans utiliser les processeurs SIMD (donc sans vraiment le programmer). C'est par exemple le cas de [20, 52]. Ces exemples ne se limitent cependant qu'à un nombre restreint de problèmes spécifiques. Le premier véritable moyen de programmer un GPU est d'utiliser ses pixel shaders tout en restant en mode rendu. Il existe deux langages/API principales permettant de réaliser cette tâche : GLSL [50] (*OpenGL Shading Language*, permettant de programmer les shaders depuis OpenGL) et HLSL (*High Level Shader Language*, utilisable depuis DirectX, mais aussi en conjonction avec OpenGL via le compilateur Cg de nVidia). Afin de traiter un problème général avec ces langages, on va généralement (1) placer les données à traiter dans une

3. On pourra remarquer que le terme *shader* est resté, malgré qu'il ne s'agisse plus de calculer les ombres propres en un point d'un triangle.

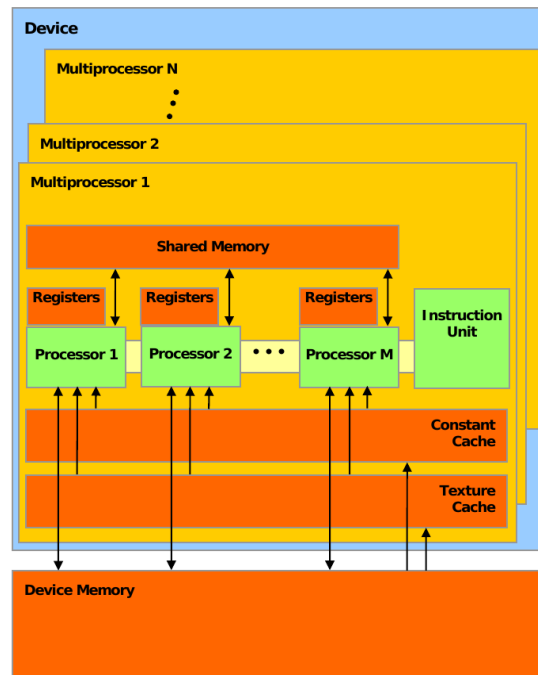


FIGURE 4.9 – Schéma architectural d'un GPU de nVidia (figure empruntée à [69])

ou plusieurs textures, (2) écrire un shader réalisant la majeure partie du traitement à effectuer, et utilisant les textures ainsi que la position du pixel sur lequel il est appelé comme données d'entrée, (3) créer un framebuffer de taille convenable et le remplir intégralement en rasterisant deux triangles utilisant le shader précédemment créé, et (4) récupérer une copie du framebuffer (au lieu de l'afficher). Si nécessaire, l'algorithme peut être implémenté en plusieurs passes. La programmation en mode compute reste assez similaire sur le principe. Au lieu de remplir des triangles, on va créer des «grilles» de threads (évaluant tous un même *noyau*, équivalant à un shader). Les principaux avantages résident dans la possibilité qu'ont, dans une certaine mesure, les threads à se synchroniser entre eux, à communiquer, ou à écrire librement dans la mémoire vidéo (et non plus simplement à un emplacement désigné). Les APIs/langages permettant d'utiliser les GPUs en mode compute sont CUDA [72] (spécifique aux GPUs de nVidia) ou OpenCL [92]. Nous nous appuyerons sur l'exemple des architectures nVidia dans la suite de cette section.

4.2.1.2 Architecture

Un GPU nVidia est composé d'un ensemble de multiprocesseurs comportant chacun plusieurs processeurs connectés à une unité d'instructions commune (et partageant donc un même compteur ordinal — c'est en cela que l'on peut parler de SIMD). Chaque processeur dispose de son propre banc de registres. Un multiprocesseur comporte également une mémoire partagée commune que chaque processeur peut lire ou écrire très rapidement (en 1 cycle si aucun autre processeur ne cherche à accéder au même banc ; il y a suffisamment de bancs pour que tous les processeurs puissent accéder à cette mémoire simultanément et sans conflit dans les meilleurs cas), ainsi que deux mémoires en lecture seule : la mémoire de constantes (de faible taille mais qui reste rapide indépendamment de la localité des accès) et la mémoire de textures (dotée d'un cache, et rapide du moment que les données désirées y sont présentes). Chaque processeur peut accéder à la mémoire centrale du GPU et partagée par tous les multiprocesseurs. Cette mémoire a une latence élevée mais un débit conséquent. Pour en tirer le meilleur parti, il convient d'y accéder par bursts,

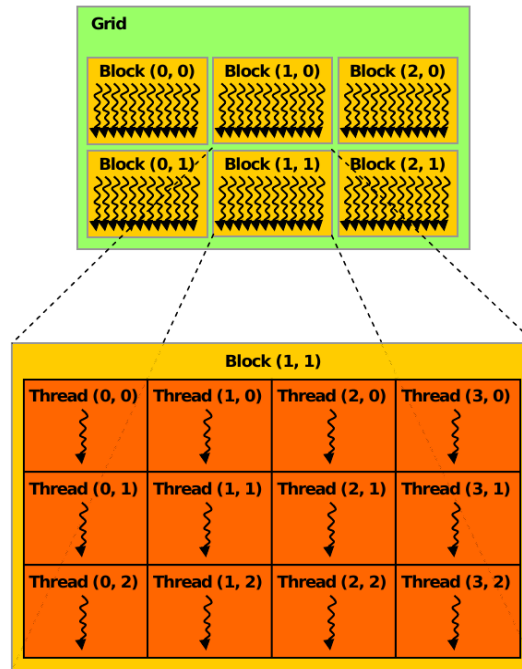


FIGURE 4.10 – Grilles et blocs de threads sous CUDA (figure empruntée à [69])

ce qui est possible en faisant collaborer tous les processeurs d'un multiprocesseur donné. Cette architecture est résumée sur la figure 4.9. Un programme utilisant le GPU est composé d'un ou plusieurs noyaux constituant la partie GPU, mais aussi de la partie hôte, s'exécutant sur le CPU et chargée de préparer les données à traiter, d'allouer la mémoire (aussi bien sur le CPU que sur le GPU), de copier les données entre le CPU et le GPU, et enfin de lancer les noyaux eux-mêmes. Ce lancement pouvant être asynchrone, le CPU peut effectuer des calculs en parallèle du traitement sur GPU.

Comme le nombre de multiprocesseurs est, d'une part, potentiellement élevé, et d'autre part, *a priori* inconnu (un même binaire peut tourner sur n'importe quel GPU nVidia de la génération pour laquelle il a été conçu, ou de toute génération subséquente), le programmeur est encouragé à utiliser un parallélisme au grain aussi faible que possible. Cela est formalisé par le concept de grille et de bloc de threads, illustré par la figure 4.10. Un bloc est un groupement de threads (d'un même noyau) tandis qu'une grille est un groupement de blocs. Les threads d'un bloc ont la particularité de s'évaluer sur un même multiprocesseur, quitte à être ordonnancés si la taille du bloc excède celle du nombre de processeurs du multiprocesseur. Comme le basculement de contexte est géré en matériel et se fait quasi instantanément, il est même recommandé [71] de créer des grilles aussi grandes que possibles; les threads en attente de données (depuis la mémoire SDRAM, en cas de miss dans le cache de texture, etc.) peuvent être descheduled pour un maximum d'efficacité. La taille d'un bloc est limitée par la quantité de registres et de mémoire partagée que requiert chaque thread. Le concept de grille est aux multiprocesseurs ce que celui de bloc est aux processeurs d'un même multiprocesseur. Un bloc de la grille va être affecté à chaque multiprocesseur. Si la grille comporte plus de blocs qu'il y a de multiprocesseurs, les blocs restants sont évalués au fur et à mesure que les multiprocesseurs finissent de traiter les blocs qui leur ont été préalablement assignés. Comme le nombre de multiprocesseur croît à chaque génération de GPU, le programmeur est, là aussi, encouragé à utiliser des grilles de grande tailles.

Une conséquence de ce découpage en blocs est que les threads d'un bloc peuvent communiquer entre eux, mais pas avec ceux de blocs différents. En effet, rien ne garantit que plusieurs blocs sont

évalués simultanément (il peut n'y avoir qu'un seul multiprocesseur) et un multiprocesseur donné ne peut changer de bloc traité avant que tous les threads le composant ne se terminent.

4.2.1.3 Problèmes

Utilisée correctement, ce genre d'architecture peut offrir une puissance de calcul considérable. Il faut cependant faire attention à un grand nombre de paramètres dont le comportement n'est pas forcément prévisible si l'on souhaite aboutir à des performances maximales. Notamment :

- Il faut judicieusement choisir dans quelle mémoire stocker les données. Pour les données écrites, il n'y a que le choix entre la mémoire SDRAM externe et la mémoire partagée, sachant qu'il va falloir de toutes façons que le résultat soit stocké dans la mémoire externe si on veut pouvoir le lire depuis le PC hôte. Pour les données fournies en entrée, c'est nettement plus épineux puisque les mémoires de textures et de constantes peuvent également être utilisées. Comme la mémoire de textures est cachée, il est stratégique de choisir les bonnes données à y placer. Placer toutes les données d'entrée en textures peut se traduire par du *cache trashing* si elles sont trop nombreuses. Similairement, la mémoire de constantes a une taille limitée, et tous les processeurs d'un multiprocesseur doivent lire le même mot en même temps (ou s'abstenir de lire) pour que l'accès se fasse en un cycle. Il faut donc choisir au plus quelques tableaux de quelques kilo-octets à y placer.
- Les tailles optimales de blocs sont potentiellement contre intuitives. Ce que l'on gagne en ordonnancement via des blocs de grandes tailles peut-être perdu du fait des contraintes sur le nombre de registres utilisables, et vice versa. De plus, il faut également choisir quelles données vont être traitées au sein d'un bloc. Là, deux critères sont à respecter : il faut qu'elles soient les plus proches possibles (si elles sont placées en SDRAM ou en textures) pour faciliter les accès, et il faut qu'elles causent le moins de branchements divergents possible. En effet, comme les threads qui s'exécutent à un moment donné sur un multiprocesseur (on parle de *half warp*) partagent un même compteur ordinal, les branchements divergents sont traités séquentiellement en désactivant tour à tour les threads non concernés par les clauses évaluées.
- Parfois, il faut choisir entre un seul grand noyau réalisant une tâche ou plusieurs noyaux plus petits en réalisant chacun une partie. La deuxième alternative permet d'économiser en registres (d'où des grilles plus grandes, donc des accès mémoire plus facilement recouverts) au prix de la pénalité liée au lancement d'une grille de threads et potentiellement, des transferts mémoire supplémentaires nécessaires pour que les noyaux successifs communiquent.
- Enfin, malgré que l'API de CUDA soit stable et compatible avec toutes les cartes nVidia qui sont apparues depuis qu'elle existe, il n'en reste pas moins que les différences entre architectures se font clairement sentir :
 - Les règles à suivre pour déclencher une lecture ou écriture en burst depuis ou vers la SDRAM changent suivant les générations.
 - La toute dernière architecture nVidia, le Fermi, permet de cacher les accès vers la SDRAM en lecture comme en écriture, ce qui change de manière drastique la donne.
 - La taille de la mémoire partagée a également évolué avec la dernière génération de GPU.
 - Les performances de l'arithmétique sur entiers ou sur flottants double précision ont également varié.

Deux conclusions s'imposent :

- Un code efficace pour une génération de GPU peut s'avérer sous-optimal pour une autre.
- Même pour une génération donnée, il n'est pas aisé de voir *a priori* comment organiser son programme.

Cet état de fait incite à penser que les approches basées sur l'exploration de l'espace de programme ont leur place dans la programmation sur GPU.

4.2.2 Modèles de calcul appropriés

4.2.2.1 Direct

Les outils de nVidia permettant d'utiliser leur GPUs en mode compute offrent deux niveaux d'abstraction : le jeu d'instructions PTX (*Parallel Thread Execution*) [73], permettant d'exprimer les noyaux en se rapprochant au maximum de la séquence d'instructions évaluée par les multiprocesseurs⁴, ainsi qu'un langage dérivé du C, CUDA, et permettant d'exprimer les noyaux comme le code hôte dans un même formalisme.

Dans un premier temps, nous allons mimer ces deux niveaux d'abstraction. L'intérêt en est multiple :

- Il est intéressant d'offrir à l'utilisateur la possibilité d'utiliser ces deux niveaux d'abstraction. C'est naturel pour le code CUDA, qui est en soi assez facile à écrire et convient dans la grande majorité des projets. La nécessité de programmer directement en PTX est plus rare, mais peut néanmoins se justifier dans les cas où le nombre de registres utilisés par un noyau est un aspect critique du point de vue des performances.
- Il est possible de vouloir intégrer du code PTX ponctuellement dans un projet écrit par ailleurs à un niveau d'abstraction plus élevé (un peu comme on ferait avec le mot-clé `asm` en C) ; ceci n'est pas possible depuis CUDA, mais rien ne nous empêche de rajouter cette possibilité dans les MoCs que nous écrivons.
- Enfin, les moteurs associés à ces MoCs pourront servir de back-end à des MoCs de niveau d'abstraction plus élevé

Comme on peut le voir sur le listing de la figure 4.11, imiter l'assembleur PTX est assez simple, et permet de se persuader une fois de plus des capacités de Haskell en terme de définition d'EDSL (*Embedded Domain Specific Language*). On définit d'abord quelques types concrets permettant de représenter les objets manipulés dans le MoC PTX, à savoir les valeurs immédiates (correspondant aux littéraux entiers et flottants), les opcodes des instructions, les modificateurs (permettant d'explicitement le comportement exact d'une instruction, par exemple en spécifiant les types et tailles des opérandes, le mode d'arrondi souhaité, les contraintes d'alignement ; la norme PTX définit rigoureusement quels modificateurs s'appliquent à quelles instructions) et les opérandes. Le modèle de calcul permet simplement de rajouter une instruction (`instr`), d'appliquer un ou plusieurs prédicats à l'évaluation d'une instruction (`at`), et de déclarer des identifiants correspondant aux ressources du GPU : registres, mémoire locale (passage de paramètres au noyau), mémoire partagée (scratchpad commun à tous les processeurs d'un multiprocesseur), mémoire globale et mémoire des constantes. Enfin, nous définissons et encourageons l'utilisateur à employer des alias pour les instructions. Ces alias utilisent le pattern matching pour s'assurer que les modificateurs utilisés avec chaque instruction sont ceux qu'elle admet. En cas d'erreur, le programme échouera à l'élaboration (ce qui est équivalent à un échec de la compilation avec `ptxas`, l'assembleur `ptx` fourni par nVidia). Bien que assez simple dans sa structure, ce code définit un sous-langage dont l'expressivité est quasi équivalente à celle de l'ISA PTX dans sa version 1.2 [70]. La figure 4.12 permet de comparer un design exprimé dans le MoC PTX à la portion correspondante du fichier `.ptx` équivalent, tel qu'on le fournirait à `ptxas`. On peut constater que notre EDSL ne souffre pas de lourdeurs ou d'aspects contre-intuitifs sur le plan syntaxique.

Dans l'absolu, on aurait pu faire la même sorte de travail pour écrire un MoC mimant CUDA (c'est à dire un sous-ensemble du C++ avec quelques nouveaux mots clé et éléments syntaxiques propres à la programmation sur GPU). Cela dit, on peut tout aussi bien repartir du MoC impératif défini à la section 3.2. Il suffira d'ajouter des primitives pour gérer les aspects propres à la programmation sur GPU. On utilisera un MoC pour l'hôte (capable d'allouer de la mémoire et de lancer des noyaux) et un pour les noyaux (capable d'utiliser les ressources propres du GPU, de synchroniser les threads d'un même warp, d'embarquer du code PTX, etc.). La structure du code correspondant, ainsi qu'un exemple de design associé, est illustrée sur les figures 4.13 et

4. Il est à noter que le «bytecode» obtenu par compilation d'un fichier PTX est ensuite traduit par le pilote de la carte graphique en code machine effectivement exécuté sur la carte ; en cela il ne s'agit pas vraiment de code assembleur

```

1  — Gestion des valeurs immédiates —
2  data PtxConst a = PtxConst a | PtxArray [PtxConst a]
3  data PtxImm = PtxImmF (PtxConst Float) | PtxImmI (PtxConst Integer)
4
5  class PtxImmediate a where toPtxImm :: a -> PtxImm
6  instance PtxImmediate Float where toPtxImm val = PtxImmF (PtxConst val)
7  instance PtxImmediate Integer where toPtxImm val = PtxImmI (PtxConst val)
8  instance PtxImmediate [Float] where
9      toPtxImm vals = PtxImmF (PtxArray (map PtxConst vals))
10 instance Integral a => PtxImmediate [a] where
11     toPtxImm vals = PtxImmI (PtxArray (map PtxConst (map toInteger vals)))
12
13 — MoC et données associées —
14 data PtxOpcode
15     = Abs | Add | AddC | And | Atom | Bar | Bra | BrkPt | Call | CNot | Cos
16       | Cvt | Div | Ex2 | Exit | Fma | Ld | Lg2 | Mad | Mad24 | Max | Membar
17       | Min | Mov | Mul | Mul24 | Neg | Not | Or | PMSEvent | Rcp | Red | Rem
18       | Ret | Rsqrt | Sad | Selp | Set | Setp | ShL | ShR | Sin | Slct | Sqrt
19       | St | Sub | SubC | Tex | Trap | Vote | Xor
20 data PtxModifier
21     = S8 | S16 | S32 | S64           — type: entiers signés
22       | U8 | U16 | U32 | U64         — type: entiers non-signés
23       | F16 | F32 | F64             — type: flottants
24       | B8 | B16 | B32 | B64        — type: bits (c.-à-d. non typé)
25       | Pred                         — type: predicats
26       | V2 | V4                       — type: vecteurs
27       | Eq | Ne | Lt | Le | Gt | Ge   — mode: opération de comparaison
28       | Rn | Rz | Rm | Rp             — mode: arrondi des flottants
29       | Rni | Rzi | Rmi | Rpi        — mode: arrondi des entiers
30       | Sat                           — mode: pas d'overflow ("add", etc.)
31       | Cc                            — mode: écrire dans le registre CC
32       | Hi | Lo | Wide                — mode: comportement de "mul"/"mad"
33       | Align Int                    — mode: alignement du mot écrit
34       | Full | Approx                 — mode: précision des ops. flottantes
35       | Ftz                          — mode: flottants sous-normaux annulés
36       | TexRef | SampleRef | SurfRef — mode: identifiants des textures
37       | Dim1D | Dim2D | Dim3D         — mode: dimensions des textures
38       | Const | Global | Local        — mémoires
39       | Param | Shared                — mémoires
40 deriving Eq
41 data PtxOperand
42     = Ident String                   — identifiant
43       | Reg Int                      — registre ordinaire
44       | Predicate Int                — registre predicat
45       | Val PtxImm                   — adressage immédiat
46       | Direct Integer               — adressage direct
47       | Indirect Integer              — adressage indirect
48
49 class Explorable em => Ptx em where
50     instr :: [Int] -> PtxOpcode -> [PtxModifier] -> [PtxOperand] -> em ()
51     at :: [PtxOperand] -> em () -> em ()
52     namedReg :: [PtxModifier] -> String -> em (PtxOperand)
53     namedLocal :: [PtxModifier] -> [Int] -> String -> em PtxOperand
54     namedShared :: [PtxModifier] -> [Int] -> String -> em PtxOperand
55     namedGlobal :: PtxImmediate a =>
56         [PtxModifier] -> [Int] -> a -> String -> em PtxOperand
57     namedConst :: PtxImmediate a =>
58         [PtxModifier] -> [Int] -> a -> String -> em PtxOperand
59
60 — Instructions PTX —
61 abs [ty] dst op | ty 'elem' [S16, S32, S64]
62     = instr [] Abs [ty] [dst, op]
63 add [ty] dst op1 op2 | ty 'elem' [U16, U32, U64, S16, S32, S64]
64     = instr [] Add [ty] [dst, op1, op2]
65 add [Sat, S32] dst@(Reg _) op1@(Reg _) op2@(Reg _)
66     = instr [] Add [Sat, S32] [dst, op1, op2]
67 — etc...

```

FIGURE 4.11 – MoC PTX

```

1 design = do
2   entree <- namedGlobal [F32] "x"
3   resultat <- namedGlobal [F32] "resultat"
4   x <- namedReg [F32] "x"
5   y <- namedReg [F32] "y"
6   z <- namedReg [F32] "z"
7   ld [Global, F32] x (Indirect entree 0)  — x := [entree]
8   mov [F32] y 0x40e00000  — y := 7.0
9   mad [F32] z x x y  — z := x^2 + y
10  st [Global, F32] (Indirect resultat 0) z — [resultat] := z
11  exit  — fin du noyau

1 .entry design
2 {
3 .global .f32 entree;
4 .global .f32 resultat;
5 .reg .f32 x, y, z;
6 ld.global.f32 x [entree]; // x := [entree]
7 mov.f32 y, 0x40e00000; // y := 7
8 mad.f32 z, x, x, y; // z := x^2 + y
9 st.global.f32 [resultat], z; // [resultat] := z
10 exit; // fin du noyau
11 }

```

FIGURE 4.12 – Programme trivial exprimé dans le MoC PTX (au dessus), comparé au fichier .ptx correspondant

4.14. (Il aurait fallu faire quelques accommodations liées à l'utilisation du type **Transfer** dans **Imperative** pour que les primitives de celui-ci puissent vraiment être utilisées de façon transparente. Par exemple, on aurait pu rendre la classe **Imperative** paramétrable par le type monadique représentant les expressions, de sorte à pouvoir passer **PtxOperand** comme arguments à **#=**, aux opérateurs arithmétiques, etc.) Pour rester concis, on a supposé dans le design donné en exemple que les fonctions réalisant le traitement lui-même (**calculerDonnees**, **chargerBuf**, **traiterBuf**, **ecrireBuf**) ont été préalablement écrites. On voit là aussi que le code résultant est assez intuitif. Une des parties les plus intéressantes des modèles de calcul proposés est la fonction **kernel**, qui prend en argument une fonction décrivant le noyau, et «renvoie» (place dans la monade) une fonction permettant de le lancer (à partir de la taille de la grille et des blocs, ainsi que des arguments propres au noyau). On notera également que la façon dont les pointeurs sont gérés (avec une distinction claire entre les espaces mémoire hôte et GPU) empêche les problèmes causés par le déréférencement de pointeurs vers la mémoire du GPU depuis l'hôte, ou *vice versa* (ce qui est encore pire). Il s'agit d'erreurs que **nvcc** (le compilateur CUDA) est incapable d'intercepter à la compilation avec son système de typage. Cet aspect est d'autant plus important que les erreurs de ce genre sont souvent graves, se traduisant au mieux par une erreur de segmentation, au pire par le plantage du driver de la carte vidéo (qui, tournant à un niveau de privilèges élevé, peut entraîner le reste du système avec lui).

4.2.2.2 Avec ordonnanceur

Les modèles de calcul vus précédemment peuvent être étendus afin de proposer des primitives de communication dont CUDA ne dispose pas de façon native (et qu'il n'est pas possible d'y définir facilement). Nous nous fixons l'objectif suivant : définir les fonctions **send** et **recieve**, permettant à un grille de threads CUDA d'envoyer et/ou de recevoir des données via la mémoire globale du GPU vers/depuis une autre grille de threads. Ces fonctions seront dans le principe similaires à **MPI_Send** et **MPI_Recv** de la bibliothèque MPI. Elles seront forcément bloquantes.

Pour y parvenir, un moteur implémentant ce MoC devra *a priori* mettre en place un ordon-

```

1  -----
2  -- Gestion des pointeurs --
3  -----
4  data HostPtr = HostPtr [PtxModifier] Integer -- mémoire hôte (type, adresse)
5  data DevicePtr = DevicePtr [PtxModifier] Integer -- mémoire GPU globale (type, adresse)
6  class Ptr a where resolvePtr :: a -> Either HostPtr DevicePtr
7  instance Ptr HostPtr where resolvePtr x = Left x
8  instance Ptr DevicePtr where resolvePtr x = Right x
9
10 -----
11 -- MoCs --
12 -----
13 class Imperative em => HostImperative em where
14     hostCalloc :: Integer -> [PtxModifier] -> em (HostPtr)
15     deviceCalloc :: Integer -> [PtxModifier] -> em (DevicePtr)
16     memcpy :: (Ptr a, Ptr b) => a -> b -> Integer -> em ()
17     memset :: Ptr a => a -> Char -> Integer -> em ()
18     free :: Ptr a => a -> em ()
19     kernel :: (forall em'. DeviceImperative em' => [DevicePtr] -> em' ())
20             -> em ([Integer] -> [Integer] -> [DevicePtr] -> em ())
21     texture :: [PtxModifier] -> [Integer] ->
22             (forall em'. DeviceImperative em' =>
23              em ( DevicePtr, [PtxOperand] -> PtxOperand -> em' ()))
24     -- etc...
25
26 class Imperative em => DeviceImperative em where
27     shared :: Integer -> [PtxModifier] -> em PtxOperand
28     register :: [PtxModifier] -> em PtxOperand
29     threadIdx :: em [PtxOperand]
30     blockIdx :: em [PtxOperand]
31     syncthreads :: em ()
32     ptx :: (forall em'. Ptx em' => em' ()) -> em ()
33     -- etc...

```

FIGURE 4.13 – Modèles de calcul pour la programmation impérative sur GPU

```

1  design' :: HostImperative em => em ()
2  design' = do
3      ker1 <- kernel $ \(donneesEntree, donneesSortie) -> do
4          bIdx <- blockIdx
5          tIdx <- threadIdx
6          buf <- shared 1024 [F32]
7          chargerBuf buf bIdx tIdx donneesEntree
8          syncthreads
9          traiterBuf buf tIdx
10         ecrireBuf buf bIdx tIdx donneesSortie
11     let grille = [64, 64]
12         bloc = [32, 32]
13         let taille = product (grille ++ bloc)
14             donneesHote <- hostCalloc taille [F32]
15             initialiserDonnees donneesHote
16             donneesGpuEntree <- deviceCalloc taille [F32]
17             donneesGpuSortie <- deviceCalloc taille [F32]
18             memcpy donneesGpuEntree donneesHote (taille*4)
19             ker1 grille bloc [donneesGpuEntree, donneesGpuSortie]
20             memcpy donneesHote donneesGpuSortie (taille*4)
21             sauvegarderDonnees donneesHote

```

FIGURE 4.14 – Exemple de programme dans ce modèle de calcul

```

1  data ThreadId = ThreadId Integer
2
3  class WithSendRecv em ptr where
4      send :: ptr -> Integer -> em ThreadId
5      recieve :: ptr -> Integer -> em ThreadId
6  class (WithSendRecv em DevicePtr, DeviceImperative em) => DeviceSuperthread em
7  class (WithSendRecv em HostPtr, HostImperative em) => HostSuperthread em
8
9  class HostImperative em => Scheduler em where
10     newThreadId :: em ThreadId
11     hostSuperthread :: ThreadId
12         -> (forall em'. HostSuperthread em' => em' ()) -> em ()
13     deviceSuperthread :: ThreadId
14         -> (forall em'. DeviceSuperthread em' => em' ()) -> em ()

```

FIGURE 4.15 – Modèle de calcul pour la programmation sur GPU avec primitives de communication entre grilles et ordonnancement

nanceur sur l'hôte capable de lancer les grilles de threads tour à tour, en fonction des données que certaines attendent et que d'autres émettent. Chaque grille de threads devra en plus sauvegarder son état au moment de chaque `recieve` pour être capable de reprendre son exécution au point où elle s'est arrêtée. Il est facile de comprendre pourquoi ce genre de chose n'est pas faisable directement et élégamment avec CUDA : cela nécessite un support d'exécution (*runtime*) situé un cran au dessus de la couche d'abstraction que CUDA propose.

La définition du modèle de calcul correspondant est partiellement donnée par la figure 4.15. La classe `WithSendRecv` est destinée à être commune à tous les MoCs dans lesquels envoyer et recevoir des données présente un sens. En plus de la monade associée au MoC, elle est paramétrée par `ptr`, un type représentant les pointeurs dans le MoC en question, et `handle`, qui y désigne une entité capable d'émettre ou de recevoir des données. La fonction `send` prend en argument un pointeur vers les données à envoyer (et leur taille) ainsi que le destinataire, tandis que `recieve` a pour paramètre l'adresse et la taille du buffer où stocker les données reçues. On définit le type concret `SuperthreadId`, qui va nous permettre d'identifier nos threads communicants (que l'on appelle superthreads, donc). Les MoCs utilisés par les superthreads sont `HostSuperthread` et `DeviceSuperthread`, et sont en fait ni plus ni moins que `HostImperative` et `DeviceImperative` enrichis de la classe `WithSendRecv`. La totalité de superthreads doit être déclarée dans le MoC `Scheduler`. Pour chaque superthread, on allouera d'abord un identifiant via `newSuperthreadId`, avant d'en définir le contenu par le biais de `hostSuperthread` ou `deviceSuperthread`. L'intérêt de procéder ainsi est d'utiliser `send` pouvoir communiquer avec des superthreads dont le contenu n'a pas encore été défini, ce qui n'aurait pas été possible si on avait, par exemple, renvoyé l'identifiant au moment de la définition.

4.2.3 Gains apportés

En somme, l'utilisation de notre approche pour la programmation sur GPU offre des gains à deux niveaux :

- La possibilité de définir de nouveaux modèles de calcul permet de potentiellement gagner en productivité
- L'utilisation d'alternatives et de l'exploration architecturale permet d'obtenir des performances élevées plus rapidement et plus simplement qu'en confiant le profiling et l'optimisation au programmeur

Il est important de noter que l'utilisation d'alternatives peut être tout à fait transparente pour l'utilisateur. Supposons par exemple que nous définissions les types abstraits `ReadBuffer` et `WriteBuffer`, représentant des tampons servant en lecture et en écriture, respectivement. On donne quelques fonctions permettant de lire/écrire ces tampons depuis le GPU, les remplir ou

récupérer leur contenu depuis l'hôte, etc. La nature de la mémoire utilisée pour chaque buffer peut être choisie par le biais d'alternatives, ce qui se traduira par des performances radicalement différentes. Malgré tout, cela reste transparent pour l'utilisateur, qui se contente de lire et écrire dans des tampons dont il ignore la nature exacte. Il en va de même des autres aspects de la programmation sur GPU : la taille de la grille, le déroulement de boucles, voire le stockage des variables individuelles en registres, mémoire partagée ou mémoire locale, peuvent être décidés au moment de l'exploration architecturale. Tout cela en offrant à l'utilisateur des abstractions faciles à utiliser et les structures de contrôle dont il a l'habitude.

Nous avons prototypé certaines de ces abstractions sur un exemple simple de rendu par lancer de rayons sur une scène codée en dur. Les paramètres explorables étaient la taille de la grille et des blocs, et l'organisation du buffer où est stocké le rendu. À chaque thread correspond le calcul d'un pixel, et la fonction d'écriture de la valeur calculée peut soit se contenter de réaliser directement un *store* dans la mémoire globale, soit d'allouer un tampon dans la mémoire partagée et faire coopérer les threads d'un warp de sorte à écrire en burst dans la SDRAM ensuite. (Il est intéressant de noter que les règles régissant les accès en burst changent de génération en génération sur les GPU nVidia ; rendre cet aspect explorable est donc particulièrement utile.) Le résultat d'un recuit simulé sur cet exemple de programme est donné sur la figure 4.16. On cherche à minimiser la durée du calcul de 50 images d'une animation en 1920×1080 sur une carte de type GeForce GTX 285. La paramétrage initial (obtenu de façon complètement aléatoire) correspond à une durée de calcul 6737 ms, soit environ 7,5 images par seconde. À la quatrième itération, on obtient un paramétrage donnant un temps total de 230 ms (soit 217 images par seconde). Elle ne sera battue qu'à la 47^{ème} itération, avec 215 ms (ou 232 images par seconde). On notera qu'il y a de nombreuses itérations sans score. Cela est dû au fait qu'un paramétrage peut être invalide (par exemple, en arrivant à court de mémoire partagée, ou en consommant plus de registres que la taille d'un bloc le permet). Ce n'est pas trop gênant ici, dans la mesure où une itération (correspondant à la génération du code C et PTX intermédiaire, sa compilation et son exécution) ne dure que quelques secondes. De plus, cela aurait pu être complètement évité en programmant et utilisant un moteur conscient de la quantité de ressources disponibles dans le GPU. Au final, on retiendra le rapport 30 au niveau du temps de calcul entre un paramétrage aléatoire et l'optimum trouvé, ce qui est considérable et justifie de l'intérêt de l'approche dans ce contexte.

4.3 Instanciation de réseaux d'interconnexions

Du fait de l'augmentation massive du nombre de transistors par puce au cours des dernières décennies, les réseaux d'interconnexions deviennent progressivement une composante de plus en plus critique d'un SoC. Les bus, sur lesquels seule une des interfaces connectée peut émettre à un instant donné, perdent en pertinence à mesure que le nombre de blocs à interconnecter croît. Une parade à ce phénomène est d'organiser les bus en hiérarchies au moyen de bridges. Une autre parade possible est d'utiliser des réseaux sur puce [16], constitués de switches raccordés par des liaisons point à point. Quelque soit l'approche choisie, l'infrastructure d'interconnexion se voit nettement complexifiée par rapport à un bus unique à l'échelle du système. La latence est typiquement plus élevée, et des phénomènes nouveaux apparaissent, tels que les éventualités de deadlock ou de congestion localisée, la possibilité pour les paquets d'arriver dans le désordre, etc. Les NoCs, en particulier, peuvent exhiber une grande variété de comportements. Le routage de paquets peut être statique ou dynamique, et les switches peuvent adopter différentes politiques pour se conformer à des contraintes de latence ou de qualité de service (QoS) [16]. Le nombre de paramètres à explorer est considérable : en plus du placement de blocs dans le réseau (ou sur les bus constituant la hiérarchie), il convient de décider des paramètres internes des switches ou des bridges (taille des FIFOs, comportement temporel, etc.). La configuration optimale d'un bloc donné varie en fonction de celle du réseau d'interconnexions à partir du moment où celui-ci communique avec d'autres blocs. Il est donc important de mener à bien l'exploration de la totalité du système simultanément.

Nous pouvons proposer un modèle de calcul destiné à la description de réseaux d'intercon-

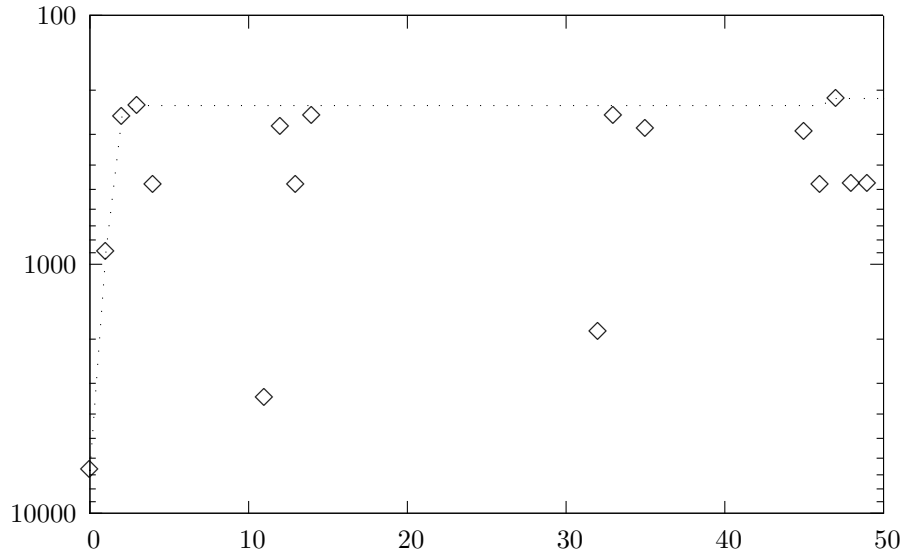


FIGURE 4.16 – Exploration par recuit simulé d’une programme de rendu graphique sur GPU. Itération en abscisse, durée du traitement (rendu de 50 images, en ms) en ordonnée. En points tillés : meilleur score obtenu avant une itération donnée.

```

1  — NoC —
2  data NodeId = NodeId [Integer] — interprétation libre (par ex., position)
3  data Interface = Interface
4
5  class (WithSendRecv em (Transfer Type), RTL em) => RTLNode em
6  class (WithSendRecv em (Transfer Type), Imperative em) => ImperativeNode em
7
8  class Explorable em => NoC em where
9      newNodeId :: em NodeId
10     rtlNode :: NodeId
11         -> (forall em'. RTLNode em' => em' ()) -> em ()
12     imperativeNode :: NodeId
13         -> (forall em'. ImperativeNode em' => em'()) -> em ()
14
15  class NoC em => MatrixNoC em where
16     newNodeIdFromPosition :: (Integer, Integer) -> em (Maybe NodeId)
17     unoccupiedPositions :: em [(Integer, Integer)]
18
19  — Propriétés additionnelles —
20  class NoC em => WithLatencyConstraints em where
21     guaranteeLatency :: NodeId -> NodeId -> Integer -> em ()
22     guaranteeOverallLatency :: Integer -> em ()
23
24  class NoC em => WithInOrderPacketDelivery em

```

FIGURE 4.17 – Organisation possible de classes pour la modélisation de NoCs

nexions. Il semble par exemple judicieux de réutiliser la classe `WithSendRecv`, définie à la section précédente, ainsi que le MoCs `Imperative` et `RTL` du chapitre 3. Comme on peut le voir sur la figure 4.17, on reprend un fonctionnement très proche du celui proposé à la section 4.2.2.2 : chaque nœud (c.-à-d. bloc connecté) dispose d'un identifiant, et est défini en deux temps : d'abord par l'allocation de l'identifiant (`newNodeId`), puis par la définition du comportement du bloc proprement dit (`rtlNode`, `imperativeNode`). Par défaut, la position du bloc dans le réseau n'est pas spécifiée (et sera donc déterminée durant l'exploration), mais il est possible pour des topologies de réseaux spécifiques de laisser le designer positionner certains blocs à des emplacements précis. Par exemple, le moteur d'un NoC organisé sous forme de matrice instanciera `MatrixNoC`, et pourra utiliser `newNodeIdFromPosition` pour forcer l'emplacement d'un bloc donné. Similairement, les propriétés spécifiques dont on veut que le réseau d'interconnexions dispose peuvent simplement être ajoutées par l'intermédiaire de classes additionnelles disposant des primitives adéquates (exemple : `WithLatencyConstraints`). On peut aussi déclarer des classes vides exprimant juste une propriété intrinsèque du réseau (exemple : `WithInOrderPacketDelivery`).

Les moteurs instanciant ces MoCs s'occupent de construire les NoCs à proprement parler. Bien que chacune des classes que nous venons de voir offre des primitives relativement élémentaires et faibles en nombre, en définir suffisamment permet d'exprimer les propriétés les plus diverses. Il en résulte que les blocs simples, ne nécessitant que les services de base offerts par une infrastructure de communication, pourront se voir instanciés au sein d'une variété d'architectures d'interconnexions, tout en permettant à ceux reposant sur des propriétés plus spécifiques d'y avoir recours. Cet aspect va également dans le sens de l'extensibilité qui était un de nos objectifs de départ.

4.4 Conclusion

Ce chapitre a présenté diverses façons de s'attaquer à certains des problèmes architecturaux propres aux designs modernes. A chaque fois, nous avons surtout insisté sur la conception des modèles de calcul dont le designer pourrait faire usage pour résoudre certains problèmes spécifiques, sans rentrer dans le détail des moteurs sous-jacents comme nous l'avons fait au chapitre 3. Néanmoins, il a été possible d'observer certaines des caractéristiques importantes de notre approche, à savoir sa versatilité, son extensibilité, son caractère intuitif à l'utilisation. Nous avons également souligné autant que possible l'intérêt bien réel de l'exploration architecturale dans chacun des domaines étudiés.

4.4. CONCLUSION

Conclusion générale

La conception niveau système est un domaine difficile mais crucial. Les méthodes conventionnelles commençant à montrer leurs limites lorsqu'il s'agit de résoudre les problématiques qui y sont propres, on peut observer un effort de recherche important dans le développement de stratégies alternatives. L'ensemble des propositions étayées dans le cadre de cette thèse s'inscrit dans cet effort.

La méthodologie pour la conception niveau système basée sur la programmation fonctionnelle qui a été présentée dans ce manuscrit dispose d'atouts de taille. Elle permet de gérer de façon native et élégante la multiplicité de niveaux d'abstractions qui résulte de l'hétérogénéité des composants constituant les SoCs et SoPCs modernes, et est dotée de capacités d'exploration architecturale particulièrement utiles pour, par exemple, configurer un composant réutilisé depuis un autre design de sorte à ce qu'il présente des performances convenables dans son nouveau contexte. Nous avons vu sur une étude de cas complète les gains offerts par l'approche proposée, ainsi qu'un petit ensemble de pistes permettant de l'appliquer à la résolution de divers problèmes architecturaux. Cela nous a permis de constater que notre méthode pouvait se généraliser au design orienté plateforme (avec, par exemple, la programmation sur GPGPU), et implémenter des tâches traditionnellement réservées à des outils séparés venant se greffer sur le flot de conception (synthèse de pipelines, instanciation de réseaux d'interconnexions). Ce dernier aspect est particulièrement notable dans la mesure où la multiplicité des outils, et plus généralement la complexité du flot de développement, peut être source de problèmes pour le designer.

Dans la mesure où la méthode proposée est basée sur ce que l'on qualifie généralement de langage embarqué, elle hérite des avantages et des inconvénients qui sont liés à cette catégorie d'approches. L'utilisation d'un langage générique tel que Haskell permet une prise en main facilitée si le designer est déjà familier avec la syntaxe et la sémantique de ce langage, et l'utilisation de compilateurs existants assure des performances convenables et la compatibilité avec des bibliothèques déjà écrites. Enfin, on bénéficie des traits propres à Haskell comme son système de classes, son aisance dans la manipulation de monades et son caractère paresseux. Certains de ces aspects sont à double tranchant (par exemple, l'évaluation paresseuse offre un grain en expressivité et potentiellement en performances au prix d'un débogage plus compliqué qu'avec une évaluation stricte) mais dans tous les cas ont fait et font l'objet de recherches et sont donc de mieux en mieux compris des programmeurs (c.f. par exemple [7, 67, 76, 77] pour la problématique du débogage). Haskell est un langage vivant qui continue à être étendu et progressivement standardisé, et offre de perspectives d'évolution intéressantes. En particulier, à long terme, on peut espérer y voir implémentés des éléments de vérification formelle par le biais de son système de types (comme le laisse présager [65], par exemple).

Une caractéristique intrinsèque de notre approche est l'accent qu'elle met sur la communication et l'interaction entre modèles de calcul. Une fois dotée d'un nombre significatif de modèles de calcul complémentaires, elle permettrait au concepteur d'accroître son confort de travail et sa productivité en lui permettant de spécifier chaque élément du design dans le paradigme qui convient le mieux, et ce sans handicaper ou complexifier la communication entre composants. Les mécaniques nécessaires pour cela sont en place et ont été illustrées dans leurs principes de fonctionnement et d'utilisation aux chapitres 2 et 3. Une des perspectives majeures du travail réalisé est donc la mise en place d'une riche bibliothèque de modèles de calculs mutuellement complémentaires permettant de mettre en évidence l'utilité de ces mécaniques pour la conception de SoCs complexes.

CONCLUSION GÉNÉRALE

Annexe A

Syntaxe de Haskell

Cette annexe est une introduction à Haskell permettant de comprendre le code présent tout au long du manuscrit. Elle se limite aux bases du langage. Le lecteur curieux pourra se tourner vers [40], un didacticiel complet et bien fait, voire vers [48], la dernière révision du rapport officiel qui définit le langage Haskell 98.

A.1 Bases

Haskell est un langage fonctionnel pur moderne, fortement et statiquement typé. Il présente un grand nombre de caractéristiques communes et similarités de syntaxe avec d'autres langages fonctionnels typés tels que Standard ML [62] ou Caml [54], mais comporte aussi son lot d'originalités, telles que son système de classes de types qui sera détaillé dans la section A.3. Haskell est un langage non-strict reposant sur l'évaluation paresseuse, et est probablement l'exemple le plus connu de langage de cette catégorie.

Comme tout programme purement fonctionnel, un programme Haskell doit être vu comme un ensemble d'équations menant au résultat désiré, plutôt qu'une suite d'instructions permettant de le calculer comme c'est le cas dans les langages impératifs. Haskell propose un système d'inférence de types qui repose sur une variante de l'algorithme de Hindley-Milner [21], et qui permet souvent – mais pas toujours – de se dispenser de spécifier les types des objets manipulés par les fonctions définissant le programme. Un exemple de programme Haskell trivial serait :

```
1 g :: Integer -> Integer -> Integer
2 g x y = x * x - y
3
4 f x = g x 2 + 5
5
6 main = putStr (show (f 15) ++ "\n")
```

L'opérateur binaire ++ permet de concaténer ses opérandes.

On peut distinguer deux types de déclarations :

- les équations, composées d'un signe égal séparant l'identifiant déclaré de l'expression qui lui est associée
- les signatures, indiquant explicitement le type d'un symbole, et composées d'un «::» séparant l'identifiant dont le type est explicité du type lui même

Les noms des types (comme **Integer**) commencent forcément par une majuscule, tandis que les identifiants ordinaires (comme **g** ou **f**) et les variables muettes (comme **x** ou **y**) commencent par une minuscule. Les identifiants composés de plusieurs mots s'écrivent en suivant la même convention qu'en Java, c'est-à-dire en capitalisant la première lettre de chaque nouveau mot, sans séparer les mots par des espaces (c.f. fonction **putStr** dans l'exemple), mais ceci n'est pas obligatoire, et il est tout à fait possible d'utiliser des underscores sans que le compilateur ne génère de messages d'erreur. Comme en Python, l'indentation compte !

Un opérateur fréquent dans les signatures est `->` ; le type `a -> b` désigne une fonction prenant en argument une expression du type `a` et s'évaluant comme une expression du type `b`. Cet opérateur est associatif de droite à gauche ; ainsi, le type de `g` de notre exemple doit se lire `Integer -> (Integer -> Integer)`, c'est à dire fonction prenant en argument un entier et renvoyant une fonction prenant en argument un entier et renvoyant un entier. On parle de fonction *curryfiée* à deux arguments ; une version *non curryfiée* de la même fonction aurait eu pour signature `(Integer, Integer) -> Integer` (prend une paire d'entiers, renvoie un entier). Les fonctions curryfiées sont la norme plutôt que l'exception dans un programme Haskell. Elles présentent l'avantage de pouvoir être évaluées partiellement. Par exemple, l'expression `g 5` est une fonction valide (dite «clôture») qui a pour signature `Integer -> Integer`.

Généralement, une expression écrite en Haskell est assez intuitive à lire. Les fonctions sont appliquées à leurs arguments en écrivant le nom de la fonction suivi de ses arguments. Les parenthèses sont facultatives. L'application de fonction à une priorité supérieure à celle de tous les autres opérateurs. Elle est naturellement associative de gauche à droite pour permettre une évaluation facile des fonctions curryfiées. Il est possible de décomposer une expression complexe en un certain nombre de déclarations locales entrant dans la composition d'une expression plus simple. Cela permet d'éviter de polluer l'espace de nommage global avec des identifiants qui ne seront utilisés que dans l'évaluation d'un nombre réduit d'expressions. Les constructions autorisant ce genre de pratique sont `let [déclarations] in [expression]`, ainsi que `[expression] where [déclarations]`. Supposons, dans l'exemple précédent, que `g` n'est utilisée que dans l'évaluation de `f`, et que `f` n'est pas utilisée ailleurs que dans `main`. On peut écrire :

```
1 main = let f x = g x 2 + 5 where g x y = x * x - y
2       in putStrLn (show (f 15) ++ "\n")
```

Terminons sur une modification de ce même exemple pour illustrer les expressions lambda. Comme une fonction est une expression comme une autre, il est possible de l'écrire directement sous forme d'expression. L'extrait de code suivant est rigoureusement équivalent au précédent :

```
1 main = let f = \x -> g x 2 + 5 where g = \x y -> x * x - y
2       in putStrLn (show (f 15) ++ "\n")
```

En fait, l'écriture que nous avons utilisée jusqu'ici, et qui plaçait les variables muettes à gauche du symbole égal dans les déclarations, est une écriture abusive. Elle se justifie par son caractère intuitif, et par sa similarité syntaxique avec la déclaration de types de données polymorphes (comme nous le verrons à la section A.2).¹

Un autre raccourci d'écriture courant allant dans le même sens est l'inclusion de conditions du coté gauche du symbole égal dans la définition d'une fonction. Ainsi, dans l'exemple suivant, `f` et `f'` sont exactement équivalentes :

```
1 f x = if x < 5 then 7 else 12
2
3 f' x | x < 5 = 7
4 f' x = 12
```

A.2 Filtrage par motif

Haskell dispose d'un système de typage riche en possibilités. Outre le mot clé `type`, permettant de définir un alias d'un autre type à la manière d'un `typedef` en C, l'utilisateur dispose de la construction `data` pour définir ses propres types algébriques de données. Un type défini par ce biais là est caractérisé par un nom et un certain nombre de constructeurs. Il peut également être

1. Par ailleurs, les habitués du lambda-calcul pourront remarquer que la syntaxe des lambda-termes en Haskell est très proche de la notation lambda couramment utilisée en informatique théorique : il suffit en effet de remplacer le caractère `<<\>` par un `<<λ>`, et l'opérateur `<<->` par un point.

paramétré par un ou plusieurs autres. L'exemple que l'on prend souvent pour illustrer tout ceci est la liste chaînée :

```
1 data Liste a = ListeVide | Noeud a (Liste a)
2
3 tete xs = case xs of Noeud t q -> t — renvoie le 1er élément
4 queue xs = case xs of Noeud t q -> q — renvoie tout sauf le 1er élément
```

Le type `Liste` est paramétré par un type représenté par la variable de type `a` (on dit que `Liste` est polymorphe). Contrairement aux types eux-même et par similarité avec les variables ordinaires, les variables de type commencent par une minuscule. Ici, `a` correspond au type des expressions dont on souhaite faire une liste. La syntaxe de paramétrage des types est identique à la syntaxe d'application de fonctions.

Le type `Liste` dispose de deux constructeurs de données :

- `ListeVide`, équivalent à un pointeur nul dans les langages qui proposent ce concept
- `Noeud`, contenant une tête de liste de type `a` et une queue de liste de type `Liste a` (la tête d'une liste est son premier élément, tandis que sa queue correspond à tous ses éléments à l'exception de sa tête)

La construction `case` permet de réaliser un filtrage par motif sur une expression. Cela permet de déterminer le constructeur et ses arguments qui ont été utilisés pour obtenir une expression donnée. À chaque argument d'un constructeur, l'utilisateur peut affecter une variable muette, un «joker» (caractère «`<`», se comportant comme une variable muette sans nom, comme nous le verrons dans l'exemple suivant), ou faire correspondre récursivement un autre motif.

Il est fréquent qu'un type de données ne comportant qu'un seul constructeur porte le même nom que celui-ci. Ce n'est pas gênant ! Les noms de types et ceux des constructeurs appartiennent à deux espaces de noms différents, et aucune collision n'en résulte. Il est en général clair d'après le contexte que l'on fait référence au type ou à son constructeur.

Comme souvent en Haskell, il existe des raccourcis syntaxiques permettant d'arriver aux mêmes fins qu'une construction `case`, au point que voir le mot clé `case` dans un programme Haskell est plutôt rare. D'ores et déjà, les arguments donnés lors de la définition d'une fonction, qu'ils soient à gauche du symbole égal ou à gauche de la flèche dans une lambda expression, sont en fait des motifs. Il en résulte qu'il est possible d'écrire :

```
1 data Liste a = ListeVide | Noeud a (Liste a)
2
3 tete (Noeud t _) = t
4 queue (Noeud _ q) = q
```

Il est même possible de donner des «noms» aux «champs» d'un type algébrique. Cela nous conduit au code suivant, très concis mais strictement équivalent aux deux écritures précédentes :

```
1 data Liste a = ListeVide | Noeud { tete :: a, queue :: Liste a }
```

En fait, cette écriture définit implicitement les deux fonctions `tete` et `queue`. Il en résulte que deux types de données ne peuvent pas avoir des champs qui portent le même nom, ce qui peut de prime abord étonner un habitué des langages structurés classiques.

Soulignons qu'il existe un type liste chaînée prédéfini, et bénéficiant d'une syntaxe un peu inhabituelle, mais sinon en tout point similaire à celui que nous venons de définir. Bien que l'exemple suivant soit invalide en Haskell, ce pseudo-code permet de bien comprendre la syntaxe associée à l'utilisation de ce type prédéfini :

```
1 data [a] = [] | a:[a]
```

On voit que «`[]`» est l'équivalent de `ListeVide`, et que «`:`» est un constructeur à deux arguments se comportant comme `Noeud`, à la différence près qu'il s'utilise de manière infixé. Insistons encore que le code précédent est invalide, ne serait-ce que du fait que les opérateurs d'arité nulle tels que «`[]`» ne peuvent être définis par l'utilisateur ; il permet cependant de bien comprendre le

comportement du type prédéfini `[a]`, correspondant aux listes simplement chaînées d'éléments de type `a`. Ce type comporte également un nombre conséquent de fonctions associées, qu'il est cette fois possible de réimplémenter sans aucun artifice syntaxique ; par exemple :

```

1 head (x:_) = x
2 tail (_:xs) = xs
3
4 map _ [] = []
5 map f (x:xs) = (f x):(map f xs)
6
7 length [] = 0
8 length (_:xs) = 1 + (length xs)

```

Les fonctions `head`, `tail`, `map`, `length` et bien d'autres font partie du *prélude*, la bibliothèque standard de Haskell 98, dont on parlera un peu plus en détail dans la section A.5. La tête et la queue d'une liste sont obtenues en appelant `head` et `tail` dessus, tandis que `map` y applique une fonction terme à terme, et `length` permet d'en connaître la longueur.

A.3 Polymorphisme

La philosophie derrière Haskell est d'exploiter au maximum le polymorphisme des types comme des fonctions. Pour cela, le langage dispose d'un système de classes de types puissant et original. Une classe de type Haskell ressemble par certains aspects au concept de classe que l'on utilise en programmation orientée objet (POO), mais son utilisation est sujette à d'importantes différences. Le polymorphisme en Haskell est en effet statique, et non dynamique. Prenons un exemple :

```

1 class Fusionnable a where
2     fusionner :: a -> a -> a
3
4 instance Fusionnable [a] where
5     fusionner x y = x ++ y — concatener x à y
6
7 instance Fusionnable Integer where
8     fusionner x y = x * y
9
10 fusionnerListe :: Fusionnable a => [a] -> a — fonction polymorphe
11 fusionnerListe [x] = x
12 fusionnerListe (x:xs) = fusionner x (fusionnerListe xs)

```

Ici, la classe `Fusionnable` définit la méthode `fusionner`, fusionnant d'une façon ou d'une autre ses deux arguments. Pour chaque type pour lequel on estime que le concept de fusion a un sens, on définit une instance de la classe `Fusionnable`. Par exemple, fusionner deux listes les concatène, tandis que fusionner deux nombres les multiplie. On peut définir ensuite des fonctions polymorphes, telles que `fusionnerListe`, qui, à partir d'une liste d'objets appartenant à la classe `Fusionnable`, les réduit de droite à gauche par le biais de la fonction `fusionner`. Ainsi, si on lui passe en argument une liste de listes, elles sont toutes concaténées entre elles ; si on lui passe en argument une liste d'entiers, c'est le produit de tous ces nombres qui est renvoyé.

Nous avons introduit un nouvel opérateur, `=>`, dont le rôle est de permettre de spécifier l'appartenance à certaines classes des variables de type. On peut l'utiliser dans les signatures de fonctions telles que `fusionnerListe`, pour expliciter leur caractère polymorphe. C'est généralement obligatoire : l'algorithme d'inférence de types de Hindley-Milner sur lequel se base Haskell ne s'applique pas aux constructions polymorphes, donc le langage a tendance à «surspécialiser» (c.-à-d. restreindre à des types concrets précis) les fonctions qui auraient pu être polymorphes mais dont la signature a été omise. On utilise également `<=>` dans les définitions des classes, toujours pour qualifier les variables de type utilisées. En particulier, si une déclaration de classe commence par `class ClasseX a => ClasseY a where ...`, on dira que la classe `ClasseY` dérive de la `ClasseX` (comme on le ferait en POO). On pourra noter la possibilité d'utiliser le mot clé `forall` pour

rendre la quantification universelle induite par l'opérateur `=>` explicite. Ainsi, `forall a. Show a => a` est équivalent au légèrement plus court `Show a => a`.

Insistons une fois de plus sur le caractère statique du polymorphisme. Il est impératif que le compilateur puisse déterminer au moment de la compilation quelle instance de classe sera utilisée au moment de l'appel d'une fonction polymorphe. Il n'y a pas de table de fonctions virtuelles, ni de concept similaire, comme c'est le cas en Java, C++, etc. Il n'est par exemple donc pas possible d'utiliser la classe `Fusionnable` comme un type – les noms de classes restent toujours à gauche du symbole `=>` dans une signatures. Il en découle qu'il n'est pas non plus possible de construire une liste chaînée comportant des éléments de types hétérogènes ayant pour point commun d'appartenir à la classe `Fusionnable`². Cela peut de prime abord dérouter les habitués de la POO.

A.4 Monades

Une monade en programmation fonctionnelle est un type de données représentant une séquence d'actions. Une définition simplifiée mais suffisante serait :

```
1 class Monad m where
2     (>>=) :: m a -> (a -> m b) -> m b
3     return :: a -> m a
```

Une monade «contient» une valeur³. L'opérateur `>>=` se lit «bind», et sert à construire une monade de type `m` contenant une valeur de type `b` à partir d'une monade du même type `m` contenant cette fois une valeur de type `a`, et d'une fonction capable de transformer cette valeur en une monade toujours du type `m` et contenant une valeur de type `b`. La fonction `return`, quant à elle, place une valeur dans une monade. Elle doit son nom à la similarité qu'elle partage avec le mot clé `return` des langages impératifs lorsque la monade considérée ressemble à une monade d'état (voir section 2.2.1). Attention : un `return` ne met pas fin à l'évaluation de la monade!

La sémantique du `bind` et du `return` varie grandement suivant l'instance de la classe `Monad` que l'on est en train de définir. Cela dit, il existe des règles que toute monade définie doit⁴ suivre⁵.

Fort heureusement, les monades sont plus faciles à utiliser qu'à définir ou qu'à comprendre. On se sert généralement de la notation `do` pour expliciter un enchaînement d'actions définissant une expression monadique, ce qui présente l'avantage de cacher l'opérateur `>>=`. On n'utilisera ces derniers que pour définir des séquences d'actions extrêmement courtes tout au plus, et de façon très occasionnelle. Dans un bloc `do`, chaque action est séparée par un `<;>` ou un retour à la ligne. L'expression résultante a une valeur identique à celle que l'on aurait obtenu si on avait utilisé l'opérateur `>>=`, des parenthèses et des lambda expressions pour arriver aux mêmes fins. Dans l'exemple suivant, `m` et `m'` sont équivalentes :

```
1 m = [1, 2, 3] >>= (\x -> [4, 5, 6] >>= (\y -> return (x * y)))
2
3 m' = do
4     x <- [1, 2, 3]
5     y <- [4, 5, 6]
6     return (x * y)
7
```

2. Un résultat proche peut tout de même être obtenu par le biais d'une technique appelée quantification existentielle [56], mais cela dépasse le cadre de cette introduction.

3. ...qui est représentée par la variable de type `a` aux lignes 2 et 3. Le type représenté par la variable `m` est donc polymorphe, et admet exactement un type en argument ; par analogie avec le *type* des fonctions, on dit que le *kind* de `m` est `* -> *` ; un type non polymorphe a toujours pour *kind* `*`, par exemple.

4. Ou plutôt devrait. Il existe une poignée d'exceptions qu'il n'est pas impératif de connaître.

5. À savoir :

- L'expression `return x >>= k` doit toujours valoir `k x`, et l'expression `m >>= return` doit toujours valoir `m` (intuitivement : `return` ne fait rien sinon placer son argument dans la monade)
- L'expression `(m >>= k) >>= h` doit toujours valoir `m >>= (\x -> k x >>= h)` (intuitivement : l'enchaînement d'actions est associatif)

```
8 main = do
9     putStr ((show m)++)"\n"
10    putStr ((show m')++)"\n"
```

La notation `do` est nettement plus lisible et facile à manipuler, surtout lorsque le nombre d'actions enchaînées est élevé. Dans cet exemple, nous avons utilisé le type `[Integer]` en guise de monade. En effet, le type liste prédéfini `[a]` est monadique : un `return` crée un singleton (liste d'un élément) contenant la valeur qui lui a été passée en argument, tandis qu'un `bind` évalue sa seconde opérande pour tous les éléments de la première, et concatène les listes ainsi obtenues entre elles. Il en résulte que `m`, tout comme `m'`, vaut `[4,5,6,8,10,12,12,15,18]`, c'est à dire `[1*4, 1*5, 1*6, 2*4, 2*5, 2*6, 3*4, 3*5, 3*6]`.

On peut également voir un `do` au niveau de la fonction `main`. En effet, comme les entrées et sorties qu'effectue un programme peuvent être vues comme une séquence d'actions, la bibliothèque standard de Haskell prévoit une monade (appelée `IO`) pour les formaliser.

Il est à noter qu'un `do` peut contenir des constructions `let` (sans `in`), qui ne sont pas considérées comme des actions. Les identifiants définis par le biais de ces `let` se comportent exactement comme s'il avaient été définis dans la première partie d'un `let [...] in do [...]`. Des exemples de cas de figure parsèment le manuscrit.

A.5 Prélude

Le prélude est la partie de bibliothèque standard de Haskell qui est automatiquement importée dans un programme, sans que le programmeur n'ait besoin d'écrire quoi que ce soit. Les types de données et fonctions qui y sont définies concernent l'arithmétique, les listes chaînées et chaînes de caractères, et les monades dont la monade d'entrées/sorties par laquelle le programme peut communiquer avec le monde extérieur. Cette section en fait une liste non exhaustive.

A.5.1 Arithmétique

Le prélude définit une hiérarchie de classes très riche dédiées à l'arithmétique. La figure A.1, empruntée à [3] (attention, l'original comporte quelques erreurs), énumère ces classes ainsi que leurs instances définies dans le prélude. Dans les exemples utilisés jusqu'ici dans cette annexe, nous avons utilisé le type `Integer` pour représenter les nombres entiers. Il s'agit, comme on peut le voir sur la figure, d'un type à précision arbitraire. Souvent, on préfère le type `Int`, qui a une dynamique limitée mais offre de meilleures performances.

Parmi les classes importantes à retenir, citons `Eq` et `Ord`, correspondant aux types disposant d'une relation égalité et de relations de comparaison, respectivement. Les types `Float` et `Double` sont également importants. Comme leur nom le laisse supposer, ils encapsulent des nombres en virgule flottante suivant la norme IEEE 754.

De nombreuses fonctions, telles que `fromInteger`, `fromEnum`, `toInteger`, `toEnum`, etc., font qu'il est plutôt facile de passer d'un type à un autre, qu'il s'agisse de types polymorphes ou non. Les conversions doivent néanmoins toujours être explicites. Le revers de la médaille de cette aisance de conversion est qu'il est difficile d'incorporer dans cette hiérarchie des types qui ne peuvent pas être immédiatement évalués comme des nombres. Cela inclut en particuliers des types représentant des arbres syntaxiques. Nous nous intéresserons à ce problème dans le chapitre 3.

A.5.2 Listes et chaînes de caractères

Le type des chaînes de caractères en Haskell est `String`, qui est en fait un simple alias pour `[Char]`, une liste simplement chaînée de caractères. Il en résulte que toutes les fonctions qui marchent sur les listes en général marchent sur les chaînes de caractères en particulier.

Parmi ces fonctions, citons juste :

- `head` et `tail`, dont nous avons vu une implémentation possible à la section A.2
- `length :: [a] -> Int`, la longueur d'une liste (voir exemple de définition à la section A.2)

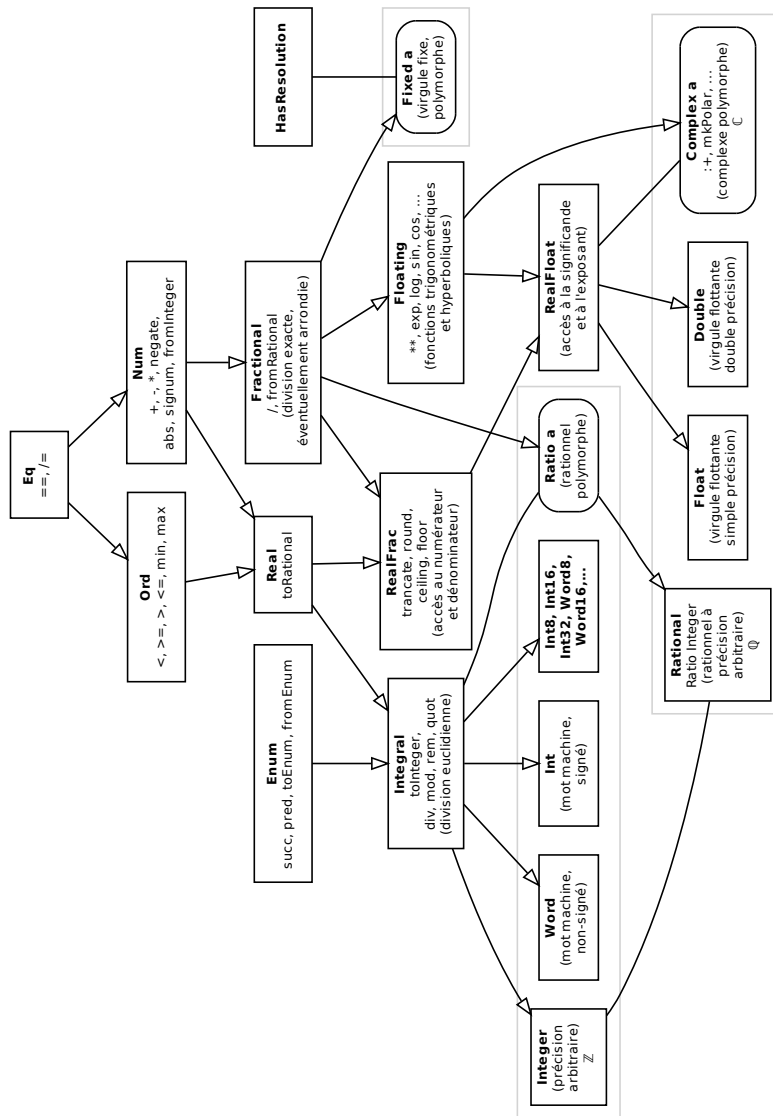


FIGURE A.1 – Hiérarchie des classes dédiées à l'arithmétique et leurs instances. Les instances sont incluses dans les rectangles gris clairs. Les types polymorphes sont ceux dont les rectangles sont arrondis. Une flèche entre deux instances désigne une spécialisation d'un type polymorphe via un alias défini par le mot clé `type`.

- `map :: (a -> b) -> [a] -> [b]`, qui applique son premier argument sur chaque élément du deuxième pour produire la liste renvoyée (voir exemple de définition à la section A.2)
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`, qui applique une fonction terme à terme sur deux listes. Par exemple, `zipWith (*) [1..3] [4..6]` vaudra `[4, 10, 18]`. L'expression `(*)` est un raccourci syntaxique valant `\x y -> x * y`. Cette écriture fonctionne avec tous les opérateurs ordinaires.
- `zip`, qui vaut `zipWith (,)`
- `foldr` et `foldl`, qui effectuent une réduction de droite à gauche et de gauche à droite, respectivement ; leurs arguments sont la fonction de réduction, une valeur de départ (souvent l'identité de la fonction de réduction considérée), et la liste à réduire ; exemples :
 - `foldr (-) 0 [1..5]` vaut 3 (c.-à-d $1 - (2 - (3 - (4 - (5 - 0))))$)
 - `foldl (-) 0 [1..5]` vaut -15 (c.-à-d $(((((0 - 1) - 2) - 3) - 4) - 5)$)
- l'opérateur `(! !)`, permettant d'accéder au $k^{\text{ième}}$ élément ; liste chaînée oblige, cette opération se fait, assez désastreusement, en $O(k)$ – on préférera donc utiliser d'autres types de données que les listes dans les cas où l'on aura un grand nombre d'accès aléatoires à effectuer
- l'opérateur de concaténation `(++)`, vu dans le tout premier exemple de cette annexe
- `take :: Int -> [a] -> [a]`, qui prend les n premiers éléments d'une liste
- `drop :: Int -> [a] -> [a]`, qui ôte les n premiers éléments d'une liste
- `takeWhile :: (a -> Bool) -> [a] -> [a]`, qui prend les n premiers éléments d'une liste, n étant l'indice (en partant de 0) du premier élément de la liste à ne pas satisfaire au prédicat passé en guise de premier argument ; `takeWhile` renvoie toute la liste si tous ses éléments satisfont à ce prédicat
- `dropWhile :: (a -> Bool) -> [a] -> [a]`, qui est à `takeWhile` ce que `drop` est à `take`
- `elem :: Eq a => a -> [a] -> Bool`, renvoyant vrai si et seulement si son premier argument fait partie de la liste passée en guise de deuxième argument ; souvent utilisé avec la syntaxe infixe `x 'elem' xs`, qui a l'avantage de rappeler la notation $x \in xs$ (notons que toute fonction d'arité deux peut utiliser ce genre de syntaxe)

A.5.3 Monades

Le prélude définit le concept de monade par le biais de la classe `Monad`, et d'un certain nombre de fonctions qui aident à la construction d'expressions monadiques. Citons en particulier `sequence_`, qui équivaut à `foldr (>>) (return ())`, où `<<>>` est une variante du `bind`⁶. Passer une liste de monades à `sequence_` est équivalent à les enchaîner les unes après les autres dans un `<<do>>`, en respectant l'ordre dans lequel elles sont rangées. Par exemple, `sequence_ [x, y, z]` équivaut à `do x ; y ; z`.

Il existe également quelques définitions d'instances de `Monad` dans le prélude, comme `[a]` que nous avons déjà vue, et `IO`, la monade d'entrées/sorties. En réalité, le type de la fonction `main` est `IO ()`. La monade d'entrées/sorties permet beaucoup de choses – comme par exemple la lecture des variables d'environnement ou l'exécution de commandes externes –, mais les fonctions du prélude qui la concernent se limitent à la lecture et écriture depuis les entrées/sorties standard et les fichiers. Parmi elles, citons :

- `putStr`, déjà vue, écrivant sur la sortie standard
- `getLine`, lisant une ligne depuis l'entrée standard
- `readFile :: FilePath -> IO String`, lisant le contenu d'un fichier et le plaçant dans la monade (`FilePath` correspond à un nom de fichier, et est en fait un alias pour `String`)
- `writeFile :: FilePath -> String -> IO ()`, écrivant un fichier à partir d'une chaîne de caractères correspondant à la totalité de son contenu

C'est l'évaluation paresseuse qui permet de lire et d'écrire des fichiers d'un bloc, sans forcément avoir à stocker la totalité contenu lu ou écrit dans la mémoire, ou à attendre que tout le fichier soit lu pour commencer les calculs, par exemple.

6. Plus précisément, on a `m >> x = m >>= _ -> x`.

A.5.4 Divers

- Le prélude comporte encore quelques classes et fonctions assez générales mais importantes :
- la classe `Show`, définissant la fonction membre `show`, correspond aux types représentables par une chaîne de caractères
 - les fonctions `fst` et `snd` renvoient respectivement le premier et le deuxième élément d'une paire
 - le type polymorphe `Maybe a`, qui dispose de deux constructeurs : `Just a` et `Nothing`; son utilisation est un peu similaire à celle d'un pointeur vers un objet de type `a` que l'on mettrait à 0 (ici : `Nothing`) pour indiquer que l'objet sur lequel il est censé pointer n'existe pas
 - le type polymorphe `Either a b`, qui dispose également de deux constructeurs : `Left a` et `Right b`; comme son nom peut le laisser supposer, il sert à représenter une expression soit du type `a`, soit du type `b`, en fonction du constructeur qui a été utilisé

Annexe B

Reconstruction tomographique

B.1 Principe de la reconstruction tomographique

La tomographie est un ensemble de techniques numériques ayant pour objectif de reconstruire un volume à partir d'un ensemble de mesures effectuées à l'extérieur de celui-ci [51]. Une variété de phénomènes physiques peuvent être utilisés afin de réaliser ces mesures. En imagerie médicale, il est possible d'utiliser les rayons X (*computed tomography*, ou CT), ou encore les paires de photons générées lors de l'annihilation d'un électron et d'un positron (*positron emission tomography*, ou PET). La tomographie trouve également son utilisation dans d'autres domaines, telles que la géophysique via la tomographie sismique, ou la microscopie (*transmission electron microscopy*, ou TEM).

En règle générale, la reconstruction tomographique est réalisée via un algorithme itératif comportant deux étapes distinctes :

- la projection, consistant à calculer les mesures au niveau des capteurs extérieurs à partir d'un candidat de volume reconstruit (on utilise pour cela un modèle direct, souvent à base de techniques telles que le lancer de rayons)
- la rétroprojection, consistant à calculer un candidat de volume reconstruit à partir des mesures extérieures (on utilise pour cela un modèle inverse, souvent très spécifique à la façon dont les mesures ont été effectuées)

La reconstruction se fait en alternant ces deux étapes pour progressivement faire converger un volume candidat vers le volume sur lequel ont été effectuées les mesures. À chaque itération, l'erreur entre les mesures obtenues via les capteurs et celles calculées par projection sur le volume candidat sert à affiner celui-ci. Au terme d'un certain nombre d'itérations, on considère que le volume candidat est suffisamment proche du volume d'origine.

Dans le reste de cette annexe, tout comme au chapitre 3, nous nous intéressons plus particulièrement à l'étape de rétroprojection en tomographie PET. La section B.2 offre une description concise des équations du modèle inverse en jeu. Ce sont ces équations qui sont utilisées aux sections 3.1 et 3.2, qui présentent deux exemples d'architectures implémentant la rétroprojection PET, ainsi que les modèles de calcul qui ont servi à leur spécification.

B.2 Rétroprojection en tomographie PET

Dans la suite, nous emprunterons les notations utilisées par [25] (comme nous l'avons fait au chapitre 3).

Un scanner PET est constitué d'un ensemble de détecteurs formant un cylindre autour du volume dont on souhaite faire l'acquisition. Le cylindre est formé de plusieurs *anneaux* équidistants, eux mêmes constitués de *détecteurs* équidistants. Lorsqu'une paire électron-positron s'annihile, les deux photons émis partent dans des directions opposées appartenant à une droite appelée ligne de réponse (LOR). Chacun de ces deux photons frappe le cylindre au niveau d'un détecteur en

particulier. Il en résulte que quatre paramètres sont nécessaires pour caractériser une LOR. On les répartit ainsi :

- l'information sur la paire d'anneaux qui ont été touchés par les photons ; on notera v_{\parallel} , la moyenne de la coordonnée axiale des deux anneaux, et Δ , la différence de coordonnée axiale entre les deux anneaux
- l'information sur la position des détecteurs touchés dans leurs anneaux respectifs ; on note ψ l'angle longitudinal de la LOR, et u_{\parallel} sa coordonnée tangentielle

On appelle sinogramme un tableau quadri-dimensionnel indicé par $(\Delta, \psi, u_{\parallel}, v_{\parallel})$ et stockant le nombre de fois qu'une LOR a été enregistrée par les capteurs. C'est sous forme de sinogramme que le scanner fournit ses acquisitions, et donc c'est à partir de lui qu'il va falloir reconstruire le volume d'origine. On notera p_{PET} le sinogramme et f_{PET}^* le volume reconstruit.

Soit $\vec{r} = (x, y, z)$, un voxel¹ du volume reconstruit. Pour tout voxel, il nous faudra sommer la contribution de toutes les LORs qui passent par lui. On dispose de deux degrés de liberté dans le choix d'une LOR passant par un voxel donné : Δ et ψ . Ainsi, on aura :

$$\begin{aligned}
 f_{\text{PET}}^*(\vec{r}) &= \iint p_{\text{PET}}(\Delta, \psi, u_{\parallel}(\psi, \vec{r}), v_{\parallel}(\psi, \Delta, \vec{r})) J_{\Delta} d\psi d\Delta \\
 u_{\parallel}(\psi, \vec{r}) &= x \cos \psi + y \sin \psi + \text{offset} \\
 v_{\parallel}(\psi, \Delta, \vec{r}) &= \frac{\Delta}{2R_a} (x \sin \psi - y \cos \psi) + z + \text{offset}
 \end{aligned} \tag{B.1}$$

J_{Δ} , R_a et les offsets sont des paramètres connus à une itération donnée de l'algorithme de reconstruction, et que nous considérons comme des constantes dans les diverses architectures étudiées, sans rentrer dans le détail de leur calcul.

1. un élément cubique de base, par analogie avec le terme pixel

Annexe C

Code source d'un coprocesseur pour la traversée de grilles hiérarchiques, niveau RTL

C.1 Original (VHDL)

```
1 package th_pkg is
2     _____
3     — Declarations de types —
4     _____
5     constant param_size: integer := 18;
6     constant profondeur_max: integer := 4;
7     constant rayons_max: integer := 256;
8     constant cote: integer := 4;
9
10    subtype profondeur_t is integer range 0 to profondeur_max;
11    subtype rayon_t is integer range 0 to rayons_max - 1;
12    subtype coordonnee_t is integer range 0 to cote ** profondeur_max - 1;
13
14    subtype coordonnee_pixel_t is integer range 0 to 2 ** 12 - 1;
15
16    subtype direction_t is integer range 0 to 6;
17
18    constant DIRECTION_O: direction_t := 0;
19    constant DIRECTION_PX: direction_t := 1;
20    constant DIRECTION_PY: direction_t := 2;
21    constant DIRECTION_PZ: direction_t := 3;
22    constant DIRECTION_NX: direction_t := 4;
23    constant DIRECTION_NY: direction_t := 5;
24    constant DIRECTION_NZ: direction_t := 6;
25
26    subtype st_t is integer range 0 to 1;
27
28    constant S: st_t := 0;
29    constant T: st_t := 1;
30
31    subtype scalaire_t is integer range
32        -(2**(param_size - 1)) to 2**(param_size - 1) - 1;
33    type parametre_t is array (st_t) of scalaire_t;
34
35    constant parametre_zero: parametre_t := (others => 0);
36
37    subtype sens_t is integer range -1 to 1;
38
```

```

39     constant POSITIF: sens_t := 1;
40     constant NEGATIF: sens_t := -1;
41
42     constant sens_zero: sens_t := 0;
43
44     subtype axe_t is integer range 0 to 2;
45     constant X: axe_t := 0;
46     constant Y: axe_t := 1;
47     constant Z: axe_t := 2;
48
49     constant axe_zero: axe_t := X;
50
51     type a_t is array(axe_t) of sens_t;
52     type triplet_parametres_t is array(axe_t) of parametre_t;
53     type position_t is array(axe_t) of coordonnee_t;
54
55     type contexte_rayon_t is record
56         x: coordonnee_pixel_t;
57         y: coordonnee_pixel_t;
58     end record;
59
60     constant contexte_rayon_zero: contexte_rayon_t := (x => 0, y => 0);
61
62     type etat_parametres_t is record
63         nouveau_ne: boolean;
64         position: position_t;
65         u: triplet_parametres_t;
66         u_e: parametre_t;
67         a: a_t;
68         delta_u: triplet_parametres_t;
69         contexte_rayon: contexte_rayon_t;
70
71         profondeur: profondeur_t; -- temp
72     end record;
73
74     constant etat_parametres_zero: etat_parametres_t := (
75         nouveau_ne => true,
76         position => (others => 0),
77         u => (others => parametre_zero),
78         u_e => parametre_zero,
79         a => (others => POSITIF),
80         delta_u => (others => parametre_zero),
81         contexte_rayon => contexte_rayon_zero,
82         profondeur => 0
83     );
84
85     type etat_qualifie_t is record
86         rayon: rayon_t;
87         profondeur: profondeur_t;
88         voxel_subdivise: boolean;
89         parametres: etat_parametres_t;
90     end record;
91
92     constant etat_qualifie_zero: etat_qualifie_t := (
93         rayon => 0,
94         profondeur => 0,
95         voxel_subdivise => false,
96         parametres => etat_parametres_zero
97     );
98
99     type fragment_t is record
100         valeur: integer range 0 to 2**15 - 1;
101         u_e: parametre_t;
102         u_s: parametre_t;
103         rayon: rayon_t;
104         contexte: contexte_rayon_t;
105         premier_fragment: boolean;

```

```

106         dernier_fragment: boolean;
107     end record;
108
109     constant fragment_zero: fragment_t := (
110         valeur => 0,
111         u_e => parametre_zero,
112         u_s => parametre_zero,
113         rayon => 0,
114         contexte => contexte_rayon_zero,
115         premier_fragment => false,
116         dernier_fragment => false
117     );
118
119     type acces_cache_t is record
120         premier_acces_pour_ce_rayon: boolean;
121         position: position_t;
122         profondeur: profondeur_t;
123     end record;
124
125     type reponse_cache_t is record
126         voxel_subdivise: boolean;
127         valeur: integer range 0 to 2**15 - 1;
128     end record;
129
130     function ceillog2(a: integer) return integer;
131     procedure set_bit_to(
132         a: inout integer;
133         bit_pos: in integer;
134         value: in boolean;
135         high: in integer
136     );
137     function get_bit(a: in integer; bit_pos: in integer) return boolean;
138 end th_pkg;
139
140 package body th_pkg is
141     function ceillog2(a: integer) return integer is
142     begin
143         assert a > 0;
144         if a = 1 then
145             return 0;
146         else
147             if (a mod 2) > 0 then
148                 return 1 + ceillog2((a/2) + 1);
149             else
150                 return 1 + ceillog2(a / 2);
151             end if;
152         end if;
153     end;
154
155     procedure set_bit_to(
156         a: inout integer;
157         bit_pos: integer;
158         value: boolean;
159         high: in integer
160     ) is
161         variable u: unsigned(ceillog2(high + 1) - 1 downto 0);
162     begin
163         u := to_unsigned(a, ceillog2(high + 1));
164         if value then
165             u(bit_pos) := '1';
166         else
167             u(bit_pos) := '0';
168         end if;
169         a := to_integer(u);
170     end;
171
172     function get_bit(a: in integer; bit_pos: in integer) return boolean is

```

```

173     begin
174         if ((a / (2 ** bit_pos)) mod 2) = 1 then
175             return true;
176         else
177             return false;
178         end if;
179     end;
180 end;

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 library libth;
6 use libth.th_pkg.all;
7
8 entity trieur_plongeur is
9     port(
10         clk: in std_ulogic;
11         reset: in std_ulogic;
12
13         eqi_valeur: in integer range 0 to 2**15 - 1;
14         eqi_etat: in etat_qualifie_t;
15         eqi_valide: in boolean;
16         eqi_ack: buffer boolean;
17
18         fragment: out fragment_t;
19         fragment_valide: buffer boolean;
20         fragment_ack: in boolean;
21
22         etat_qualifie: out etat_qualifie_t;
23         etat_qualifie_reinsérer: out boolean;
24         etat_qualifie_valide: buffer boolean;
25         etat_qualifie_ack: in boolean
26     );
27 end entity trieur_plongeur;
28
29 architecture rtl of trieur_plongeur is
30     type mode_t is (PIPELINE_TRI, ITERATEUR_PLONGEON);
31     signal mode_s0, mode_s1: mode_t;
32
33     type compensation_t is array (axe_t) of scalaire_t;
34     type tableau_operandes_t is array (2 downto 0) of parametre_t;
35     type tableau_resultats_t is array (2 downto 0) of boolean;
36
37     signal compareur_operande_a_s0: tableau_operandes_t;
38     signal compareur_operande_b_s0: tableau_operandes_t;
39     signal compareur_resultat_s0: tableau_resultats_t;
40
41     type et1_t is record
42         etat: etat_qualifie_t;
43         axe_si_vrai: axe_t;
44         ull: parametre_t;
45         valeur: integer range 0 to 2**15 - 1;
46     end record;
47
48     -- signal eqi_ack_s0: boolean; (on utilise directement la sortie)
49
50     signal et1_s0, et1_s1: et1_t;
51     signal et1_valide_s0, et1_valide_s1: boolean;
52     signal et1_ack_s0: boolean;
53
54     type et2_t is record
55         etat: etat_qualifie_t;
56         axe: axe_t;
57         valeur: integer range 0 to 2**15 - 1;
58     end record;

```

```

59
60     signal et2_s0 , et2_s1: et2_t;
61     signal et2_valide_s0 , et2_valide_s1: boolean;
62     signal et2_ack_s0: boolean;
63
64     signal iteration_s0 , iteration_s1: integer range 0 to 15;
65     signal etat_s0 , etat_s1: etat_qualifie_t;
66
67     signal axe_s0: axe_t;
68     signal sens_s0: sens_t;
69     signal hauteur_s0: integer; — FIXME: mettre des bornes?
70     signal increment_s0: integer;
71
72     signal fragment_s0 , fragment_s1: fragment_t;
73     signal fragment_valide_s0 , fragment_valide_s1: boolean;
74     signal etat_qualifie_s0 , etat_qualifie_s1: etat_qualifie_t;
75     signal etat_qualifie_reinserer_s0 , etat_qualifie_reinserer_s1: boolean;
76     signal etat_qualifie_valide_s0 , etat_qualifie_valide_s1: boolean;
77 begin
78     fragment <= fragment_s0;
79     fragment_valide <= fragment_valide_s0;
80     etat_qualifie <= etat_qualifie_s0;
81     etat_qualifie_reinserer <= etat_qualifie_reinserer_s0;
82     etat_qualifie_valide <= etat_qualifie_valide_s0;
83
84     synchrone: process(clk)
85     begin
86         if rising_edge(clk) then
87             if reset = '1' then
88                 mode_s0 <= PIPELINE_TRI;
89                 et1_s0 <= (
90                     etat => etat_qualifie_zero ,
91                     axe_si_vrai => axe_zero ,
92                     u1_l => parametre_zero ,
93                     valeur => 0
94                 );
95                 et1_valide_s0 <= false;
96                 et2_s0 <= (
97                     etat => etat_qualifie_zero ,
98                     axe => axe_zero ,
99                     valeur => 0
100                );
101                 et2_valide_s0 <= false;
102                 iteration_s0 <= 0;
103                 etat_s0 <= etat_qualifie_zero;
104                 fragment_s0 <= fragment_zero;
105                 fragment_valide_s0 <= false;
106                 etat_qualifie_s0 <= etat_qualifie_zero;
107                 etat_qualifie_valide_s0 <= false;
108             else
109                 mode_s0 <= mode_s1;
110                 et1_s0 <= et1_s1;
111                 et1_valide_s0 <= et1_valide_s1;
112                 et2_s0 <= et2_s1;
113                 et2_valide_s0 <= et2_valide_s1;
114                 iteration_s0 <= iteration_s1;
115                 etat_s0 <= etat_s1;
116                 fragment_s0 <= fragment_s1;
117                 fragment_valide_s0 <= fragment_valide_s1;
118                 etat_qualifie_s0 <= etat_qualifie_s1;
119                 etat_qualifie_reinserer_s0
120                     <= etat_qualifie_reinserer_s1;
121                 etat_qualifie_valide_s0
122                     <= etat_qualifie_valide_s1;
123             end if;
124         end if;
125     end process synchrone;

```

```

126
127     compareurs: process (
128         compareur_operande_a_s0 ,
129         compareur_operande_b_s0
130     )
131         variable a, b: integer;
132         variable signe_t_t: boolean;
133     begin
134         for i in 0 to 2 loop
135             a := compareur_operande_a_s0(i)(S);
136             a := a * compareur_operande_b_s0(i)(T);
137             b := compareur_operande_b_s0(i)(S);
138             b := b * compareur_operande_a_s0(i)(T);
139             signe_t_t :=
140                 (compareur_operande_a_s0(i)(T) < 0)
141                 xor (compareur_operande_b_s0(i)(T) < 0);
142             compareur_resultat_s0(i) <=
143                 (a <= b) xor signe_t_t;
144         end loop;
145     end process;
146
147     combinatoire: process(
148         eqi_valeur ,
149         eqi_etat ,
150         eqi_valide ,
151         fragment_ack ,
152         etat_qualifie_ack ,
153         compareur_resultat_s0 ,
154         mode_s0 ,
155         et1_s0 ,
156         et1_valide_s0 ,
157         et1_ack_s0 ,
158         et2_s0 ,
159         et2_valide_s0 ,
160         et2_ack_s0 ,
161         iteration_s0 ,
162         etat_s0 ,
163         fragment_s0 ,
164         fragment_valide_s0 ,
165         etat_qualifie_s0 ,
166         etat_qualifie_reinsérer_s0 ,
167         etat_qualifie_valide_s0
168     )
169         variable stall: boolean;
170         variable read_stall: boolean;
171         variable write_stall: boolean;
172
173         variable eqi_read_v: boolean;
174         variable et1_write_v: boolean;
175         variable et1_read_v: boolean;
176         variable et2_write_v: boolean;
177         variable et2_read_v: boolean;
178         variable etat_qualifie_write_v: boolean;
179         variable fragment_write_v: boolean;
180
181         variable eqi_valeur_v: integer range 0 to 2**15 - 1;
182         variable eqi_etat_v: etat_qualifie_t;
183         variable eqi_valide_v: boolean;
184         variable eqi_ack_v: boolean;
185         variable mode_v, mode_v2: mode_t;
186         variable compareur_operande_a_v: tableau_operandes_t;
187         variable compareur_operande_b_v: tableau_operandes_t;
188         variable compareur_resultat_v: tableau_resultats_t;
189         variable iteration_v: integer range 0 to 15;
190         variable etat_v: etat_qualifie_t;
191         variable et1_v: et1_t;
192         variable et1_valide_v: boolean;

```

```

193     variable et1_ack_v: boolean;
194     variable et2_v: et2_t;
195     variable et2_valide_v: boolean;
196     variable et2_ack_v: boolean;
197     variable axe_v: axe_t;
198     variable sens_v: sens_t;
199     variable hauteur_v: integer;
200     variable increment_v: integer;
201     variable coordonnee_incrementee_v: coordonnee_t;
202     variable compensation_v: compensation_t;
203     variable fragment_v: fragment_t;
204     variable fragment_valide_v: boolean;
205     variable fragment_ack_v: boolean;
206     variable etat_qualifie_v: etat_qualifie_t;
207     variable etat_qualifie_reinsérer_v: boolean;
208     variable etat_qualifie_valide_v: boolean;
209     variable etat_qualifie_ack_v: boolean;
210     variable u_demi_v: parametre_t;
211     variable shift_v: integer;
212
213     — Temporaire
214     type termes_t is array (natural range 0 to 7) of parametre_t;
215     type ttermes_t is array (axe_t) of termes_t;
216     variable ttermes: ttermes_t;
217
218     — Combinatoire (datapath)
219     procedure calculer_compensation(
220         i_v: integer;
221         profondeur_determinee_v: boolean
222     ) is
223         variable value_v: boolean;
224         variable p_v: integer;
225     begin
226         for l in axe_t'low to axe_t'high loop
227             value_v := get_bit(
228                 et2_s0.etat.parametres.position(1),
229                 i_v
230             ) xor (et2_s0.etat.parametres.a(1) = POSITIF);
231             if
232                 not profondeur_determinee_v
233                 and i_v <= profondeur_max * ceillog2(cote) - 2
234             then
235                 p_v := ceillog2(cote) * profondeur_max;
236                 p_v := p_v - i_v;
237                 if (value_v) then
238                     ttermes(1)(i_v)(S) :=
239                         etat_qualifie_v.parametres
240                             .delta_u(1)(S) / (2 ** p_v);
241                     ttermes(1)(i_v)(T) :=
242                         etat_qualifie_v.parametres
243                             .delta_u(1)(T) / (2 ** p_v);
244                 end if;
245                 set_bit_to(
246                     etat_qualifie_v.parametres.position(1),
247                     i_v, false, coordonnee_t'high
248                 );
249             end if;
250         end loop;
251     end;
252
253     procedure tri_etage1 is
254     begin
255         compareur_operande_a_v(0) := eqi_etat.parametres.u(X);
256         compareur_operande_b_v(0) := eqi_etat.parametres.u(Y);
257         et1_write_v := true;
258         et1_v.etat := eqi_etat;
259         et1_v.valeur := eqi_valeur;

```

```

260         if (comparateur_resultat_s0(0)) then
261             et1_v.u1_l := eqi_etat.parametres.u(X);
262             et1_v.axe_si_vrai := X;
263         else
264             et1_v.u1_l := eqi_etat.parametres.u(Y);
265             et1_v.axe_si_vrai := Y;
266         end if;
267     end;
268
269     procedure tri_etage2 is
270     begin
271         compareur_operande_a_v(1) := et1_s0.u1_l;
272         compareur_operande_b_v(1) := et1_s0.etat.parametres.u(Z);
273         et1_read_v := true;
274         et2_write_v := true;
275         et2_v.etat := et1_s0.etat;
276         et2_v.valeur := et1_s0.valeur;
277         if (comparateur_resultat_s0(1)) then
278             et2_v.axe := et1_s0.axe_si_vrai;
279         else
280             et2_v.axe := Z;
281         end if;
282     end;
283
284     procedure tri_etage3 is
285     variable profondeur_v: profondeur_t;
286     variable profondeur_determinee_v: boolean;
287     variable dernier_fragment_v: boolean;
288     variable continuer_a_remonter_v: boolean;
289     variable tmp_v: unsigned(
290         ceillog2(cote) * profondeur_max - 1 downto 0
291     );
292     variable i_v: integer;
293     variable l_v: axe_t;
294
295     begin
296         profondeur_v := et2_s0.etat.profondeur;
297         profondeur_determinee_v := false;
298         et2_read_v := true;
299         sens_v := et2_s0.etat.parametres.a(axe_v);
300         coordonnee_incrementee_v
301             := et2_s0.etat.parametres.position(axe_v);
302
303         tmp_v := to_unsigned(
304             coordonnee_incrementee_v,
305             ceillog2(cote) * profondeur_max
306         );
307
308         etat_qualifie_v := et2_s0.etat;
309
310         for p in profondeur_max downto 1 loop
311             continuer_a_remonter_v := true;
312             for c in 0 to ceillog2(cote) - 1 loop
313                 i_v := profondeur_max - p;
314                 i_v := i_v * ceillog2(cote) + c;
315                 continuer_a_remonter_v :=
316                     continuer_a_remonter_v and (
317                         tmp_v(i_v) = '1'
318                         xor sens_v = NEGATIF
319                     );
320                 ttermes(X)(i_v) := parametre_zero;
321                 ttermes(Y)(i_v) := parametre_zero;
322                 ttermes(Z)(i_v) := parametre_zero;
323             end loop;
324             if
325                 (not profondeur_determinee_v)
326                 and p <= et2_s0.etat.profondeur

```

```

327         if not continuer_a_remonter_v then
328             profondeur_determinee_v := true;
329             profondeur_v := p;
330         end if;
331         for c in 0 to ceillog2(cote) - 1 loop
332             i_v := profondeur_max - p;
333             i_v := i_v * ceillog2(cote) + c;
334             calculer_compensation(
335                 i_v,
336                 profondeur_determinee_v
337             );
338         end loop;
339     end if;
340 end loop;
341 for l in axe_t'low to axe_t'high loop
342     etat_qualifie_v.parametres.u(1) :=
343         etat_qualifie_v.parametres.u(1)
344         + (((
345             ttermes(1)(0)
346             + ttermes(1)(1)
347         ) + (
348             ttermes(1)(2)
349             + ttermes(1)(3)
350         )) + ((
351             ttermes(1)(4)
352             + ttermes(1)(5)
353         ) + (
354             ttermes(1)(6)
355             + ttermes(1)(7)
356         )));
357 end loop;
358
359 coordonnee_incrementee_v
360     := etat_qualifie_v.parametres.position(axe_v);
361
362 if not profondeur_determinee_v then
363     dernier_fragment_v := true;
364 else
365     dernier_fragment_v := false;
366 end if;
367
368 fragment_write_v := true;
369 fragment_v.valeur := et2_s0.valeur;
370 fragment_v.u_e := et2_s0.etat.parametres.u_e;
371 fragment_v.u_s := et2_s0.etat.parametres.u(axe_v);
372 fragment_v.rayon := et2_s0.etat.rayon;
373 fragment_v.contexte
374     := et2_s0.etat.parametres.contexte_rayon;
375 fragment_v.premier_fragment
376     := et2_s0.etat.parametres.nouveau_ne;
377 fragment_v.dernier_fragment := dernier_fragment_v;
378
379 etat_qualifie_write_v := true;
380 etat_qualifie_reinsérer_v := not dernier_fragment_v;
381 etat_qualifie_v.profondeur := profondeur_v;
382 etat_qualifie_v.parametres.profondeur := profondeur_v;
383 etat_qualifie_v.parametres.nouveau_ne := false;
384 etat_qualifie_v.parametres.u_e
385     := etat_qualifie_v.parametres.u(axe_v);
386
387 hauteur_v := (profondeur_max - profondeur_v);
388 hauteur_v := hauteur_v * ceillog2(cote);
389 if (sens_v = POSITIF) then
390     increment_v := 2**hauteur_v;
391 else
392     increment_v := -1 * (2 ** hauteur_v);
393 end if;

```

```

394
395     etat_qualifie_v.parametres.position(axe_v)
396         := (coordonnee_incrementee_v + increment_v)
397         mod (cote ** profondeur_max);
398     etat_qualifie_v.parametres.u(axe_v)
399         := etat_qualifie_v.parametres.u(axe_v)
400         + shift_right(
401             etat_qualifie_v.parametres
402                 .delta_u(axe_v),
403             ceillog2(cote) * profondeur_v
404         );
405 end;
406
407 procedure plongeon is
408     type tableau_demis_aux_t is
409         array (natural range 0 to 10) of parametre_t;
410     type tableau_demis_t is
411         array (axe_t) of tableau_demis_aux_t;
412     variable tableau_demis: tableau_demis_t;
413 begin
414     hauteur_v := profondeur_max - etat_v.profondeur;
415     hauteur_v := hauteur_v * ceillog2(cote);
416     hauteur_v := hauteur_v - (iteration_v + 1);
417     for i in 0 to 10 loop
418         for l in axe_t'low to axe_t'high loop
419             tableau_demis(l)(i) :=
420                 etat_v.parametres.u(l)
421                 - shift_right(
422                     etat_v.parametres
423                         .delta_u(l), i
424                 );
425         end loop;
426     end loop;
427     for l in axe_t'low to axe_t'high loop
428         shift_v := etat_v.profondeur * ceillog2(cote);
429         shift_v := shift_v + iteration_v + 1;
430         u_demi_v := tableau_demis(l)(shift_v);
431         compareur_operande_a_v(l)
432             := etat_v.parametres.u_e;
433         compareur_operande_b_v(l)
434             := u_demi_v;
435         if (
436             compareur_resultat_v(l)
437             = (etat_v.parametres.a(l) = NEGATIF)
438         ) then
439             if (etat_v.parametres.a(l)=NEGATIF) then
440                 etat_v.parametres.u(l)
441                     := u_demi_v;
442             end if;
443             set_bit_to(
444                 etat_v.parametres.position(l),
445                 hauteur_v, true, coordonnee_t'high
446             );
447         else
448             if (etat_v.parametres.a(l)=POSITIF) then
449                 etat_v.parametres.u(l)
450                     := u_demi_v;
451             end if;
452             set_bit_to(
453                 etat_v.parametres.position(l),
454                 hauteur_v, false, coordonnee_t'high
455             );
456         end if;
457     end loop;
458     etat_v.parametres.nouveau_ne := false;
459 end;
460 begin

```

```

461     stall := false;
462     read_stall := false;
463     write_stall := false;
464
465     — Valeurs par default des locks
466     eqi_read_v := false;
467     et1_write_v := false;
468     et1_read_v := false;
469     et2_write_v := false;
470     et2_read_v := false;
471     etat_qualifie_write_v := false;
472     fragment_write_v := false;
473
474     — Valeurs par default des variables
475     eqi_valeur_v := eqi_valeur;
476     eqi_etat_v := eqi_etat;
477     eqi_valide_v := eqi_valide;
478     eqi_ack_v := false;
479     mode_v := mode_s0;
480     mode_v2 := mode_s0;
481     compareur_operande_a_v := (others => parametre_zero);
482     compareur_operande_b_v := (others => parametre_zero);
483     compareur_resultat_v := compareur_resultat_s0;
484     iteration_v := iteration_s0;
485     etat_v := etat_s0;
486     et1_v := et1_s0;
487     et1_valide_v := et1_valide_s0;
488     et1_ack_v := false;
489     et2_v := et2_s0;
490     et2_valide_v := et2_valide_s0;
491     et2_ack_v := false;
492     axe_v := et2_s0.axe;
493     sens_v := sens_zero;
494     hauteur_v := 0;
495     increment_v := 0;
496     compensation_v := (others => 0);
497     coordonnee_incrementee_v := 0;
498     fragment_v := fragment_zero;
499     fragment_valide_v := fragment_valide_s0;
500     fragment_ack_v := fragment_ack;
501     etat_qualifie_v := etat_qualifie_s0;
502     etat_qualifie_reinsérer_v := true;
503     etat_qualifie_valide_v := etat_qualifie_valide_s0;
504     etat_qualifie_ack_v := etat_qualifie_ack;
505     u_demi_v := parametre_zero;
506     shift_v := 0;
507
508     — Combinatoire (FSM)
509     if (mode_v = PIPELINE_TRI) and (iteration_v = 0) then
510         eqi_read_v := true;
511         if (eqi_etat_v.voxel_subdivise) then
512             mode_v := ITERATEUR_PLONGEON;
513             etat_v := eqi_etat_v;
514             iteration_v := 0;
515         else
516             mode_v := PIPELINE_TRI;
517         end if;
518     end if;
519
520     if (mode_v = PIPELINE_TRI) then
521         tri_etage1;
522         tri_etage2;
523         tri_etage3;
524         mode_v2 := mode_v;
525     else — elsif (mode_v = ITERATEUR_PLONGEON) then
526         plongeon;
527         mode_v2 := mode_v;

```

```

528
529
530         if (iteration_v = ceillog2(cote) - 1) then
531             etat_qualifie_write_v := true;
532             etat_v.profondeur := etat_v.profondeur + 1;
533             etat_v.parametres.profondeur
534                 := etat_v.profondeur;
535             etat_qualifie_v := etat_v;
536             iteration_v := 0;
537             mode_v2 := PIPELINE_TRI;
538
539         else
540             iteration_v := iteration_v + 1;
541         end if;
542
543     end if;
544
545     — Abaissement des signaux de validite si necessaire
546     et1_valide_v := et1_valide_v and not et1_ack_s0;
547     et2_valide_v := et2_valide_v and not et2_ack_s0;
548     fragment_valide_v := fragment_valide_v and not fragment_ack;
549     etat_qualifie_valide_v := etat_qualifie_valide_v
550         and not etat_qualifie_ack;
551
552     — Assignation des signaux (s0)
553     compareur_operande_a_s0 <= compareur_operande_a_v;
554     compareur_operande_b_s0 <= compareur_operande_b_v;
555     axe_s0 <= axe_v;
556     sens_s0 <= sens_v;
557     hauteur_s0 <= hauteur_v;
558     increment_s0 <= increment_v;
559
560     — Assignation des signaux (s1)
561     mode_s1 <= mode_s0;
562     iteration_s1 <= iteration_s0;
563     fragment_s1 <= fragment_s0;
564     etat_qualifie_s1 <= etat_qualifie_s0;
565     etat_qualifie_reinsérer_s1 <= etat_qualifie_reinsérer_s0;
566     etat_s1 <= etat_s0;
567     et1_s1 <= et1_s0;
568     et2_s1 <= et2_s0;
569
570     — Combinatoire (controle)
571     if mode_v = PIPELINE_TRI then
572         read_stall := eqi_read_v and not eqi_valide;
573         write_stall := et1_write_v and et1_valide_v;
574         stall := read_stall or write_stall;
575
576         if not stall then
577             mode_s1 <= mode_v2;
578             iteration_s1 <= iteration_v;
579             etat_s1 <= etat_v;
580             et1_s1 <= et1_v;
581             et1_valide_v := et1_valide_v or et1_write_v;
582             eqi_ack_v := eqi_read_v;
583         end if;
584
585         read_stall := et1_read_v and not et1_valide_s0;
586         write_stall := et2_write_v and et2_valide_v;
587         stall := read_stall or write_stall;
588
589         if not stall then
590             et2_s1 <= et2_v;
591             et2_valide_v := et2_valide_v or et2_write_v;
592             et2_ack_v := et2_read_v;
593         end if;
594
595         read_stall := false;
596         read_stall := read_stall or (
597             et2_read_v and not et2_valide_s0

```

```

595         );
596         write_stall := false;
597         write_stall := write_stall or (
598             etat_qualifie_write_v
599             and etat_qualifie_valide_v
600         );
601         write_stall := write_stall or (
602             fragment_write_v
603             and fragment_valide_v
604         );
605         stall := read_stall or write_stall;
606
607         if not stall then
608             fragment_s1 <= fragment_v;
609             fragment_valide_v :=
610                 fragment_valide_v
611                 or fragment_write_v;
612             etat_qualifie_s1 <= etat_qualifie_v;
613             etat_qualifie_reinsérer_s1
614                 <= etat_qualifie_reinsérer_v;
615             etat_qualifie_valide_v :=
616                 etat_qualifie_valide_v
617                 or etat_qualifie_write_v;
618             et1_ack_v := et2_read_v;
619         end if;
620     else — elsif mode_v = ITERATEUR_PLONGEON then
621         read_stall := eqi_read_v and not eqi_valide;
622         write_stall := etat_qualifie_write_v and etat_qualifie_valide_v
623         stall := read_stall or write_stall;
624
625         if not stall then
626             mode_s1 <= mode_v2;
627             iteration_s1 <= iteration_v;
628             fragment_s1 <= fragment_v;
629             fragment_valide_s1 <= fragment_valide_v
630                 or fragment_write_v;
631             etat_qualifie_s1 <= etat_qualifie_v;
632             etat_qualifie_reinsérer_s1
633                 <= etat_qualifie_reinsérer_v;
634             etat_qualifie_valide_v :=
635                 etat_qualifie_valide_v
636                 or etat_qualifie_write_v;
637             etat_s1 <= etat_v;
638             eqi_ack_v := eqi_read_v;
639             — et1_s1 <= et1_v;
640             — et1_valide_v := et1_valide_v or et1_write_v;
641             — et2_s1 <= et2_v;
642             — et2_valide_v := et2_valide_v or et2_write_v;
643         end if;
644     end if;
645
646     — Affectation des signaux de validite
647     eqi_ack <= eqi_ack_v;
648     et1_ack_s0 <= et1_ack_v;
649     et1_valide_s1 <= et1_valide_v;
650     et2_ack_s0 <= et2_ack_v; — XXX: corrigé (était et1_ack_v)
651     et2_valide_s1 <= et2_valide_v; — XXX: corrigé (pareil)
652     fragment_valide_s1 <= fragment_valide_v;
653     etat_qualifie_valide_s1 <= etat_qualifie_valide_v;
654 end process combinatoire;
655 end; — architecture rtl of trieur_plongeur

```

C.2 Formulation dans le MoC RTL calquée sur la version VHDL

```

1  import Memnoc.Base
2  import Memnoc.Exploration
3  import Memnoc.Transfer
4  import Memnoc.Engines.StdRTL
5  import Memnoc.Output.Chronogram
6
7  _____
8  -- Declaration de types et constantes --
9  _____
10
11 -- Les types polymorphes doivent toujours être explicites avec Hindley-Milner:
12 profondeurMax :: Num a => a
13
14 paramSize = 18
15 profondeurMax = 4
16 rayonsMax = 256
17 cote = 4
18
19 profondeurT = uint (intlog (profondeurMax+1))
20 rayonT = uint (intlog rayonsMax)
21 coordonneeT = uint (intlog (cote ^ profondeurMax))
22
23 coordonneePixelT = uint 12
24
25 directionT = uint (intlog (6 + 1))
26 directionO = 0
27 directionPx = 1
28 directionPy = 2
29 directionPz = 3
30 directionNx = 4
31 directionNy = 5
32 directionNz = 6
33
34 stT = uint 1
35 stVs = 0
36 stVt = 1
37
38 scalaireT = sint paramSize
39 parametreT = array scalaireT 2
40
41 sensT = sint 2
42 positif = 1
43 negatif = -1
44
45 axeT = uint (intlog 3)
46 axeX :: Num a => a
47 axeY :: Num a => a
48 axeZ :: Num a => a
49 axeX = 0
50 axeY = 1
51 axeZ = 2
52
53 aT = array sensT 3
54 tripletParametresT = array parametreT 3
55 positionT = array coordonneeT 3
56
57 contexteRayonT = struct
58     [ "x" :-: coordonneePixelT
59       , "y" :-: coordonneePixelT
60     ]
61
62 etatParametresT = struct
63     [ "nouveau_ne" :-: bool
64       , "position" :-: positionT
65       , "u" :-: tripletParametresT
66       , "u_e" :-: parametreT
67       , "a" :-: aT

```

ANNEXE C. COPROCESSEUR POUR LA TRAVERSÉE DE GRILLES HIÉRARCHIQUES

```

68     , "delta_u"  :- tripletParametresT
69     , "contexte_rayon"  :- contexteRayonT
70     , "profondeur"  :- profondeurT
71   ]
72
73   etatQualifieT = struct
74     [ "rayon"  :- rayonT
75     , "profondeur"  :- profondeurT
76     , "voxel_subdivise"  :- bool
77     , "parametres"  :- etatParametresT
78     ]
79
80   fragmentT = struct
81     [ "valeur"  :- uint 15
82     , "u_e"    :- parametreT
83     , "u_s"    :- parametreT
84     , "rayon"  :- rayonT
85     , "contexte"  :- contexteRayonT
86     , "premierFragment"  :- bool
87     , "dernierFragment"  :- bool
88     ]
89
90   accesCacheT = struct
91     [ "premier_acces_pour_ce_rayon"  :- bool
92     , "position"  :- positionT
93     , "profondeur"  :- profondeurT
94     ]
95
96   reponseCacheT = struct
97     [ "voxelSubdivise"  :- bool
98     , "valeur"  :- uint 15
99     ]
100
101   -----
102   -- Procedures et fonctions --
103   -----
104
105   -- Non synthetisable tel quel:
106   ceillog2 :: Transfer Type -> Transfer Type
107   ceillog2 n = (n /<= 1) /?/ (0, 1 /+ / ceillog2(n />> 1))
108
109   -- Pour info, version synthétisable naïve:
110   ceillog2' :: Transfer Type -> Transfer Type
111   ceillog2' n =
112     let
113       aux :: Transfer Type -> Integer -> Transfer Type
114       aux n d | d == 0 = 0
115       aux n d = (n /<= 1) /?/ (0, 1 /+ / aux (n />> 1) (d - 1))
116     in do
117       s <- getSize n
118       aux n s
119
120
121   -----
122   -- Trieur plongeur --
123   -----
124
125   trieurPlongeur :: RTL em
126   => Transfer Type -> Transfer Type -> Transfer Type -> Transfer Type
127   -> Transfer Type -> Transfer Type -> Transfer Type
128   -> Transfer Type -> Transfer Type
129   -> Transfer Type -> Transfer Type
130   -> em ()
131
132
133   trieurPlongeur
134     eqiValeurIn eqiEtatIn eqiValide eqiAck

```

```

135     fragmentOut fragmentValideOut fragmentAck
136     etatQualifieOut etatQualifieReinsérerOut
137     etatQualifieValideOut etatQualifieAck
138   = do
139     let modeT = uint 1
140         let pipelineTri = 0
141         let iterateurPlongeon = 1
142
143     mode <- reg modeT
144
145     let compensationT = array scalaireT 3
146         let tableauOperandesT = array parametreT 3
147         let tableauResultatsT = array bool 3
148
149     compareurOperandeA <- node tableauOperandesT
150     compareurOperandeB <- node tableauOperandesT
151     compareurResultats <- node tableauResultatsT
152
153     let et1T = struct
154         [ "etat" :-: etatQualifieT
155           , "axe_si_vrai" :-: axeT
156           , "u1_1" :-: parametreT
157           , "valeur" :-: uint 15
158         ]
159
160     et1 <- reg et1T
161     et1Valide <- reg bool
162     et1Ack <- node bool
163
164     let et2T = struct
165         [ "etat" :-: etatQualifieT
166           , "axe" :-: axeT
167           , "valeur" :-: uint 15
168         ]
169
170     et2 <- reg et2T
171     et2Valide <- reg bool
172     et2Ack <- node bool
173
174     iteration <- reg (uint 4)
175     etat <- reg etatQualifieT
176
177     axe <- node axeT
178     sens <- node sensT
179     hauteur <- node (uint 4)
180     increment <- node (uint 24)
181
182     fragment <- reg fragmentT
183     fragmentValide <- reg bool
184     etatQualifie <- reg etatQualifieT
185     etatQualifieReinsérer <- reg bool
186     etatQualifieValide <- reg bool
187
188   -- begin --
189
190     fragmentOut #= fragment
191     fragmentValideOut #= fragmentValide
192     etatQualifieOut #= etatQualifie
193     etatQualifieReinsérerOut #= etatQualifieReinsérer
194     etatQualifieValideOut #= etatQualifieValide
195
196   0 'process' do
197     a <- var (sint 32)
198     b <- var (sint 32)
199     signe <- var bool
200
201     [0..2] 'for' \i -> do

```

ANNEXE C. COPROCESSEUR POUR LA TRAVERSÉE DE GRILLES HIÉRARCHIQUES

```

202         a #= (comparateurOperandeA |> i |> stVs)
203         a #= a ./ (comparateurOperandeB |> i |> stVt)
204         b #= (comparateurOperandeB |> i |> stVs)
205         b #= b ./ (comparateurOperandeA |> i |> stVt)
206         signe #=
207             ((comparateurOperandeA |> i |> stVt) /</ 0)
208             /> ((comparateurOperandeB |> i |> stVt) /</ 0)
209         (comparateurResultats |> i) #= (a /<=/ b) /> signe
210
211     0 'process' do
212         stall <- var bool
213         readStall <- var bool
214         writeStall <- var bool
215
216         eqiReadV <- var bool
217         et1WriteV <- var bool
218         et1ReadV <- var bool
219         et2WriteV <- var bool
220         et2ReadV <- var bool
221         etatQualifieWriteV <- var bool
222         fragmentWriteV <- var bool
223
224         eqiValeurV <- var (uint 15)
225         eqiEtatV <- var etatQualifieT
226         eqiValideV <- var bool
227         eqiAckV <- var bool
228         modeV <- var modeT
229         modeV2 <- var modeT
230         comparateurOperandeAV <- var tableauOperandesT
231         comparateurOperandeBV <- var tableauOperandesT
232         comparateurResultatV <- var tableauResultatsT
233         iterationV <- var (uint 4)
234         etatV <- var etatQualifieT
235         et1V <- var et1T
236         et1ValideV <- var bool
237         et1AckV <- var bool
238         et2V <- var et2T
239         et2ValideV <- var bool
240         et2AckV <- var bool
241         axeV <- var axeT
242         sensV <- var sensT
243         hauteurV <- var (uint 4)
244         incrementV <- var (uint 24)
245         coordonneeIncrementeeV <- var coordonneeT
246         compensationV <- var compensationT
247         fragmentV <- var fragmentT
248         fragmentValideV <- var bool
249         fragmentAckV <- var bool
250         etatQualifieV <- var etatQualifieT
251         etatQualifieReinsérerV <- var bool
252         etatQualifieValideV <- var bool
253         etatQualifieAckV <- var bool
254         uDemiV <- var parametreT
255         shiftV <- var (uint 8)
256
257         let termesT = array parametreT 8
258         let tTermesT = array termesT 3
259         tTermes <- var tTermesT
260
261         let calculerCompensation iV profondeurDetermineeV = do
262             valueV <- var bool
263             pV <- var (uint 32)
264             [0..2] 'for' \l -> do
265                 c1 <- node bool
266                 c2 <- node bool
267                 cond <- var bool
268                 c1 #= et2 |."etat.parametres.position" |> 1 |> iV

```

```

269         c2 #= et2 |."etat.parameters.a">1 /==/ positif
270         valueV #= c1 /^/ c2;
271         cond #= iV /<=/ (profondeurMax*intlog(cote)-2)
272         cond #= cond /&/ bwn profondeurDetermineeV
273         cond 'iff ' do
274             pV #= intlog(cote) * profondeurMax
275             pV #= pV - iV
276             valueV 'iff ' do
277                 let dul = etatQualifieV
278                     |."parameters.delta_u"
279                 tTermes|>1|>iV|>stVs #=
280                     (dul|>stVs)//(1/<</pV)
281                 tTermes|>1|>iV|>stVs #=
282                     (dul|>stVt)//(1/<</pV)
283                 etatQualifieV |."parameters.position"
284                     |>1|>iV #= 0
285
286     let triEtagel = do
287         compareurOperandeAV|>0 #= eqiEtatIn |."parameters.u">axeX
288         compareurOperandeBV|>0 #= eqiEtatIn |."parameters.u">axeY
289         et1WriteV #= 1
290         et1V |."etat" #= eqiEtatIn
291         et1V |."valeur" #= eqiValeurIn
292         (compareurResultats|>0) 'iff ' do
293             et1V |."u1_l" #= eqiEtatIn |."parameters.u">axeX
294             et1V |."axe_si_vrai" #= axeX
295         1 'elsiff ' do
296             et1V |."u1_l" #= eqiEtatIn |."parameters.u">axeY
297             et1V |."axe_si_vrai" #= axeY
298
299     let triEtagel2 = do
300         compareurOperandeAV|>1 #= et1 |."u1_l"
301         compareurOperandeBV|>1 #= et1 |."etat.parameters.u">axeZ
302         et1ReadV #= 1
303         et1WriteV #= 1
304         et2V |."etat" #= et1 |."etat"
305         et2V |."valeur" #= et1 |."valeur"
306         et2V |."axe" #= (compareurResultats|>1)?/
307             (et1 |."axe_si_vrai", axeZ)
308
309     let triEtagel3 = do
310         profondeurV <- var profondeurT
311         profondeurDetermineeV <- var bool
312         dernierFragmentV <- var bool
313         continuerARemonterV <- var bool
314         tmpV <- var (uint ((intlog cote) * profondeurMax))
315         iV <- var (uint 16)
316         lV <- var axeT
317
318         profondeurV #= et2 |."etat.profondeur"
319         profondeurDetermineeV #= 0
320         et2ReadV #= 1
321         sensV #= et2 |."etat.parameters.a">axeV
322         coordonneeIncrementeeV #=
323             et2 |."etat.parameters.position">axeV
324         tmpV #= coordonneeIncrementeeV
325         etatQualifieV #= et2 |."etat"
326
327         (reverse [1..profondeurMax]) 'for ' \p -> do
328             continuerARemonterV #= 1
329             [0..(intlog cote - 1)] 'for ' \c -> do
330                 iV #= profondeurMax /-/ p
331                 iV #= (iV ./ intlog cote) /+ / c
332                 continuerARemonterV #=
333                     continuerARemonterV
334                     /&/ ( tmpV|>iV /==/ 1
335                         /^/ sensV /==/ negatif )

```

```

336         tTermes|>axeX|>iV #= 0
337         tTermes|>axeY|>iV #= 0
338         tTermes|>axeZ|>iV #= 0
339     let pPetit = p /<=/ et2|. "etat.profondeur"
340     (bwn profondeurDetermineeV /&/ pPetit) 'iff ' do
341         (bwn continuerARemonterV) 'iff ' do
342             profondeurDetermineeV #= 1
343             profondeurV #= p
344         [0..intlog cote - 1] 'for ' \c -> do
345             iV #= profondeurMax - p
346             iV #= iV * intlog cote + c
347             calculerCompensation
348             iV profondeurDetermineeV
349
350     [axeX..axeZ] 'for ' \l ->
351         etatQualifieV|. "parametres.u">1 #=
352         etatQualifieV|. "parametres.u">1
353         /+| (tTermes|>1|>0)
354         /+| (tTermes|>1|>1)
355         /+| (tTermes|>1|>2)
356         /+| (tTermes|>1|>3)
357         /+| (tTermes|>1|>4)
358         /+| (tTermes|>1|>5)
359         /+| (tTermes|>1|>6)
360         /+| (tTermes|>1|>7)
361
362     coordonneeIncrementeeV #=
363         etatQualifieV|. "parametres.position">axeV
364
365     dernierFragmentV #= bwn profondeurDetermineeV
366
367     fragmentWriteV #= 1
368     fragmentV|. "valeur" #= et2|. "valeur"
369     fragmentV|. "u_e" #= et2|. "etat.parametres.u_e"
370     fragmentV|. "u_s" #= et2|. "etat.parametres.u">axeV
371     fragmentV|. "rayon" #= et2|. "etat.rayon"
372     fragmentV|. "contexte" #=
373         et2|. "etat.parametres.contexte_rayon"
374     fragmentV|. "premier_fragment" #=
375         et2|. "etat.parametres.nouveau_ne"
376     fragmentV|. "dernier_fragment" #= dernierFragmentV
377
378     etatQualifieWriteV #= 1
379     etatQualifieReinsererV #= bwn dernierFragmentV
380     etatQualifieV|. "profondeur" #= profondeurV
381     etatQualifieV|. "parametres.profondeur" #= profondeurV
382     etatQualifieV|. "parametres.nouveau_ne" #= 0
383     etatQualifieV|. "parametres.ue" #=
384         etatQualifieV|. "parametres.u">axeV
385
386     hauteurV #= (profondeurMax - profondeurV)*(intlog cote)
387     incrementV #= ((sensV==/positif)/?(1,-1))/<</hauteurV
388
389     etatQualifieV|. "parametres.position">axeV #=
390         (coordonneeIncrementeeV + incrementV)
391         /%/ (1 /<<</ ((intlog cote) ./ profondeurMax))
392     etatQualifieV|. "parametres.u">axeV #=
393         etatQualifieV|. "parametres.u">axeV
394         /+| (1 /<<</ ((intlog cote) ./ profondeurV))
395
396     let plongeon = do
397         let tableauDemisAuxT = array parametreT 11
398         let tableauDemisT = array tableauDemisAuxT 3
399         tableauDemis <- var tableauDemisT
400
401         hauteurV #= profondeurMax - etatV|. "profondeur"
402         hauteurV #= hauteurV ./ (intlog cote)

```

```

403 hauteurV ## hauteurV /-/ (iterationV /+/ 1)
404 [0..10] 'for ' \i -> [axeX..axeZ] 'for ' \l -> do
405   let i' = fromIntegral i
406   tableauDemis|>1|>i ##
407     etatV|. "parametres.u"|>1 /-/
408     (etatV|. "parametres.delta_u"|>1 />>/ i')
409 [axeX..axeZ] 'for ' \l -> do
410   shiftV ## etatV|. "profondeur" /./ (intlog cote)
411   shiftV ## shiftV /+/ iterationV /+/ 1
412   uDemiV ## tableauDemis|>1|>shiftV
413   compareurOperandeAV|>1 ##
414     etatV|. "parametres.u.e"
415   compareurOperandeBV|>1 ##
416     uDemiV
417   let neg = etatV|. "parametres.a"|>1 /==/ negatif
418   let pos = etatV|. "parametres.a"|>1 /==/ positif
419   (compareurResultatV|>1 /==/ neg) 'iff ' do
420     neg 'iff ' do
421       etatV|. "parametres.u"|>1 ## uDemiV
422       etatV|. "parametres.position"|>1|>hauteurV ##
423         1
424     1 'elsiff ' do
425       pos 'iff ' do
426         etatV|. "parametres.u"|>1 ## uDemiV
427         etatV|. "parametres.position"|>1|>hauteurV ##
428           0
429   etatV|. "parametres.nouveau.ne" ## 0
430
431 -- begin --
432
433 stall ## 0
434 readStall ## 0
435 writeStall ## 0
436
437 eqiReadV ## 0
438 et1WriteV ## 0
439 et1ReadV ## 0
440 et2WriteV ## 0
441 et2ReadV ## 0
442 etatQualifieWriteV ## 0
443 fragmentValideV ## 0
444
445 eqiValeurV ## eqiValeurIn
446 eqiEtatV ## eqiEtatIn
447 eqiValideV ## eqiValide
448 eqiAckV ## 0
449 modeV ## mode
450 modeV2 ## mode
451 [axeX..axeZ] 'for ' \l -> do
452   compareurOperandeAV |> 1 |> stVs ## 0
453   compareurOperandeAV |> 1 |> stVt ## 0
454 compareurResultatV ## compareurResultats
455 iterationV ## iteration
456 etatV ## etat
457 et1V ## et1
458 et1ValideV ## et1Valide
459 et1AckV ## 0
460 et2V ## et2
461 et2ValideV ## et2Valide
462 et2AckV ## 0
463 axeV ## et2|. "axe"
464 sensV ## 0
465 hauteurV ## 0
466 incrementV ## 0
467 [0..2] 'for ' \i -> compensationV |> i ## 0
468 coordonneeIncrementeeV ## 0
469 fragmentV|. "valeur" ## 0

```

```

470     fragmentV |." u_e" |> stVs #= 0
471     fragmentV |." u_e" |> stVt #= 0
472     fragmentV |." u_s" |> stVs #= 0
473     fragmentV |." u_s" |> stVt #= 0
474     fragmentV |." rayon" #= 0
475     fragmentV |." contexte.x" #= 0
476     fragmentV |." contexte.y" #= 0
477     fragmentV |." premier_fragment" #= 0
478     fragmentV |." dernier_fragment" #= 0
479     fragmentValideV #= fragmentValide
480     fragmentAckV #= fragmentAck
481     etatQualifieV #= etatQualifie
482     etatQualifieReinsererV #= 1
483     etatQualifieValideV #= etatQualifieValide
484     etatQualifieAckV #= etatQualifieAck
485     uDemiV |> stVs #= 0
486     uDemiV |> stVt #= 0
487     shiftV #= 0
488
489     — Combinatoire (FSM)
490     ((modeV /= pipelineTri) /&/ (iterationV /= 0)) 'iff' do
491         eqiReadV #= 1
492         (eqiEtatV |." voxel_subdivise") 'iff' do
493             modeV #= iterateurPlongeon
494             etatV #= eqiEtatV
495             iterationV #= 0
496         1 'elsiff' do
497             modeV #= pipelineTri
498
499     (modeV /= pipelineTri) 'iff' do
500         triEtage1
501         triEtage2
502         triEtage3
503         modeV2 #= modeV
504     1 'elsiff' do
505         plongeon
506         modeV2 #= modeV
507
508         (iterationV /= (intlog cote /- 1)) 'iff' do
509             etatQualifieWriteV #= 1
510             etatV |." profondeur" #= etatV |." profondeur" + 1
511             etatV |." parametres.profondeur" #=
512                 etatV |." profondeur"
513             etatQualifieV #= etatV
514             iterationV #= 0
515             modeV2 #= pipelineTri
516         1 'elsiff' do
517             iterationV #= iterationV /+ 1
518
519     — Abaissement des signaux de validité si nécessaire
520     et1ValideV #= et1ValideV /&/ bwn et1Ack
521     et2ValideV #= et2ValideV /&/ bwn et2Ack
522     fragmentValideV #= fragmentValideV /&/ bwn fragmentAck
523     etatQualifieValideV #= etatQualifieValideV
524         /&/ bwn etatQualifieAck
525
526     — Assignment des signaux (noeuds)
527     comparateurOperandeA #= comparateurOperandeAV
528     comparateurOperandeB #= comparateurOperandeBV
529     axe #= axeV
530     sens #= sensV
531     hauteur #= hauteurV
532     increment #= incrementV
533
534     — Combinatoire (contrôle)
535     (modeV /= pipelineTri) 'iff' do
536         readStall #= eqiReadV /&/ bwn eqiValide

```

```

537     writeStall #= et1WriteV /&/ et1ValideV
538     stall #= readStall /|| writeStall
539
540     (bwn stall) 'iff' do
541         mode #= modeV2
542         iteration #= iterationV
543         etat #= etatV
544         et1 #= et1V
545         et1ValideV #= et1ValideV /|| et1WriteV
546         eqiAckV #= eqiReadV
547
548     readStall #= et1ReadV /&/ bwn et1Valide
549     writeStall #= et2WriteV /&/ et2ValideV
550     stall #= readStall /|| writeStall
551
552     (bwn stall) 'iff' do
553         et2 #= et2V
554         et2ValideV #= et2ValideV /|| et2WriteV
555         et2AckV #= et2ReadV
556
557     readStall #= et2ReadV /&/ bwn et2Valide
558     writeStall #= 0
559     writeStall #= writeStall /||
560         ( etatQualifieWriteV
561           /&/ etatQualifieValideV)
562     writeStall #= writeStall /||
563         ( fragmentWriteV
564           /&/ fragmentValideV)
565     stall #= readStall /&/ writeStall
566
567     (bwn stall) 'iff' do
568         fragment #= fragmentV
569         fragmentValideV #=
570             fragmentValideV /|| fragmentWriteV
571         etatQualifie #= etatQualifieV
572         etatQualifieReinserer #= etatQualifieReinsererV
573         etatQualifieValideV #=
574             etatQualifieValideV /|| etatQualifieWriteV
575         et1AckV #= et2ReadV
576 1 'elsif' do — modeV /=/ iterateurPlongeon
577     readStall #= eqiReadV /&/ bwn eqiValide
578     writeStall #= etatQualifieWriteV /&/ etatQualifieValideV
579     stall #= readStall /|| writeStall
580
581     (bwn stall) 'iff' do
582         mode #= modeV2
583         iteration #= iteration
584         fragment #= fragmentV
585         fragmentValide #=
586             fragmentValideV /|| fragmentWriteV
587         etatQualifie #= etatQualifieV
588         etatQualifieReinserer #= etatQualifieReinsererV
589         etatQualifieValideV #=
590             etatQualifieValide /|| etatQualifieWriteV
591         etat #= etatV
592         eqiAckV #= eqiReadV
593
594 — Affectation des signaux de validité
595 eqiAck #= eqiAckV
596 et1Ack #= et1AckV
597 et1Valide #= et1ValideV
598 et2Ack #= et2AckV
599 et2Valide #= et2ValideV
600 fragmentValide #= fragmentValideV
601 etatQualifieValide #= etatQualifieValideV

```

C.3 Formulation dans le MoC RTL avec découpage automatique du pipeline

```

1  import Prelude
2  import Memnoc.Base
3  import Memnoc.Exploration
4  import Memnoc.Transfer
5  import Memnoc.Engines.StdRTL
6  import Memnoc.Output.Chronogram
7
8  -----
9  -- Declaration de types et constantes --
10 -----
11
12 -- Les types polymorphes doivent toujours être explicites avec Hindley-Milner:
13 profondeurMax :: Num a => a
14
15 paramSize = 18
16 profondeurMax = 4
17 rayonsMax = 256
18 cote = 4
19
20 profondeurT = uint (intlog (profondeurMax+1))
21 rayonT = uint (intlog rayonsMax)
22 coordonneeT = uint (intlog (cote ^ profondeurMax))
23
24 coordonneePixelT = uint 12
25
26 directionT = uint (intlog (6 + 1))
27 directionO = 0
28 directionPx = 1
29 directionPy = 2
30 directionPz = 3
31 directionNx = 4
32 directionNy = 5
33 directionNz = 6
34
35 stT = uint 1
36 stVs = 0
37 stVt = 1
38
39 scalaireT = sint paramSize
40 parametreT = array scalaireT 2
41
42 sensT = sint 2
43 positif = 1
44 negatif = -1
45
46 axeT = uint (intlog 3)
47 axeX :: Num a => a
48 axeY :: Num a => a
49 axeZ :: Num a => a
50 axeX = 0
51 axeY = 1
52 axeZ = 2
53
54 aT = array sensT 3
55 tripletParametresT = array parametreT 3
56 positionT = array coordonneeT 3
57
58 contexteRayonT = struct
59     [ "x" :-: coordonneePixelT
60     , "y" :-: coordonneePixelT
61     ]
62

```

```

63 etatParametresT = struct
64     [ "nouveau_ne" -:- bool
65       , "position" -:- positionT
66       , "u" -:- tripletParametresT
67       , "u_e" -:- parametreT
68       , "a" -:- aT
69       , "delta_u" -:- tripletParametresT
70       , "contexte_rayon" -:- contexteRayonT
71       , "profondeur" -:- profondeurT
72     ]
73
74 etatQualifieT = struct
75     [ "rayon" -:- rayonT
76       , "profondeur" -:- profondeurT
77       , "voxel_subdivise" -:- bool
78       , "parametres" -:- etatParametresT
79     ]
80
81 fragmentT = struct
82     [ "valeur" -:- uint 15
83       , "u_e" -:- parametreT
84       , "u_s" -:- parametreT
85       , "rayon" -:- rayonT
86       , "contexte" -:- contexteRayonT
87       , "premierFragment" -:- bool
88       , "dernierFragment" -:- bool
89     ]
90
91 accesCacheT = struct
92     [ "premier_acces_pour_ce_rayon" -:- bool
93       , "position" -:- positionT
94       , "profondeur" -:- profondeurT
95     ]
96
97 reponseCacheT = struct
98     [ "voxelSubdivise" -:- bool
99       , "valeur" -:- uint 15
100     ]
101
102 -----
103 -- Procedures et fonctions --
104 -----
105
106 -- Non synthetisable tel quel:
107 ceillog2 :: Transfer Type -> Transfer Type
108 ceillog2 n = (n /<=/ 1) /?/ (0, 1 /+ / ceillog2(n />>/ 1))
109
110 -- Pour info, version synthétisable naïve:
111 ceillog2' :: Transfer Type -> Transfer Type
112 ceillog2' n =
113     let
114         aux :: Transfer Type -> Integer -> Transfer Type
115         aux n d | d == 0 = 0
116         aux n d = (n /<=/ 1) /?/ (0, 1 /+ / aux (n />>/ 1) (d - 1))
117     in do
118         s <- getSize n
119         aux n s
120
121
122 -----
123 -- Trieur plongeur --
124 -----
125
126
127 trieurPlongeur :: RTL em
128     => Transfer Type -> Transfer Type -> Transfer Type -> Transfer Type
129     -> Transfer Type -> Transfer Type -> Transfer Type

```

```

130     -> Transfer Type -> Transfer Type
131     -> Transfer Type -> Transfer Type
132     -> em ()
133
134   trieurPlongeur
135     eqiValeurIn eqiEtatIn eqiValide eqiAck
136     fragmentOut fragmentValideOut fragmentAck
137     etatQualifieOut etatQualifieReinsérerOut
138     etatQualifieValideOut etatQualifieAck
139   = do
140     let modeT = uint 1
141         let pipelineTri = 0
142         let iterateurPlongeur = 1
143
144     mode <- reg modeT
145
146     let compensationT = array scalaireT 3
147         let tableauOperandesT = array parametreT 3
148         let tableauResultatsT = array bool 3
149
150     compareurOperandeA <- node tableauOperandesT
151     compareurOperandeB <- node tableauOperandesT
152     compareurResultats <- node tableauResultatsT
153
154     iteration <- reg (uint 4)
155     etat <- reg etatQualifieT
156
157     sens <- node sensT
158     hauteur <- node (uint 4)
159     increment <- node (uint 24)
160
161     fragment <- reg fragmentT
162     fragmentValide <- reg bool
163     etatQualifie <- reg etatQualifieT
164     etatQualifieReinsérer <- reg bool
165     etatQualifieValide <- reg bool
166
167     — begin —
168
169     fragmentOut #= fragment
170     fragmentValideOut #= fragmentValide
171     etatQualifieOut #= etatQualifie
172     etatQualifieReinsérerOut #= etatQualifieReinsérer
173     etatQualifieValideOut #= etatQualifieValide
174
175     let (tri :: Transfer' (Transfer Type, Transfer Type, Transfer Type)) = do
176       — interface —
177       eqiEtatIn <- node etatQualifieT
178       etatQualifie <- node etatQualifieT
179       fragment <- node fragmentT
180
181       — combinatoire —
182       uLL <- node parametreT
183       axeSiVrai <- node axeT
184       axe <- node axeT
185       coordonneeIncrementee <- node coordonneeT
186
187       let termesT = array parametreT 8
188           let tTermesT = array termesT 3
189           tTermes <- node tTermesT
190
191       let calculerCompensation i profondeurDeterminee = do
192         value <- node bool
193         p <- node (uint 32)
194         [0..2] 'for' \l -> do
195           c1 <- node bool
196           c2 <- node bool

```

```

197     cond <- node bool
198     c1 #= eqiEtatIn |." etat . parametres . position" |> 1 |> i
199     c2 #= eqiEtatIn |." etat . parametres . a" |> 1 /==/ positif
200     value #= c1 ^/ c2;
201     cond #=
202         ( i /<=/ ( profondeurMax * intlog ( cote ) - 2 ) )
203         /&/ bwn profondeurDeterminee
204     mux' cond ( do
205         p #= ( intlog ( cote ) * profondeurMax ) /-/ i
206         mux' value ( do
207             let dul = etatQualifie
208                 |." parametres . delta_u"
209             tTermes|>1|>i|>stVs #=
210                 ( dul|>stVs ) / / ( 1 / << / p )
211             tTermes|>1|>i|>stVs #=
212                 ( dul|>stVt ) / / ( 1 / << / p ) )
213             ( return () )
214             etatQualifie |." parametres . position"
215                 |>1|>i #= 0 ) ( return () )
216
217     compareurOperandeA|>0 #= eqiEtatIn |." parametres . u" |>axeX
218     compareurOperandeB|>0 #= eqiEtatIn |." parametres . u" |>axeY
219     mux' ( compareurResultats|>0 )
220     ( do
221         u1L #= eqiEtatIn |." parametres . u" |>axeX
222         axeSiVrai #= axeX )
223     ( do
224         u1L #= eqiEtatIn |." parametres . u" |>axeY
225         axeSiVrai #= axeY )
226
227     compareurOperandeA|>1 #= u1L
228     compareurOperandeB|>1 #= eqiEtatIn |." etat . parametres . u" |>axeZ
229     axe #= ( compareurResultats|>1 ) ? / ( axeSiVrai , axeZ )
230
231     profondeur <- node profondeurT
232     profondeurDeterminee <- node bool
233     dernierFragment <- node bool
234     continuerARemonter <- node bool
235     tmp <- node ( uint ( ( intlog cote ) * profondeurMax ) )
236     i <- node ( uint 16 )
237     l <- node axeT
238
239     profondeur #= eqiEtatIn |." etat . profondeur"
240     profondeurDeterminee #= 0
241     sens #= eqiEtatIn |." etat . parametres . a" |>axe
242     coordonneeIncrementee #=
243         eqiEtatIn |." etat . parametres . position" |>axe
244     tmp #= coordonneeIncrementee
245     etatQualifie #= eqiEtatIn |." etat"
246
247     ( reverse [ 1 .. profondeurMax ] ) 'for ' \p -> do
248         continuerARemonter #= 1
249         [ 0 .. ( intlog cote - 1 ) ] 'for ' \c -> do
250             i #= profondeurMax /-/ p
251             i #= ( i /./ intlog cote ) /+ / c
252             continuerARemonter #=
253                 continuerARemonter
254                 /&/ ( tmp|>i /==/ 1
255                     / ^ / sens /==/ negatif )
256             tTermes|>axeX|>i #= 0
257             tTermes|>axeY|>i #= 0
258             tTermes|>axeZ|>i #= 0
259         let pPetit = p /<=/ eqiEtatIn |." etat . profondeur"
260         mux' ( bwn profondeurDeterminee /&/ pPetit ) ( do
261             mux' ( bwn continuerARemonter ) ( do
262                 profondeurDeterminee #= 1
263                 profondeur #= p ) ( return () )

```

```

264         [0..intlog cote - 1] 'for' \c -> do
265             i  $\#$  profondeurMax - p
266             i  $\#$  i * intlog cote + c
267             calculerCompensation
268             i profondeurDeterminee) (return())
269
270     [axeX..axeZ] 'for' \l ->
271         etatQualifie |."parametres.u">|1  $\#$ 
272             etatQualifie |."parametres.u">|1
273             /+/ (tTermes|>1|>0)
274             /+/ (tTermes|>1|>1)
275             /+/ (tTermes|>1|>2)
276             /+/ (tTermes|>1|>3)
277             /+/ (tTermes|>1|>4)
278             /+/ (tTermes|>1|>5)
279             /+/ (tTermes|>1|>6)
280             /+/ (tTermes|>1|>7)
281
282     coordonneeIncrementee  $\#$ 
283         etatQualifie |."parametres.position">axe
284
285     dernierFragment  $\#$  bwn profondeurDeterminee
286
287     —fragmentWriteV  $\#$  1
288     fragment |."valeur"  $\#$  eqiEtatIn |."valeur"
289     fragment |."u_e"  $\#$  eqiEtatIn |."etat.parametres.u_e"
290     fragment |."u_s"  $\#$  eqiEtatIn |."etat.parametres.u">axe
291     fragment |."rayon"  $\#$  eqiEtatIn |."etat.rayon"
292     fragment |."contexte"  $\#$ 
293         eqiEtatIn |."etat.parametres.contexte.rayon"
294     fragment |."premier_fragment"  $\#$ 
295         eqiEtatIn |."etat.parametres.nouveau_ne"
296     fragment |."dernier_fragment"  $\#$  dernierFragment
297
298     —etatQualifieWriteV  $\#$  1
299     etatQualifieReinsérer  $\#$  bwn dernierFragment
300     etatQualifie |."profondeur"  $\#$  profondeur
301     etatQualifie |."parametres.profondueur"  $\#$  profondeur
302     etatQualifie |."parametres.nouveau_ne"  $\#$  0
303     etatQualifie |."parametres.ue"  $\#$ 
304         etatQualifie |."parametres.u">axe
305
306     hauteur  $\#$  (profondeurMax - profondeur)*(intlog cote)
307     increment  $\#$  ((sens $\#$ positif)?/(1,-1))/<</hauteur
308
309     etatQualifie |."parametres.position">axe  $\#$ 
310         (coordonneeIncrementee + increment)
311         /%/ (1 /<<</ ((intlog cote) /./ profondeurMax))
312     etatQualifie |."parametres.u">axe  $\#$ 
313         etatQualifie |."parametres.u">axe
314         /+/ (1 /<<</ ((intlog cote) /./ profondeur))
315
316     return (eqiEtatIn , etatQualifie , fragment)
317
318 ( inPValide , inPAck , outPValide , outPAck ,
319   (eqiEtatInP , etatQualifieOutP , fragmentOutP) ) <- stdPipeline 3 tri
320
321 inPValide  $\#$  eqiValide /&/ (mode / $\#$ / pipelineTri)
322 outPAck  $\#$  fragmentAck /&/ etatQualifieAck /&/ (mode / $\#$ / pipelineTri)
323 eqiEtatInP  $\#$  eqiEtatIn
324
325 0 'process' do
326     a <- var (sint 32)
327     b <- var (sint 32)
328     signe <- var bool
329
330     [0..2] 'for' \i -> do

```

```

331     a #= (comparateurOperandeA |> i |> stVs)
332     a #= a ./ (comparateurOperandeB |> i |> stVt)
333     b #= (comparateurOperandeB |> i |> stVs)
334     b #= b ./ (comparateurOperandeA |> i |> stVt)
335     signe #=
336         ((comparateurOperandeA |> i |> stVt) /</ 0)
337         / ^ / ((comparateurOperandeB |> i |> stVt) /</ 0)
338     (comparateurResultats |> i) #= (a /<=/ b) / ^ / signe
339
340 0 'process' do
341     stall <- var bool
342     readStall <- var bool
343     writeStall <- var bool
344
345     eqiReadV <- var bool
346     etatQualifieWriteV <- var bool
347     fragmentWriteV <- var bool
348
349     eqiValeurV <- var (uint 15)
350     eqiEtatV <- var etatQualifieT
351     eqiValideV <- var bool
352     eqiAckV <- var bool
353     modeV <- var modeT
354     modeV2 <- var modeT
355     comparateurOperandeAV <- var tableauOperandesT
356     comparateurOperandeBV <- var tableauOperandesT
357     comparateurResultatV <- var tableauResultatsT
358     iterationV <- var (uint 4)
359     etatV <- var etatQualifieT
360     sensV <- var sensT
361     hauteurV <- var (uint 4)
362     incrementV <- var (uint 24)
363     compensationV <- var compensationT
364     fragmentV <- var fragmentT
365     fragmentValideV <- var bool
366     fragmentAckV <- var bool
367     etatQualifieV <- var etatQualifieT
368     etatQualifieReinsererV <- var bool
369     etatQualifieValideV <- var bool
370     etatQualifieAckV <- var bool
371     uDemiV <- var parametreT
372     shiftV <- var (uint 8)
373
374 let plongeon = do
375     let tableauDemisAuxT = array parametreT 11
376     let tableauDemisT = array tableauDemisAuxT 3
377     tableauDemis <- var tableauDemisT
378
379     hauteurV #= profondeurMax - etatV |."profondeur"
380     hauteurV #= hauteurV ./ (intlog cote)
381     hauteurV #= hauteurV /- / (iterationV /+ / 1)
382     [0..10] 'for' \i -> [axeX..axeZ] 'for' \l -> do
383         —let i' = fromIntegral i
384         —tableauDemis|>l|>i #=
385             etatV |."parametres.u"|>l /- /
386             (etatV |."parametres.delta_u"|>l />> / i')
387         return ()
388     [axeX..axeZ] 'for' \l -> do
389         shiftV #= etatV |."profondeur" ./ (intlog cote)
390         shiftV #= shiftV /+ / iterationV /+ / 1
391         uDemiV #= tableauDemis|>l|>shiftV
392         comparateurOperandeAV|>l #=
393             etatV |."parametres.u_e"
394         comparateurOperandeBV|>l #=
395             uDemiV
396         let neg = etatV |."parametres.a"|>l /== / negatif
397         let pos = etatV |."parametres.a"|>l /== / positif

```

```

398         (comparateurResultatV|>1 /==/ neg) 'iff' do
399             neg 'iff' do
400                 etatV|. "parametres.u"|>1 #= uDemiV
401                 etatV|. "parametres.position"|>1|>hauteurV #=
402                     1
403             1 'elsiff' do
404                 pos 'iff' do
405                     etatV|. "parametres.u"|>1 #= uDemiV
406                     etatV|. "parametres.position"|>1|>hauteurV #=
407                         0
408                 etatV|. "parametres.nouveau.ne" #= 0
409
410     -- begin --
411
412     stall #= 0
413     readStall #= 0
414     writeStall #= 0
415
416     eqiReadV #= 0
417     etatQualifieWriteV #= 0
418     fragmentValideV #= 0
419
420     eqiValeurV #= eqiValeurIn
421     eqiEtatV #= eqiEtatIn
422     eqiValideV #= eqiValide
423     eqiAckV #= 0
424     modeV #= mode
425     modeV2 #= mode
426     [axeX..axeZ] 'for' \l -> do
427         comparateurOperandeAV |> 1 |> stVs #= 0
428         comparateurOperandeAV |> 1 |> stVt #= 0
429     comparateurResultatV #= comparateurResultats
430     iterationV #= iteration
431     etatV #= etat
432     sensV #= 0
433     hauteurV #= 0
434     incrementV #= 0
435     [0..2] 'for' \i -> compensationV |> i #= 0
436     fragmentV|. "valeur" #= 0
437     fragmentV|. "u_e" |> stVs #= 0
438     fragmentV|. "u_e" |> stVt #= 0
439     fragmentV|. "u_s" |> stVs #= 0
440     fragmentV|. "u_s" |> stVt #= 0
441     fragmentV|. "rayon" #= 0
442     fragmentV|. "contexte.x" #= 0
443     fragmentV|. "contexte.y" #= 0
444     fragmentV|. "premier_fragment" #= 0
445     fragmentV|. "dernier_fragment" #= 0
446     fragmentValideV #= fragmentValide
447     fragmentAckV #= fragmentAck
448     etatQualifieV #= etatQualifie
449     etatQualifieReinsererV #= 1
450     etatQualifieValideV #= etatQualifieValide
451     etatQualifieAckV #= etatQualifieAck
452     uDemiV |> stVs #= 0
453     uDemiV |> stVt #= 0
454     shiftV #= 0
455
456     -- Combinatoire (FSM)
457     ((modeV /==/ pipelineTri) /&/ (iterationV /==/ 0)) 'iff' do
458         eqiReadV #= 1
459         (eqiEtatV|. "voxel_subdivise") 'iff' do
460             modeV #= iterateurPlongeon
461             etatV #= eqiEtatV
462             iterationV #= 0
463         1 'elsiff' do
464             modeV #= pipelineTri

```

```

465
466 (modeV /=/ pipelineTri) 'iff' do
467   etatQualifieV #= etatQualifieOutP
468   etatQualifieWriteV #= outPValide
469   fragmentV #= fragmentOutP
470   fragmentWriteV #= outPValide
471   eqiAckV #= inPAck
472   modeV2 #= modeV
473 1 'elsiff' do
474   plongeon
475   modeV2 #= modeV
476
477   (iterationV /=/ (intlog cote /-/ 1)) 'iff' do
478     etatQualifieWriteV #= 1
479     etatV |."profondeur" #= etatV |."profondeur" + 1
480     etatV |."parametres.profondeur" #=
481       etatV |."profondeur"
482     etatQualifieV #= etatV
483     iterationV #= 0
484     modeV2 #= pipelineTri
485 1 'elsiff' do
486     iterationV #= iterationV /+ 1
487
488 — Abaissement des signaux de validité si nécessaire
489 fragmentValideV #= fragmentValideV /&/ bwn fragmentAck
490 etatQualifieValideV #= etatQualifieValideV
491 /&/ bwn etatQualifieAck
492
493 — Assignment des signaux (noeuds)
494 compareteurOperandeA #= compareteurOperandeAV
495 compareteurOperandeB #= compareteurOperandeBV
496 sens #= sensV
497 hauteur #= hauteurV
498 increment #= incrementV
499
500 — Combinatoire (contrôle)
501 (modeV /=/ pipelineTri) 'iff' do
502   readStall #= eqiReadV /&/ bwn eqiValide
503   writeStall #= 0
504   writeStall #= writeStall /||/
505     ( etatQualifieWriteV
506       /&/ etatQualifieValideV)
507   writeStall #= writeStall /||/
508     ( fragmentWriteV
509       /&/ fragmentValideV)
510   stall #= readStall /&/ writeStall
511
512   (bwn stall) 'iff' do
513     fragment #= fragmentV
514     fragmentValideV #=
515       fragmentValideV /||/ fragmentWriteV
516     etatQualifie #= etatQualifieV
517     etatQualifieReinsérer #= etatQualifieReinsérerV
518     etatQualifieValideV #=
519       etatQualifieValideV /||/ etatQualifieWriteV
520     eqiAckV #= eqiReadV
521 1 'elsiff' do — modeV /=/ itérateurPlongeon
522   readStall #= eqiReadV /&/ bwn eqiValide
523   writeStall #= etatQualifieWriteV /&/ etatQualifieValideV
524   stall #= readStall /||/ writeStall
525
526   (bwn stall) 'iff' do
527     mode #= modeV2
528     iteration #= iteration
529     fragment #= fragmentV
530     fragmentValide #=
531       fragmentValideV /||/ fragmentWriteV

```

ANNEXE C. COPROCESSEUR POUR LA TRAVERSÉE DE GRILLES HIÉRARCHIQUES

```
532          etatQualifie #= etatQualifieV
533          etatQualifieReinserer #= etatQualifieReinsererV
534          etatQualifieValideV #=
535              etatQualifieValide || etatQualifieWriteV
536          etat #= etatV
537          eqiAckV #= eqiReadV
538
539      — Affectation des signaux de validité
540      eqiAck #= eqiAckV
541      fragmentValide #= fragmentValideV
542      etatQualifieValide #= etatQualifieValideV
```


Bibliographie

- [1] Amba open specifications.
<http://www.arm.com/products/system-ip/amba/amba-open-specifications.php>.
- [2] Bluespec.
<http://www.bluespec.com/>.
- [3] The hierarchy of numeric typeclasses in haskell.
<http://nix-tips.blogspot.com/2010/07/hierarchy-of-numeric-typeclasses-in.html>.
- [4] International technology roadmap for semiconductors.
<http://public.itrs.net>.
- [5] Ieee standard system c language reference manual. *IEEE Std 1666-2005*, 2006.
- [6] Accellera Organization, Inc. *SystemVerilog 3.1a language reference manual*, 2004.
- [7] Tristan O. R. Allwood, Simon L. Peyton Jones, and Susan Eisenbach. Finding the needle : stack traces for ghc. In Stephanie Weirich, editor, *Haskell*, pages 129–140. ACM, 2009.
- [8] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, 1987.
- [9] Charles André, Frédéric Mallet, Aamir Mehmood Khan, and Robert De Simone. Modeling spirit ip-xact with uml marte.
- [10] Arvind. Bluespec : A language for hardware design, simulation, synthesis and verification invited talk. In *MEMOCODE '03 : Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.
- [11] G. Ascia, V. Catania, and M. Palesi. A ga-based design space exploration framework for parameterized system-on-a-chip platforms. *Evolutionary Computation, IEEE Transactions on*, 8(4) :329 – 346, 2004.
- [12] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
- [13] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. Clash : Structural descriptions of synchronous hardware using haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design : Architectures, Methods and Tools, Lille, France*, pages 714–721, USA, September 2010. IEEE Computer Society.
- [14] Stephan Beal. Supermacros : powerful, maintainable preprocessor macros in c++. August 2004.
- [15] Pete Becker. Working draft, standard for programming language c++. Technical Report N3242=11-0012, February 2011.
- [16] L. Benini and G. De Micheli. Networks on chips : a new soc paradigm. *Computer*, 35(1) :70–78, Jan 2002.
- [17] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava : Hardware design in haskell. In *ICFP '98 : Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, NY, USA, 1998. ACM Press.

- [18] M.S. Bright and T. Arslan. Synthesis of low-power dsp systems using a genetic algorithm. *Evolutionary Computation, IEEE Transactions on*, 5(1) :27–40, February 2001.
- [19] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *In POPL '05 : Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.
- [20] Franklin C. Crow. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11 :242–248, July 1977.
- [21] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.
- [22] Jean-Luc Dekeyser, Imran Rafiq Quadri, and Abdoulaye Gamatié. Tutorial : Using the UML profile for MARTE to MPSoC co-design dedicated to signal processing. In *Colloque International Télécom 2009 et 6èmes Journées JFMMA*, Agadir, Morocco, March 2009.
- [23] Thiago R. dos Santos, Daniel D. Abdala, and Aldo von Wangenheim. Three-dimensional visualization of radiological images using octrees. In *CBMS '04 : Proceedings of the 17th IEEE Symposium on Computer-Based Medical Systems (CBMS'04)*, page 44, Washington, DC, USA, 2004. IEEE Computer Society.
- [24] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *MICRO 25*, 1992.
- [25] N. Gac, S.Mancini, M.Desvignes, and D.Houzet. High speed 3d tomography on cpu, gpu and fpga. *EURASIP Journal on Embedded systems*, Special issue : Design and Architectures for Signal Image Processing (to be published), 2008.
- [26] Jiri Gaisler. A structured vhdl design method. In *Fault-tolerant microprocessors for space applications*, pages 41–50.
- [27] T. Givargis and F. Vahid. Platune : a tuning framework for system-on-a-chip platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(11) :1317 – 1327, November 2002.
- [28] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications*, 4(10) :15–22, October 1984.
- [29] Richard Goering. Platform-based design : a choice, not a panacea. *EE Times*, Sep 2002.
- [30] Brian Goetz. Jsr 335 : Lambda expressions for the java™ programming language. <http://jcp.org/en/jsr/detail?id=335>, 2010.
- [31] Henri Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6) :623–629, June 1971.
- [32] T. Guenther, C. Poliwoda, C. Reinhart, J. Hesser, R. Maenner, H. Meinzer, and H. Baur. Virim : A massively parallel processor for real-time volume visualization in medicine. Technical report, 1995.
- [33] HaskellWiki. Parameter order. http://www.haskell.org/haskellwiki/index.php?title=Parameter_order&oldid=33815, February 2010.
- [34] V. Havran. A Summary of Octree Ray Traversal Algorithms. *Ray Tracing News*, 12(2) :cca 10 pages, December 1999. Available from <http://www.acm.org/tog/resources/RTNews/html/rtnv12n2.html>.
- [35] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.
- [36] Paul S Heckbert. Survey of texture mapping. *IEEE Comput. Graph. Appl.*, 6 :56–67, November 1986.
- [37] Fernando Herrera and Eugenio Villar. A framework for heterogeneous specification and design of electronic embedded systems in systemc. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3) :1–31, 2007.

- [38] Ralf Hinze and Ross Paterson. Finger trees : a simple general-purpose data structure. *J. Funct. Program.*, 16 :197–217, March 2006.
- [39] W. A. Howard. *The formulae-as-types notion of construction*, pages 480–490. Academic Press, London-New York, 1980.
- [40] Paul Hudak and John Peterson. A gentle introduction to haskell 98. <http://www.haskell.org/tutorial/haskell-98-tutorial.pdf>, October 1999.
- [41] IEEE. *Std 1076-2000 : IEEE Standard VHDL Language Reference Manual*, 2000.
- [42] IEEE. *Std 1364-2001 : IEEE Standard Verilog Hardware Description Language*, 2001.
- [43] IEEE. *Std 1365-2005 : IEEE Standard Verilog Hardware Description Languages*, 2005.
- [44] IEEE. *Std 1076-2008 : IEEE Standard VHDL Language Reference Manual*, 2008.
- [45] IEEE. *Std 1685-2009 : IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, 2009.
- [46] Shahabuddin L. Inamdar. Vhdl coding style guidelines and synthesis : A comparative approach. Master’s thesis, University of South Florida, October 2004.
- [47] Jaakko Järvi and John Freeman. Lambda functions for c++0x. In *Proceedings of the 2008 ACM symposium on Applied computing, SAC ’08*, pages 178–183, New York, NY, USA, 2008. ACM.
- [48] Simon Peyton Jones et al. Haskell 98 language and libraries : The revised report. <http://haskell.org/onlinereport/index.html>, December 2002.
- [49] T. Kam and P.A. Subrahmanyam. Comparing layouts with hdl models : a formal verification technique. In *Computer Design : VLSI in Computers and Processors, 1992. ICCD ’92. Proceedings., IEEE 1992 International Conference on*, pages 588–591, October 1992.
- [50] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language*. The Khronos Group Inc., July 2010.
- [51] Paul E. Kinahan, Michel Defrise, and Rolf Clackdoyle. Analytic image reconstruction methods. *Emission Tomography : The Fundamentals of PET and SPECT*, pages 421–442, 2004.
- [52] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(Annual Conference Series) :451–458, 1994.
- [53] Sang-Joon Lee and K. Raahemifar. Fpga placement optimization methodology survey. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 001981–001986, May 2008.
- [54] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.12 documentation and user’s manual, 2010.
- [55] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10 :707, 1966.
- [56] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16, 1994.
- [57] I. Sobanski M. Zys, E. Vaumorin. Straightforward ip integration with ip-xact rtl-tlm switching. 2008.
- [58] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3) :153–166, 1990.
- [59] S. Mancini and R. Pacalet. Etude du lancer de rayon en géométrie projective. In *Journées Courbes, Surfaces et Algorithmes*, September 1999.
- [60] Stéphane Mancini, Lionel Pierrefeu, Zahir Larabi, and Yves Mathieu. Calibrating a predictive cache emulator for soc design. In *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, pages 273–280, 2010.

- [61] Deepak Mathaikutty, Hiren Patel, Sandeep Shukla, and Axel Jantsch. Sml-sys : a functional framework with multiple models of computation for modeling heterogeneous system. *Design Automation for Embedded Systems*, 12 :1–30, 2008.
- [62] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, rev sub edition, May 1997.
- [63] Gordon Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting*, 1975.
- [64] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.
- [65] J. Garrett Morris and Mark P. Jones. Instance chains : type class programming without overlapping instances. *SIGPLAN Not.*, 45 :375–386, September 2010.
- [66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, 1966.
- [67] Henrik Nilsson. Declarative debugging for lazy functional languages, 1998.
- [68] Ulf Norell. Dependently typed programming in Agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.
- [69] NVIDIA. *CUDA Programming Guide 2.0*, July 2008.
- [70] NVIDIA. *PTX : Parallel Thread Execution ISA Version 1.2*, June 2008.
- [71] NVIDIA. *CUDA C Programming Best Practices Guide 2.3*, July 2009.
- [72] NVIDIA. *CUDA Reference Manual 2.3*, July 2009.
- [73] NVIDIA. *PTX : Parallel Thread Execution ISA Version 2.0*, January 2010.
- [74] Hiren D. Patel, Sandeep K. Shukla, and Reinaldo A. Bergamaschi. Heterogeneous behavioral hierarchy extensions for systemc. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(4) :765–780, 2007.
- [75] L. Pirmez, A. Pedroza, M. Rahmouni, A. Mesquita, P. Kission, and A.A. Jerraya. Analysis of different protocol description styles in vhdl for high-level synthesis. In *Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European*, pages 490–495, September 1996.
- [76] Bernard Pope and Lee Naish. Practical aspects of declarative debugging in haskell 98. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '03, pages 230–240, New York, NY, USA, 2003. ACM.
- [77] Bernard Pope and Supervisor Lee Naish. Buddha - a declarative debugger for haskell, 1998.
- [78] Rodric M. Rabbah and Krishna V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Trans. Embed. Comput. Syst.*, 2 :186–218, May 2003.
- [79] Raúl Regidor, Félix Tobajas, Valentín Armas, Sebastián López, Gustavo M. Callicó, José López, and Roberto Sarmiento. Topological optimization of memory location in noc based multimedia systems using arteris nocexplorer. In *Conference on Design of Circuits and Integrated Systems*, 2007.
- [80] J. Revelles, C. Ureña, and M. Lastra. An efficient parametric algorithm for octree traversal, 2000.
- [81] John C. Reynolds. An introduction to polymorphic lambda calculus. In *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, 1994.
- [82] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1) :17–32, January 2004.
- [83] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EE Times*, Feb 2002.
- [84] Alberto L. Sangiovanni-Vincentelli. Quo vadis sld : Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3) :467–506, March 2007.

- [85] Tim Sheard. Another look at hardware design languages, December 2005. Available from : <http://www.cs.pdx.edu/~sheard/> .
- [86] Tim Sheard. Putting curry-howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*. ACM Press, September 2005.
- [87] Tim Sheard. Types and hardware description languages. In Andrew Martin, Carl Seger, and Mary Sheeran, editors, *Hardware Design and Functional Languages, A satellite event of ETAPS*, volume 5161, 2007. Available from : <http://www.cs.pdx.edu/~sheard/> .
- [88] M. Sheeran. Hardware design and functional programming : a perfect match. *Journal of Universal Computer Science*, 11(7) :1135–1158, 2005.
- [89] Gerard J.M. Smit, Jan Kuper, and Christiaan P.R. Baaij. A mathematical approach towards hardware design. In P.M. Athanas, J. Becker, J. Teich, and I. Verbauwhede, editors, *Dynamically Reconfigurable Architectures*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, December 2010. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI).
- [90] Wilson Snyder. The verilog preprocessor : Force for ‘good and ‘evil. In *Proceedings of SNUG Boston 2010*, September 2010.
- [91] J. Spackman and P. Willis. The smart navigation of a ray through an oct-tree. *Computer and Graphics*, 15(2) :185–194, 1991.
- [92] J.E. Stone, D. Gohara, and Guochun Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3) :66–73, 2010.
- [93] Radoslaw Szymanek, Francky Catthoor, and Krzysztof Kuchcinski. Time-energy design space exploration for multi-layer memory architectures. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1, DATE '04*, pages 10318–, Washington, DC, USA, 2004. IEEE Computer Society.
- [94] The Coq Development Team. The coq proof assistant - reference manual version 8.3. Technical report, INRIA, 2010.
- [95] Tomasz Toczek and Stéphane Mancini. Efficient memory management for uniform and recursive grid traversal. In Guy Gogniat, Dragomir Milojevic, Adam Morawiec, and Ahmet Erdogan, editors, *Algorithm-Architecture Matching for Signal and Image Processing*, volume 73 of *Lecture Notes in Electrical Engineering*, pages 27–51. Springer Netherlands, 2011.
- [96] E. Villar and P. Sanchez. Cad tools for synthesis. In *Industrial Electronics, 1995. ISIE '95., Proceedings of the IEEE International Symposium on*, volume 1, pages 27–32 vol.1, 10-14 1995.
- [97] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.
- [98] Jonathan Woodruff, Greg Chadwick, and Simon Moore. Cache Tracker : A Key Component for Flexible Many-Core Simulation on FPGAs. In Omar Hammami and Sandra Larrabee, editors, *WARP - 5th Annual Workshop on Architectural Research Prototyping*, Saint Malo France, 2010.
- [99] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 67–77, New York, NY, USA, 2006. ACM.