Une syntaxe améliorée pour la description de circuits

Aliaume Lopez
Sous la direction du Professeur Dan R. Ghica
University of Birmingham, UK

1 Juin 2016 – 16 Juillet 2016

Table des matières

Table des figures

⊬

1 Préambule

1.1 L'université de Birmingham

L'université de Birmingham (University of Birmingham) n'est pas à confondre avec « Birmingham University » située dans la même ville en Angleterre, mais n'ayant presque rien d'autre en commun. Elle regroupe plusieurs départements répartis dans les différents bâtiments du campus plutôt vaste situé au sud de la ville.

Elle est desservie par les transports en communs (bus, train), bien qu'il soit possible d'y aller en vélo ou à pied depuis le quartier étudiant.

1.2 L'environnement de recherche

Mon maître de stage était le professeur Dan R. Ghica du département d'informatique de l'université de Birmingham. Il supervisait d'autres étudiants en même temps que moi, mais ils étaient plus âgés et préparaient leur thèse ou un post-doctorat. Malgré mon inexpérience, j'ai été très bien intégré dans le groupe de travail qui se réunissait le lundi matin, ainsi que dans le *Theory Group* qui réunit les étudiants du département d'informatique le mardi après-midi pour présenter les avancées de chacun.

Les sujets de recherches des différents étudiants, bien que diférents, avaient tous une orientation similaire vers la sémantique et la théorie des graphes. C'est pourquoi les réunions étaient bénéfiques pour avoir une compréhension plus globale du sujet.

Durant le stage, je me suis vu fournir un bureau muni d'un tableau véléda, et d'une carte afin de pouvoir rentrer dans le bâtiment en dehors des horaires d'ouverture au public (8h30 – 17h30).

1.3 Le sujet de recherche

Dans un premier temps mon sujet de recherche portait sur les diagrammes et les relations algébriques qu'on pouvait envisager pour construire une syntaxe de description de scènes 3D qui soit élégante, avec une phase efficace de compilation mais surtout ayant des propriétés mathématiques intéressantes (comme la *distributivité* par exemple).

Après avoir cherché de la documentation et commencé à réfléchir, il s'est trouvé que dans le cadre 2D il existe déjà une bibliothèque pour le langage de programmation `Haskell` [9] qui répond exactement au problème adressé, et ce de manière élégante. Après avoir lu plusieurs documents et conférences en lignes, il m'est apparu que leur solution était de loin le meilleur compromis possible entre simplicité et expressivité. C'est pourquoi mon rapport ne porte pas sur ce sujet mais sur le suivant qui me fût donné.

Par la suite, mon travail a été directement relié à l'écriture d'un papier soumis par Dan R. Ghica et Achim Jung à la fin de mon stage pour la conférence POPL 17. L'écriture et le cadre théorique étaient déjà presque terminés à mon arrivée, ainsi mon travail a été orienté vers la mise en pratique et l'obtention d'un programme faisant preuve de concept, bien qu'une grande partie du temps fût passé dans la compréhension des aspects théoriques.

2 Étude des circuits via la théorie des catégories

Il existe une relation étroite entre la théorie des catégories [3] et les diagrammes respectant certaines propriétés comme le montre l'article de P. Selinger [6].

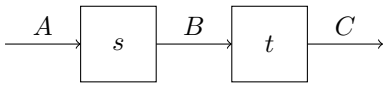
2.1 Première approche

Soit $\mathcal{C}$ une catégorie libre, on peut associer à chaque morphisme de $\mathcal{C}$ une représentation en terme de diagramme. En particulier comme la catégorie est libre, étant donné un ensemble de morphismes fixés, il n'y a pas de relation particulière engendrée par la composition de ces morphismes, ce qui permet de construire des diagrammes de plus en plus grands sans avoir à déterminer s'il existe une « réduction » à effectuer.

Cette construction permet donc de parler de diagrammes séquentiels, qui ne sont en réalité qu'une suite de composition de morphismes, puisque c'est la seule construction possible. Les diagrammes générés sont semblables à des chaines de blocs déposés de manière séquentielle comme on peut le constater sur la figure 1.

Afin de construire des diagrammes plus intéressants, on peut ajouter des constructions (libres) à la catégorie, et pour modéliser des relations particulières on peut ajouter des axiomes d'équivalence entre deux expressions (quotienter par des relations d'équivalence entre morphismes).

$$t \circ s : A \to C$$



$$s : A \to B \quad t : B \to C$$

FIGURE 1 – Le diagramme de la composition de morphismes

2.2 Utilisation d'un modèle formel pour une sémantique opérationnelle

L'idée principale de l'article est de fournir un modèle de calcul sur les circuits à partir de leur description algébrique qui soit opérationnel. Pour cela, une première étape est de construire une catégorie qui permet de modéliser des circuits, puis d'utiliser les axiomes de cette catégorie pour réduire étape par étape les expressions.

Cette deuxième phase est très coûteuse si on considère uniquement un système de ré-écriture syntaxique, car de nombreux axiomes de distributivité et de *fonctorialité* forcent à faire des choix très nombreux durant la ré-écriture et dont l'optimalité n'est pas garantie. C'est pourquoi l'expression algébrique est traduite en terme de *graphe dirigé acyclique* qui correspond à la représentation en terme de diagrammes. Cette transformation permet d'utiliser les techniques de « graph rewriting » [7] qui donnent des performances et une simplicité d'utilisation bien plus importantes.

3 Sujet de recherche

3.1 Cadre de l'étude

L'étude se place spécifiquement dans le cadre de circuits digitaux[1] et les valeurs possibles dans un fil sont $\bot$ (pas de signal), H (high), L (low), $\top$ (court-circuit).

Afin de modéliser ces circuits fidèlement, une catégorie symétrique monoïdale tracée munie d'un monoïde de Frobenius [5] est utilisée. Précisément, cela veut dire qu'il existe un certain nombre d'opérateurs (voir figure 2) et d'axiomes (qui ne sont pas décrits car prenant beaucoup de place).

Le monoïde de Frobenius correspond simplement à l'ajout de deux morphismes particuliers *fork* et *join*, le premier permettant de dupliquer une valeur en « fourchant » un fil, l'autre permettant de fusionner deux valeurs en « joignant » deux fils. Il faut de plus ajouter un circuit w qui correspond à un circuit qui ignore son entrée, symétrique de la valeur $\bot$ qui construit une valeur « vide ».

L'étude se fait dans une catégorie où les objets sont des entiers naturels ce qui permet de caractériser un morphisme par son nombre d'entrées et son nombre de sorties, et donc de modéliser des blocs dans un circuit digital. Afin de clarifier les règles on peut donner un système de type qui permet de déterminer uniquement, à partir de l'expression d'un morphisme son nombre d'entrée et de sorties comme c'est fait figure 3.

Par la suite la notion de *délai* est ajoutée, qui permet d'exprimer un décalage discret dans le temps, et ainsi étudier des réponses d'un circuit dans le temps. Pour des raisons de simplicité, les axiomes régissant les *délais* ne sont pas explicités mais peuvent être trouvées dans un précédent article de Dan R. Ghica [2].

3.2 Problèmes posés

3.2.1 Syntaxe

Le problème de cette étude est que l'écriture de circuits sous forme algébrique n'est pas canonique (à cause de la distributivité) et qu'elle est très sensible aux petits changements dans le circuit.

Un exemple frappant est celui du circuit figure 4 qui s'écrit simplement $(F \otimes G) \cdot H$ mais qui en ajoutant simplement un unique fil devient le circuit figure 5 dont l'expression est alors :

$$\mathrm{Tr}_1 \left[(s_{1,1} \otimes \mathrm{Id}_1) \cdot (\mathrm{Id}_1 \otimes j) \cdot (F \otimes G) \cdot (\mathrm{Id}_1 \otimes f \otimes \mathrm{Id}_1) \cdot (s_{1,1} \otimes \mathrm{Id}_2) \right] \cdot H$$

En revanche dans une description plus classique [8] des circuits en terme de graphes comme en `VHDL` ou `Verilog`, les deux descriptions, bien que verbeuses, ne diffèrent que d'une seule

1. Qui ont une notion précise d'entrée et de sortie, contrairement aux circuits dits analogiques

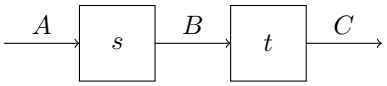
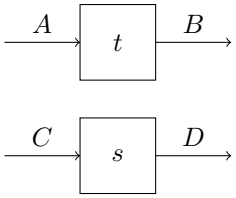
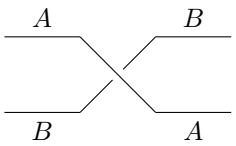
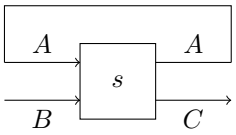
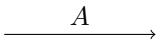
Construction	Représentation	Signification
$t \circ s$	$t \circ s : A \to C$  $s : A \to B \quad t : B \to C$	Composition séquentielle
$t \otimes s$	 	Composition parallèle
$s_{A,B}$	 	Symétrie
$\mathrm{Tr}_A(s)$	 	Boucle de retour
Id_A	 	Identité de A vers A (fils simples)

FIGURE 2 – Constructions dans une TSMC

$$\frac{}{\mathrm{n} : n \to n}$$

$$\frac{}{\mathrm{f} : 1 \to 2}$$

$$\frac{}{\mathrm{j} : 2 \to 1}$$

$$\frac{}{\mathrm{w} : 1 \to 0}$$

$$\frac{}{\bot : 0 \to 1}$$

$$\frac{\phi_1 : n_1 \to m_1 \qquad \phi_2 : n_2 \to m_2}{\phi_1 \otimes \phi_2 : (n_1 + n_2) \to (m_1 + m_2)}$$

$$\frac{\phi_1 : n \to p \qquad \phi_2 : p \to m}{\phi_1 \,.\, \phi_2 : n \to m}$$

$$\frac{}{x_{i,j} : (i + j) \to (i + j)}$$

$$\frac{\phi : n + 1 \to m + 1}{\mathrm{Tr}\ \phi : n \to m}$$

FIGURE 3 – Le système de type pour la syntaxe algébrique

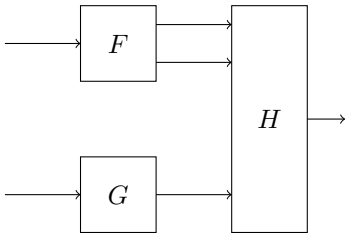


FIGURE 4 – Circuit exprimé aisément avec la syntaxe algébrique

ligne. La syntaxe actuelle ne permet donc pas la modification incrémentale d'un circuit, et demande beaucoup d'effort pour être lue.

C'est pourquoi j'ai été chargé de construire une nouvelle syntaxe qui serait intermédiaire entre la description relationnelle (graphe) et purement algébrique (théorie des catégories) et dans l'idéal permettrait de faire un lien entre les deux.

3.2.2 Architecture logicielle, programme

Un logiciel écrit en `Ocaml` était déjà écrit, faisant environ 1500 lignes et s'occupant de la réduction de graphes et de l'export sous format pdf. Sur ce programme, les demandes ont été les suivantes :

1. Jusqu'à présent, les circuits étaient construits explicitement dans le code source en utilisant une EDSL. L'objectif était alors de construire un code qui lit un fichier externe où le circuit est décrit dans la nouvelle syntaxe, ce qui permet d'utiliser plus simplement le programme (et ne pas avoir à écrire les circuits au milieu du code).
2. Après avoir construit cette interface entre le code déjà présent et un fichier externe, il a été

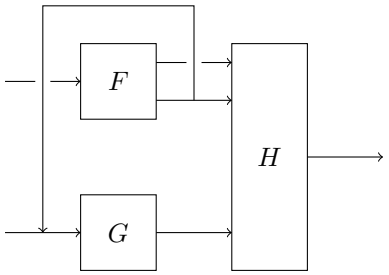


FIGURE 5 – Circuit exprimé laborieusement avec la syntaxe algébrique

nécessaire d'ajouter des règles pour traiter les *délais* dans les circuits.

3. Par la suite certains problèmes de correction ont été réglés, pour au final amener à une réécriture complète de la partie déjà présente afin de corriger les problèmes de performances et la lisibilité du code.

4 Résultats obtenus

4.1 Introduction d'un nouvel opérateur

La nouvelle syntaxe se fait en deux temps :

1. Ajouter la notion de variable-circuit
2. Donner une syntaxe pour lier des variables [2]

Cette construction vient de l'idée qu'un circuit est en quelque sorte une fonction (un morphisme) et donc qu'on peut passer d'une approche *point-free* [3] à une approche *point-wise* où les arguments sont rendus explicites.

La construction la plus naturelle de ce point de vue est celle du lambda-calcul, qui introduit un opérateur $\lambda x.F$, qui permet d'utiliser la *variable* x dans *l'expression* F.

4.1.1 Notion de variable

C'est donc naturellement que l'on ajoute la notion de *variable* à la syntaxe des circuits. Il faut remarquer que cela éloigne considérablement la syntaxe de son fondement provenant de la théorie des catégories, et qu'il faudra désormais faire attention aux variables libres dans les expressions.

Pour des raisons de symétrie dans les circuits dirigés (on peut construire un circuit en prenant exactement le symétrique d'un autre circuit) il y aura deux types de variables : les variables productrices, qui permettent de donner une valeur, et les variables consommatrices, qui vont récupérer une valeur.

Il faut bien faire attention à parler de valeurs, contrairement au lambda-calcul où valeurs et fonctions coexistent dans un même espace et peuvent être passées comme variables la théorie dans laquelle évolue les circuits fait une différence entre morphismes et objets [4]. Ainsi de manière intuitive une variable va pouvoir être remplacée par une valeur, c'est à dire un *fil* dans la représentation sous forme de diagrammes (et non pas par un circuit).

Une variable productrice peut être utilisée comme une source de valeurs (un circuit qui produit une valeur) et une variable consommatrice utilisée comme un puits de valeurs (un circuit qui consomme une valeur). L'idée étant qu'une variable sera remplacée par un *fil* tiré vers une autre

2. Au sens de la distinction variable *liée*/variable *libre*
3. Qui n'explicite pas les arguments
4. Ce n'est pas une catégorie monoïdale close [4]

partie du circuit par la suite. Cela permet d'intégrer les variables dans les expressions classiques (composition séquentielle, parallèle, trace ...).

4.1.2 Progression jusqu'à *link*

L'idée naturelle est alors d'introduire des opérateurs qui permettent de lier les variables, ces deux opérateurs seront symétriques, l'un étant nommé IN l'autre OUT.

L'opérateur IN est l'équivalent de l'opérateur λ, il permet de lier une variable productrice. Quand on écrit `IN x. F`, on ajoute un nouveau fil d'entrée qui se sépare pour aller rejoindre chaque occurrence de la variable x dans l'expression F.

De manière symétrique, quand on écrit `F . OUT x`, un nouveau fil de sortie est ajouté, et toute occurrence de x dans F est reliée à ce nouveau fil.

Si cette image ne fonctionne pas, on peut voir à l'inverse, que ces deux opérateurs permettent de *nommer* le premier fil entrant/sortant d'un circuit et le ré-utiliser plusieurs fois dans l'expression.

Malheureusement, ces deux opérateurs ne diminuent presque pas le labeur qui consiste à lier deux parties éloignées d'une expression. En effet, après avoir nommé les deux parties, il reste encore à exprimer qu'elles sont reliées via la syntaxe classique. C'est pourquoi un dernier opérateur est nécessaire : *link*. Cet opérateur va tout simplement relier une variable productrice et une variable consommatrice.

Si chaque variable est utilisée une unique fois, c'est simplement tirer un câble de la consommatrice vers la productrice, ainsi les valeurs produites sont exactement celles qui étaient consommées avant. Dans le cas contraire, il faut dans un premier temps fusionner les valeurs consommées en une seule (avec *join*) puis distribuer cette valeur aux différents points de production (avec *fork*).

La sémantique précise de cet opérateur est donnée plus tard dans le rapport, et pour le moment cette intuition est suffisante pour comprendre ce qui suit.

4.1.3 De l'algébrique au relationnel

Comme la nouvelle syntaxe est simplement une surcouche de la syntaxe précédente, on peut clairement décider de ne pas l'utiliser et rester dans le style *algébrique*. Toutefois, on peut aussi décider de l'utiliser tout le temps, et imiter le style *relationnel* (qui déclare le circuit comme un graphe par sa liste d'arêtes et de nœuds). En effet, il suffit pour cela de nommer chaque entrée et sortie de chaque bloc B_i en écrivant :

$$(x_1 \otimes \cdots \otimes x_n) \cdot B_i \cdot (y_1 \otimes \dots y_m)$$

Où les x_i sont des variables productrices et les y_i des variables consommatrices.

On peut ensuite mettre en parallèle toutes ces expressions au sein d'une expression F. Enfin, pour chaque lien entre deux blocs, ajouter un lien avec *link* entre les deux variables correspondantes.

Cette construction suit précisément le style des déclarations dans des langages comme `VHDL` ou `Verilog`, et l'objectif d'avoir une syntaxe polyvalente est donc clairement atteint.

4.2 Expression des opérateurs à partir de link

Un premier constat est qu'il n'est pas nécessaire d'avoir `IN`, `OUT` et `Link` car ce dernier permet d'exprimer de manière très simple les deux précédents. Ainsi, on peut uniquement se concentrer sur l'opérateur *link*, et c'est ce qui est fait par la suite.

Un autre constat intéressant est le suivant : si on construit l'opérateur *link* comme un sucre syntaxique pour la syntaxe algébrique issue de la théorie des catégories, alors un certain nombre de constructions peuvent s'exprimer directement à partir de cet opérateur (figure 6).

Nom	Opérateur	Expression avec link
Symétrie	$s_{1,1}$	`link x:y z:t for` `(x ⊗ z) ⊗ (t ⊗ y)`
Trace	$\mathrm{Tr}_1(F)$	`link a:b for` `(:b ⊗ Id) · F · (a: ⊗ Id)`
Fork	f	`link a:b for` `a: ⊗ :b ⊗ :b`
Join	j	`link a:b for` `a: ⊗ a: ⊗ :b`

FIGURE 6 – Opérateurs exprimables à partir de *link*

C'est pourquoi une définition minimale du langage peut être l'opérateur *link* ainsi que la composition séquentielle, parallèle et les circuits constants ($\bot$, w ...).

Il est possible que l'opérateur *link* puisse engendrer les circuits $\bot$ et w, permettant ainsi d'englober dans un seul opérateur l'intégralité du monoïde de Frobenius associé au quadruplet $(f, j, \bot, w)$ (voir [5]). L'idée principale derrière cette intuition est que les axiomes semblent garantir que tout circuit sans entrée et avec n sorties sont équivalents, or quand n vaut 1 cela construit précisément un circuit équivalent à $\bot$.

4.2.1 Axiomatique alternative

Comme un certain nombre de constructions peuvent être directement exprimées à partir de l'opérateur *link*, une question naturelle est la suivante : quels sont les axiomes nécessaires et suffisants qu'il faut imposer à cet opérateur pour que les expressions des autres constructions vérifient leurs axiomes respectifs ?

Par exemple l'expression de la symétrie $s_{1,1}$ doit vérifier la propriété $s_{1,1}^{-1} = s_{1,1}$ (la symétrie est son propre inverse).

Après avoir étudié les axiomes de l'opérateur trace et ceux de la symétrie [6] un ensemble d'axiomes assez naturels pour l'opérateur *link*.

Alpha équivalence : Le nom des variables liées n'importe pas dans une expression, la définition étant exactement la même que l'alpha-équivalence en lambda-calcul.

Tightening : Provient de l'axiome éponyme pour la trace, et signifie qu'il est possible d'étendre l'application de l'opérateur *horizontalement* (pour la composition séquentielle) sous certaines conditions.

```
G · (link a:b for F) · H
=
link a:b for G · F · H
```

Quand a et b ne sont pas libres dans G et dans H. L'opération graphique est visible figure 7.

Sliding : Une fois de plus provenant des axiomes de l'opérateur trace, il permet en un certain sens de parler de transitivité pour les boucles de retour.

```
link a:b for
    (Id ⊗ (:b · G) ⊗ Id) · F · (Id ⊗ a: ⊗ Id)
=
link a:b for
    (Id ⊗ :b ⊗ Id) · F · (Id ⊗ (G · a:) ⊗ Id)
```

Quand a et b ne sont pas libres dans F. Il est aussi demandé que a et b ne soient pas libres dans G, mais cette condition ne semble pas être nécessaire.

Si l'on cherche à exprimer des circuits de contrôle de flots (comme c'est le cas ici) il faut ajouter la possibilité d'avoir de multiples occurrences de b dans l'équation (toujours suivie de G).

Si l'on cherche à exprimer des circuits de flots de données (ce qui n'est pas le cas ici) il faut autoriser de multiples occurrences de a dans l'équation.

Vanishing : Permet d'exprimer le fait que des liens indépendants peuvent être faits en une seule fois, et que les liens triviaux peuvent être omis.

```
link a:b for
link c:d for
    F
=
link a:b c:d for
    F
```

L'omission de liens triviaux s'écrit simplement `link a:b F = F` quand a et b ne sont pas des variables libres pour F.

Strength : C'est l'équivalent de l'axiome de *tightening* mais pour la composition parallèle.

```
G ⊗ (link a:b for F) ⊗ H
=
link a:b for G ⊗ F ⊗ H
```

Quand a et b ne sont pas libres dans G ni dans H.

On peut remarquer que contrairement à l'axiome éponyme pour l'opérateur trace, le circuit F n'a pas besoin d'être placé au dessus, car le lien ne prend pas en compte la position des variables dans le circuit.

Substitutivity (1) : Si deux variables productrices sont liées au même ensemble de variables consommatrices alors on peut confondre les deux variables dans le circuit.

Substitutivity (2) : Si deux variables consommatrices sont liées au même ensemble de variables productrices alors on peut confondre les deux variables dans le circuit.

Identity : L'identité peut être construite à partir de *link* :

```
(link a:b for
    a: · :b) · F
=
F
```

On peut remarquer que la commutativité de l'identité peut se démontrer à partir de l'axiome *sliding*.

Sliding pour les variables (transitivité) : Dans sa forme la plus simple on demande l'égalité :

```
link a:b for
    (:c · a:) ⊗ :b
=
:c
```

Mais celle-ci est triviale car :

```
link a:b for
    (:c · a:) ⊗ :b
=
link a:b for
    (:c ⊗ 0) · (a: ⊗ :b)
=
link a:b for
    :c · (a: ⊗ :b)
=
:c · (link a:b a ⊗ :b)
=
:c · Id
=
:c
```

Mais une règle plus générale est nécessaire pour pouvoir faire des preuves :

```
link a:b for
    F ⊗ (:c · a:) ⊗ G
=
F[b/c] ⊗ G[b/c]
```

Quand a est libre dans F ainsi que dans G, et où $F[b/c]$ signifie que l'on remplace b par c dans l'expression de F, en prenant soin de ne remplacer que les occurences de b qui sont libre dans F (substitution comme dans le lambda-calcul).

4.2.2 Propriétés de l'axiomatique alternative

Cette axiomatique alternative est *correcte* pour la représentation en terme de diagrammes de circuits digitaux, elle permet de plus assez facilement de *déduire* les propriétés de l'expression de la trace, de la symétrie, de la jointure et de la fourche. Il est conjecturé qu'elle est aussi *complète* [5], mais aucune ébauche de preuve n'est commencée.

4.3 Description de la syntaxe

La description formelle de la syntaxe dans son état final est visible figure 8. Elle contient spécifiquement deux types de transistors MOFSET, dont la table de vérité est écrite dans le programme pour effectuer des réduction sur des circuits contenant ces transistors, ainsi qu'un circuit MUX (multiplexer) bien qu'on puisse l'exprimer à partir des transistors.

4.3.1 Analyse syntaxique associée

La syntaxe peut-être mise sous forme d'une grammaire LL(5) (voir figure 9) ce qui rend l'écriture d'un analyseur syntaxique descendant très facile.

On ajoute deux constructions pour simplifier l'écriture : SEQ et PAR. L'idée est tout simplement que dans un bloc SEQ les retours à la lignes sont transformés en composition séquentielle et dans un bloc PAR les retours à la ligne sont transformés en composition parallèle [6]. Cette opération de transformation se fait en temps linéaire avec une pile.

L'analyseur syntaxique est donc découpé en deux phases : une première de ré-écriture qui interprète les blocs SEQ/PAR, ignore les espaces et tabulations multiples, suivie d'une deuxième qui va effectuer l'analyse du texte ainsi transformé.

5. Si deux expressions donnent des circuits identiques alors on peut prouver l'équivalence des deux expressions avec les axiomes

6. En mettant implicitement des parenthèses autour des lignes pour conserver le sens des circuits sur chaque ligne

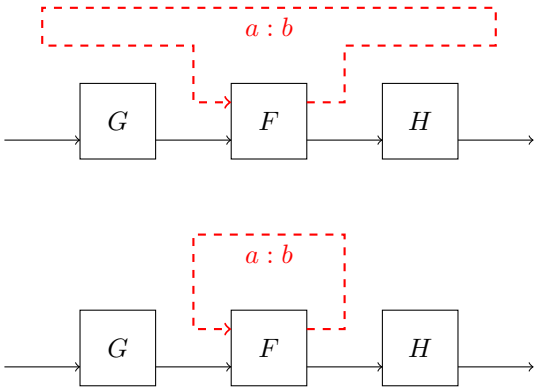


FIGURE 7 – Description visuelle de l'équivalence décrite par l'axiome *tightening*

⟨*toplevel*⟩ ::= '`let`' ⟨*circuit-name*⟩ '=' ⟨*expr*⟩ '`and`' ⟨*toplevel*⟩
| ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*expr*⟩ ⊗ ⟨*expr*⟩
| ⟨*expr*⟩ '`.`' ⟨*expr*⟩
| '`n`' ∈ ℕ
| '`id`'
| '`FORK`'
| '`JOIN`'
| '`BOT`'
| '`TOP`'
| '`HIGH`'
| '`LOW`'
| '`NMOS`'
| '`PMOS`'
| '`MUX`'
| '`link`' ⟨*bindings*⟩ '`for`' ⟨*expr*⟩
| '`:`'⟨*variable*⟩
| ⟨*variable*⟩'`:`'
| ⟨*circuit-name*⟩

⟨*bindings*⟩ ::= ⟨*variable*⟩ '`:`' ⟨*variable*⟩
| ⟨*variable*⟩ '`:`' ⟨*variable*⟩ '`space`' ⟨*bindings*⟩

⟨*variable*⟩ ::= '`[a-z][a-zA-Z0-9]*`'

⟨*circuit-name*⟩ ::= '`[A-Z][a-zA-Z0-9]*`'
| '`[A-Z][a-zA-Z0-9]*`'

FIGURE 8 – Description de la syntaxe en BNF

```
P  -> SP'
P' -> *SP' | eps
S  -> ES'
S' -> .ES' | eps
E  -> link VARS P
    | (P)
    | INT
    | CONST
    | id
    | :VAR
    | VAR:
    | CIRC
INT  -> [0-9]+
VAR  -> [a-z][a-zA-Z0-9]*
VARS -> VAR:VAR for | VAR:VAR VARS
CIRC -> [A-Z][a-zA-Z0-9]*
CONST -> "MUX" | "HIGH"
       | "LOW" | "NMOS" | "PMOS"
       | "BOT" | "TOP"  | "FORK" | "JOIN"
```

FIGURE 9 – Grammaire algébrique associée à la syntaxe des expressions

4.4 Sémantique en terme de diagrammes

La sémantique peut être donnée en terme de graphes usuels, mais pour des raisons de simplicité, elle est donnée par une construction légèrement plus complexe, qui est très proche de la représentation intermédiaire utilisée dans le programme afin appliquer les règles de ré-écriture.

4.4.1 Structure sémantique

La sémantique d'une expression est donnée par la construction d'un graphe dirigé acyclique *tracé* avec une forme spécifique pour les délais, ainsi que des entrées et des sorties spécifiées. Ce qui correspond à la donnée de certains composants :

Entrées : une *liste*[7] de noeuds d'entrée

Sorties : une *liste* de noeuds de sortie

Trace : une *liste* de noeuds possédant une unique arête entrante et une unique arête sortante

Délais : une *liste* de noeuds étant des délais

Noeuds : le reste des noeuds dans un ensemble non ordonné

Étiquettes : une fonction partielle de l'ensemble de tous les noeuds du graphe vers des étiquettes donnant la fonction du noeud (délai, valeur, NMOS, MUX, FORK ...)

Arêtes : les arêtes dirigées entre les noeuds du graphe

Il faut préciser que les différentes listes de nœuds sont disjointes, afin de simplifier l'étude des graphes par la suite. Un schéma expliquant quelle est la forme du graphe est dessiné figure 10. Le fait que toute expression algébrique décrivant un circuit puisse être mis sous cette forme (avec une trace globale) est un théorème démontré dans un premier article de mon superviseur [2].

7. l'ordre est important

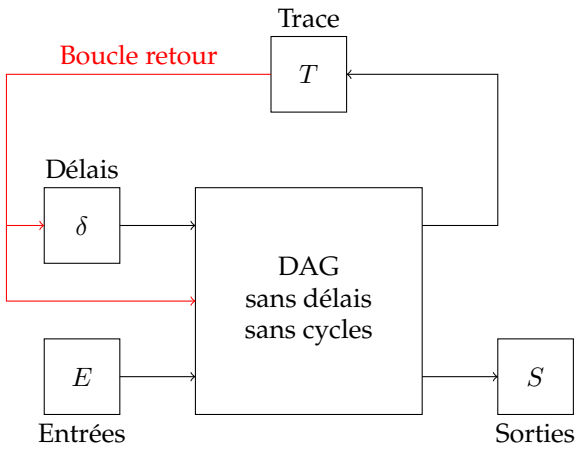


FIGURE 10 – Forme tracée avec délais explicites d'un graphe

4.4.2 Sémantique de la syntaxe déjà en place

La sémanitque de la syntaxe déjà en place est relativement simple, puisque chaque opération est déjà décrite en terme de diagramme dans la figure 2. La seule chose à faire est de bien garder en mémoire quels sont les noeuds tracés, et ceux avec des délais, mais ce n'est pas très difficile graphiquement.

4.4.3 Sémantique du nouvel opérateur

La construction est relativement laborieuse, et ressemble en réalité plus à un algorithme qu'à une description. Afin de pouvoir donner la sémantique, il faut modifier le domaine sur lequel on travaille et ajouter deux étiquettes particulières : *ivar* et *ovar* qui permettent de détecter des nœuds correspondant à des variables libres dans l'expression.

Ainsi, on ne pourra transformer en un graphe classique qu'une expression dans la nouvelle syntaxe qui serait *close* [8].

Sémantique d'une variable productrice :

$$\begin{cases} E & = \emptyset \\ S & = \{2\} \\ T & = \emptyset \\ \delta & = \emptyset \\ N & = \{1\} \\ \text{labels} & = (1, \mathit{ivar}) \\ \text{edges} & = \{(1,2)\} \end{cases}$$

8. Qui ne possède pas de variable libre

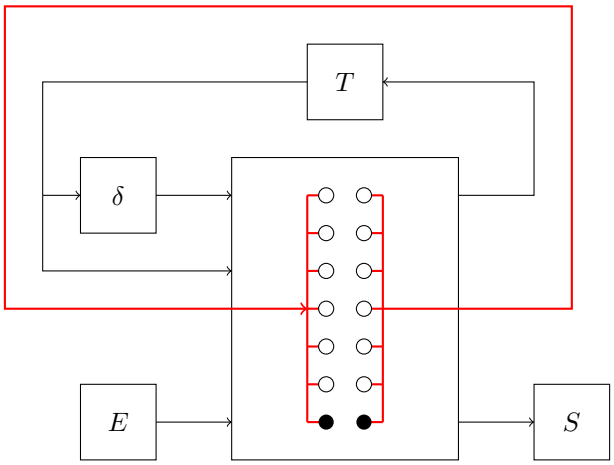


FIGURE 11 – Représentation graphique du lien de deux variables

Sémantique d'une variable consommatrice :

$$\begin{cases} E & = \{2\} \\ S & = \emptyset \\ T & = \emptyset \\ \delta & = \emptyset \\ N & = \{1\} \\ \text{labels} & = (1, \mathit{ovar}) \\ \text{edges} & = \{(2,1)\} \end{cases}$$

Sémantique d'un lien : Soit $(E, S, T, \delta, N, \text{edges}, \text{labels})$ la sémantique de M dans l'expression `link` a_1`:`b_1 `...` a_n`:`b_m `for` M.

1. Pour chaque variable productrice liée a_i, construire un nouveau nœud c_i.
2. Pour chaque variable consommatrice liée b_i, construire un noveau nœeud d_i
3. Pour chaque noeud dans M ayant l'étiquette b_i, supprimer l'étiquette et ajouter un fil sortant du nœud. Joindre tous les fils en un seul, puis joindre $\perp$ à ce fil [9], et enfin connecter ce dernier à d_i [10].
4. Faire de même symétriquement pour les variables consommatrices
5. Pour chaque paire liée a_i :b_i, relier le c_i et le d_i correspondant [11].

Une représentation graphique dans le cas où seulement deux variables sont liées est visible figure 11.

Sémanitque des autres constructions : On peut reprendre exactement la sémantique précédente pour les autres opérateurs son domaine de définition est strictement inclus dans celui de la nouvelle sémantique.

9. Ajouter $\perp$ ne change rien au résultat, mais permet de traiter le cas où b_i n'apparaît pas dans le circuit
10. Les axiomes du monoïde de Frobenius garantissent que l'ordre dans lequel on joint n'influence pas le résultat
11. Si une variable est liée plusieurs fois, il faut utiliser une fois de plus *fork* et *join* sur les câbles

4.5 Système de type

Afin de vérifier et rendre explicite les opérations les plus courantes sur les circuits (composition séquentielle et parallèle) un système de type très simple est mis en place. Le système mis en place est plus complexe que nécessaire dans l'état actuel des choses mais il est conçu en sachant que l'objectif final est d'ajouter des variables représentant un groupe de fils dont la taille n'a pas à être explicitée [12]. Cet objectif n'est pas encore atteint faute de temps, mais tout le système autour est en place, et on peut dors et déjà utiliser le polymorphisme pour le circuit identité.

La raison principale qui justifie l'absence de variable avec un type polymorphe est que la syntaxe serait alors extrêmement ambiguë et donc qu'il faut ajouter à celle-ci des annotations de type, ce qui n'a pas été fait faute de temps.

Il est déjà possible d'avoir des expressions non uniquement typables avec les identités comme dans l'exemple id $\otimes$ id qui a une infinité de types, mais la possibilité de spécifier le nombre de fils dans l'identité via un nombre règle ce problème.

4.5.1 Description des séquents

Le système de type se présente sous une forme simple : un type est associé à un séquent $\Gamma|\Delta \vdash F : x \to y$ qui signifie exactement

> Le circuit F a le type $x \to y$, c'est-à-dire possède x entrées et y sorties, dans le contexte $\Gamma|\Delta$, où Γ contient les variables libres qui produisent une sortie, et Δ les variables libres qui consomment une entrée.

Les règles décrites figure 12 permettent de construire des preuves qu'une expression possède un certain type. On vérifie aisément que le système de type est *cohérent* par rapport à la sémantique [13] : le type correspond bien au nombre d'entrées et de sorties d'un circuit, et deux expressions différentes d'un même circuit ont bien le même type.

Une conséquence immédiate est que toute expression valide possède un type puisqu'elle représente un circuit, et donc un nombre d'entrées et de sorties. On peut aussi montrer qu'une expression ne possédant pas de type ne représente pas un circuit, et qu'une expression possédant plusieurs types représente plusieurs circuits différents.

4.5.2 Algorithme de résolution

Les contraintes étant seulement linéaires, on peut résoudre le système de type (trouver le type de chaque sous expression) avec un simple algorithme :

1. Construire la matrice associée aux équations de types
2. Utiliser un pivot de Gauss pour simplifier le système et détecter une absence de solution.
3. S'il y a une solution, on peut déterminer si elle est unique simplement grâce à la forme de la matrice après réduction.
4. S'il y a une unique solution, vérifier si elle est positive et entière.
5. Si c'est le cas le vecteur solution détermine le type de chaque sous-expression

Un message spécifique d'erreur peut être associé à chaque étape et on peut même parfois déterminer quelle partie de l'expression cause l'erreur de type, c'est-à-dire possède deux types incompatibles ou bien une infinité de types, comme on peut le constater sur la figure 13.

12. Ou au moins pas de manière systématique
13. Au sens de la sémantique étendue qui comprend les variables

4.6 Programme

L'architecture du programme est séquentielle comme suit :

1. Lire le fichier spécifiant le circuit
2. Utiliser `lexer.ml` pour mettre le texte dans une forme plus simple à analyser ainsi que supprimer certains éléments de syntaxe (espaces, tabulations ...)
3. Utiliser `parser.ml` pour construire un arbre de syntaxe abstrait comme défini dans `ast.ml`
4. Calculer les types des sous expressions en utilisant `typesystem.ml` qui lui-même fait appel à `solver.ml` pour effectuer ce travail.
5. Transformer récursivement l'expression typée en un graphe comme défini dans `dag.ml` en utilisant les fonctions de `compiler.ml`
6. Une fois cette structure construite, la convertir dans un format plus optimisé et simple à ré-écrire défini dans `ptg.ml`
7. Enfin, appliquer les règles de ré-écriture sur cet objet en utilisant le fichier `rewriting.ml`
8. Une fois les règles appliquées (ou bien entre chaque application) exporter un document sous forme *graphviz* via le module `dot.ml` et le compiler vers un graphe dessiné dans un fichier pdf.

Au total, le programme fait 3616 lignes [14] et n'utilise que la librairie standard, ce qui reste très raisonnable, surtout grâce à la répartition en modules qui facilite grandement la compréhension de petites entités (voire figure 14).

Le parser est basé sur une version très simpliste et non efficace de combniateurs d'analyseurs syntaxiques [1] pour des raisons de rapidité d'écriture, parce que la grammaire est très simple à lire algorithmiquement, mais aussi car la construction initiale du circuit représente un temps négligeable en comparaison du traitement ultérieur par le système de ré-écriture.

14. Le 10/08/2016

$$\frac{}{\Gamma \mid \Delta \vdash \mathsf{n} : n \to n}$$

$$\frac{}{\Gamma \mid \Delta \vdash \mathsf{id} : k \to k}$$

$$\frac{}{\Gamma \mid \Delta \vdash \mathsf{w} : 1 \to 0}$$

$$\frac{}{\Gamma \mid \Delta \vdash \bot : 0 \to 1}$$

$$\frac{\Gamma \mid \Delta \vdash \phi_1 : n_1 \to m_1 \qquad \Gamma \mid \Delta \vdash \phi_2 : n_2 \to m_2}{\Gamma \mid \Delta \vdash \phi_1 \otimes \phi_2 : (n_1 + n_2) \to (m_1 + m_2)}$$

$$\frac{\Gamma \mid \Delta \vdash \phi_1 : n \to p \qquad \Gamma \mid \Delta \vdash \phi_2 : p \to m}{\Gamma \mid \Delta \vdash \phi_1 \,.\, \phi_2 : n \to m}$$

$$\frac{\Gamma, b_1 \dots b_n \mid \Delta, a_1 \dots a_n \vdash \phi : n \to m}{\Gamma \mid \Delta \vdash \mathsf{link}\ a_1 : b_1 \ \dots\ a_n : b_n\ \phi : n \to m}$$

$$\frac{}{\Gamma, a \mid \Delta \vdash a : 0 \to 1}$$

$$\frac{}{\Gamma \mid \Delta, a \vdash a : 1 \to 0}$$

FIGURE 12 – Le système de type avec des variables monomorphes

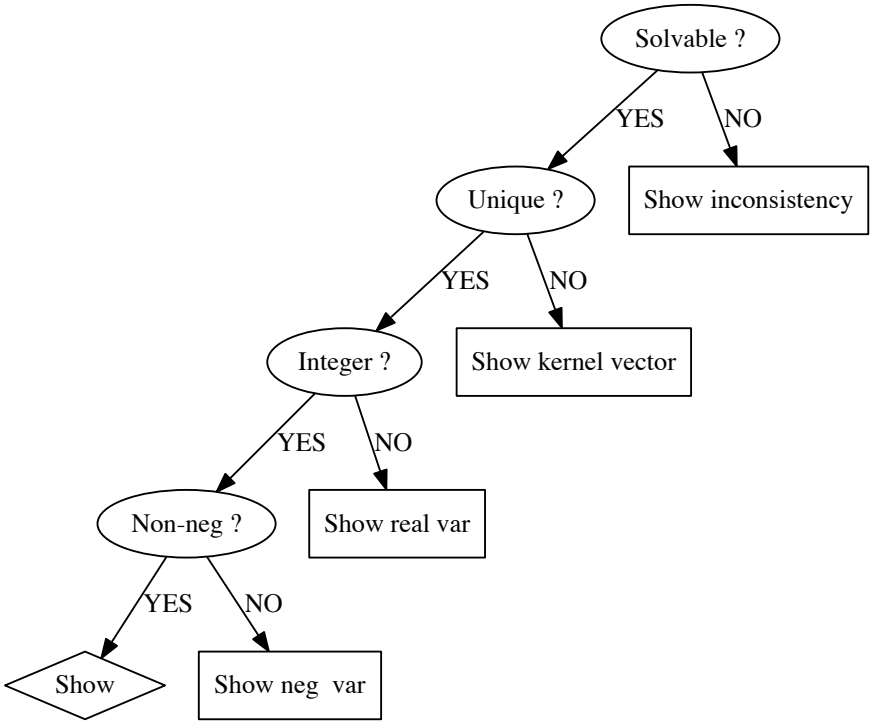


FIGURE 13 – Algorithme de résolution des types

Nom	**Taille**	**Utilité**
utils.ml	216 lignes	Fonctions utilitaires inclassables
dot.ml	126 lignes	EDSL pour construire des graphes au format GraphViz
ast.ml	133 lignes	Définition et fonction de traitement de l'arbre de syntaxe abstrait
solver.ml	323 lignes	Résolution des systèmes linéaires provenant du typage
typesystem.ml	301 lignes	Construction des types d'une expression
lexer.ml	106 lignes	Lexer de la syntaxe
parser.ml	284 lignes	Analyseur syntaxique de la syntaxe
dags.ml	342 lignes	Définition d'une structure intermédiaire « DAG »
compiler.ml	141 lignes	Passage d'un AST vers un DAG
ptg.ml	743 lignes	La définition des graphes sur lesquels les réductions oppèrent
rewriting.ml	580 lignes	Le système de ré-écriture de « ptg »
tests.ml	14 lignes	Le lanceur des tests unitaires contenus dans les différents modules
circuits.ml	307 lignes	Le point d'entrée du programme qui met en place le système

FIGURE 14 – Les différents fichiers et leurs caractéristiques

5 Modification de la partie « rewriting »

Les deux dernières semaines du stage ont entièrement été consacrées à la modification de la partie qui met en place le système de ré-écriture de graphes.

Jusqu'alors les différentes parties de mon programme devaient s'interfacer avec ce bloc déjà existant, ce qui explique par exemple la présence de deux structures pour représenter les graphes dans la version finale (en plus des considérations de performance).

Dans un premier temps, le travail était d'analyser le système et détecter où étaient les problèmes de performance ou les erreurs de programmation, puis de proposer des solutions.

5.1 Analyse de la ré-écriture

Il y a deux types de modifications de graphes dans le système :

1. Ré-écriture locale qui fonctionne en deux phases
   (a) Une reconnaissance de motif
   (b) Une modification locale du motif
2. Ré-écriture globale qui nécessite d'avoir déjà enregistré certaines structures particulières dans la représentation du graphe

Les règles globales permettent par exemple de traiter la ré-écriture de l'opérateur *trace* qui sinon serait très complexe. C'est la raison pour laquelle on utilise la représentation particulière figure 10 (celle utilisée pour donner la sémantique).

5.2 Structure de donnée utilisée

Une simple analyse des règles locale de remplacement montre qu'une grande partie de ces règles utilisent à la fois la strucutre de graphe et celle du graphe dual. En effet, dans la phase de reconnaissance de motif, on recherche un type de nœud particulier, et ensuite ses voisins quelque soit le sens des arêtes. Dans le programme déjà en place avant mon arrivée, un temps non négligeable [15] était passé à calculer ces arêtes inverses.

C'est pourquoi la structure contient le graphe et le graphe dual, en maintenant la cohérence des deux représentations. Cela possède un coût en mémoire (double la taille) mais demande beaucoup moins d'opérations en temps de calcul.

Enfin, pour les circuits, l'ordre est important pour les transistors ou le multiplexer. Jusqu'à présent, ce qui était fait était d'avoir des nœuds intermédiaires spéciaux, avec une étiquette entière donnant l'ordre les entrées, car le graphe ne permettait pas de conserver l'ordre.

Une solution élégante qui provient de l'ajout des arêtes dans les deux sens est qu'on peut enregistrer l'ordre d'entrée et de sortie des arêtes. Malheureusement le maintien des invariants (ordre) devient très vite pénible, surtout quand deux nœuds ont plusieurs arêtes entre eux. C'est pourquoi un autre niveau d'indirection est ajouté, qui permet de nommer explicitement les arêtes, possédant alors un identifiant unique au même titre que les nœuds.

La structure de donnée finale est présentée figure 15.

5.3 Résultats sur le programme

La ré-écriture a rendu le code beaucoup plus clair et lisible, bien qu'elle ait aussi augmenté le nombre de lignes de code. Les problèmes de performances semblent être résolus, mais encore beaucoup de parties du code peuvent encore être rendues plus efficaces.

Malheureusement, cette ré-écriture a introduit certaines régressions, principalement sur le traitement des délais (qui sont pour l'instant non-utilisables). Malgré cela, le code devenu plus simple devrait pouvoir permettre leur ré-introduction avec un peu de temps.

15. Environ 80%

```
   type ptg = {
    (* naturally have a notion of order *)
    iports : nid list;
    oports : nid list;

    (*
     * impose a notion of order to simplify
     * ulterior modification
     *)
    traced : nid list; (* 1 input 1 output *)
    delays : nid list; (* 1 input 1 output *)

    (* nodes that are not iport or oport or traced or delays *)
    nodes  : nid list;

    (* the labels mapping *)
    labels  : label mapping;

    (* edges in right order     *)
    edges      : eid list mapping;

    (* edges for the dual graph *)
    co_edges   : eid list mapping;

    (* arrows : the pair of nodes that corresponds
     * to an edge
     *
     * edge -> (node, node)
     *)
    arrows     : (nid * nid) mapping;
};;
```

FIGURE 15 – Nouvelle structure de donnée (extrait du code source)

6 Conclusion

Le sujet du stage a été très intéressant et instructif sur beaucoup de points, tant théoriques que pratiques, et il reste énormément de travail à accomplir. Le projet était très vaste et par conséquent, bien que servant bien le but de preuve de concept, le programme possède beaucoup de limitations, on peut y trouver (de manière non exhaustive) :

— L'absence de bus de valeurs (groupes de fils) pour les variables
— L'analyse syntaxique ne couvre que les expression et non pas la description complète de la grammaire
— Certaines parties du code proviennent encore de l'ancienne version, et posent des problèmes de performances. C'est le cas pour ramasse-miette qui détecte les parties inutilisées d'un circuit.
— Les délais ne sont pas encore traités
— L'interface du programme est trop simpliste

L'étude théorique aussi possède certaines lacunes. Ainsi, il a été constaté que la forme utilisée permet sous certaines hypothèses de calculer en un déroulage de trace la première valeur d'un circuit possédant des délais. Malheureusement, retrouver dans le graphe cette valeur n'est pas encore totalement formalisé, et la réduction n'en prend pas compte. Un sujet de recherche encore actif est aussi le traitement des *waveforms* (suites temporelles de valeurs) qui peuvent être utilisées simplement avec des délais, mais possèdent des propriétés plus intéressantes.

Néanmoins, l'objectif initial qui était d'avoir une syntaxe plus évoluée et un programme fonctionnel qui permet de mettre en pratique les constructions théoriques étudiées. Ainsi, on peut se réferer aux figure 16 et 17 pour l'évolution d'un circuit simple, et à 18 puis 19 pour un circuit un peu plus compliqué. En réalité, les deux circuits sont identiques, seulement l'un utilise le multiplexer, et l'autre le décompose en transistors.

Ce séjour à l'étranger a été très agréable, tant par le stage en lui-même que par la découverte d'un pays étranger, et je recommande vivement aux étudiants de choisir un stage dans un pays étranger, ne serait-ce que pour approfondir sa connaissance de la langue qui y est parlée, mais aussi découvrir une culture différente.

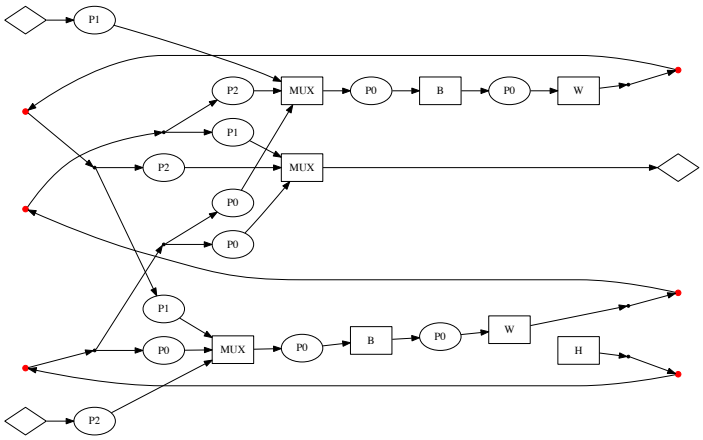


FIGURE 16 – Un circuit très simple avant réduction

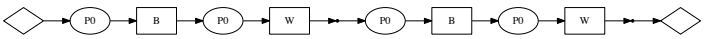


FIGURE 17 – Un circuit très simple après réduction

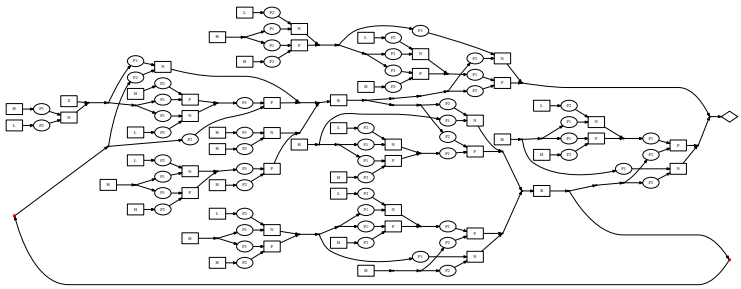


FIGURE 18 – Un circuit assez large avant réduction

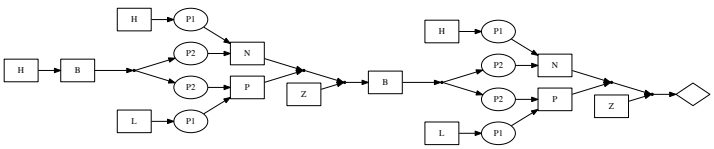


FIGURE 19 – Un circuit assez large après réduction

Références

[1] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 1–12, New York, NY, USA, 2016. ACM.

[2] Achim Jung and Dan R. Ghica. Categorical semantics of digital circuits. In *Proceedings of the 16th conference on Formal Methods in Computer-Aided Design*, FMCAD 2016. FMCAD, 2016.

[3] NCat Lab. Category theory, 2016. [Online ; accesssed 8-August-2016].

[4] NCat Lab. Closed monoidal category, 2016. [Online ; accessed 10-August-2016].

[5] Paul-André Melliès. *Dialogue Categories and Frobenius Monoids*, pages 197–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] P. Selinger. A survey of graphical languages for monoidal categories. *ArXiv e-prints*, August 2009.

[7] Wikipedia. Graph rewriting — wikipedia, the free encyclopedia, 2016. [Online ; accessed 8-August-2016].

[8] Wikipédia. Conception de circuits intégrés — wikipédia, l'encyclopédie libre, 2015. [En ligne ; Page disponible le 10-août-2016].

[9] Ryan Yates and Brent A. Yorgey. Diagrams : A functional edsl for vector graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, FARM 2015, pages 4–5, New York, NY, USA, 2015. ACM.