Graphe unificateur

Par Vincent Lesbros

2020/04/26

Page : https://www.cyclonium.com/atelier/unification/unificateur.html

Résumé : Implémentation JavaScript d’un unificateur basé sur la construction d’un graphe.

Équation d’arbres

Définitions

On considère un ensemble d’***expressions*** composées de ***variables*** et de ***termes***.

Un ***terme*** possède de 0 à *n* ***arguments*** qui sont des *expressions*.

L’***arité*** du *terme* est le nombre de ses *arguments*.

Un *terme* d’*arité* nulle est une ***constante***.

Un *terme* d’arité supérieur à 0 est un ***prédicat***.

Toutes les *expressions* sont nommées. Par convention, le nom d’une *variable* commence par une majuscule, alors que les noms des *prédicats* et des *constantes* commencent par une minuscule.

Un ***contexte*** est un ensemble de *variables* identifiées par leur nom.

Une ***variable*** appartient à un ***contexte*** et est soit **libre**, soit **liée** :
- une variable est ***libre*** tant qu’on ne lui a pas affecté de valeur.
- une variable est ***liée*** si on lui a affecté une *expression* différente d’elle-même en valeur.

Il s’en suit qu’une expression peut être représentée par une arborescence, ou graphe dirigé acyclique, mais pouvant contenir des éléments partagés, et conservant l’ordre des descendants des nœuds (voir le graphe ci-dessous).

Équation

On pose l’équation en écrivant une ***clause*** de la forme :

expression gauche = expression droite

où les *variables* de l’*expression* gauche sont dans le même *contexte* que celles de l’*expression* droite.

Exemple :
la clause
f(a, X) = f(a, g(Y, X))
représentée graphiquement ci-contre.
La variable X est partagée par les deux expressions.

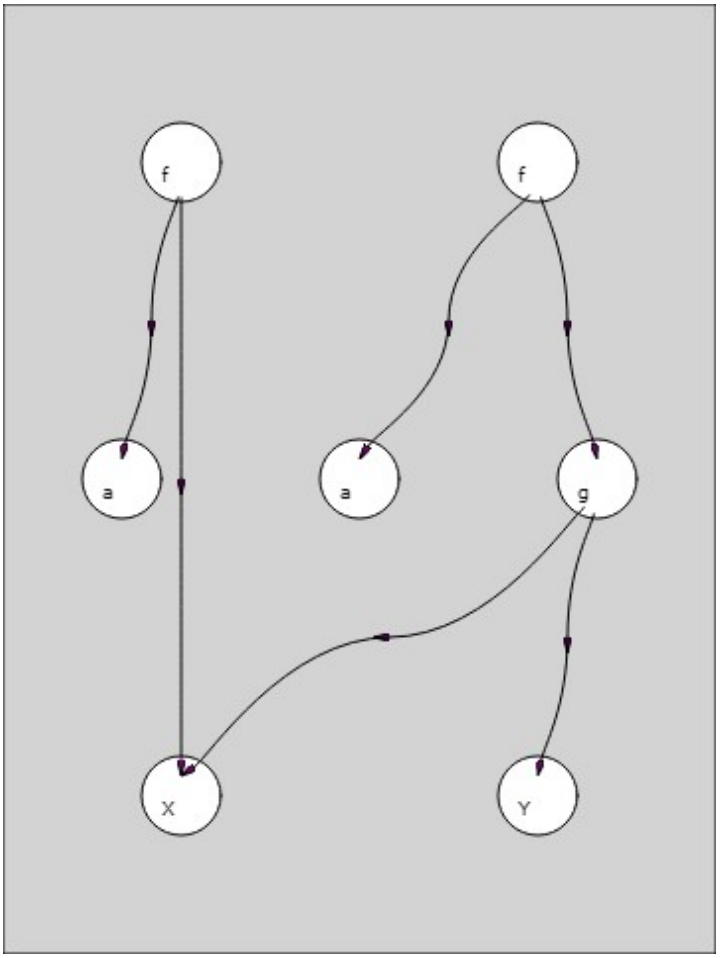


figure 1

L’unification

L’unification consiste à trouver les valeurs à affecter aux variables d’une *clause* pour que les *expressions* droite et gauche deviennent comparable si on remplace les variables liées par leur valeur dans les arborescence gauche et droite avec les impératifs suivants :

- deux constantes s’unifient si elle portent le même nom.
- deux prédicats s’unifient s’ils portent le même nom, on la même arité, et que chacun des arguments respectifs dans l’ordre, s’unifient entre-eux.
- une variable libre peut s’unifier avec toute expression, elle devient alors liée à cette expression.
- une variable liée ne peut plus s’unifier qu’avec des expressions unifiable avec sa valeur d’après les points précédents.

Comme une variable peut être liée à une autre, liée elle-même à une troisième, etc... la valeur à unifier est la valeur de la dernière variable liée de la chaîne.

Grammaire

Dans le champ de saisie du haut de la page on peut inscrire des expressions conformes à la grammaire suivante :

Soit :
- une **clause** à unifier

Une **clause** est :
- une **expression** gauche, le signe « = », une **expression** droite.

Une **expression** est soit :
- un **terme**
- une **variable**

Un **terme** est soit :
- un **prédicat**
- une **constante**

Une **constante** est un **identificateur** commençant par une minuscule.
Les **identificateurs** commencent par une lettre et sont éventuellement suivis par des lettres et chiffres ou le blanc souligné « _ ».

Une **variable** est un identificateur commençant par une majuscule *a la Prolog*.

Un **prédicat** est un identificateur formé comme une constante, et suivi par une liste non vide d'**arguments** entre parenthèses et séparés par des virgules.

Les **arguments** sont des **expressions**.

Les éléments peuvent être séparés par des espaces et des retours à la ligne.

Exemples

constante : a
variable : X
prédicat : f(a, g(X))
clause : f(a, g(X,a)) = f(X, Y)
Dans le tableau suivant on a noté les expressions gauches en première colonne, les expressions droites en première ligne, et les cases contiennent le résultat de l'unification de chaque clause.

	a	**b**	**X**	**f(a)**	**f(X)**	**f(X,Y)**
a	⊨	⊭	⊨ {X ↦ a}	⊭	⊭	⊭
X	⊨ {X ↦ a}	⊨ {X ↦ b}	⊨ {X}	⊨{X ↦ f(a)}	⊨ {X ↦ f(X ↦ f(X ↦ f(X ↦ ...)))}	⊨ {X ↦ f(X ↦ f(X ↦ f(X ↦ ..., Y), Y), Y), Y}
f(a)	⊭	⊭	⊨ {X ↦ f(a)}	⊨	⊨ {X ↦ a}	⊭
f(a, g(X))	⊭	⊭	⊨	⊭	⊭	⊨ {X ↦ a, Y ↦ g(X ↦ a)}

table 1

⊨ indique une réussite et ⊭ un échec.
Le contexte à la fin de l'unification est donné entre accolades. Les variables libres s'impriment avec juste leur nom, et les variables liées avec leur nom et une flèche vers la valeur liée. Dans le cas ou le terme lié contient un sous-terme contentant lui-même la variable, ou dans tout cas cyclique de liaison de variables, des points de suspensions sont affichés au bout de trois tours de boucle.

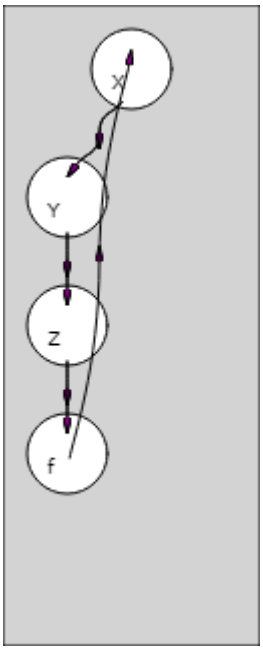
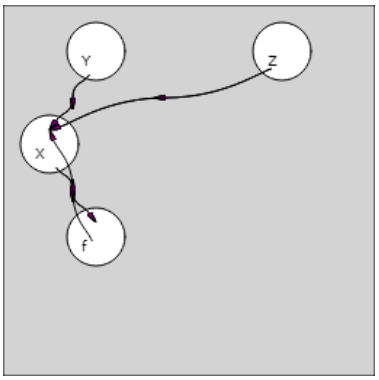
Par exemple : avec la clause h(X, Y, Z) = h(Y, Z, f(X)) on lie X à Y, Y à Z et enfin Z à f(X). On forme donc une boucle et le contexte final s'imprimerait : {X ↦ Y ↦ Z ↦ f(X ↦ Y ↦ Z ↦ f(X ↦ Y ↦ Z ↦ f(X ↦ ...))), Y ↦ Z ↦ f(X ↦ Y ↦ Z ↦ f(X ↦ Y ↦ Z ↦ f(X ↦ Y ↦ ...))), Z ↦ f(X ↦ Y ↦ Z ↦ f(X ↦ Y ↦ Z ↦ f(X ↦ Y ↦ Z ↦ ...)))} C'est tellement plus simple sous forme de graphe (ci-contre) :	
Avec la clause inversée : h(Y, Z, f(X)) = h(X, Y, Z) On lie Y à X, puis en unifiant Z à Y, on lie Z à X c'est-à-dire la dernière valeur de la chaîne, X étant toujours libre. En unifiant enfin f(X) à Z, on cherche la valeur de Z : Z vaut Y, et Y vaut X, donc on unifie f(X) à X simplement en liant X à f(X). Il y a toujours une boucle : {Y ↦ X ↦ f(X ↦ f(X ↦ f(X ↦ ...))), Z ↦ X ↦ f(X ↦ f(X ↦ f(X ↦ ...))), X ↦ f(X ↦ f(X ↦ f(X ↦ ...)))}	

figure 2

Méthode d'unification

L'idée pour unifier deux expressions est simplement de construire le graphe d'unification en associant les nœuds correspondants des deux expressions dans un parcours simultané. Lors de la création d'un nœud de ce graphe, les éléments face à face sont confrontés.

Base

Premièrement nous utilisons l'outil « Peg.js » `http://pegjs.org/` de David Majda pour créer un parseur capable de lire une clause sous forme de texte et former une structure d'objets Javascript correspondante.

Nous utilisons ensuite notre bibliothèque de fonctions traitant les graphes pour explorer ces objets tout en construisant un graphe. La ***figure 1*** montre un tel graphe.
Voici la fonction qui produit le graphe à partir du résultat de parsing[1] :

```
function grapheDeLaClause(gauche, droite) {
	function fonctionFils(noeud) {
		// noeud : Variable ou Terme
		return noeud.fils();
	}
	function fonctionÉtiquette(noeud) {
		return noeud.nom;
	}

	let g = new GrapheFiliation(fonctionFils, fonctionÉtiquette);
	g.noeud(gauche);
	g.noeud(droite);
	return g;
}
```

Les arguments gauche et droite sont les deux expressions. La classe `GrapheFiliation` de la bibliothèque permet de construire un graphe dirigé multiple et ordonné simplement à partir de deux fonctions :
- une fonction ramenant les fils d'un nœud étant donné ce nœud.
- une fonction pour donner l'étiquette à afficher sur un nœud dans la représentation graphique.

Ensuite, on ajoute des nœuds au graphe avec la méthode ***noeud***. En l'occurence, on ajoute la partie gauche et la partie droite de la clause.

Pour comprendre, il faut savoir comment nous avons représenté les clauses, termes, variables et contexte.

Une classe `UContexte` représente un contexte. Chaque instance contient un dictionnaire des noms de variable vers les variables. Une nouvelle variable n'est créée que par l'intermédiaire d'un contexte. Un nouveau contexte est créé à chaque analyse (un par *parsing*).

Une classe `UVariable` représente les variables
Une classe `UTerme` représente les prédicats et les constantes, la seule différence étant l'arité.

Pour la construction de ce premier graphe, la méthode ***fils*** est définie de la façon suivante dans la classe `UTerme` :

```
fils() {
	return this.arguments;
}
```

Le terme, constante ou prédicat ramène la liste de ses arguments, vide pour une constante.
Pour la classe `UVariable`, il n'y a pas de fils :

```
fils() {
	// dans le cadre du graphe de clause à unifier, la variable n'a pas de fils
	return [];
}
```

Graphe d'unification

Pour unifier, on construit un graphe où les nœuds sont des paires ou couples d'éléments à unifier.
Les noeuds pourront porter les résultats locaux de l'unification, réussite, échec, cause de l'échec.

1 analyse syntaxique

Pour ne pas refaire plusieurs fois l'unification de deux nœuds identiques, on crée un dictionnaire à deux étages pour retrouver les nœuds :

Le premier dictionnaire prend en clef un élément gauche, et délivre un second dictionnaire, qui à partir de la clef élément droite donne le nœud s'il existe.

Il suffit juste de décrire *« comment trouver les fils »* d'une paire donnée dans le graphe, et une fonction pour étiqueter les résultats.

Voici le code de la fonction ***unifier*** constructrice du graphe d'unification :

```
function unifier(gauche, droite) {
	// gauche et droite sont des termes ou variables : les expressions à unifier
	let gdico = new Map(); // g -> Map() d -> noeud : {gauche: g, droite: d, résultat:
échec ou réussite, cause: dans le cas d'un échec }
	let g = new GrapheFiliation(fonctionFils, fonctionÉtiquette);
	let racine = dico(gauche, droite);
	g.noeud(racine);

	function dico(g, d) {
		let dicogauche = gdico.get(g);
		if (!dicogauche) {
			dicogauche = new Map();
			gdico.set(g, dicogauche);
		}
		let noeudGaucheDroite = dicogauche.get(d);
		if (!noeudGaucheDroite) {
			noeudGaucheDroite = { gauche: g, droite: d };
			dicogauche.set(d, noeudGaucheDroite);
		}
		return noeudGaucheDroite;
	}

	function fonctionFils(noeudGD) {
		let g1 = noeudGD.gauche.déréférencer();
		let d1 = noeudGD.droite.déréférencer();
		// g1 et d1 sont des termes ou des variables
		let comparaison = g1.confronte(d1);
		if (comparaison === true) {
			noeudGD.résultat = true;
			return []
		}
		if (comparaison === false) {
			noeudGD.résultat = false;
			noeudGD.cause = { gauche: g1, droite: d1 };
			return []
		}
		// une liste de fils à créer :
		// { àgauche: gauche.arguments, àdroite: this.arguments }
		let fils = [];
		for (let k = 0; k < comparaison.àgauche.length; ++k) {
			let filsK = dico(comparaison.àgauche[k], comparaison.àdroite[k]);
			fils.push(filsK);
		}
		return fils;
	}

	function fonctionÉtiquette(noeudGD) {
		let test = noeudGD.gauche.nom + "~" + noeudGD.droite.nom;
		if (noeudGD.résultat === true) {
			return "⊨ " + test; // VRAI
```

```
                }
                if (noeudGD.résultat === false) {
                        return "⊭ " + test; // PAS VRAI
                }
                else {
                        return test;
                }
        }

        return g;
}
```

La fonction ***unifier*** utilise trois sous-fonctions. La première ***dico***, retrouve ou fabrique les instances de nœuds de la forme `{ gauche: g, droite: d }`. où *g* et *d* sont des expressions trouvées face à face au cours du parcours de construction.

Ce qui est important, c'est la fonction `fonctionFils` qui reçoit un nœud gauche/droite et doit déterminer ses fils dans le « graphe de calcul » de l'unification.

On commence par y déréférencer les expressions :

Pour un terme, prédicat ou contante, déréférencer ne fait rien :

```
déréférencer() {
        return this;
}
```

Ou, plutôt, ramène lui-même. Mais pour une variable :

```
déréférencer() {
        // il faut aller chercher la valeur, le plus loin possible, sans perdre la référence
        if (this.estLiée) return this.valeur.déréférencer();
        return this;
}
```

On avance dans la chaîne de liaison tant qu'on n'a pas une variable libre ou un terme.

Vient le moment de confronter les parties gauche et droite déréférencées *g1* et *d1*.

```
let comparaison = g1.confronte(d1);
```

La responsabilité est donnée aux expressions par la méthode `confronte` et la sous-méthode `confronteTerme` par *double dispatching*[2].

Commençons par les variables :

```
confronte(droite) {
        // une variable est confrontée à un terme ou une variable droite
        // 1/ si c'est la même : réussite
        if (this === droite) {
                return true;
        }
        // 2/ si la variable est libre : on l'affecte et on réussit
        if (this.estLibre) {
                this.affecter(droite); // ça peut être une variable ou un terme
                return true;
        }
        // sinon on confronte avec la valeur
```

2 Quand un objet reçoit un message, on connaît sa classe mais pas celle de l'argument. En renvoyant un message à l'argument qui intègre dans le nom de la méthode la classe du premier récepteur, on connaît alors les classes des deux objets en présence.

```
        // note : l'argument droite à été déréférencé (et le récepteur aussi).
        return this.valeur.confronte(droite);
}
confronteTerme(gauche) {
        // gauche n'est pas une variable
        if (this.estLibre) {
                this.affecter(gauche);
                return true;
        }
        return this.valeur.confronteTerme(gauche);
}
```

On résumera les cas ci-dessous dans un tableau (table 2).

Voici le code pour les termes :

```
confronte(droite) {
        // 1/ si c'est le même terme : réussite
        if (this === droite) {
                return true;
        }
        // sinon, on sait que le récepteur est un terme et on « double-dispatch » :
        return droite.confronteTerme(this);
}
confronteTerme(gauche) {
        // gauche est garanti être un terme.
        // les prédicats et constantes doivent porter le même nom pour s'unifier
        if (this.nom !== gauche.nom)
                return false;
        // les prédicats doivent avoir la même arité pour s'unifier
        if (this.arité() !== gauche.arité())
                return false;
        // et enfin :
        // chaque argument doit s'unifier avec l'argument correspondant.
        // on prépare du travail pour plus tard.
        return { àgauche: gauche.arguments, àdroite: this.arguments };
}
```

En résumé :

Pour les variables	**La même variable**	**Une expression quelconque**	**Un terme (gauche)**	
Une variable	**Réussite**.			
Une variable libre		Lier la variable à l'expression et **Réussite**.	Lier la variable (de droite) au terme (gauche) et **Réussite**.	
Une variable liée		Confronter la valeur (déjà déréférencée) de la variable à l'expression.	Confronter la valeur (déjà déréférencée) de la variable (de droite) au terme (gauche).	

Pour les termes	**Le même terme**	**Une expression quelconque**	**Un terme (gauche) de nom différent ou d'arité différente**	**Un terme (gauche) de même nom et arité**
Un terme	**Réussite**.	Double dispatch en renvoyant le message à droite.	**Échec**	Préparer la suite en consignant les arguments de gauche et de droite.

table 2

Revenons à la fonction `fonctionFils` de l'unificateur. On vient de récupérer, pour un nœud gauche/droite du graphe en construction le résultat de la confrontation décrit ci-dessus.

- En cas de **Réussite**, le noeud est affublé du résultat VRAI et il n'y a pas de fils à construire.
- En cas d'**Échec**, on marque le noeud à FAUX, on décrit la cause de l'échec en donnant les éléments déréférencés *g1* et *d1* comme explication. Il n'y a pas de fils à construire non plus. L'échec sera global de toute façon.
- Enfin on construit la liste de fils à explorer à partir de l'objet ramené par la case « Préparer la suite » et en utilisant la fonction de dictionnaire pour ne pas réanalyser les mêmes cas.

Nous aurons donc trois « types » de noeuds à la fin, des feuilles d'échec ou de réussite, et des noeuds de valeur indéterminée, avec de la descendance.

Exemple de graphe d'unification

On part de la clause : f(X, g(X)) = f(g(f(b), a), g(g(Y, Z)))

f(X, g(X)) = f(g(f(b), a), g(g(Y, Z)))

Les deux expressions droite et gauche sont séparées, il n’y a pas de variable en commun, mais les variables, X, Y et Z sont dans le même contexte.

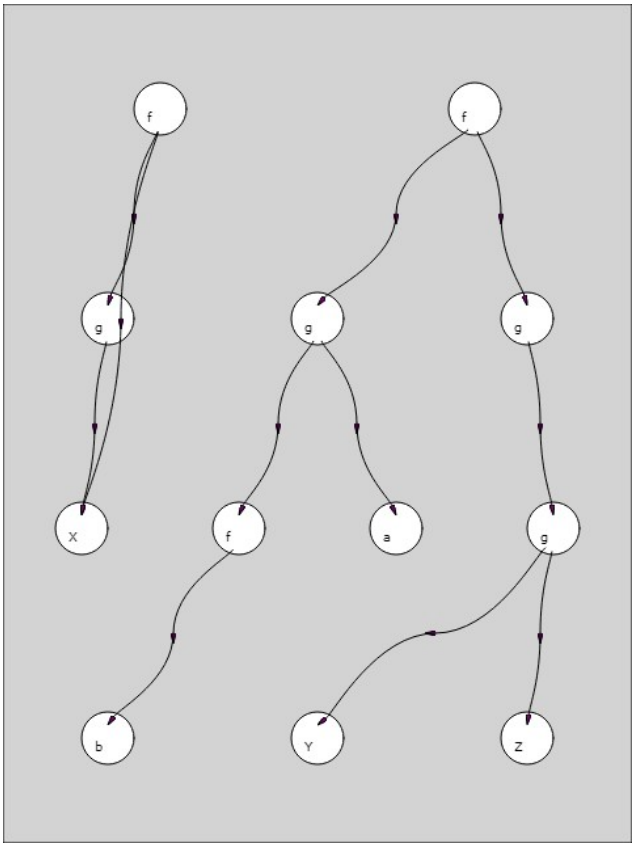


figure 3 La clause à unifier f(X, g(X)) = f(g(f(b), a), g(g(Y, Z)))

Le graphe d’unification

<{f~f,⊨ X~g,g~g,X~g,⊨ f~Y,⊨ a~Z}, {(f~f → ⊨ X~g),(f~f → g~g),(g~g → X~g),(X~g → ⊨ f~Y), (X~g → ⊨ a~Z)}>

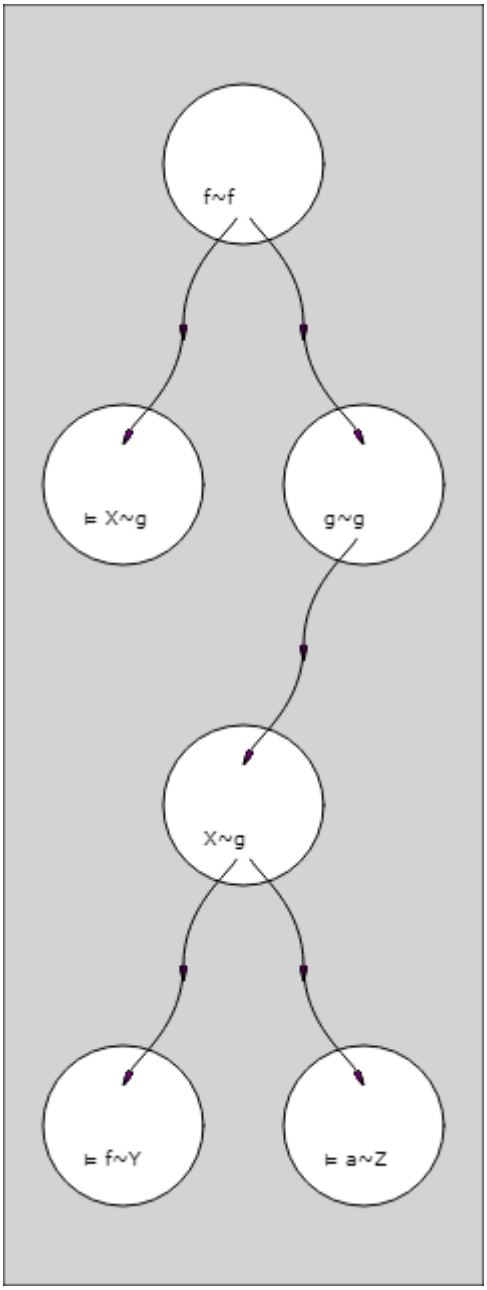


figure 4 Le graphe d’unification

Les étapes de construction du graphe :

1/ f(X, g(X)) = f(g(f(b), a), g(g(Y, Z)))

On commence par les deux premiers prédicats en f, de même arité. On crée le premier nœud f~f, et on prépare la suite : les deux fils X = g(f(b), a) et g(X) = g(g(Y, Z))

2/ X = g(f(b), a)

Mais oui, c’est vrai : X était libre. X devient liée à g(f(b), a), le nœud ⊨ X~g est une réussite.

3/ g(X) = g(g(Y, Z))

Nœud g~g, le nom et l’arité sont les mêmes. Donc idem, on prépare une suite : X = g(Y, Z), cette fois il n’y a qu’un fils : le noeud X~g.

4/ X = g(Y, Z)

X est liée à g(f(b), a). Donc en déréférençant, on doit observer : g(f(b), a) = g(Y, Z). Deux termes en g d'arité 2, il y a deux fils : f(b) = Y et a = Z.

5/ f(b) = Y

f(b) est un terme, il renvoie à la résolution de Y = f(b). Comme Y est libre, on lui affecte f(b). Le noeud ⊨ f~Y représente une réussite.

6/ a = Z

De la même façon, Z se retrouve liée à a dans la réussite : ⊨ a~Z.

Le résultat est le contexte suivant :

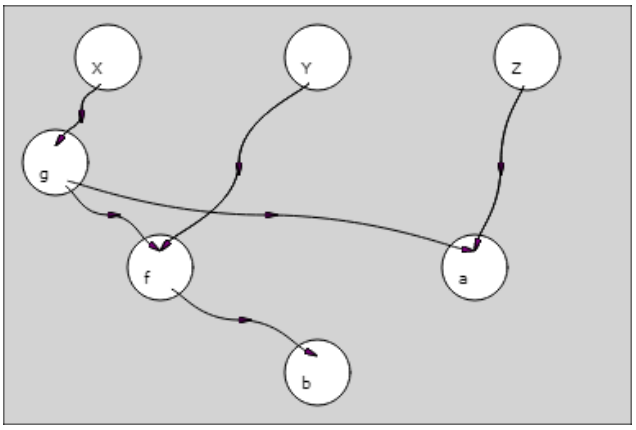
{X ↦ g(f(b), a), Y ↦ f(b), Z ↦ a} On peut voir que la valeur de Y est partagée avec un sous-terme de la valeur de X (étape 4).	

figure 5 Le graphe du contexte

L'unification est une réusite car aucun nœud du graphe d'unification (figure 4) n'est un échec.

Pour comparer avec un autre algorithme, la *Méthode Robinson*, la page :

https://www.cyclonium.com/atelier/unification/robinsonGraphe.html

Appliquée sur le même exemple, donne la trace suivante :
<= unifier : f(X, g(X)) et f(g(f(b), a), g(g(Y, Z))) σ = {}
<= reformuler : f(X, g(X)) et f(g(f(b), a), g(g(Y, Z))) σ = {}
première substitution : f(X, g(X)) et f(g(f(b), a), g(g(Y, Z))) σ = {}
inégalité : f(X, g(X)) et f(g(f(b), a), g(g(Y, Z))) σ = {}
disagreement : [X , g(f(b), a)]
inégalité : f(g(f(b), a), g(g(f(b), a))) et f(g(f(b), a), g(g(Y, Z))) σ = {X ↦ g(f(b), a)}
disagreement : [f(b) , Y]
inégalité : f(g(f(b), a), g(g(f(b), a))) et f(g(f(b), a), g(g(f(b), Z))) σ = {X ↦ g(f(b), a), Y ↦ f(b)}
disagreement : [a , Z]
=> unifier : Vrai σ = {X ↦ g(f(b), a), Y ↦ f(b), Z ↦ a}

Appliquer un test *Occur-check* ou résolution avec des contextes à graphes circulaires ?

Par exemple, si on tente d'unifier h(X,f(X, Y)) à h(Z, Z)

On trouve simplement :

Unification vraie dans le contexte : {X ↦ Z ↦ f(X ↦ Z ↦ f(X ↦ Z ↦ f(X ↦ ..., Y), Y), Y), Y, Z ↦ f(X ↦ Z ↦ f(X ↦ Z ↦ f(X ↦ Z ↦ ..., Y), Y), Y)}.

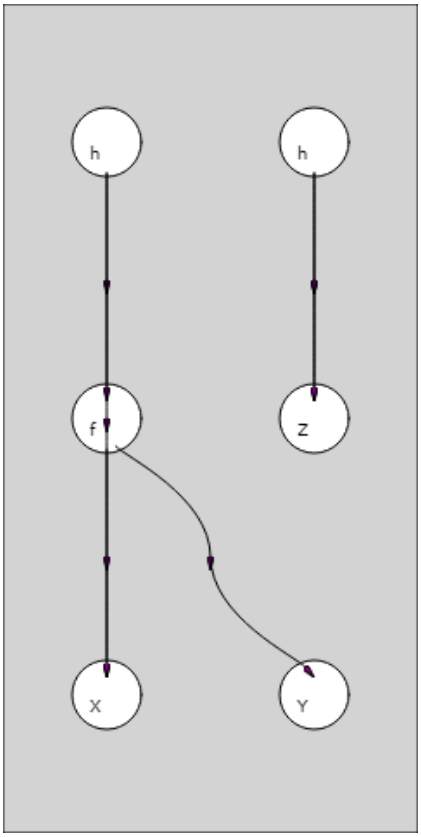
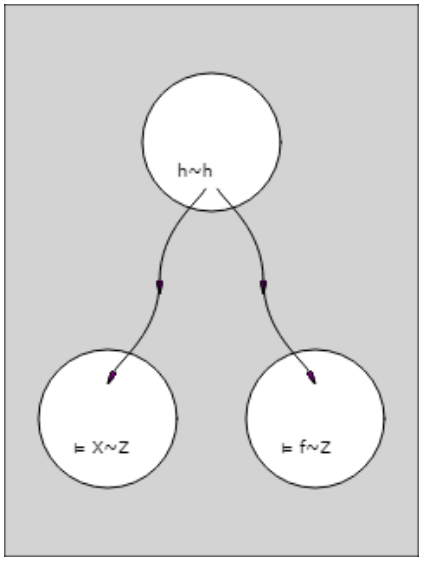
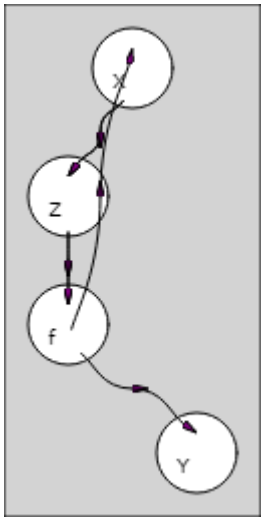
La clause : h(X,f(X, Y)) = h(Z, Z)	Le graphe d'unification	Le graphe du contexte, formant un cycle : X est liée à Z qui est liée à un terme contenant X.
		

figure 6

L'unification ne pose pas de problème ; les termes en h délèguent à leurs deux fils. Le premier trouve une variable X libre qui est liée à Z, et le second lie Z au terme qui lui fait face : f(X, Y).