

Compilateur pour programmes polygraphiques en TOM

Aurelien Monot

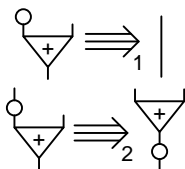
15/02/2007

Table des matières

1	Introduction	2
2	Programmes polygraphiques	3
2.1	Polygraphes	3
2.2	Programmes polygraphiques	5
2.2.1	Constructeurs	5
2.2.2	Structures	6
2.2.3	Fonctions	6
2.2.4	Calculs	7
2.3	Modèles utilisés	8
2.3.1	Retour sur les structures	8
2.3.2	Gom	8
2.3.3	XML	11
3	Reecriture de termes sur les programmes polygraphiques avec TOM	11
3.0.4	Hooks Gom	11
3.1	Normalisation	11
3.2	Gravite	12
3.3	regles	12
3.4	strategies	12
4	Compilateur	12
4.1	architecture d'un programme polygraphique	12
4.1.1	partie commune	12
4.1.2	partie specifique	12
4.2	Fonctionnement du compilateur	12
4.2.1	generation de la partie commune	12
4.2.2	generation de la partie specifique	12
5	Outils développés en parallèle	13
5.1	transformation term vers xml et inverse	13
5.2	bibliotheque de fonctions	13
5.3	generateur de programme.xml	13
5.4	lecture du resultat	13
5.5	generateur de tests	13
6	Conclusion	13
6.1	points-clés	13
6.2	ouverture	13

1 Introduction

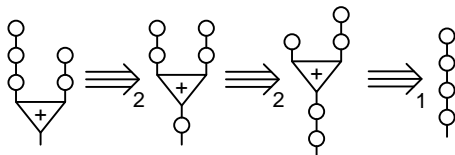
Programmes polygraphiques - L'une des activités au sein de l'équipe Pareo du LORIA consiste à construire des programmes d'un nouveau type, en se basant sur la structure des polygraphes. En voici un exemple :



Ce *programme* permet de calculer l'addition sur les entiers naturels. Par quelle magie ? Tout d'abord, on construit les entiers naturels comme des *cir-*

cuits avec \circ et \bullet . Ainsi, on a : $0 = \circ$, $1 = \bullet$, et $2 = \bullet \circ$. Ensuite, il suffit de brancher nombres de notre choix sur les entrées de notre fonction

addition : \triangle_{+} . Enfin, pour effectuer le calcul, on utilise notre *programme* : les expressions \Rightarrow_i définissent les transformations subies par le circuit lors de l'exécution. Ainsi, si on reconnaît un membre gauche d'une de ces expressions, on le remplace par le membre droit correspondant. On continue de calculer en procédant ainsi tant qu'on le pourra pour obtenir le résultat. Essayons donc de calculer $2 + 1$ avec ce *programme* :



On obtient bien 3

Ces programmes présentent de nombreux intérêts du fait de leur structure de polygraphes. Les polygraphes peuvent être vus comme un système de réécritures pour circuits algébriques. Ainsi, certaines de leur propriétés calculatoire en terme de terminaison, de confluence et de complexité ont été étudiées (références...Albert Burroni, Yves Lafont, Yves Guiraud...). Il serait ainsi possible de développer des nouveaux outils d'analyse de code, dans le but de produire des certificats de qualité pour les programmes. On peut également imaginer la conception de processeur dédiés, qui calculent naturellement de façon polygraphique.

Objectifs du projet - Avant cela, on cherche à construire et exécuter des programmes polygraphiques sur une machine classique. L'objectif du projet donc de participer à cette concrétisation en développant un compilateur

permettant de programmes polygraphiques. Plus précisément, le but est de concevoir une structure de donnée et des algorithmes pour :

- *Importer et exporter une représentation XML des programmes polygraphiques.*
- *Effectuer les opérations de recherche et de remplacement de sous-circuits.*

Tom - Tom est un langage de programmation permettant d'ajouter des fonctionnalités de filtrage à Java (ainsi que d'autres langages de programmation). Ainsi, il constitue un outil idéal pour mettre en oeuvre les systèmes fondés sur des règles de réécriture et donc en particulier les programmes polygraphiques. Le choix de l'usage de TOM pour ce projet est donc assez naturel. L'implantation du projet est donc réalisée en Java étendu avec Tom.

Interpréteur ou compilateur ? - On remarquera, que l'énoncé du sujet laisse la liberté de réaliser un compilateur ou un interpréteur. Ici, le choix de réaliser un compilateur a été fait. Ce choix semblait plus simple à mettre en oeuvre. On distinguera ainsi au cours du rapport le fonctionnement du programme produit par le compilateur et la production de ce programme par le compilateur.

On reviendra tout d'abord plus en détail sur la structure des programmes polygraphiques. Nous verrons ensuite comment mettre en oeuvre un programme polygraphique avec Tom. Enfin, nous étudierons le fonctionnement du compilateur permettant d'obtenir ces programmes en Tom.

2 Programmes polygraphiques

Un des objectifs du projet, et un pré-requis indispensable pour la suite du travail est de modéliser la structure des programmes polygraphiques. Pour cela, nous reviendrons sur la structure des polygraphes, nous étudierons ensuite les spécificités de nos programmes par rapport aux polygraphes puis établirons le modèle utilisé pour la suite du projet.

2.1 Polygraphes

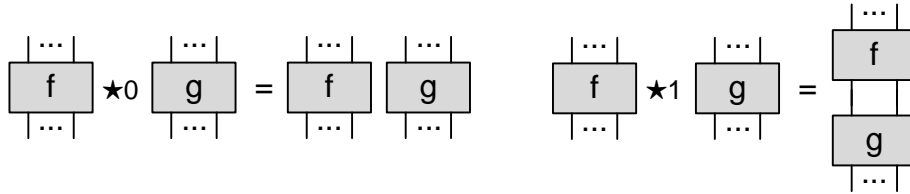
Une façon d'aborder les polygraphes consiste à les considérer comme un système de réécriture sur des circuits algébriques. On peut aussi penser à des circuits électriques pour mieux comprendre. On introduit ici successivement les polygraphes de dimension 1 à 3.

Types - Tout d'abord, il y a les *files*, appelés *1-cellules*. Chaque fil transporte une information d'un type élémentaire. On construit des types plus complexes en associant plusieurs 1-cellules en formant ainsi un *1-chemin* en mettant en parallèle plusieurs fils. Voici un exemple de 1-chemin constitué d'un triplet de 1-cellules associées respectivement à un entier naturel, une liste et un booléen :

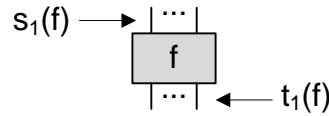


Le montage en parallèle (ou juxtaposition) est noté \star_0 .

Opérations - Les *opérations* sont représentées par des circuits, appelés *2-chemins*. Ils sont orientés, habituellement du haut vers le bas. Les opérateurs sont représentés par des composants appelés *2-cellules*. On compose les 2-chemins ou associant les 2-cellules de deux façons : soit en les montant en parallèle (juxtaposition notée \star_0), soit en les montant en série (connection notée \star_1) :

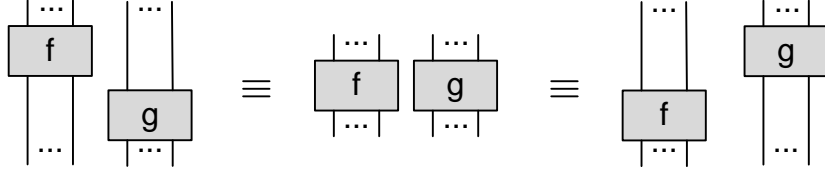


Pour chaque composant, on appelle les fils qui entrent sa *source* et les fils qui sortent son *but*. Ainsi, on appelle *1-source* (noté s_1) et *1-but* (noté t_1) les 1-chemins entrant et sortant d'un 2-cellule (et par extension sortant d'un 2-chemin).



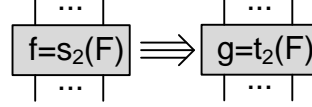
Comme les sources et buts des 2-cellules sont typés, si on veut connecter deux 2-cellules f et g ($f \star_1 g$) comme sur le précédent schéma, cela ne peut se faire que si le 1-but de f est égal au 1-source de g .

Une particularité importante des 2-chemins est que l'on peut représenter une même opération de plusieurs manières. En particulier, s'il est interdit de croiser ou casser des fils, on peut néanmoins les étirer ou les contracter, ce que l'on peut écrire graphiquement ou algébriquement :



$$(f \star_0 s_1(g)) \star_1 (t_1(f) \star_0 g) \equiv f \star_0 g \equiv (s_1(f) \star_0 g) \star_1 (f \star_0 t_1(g))$$

Calculs - Enfin, on dispose de *3-chemins* ou *chemins de réécriture*. Ils transforment un 2-chemin donné : sa 2-source, en un autre 2-chemin appelé son 2-but. Ces 3-chemins sont composées de *3-cellules* correspondant à des règles de réécriture locales. Pour constituer une 3-cellule (ou règle de réécriture), il est nécessaire que sa 2-source (ou *membre gauche*) et son 2-but (ou *membre droit*), aient mêmes 1-source et 1-but.



Ainsi on peut avoir la 3-cellule $F : f \Rightarrow g$ si et seulement si :

$$\begin{aligned} s_1((s_2(F)) = s_1((t_2(F)) \ \&\& \ t_1((s_2(F)) = t_1((t_2(F)) \\ \iff \\ s_1(f) = s_1(g) \ \&\& \ t_1(f) = t_1(g) \end{aligned}$$

On pourrait encore aller plus loin dans la compréhension de la structure des polygraphies, mais ce qui a été présenté suffit dans le cadre de ce projet pour introduire les programmes polygraphiques.

2.2 Programmes polygraphiques

Définition : Un *programme polygraphique* est un polygraphe particulier de dimension 3. L'ensemble de 2-cellules d'un programme polygraphique se divise en trois catégories distinctes : les *structures*, les *constructeurs* et les *fonctions*. De plus, l'ensemble de 3-cellules d'un programme polygraphique se divise suivant deux catégories : les *règles de structures* et les *règles de fonctions*.

2.2.1 Constructeurs

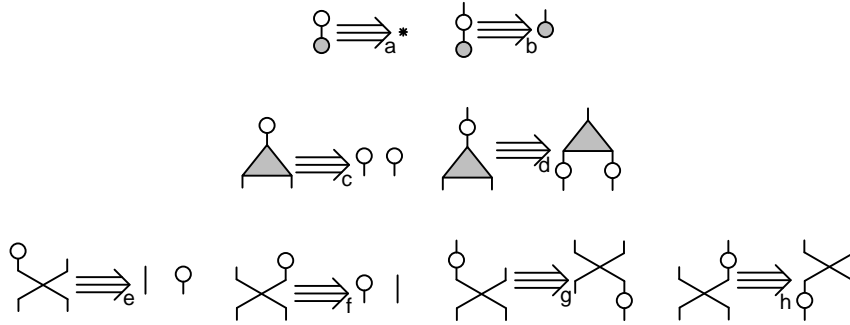
On définit l'*arité* d'une 2-cellule par la taille (la largeur en nombre de fils) de sa source et sa *coarité* par la taille de son but. Les constructeurs sont définis de façon générale comme les 2-cellule avec une coarité égale à un. Si on revient à l'exemple de l'introduction sur les entiers naturels, on avait défini les constructeurs $Zero : \circlearrowleft$ et $Succ : \circlearrowleft$

2.2.2 Structures

Cellules - À chaque 1-cellule, on associe trois structures : un effaceur, une duplication et un croisement. De plus, pour chaque combinaison de deux types distincts, on associe deux cellules de croisement (une pour chaque sens). Par exemple, si on revient aux entiers naturels, on définit les structures

suivantes : ,  et  respectivement pour l'effacement, la duplication et le croisement entre entiers naturels.



Règles - Ensuite, on associe à ces structures des 3-cellules de structures définissant des règles de réécritures. On a ainsi autant de 3-cellules que l'on a de combinaisons possibles de constructeurs en entrée de structures. Sur les entiers naturels, on aura donc les règles de structures suivantes :



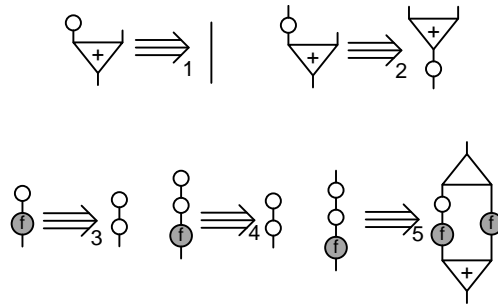
Remarque : on note $*$ le 1-chemin vide.

2.2.3 Fonctions

Cellules - Les fonctions correspondent aux autres 2-cellules. Il n'y a pas de contraintes sur leurs sources et buts. Ils correspondent aux autres opérateurs que les structures. Par exemple, on définit sur les entiers naturels les fonctions

suivantes : l'*addition*  et *fibonacci* . Elles ont le même but mais des sources différentes.

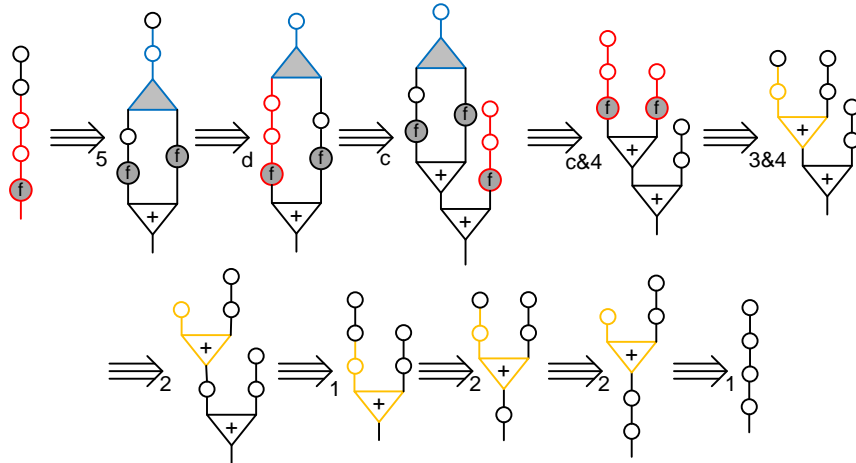
Règles - Les règles associées aux fonctions correspondent aux règles de calculs. Elles sont au coeur des programmes polygraphiques. Comme pour les structures, elles décrivent les transformations à effectuer lorsque l'on a des constructeurs au voisinage des fonctions (au lieu des structures). Elles sont donc de la forme $C \star_1 f$ avec C un 2-chemin composé de constructeur et f une fonction. Par exemple, pour les deux fonctions introduites au-dessus, on associe les règles suivantes :



Remarque : Il a été choisi de distinguer les cellules de structures des cellules de fonctions utilisées pour le calcul. En effet, dans les langages de programmations classiques, les permutations, duplications et écrasements sont fournis via la syntaxe. Ce n'est pas le cas pour les polygraphes. Dans un premier temps, on les distinguera donc des fonctions même si elles ont un fonctionnement similaire.

2.2.4 Calculs

Pour effectuer des calculs avec des programmes polygraphiques, on considère un 2-chemin puis on applique tant que possible les règles de réécritures décrites par les 3-cellules de fonctions et de structures décrites par le programme. On essaie donc de calculer $\text{fibonacci}(3)$ avec les éléments introduits précédemment :



On remarque au passage que l'on peut effectuer plusieurs opérations de réécriture d'une étape du calcul à une autre.

2.3 Modèles utilisés

On construit à présent un modèle UML des programmes polygraphiques (fig 1). Cela servira de base pour mettre en place les structure de données qu'on utilisera pour programmer le comportement des programmes polygraphiques (en Tom) et exporter les données (en XML). On construit au passage un modèle UML pour les polygraphes que l'on met en parallèle avec le premier pour illustrer les liens étroits entre les deux structures (Fig2).

2.3.1 Retour sur les structures

Comme énoncé, précédemment, les cellules de structures sont un peu particulières dans le sens où l'on est obligé de les expliciter qu'on on calcul avec des polygraphes de dimension 3 mais qu'elles ont un comportement identique aux cellules de fonction. De plus, on remarque que la connaissance de l'ensemble des 1-cellules utilisées dans le programme permet de connaître toutes les 2-cellules de structures nécessaires. Ensuite, la connaissance de l'ensemble des constructeurs utilisés dans le programme permet alors de définir de manière systématique les 3-cellules de structure (on reviendra d'ailleurs sur ce point ultérieurement). Une idée simplificatrice pour la suite du travail, surtout dans le but d'effectuer du filtrage, consiste à assimiler les 2-cellules de structure au x 2-cellules de fonction. Par contre la distinction constructeur/fonction reste indispensable.

2.3.2 Gom

Afin de pouvoir travailler de la manière la plus efficace possible avec Tom, on représentera les programmes polygraphiques en termes Gom. Gom est un générateur de données typées. Cela revient à définir une grammaire pour les programmes polygraphiques. Ainsi, pour le projet, nous utiliserons les termes Gom suivants :

```
OnePath = Id()
  | OneCell (Name:String)
  | OneC0 (OnePath*)

TwoPath = TwoId (onePath:OnePath)
  | TwoCell (Name:String,Source:OnePath,Target:OnePath,Type:CellType)
  | TwoC0 (TwoPath*)
  | TwoC1 (TwoPath*)

ThreePath = ThreeCell (Name:String,Source:TwoPath,Target:TwoPath)

CellType = Constructor()
  | Function()
```

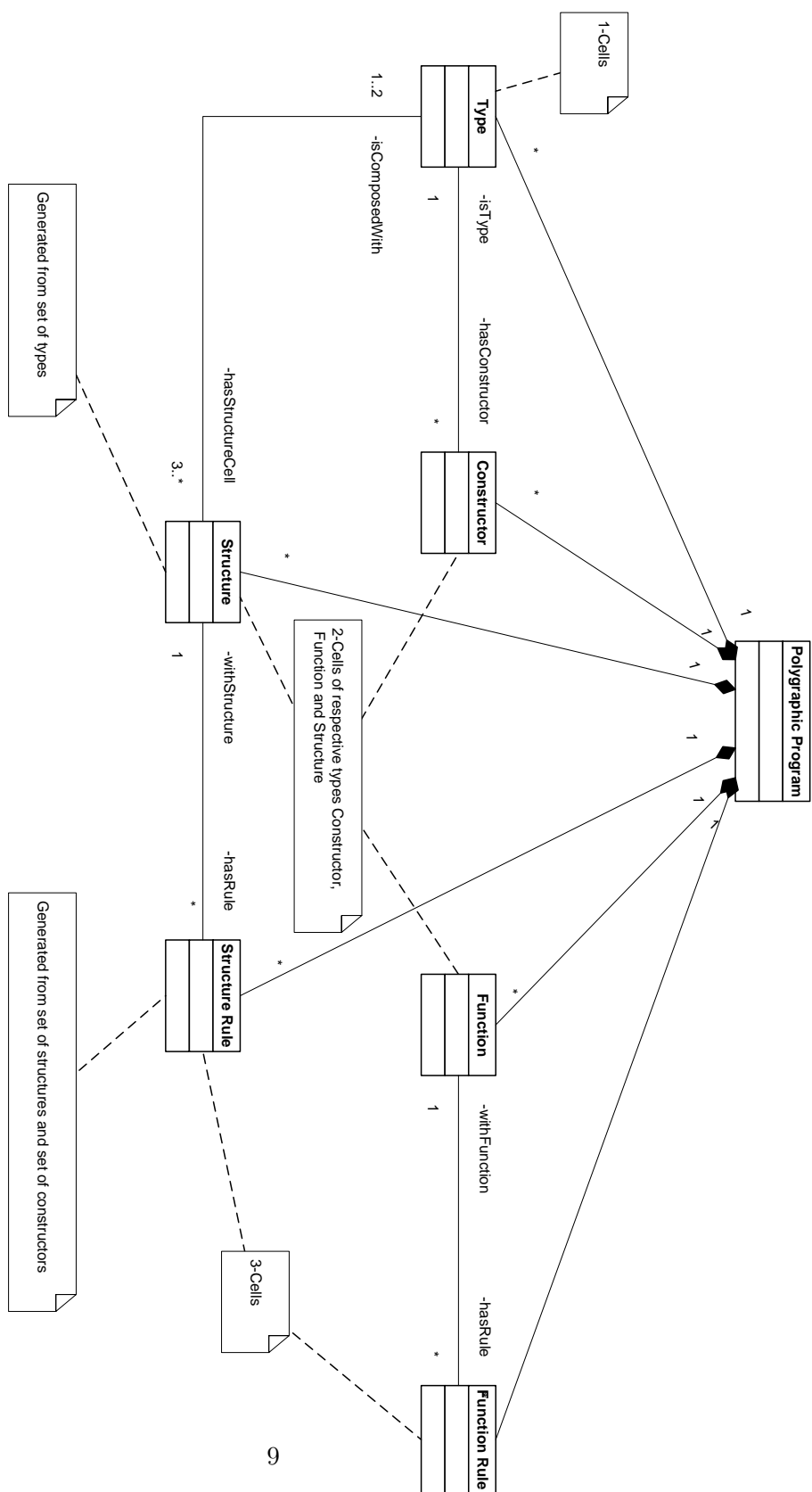


FIG. 1 – Modèle UML des programmes polygraphiques

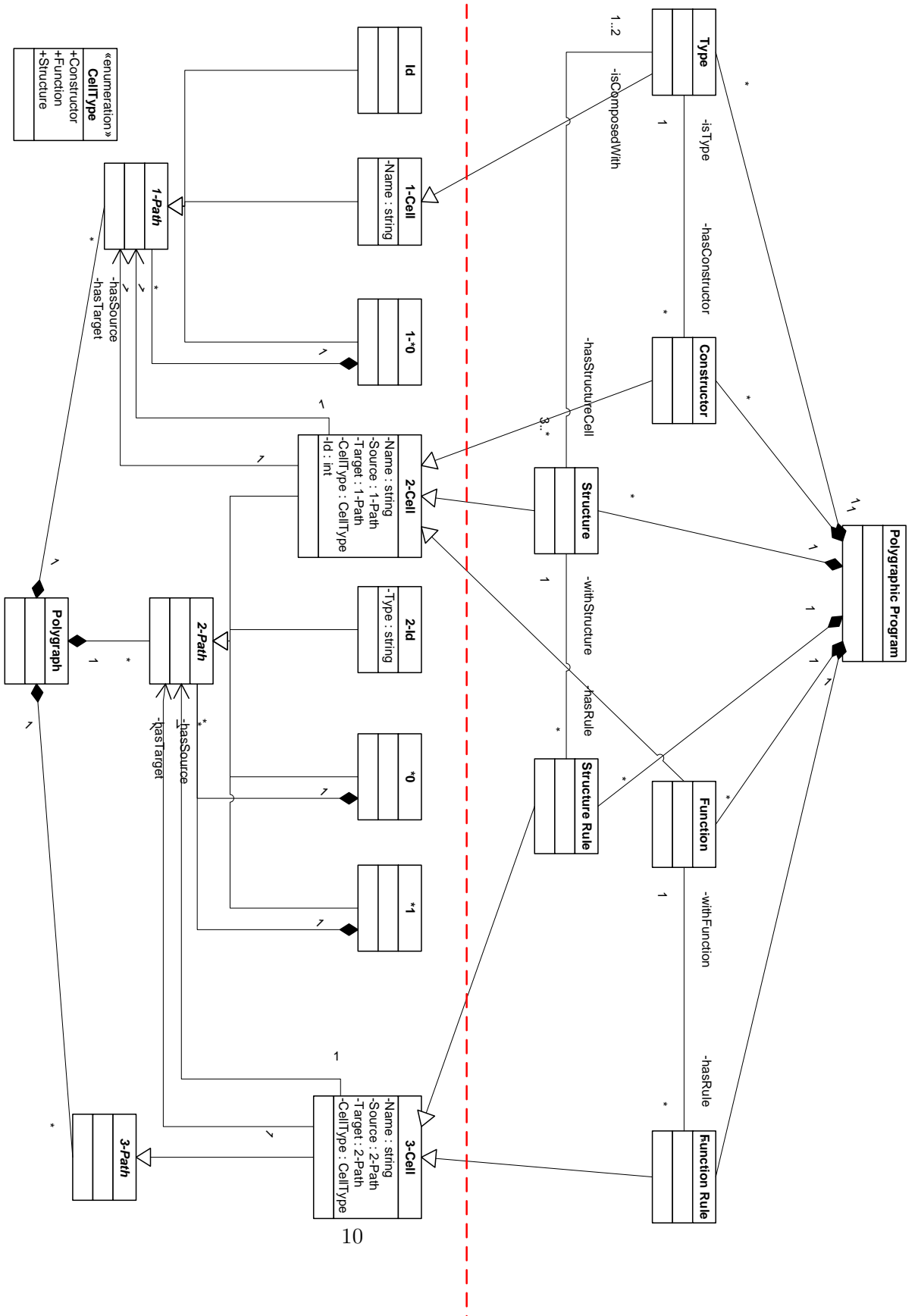


FIG. 2 – Lien entre polygraphes et programmes polygraphiques

1-chemins - $\text{Id}()$ représente le 1-chemin (*OnePath*) vide $*$. On choisit de définir un type par son nom. Pour cela, on associe un attribut *Name* aux 1-cellules (*OneCells*). On définit \star_0 sur les 1-chemins à l'aide de l'opérateur $\text{OneC0}(\text{OnePath}^*)$, OnePath^* étant une liste de 1-chemins. C'est un opérateur associatif avec le chemin vide $\text{Id}()$ comme élément neutre.

Types de cellules - On définit les deux types de cellules (*CellTypes*) *Constructor* et *Function*, respectivement pour les constructeurs et les fonctions. Comme énoncé précédemment, on ne distinguera plus les structures des fonctions pour mener les calculs.

2-chemins - On choisit de définir une 2-cellule (*TwoCell*) par un nom, sa 1-source, son 1-but et son type, d'où les attributs *Name*, *Source* et *Target* de type 1-chemin et le type de type *CellType*. Ensuite, on simule la possibilité d'étirer des fils avec les termes *TwoId*. Ils correspondent à la représentation de leur attribut de type 1-chemin dans l'espace des polygraphes de dimension 2. De façon similaire aux 1-chemins, on définit sur les 2-chemins (*TwoPaths*) les opérateurs $\text{TwoC0}(\text{TwoPath}^*)$ et $\text{TwoC1}(\text{TwoPath}^*)$ respectivement pour les constructions \star_0 et \star_1 . Ce sont aussi des opérateurs associatifs à élément neutre. L'élément neutre étant la représentation du chemin vide par le terme $\text{TwoId}()$ i.e. $\text{TwoId}(\text{Id}())$.

Exemples écritures en termes, représentation graphique et arbre

2.3.3 XML

3 Reécriture de termes sur les programmes polygraphiques avec TOM

(fonctionnement du programme généré par le compilateur)

réécriture avec tom

Possibilité de représenter un même circuit avec plusieurs arbres équivalents (\star_0 et \star_1)

3.0.4 Hooks Gom

vérifications à la construction et report des erreurs, vérification syntaxique

3.1 Normalisation

on ne peut pas avoir de forme unique (reference ?) mais on peut le faire assez bien au-dessus des fonctions

3.2 Gravite

on fait tomber les constructeurs vers les fonctions

3.3 regles

membre gauche=pattern
membre droit=action
pb des connections

3.4 strategies

comment parcourir un arbre au plus efficace ?
differentes idées
solution retenue

4 Compileur

4.1 architecture d'un programme polygraphique

schema xml jusqu'au .class

4.1.1 partie commune

normalisation, gravité, termes gom, transformations xml vers term et vice-versa

4.1.2 partie specifique

ensemble de cellules et de règles

4.2 Fonctionnement du compilateur

4.2.1 generation de la partie commune

copie de fichiers dans le dossier de destination du programme

4.2.2 generation de la partie specifique

au sein du programme final

- génération des cellules de structures
- génération des règles de structures
- transformation des règles en stratégies
- stratégie de parcours global

5 Outils développés en parallèle

5.1 transformation term vers xml et inverse

classe dans la library (la seule pour l'instant)

5.2 bibliotheque de fonctions

nat, list et booléens ainsi que des opérations élémentaires $+$, $/$, $*$, $-$, and, or, sort, merge, cube, carre, fibonacci sur listes et entiers

5.3 generateur de programme.xml

generation d'un programme polygraphique en XML avec les éléments sus-mentionnés

5.4 lecture du resultat

lecture d'un résultat de type nat, list ou booléens s'il ne reste que des constructeurs

5.5 generateur de tests

génération d'un ensemble de tests à partir des éléments de la bibliothèque de cellules de base
assez facile de changer ou ajouter des tests
génération d'un programme qui effectue les tests (codés en xml) sur un programme cible

6 Conclusion

6.1 points-clés

- tom parfaitement adapté
- plusieurs formes polygraphiques donc normalisation
- generation des cellules et regles de structures
- generation des regles de reecritures en TOM

6.2 ouverture

- representation graphique : travaille de GB avec des pointeurs et blender
- editeur graphique pour eclipse en utilisant GMF a partir du metamodelle présenté
- garder la bibliothèque de base pour la proposer lors de la création d'un programme (attention, ajouter des fonctions inutiles ralentit l'exécution du programme)

- TPDB : grande bibliothèque de tests, convertir les exemples automatiquement en programme polygraphique ??