RAIRO. Informatique théorique et applications

J.-L. Durieux

E. Saint-James

Une charpente de semi-unification

RAIRO. Informatique théorique et applications, tome 22, nº 2 (1988), p. 173-226

<http://www.numdam.org/item?id=ITA_1988__22_2_173_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 22, n° 2, 1988, p. 173 à 225)

UNE CHARPENTE DE SEMI-UNIFICATION (*)

par J.-L. DURIEUX ([1]) et E. SAINT-JAMES ([2])

Communiqué par J. E. PIN

Résumé. – *Un algorithme de Semi-Unification permettant la recherche de segments de listes de longueur variable dans des listes arborescentes du type S-expressions Lisp est développé par transformations successives à partir d'une définition de la Semi-Unification comme opération inverse de la substitution. La définition de la substitution et ses transformées successives sont exprimées sous forme d'équations récursives par cas disjoints, dans le style de J. Darlington. Les transformations principales sont l'inversion pour obtenir une première version, puis l'explicitation et la capture de la continuation pour obtenir des versions utilisables. La structure de l'algorithme obtenu peut servir de charpente pour la construction d'autres algorithmes, applicables à des structures arborescentes n-aires, ou à des compositions fonctionnelles telles que la logique combinatoire, ou pour l'introduction d'autres opérateurs de filtrage courants des systèmes à règles de productions.*

Abstract. – *This paper is mainly the development of a pattern-matching algorithm by stepwise transformations starting from a definition of pattern-matching as an inverse of substitution. This pattern-matching algorithm is able to recognize sublists of arbitrary length in tree-structured lists, mainly Lisp-like S-expressions. The early definitions and the intermediate results are expressed by means of recursive equations with separate cases, following J. Darlington's way. The main transformations are inversion and then making the continuation explicit and catching it under the form of auxiliary parameters. The structure of the resulting algorithm may be seen as a framework to develop similar algorithms, adapted to n-ary tree structures or to functional expressions like combinatory logic, or to add other elementary patterns like those occuring in rules based systems.*

1. INTRODUCTION

1.1. Motivations

Les algorithmes de semi-unification, ou **filtrage**, on fait l'objet d'une littérature abondante en raison de leurs nombreuses applications. Mais ces ouvrages

(*) Reçu juin 1986.
([1]) Laboratoire CERT-DERI, Toulouse.
([2]) Laboratoire L.I.T.P., Université Paris-VI.

exposent le plus souvent un algorithme adapté à un problème particulier, avec des solutions *ad hoc* qui occultent les concepts fondamentaux de **continuation** et de **type comportemental** pourtant sous-jacents. Nous exposons ici ces deux concepts qui déterminent une ossature générale pour de tels algorithmes, mais surtout, nous essayons de les faire surgir de l'expression même du problème par une suite de transformations progressives.

Les problèmes que nous traitons sont ceux du filtrage de listes arborescentes binaires, type LISP. Les applications que nous avons réalisées concernent l'**invocation par filtrage** dans les langages d'**Acteurs** et les algorithmes d'**abstraction** en **logique combinatoire** pour lesquels le filtrage constitue la forme d'expression la plus adaptée. Il s'agit d'interprétation dans les deux cas, mais le passage à une forme compilée peut s'en déduire par évaluation partielle [6, 7] (Emanuelson, 1980, 1982). La compilation ne répondait cependant pas à un besoin pour nos problèmes : Les données à filtrer ne sont pas très volumineuses mais le filtrage est utilisé fréquemment; il doit s'identifier au mécanisme de mise en correspondance entre paramètres et arguments dans le cas trivial d'invocation d'une fonction [5] (Durieux, 82); l'abondance des filtres en cours de mise au point limite surtout la place en mémoire, et on ne peut consacrer trop de temps à un prétraitement [13] (Saint-James, 82).

1.2. Démarche et principes de base

Comme beaucoup, nous considérons les filtres comme un **langage de programmation**, et le filtrage comme une interprétation, une évaluation d'un programme de filtrage appliqué à une donnée.

L'utilisation des concepts de **mémoire** et de **continuation** de la **sémantique dénotationnelle** permet alors d'exprimer le problème sous une forme qui conduit directement à une solution exempte d'artifices *ad hoc* (variables et indicateurs divers, linéarisations, etc.). [16] (Wand, 80). C'est cette idée qui est à l'origine de l'algorithme présenté. Mais nous essayons ici de faire surgir cette idée à partir d'un développement systématique par transformations successives d'une définition algébrique du problème du filtrage. [4, 17] (Darlington, 82; Wile, 83). Les concepts ci-dessus s'introduisent alors comme résultats de ces transformations.

Dans les premières étapes, nous introduisons les objets et opérations qui sont nécessaires à la définition du problème. Les propriétés de ces objets et opérations sont équationnelles et indiquent simplement les règles de calcul permises.

Les notions ainsi introduites initialement sont celles :

– de **données** et de **filtres**, expressions de même structure, bâties avec les constructeurs usuels des listes sur des objets élémentaires distincts,

– de **mémoire** (qui s'identifie avec le graphe d'une fonction de substitution) et les opérations correspondantes **d'application** et de « **sommation** », ainsi que dans la deuxième partie l'opération de concaténation et une opération de « **ou** » **non déterministe** qui exprime notre souhait de « trouver le bon chemin ».

La définition du filtrage en induit naturellement une expression récursive : la structure du filtre et de la donnée détermine la valeur du résultat, exprimée en fonction des résultats de filtrages partiels grâce aux opérations ci-dessus.

Formellement, le filtrage est **l'opération inverse**, non nécessairement déterministe, de la **substitution**. D'un point de vue Algorithmique, le filtrage est la construction dynamique de cette substitution sous la forme d'une **mémoire** mise à jour par étapes successives. Nous passons de la définition à l'expression algorithmique en l'inversant cas par cas. Cette méthode de construction a été étudiée par D. Eppstein [8] (Eppstein, 85) à la suite de R. E. Korf [11] (Korf, 81).

Les étapes suivantes consistent à transformer ces expressions initiales en « déplaçant la sémantique » des opérations vers le mécanisme d'évaluation récursive et passage d'arguments. Il ne s'agit pas de **traduction** de ces opérations sous forme de fonctions auxiliaires, mais **d'élimination** en utilisant leurs propriétés pour déplacer les symboles dans les équations. Les propriétés glissent progressivement vers les définitions de fonctions et les passages d'arguments. Les opérations se vident de leur sens jusqu'à n'être plus que des délimiteurs.

Nous n'appliquons pas seulement des schémas de pliage et dépliage, mais une stratégie d'**explicitation du contexte**, puis de « **capture de continuation** » introduite par M. Wand. [16] (Wand, 80). Ces techniques diffèrent de celles de [12] (Pettorossi, 86) qui recherche une périodicité dans les appels récursifs pour mémoriser les résultats utiles (« tupling ») mais sont aussi plus dirigées que celles de [4] (Darlington, 82) : il n'est pas nécessaire d'improviser des « eureka », car les propriétés des opérations permettent d'organiser le contexte, et la transformation de Wand joue le rôle du « tupling ».

Les emboîtements de filtres dans les structures arborescentes nécessitent de tenir compte à chaque instant du contexte. Il faut pouvoir reprendre le filtrage aux points où des choix sont possibles en exploitant les contraintes découvertes ultérieurement. C'est précisément le but du concept de **continuation** de figer un calcul dans son contexte pour en disposer comme d'un argument supplémentaire. C'est ici l'élimination du « ou » non déterministe qui en fera apparaître progressivement la nécessité. Une suite de regroupements et modifications du contexte permis par les propriétés des opérations utilisées fait apparaître des sous-termes importants, que la transformation de Wand change alors en mémoire et en continuations de succès et d'échec.

1.3. L'algorithme obtenu

Le filtrage se réalise ainsi en une suite d'**états de filtrage**, caractérisés par :

– le filtre et la donnée en cours de filtrage, localement.

– la mémoire, qui incarne la partie gauche, déjà traitée.

– la continuation de succès, en cas de réussite du filtrage local.

– la continuation de reprise, en cas d'échec du filtrage local; celle-ci précise la conduite à tenir dans un tel cas : arrêt définitif, ou bien nouvelles tentatives à partir des choix rencontrés.

Les filtres élémentaires se classent alors en **types comportementaux**, selon les typologies suivantes :

– Normal : N'enrichit aucune des deux continuations.

– Conjonctif : (et) Enrichissement de la continuation de succès.

– Disjonctif : (ou) Enrichissement de la continuation de reprise.

– Variables : Enrichissement ou modification de la mémoire.

– Constantes : Ne modifient pas la mémoire.

Ces deux typologies se croisent pour décrire le comportement des filtres élémentaires, que nous considérons, mais d'autres filtres élémentaires peuvent s'ajouter dans ce schéma. C'est ce qui justifie le titre de « Charpente de Semi-Unification » donné à ce papier.

Nous donnons pour conclure le code Le-Lisp des fonctions obtenues, où les opérations initiales se sont concrétisées en de simples doublets, leur signification ayant migré dans les appels de fonctions. Les appels de fonctions étant post-récursifs croisés, la traduction en style impératif est triviale sinon automatique.

2. DÉFINITION FORMELLE DU FILTRAGE

2.1. Constructeurs et types génériques

DÉFINITION : On appelle signature un triplet $\langle S, \Sigma$, **profil** $\rangle$ où S est un ensemble fini de symboles appelés « Sortes », Σ un ensemble fini de symboles appelés opérateurs, et **profil** une fonction de Σ dans l'ensemble des paires (**w**, **s**) où **w** est une chaîne de symboles de S, éventuellement vide, **s** un symbole de S. Il est habituel d'écrire ces paires (**w**, **s**) sous la forme **W** $- >$ **s**.

Il est bien connu qu'une signature ainsi donnée définit une algèbre de termes, représentant canonique de l'algèbre initiale dans la catégorie des algèbres de même signature. [1] (Arbib et Manes, 75). Une telle algèbre initiale peut être étendue en une algèbre libre en introduisant des variables, mais nous réserverons cette extension à la désignation de schémas, dénotant un ensemble de termes avant une forme commune dans les expressions que nous traiterons. Les symboles qui n'auront pas été utilisés dans une signature seront implicitement de telles variables.

Nous considérons dans ce qui suit des signatures à deux sortes, $S = \{\mathbf{A}, \mathbf{T}\}$ où **A** désignera la sorte des Atomes, indécomposables, et **T** la sorte des Termes composés en appliquant les opérateurs de Σ aux Atomes et aux Termes.

Si Σ-Algèbre est le nom d'une signature à une sorte $\langle \{\mathbf{T}\}, \Sigma$, **profil** $\rangle$, et **Atomes** un ensemble arbitraire, nous noterons $\langle\langle$ Σ-Algèbre $\rangle\rangle$ **Atomes** l'algèbre à deux sortes $\langle \{\mathbf{A}, \mathbf{T}\}, \Sigma + \{\mathbf{i}\}$, **profil2** $\rangle$ où **profil2(s)** = si **s** = **i** alors **A** $- >$ **T** sinon **profil(s)**, **A** a pour support l'ensemble **Atomes**, **i** désigne l'injection canonique qui transforme chaque atome **a** en le terme correspondant **i(a)**. La sorte **T** a alors pour support l'ensemble des termes ainsi construits. Nous conviendrons d'identifier **i(a)** et **a** pour alléger l'écriture des termes.

L'informaticien peut y voir un type générique [9], mais le mathématicien pourra préférer la construction catégorique d'un foncteur adjoint au foncteur d'oubli. [1] (Arbib et Manes, 75). Cette construction nous permettra de construire nos algèbres de données et de filtres sur une même signature, et d'introduire les opérateurs de construction de filtres élémentaires de manière analogue. Ceci nous permettra d'introduire des opérateurs nouveaux, et aussi d'appliquer éventuellement plusieurs opérateurs distincts sur les occurences d'un même symbole.

Ayant défini notre métalangage, à partir de maintenant, nous allons donner un sens plus technique aux mots « symboles » et « variables » qui désigneront des constituants élémentaires des objets que nous traiterons. Ce seront les éléments d'ensembles explicitement désignés.

2.2. Algèbres des données et des filtres

Nous nous plaçons dans le cadre d'arbres binaires, ou des S-expressions LISP. Les arbres n-aires sont souvent représentés par de tels arbres binaires. Leur traitement direct nécessite des indices qui alourdissent les notations sans modifier le fond du problème.

Les Données et les Filtres sont des expressions construites respectivement sur des données élémentaires et des filtres élémentaires à l'aide des mêmes constructeurs. Nous allons en donner une définition commune en termes de S-algèbres, algèbres définies à partir des constructeurs des S-expressions de Mac-Carthy. Les ensembles d'objets élémentaires sont des paramètres de cette définition.

On appelle S-algèbre sur un ensemble d'atomes **A** l'algèbre hétérogène de termes **T** :

Sortes : A(atomes), T(termes).

Constructeurs : cons : $T \times T \to T$, nil : $\to T$.

Injection des atomes : $i : A \to T$.

Pour alléger les notations, on emploie usuellement les règles d'abréviation suivantes :

— $\text{cons}(t_1, t_2) = (t_1 . t_2)$ et $\text{cons}(t_1, (t_2 \ldots t_n)) = (t_1\, t_2 \ldots t_n)$

— $\text{cons}(t_1, \text{nil}) = (t_1 . \text{nil}) = (t_1)$.

Plus généralement nous omettrons les parenthèses et les points autant que cela sera possible.

Cette construction générique produit une S-algèbre pour tout ensemble d'atomes **A**, que nous noterons « S. algèbre »(**A**). Les Données et des Filtres sont construits à l'aide des mêmes constructeurs ci-dessus; seul l'ensemble **A** les différencie ([3]).

([3]) Ce type de construction se généralise aussi à un ensemble de constructeurs quelconques **Q** (*voir* Arbib and Manes, [1]), permettant de définir des Q-algèbres « Q-alg »(**A**). Cette généralisation permettrait d'étendre nos algorithmes à des arbres quelconques, mais la définition des segments devra aussi être adaptée.

S. algèbre des Données :

C'est la S. algèbre « S. algèbre » (**Symb**) sur un ensemble arbitraire **Symb** de symboles, qui sont les données atomiques. Nous la nommerons **Données.**

Exemples : a, b, ..., x, y, z, sont des données atomiques si l'ensemble des symboles contient les lettres minuscules. $(a.b)$ et $((a\,b)(y\,x))$ sont des données.

S. algèbre des Filtres :

C'est la S. algèbre sur l'ensemble des filtres élémentaires, obtenus en appliquant un opérateur classe de filtres, « $''$ » : Symb $\rightarrow$ Const ou « : » : Symb $\rightarrow$ Var, sur un symbole.

Exemples : $''a$, $''b$, ..., $''x$, $''y$, $''z$ sont des filtres de **constantes**, éléments de **Const** = « $''$ » (**Symb**) : $:a$, $:b$, ..., $:x$, $:y$, $:z$ sont des filtres de **variables,** éléments de **Var** = « : » (**Symb**) $(''a : x\,(''x : a))$ est un filtre composé.

Nous désignerons cette algèbre par **Filtres** = « S. algèbre » (**Const** + **Var**). Nous l'étendrons en ajoutant une nouvelle classe de filtres élémentaires dans la section 4. Enfin, nous emploierons le vocable **S-expr** pour les termes d'une S. algèbre quelconque, « S. algèbre » (**Symb**) ou « S. algèbre » (**Const** + **Var**) indifféremment.

2.3. Mémoires et filtrage

Nous appellerons **mémoire** toute fonction à domaine de définition fini de l'ensemble **Symb** dans l'algèbre des données. On la représente par l'ensemble fini des couples qui constituent son graphe : $\{s \rightarrow \mathbf{m}(s)\}$; $\mathbf{m}(s)$ désigne l'image du symbole s par la mémoire $\mathbf{m}$. Ceci permettra d'appliquer aux mémoires les opérations et relations usuelles pour les ensembles.

Cette notion de mémoire est une version très simplifiée de celle définie dans [5] (Durieux, 81) sous le nom d'environnement. Nous l'utiliserons ici pour définir la substitution et le filtrage, et plus loin pour enregistrer les résultats successifs de filtrages partiels, d'où son nom.

Opérations sur les mémoires

Identifiant les mémoires à leurs graphes, il est naturel de les réunir par l'opération ensembliste $\cup$ d'union. Malheureusement, cette opération n'est pas interne : la réunion de deux graphes m_1 et m_2 n'est pas le graphe d'une mémoire m s'il existe un symbole x tel que $\mathbf{m}_1(\mathbf{x})$ et $\mathbf{m}_2(\mathbf{x})$ soient définis et distincts.

Nous sommes donc amenés à définir l'opération **partielle** $\langle + \rangle$:

$$\mathbf{m}_1 \langle + \rangle \mathbf{m}_2 = \mathbf{m}_1 \cup \mathbf{m}_2 \text{ lorsque } \mathbf{m}_1 \cup \mathbf{m}_2 \text{ est une mémoire}$$

Cette opération vérifie évidemment toutes les propriétés de $\cup$ sur son domaine de définition : **associativité, commutativité, idempotence,** et la mémoire vide est son élément neutre.

2.4. Mémoires, substitutions et filtrage

DÉFINITION : **substitution** associée à une mémoire **m** :

C'est le *morphisme* $\mathbf{h_m}$: **Filtres** → **Données** naturellement induit par les images des filtres atomiques :

$$\forall x \in \mathbf{Symb},\ \mathbf{h_m}(''x) = x$$

$$\forall x \in \mathbf{Symb},\ \mathbf{h_m}(:x) = m(x)$$

La généralisation à la S. algèbre est classique :

$$\forall t_1,\ t_2 \in \mathbf{Filtres},\ \mathbf{h_m}(t_1 . t_2) = (\mathbf{h_m}(t_1) . \mathbf{h_m}(t_2)) \in \mathbf{Donn\acute{e}es}$$

$$\mathrm{nil} \in \mathbf{Filtres},\ \mathbf{h_m}(\mathrm{nil}) = \mathrm{nil} \in \mathbf{Donn\acute{e}es}$$

Le morphisme $\mathbf{h_m}$ est évidemment complètement défini par la mémoire **m**.

Remarque : si **m** est une mémoire telle que $\mathbf{h_m}$ **(filtre) = donnée,** toute mémoire **m′** contenant **m** vérifie aussi $\mathbf{h_{m'}}$ **(filtre) = donnée.**

Le problème du filtrage d'un filtre **filtre** et d'une donnée **donnée** consiste à chercher s'il existe une mémoire **m** telle que : $\mathbf{h_m}$ **(filtre) = donnée.**

Cette « définition » est insuffisante, car non-déterministe en raison de la remarque ci-dessus. Pour préciser la solution choisie, le filtrage consiste à chercher, si elle existe, la « plus petite » mémoire **m** au sens de la seule relation d'ordre que la définition des mémoires nous inspire : l'inclusion ensembliste $\subset$ des graphes et telle que $\mathbf{h_m}$ **(filtre) = donnée.**

Ce choix est justifié par la remarque suivante :

Remarque : Si $\mathbf{m} \subset \mathbf{m'}$ et $\mathbf{h_m}$ **(filtre) = donnée,** alors $\mathbf{h_{m'}}$ **(filtre) = donnée.**

La plus petite solution, si elle existe, suffit donc à définir toutes les autres :

$$\{\mathbf{m} \mid \mathbf{h_m}(\mathrm{filtre}) = \mathrm{donn\acute{e}e}\} = \{\mathbf{m} \mid \mathbf{min} \subset \mathbf{m},\ \mathbf{h_{min}}(\mathrm{filtre}) = \mathrm{donn\acute{e}e}\}$$

Une intersection $\{\mathbf{m} \mid \mathrm{P}(\mathbf{m})\} \cap \{\mathbf{m'} \mid \mathrm{P'}(\mathbf{m'})\}$ de deux ensembles de solutions ainsi minorés respectivement par **min** et **min′** sera donc minorée par $\mathbf{min} \langle + \rangle \mathbf{min'}$.

Plus généralement, un ensemble de solutions est caractérisé par ses solutions minimales.

DÉFINITION : Le **filtrage**, ou semi-unification d'un filtre et d'une donnée est la fonction **match** :

$$\mathbf{match}\,(\text{filtre, donnée}) = \bigcap \{\mathbf{m} \mid \mathbf{h_m}\,(\text{filtre}) = \text{donnée}\}.$$

Un **filtre** peut donc déjà être **interprété** de deux façons opposées :

– comme un « format » à remplir avec les valeurs enregistrées dans une mémoire. L'interprète est alors $\mathbf{h_m}$;

– comme un « filtre » au sens habituel, cherchant **m** à partir de **donnée**.

3. TECHNIQUES DE CONSTRUCTION ET DE TRANSFORMATION

3.1. Programmation récursive par cas. Pliage, dépliage

DÉFINITION : Une définition récursive par cas de la fonction **f** est un système d'équations de la forme :

$$[\mathbf{1} \leqq \mathbf{i} \leqq \mathbf{p}], \qquad \mathbf{f}\,(\theta_i) = \rho_i\,(\theta_i) \quad \{\text{cas d'arrêt}\}$$

et

$$[\mathbf{p}+\mathbf{1} \leqq \mathbf{j} \leqq \mathbf{p}+\mathbf{q}], \qquad \mathbf{f}\,(\theta_j) = \varphi_j \circ \mathbf{f} \circ \psi_j\,(\theta_j) \quad \{\text{cas récursifs}\}$$

où les termes θ_i ou θ_j sont des schémas et définissent la forme des arguments qui correspondent à ce cas; nous supposons de plus que les schémas ne se chevauchent pas (*i. e.* : les ensembles de termes représentés sont disjoints). Par commodité, nous acceptons des définitions où un schéma θ_m peut être plus général qu'un autre, θ_n et cette ambiguïté est résolue en choisissant le schéma le moins général θ_n et en rejetant θ_m.

Dans les cas récursifs, φ_j et ψ_j peuvent contenir aussi des occurences de f, mais une seule occurence est suffisante pour distinguer les deux sortes de cas, et c'est elle qui est mise en évidence dans la composition fonctionnelle $\varphi_j \circ \mathbf{f} \circ \psi_j\,(\theta_j)$.

Cette forme de définition a été largement utilisée et popularisée pour la définition des fonctions exprimant la sémantique dénotationnelle des langages de programmation.

Plusieurs langages fonctionnels actuels, par exemple Hope [4], KRC [15] et Miranda à sa suite font appel à cette forme pour la définition des fonctions.

Le langage Hope a été défini par J. Darlington en vue de faciliter le développement de programmes par les opérations de **pliage, dépliage** et calculs algébriques (usage des égalités définies par le système et de celles de l'algèbre qui permet de l'interpréter).

Une étude des schémas de programmes exprimés sous cette forme et de leurs différences avec les schémas classiques a été faite par Courcelle et Lavandier [3]. La définition des fonctions ne fait plus intervenir de fonctions non strictes (conditionnelles), et la manipulation des définitions est rendue plus facile à justifier : les « cas » peuvent être considérés comme des fonctions partielles, et leur juxtaposition correspond à une opération entre ces fonctions partielles. Nous n'en dirons pas plus ici, notre but n'étant pas de développer la théorie de ces définitions mais seulement d'en exploiter quelques avantages pour résoudre le problème du filtrage.

DÉFINITION : Une équation est dite **post-récursive** si et seulement si elle est de la forme :

$$\mathbf{f}(\theta_j) = \mathbf{f} \circ \psi_j(\theta_j) \text{ où } \psi_j \text{ ne contient pas d'occurrences de } \mathbf{f};$$

Les équations de cette forme sont bien connues pour être en fait de simples boucles, ou itérations. Il est très facile de traduire un système de telles équations post-récursives en une boucle unique, voire même en langage d'assemblage. Nous les considérons comme un espace de solutions pour le développement de nos algorithmes, un langage cible donc, bien que nous donnions aussi ultimement le code LISP qui a été effectivement exécuté.

La transformation de Wand, comme nous le verrons, tend à produire des équations post-récursives.

3.2. Inversion

3.2.1. *Technique de Korf et Eppstein*

La technique d'inversion de fonctions semble un moyen naturel pour définir facilement des problèmes de recherche de termes **t** dont les propriétés sont exprimés sous la forme d'équations $\mathbf{f}(\mathbf{t}) = \mathbf{v}$, ou même $\mathbf{f}(\mathbf{t}, \mathbf{u}) = \mathbf{v}$, le concept d'inversion devenant alors un peu plus délicat à définir. Cette idée a été introduite par Korf [11] comme technique pour la synthèse automatique de programmes, et reprise par Eppstein [8] qui en donne une présentation plus systématique. Ces deux auteurs travaillent sur des définitions de fonctions Lisp, utilisant la conditionnelle généralisée COND et donnent des règles pour inverser les prédicats de sélection des clauses. Le cadre des définitions récursives par cas facilite grandement la définition de la technique d'inversion, car les divers arguments y apparaissent sous la forme de schémas explicites. Nous donnons ci-dessous une version de l'inversion adaptée à ce cadre, et l'inversion des prédicats de Eppstein n'y est plus nécessaire, mais l'idée principale et le découpage par clauses a été introduite par eux. On peut également trouver l'idée d'inversion dans l'ouvrage de D. Gries à propos de programmes impératifs, mais la technique y devient moins systématique et plus complexe.

L'équation définissant le problème est :

$$\mathbf{f}(\mathbf{t}, \mathbf{u}) = \mathbf{v},$$

et on cherche à trouver $\mathbf{t}$ en fonction de $\mathbf{u}$ et $\mathbf{v}$.

Cette situation est analogue à la recherche d'une preuve de $\forall \mathbf{u}.\forall \mathbf{v}.\ni \mathbf{t}.(\mathbf{f}(\mathbf{t}, \mathbf{u}) = \mathbf{v})$ en logique, et l'inversion s'apparente à l'introduction d'une « fonction de Skolem » $\sigma(\mathbf{u}, \mathbf{v})$, qu'on cherche ensuite à définir par cas à partir des cas de $\mathbf{f}$. Mais ici, il n'y a pas de quantifieurs universels et la fonction n'est pas nécessairement une fonction totale.

La technique employée, manuellement pour le moment, consiste à mettre en correspondance, à « unifier » l'équation

$$\mathbf{f}(\sigma(\mathbf{u}, \mathbf{v}), \mathbf{u}) = \mathbf{v}$$

avec les différents cas de la définition de $\mathbf{f}$:

$$[\mathbf{1} \leqq \mathbf{i} \leqq \mathbf{p}], \quad \mathbf{f}(\theta_i, \mathbf{u}_i) = \rho_i(\theta_i, \mathbf{u}_i) \quad \{\text{cas d'arrêt}\}$$
$$[\mathbf{p}+\mathbf{1} \leqq \mathbf{j} \leqq \mathbf{p}+\mathbf{q}], \quad \mathbf{f}(\theta_j, \mathbf{u}_j) = \varphi_j \circ \mathbf{f}(\psi_j(\theta_j), \mathbf{u}_j) \quad \{\text{cas récursifs}\}$$

On obtient ainsi des équations par cas définissant $\sigma(\mathbf{u}, \mathbf{v})$ pour les cas d'arrêt :

$$[\mathbf{1} \leqq \mathbf{i} \leqq \mathbf{p}] \quad \mathbf{f}(\theta_i, \mathbf{u}_i) = \rho_i(\theta_i, \mathbf{u}_i)$$

produisant par identification

$$[\mathbf{1} \leqq \mathbf{i} \leqq \mathbf{p}] \quad \sigma(\mathbf{u}_i, \rho_i(\theta_i, \mathbf{u}_i)) = \theta_i$$

Pour les cas récursifs il ne se dégage pas de règle générale évidente :

$$[\mathbf{p}+\mathbf{1} \leqq \mathbf{j} \leqq \mathbf{p}+\mathbf{q}] \quad \mathbf{f}(\theta_j, \mathbf{u}_j) = \varphi_j \circ \mathbf{f}(\psi_j(\theta_j), \mathbf{u}_j)$$

conduit aussi à identifier

$$\sigma(\mathbf{u_j}, \varphi_j \circ \mathbf{f}(\psi_j(\theta_j), \mathbf{u_j})) = \theta_j,$$

mais, en général, la forme du second argument n'est pas celle d'un schéma de termes.

Cependant, et ce sera le cas pour notre problème, lorsque le contexte φ_j se réduit à un opérateur de la signature composant directement des occurences de **f** appliquées directement à des sous-termes de θ_j, le découpage est possible. Une étude générale des systèmes pour lesquels ce découpage est possible reste à faire, l'exemple développé ci-dessous suffit à en montrer l'intérêt.

Notons aussi que l'inversion d'une fonction ne produit pas nécessairement une fonction. Notre espace de travail est en fait constitué de fonctions partielles et multivoques, de **relations**. Ce point peut être résolu par la prise en compte de l'ensemble des parties. Les opérations que nous serons conduits à introduire auront pour but de simuler ce passage à l'ensemble des parties.

3.2.2. *Un algorithme trivial de filtrage*

Cet algorithme traduit immédiatement la définition, et son principal mérite est d'être évidemment correct. Il est tout aussi évidemment inefficace et limité, et le reste de ce travail consiste à l'améliorer. Il s'exprime sous la forme de clauses disjointes, définissant une fonction partielle : **match**.

La fonction **match** : **Filtres x Données** $\rightarrow$ **Mémoires** vérifie

$$\mathbf{match}\,(\text{filtre, donnée}) = \mathbf{m} \text{ si et seulement si } \mathbf{h_m}(\text{filtre}) = \text{donnée}$$

Rappelons les propriétés du morphisme $\mathbf{h_m}$: **Filtres** $\rightarrow$ **Données**

cas d'arrêt :

$$\text{nil} \in \mathbf{Filtres},\ \mathbf{h_m}(\text{nil}) = \text{nil} \in \mathbf{Données}$$

$$\forall x \in \mathbf{Symb},\ \mathbf{h_m}(''x) = x$$

$$\forall x \in \mathbf{Symb},\ \mathbf{h_m}(:x) = \mathbf{d} \text{ ssi } \mathbf{d} = \mathbf{m}(x)$$

cas récursifs :

$$\forall t_1,\ t_2 \in \mathbf{Filtres},\ \mathbf{h_m}(t_1 . t_2) = (\mathbf{h_m}(t_1) . \mathbf{h_m}(t_2)) \in \mathbf{Données}$$

Il reste à exprimer **match** cas par cas, en inversant ces égalités; On y suppose la variable **m** quantifiée universellement.

De nil $\in$ **Filtres**, $\mathbf{h_m}$(nil) = nil $\in$ **Données** on déduit :

$$\mathbf{match}(\text{nil, nil}) = \bigcap \{m \mid \mathbf{h_m}(\text{nil}) = \text{nil}\} = \{\ \}$$

De $\forall x \in \mathbf{Symb}$, $\mathbf{h_m}(''x)=x$ on déduit :

$$\mathbf{match}(''x,\ x)=\bigcap\{m \mid \mathbf{h_m}(''x)=x\}=\{\ \}$$

De $\forall x \in \mathbf{Symb}$, $\mathbf{h_m}(:x)=\mathbf{d}$ ssi $\mathbf{d}=\mathbf{m}(x)$ on déduit :

$$\mathbf{match}(:x,\ \mathbf{d})=\bigcap\{m \mid \mathbf{h_m}(:x)=\mathbf{d}\}=\{x \to \mathbf{d}\}$$

De $\forall t_1,\ t_2 \in \mathbf{Filtres}$, $\mathbf{h_m}(t_1 . t_2)=(\mathbf{h_m}(t_1) . \mathbf{h_m}(t_2)) \in \mathbf{Données}$ on déduit :

$$\begin{aligned}\mathbf{match}((t_1 . t_2),\ (d_1 . d_2)) &= \bigcap\{\mathbf{m} \mid \mathbf{h_m}(t_1)=d_1 \ \mathbf{et}\ \mathbf{h_m}(t_2)=d_2\}\\ &= \bigcap(\{\mathbf{m} \mid \mathbf{h_m}(t_1)=d_1\} \cap \{\mathbf{m} \mid \mathbf{h_m}(t_2)=d_2\})\\ &= (\bigcap\{m \mid \mathbf{h_m}(t_1)=d_1\}) \cup (\bigcap\{m \mid \mathbf{h_m}(t_2)=d_2\})\end{aligned}$$

si et seulement si cette union est une mémoire.

$$=\mathbf{match}(t_1,\ d_1)\langle + \rangle \mathbf{match}(t_2 . d_2)$$

Conclusion : En éliminant les preuves des clauses, on obtient :

ALGORITHME 1 :

$$\mathbf{match}(\text{nil},\ \text{nil})=\{\ \}$$
$$\mathbf{match}(''x,\ x)=\{\ \}$$
$$\mathbf{match}(:x,\ \mathbf{d})=\{x->\mathbf{d}\}$$
$$\mathbf{match}((t_1 . t_2),\ (d_1 . d_2))=\mathbf{match}(t_1,\ d_1)\langle + \rangle \mathbf{match}(t_2 . d_2)$$

La fonction **match** n'est pas définie dans les autres cas. C'est donc la plus petite fonction définie par les clauses ci-dessus.

Il est clair que ce système de clauses s'écrit presque sans modifications dans tout bon langage fonctionnel et même en LISP. Mais l'usage de l'opération $\langle + \rangle$ est coûteux puisque il entraîne une récursion non terminale et une recherche d'intersection d'ensembles. Il faut donc essayer de l'éliminer ou de le mettre sous une forme moins coûteuse.

Remarquons aussi que la fonction **match** constitue un *nouvel interprète* pour le langage **Filtres**. A tout filtre **filtre**, elle associe une fonction, ou mieux, un algorithme de **données** $\to$ **mémoires**. Cette vision des choses n'est rien d'autre que la transformation appelée **curryfication** par les spécialistes des Sémantiques et Modèles du λ-calcul. D'un point de vue de Praticien, cela conduit très naturellement à voir les filtres comme des programmes, à penser en termes de **continuation**, ou contexte d'évaluation, ce que nous développons dans la suite, voire de **compilation** des filtres, ce que nous ne traiterons pas.

3.2.3. *Généralisation à une Signature arbitraire*

Cette construction s'étend aisément à une signature arbitraire. Si Σ-Algèbre est le nom d'une signature à une sorte $\langle \{\mathrm{T}\}, \Sigma, \mathbf{profil} \rangle$, on définit $\langle\langle \Sigma\text{-Algèbre} \rangle\rangle (\mathbf{Symb})$ comme l'algèbre des données et $\langle\langle \Sigma\text{-Algèbre} \rangle\rangle (\mathbf{Const}+\mathbf{Var})$ comme l'algèbre des filtres.

En l'absence d'équations propre aux opérateurs de ces algèbres, le schéma d'inversion s'applique sans problème :

Le morphisme de substitution se généralise à tout opérateur ω de Σ, d'arité n :

$$\forall t_1, \ldots, t_n \in \mathbf{Filtres},\ \mathbf{h_m}(\omega(t_1 \ldots t_n)) = \omega(\mathbf{h_m}(t_1) \ldots \mathbf{h_m}(t_n)) \in \mathbf{Données}$$

on en déduit :

$$\begin{aligned}
\mathbf{match}(\omega(t_1 \ldots t_n), \omega(d_1 \ldots d_n)) &= \bigcap \{\mathbf{m} \mid \mathbf{h_m}(t_1) = d_1 \textbf{ et} \ldots \mathbf{h_m}(t_n) = d_n\} \\
&= \bigcap (\{\mathbf{m} \mid \mathbf{h_m}(t_1) = d_1\} \cap \ldots \{\mathbf{m} \mid \mathbf{h_m}(t_n) = d_n\}) \\
&= (\bigcap \{\mathbf{m} \mid \mathbf{h_m}(t_1) = d_1\}) \cup \ldots (\bigcap \{\mathbf{m} \mid \mathbf{h_m}(t_n) = d_n\})
\end{aligned}$$

si et seulement si cette union est une mémoire.

$$= \mathbf{match}(t_1, d_1) \langle + \rangle \ldots \langle + \rangle \mathbf{match}(t_n . d_n)$$

Le problème n'est donc pas plus complexe, mais l'écriture est plus lourde. Nous ne dirons donc rien de plus sur cette généralisation. Le cas d'une algèbre avec équations est beaucoup plus délicat, nous donnerons quelques idées sur ce point dans la partie 4. Mais la complexité du seul cas de l'opérateur binaire associatif de concaténation laisse peu d'espoir pour une généralisation systématique.

3.3. La transformation de Wand

3.3.1. *Continuation et capture de continuation*

Son schéma général est en 2 étapes :

1. **Expliciter la continuation**, le contexte au sens littéral du terme.

2. **Transformer et capturer** cette continuation, comme argument supplémentaire.

Cette prise en compte explicite du contexte d'évaluation, ou **continuation** du calcul courant, est fondamentale dans les transformations qui vont suivre. C'est une **méthode de transformation** pratiquée plus ou moins explicitement depuis une dizaine d'années par quelques LISPiens endurcis. La première publication officielle à son sujet est celle de M. Wand [16], c'est pourquoi nous l'appellerons ici **Transformation de Wand.** Mais le premier informaticien qui a utilisé une pile pour réaliser un appel récursif l'a déjà appliquée implicitement.

Cette technique s'inspire du concept de continuation de la sémantique dénotationnelle : la suite du calcul en cours est considérée comme l'application d'une fonction au résultat intermédiaire courant; cette fonction est appelée continuation.

Le concept de continuation a d'abord été utilisé pour définir la sémantique des instructions de saut ou de sortie de boucle ou de procedure dans des langages impératifs : il était nécessaire de faire apparaître explicitement la suite du calcul en cours pour pouvoir exprimer la possibilité de la changer. Ce concept s'étend aux expressions de type fonctionnel, la suite du calcul en cours s'identifie alors au contexte de la sous-expression actuellement à réduire; ce contexte peut être lui-même déjà partiellement réduit, en fonction de la stratégie d'évaluation courante.

Le paradigme sous-jacent est de voir les données du programme comme un langage et le programme comme un interprète de ce langage. Les concepts de la sémantique s'appliquent alors très naturellement, et celui de continuation est adapté à résoudre les problèmes arborescents qui échappent à un parcours simple.

L'article de M. Wand présente cette technique sous la forme d'une suite d'exemples incluant la recherche α-β, et l'évaluation d'expressions booléennes. Il ne donne pas de définition formelle et générale de son schéma de transformation, mais une forme adaptée à chaque exemple traité.

Une forme plus générale peut être induite des exemples de [16] en exploitant la notion de système de définitions récursives par cas, les exemples étant d'ailleurs définis sous cette forme. Nous allons donc exprimer la transformation de Wand sous la forme de réécritures syntaxiques sur les systèmes de définitions récursives par cas. L'idée elle-même ne dépend pas de cette forme de définition des fonctions, la forme Lisp pourrait aussi convenir.

3.3.2. *Règles de la transformation de Wand pour les systèmes récursifs par cas*

Une fonction $\mathbf{f} : \mathbf{X} \to \mathbf{Y}$ est définie par cas :

$$[\mathbf{1} \leqq \mathbf{i} \leqq \mathbf{p}] \quad \mathbf{f}(\theta_i) = \rho_i(\theta_i) \ \{\text{arrêt; } \rho_i : \mathbf{X} \to \mathbf{Y}\}$$

$$[\mathbf{p}+\mathbf{1} \leqq \mathbf{j} \leqq \mathbf{p}+\mathbf{q}] \quad \mathbf{f}(\theta_j) = \varphi_j \circ \mathbf{f} \circ \psi_j(\theta_j) \ \{\text{boucles; } \psi_j : \mathbf{X} \to \mathbf{X},\ \varphi_j : \mathbf{Y} \to \mathbf{Y}\}$$

Un système de définitions par cas de la même fonction est obtenu par application des règles suivantes :

Introduction d'une fonction auxiliaire

$$\mathbf{f} : \mathbf{X} \to \mathbf{Y}$$

$$\mathbf{F} : (\mathbf{X}_{\mathbf{x}}(\mathbf{Y} \to \mathbf{Y})) \to \mathbf{Y},$$
$$\mathbf{f}(\theta) = \mathbf{F}(\theta, \mathbf{id})$$
où **id** est la fonction identité sur **Y**

Introduction des cas d'arrêt :

$$\mathbf{f}(\theta_i) = \rho_i(\theta_i) \ \{\text{avec } \rho_i : \mathbf{X} \to \mathbf{Y};\ \gamma : \mathbf{Y} \to \mathbf{Y}\}$$

$$\mathbf{F}(\theta_i, \gamma) = \gamma \circ \rho_i(\theta_i) \ \{\text{avec } \rho_i : \mathbf{X} \to \mathbf{Y};\ \gamma : \mathbf{Y} \to \mathbf{Y}\}$$

Introduction des cas récursifs :

$$\mathbf{f}(\theta_j) = \varphi_j \circ \mathbf{f} \circ \psi_j(\theta_j) \ \{\text{avec } \psi_j : \mathbf{X} \to \mathbf{X},\ \varphi_j : \mathbf{Y} \to \mathbf{Y}\}$$

$$\mathbf{F}(\theta_j, \gamma) = \mathbf{F}(\psi_j(\theta_j), \gamma \circ \varphi_j) \ \{\text{avec } \psi_j : \mathbf{X} \to \mathbf{X},\ \gamma \circ \varphi_j : \mathbf{Y} \to \mathbf{Y}\}$$

Dans les règles ci-dessus, la variable γ dénote toute fonction de $\mathbf{Y} \to \mathbf{Y}$, et constitue le « **paramètre de continuation** » de **F**. Le cadre de travail idéal est donc celui d'un langage fonctionnel vrai, avec des arguments et résultats fonctionnels au contraire de Lisp [4, 5, 14, 15, 16]. Cependant, dans le développement qui va suivre nous éliminerons cette contrainte par le choix des arguments et l'introduction éventuelle de fonctions auxiliaires pour simuler l'application des continuations.

La validité de ces règles repose sur le fait qu'elles ne sont qu'une simple paraphrase des règles de composition des fonctions à plusieurs arguments en termes de composition de fonctions monadiques, via l'opérateur de composition fonctionnelle « $\circ$ ». Ceci suppose implicitement que les symboles fonctionnels employés désignent des fonctions au sens mathématique usuel, les fonctions partielles et multivoques étant interprétées comme des fonctions sur les ensembles des parties.

Une étude plus fine de la validité de ce type de règles en fonction des interprétations possibles des systèmes d'équations récursives par cas reste à faire, mais c'est un tout autre travail que le développement qui nous intéresse ici.

Les schémas donnés ci-dessus s'appliquent en principe à des fonctions **f** à un seul argument de domaine **X**, mais rien n'empêche ce domaine **X** d'être un produit cartésien, tout comme **Y** aussi. Nous utiliserons largement cette possibilité dans la suite.

3.4. Améliorations de l'algorithme 1

Observons à nouveau le texte de l'algorithme obtenu :

ALGORITHME 1 :

$$\mathbf{match}\,(\text{nil, nil}) = \{\quad\}$$

$$\mathbf{match}\,({''x},\ x) = \{\quad\}$$

$$\mathbf{match}\,(:x,\ \mathbf{d}) = \{x \to \mathbf{d}\}$$

$$\mathbf{match}\,((t_1 . t_2), (d_1 . d_2)) = \mathbf{match}\,(t_1,\ d_1) \langle + \rangle\, \mathbf{match}\,(t_2 . d_2)$$

On constate que l'algorithme doit être appliqué récursivement dans le contexte d'une réunion $\langle + \rangle$ de mémoires :

$$\mathbf{match}\,(t_1,\ d_1) \langle + \rangle \ldots \qquad \text{ou bien} \quad \mathbf{m} \langle + \rangle\, \mathbf{match}\,(t_2 . d_2)$$

On a donc à envisager le cas général : $\mathbf{match}\,(t,\ d) \langle + \rangle\, \mathbf{m}$

3.4.1. *Expliciter la continuation*

Remplaçant match par sa définition, on a :

$$\mathbf{match}\,(\text{nil, nil}) \langle + \rangle\, \mathbf{m} = \{\quad\} \langle + \rangle\, \mathbf{m}$$

$$\mathbf{match}\,({''x},\ x) \langle + \rangle\, \mathbf{m} = \{\quad\} \langle + \rangle\, \mathbf{m}$$

$$\mathbf{match}\,(:x,\ \mathbf{d}) \langle + \rangle\, \mathbf{m} = \{x \to \mathbf{d}\} \langle + \rangle\, \mathbf{m}$$

$$\mathbf{match}\,((t_1 . t_2),\ (d_1 . d_2)) \langle + \rangle\, \mathbf{m} = \mathbf{match}\,(t_1,\ d_1) \langle + \rangle\, \mathbf{match}\,(t_2 . d_2) \langle + \rangle\, \mathbf{m}.$$

3.4.2. *Transformer et capturer*

L'opération $\langle + \rangle$, étant identique à l'union ensembliste sur son domaine de définition, y vérifie les propriétés de cette union. Donc :

$$\mathbf{match}(\text{nil}, \text{nil}) \langle + \rangle \mathbf{m} = \mathbf{m}$$

$$\mathbf{match}(''x, x) \langle + \rangle \mathbf{m} = \mathbf{m}$$

$$\mathbf{match}(: x, \mathbf{d}) \langle + \rangle \mathbf{m} = \{ x \to \mathbf{d} \} \langle + \rangle \mathbf{m}$$

$$\mathbf{match}((t_1 . t_2), (d_1 . d_2)) \langle + \rangle \mathbf{m} = \mathbf{match}(t_1, d_1) \langle + \rangle (\mathbf{match}(t_2 . d_2) \langle + \rangle \mathbf{m}).$$

La transformation de Wand s'applique ici sous une forme simplifiée : c'est un simple « changement de variable », sur le nom et l'arité de la fonction, permettant de capturer la continuation.

Posons en effet :

$$\mathbf{match\,2}(t, d, \mathbf{m}) = \mathbf{match}(t, d) \langle + \rangle \mathbf{m}$$

Cette fonction **match 2** correspond à la fonction **F** de Wand, à la légère différence que l'argument fonctionnel de continuation $\gamma = \lambda \mathbf{x} . \mathbf{x} \langle + \rangle \mathbf{m}$ est remplacé simplement par **m** qui suffit au calcul. Dans la suite, nous utiliserons couramment cette variante qui allège les notations en éliminant les $\lambda \mathbf{x} . \mathbf{x}$ inutiles.

On obtient alors l'algorithme semi-postrécursif :

ALGORITHME 2 :

$$\mathbf{match\,2}(\text{nil}, \text{nil}, \mathbf{m}) = \mathbf{m}$$

$$\mathbf{match\,2}(''x, x, \mathbf{m}) = \mathbf{m}$$

$$\mathbf{match\,2}(: x, \mathbf{d}, \mathbf{m}) = \{ x \to \mathbf{d} \} \langle + \rangle \mathbf{m}$$

$$\mathbf{match\,2}((t_1 . t_2), (d_1 . d_2), \mathbf{m}) = \mathbf{match\,2}(t_1, d_1, \mathbf{match\,2}(t_2, d_2, \mathbf{m})).$$

match 2 est évidemment équivalent à **match**, mais plus efficace puisque l'appel le plus externe est post-récursif et peut être interprété ou compilé de manière itérative. De plus, l'opération $\langle + \rangle$ a pratiquement disparu : elle est localisée dans la troisième clause, et son opérande gauche est réduit à **une paire symbole → valeur**.

Remarque : Du fait du mécanisme d'appel par valeur usuel dans de nombreux langages actuels et en particulier en Lisp, l'argument **match 2** $(t_2, d_2, \mathbf{m})$ est ainsi évalué en premier.

On a aussi, par commutativité, l'égalité :

$$\mathbf{match}((t_1 . t_2), (d_1 . d_2)) \langle + \rangle \mathbf{m} = \mathbf{match}(t_2 . d_2) \langle + \rangle (\mathbf{match}(t_1, d_1) \langle + \rangle \mathbf{m})$$

qui donne la recherche « depth-first », plus habituelle :

$$\mathbf{match2}((t_1 . t_2), (d_1 . d_2)) \langle + \rangle \mathbf{m} = \mathbf{match2}(t_2, d_2, \mathbf{match2}(t_1, d_1, \mathbf{m})).$$

C'est l'algorithme classique de filtrage du premier ordre. L'argument **m** étant évalué, le calcul des ajouts élémentaires $\{x \rightarrow \mathbf{d}\} \langle + \rangle \mathbf{m}$ peut se faire effectivement à chaque étape.

3.4.3. *Autres stratégies de transformation*

Ainsi, **l'argument accumulateur**, ou « **sac** », technique classique [2] pour éliminer les appels récursifs enveloppés dans des expressions à opérateur associatif, est **une expression compacte d'une continuation capturée** [16]. Bien sûr, c'est de la routine, et on sait faire de même avec des pliages et dépliages, au prix de l'introduction d'astuces appelées « **euréka** » [4]. Mais la transformation de Wand a le mérite de la généralité et de la simplicité : elle aide à inventer des **euréka** au même titre que d'autres approches [2, 12].

Après cette présentation des définitions et des techniques de base, nous allons maintenant les appliquer au développement d'un exemple moins trivial : un algorithme de semi-unification de listes arborescentes, les filtres contenant éventuellement des variables dites « **de segment** » dans le jargon habituel des pratiquants de cette activité, surtout dans le cadre de l'intelligence artificielle et des langages « à règles de production » [5, 6, 13].

Cet exemple présente un double intérêt :

— c'est un problème directement issu des techniques de l'intelligence artificielle et du calcul formel. Contrairement à celui de l'unification, il n'a pas fait l'objet de recherches théoriques mais seulement d'une intense activité de programmation, chacun introduisant ses propres trucs pour résoudre la principale difficulté : la reprise de la recherche après un succès apparent.

— il nous permet de mettre en œuvre une technique de développement par transformations des définitions, qui n'est pas encore au stade d'automatisation ou de rigueur souhaitable, mais qui laisse espérer d'autres succès dans la suite.

4. FILTRAGE ASSOCIATIF, OU DU SECOND ORDRE

Le nom « filtrage du second ordre » est issu de la terminologie de G. Huet. En effet, c'est dans des unifications ou semi-unifications du second ordre qu'on est amené à instancier les variables fonctionnelles à des compositions arbitrairement longues de fonctions primitives. Le cas des filtrages de listes est évidemment analogue à celui de fonctions monadiques, d'où son nom.

En effet, une variable fonctionnelle **F** peut alors prendre pour valeur une composition arbitraire de fonctions :

$$\mathbf{F}=\mathbf{f}_1 \circ \mathbf{f}_2 \circ \ldots \circ \mathbf{f}_n.$$

Ce problème revient donc à celui du filtrage associatif avec élément neutre. Dans le cas de l'analyse d'expressions de la logique combinatoire, on recherche des combinateurs de longueur quelconque, qui sont évidemment des termes fonctionnels.

Bien sûr, ce problème est plus simple que celui de l'unification du second ordre en λ-calcul, traité par Huet, mais nous nous proposons d'en faire un développement complet depuis la définition jusqu'au programme exécutable en explicitant les étapes sous forme de transformations.

D'autre part, l'algorithme de G. Huet utilise des notions telles que les ensembles de désagrément (« disagreement set »), qui reposent sur la réalisation effective du type abstrait ensemble, et développe son espace d'états. Notre construction cherche au contraire à obtenir directement une version exécutable exprimée à l'aide de structures de données courantes dans les langages de programmation usuels, ou réalisables sans difficulté particulière. De plus, nous cherchons à détecter le plus rapidement possible les échecs, et à pénaliser au minimum le traitement des cas simples. La structure de notre construction est donc très différente de celle de G. Huet, et ne s'en inspire pas.

4.1. Listes, concaténation, et variables de segments

DÉFINITION : On appelle **segment** toute *s*-expression qui peut s'exprimer par les deux règles :

$$\vdash \mathbf{segment}(\mathrm{nil})$$

$$\mathbf{S\text{-}expr}(t),\ \mathbf{segment}(S) \vdash \mathbf{segment}((t\,.\,S))$$

Note : Nous préférons ici « segment » à « liste » par référence aux « variables de segment » qui correspondent à une partie de liste.

L'usage a fait cette distinction pour des raisons pragmatiques. Dans (**abcd**) par exemple, la partie ...**bc**... n'est pas physiquement identique à la liste (**bc**), le dernier pointeur étant sur (**d**) dans le premier cas et sur **nil** dans le second. Du point de vue algébrique, il n'y a pas de différence, car nous ne faisons pas intervenir la représentation physique : les « segments » sont définis comme habituellement les listes, et ont donc les mêmes propriétés. Le point de vue pragmatique conduit cependant à différencier la reconnaissance de la partie ...**bc**... dans (**abcd**) de celle de (**bc**) dans (**a**(**bc**)**d**) et le remplacement systématique de « segment » par « liste » dans la suite conduirait à des ambiguïtés et des difficultés d'expression qu'il vaut mieux éviter.

L'introduction des segments peut être vue comme une *extension* de l'algèbre des S-expressions, mais associée à une inclusion de la sorte **segment** dans la sorte **S-expr**.

L'opération naturelle sur les segments est la **concaténation**, que nous noterons @ (append); il est préférable de la définir plus généralement sur les S-expressions.

DÉFINITION :

$$@ : \mathbf{segment} \times \mathbf{S\text{-}expr} \rightarrow \mathbf{S\text{-}expr}$$

$$\mathbf{nil} \mathbin{@} s = s$$

$$(t\,.\,s) \mathbin{@} s' = (t\,.\,s \mathbin{@} s')$$

Remarque 1 : Si **segment** (s) alors l'opération @ peut être éliminée de toute expression de la forme $s \mathbin{@} s'$, ce qui s'écrit sous forme compacte :

$$\mathbf{segment}\,(s),\ \mathbf{S\text{-}expr}\,(s') \vdash \mathbf{S\text{-}expr}\,(s \mathbin{@} s')$$

La définition de @ n'introduit donc pas de termes irréductibles nouveaux dans les S-algèbres.

Remarque 2 : On peut démontrer de manière similaire que la restriction de @ aux segments est une **loi de composition interne** sur les segments :

$$\mathbf{segment}\,(s),\ \mathbf{segment}\,(s') \vdash \mathbf{segment}\,(s \mathbin{@} s')$$

Nous noterons par le même symbole @ la restriction de @ aux segments :

$$@ : \text{segment} \times \text{segment} \to \text{segment}$$

$$\text{nil} @ s = s$$

$$(t . s) @ s' = (t . s @ s')$$

Remarque 3 : Il est bien connu que la loi de composition @ ainsi définie est associative et admet nil pour élément neutre à droite. Les preuves formelles à partir des définitions ci-dessus sont sans difficulté. En se restreignant aux segments, on a :

$$@ : \text{segment} \times \text{segment} \to \text{segment}$$

$$\text{nil} @ s = s$$

$$s @ \text{nil} = s$$

$$(s @ s) @ s'' = s @ (s' @ s'')$$

Il reste maintenant à étendre le morphisme de substitution à l'algèbre ainsi étendue.

Par chance cette extension est possible sous forme de morphisme.

PROPOSITION :

$$\mathbf{h_m}((f_1 @ f_2)) = \mathbf{h_m}(f_1) @ \mathbf{h_m}(f_2)$$

Preuve : Une induction immédiate.

$$\mathbf{h_m}((\text{nil} @ s)) \underset{[\text{équation}]}{=} \mathbf{h_m}(s) \underset{[\text{équation}]}{=} \text{nil} @ \mathbf{h_m}(s) \underset{[\text{morphisme}]}{=} \mathbf{h_m}(\text{nil}) @ \mathbf{h_m}(s)$$

$$\begin{aligned}\mathbf{h_m}((t.s) @ s')) &= \mathbf{h_m}((t.(s @ s')) \quad [\text{équation}]\\ &= (\mathbf{h_m}(t).\mathbf{h_m}(s @ s')) \quad [\text{morphisme}]\\ &= (\mathbf{h_m}(t).(\mathbf{h_m}(s) @ \mathbf{h_m}(s')) \quad [\text{induction}]\\ &= (\mathbf{h_m}(t).\mathbf{h_m}(s)) @ \mathbf{h_m}(s') \quad [\text{équation}]\\ &= \mathbf{h_m}((t.s)) @ \mathbf{h_m}(s') \quad [\text{morphisme}].\end{aligned}$$

4.2. Variables et filtrage de segments

DÉFINITIONS :

- On adjoint aux opérateurs classes de filtres « " » et « : » le nouvel opérateur classe de filtres « ! ».
- Ce nouvel opérateur étend le domaine des filtres atomiques en :

$$\textbf{Const} \cup \textbf{Var} \cup \textbf{Segm}, \text{ où } \textbf{Segm} = \text{« ! »} \textbf{Symb})$$

- Cette extension s'étend aux Filtres en produisant :

$$\textbf{Filtres}' = \text{« } S.\text{algèbre »} (\textbf{Const} \cup \textbf{Var} \cup \textbf{Segm})$$

- Le morphisme de **substitution** associé à une mémoire **m** s'étend de **Filtres** à **Filtres**′ en posant la clause supplémentaire :

$$\mathbf{h_m}((!x.f)) = \mathbf{m}(x) @ \mathbf{h_m}(f)$$

Mais cette extension est partielle : Une variable de segment isolée au plus haut niveau n'a en principe pas de sens. Une variable de segment en position « cdr » n'a en principe pas de sens non plus. On peut donner à cette objection deux réponses :

Réponse 1 : **Les variables de Segm ne sont qu'une abréviation permettant d'éliminer les opérateurs @. Il en résulte immédiatement que les variables de Segm ne peuvent apparaître qu'en position car.**

Cette extension naturelle de $\mathbf{h_m}$ aux segments permet d'interpréter les variables de segments en posant :

$$(!x.t) =: x @ t$$

Ce qui est une vision bien naturelle d'une variable de segment : une variable désignant un objet dans le monoïde des segments. On a alors en particulier :

$$(! x . \text{nil}) = : x \,@\, \text{nil} = : x$$

i. e. : une **variable** en position **cdr** est identique à un **segment** en position **dernier car**.

Réponse 2 : **Prolongement par analogie :** $\mathbf{h_m}(! x) = \mathbf{d}$ ssi $\mathbf{m}(x) = (\mathbf{d} . \text{nil})$.

Avec cette convention, on constate que $\mathbf{h_m}((f_1 . f_2)) = \mathbf{h_m}((f_1 . \text{nil}) \,@\, f_2)$ si f_1 n'est pas de la forme $! x$. Cette règle ne doit surtout pas être appliquée effectivement, car on aurait alors :

$$\mathbf{h_m}((f . \text{nil})) = \mathbf{h_m}((f . \text{nil}) \,@\, \text{nil}) = \mathbf{h_m}((f . \text{nil})) \ldots$$

Mais cela veut dire qu'on a un **prolongement** $S \leftrightarrow (S . \text{nil}) = (S)$ des **S-expr en position car** dans les **segments**, à l'exception des **variables de segment**, $\mathbf{h_m}((! x . \mathbf{s}) = \mathbf{m}(x) \,@\, \mathbf{s}$ pour lesquelles $\mathbf{h_m}$ enlève l'écorce $(S . \text{nil}) = (S)$.

Cette extension ne pose pas de problèmes avec les filtres bien construits par la règle de **@-élimination** puisque les cas rajoutés ne peuvent pas y apparaître. D'autre part, elle semble assez cohérente par passage aux contextes : $\mathbf{h_m}(! x . ! x) = \mathbf{m}(x) \,@\, \mathbf{h_m}(! x)$.

On doit alors avoir $\mathbf{m}(x) = (\mathbf{d} . \text{nil})$ et il en résulte

$$\begin{aligned}\mathbf{h_m}(! x . ! x) &= \mathbf{m}(x) \,@\, \mathbf{h_m}(! x)\\ &= (\mathbf{d} . \text{nil}) \,@\, \mathbf{d}\\ &= (\mathbf{d} . (\text{nil} \,@\, \mathbf{d})) = (\mathbf{d} . \mathbf{d})\end{aligned}$$

D'autre part, on a :

$$\begin{aligned}\mathbf{h_m}(: x . ! x) &= (\mathbf{m}(x) . \text{nil}) \,@\, \mathbf{h_m}(! x)\\ &= (\mathbf{m}(x) . \mathbf{h_m}(! x))\\ &= ((\mathbf{d} . \text{nil}) . \mathbf{d}) = ((\mathbf{d}) . \mathbf{d})\end{aligned}$$

L'objection de définition partielle subsiste, puisque la mémoire **m** doit vérifier $\mathbf{m}(x) = (\mathbf{d})$ pour tout filtre élémentaire **Segm** en position **cdr** ou **isolée**. Mais cette extension par analogie peut être intéressante.

4.3. L'opérateur de Choix Non-Déterministe : « | »

L'introduction de @ est un enrichissement des S-algèbres, augmentant leur signature. Nous avons vu comment étendre le morphisme de substitution et donc son inverse. Cependant, l'opérateur @ vérifie des équations. Le morphisme de substitution se généralisant à tout opérateur ω de Σ, d'arité n, l'ensemble des solutions vérifie :

$$\{\mathbf{m} \mid \mathbf{h_m}(\omega(t_1 \ldots t_n)) = \omega(d_1 \ldots d_n)\} \\ = (\{\mathbf{m} \mid \mathbf{h_m}(t_1) = d_1\} \cap \ldots \{\mathbf{m} \mid \mathbf{h_m}(t_n) = d_n\})$$

En présence d'une équation $\omega(t_1 \ldots t_n) = \Omega(t'_1 \ldots t'_p)$ et de l'équation correspondante sur les données $\omega(d_1 \ldots d_n) = \Omega(d'_1 \ldots d'_p)$, la technique d'inversion conduit à trouver cette fois pour ensemble de solutions :

$$\{\mathbf{m} \mid \mathbf{h_m}(\omega(t_1 \ldots t_n)) = \omega(d_1 \ldots d_n)\} = \{\mathbf{m} \mid \mathbf{h_m}(\Omega(t'_1 \ldots t'_p)) = \Omega(d'_1 \ldots d'_p)\} \\ = (\{\mathbf{m} \mid \mathbf{h_m}(t_1) = d_1\} \cap \ldots \{\mathbf{m} \mid \mathbf{h_m}(t_n) = d_n\}) \\ \cup (\{\mathbf{m} \mid \mathbf{h_m}(t'_1) = d'_1\} \cap \ldots \{\mathbf{m} \mid \mathbf{h_m}(t'_p) = d'_p\}) \cup \ldots,$$

les diverses possibilités résultant des diverses équations dont les opérateurs sont compatibles.

Les opérations sur ces ensembles de solutions sont les opérations usuelles sur les ensembles, et vérifient donc les propriétés habituelles : associativité, commutativité, idempotence, distributivité. Un ensemble de solutions étant caractérisé par ses éléments minimaux, nous avons vu que l'opération $\cap$ se reflète dans l'opération $\langle + \rangle$, qui est l'union des éléments minimaux si cette union est une mémoire. Le cas de l'intersection vide correspond à celui où l'union des représentations n'est pas une mémoire. Le passage de l'ensemble aux éléments minimaux se traduit par la transformation de l'intersection en $\langle + \rangle$, l'ensemble vide de solutions n'a pas d'élément minimal et n'est donc pas représenté par une mémoire. L'opération $\cup$ sur les ensembles de solutions sera représentée par l'opérateur | sur les mémoires, puisqu'il ne doit pas être confondu avec l'union de mémoires, $\langle + \rangle$. Il y a une différence d'un niveau du point de vue de l'interprétation ensembliste. Une autre raison de l'introduction de cet opérateur est de garder une plus grande liberté pour interpréter les symboles et éviter les connotations liées à leurs usages courants.

PROPRIÉTÉS DE $\langle + \rangle$ ET $|$

$\langle + \rangle$ est associatif, commutatif, idempotent, et admet la mémoire vide pour élément neutre
$|$ est associatif, commutatif, idempotent.

Distributivité :

$$(\alpha | \beta) \langle + \rangle \gamma = (\alpha \langle + \rangle \gamma) | (\beta \langle + \rangle \gamma)$$

Ces propriétés de $\langle + \rangle$ et $|$ se déduisent aisément de celles de l'intersection et de l'union d'ensembles de solutions, qu'elles traduisent sur les mémoires. La distributivité a un effet de recopiage qui permet le retour-arrière (backtracking).

INTERPRÉTATION OPÉRATIONNELLE DE $|$: En l'absence de toute nouvelle règle, nous pourrions obtenir ainsi une expression de l'ensemble des solutions possibles. Cependant, nous ne cherchons à trouver qu'une solution, et la première peut convenir. Dans ce but, l'opérateur de choix peut s'interpréter opérationnellement comme :

$$\mathbf{m} | \mathbf{m}' = \mathbf{m} \text{ si } \mathbf{m} \text{ est définie, } \mathbf{m}' \text{ sinon.}$$

Cette interprétation opérationnelle joue le rôle du **cut** de Prolog, et permet d'arrêter le calcul à la première solution trouvée. Nous appliquerons cette règle pour réduire les expressions dont le premier terme, à gauche de $|$, est totalement réduit.

Remarque : Si nous appliquons arbitrairement la commutativité de $|$, nous pouvons trouver alors n'importe quelle solution. Si nous n'utilisons pas cette propriété, et si nous évitons de permuter les expressions non réduites entre elles, la suite des réductions correspond à un parcours d'arbre en profondeur d'abord (« depth-first »).

Nous n'essaierons pas de définir ces deux opérateurs sous forme de fonctions, mais nous utiliserons ces règles de calcul pour transformer nos systèmes.

4.4. Algorithme de filtrage des segments

La définition initiale de match reste valide :

$$\mathbf{match} : \mathbf{Filtres} \times \mathbf{Données} \to \mathbf{Mémoires} \text{ vérifie}$$

$$\mathbf{match}(\mathbf{filtre} \times \mathbf{donnée}) = \mathbf{m} \quad \text{si et seulement si :} \quad \mathbf{h_m}(\mathbf{filtre}) = \mathbf{donnée}$$

Mais le morphisme $\mathbf{h_m} : \mathbf{Filtres} \to \mathbf{Données}$ est étendu aux filtres avec segments. Sa restriction aux filtres sans segments est invariante. Sa nouvelle définition est :

$$\forall x \in \mathbf{Symb}, \quad \mathbf{h_m}(''x) = x$$

$$\forall x \in \mathbf{Symb}, \quad \mathbf{h_m}(:x) = \mathbf{d} \quad \text{ssi} \quad \mathbf{d} = \mathbf{m}(x)$$

$$\forall t_1, t_2 \in \mathbf{Filtres}, \quad \mathbf{h_m}(t_1 . t_2) = (\mathbf{h_m}(t_1) . \mathbf{h_m}(t_2)) \in \mathbf{Données}$$

$$\mathrm{nil} \in \mathbf{Filtres}, \quad \mathbf{h_m}(\mathrm{nil}) = \mathrm{nil} \in \mathbf{Données}$$

$$\forall x \in \mathbf{Symb}, \quad \mathbf{h_m}((!x . f)) = \mathbf{h_m}((:x @ f)) = \mathbf{m}(x) @ \mathbf{h_m}(f)$$

Appliquant à nouveau la technique d'inversion sur cette définition, on obtient évidemment les mêmes résultats pour les premiers cas. Le cas des filtres de segment est nouveau.

De $(!x . t) = :x @ t$ on déduit :

$$\mathbf{match}((!x . f), d_1 @ d_2) = \mathbf{match}((:x @ f), d_1 @ d_2)$$

De $\mathbf{h_m}(t_1 @ t_2) = (\mathbf{h_m}(t_1) @ \mathbf{h_m}(t_2))$ on déduit :

$$\mathbf{match}((t_1 @ t_2), (d_1 @ d_2)) = \mathbf{match}(t_1, d_1) \langle + \rangle \mathbf{match}(t_2 . d_2)$$

ce qui s'instancie, pour $t_1 = :x$ et $t_2 = f$:

$$\mathbf{match}(:x, d_1) \langle + \rangle \mathbf{match}(f . d_2) = \{x \to d_1\} \langle + \rangle \mathbf{match}(f . d_2)$$

Mais **le découpage d'une donnée d** en $d = d_1 @ d_2$ **n'est pas unique**. Ceci se traduit par des règles différentes selon les découpages possibles de d en $d_1 @ d_2$:

On a en effet : $\mathrm{nil} @ s = s$ et $s @ (s' @ s'') = (s @ s') @ s''$

On peut donc découper nil : nil = nil @ nil

$$\mathbf{match}((\,!\,x.f),\ \text{nil}) = \mathbf{match}((\,:\,x\,@\,f),\ \text{nil}\ @\ \text{nil})$$
$$= \{ x \to \text{nil} \} \langle + \rangle\, \mathbf{match}(f,\ \text{nil})$$

Pour découper $(t.s)$, on a :

$$(t.s) = \text{nil}\ @\ (t.s),$$

d'où

$$\mathbf{match}((\,!\,x.f),\ (t.s)) = \mathbf{match}((\,:\,x\,@\,f),\ \text{nil}\ @\ (t.s))$$

et aussi $(t.s) = (t.d_1\ @\ d_2)$ avec $d_1\ @\ d_2 = s$ d'où

$$\begin{aligned}\mathbf{match}((\,!\,x.f),\ (t.s)) &= \mathbf{match}((\,:\,x\,@\,f),\ (t.d_1\ @\ d_2))\\ &= \mathbf{match}((\,:\,x\,@\,f),\ (t.d_1)\ @\ d_2)\\ &= \mathbf{match}((t.\,:\,x')\ @\,f,\ (t.d_1)\ @\ d_2)\\ &\text{avec } \mathbf{m}(x) = (t.\mathbf{m}(x'))\end{aligned}$$

Remarque : Il s'agit ici encore d'égalités, puisque nous avons remplacé des expressions par des expressions égales. Ces transformations sont réversibles. La décomposition argument par argument, elle, n'est pas réversible. Il faut rester attentif à ne rien perdre.

De la définition des segments, on peut déduire que

$$\mathbf{segment}(\text{nil}) \text{ ou bien } \mathbf{segment}\ ((t.S))$$

d'où :

$$\{ (d_1,\ d_2) \mid (d_1\ @\ d_2) = (t.s) \}$$
$$= \{ (\text{nil},\ (t.s)) \} \cup \{ (d_1,\ d_2) \mid d_1 = (t.s'),\ s'\ @\ d_2 = s \}.$$

Ces deux découpages possibles de $(t.s)$ permettent donc d'écrire :

$$\mathbf{match}((!\,x.f),\ (t.s))$$
$$= \mathbf{match}((:x\ @\ f),\ \text{nil}\ @\ (t.s)) \mid \mathbf{match}((t.\ :x')\ @\ f,\ (t.d_1)\ @\ d_2)$$

avec $\mathbf{m}(x) = (t.\mathbf{m}(x'))$, toutes les décompositions possibles de $(t.s)$ en $(d_1\ @\ d_2)$ étant couvertes par cette expression.

Il reste à la réduire sous-expression par sous-expression, en considérant les décompositions correspondantes comme non-ambiguës. Chaque réduction séparée remplace en fait un ensemble complet de solutions par un sous-ensemble, celui des solutions qui satisfont le découpage qui est exprimé par cette sous-expression. C'est l'expression complète de toutes les décompositions possibles qui assure que l'égalité est vérifiée.

$$\mathbf{match}(\,!x.f),(t.s))=[\{x\to \mathrm{nil}\}\langle + \rangle\,\mathbf{match}(f,(t.s))]\,|$$
$$[\{x\to(t.x')\}\langle + \rangle\,\mathbf{match}((\,!x'.f),s)]$$

Note : dans le cas général d'une algèbre avec des équations entre opérateurs, on peut procéder de la même manière, mais l'examen de toutes les décompositions possibles risque de provoquer une explosion combinatoire. D'autre part, il ne semble pas facile de donner une description des ensembles de décompositions possibles en fonction des ensembles d'équations. Il faudra donc traiter chaque cas en particulier. Remarquons toutefois que la décomposition de chaque cas introduit des opérateurs $\langle + \rangle$, l'énumération des décompositions possibles introduit des opérateurs $|$, et les règles de calcul qui relient $\langle + \rangle$ et $|$ ne dépendent pas des opérateurs de la signature. La structure globale de l'algorithme restera donc la même.

ALGORITHME 3 :

$$\mathbf{match}\,(\mathrm{nil},\,\mathrm{nil})=\{\quad\}$$
$$\mathbf{match}\,(''x,\,x)=\{\quad\}$$
$$\mathbf{match}\,(\,:x,\,\mathbf{d})=\{x\to\mathbf{d}\}$$
$$\mathbf{match}\,((\,!\,x.f),\,\mathrm{nil})=\{x\to\mathrm{nil}\}\langle + \rangle\,\mathbf{match}\,(f,\,\mathrm{nil})$$
$$\mathbf{match}((\,!x.f),\,(t.s))=[\{x\to\mathrm{nil}\}\langle + \rangle\,\mathbf{match}(f.\;(t.s))]$$
$$|[\{x\to(t.x')\}\langle + \rangle\,\mathbf{match}((\,!x'.f),\,s)]$$
$$\mathbf{match}\,((t_1.t_2),(d_1.d_2))=\mathbf{match}\,(t_1,d_1)\langle + \rangle\,\mathbf{match}\,(t_2.d_2)$$

pour $t_1 \neq !x$.

Extension aux filtres de segments isolés : on a simplement un cas de plus, exclusif, donc indépendant des autres :

$$\mathbf{match}(\,!x,\,\mathbf{d})=\{x\to(\mathbf{d})\}\qquad \text{où}\quad (\mathbf{d})=(\mathbf{d}.\mathrm{nil})\ \text{comme convenu.}$$

Nous en ferons abstraction pour la suite, car ce cas est très similaire à **match**$(\,:x,\,\mathbf{d})$.

Il ne reste plus qu'à essayer **d'éliminer ces deux opérateurs** $\langle + \rangle$ et $|$. Comme précédemment, nous allons appliquer la transformation de Wand en essayant de simplifier les arguments fonctionnels. Ceci introduira un accumulateur et des continuations de succès et d'échec.

5. AMÉLIORATIONS DE L'ALGORITHME

5.1. Expliciter le contexte

On commence, comme précédemment, par expliciter les contextes. Ceux-ci résultent des appels récursifs de match dans l'algorithme 3; $\langle + \rangle$ étant commutatif, on a :

$$\mathbf{match}((\,!\,x.f),\ \mathrm{nil}) = \{x \to \mathrm{nil}\} \langle + \rangle \mathbf{match}(f,\ \mathrm{nil})$$
$$= \mathbf{match}(f,\ \mathrm{nil}) \langle + \rangle \{x \to \mathrm{nil}\}$$

$$\mathbf{match}((\,!\,x.f),\ (t.s))$$
$$= [\{x \to \mathrm{nil}\} \langle + \rangle \mathbf{match}(f,\ (t.s))] \,|\, [\{x \to (t.x')\} \langle + \rangle \mathbf{match}((\,!\,x'.f),\ s)]$$
$$= \mathbf{match}(f,\ (t.s)) \langle + \rangle \{x \to \mathrm{nil}\} \,|\, [\{x \to (t.x')\} \langle + \rangle \mathbf{match}((\,!\,x'.f),\ s)]$$
$$\mathbf{match}((t_1.t_2),(d_1.d_2)) = \mathbf{match}(t_1,\ d_1) \langle + \rangle \mathbf{match}(t_2.d_2).$$

Remarque : La forme la plus générale de ces appels est donc

$$\mathbf{match}(f,\ d) \langle + \rangle \mathbf{m} \,|\, \mathbf{reprise}$$

où **reprise** désigne la sous-expression qui suit la première occurrence de $|$ et **m** désigne la sous-expression qui suit la première occurrence de $\langle + \rangle$, avant $|$.

Les cas de la forme $\mathbf{match}(f,\ d) \langle + \rangle \mathbf{m}$ en sont des cas particuliers, pour lesquels **reprise** est indéfini : on a alors $\mathbf{match}(f,\ d) \langle + \rangle \mathbf{m} \,|\, \mathbf{reprise} = \mathbf{reprise} \,|\, \mathbf{match}(f,\ d) \langle + \rangle \mathbf{m}$ puisque $|$ est commutatif, et en appliquant l'interprétation opérationnelle, on obtient $\mathbf{match}(f,\ d) \langle + \rangle \mathbf{m}$.

PROPOSITION : *Cette forme* $\mathbf{match}(f,\ d) \langle + \rangle \mathbf{m} \,|\, \mathbf{reprise}$ *reste stable par emboîtements successifs.*

En effet, si « l'appel récursif » $\mathbf{match}(f,\ d)$ dans $\mathbf{match}(f,\ d) \langle + \rangle \mathbf{m} \,|\, \mathbf{reprise}$ est réécrit lui-même $\mathbf{match}(f',\ d') \langle + \rangle \mathbf{m}' \,|\, \mathbf{reprise}'$, on obtient une expression de la forme :

$$\mathbf{match}(f',\ d') \langle + \rangle \mathbf{m}' \,|\, \mathbf{reprise}') \langle + \rangle \mathbf{m} \,|\, \mathbf{reprise}$$

Le premier opérande de l'occurrence dominante de $|$ dans cette expression n'est donc plus de la forme simple $\mathbf{match}(f, d) \langle + \rangle \mathbf{m}$. Cependant, en raison de la distributivité, on a :

$$(\mathbf{match}\ (f', d') \langle + \rangle \mathbf{m}' \mid \mathbf{reprise}') \langle + \rangle \mathbf{m} \mid \mathbf{reprise}$$
$$= ((\mathbf{match}(f', d') \langle + \rangle \mathbf{m}' \langle + \rangle \mathbf{m}) \mid (\mathbf{reprise}' \langle + \rangle \mathbf{m})) \mid \mathbf{reprise}$$

et, l'opérateur étant associatif, on obtient finalement :

$$\mathbf{match}(f', d') \langle + \rangle (\mathbf{m}' \langle + \rangle \mathbf{m}) \mid ((\mathbf{reprise}' \langle + \rangle \mathbf{m}) \mid \mathbf{reprise})$$

Cette dernière expression est bien de la forme générale qui nous intéresse. Cette propriété ne fait intervenir que les règles de calcul propres aux deux opérateurs $\langle + \rangle$ et $|$; on peut donc espérer la retrouver dans le développement d'un algorithme de filtrage pour une signature quelconque, mais la forme des cas élémentaires dépend de la signature et des équations.

Cette propriété de stabilité de la forme est fondamentale pour la suite, c'est elle qui garantit que les calculs vont atteindre un « régime permanent » à partir duquel il sera possible de définir l'espace des états et d'identifier les arguments signifiants. Il y a donc une certaine parenté avec l'approche de Pettorossi [12], qui recherche un régime permanent dans les traces du calcul.

Dans le cas du filtrage des S-expressions, il est encore possible de faire quelques remarques complémentaires qui permettront de mieux structurer le contexte de calcul, donc ultimement de décomposer la continuation en parties homogènes, qui pourront alors être exploitées plus finement que la continuation complète. Mais cette décomposition est dépendante des cas définis.

Remarque : dans les cas récursifs, on peut, et on doit différencier $\mathbf{match}(f, \mathrm{nil}) \langle + \rangle \{x \to \mathrm{nil}\}$ de $\mathbf{match}(t_1, d_1) \langle + \rangle \mathbf{match}(t_2 . d_2)$. En effet, si ces deux expressions se trouvent emboîtées dans un contexte $[\ldots \langle + \rangle \mathbf{m} \mid \mathbf{reprise}]$, et si $\mathbf{m}$ est déjà réduite en un ensemble de paires, le calcul de l'expression $\{x \to \mathrm{nil}\} \langle + \rangle \mathbf{m}$ est immédiatement possible. Mais celui de $\mathbf{match}(t_2 . d_2) \langle + \rangle \mathbf{m}$ ne l'est pas, car il faut réduire $\mathbf{match}(t_2 . d_2)$ pour pouvoir le faire effectivement.

L'associativité et la commutativité de $\langle + \rangle$ vont nous permettre de regrouper ensemble tous les termes réduits d'une part, tous les termes non réduits de l'autre. Ce regroupement va construire un accumulateur des résultats partiels, permettant donc d'anticiper la réduction finale des expressions de la forme $\mathbf{m}_1 \langle + \rangle \ldots \mathbf{m}_n$ et de détecter les échecs avant de décomposer le problème complètement. Cette détection des échecs permettra de réduire l'expression totale à sa sous-expression **reprise** en appliquant l'interprétation opérationnelle de $|$.

On est donc conduit à transformer la forme générale des contextes :

La forme la plus générale des appels est donc

$$\mathbf{match}\,(f,\ d)\langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$$

où **reprise** désigne la sous-expression qui suit la première occurrence de $|$ et **m** désigne la partie réduite de la sous-expression qui suit la première occurrence de $\langle + \rangle$ jusqu'à $|$, et **succès** la partie non réduite.

PROPOSITION : *Cette forme* $\mathbf{match}\,(f,\ d)\langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$ *reste stable par emboîtements successifs.*

En effet, il suffit de le vérifier comme précédemment :

$$(\mathbf{match}\,(f,\ d)\langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}) \langle + \rangle \mathbf{m}' \langle + \rangle \mathbf{succès}' \,|\, \mathbf{reprise}'$$

se met sous la forme

$$\mathbf{match}\,(f,\ d)\langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \langle + \rangle \mathbf{m}' \langle + \rangle \mathbf{succès}' \,|\, \mathbf{reprise}''$$

d'après la proposition précédente, et il reste à appliquer l'associativité et la commutativité de $\langle + \rangle$ pour obtenir la forme

$$\mathbf{match}\,(f,\ d)\langle + \rangle \mathbf{m} \langle + \rangle \mathbf{m}' \langle + \rangle \mathbf{succès} \langle + \rangle \mathbf{succès}' \,|\, \mathbf{reprise}''$$

Notons encore que cette proposition ne fait intervenir que des propriétés de $\langle + \rangle$ et $|$. Elle est donc généralisable indépendamment de la signature des filtres et des données.

5.2. Simplifier et transformer le contexte

Il nous reste donc à ajouter la forme générale des contextes aux deux membres de chaque équation dans l'algorithme 3, de manière à définir son « régime permanent ». Puis nous tenterons de simplifier les membres droits ainsi obtenus cas par cas.

ALGORITHME 3 *avec contextes. Version* 1 :

$$\mathbf{match}\,(\mathrm{nil},\ \mathrm{nil})\ \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$$
$$= \{\quad\} \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$$

$$\mathbf{match}\,({''x},\ x)\ \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} = \{\quad\} \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$$

$$\textbf{match}\ (:x, \mathbf{d})\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \{ x \to \mathbf{d} \}\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ ((!x.f), \text{nil})\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \{ x \to \text{nil} \}\ \langle + \rangle\, \textbf{match}\ (f, \text{nil})\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ ((!x.f), (t.s))\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = [[\{ x \to \text{nil} \}\ \langle + \rangle\, \textbf{match}\ (f, (t.s))] \mid [\{ x \to (t.x') \}\ \langle + \rangle\, \textbf{match}\ ((!x'.f), s)]]\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ ((t_1.t_2), (d_1.d_2))\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \textbf{match}\ (t_1, d_1)\ \langle + \rangle\, \textbf{match}\ (t_2.d_2)\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise}$$

pour $t_1 \neq !x$.

5.2.1. *Simplifications et mise en ordre des termes*

Appliquant les règles de calcul sur $\langle + \rangle$ et $|$, on obtient :

ALGORITHME 3 *avec contextes. Version* 2 :

$$\textbf{match}\ (\text{nil}, \text{nil})\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \mathbf{m}\ \langle + \rangle\ \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ ('' x, x)\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \mathbf{m}\ \langle + \rangle\ \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ (:x, \mathbf{d})\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \{ x \to \mathbf{d} \}\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ ((!x.f), \text{nil})\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = \textbf{match}\ (f, \text{nil})\ \langle + \rangle\ \{ x \to \text{nil} \}\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise}$$

$$\textbf{match}\ (!x.f), (t.s))\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = (^4)\ \textbf{match}\ (f, (t.s))\ \langle + \rangle\ \{ x \to \text{nil} \}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \textbf{succès} \mid [\textbf{match}\ ((!x'.f), s)\ \langle + \rangle \{ x \to (t.x') \}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \textbf{succès}] \mid \textbf{reprise}$$

$$\textbf{match}\ ((t_1.t_2), (d_1.d_2))\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \textbf{succès} \mid \textbf{reprise} = (^5)\ \textbf{match}\ (t_1, d_1)\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \textbf{match}\ (t_2.d_2)\ \langle + \rangle\ \textbf{succès} \mid \textbf{reprise}$$

pour $t_1 \neq\ !\ x$.

---

(4) Distributivité de $\langle + \rangle$ par rapport à $|$; elle propage les contraintes et assure une recopie.

(5) Associativité et Commutativité de $\langle + \rangle$; différencie accumulateur et continuation de succès.

Cette deuxième version, équivalente à la première, nécessite encore que la réduction des expressions contenant des opérateurs $\langle + \rangle$ et $|$ soit faite. Nous n'allons pourtant pas écrire de fonctions pour représenter ces opérateurs, mais nous allons au contraire essayer de les éliminer.

5.2.2. *Elimination des opérateurs* $\langle + \rangle$ *et* $|$

Ces deux opérateurs pourraient être réalisés effectivement : $\langle + \rangle$ est un opérateur classique sur les tables des symboles, et l'opérateur $|$ correspond au « ? » de ML, et au back-track de PROLOG. Il est possible de le réaliser en Lisp, par exemple à l'aide de couples catch-throw. Mais il est plus intéressant de mettre à profit l'explicitation des contextes pour améliorer l'algorithme.

On essaie d'anticiper le résultat de l'opération $\langle + \rangle$, de manière à simplifier et à expliciter les cas de succès ou d'échecs le plus vite possible.

Idées directrices

(i) les définitions récursives par cas expriment la possibilité de réduire des expressions de la forme $\mathbf{f}(\theta_i)$ en les remplaçant par le membre droit correspondant. De plus, dans une vision opérationnelle de ces définitions, on peut s'intéresser plus particulièrement à la forme réductible la plus externe et la plus à gauche, ce qui correspond au parcours d'arbre le plus répandu dans les interprètes actuels. Dans un souci d'optimisation, on cherche alors à ce que cette forme réductible soit la racine de toute l'expression, ce qui constitue le cas dit post-récursif. En bref, ça ne mange pas de pile...

(ii) Lorsque la racine, l'opérateur dominant toute l'expression, est $\langle + \rangle$ ou $|$, l'expression entière ne peut en principe être réduite que par l'application d'une équation $(\alpha \langle + \rangle \beta = \gamma)$ ou bien $(\alpha | \beta = \gamma)$ correspondante. Comme nous disposons seulement des équations 4.3, mais non d'une définition vraiment calculable, nous ne pourrons que réordonner les termes dans cette expression principale.

(iii) Les permutations faites ont pour but de mettre les expressions sous une forme telle que la transformation de Wand produise un système d'équations post-récursives sans reconstituer exactement la pile d'exécution sous forme d'argument.

Pour désigner cette technique de manière imagée, on peut dire qu'on **fait glisser** la sémantique des opérateurs $\langle + \rangle$ et $|$ vers la fonction match et les fonctions auxiliaires qu'on peut introduire pour faciliter l'expression. Passons maintenant à l'examen de chaque cas, pour déterminer les simplifications éventuelles.

Cas des constantes

Remarque : Les deux premiers cas se réduisent de manière semblable :

$$\mathbf{match}\ (\text{nil}, \text{nil}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} = \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$$

$$\mathbf{match}\ ({}''x, x) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} = \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$$

Il est donc judicieux de les traiter ensemble. D'autre part, les réductions suivantes se définissent en fonction de la forme de **m**, **succès** et **reprise**. Notre centre d'intérêt se déplace donc ici des arguments f et d vers le reste de l'expression, qu'il faut étudier aussi en distinguant des cas. **Chacune de ces raisons** peut justifier l'introduction d'un sous-programme, ici sous la forme d'une fonction auxiliaire, que nous appellerons **suite**. Cette fonction n'est qu'une astuce technique pour réordonner les sous-expressions; du point de vue des valeurs, c'est la fonction identique.

Il faut aussi expliciter les cas pour lesquels les premiers termes ne sont pas définis, de manière à appliquer la règle $\mathbf{m} \,|\, \mathbf{m}' = \mathbf{m}'$ si **m** indéfini. L'expression globale se réduira alors à **reprise**.

$$\mathbf{match}\ (\text{nil}, \mathbf{d}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} = \mathbf{reprise} \ \text{si}\ \mathbf{d} \neq \text{nil}$$

$$\mathbf{match}\ ({}''x, \mathbf{d}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} = \mathbf{reprise} \ \text{si}\ \mathbf{d} \neq x.$$

Cas des variables simples

Remarque : L'expression $\mathbf{match}(:x, \mathbf{d}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$ se réduit en **reprise** lorsque $\mathbf{match}(:x, \mathbf{d}) \langle + \rangle \mathbf{m}$ n'est pas défini.

En effet,

$$\begin{aligned}&\mathbf{match}(:x, \mathbf{d}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}\\ &\qquad = \{ x \to \mathbf{d} \} \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}.\end{aligned}$$

On a alors

$$\begin{aligned}&\{ x \to \mathbf{d} \} \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}\\ &\qquad = \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} \quad \text{si} \quad \mathbf{m}(x) = \mathbf{d},\\ &\qquad\quad \mathbf{reprise} \quad \text{si} \quad \mathbf{m}(x) \neq \mathbf{d},\\ &\qquad\quad (\{ x \to \mathbf{d} \} \cup \mathbf{m}) \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise} \ \text{sinon}.\end{aligned}$$

Cette dernière forme peut s'écrire $\mathbf{m}' \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}$, avec $\mathbf{m}' = (\{ x \to \mathbf{d} \} \cup \mathbf{m})$.

Cas des segments correspondant à une donnée vide

$$\begin{aligned}&\mathbf{match}\ ((\,!\ x.f),\ \mathrm{nil})\ \langle + \rangle\,\mathbf{m}\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\\&\quad=\mathbf{match}\ (f,\ \mathrm{nil})\ \langle + \rangle\,\mathbf{m}\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\ \text{si}\ \mathbf{m}(x)=\mathrm{nil},\\&\qquad\qquad\mathbf{reprise}\quad \text{si}\quad \mathbf{m}(x)\neq\mathrm{nil},\\&\qquad\mathbf{match}(f,\ \mathrm{nil})\,\langle + \rangle(\{x\to\mathrm{nil}\}\cup\mathbf{m})\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\ \text{sinon.}\end{aligned}$$

Cas des segments correspondant à une donnée non vide

$$\begin{aligned}&\mathbf{match}\ ((\,!\ x.f),\ (t.s))\ \langle + \rangle\,\mathbf{m}\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\\&\quad=\mathbf{match}\ (f,\ (t.s))\ \langle + \rangle\ \{\,x\to\mathrm{nil}\,\}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}\\&\quad|\,[\mathbf{match}\ ((\,!\ x'.f),\ s)\ \langle + \rangle\ \{\,x\to(t.x')\,\}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}]\ |\ \mathbf{reprise}\end{aligned}$$

On peut anticiper le calcul de $\{x\to\mathrm{nil}\}\langle + \rangle\mathbf{m}$, ce qui conduit aux cas :

$$\begin{aligned}&\mathbf{match}\ ((\,!\ x.f),\ (t.s))\ \langle + \rangle\,\mathbf{m}\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\\&\quad=\mathbf{match}\ (f,\ (t.s))\ \langle + \rangle\,\mathbf{m}\ \langle + \rangle\ \mathbf{succès}\\&\quad|\,[\mathbf{match}\ ((\,!\ x'.f),\ s)\ \langle + \rangle\ \{\,x\to(t.x')\,\}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}]\ |\ \mathbf{reprise}\\&\qquad\qquad \text{si}\ \mathbf{m}\ (x)=\mathrm{nil},\\&\quad[\mathbf{match}\ ((\,!\ x'.f),\ s)\ \langle + \rangle\ \{\,x\to(t.x')\,\}\ \langle + \rangle\ \mathbf{m}\,\langle + \rangle\ \mathbf{succès}]\ |\ \mathbf{reprise}\\&\qquad\qquad \text{si}\ \mathbf{m}\ (x)\neq\mathrm{nil},\\&\quad\mathbf{match}\ (f,\ (t.s))\ \langle + \rangle\ \{\,x\to\mathrm{nil}\,\}\ \cup\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}\\&\quad|\,[\mathbf{match}\ ((\,!\ x'.f),\ s)\,\langle + \rangle\ \{\,x\to(t.x')\,\}\ \cup\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}]\ |\ \mathbf{reprise}\\&\qquad\qquad \text{sinon}\ \mathbf{m}(x)\ \text{indéfini.}\end{aligned}$$

Remarque : Si $\mathbf{m}(x)=\mathrm{nil}$, on sait que $\{x\to(t.x')\}\langle + \rangle\mathbf{m}$ est indéfini, et on peut donc éliminer le terme

$$[\mathbf{match}((\,!x'.f),\ s)\,\langle + \rangle\{x\to(t.x')\}\langle + \rangle\mathbf{m}\langle + \rangle\mathbf{succès}].$$

Plus généralement, si $\mathbf{m}(x)=\mathbf{d}$, et si $(t.s)=\mathbf{d}\ @\ s'$, alors on a

$$\begin{aligned}&\mathbf{match}\ ((\,!\ x.f),\ (t.s))\ \langle + \rangle\,\mathbf{m}\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\\&\qquad=\mathbf{match}\ (f,\ s')\ \langle + \rangle\,\mathbf{m}\,\langle + \rangle\,\mathbf{succès}\,|\,\mathbf{reprise}\end{aligned}$$

En effet, cela résulte de la définition de **match** comme inverse de $\mathbf{h}_{\mathbf{m}'}$ avec $\mathbf{m}\subset\mathbf{m}'$:

$$\mathbf{h}_{\mathbf{m}'}\ ((\,!\ x.f))=\mathbf{h}_{\mathbf{m}'}\ ((\,:\ x\ @\ f))=\mathbf{d}\ @\ \mathbf{h}_{\mathbf{m}'}(f).$$

Si $(t.s) \neq \mathbf{d} @ s'$, alors

$$\mathbf{match}\ ((\,!\ x.f), (t.s))\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \mathbf{succès} \,|\, \mathbf{reprise} = \mathbf{reprise}.$$

Cas des paires

Il convient d'y ajouter la réduction à **reprise** en cas d'échec

$$\mathbf{match}\ ((t_1.t_2), d)\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \mathbf{succès} \,|\, \mathbf{reprise} = \mathbf{reprise} \text{ si } \mathbf{d} = \text{nil ou } \mathbf{d} \in \mathbf{Symb}.$$

5.2.3. *Expression des fonctions auxiliaires*

Structures des sous-expressions non réduites.

Remarque : la forme générale de la sous-expression **succès** est :

$$\mathbf{succès} = \mathbf{match}\ (\mathbf{t}_{2i}, \mathbf{d}_{2i})\ \langle + \rangle \ldots \mathbf{match}(\mathbf{t}_{22}, \mathbf{d}_{22})\ \langle + \rangle\, \mathbf{match}\ (\mathbf{t}_{21}, \mathbf{d}_{21})$$

En effet, dans la Version 2 de l'algorithme 3, le seul cas qui modifie cette sous-expression, c'est-à-dire la partie non réduite de la sous-expression entre la première occurrence de $\langle + \rangle$ et la première occurrence de $|$ est celui des paires :

$$\mathbf{match}\ ((t_1.t_2), (d_1.d_2))\ \langle + \rangle\, \mathbf{m} \langle + \rangle\, \mathbf{succès} \,|\, \mathbf{reprise}$$
$$= \mathbf{match}\ (t_1, d_1)\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{match}\ (t_2.d_2)\ \langle + \rangle\ \mathbf{succès}\ |\ \mathbf{reprise}.$$

L'induction sur le nombre **i** de fois où cette équation est appliquée est sans difficulté. Notons qu'ici, la forme des cas intervient, et donc que cette partie du développement est sensible à la signature des algèbres des filtres et des données. En cas de généralisation, il faudra donc établir un résultat analogue, adapté au problème traité.

Remarque : la forme générale de la sous-expression **reprise** est :

$$\begin{aligned}\mathbf{reprise} = {} & [\mathbf{match}\,((\,!\,x'_i.f_i), s_i) \langle + \rangle \{x_i \rightarrow (t_i.x'_i)\} \cup \mathbf{m}_i \langle + \rangle \mathbf{succès}_i] \\ & |\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ & |[\mathbf{match}\,((\,!\,x'_2.f_2), s_2) \langle + \rangle \{x_2 \rightarrow (t_2.x'_2)\} \cup \mathbf{m}_2 \langle + \rangle \mathbf{succès}_2] \\ & |[\mathbf{match}\,((\,!\,x'_1.f_1), s_1) \langle + \rangle \{x_1 \rightarrow (t_1.x'_1)\} \cup \mathbf{m}_1 \langle + \rangle \mathbf{succès}_1]\end{aligned}$$

En effet, dans la Version 2 de l'algorithme 3, le seul cas qui modifie le contexte droit de la première occurrence de l'opérateur | est :

$$\mathbf{match}\ ((\,!\ x.f),\ (t.s))\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès} \mid \mathbf{reprise}$$
$$= \mathbf{match}\ (f,\ (t.s))\ \langle + \rangle\ \{\ x \rightarrow \text{nil}\ \}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}$$
$$\mid [\mathbf{match}\ ((\,!\ x'.f),\ s)\ \langle + \rangle\ \{\ x \rightarrow (t.x')\ \}\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{succès}] \mid \mathbf{reprise}.$$

Là encore, l'induction sur le nombre **i** d'applications de ce cas est sans difficulté.

suite (**m** ⟨ + ⟩ **succès** | **reprise**) nécessite de réduire **succès** et **reprise**.

La forme générale de la sous-expression **succès** étant

$$\mathbf{succès} = \mathbf{match}\ (\mathbf{t}_{2i},\ \mathbf{d}_{2i})\ \langle + \rangle\ \ldots\ \mathbf{match}\ (\mathbf{t}_{22},\ \mathbf{d}_{22})\ \langle + \rangle\ \mathbf{match}\ (\mathbf{t}_{21},\ \mathbf{d}_{21})$$

il convient de distinguer les deux cas $\mathbf{i}=0$ et $\mathbf{i}\neq 0$.

$\mathbf{i}=0$: **suite** (**m** | **reprise**) = **m**, d'après l'interprétation opérationnelle de |.

$$\mathbf{i}\neq 0 : \mathbf{suite}\ (\mathbf{m}\ \langle + \rangle\ \mathbf{match}\ (\mathbf{t}_{2i},\ \mathbf{d}_{2i})\ \langle + \rangle\ \ldots\ \mathbf{match}\ (\mathbf{t}_{21},\ \mathbf{d}_{21}) \mid \mathbf{reprise})$$
$$= \mathbf{match}\ (\mathbf{t}_{2i},\ \mathbf{d}_{2i})\ \langle + \rangle\ \mathbf{m}\ \langle + \rangle\ \mathbf{match}\ (\mathbf{t}_{2i-1},\ \mathbf{d}_{2i-1})$$
$$\langle + \rangle\ \ldots\ \mathbf{match}\ (\mathbf{t}_{21},\ \mathbf{d}_{21}) \mid \mathbf{reprise}.$$

Note : On a simplement appliqué la commutativité de ⟨ + ⟩. Le choix de la première sous-expression détermine le parcours d'arbre choisi ; ce choix est arbitraire, ici c'est en « depth-first » en raison de la gestion en pile des expressions à réduire.

Enfin, nous allons faire de même en introduisant une autre variante technique de la fonction identique : la fonction auxiliaire **arrière**, qui a pour but de distinguer le premier terme d'une expression **reprise**.

la forme générale de la sous-expression **reprise** étant :

$$\mathbf{reprise} = [\mathbf{match}\,((\,!\,x'_i.f_i),\,s_i)\ \langle + \rangle\ \{\,x_i \rightarrow (t_i.x'_i)\,\} \cup \mathbf{m}_i\ \langle + \rangle\ \mathbf{succès}_i] \mid \ldots$$
$$\ldots \mid [\mathbf{match}\,((\,!\,x'_1.f_1),\,s_1)\ \langle + \rangle\ \{\,x_1 \rightarrow (t_1.x'_1)\,\} \cup \mathbf{m}_1\ \langle + \rangle\ \mathbf{succès}_1]$$

nous pouvons écrire :

$$\mathbf{i}\neq 0: \mathbf{arrière}([\mathbf{match}((\,!\,x_i'.f_i), s_i)\langle + \rangle\{x_i \to (t_i.x_i')\} \cup \mathbf{m}_i \langle + \rangle \mathbf{succès}_i] \mid \ldots$$
$$\ldots \mid [\mathbf{match}((\,!\,x_1'.f_1), s_1)\langle + \rangle\{x_1 \to (t_1.x_1')\} \cup \mathbf{m}_1 \langle + \rangle \mathbf{succès}_1])$$
$$= [\mathbf{match}((\,!\,x_i'.f_i), s_i)\langle + \rangle\{x_i \to (t_i.x_i')\} \cup \mathbf{m}_i \langle + \rangle \mathbf{succès}_i] \mid \ldots$$
$$\ldots \mid [\mathbf{match}((\,!\,x_1'.f_1), s_1)\langle + \rangle\{x_1 \to (t_1.x_1')\} \cup \mathbf{m}_1 \langle + \rangle \mathbf{succès}_1]$$

$\mathbf{i}=0$: le résultat, étant indéfini, n'a pas besoin d'être défini.

Note : l'introduction de ces deux avatars de la fonction identique peut sembler inutile et étrange.

En fait, elle est motivée par le souhait d'éliminer les opérateurs $\langle + \rangle$ et $\mid$ en jouant sur les passages de paramètres. Les sous-expressions **m**, **succès** et **reprise** seront transformées en paramètres par la transformation de Wand légèrement adaptée. Les opérateurs seront donc vidés de leur signification fonctionnelle par cette transformation, les expressions devenant des données neutres dont il faut reprendre la réduction lorsqu'elles réapparaissent comme résultats. Le but des « fonctions **suite** et **arrière** est de faire reprendre la réduction si c'est nécessaire. La continuation appropriée est alors « relancée », ce qui veut dire du point de vue de nos équations que l'expression courante de l'état du calcul est réordonnée de manière à remettre en tête une expression de la forme **match** (f, d) s'il en reste une à réduire. Mais cela ne se fait pas tout seul.

La fonction **suite** introduite ici sous la forme de deux règles, est une réalisation partielle des deux opérations $\langle + \rangle$ et $\mid$. En effet, elle regroupe une équation de $\mid$ pour le cas $\mathbf{i}=\mathbf{0}$ et les équations exprimant l'associativité et la commutativité de $\langle + \rangle$ pour le cas $\mathbf{i}\neq\mathbf{0}$. Une autre réalisation partielle est donnée par **arrière** et par les cas de la fonction **match** où ces équations ont aussi été appliquées.

Il n'y a donc **pas de réalisation complète** de $\langle + \rangle$ et $\mid$, mais seulement une **réalisation suffisante** : les définitions récursives par cas que nous avons données permettent d'éliminer ces deux opérateurs dans toutes les expressions qui peuvent apparaître au cours du calcul.

En regroupant toutes les équations qui ont été trouvées, et qui ne sont que le résultat de l'application d'équations et d'égalités sur les cas de la définition de **match**, on obtient une nouvelle version, équivalente aux précédentes. Rappelons que par convention, et pour alléger l'écriture, un schéma est satisfait seulement si aucun schéma moins général n'est aussi satisfait.

Ainsi, (nil, nil) n'est pas une instance de (nil, **d**).

ALGORITHME 4 :

match (nil, nil) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **suite** (**m** $\langle + \rangle$ **succès** | **reprise**)

match (nil, **d**) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **arrière** (**reprise**)

match ($''x$, x) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **suite** (**m** $\langle + \rangle$ **succès** | **reprise**)

match ($''x$, **d**) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **arrière** (**reprise**)

match(: x, **d**) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **suite** (**m** $\langle + \rangle$ **succès** | **reprise**) si $\mathbf{m}(x) = \mathbf{d}$
arrière (**reprise**) si $\mathbf{m}(x) = \mathbf{d}'$ et $\mathbf{d}' \neq \mathbf{d}$
suite ($\{ x \to \mathbf{d} \} \cup \mathbf{m}$ $\langle + \rangle$ **succès** | **reprise** si $\mathbf{m}(x)$ non défini.

match ((! $x.f$), nil) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **match** (f, nil) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise** si $\mathbf{m}(x) = \text{nil}$,
arrière (**reprise**) si $\mathbf{m}(x) = \mathbf{d}$ et $\mathbf{d}' \neq \text{nil}$,
match (f, nil) $\langle + \rangle$ ($\{ x \to \text{nil} \} \cup \mathbf{m}$) $\langle + \rangle$ **succès** | **reprise**, si $\mathbf{m}(x)$ non défini.

match ((! $x.f$), $(t.s)$) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **match** (f, s') $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise** si $\mathbf{m}(x)$ défini et $(t.\mathrm{s}) = \mathbf{m}(x) \mathbin{@} s'$,
arrière (**reprise**) si $\mathbf{m}(x)$ défini et $(t.s) \neq \mathbf{m}(x) \mathbin{@} s'$,
[**match** (f, $(t.s)$) $\langle + \rangle$ $\{ x \to \text{nil} \} \cup \mathbf{m}$ $\langle + \rangle$ **succès**]
| [**match** ((! $x'.f$), s) $\langle + \rangle$ $\{ x \to (t.x') \} \cup \mathbf{m}$ $\langle + \rangle$ **succès**]
| **reprise** si $\mathbf{m}(x)$ non défini,

match $(t_1 . t_2)$, $(d_1 . d_2)$) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **match** (t_1, d_1) $\langle + \rangle$ **m** $\langle + \rangle$ **match** (t_2, d_2) $\langle + \rangle$ **succès** | **reprise**

match $(t_1 . t_2)$, d) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**
= **arrière** (**reprise**)

fonctions auxiliaires :

$$\mathbf{suite}\ (\mathbf{m} \mid \mathbf{reprise}) = \mathbf{m}$$

$$\mathbf{suite}\ (\mathbf{m} \langle + \rangle \mathbf{match}\ (\mathbf{t}_{2i}, \mathbf{d}_{2i}) \langle + \rangle \mathbf{succès}' \mid \mathbf{reprise})$$
$$= \mathbf{match}\ (\mathbf{t}_{2i}, \mathbf{d}_{2i}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès}' \mid \mathbf{reprise}$$

$$\mathbf{arrière}\,([\mathbf{match}\,((\,!\,x'_i\,.f_i), s_i) \langle + \rangle \{ x_i \to (t_i\,.\,x'_i) \} \cup \mathbf{m}_i \langle + \rangle \mathbf{succès}_i] \mid \mathbf{reprise}')$$
$$= \mathbf{match}\,((\,!\,x'_i\,.f_i), s_i) \langle + \rangle \{ x_i \to (t_i\,.\,x'_i) \} \cup \mathbf{m}_i \langle + \rangle \mathbf{succès}_i \mid \mathbf{reprise}'.$$

Cette définition récursive par cas recouvre toutes les expressions qui peuvent apparaître lors du calcul de **match**(f, d). Le membre droit correspondant étant obtenu par remplacement successifs de **match**(f, d) par une expression égale, la valeur calculée est celle de **match**(f, d). Appliquons maintenant la transformation de Wand pour transformer le contexte en arguments.

5.3. Une adaptation de la transformation de Wand

Les calculs qui ont conduit à l'algorithme 4 ont déjà anticipé cette transformation en explicitant les contextes possibles. Mais la conversion explicite en continuations n'a pas été faite, pour permettre l'application des règles de calcul sans les transcrire en termes de compositions de fonctions. L'application immédiate de la transformation de Wand conduirait à une fonction auxiliaire à 3 arguments, le troisième étant en général de la forme $\lambda\xi\,.\,\xi \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \mid \mathbf{reprise}$. Les calculs précédents se transcrivent alors mécaniquement, mais cela nécessite de rajouter des contextes gauches « $\lambda\xi\,.\,\xi \langle + \rangle$ » et une étape supplémentaire de β-réduction pour chaque composition. Nous avons donc retardé l'application de la transformation proprement dite.

Le contexte est composé ici de 3 termes bien distincts, que les compositions fonctionnelles de continuations modifient séparément. Plus précisément, la continuation est une expression de la forme $\lambda\xi\,.(\xi \langle + \rangle \tau_1 \langle + \rangle \tau_2 \mid \tau_3)$ où les opérateurs sont associatifs et commutatifs et ont des éléments neutres n_+ et $n_|$. Il est possible de définir un schéma de transformation qui respecte cette structure du contexte. Les règles de Wand deviennent :

Définition d'une fonction auxiliaire :

$$\mathbf{f} : \mathbf{X}_{1\times}\, \mathbf{X}_2 \to \mathbf{Y}$$
$$\mathbf{F} : (\mathbf{X}_{1\times}\, \mathbf{X}_{2\times}\, \mathbf{Y}_\times\, \mathbf{Y}_\times \mathbf{Y}) \to \mathbf{Y},$$
$$\mathbf{f}\,(\theta_1, \theta_2) = \mathbf{F}\,(\theta_1, \theta_2, n_+, n_+, n_|)$$

Transformation des cas d'arrêt :

$$\mathbf{f}(\theta_1, \theta_2) = \rho_i(\theta_1, \theta_2)$$
$$\mathbf{F}(\theta_1, \theta_2, \tau_1, \tau_2, \tau_3) = \rho_i(\theta_1, \theta_2) \langle + \rangle \tau_1 \langle + \rangle \tau_2 \mid \tau_3$$

Transformation des cas récursifs :

$$\mathbf{f}(\theta_1, \theta_2) = \mathbf{f}(\psi(\theta_1, \theta_2)) \langle + \rangle \delta\tau_1 \langle + \rangle \delta\tau_2 \mid \delta\tau_3 \quad \{ \text{avec } \psi : \mathbf{X}^2 \to \mathbf{X}^2 \}$$
$$\mathbf{F}(\theta_1, \theta_2, \tau_1, \tau_2, \tau_3) = \mathbf{F}(\psi(\theta_1, \theta_2), \tau_1 \langle + \rangle \delta\tau_1, \tau_2 \langle + \rangle \delta\tau_2, \tau_3 \mid \delta\tau_3)$$

Fonction auxiliaire de terminaison

Une dernière adaptation est nécessaire pour éviter le calcul effectif des opérations $\langle + \rangle$ et $|$, et elle permet alors de considérer ces opérateurs comme de simples constructeurs d'expressions.

Remplacer chaque cas d'arrêt
$$\mathbf{F}(\theta_1, \theta_2, \tau_1, \tau_2, \tau_3) = \rho_i(\theta_1, \theta_2) \langle + \rangle \tau_1 \langle + \rangle \tau_2 \mid \tau_3$$
par une fonction auxiliaire simplifiant le membre droit :
$$\mathbf{F}(\theta_1, \theta_2, \tau_1, \tau_2, \tau_3) = \alpha_i(\theta_1, \theta_2, \tau_1, \tau_2, \tau_3)$$

Remarque : On travaille ainsi sur l'interprétation libre de la signature $\{\langle + \rangle, |\}$ en faisant abstraction des équations. Les opérateurs ne sont pas interprétés dans les appels récursifs mais seulement à la fin, les expressions peuvent donc être considérées comme de simples structures de données. La continuation perd donc son caractère fonctionnel, et ne nécessite plus le calcul d'une fermeture. Un lisp classique est donc suffisant. L'effet de fermeture a été remplacé par l'anticipation formelle des β-réductions.

6. LE CODE LE-LISP DE LA FONCTION DE SEMI-UNIFICATION

Ce code est le résultat de l'application presque à la lettre des règles de transformation de la partie 5.3 sur l'algorithme 4, d'où le nom de la fonction auxiliaire **match4** qui est introduite. On a simplement groupé les cas qui correspondent à un même schéma de filtre ou de données pour éviter de refaire les tests et pour réduire la taille des conditionnelles. Les opérateurs $\langle + \rangle$ et $|$ sont représentés par des simples **cons**, en raison de la dernière remarque précédente. Le texte des fonctions le-lisp donné ci-dessous est celui de fichiers exécutables incluant quelques commodités minimales de dialogue. Seuls les commentaires ont été ajoutés ou leur police modifiée. Les cas correspondants de l'algorithme 4 sont rappelés à chaque étape.

6.1. La fonction principale match

La fonction principale match appelle la fonction auxiliaire match4 étendue aux continuations selon le schéma de Wand :

```
(de match (f d)
      (prin 'filtre) (pwrite f)
      (print 'donnee d)
      (print 'environnement
      (match4 f d nil nil nil))
  )
```

6.2. Sa version transformée par capture de continuations

Nous avons regroupé sous forme de clauses les cas associés à un même opérateur de filtrage. Chaque fonction ainsi introduite définit le *type comportemental* de l'opérateur de filtrage, c'est-à-dire les modifications des paramètres d'appel qui lui sont associées. Nous en donnons une classification informelle pour guider la réutilisation de ces fonctions dans un filtrage étendu.

```
(de match4 ( f d m s r)
  (cond
      ((null f) (nil4 f d m s r))
      ((iscste f) (cste4 f d m s r))
      ((isvar f) (var4 f d m s r))
      ((isseg f) (seg4 f d m s r))
      ((ispair f) (pair4 f d m s r))
      )
  )
```

6.2.1. *Définition des divers cas*

```
(de iscste (f)
  (and (consp f) (eq (car f) quote)))
(de isvar (f)
  (and (consp f) (eq (car f) indvar)))
(de isseg (f); on observe de l'étage supérieur;
  (and (consp f) (consp (car f)) (eq (caar f) indseg)))
(de ispair (f)
  (consp f))
```

6.2.2. *Définition des traitements appropriés aux divers cas*

Le traitement des filtres vides (et des fins de filtres).

Il suffit de regrouper tous les cas où le filtre est égal à **nil** :

match (nil, nil) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **suite (m** $\langle + \rangle$ **succès** | **reprise)**

match (nil, **d**) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **arrière (reprise)**

```
(de nil4 ( f d m s r)
  (if d (arrière r)
      (suite m s r)))
```

Le traitement des constantes

Il suffit de regrouper tous les cas où le filtre est de la forme ″**x** :

match (″x, x) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **suite (m** $\langle + \rangle$ **succès** | **reprise)**

match (″x, **d**) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **arrière (reprise)**

```
(de cste4 ( f d m s r)
          (if (eq (cadr f)
   ;accès à la constante x dans ″x;
                    d)
                 (suite m s r)
                 (arrière r)))
```

Le type comportemental est « **Constantes** »: mémoire invariante

Le cas des variables

Le cas des des variables s'obtient en regroupant les 3 cas où le filtre est de la forme : x :

match (: x, **d**) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **suite (m** $\langle + \rangle$ **succès** | **reprise)** si **m** (x) = **d**

arrière (reprise) si **m** (x) = **d**′ et **d**′ $\neq$ **d**

suite ({x → **d**} $\cup$ **m** $\langle + \rangle$ **succès** | **reprise)** si **m** (x) non défini.

```
(de var4 ( f d m s r)
  (let ((v (cadr f))) ;accès au nom de la variable, x dans : x;
    (let ((val (assq v m))) ;si déjà définie la paire (x . m (x));
      (cond
        ((eq (cdr val) d) (suite m s r)); m (x) = d;
        (val              (arrière r)); m (x) ≠ d;
        ('else
          (suite` ((,v . ,d) . ,m)
            s r)
          )
        )
      )
    )
)
```

Le type comportemental associé est « **Variables** » : « enrichissement ou modification de la mémoire. »

;*les paires enrichissent la continuation de succès: s*;

match $((t_1 . t_2), (d_1 . d_2)) \langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **match** $(t_1, d_1) \langle + \rangle$ **m** $\langle + \rangle$ **match** $(t_2, d_2) \langle + \rangle$ **succès** | **reprise**

match $((t_1 . t_2), d) \langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

= **arrière (reprise)**

```
(de pair4 (f d m s r)
  (if (consp d)
      (match4
              (car f)
              (car d)
              m
              ((, (cdr f)., (cdr d))., s)
              r)
      (arrière r)
  )
)
```

Le type comportemental associé est « **Conjonctif** » : enrichissement de la continuation de succès. Il correspond à une accumulation de contraintes, et donc à une conjonction. Le filtrage de constructeurs n-aires a un type comportemental analogue.

Il en est de même lorsqu'on introduit des filtres du type « et » : (and *f* 1 *f* 2). L'augmentation de la continuation de succès est alors

(((, (caddr *f*)., (car *d*). (, (cdr *f*)., (cdr *d*))., *s*))

où (caddr *f*) désigne la partie *f* 2 du filtre courant.

A l'opposé, les variables de segment, qui suivent, ont pour type comportemental :

« **Variable** » : Enrichissement ou Modification de la mémoire

« **Disjonctif** » : Enrichissement de la continuation d'échec.

La première partie correspond à l'instanciation d'une variable nommée. La deuxième partie correspond au non-déterminisme qui résulte de l'inversion d'une fonction non injective, donc à une disjonction.

On peut étendre le filtrage par l'ajout de segments indéterminés, à l'image des symboles « * » du shell UNIX; dans ce cas, le type comportemental associé est seulement « Disjonctif ».

Le traitement d'un filtre logique « ou », (or f 1 f 2), correspond à ce même type comportemental.

Plus généralement, de nouveaux opérateurs de filtrage [12] peuvent être définis dans ce cadre, en précisant leur sémantique par la valeur des paramètres

(filtre, donnée, mémoire, succès, échec)

pour l'appel suivant de la fonction de filtrage. Ceci évite de repartir d'une spécification de ces opérateurs à partir des substitutions. Cependant, une telle définition peut être préférable pour leur donner un sens intuitif. Il faut alors refaire un développement semblable à celui que nous avons décrit pour justifier les règles de calcul.

,le cas des variables de segment;

match (! $x.f$), (nil) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

$=$ **match** (f, nil) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise** *si* **m**$(x)=$nil,

arrière (reprise) si **m** $(x)=$ **d** et **d**$' \neq$ nil,

match (f, nil) $\langle + \rangle$ ($\{x \to \text{nil}\}$ $\cup$ **m**) $\langle + \rangle$ **succès** | **reprise**, si **m**(x) non défini.

match ((! $x.f$), $(t.s)$) $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise**

$=$ **match** (f, s') $\langle + \rangle$ **m** $\langle + \rangle$ **succès** | **reprise** si **m**(x) défini et $(t.s)=$ **m** (x) @ s',

arrière (reprise) si **m** (x) défini et $(t.s) \neq$ **m** (x) @ s',

[**match** (f, $(t.s)$) $\langle + \rangle$ $\{x \to \text{nil}\}$ $\cup$ **m** $\langle + \rangle$ **succès**]

| [**match** ((! $x'.f$), s) $\langle + \rangle$ $\{x \to (t.x')\}$ $\cup$ **m** $\langle + \rangle$ **succès**]

| **reprise** si **m** (x) non défini,

```
(de seg4 (f d m s r)     ;filtre et donnée n'ont pas encore éclaté, on domine!;
  (let ((v (cadr (car f))))     ;accès au nom de la variable;
    (let ((val (assq v m)))     ;si déjà définie la paire (v.mv);
      (cond
        (val (cheq (cdr val) (cdr f) d m s r))
        ('else
          (let (futur`(, v., nil)))     ;valeur modifiable;
            (let (m`(, futur., m))) ;{x → nil} ∪ m;
              (match4 (cdr f)
                d
                m
                s
                `((, futur, (cdr f), d, m, s).,r)
                ))))
        )
      )
    )
)
```

;et son aide si le segment a une valeur dans m;
réalise les cas :

$$\mathbf{match}(f, s') \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}\ \textit{si}\ \mathbf{m}(x)\ \text{défini et}\ (t.s) = \mathbf{m}(x) @ s',$$

$$\mathbf{arrière}(\mathbf{reprise}) \qquad \text{si}\ \mathbf{m}(x)\ \text{défini et}\ (t.s) \neq \mathbf{m}(x) @ s',$$

```
(de cheq (val f d m s r)     ;vérifie que val est préfixe de d;
  (cond
    ((null val)
      (match4 f d m s r))     ;ça finit bien;
    ((eq (car val) (car d))
      (cheq (cdr val) f (cdr d) m s r)) ;ça n'est pas fini;
    ('else
      (arrière r))     ;ça finit mal;
    )
  )
```

;les fonctions auxiliaires;

$$\mathbf{suite}(\mathbf{m} \langle + \rangle \mathbf{succès} \,|\, \mathbf{reprise}) = \mathbf{m},$$

$$\mathbf{suite}\ (\mathbf{m} \langle + \rangle \mathbf{match}\ (\mathbf{t}_{2i}, \mathbf{d}_{2i}) \langle + \rangle \mathbf{succès}' \perp \mathbf{reprise})$$

$$= \mathbf{match}\ (\mathbf{t}_{2i}, \mathbf{d}_{2i}) \langle + \rangle \mathbf{m} \langle + \rangle \mathbf{succès}' \perp \mathbf{reprise}$$

```
(de suite (m s r)     ;suite du travail en cas de succès;
  (if s
    (let (     ;éclater s, puis relancer;
        (f (caar s))
        (d (cdar s))
        (s (cdr s))
        )
      (match4 f d m s r))
    m)
  )
```

$$\textbf{arrière}\ ([\textbf{match}\ ((!\,x'_i\,.\,f_i),\ s_i)\langle + \rangle \{x_i \to (t_i\,.\,x'_i)\} \cup \mathbf{m}_i \langle + \rangle\ \textbf{succès}_i]\ |\ \textbf{reprise}')$$
$$= \textbf{match}\ ((!\,x'_i\,.\,f_i),\ s_i)\langle + \rangle \{x_i \to (t_i\,.\,x'_i)\} \cup \mathbf{m}_i \langle + \rangle\ \textbf{succès}_i\ |\ \textbf{reprise}'$$

```
(de arrière (r)      ;suite du travail en cas d'échec;
  (if r
    (let (
        (bloc (car r)) ;éclater r;
      (r (cdr r))
      )
      (let (      ;éclater bloc, puis essayer de relancer;
          (futur (nextl bloc))      ;car;
          (f (nextl bloc))        ;cadr;
          (d (nextl bloc))        ;caddr;
          (m (nextl bloc))        ;cadddr;
          (s (nextl bloc))        ;caddddr;
          )
        (if (atom d)
            (arrière r)      ;bloc courant epuise;
            ;sinon mise à jour et progression dans bloc;
          (setq futur
              (cdr (rplacd futur
                 (, (nextl d)))))
            ;et ça repart sur le bloc mis à jour;
          (match4
              f
              d
              m
              s
              `((, futur, f, d, m, s)., r)
              )
          )
        )
      )
    )
  )
```

7. CONCLUSIONS

Le programme ainsi obtenu a été effectivement utilisé comme composant d'interprète (Plasma) et d'outils d'abstraction fonctionnelle (AVEC, SANS). A cette occasion, il a fait l'objet de quelques extensions dont nous ne parlerons pas, et de quelques optimisations dont nous esquisserons l'idée directrice ci-dessous. Nous considérons le programme présenté ici comme une charpente pour d'autres programmes de même nature, et en plus de sa structure, nous avons essayé de mettre en évidence les raisons pour lesquelles cette structure se dégage de l'expression du problème. Ainsi, nous pensons faciliter l'adaptation de ce programme à des problèmes et des circonstances similaires mais non identiques à ceux que nous avons effectivement traités.

L'idée originelle de la structure de l'algorithme remonte à 1978, et consiste à considérer le filtre comme un programme, et la semi-unification comme un interprète de ce programme. La notion de continuation vient alors naturellement pour résoudre les problèmes de reprise en marche d'une recherche considérée comme terminée. La réalisation du programme est aisée, mais la justification pose des problèmes délicats. En effet, le programme fonctionne comme un tout, et il est difficile de trouver un sens raisonnable à ses résultats intermédiaires, l'expression formelle en est illisible.

Le développement donné ici ne constitue pas à proprement parler une preuve du programme, car certains passages peuvent sembler un peu discutables, le point le plus faible étant le caractère exhaustif de l'examen des cas d'échec local. Sur ce point, une assistance de la machine pour l'énumération de ces cas à partir de la définition des domaines des arguments aurait été précieuse. Mais nous nous sommes efforcés de donner les raisons des règles employées et des décompositions appliquées, ce qui jalonne le chemin pour une preuve au sens strict.

7.1. Capacités et performances

Le programme est capable de reprendre la recherche d'une nouvelle solution dans un sous-arbre pour lequel une première solution avait été trouvée et considérée comme satisfaisante. Ainsi, par exemple, la semi-unification d'une structure de la forme

(! avant (! sujet ″est ! complément)

! entre (! autre_sujet ″est ! complément) ! après)

avec une donnée de la forme

(... (la chatte dont le pelage est roux est sur la chaise) ...

(le coussin est sur la chaise) ...)

Cette capacité est particulièrement utile dans le cas d'algorithmes d'abstraction fonctionnelle en logique combinatoire, où le « cons » représente en fait l'application d'une fonction, d'un combinateur.

Ainsi, l'abstraction du célèbre combinateur **S** repose sur la reconnaissance d'un filtre de la forme $(!D:Z(!M:Z)!F)$, et l'utilisation de la mémoire obtenue pour instancier $(''\mathbf{S}:D:M:Z!F)$.

La définition de combinateurs plus élaborés est possible, et conduit à des filtres de complexité en longueur et profondeur arbitraire.

La reprise de la recherche sur la suite de la première sous-liste alors que cette recherche avait réussi une première fois s'apparente alors au mécanisme principal de recherche de Prolog. Une légère différence est due à l'arrêt sur la première solution, qui correspond au « cut ». En Prolog, l'effet d'un « cut » est définitif. Ici, l'effet du « cut » sur le sous-arbre est annulé en cas d'échec dû à la suite de la recherche dans le contexte droit du sous-arbre.

Les performances moyennes d'un tel programme sont complexes à évaluer et leur estimation peut faire l'objet d'un développement aussi long que celui du programme lui-même. Nous pouvons en donner ici une estimation sommaire :

Si le filtre ne contient ni sous-listes ni variables de segment, le nombre de tests et de liaisons est linéaire en fonction de la taille minimale du filtre et de la donnée. La place occupée est proportionnelle au nombre $\mathbf{V}$ de variables, soit $\mathbf{V}.\ \mathbf{m}$ si $\mathbf{m}$ désigne la place occupée par une liaison en mémoire.

Chaque correspondance sous-liste $\langle - \rangle$ sous-liste entraîne la consommation de deux doublets pour enrichir la continuation de succès. Rien ne change du point de vue du nombre d'étapes élémentaires (tests et liaisons). Si le filtre ne contient pas de variables de segments, le temps est proportionnel au nombre total $(\mathbf{N}+\mathbf{F})$ de nœuds et de feuilles du filtre et de la donnée. (Ils sont égaux en cas de succès).

Chaque variable de segment peut provoquer un nombre de reprises proportionnel à la longueur de la fin de la sous-liste de la donnée qui correspond à la sous-liste où se trouve la variable de segment. Une majoration très grossière du nombre de reprises est donc $\mathbf{L}^{\mathbf{s}}$, où $\mathbf{L}$ est la longueur de la plus longue sous-liste de la donnée, et $\mathbf{s}$ le nombre de variables de segment du filtre.

Chaque reprise peut entraîner au maximum un parcours complet du filtre. Globalement, on peut majorer le temps de calcul par $(\mathbf{N}+\mathbf{F}).\mathbf{L}^{\mathbf{s}}$.

La taille occupée est majorée par $(\mathbf{V}+\mathbf{s})\,\mathbf{m}+2.\mathbf{N}+\mathbf{s}\,\mathbf{b}$, où b désigne la taille d'un bloc de reprise. Cette estimation est très grossière et pessimiste, mais elle indique nettement qu'il faut en priorité réduire le nombre des reprises, en utilisant la position des variables de segment.

7.2. Optimisation

7.2.1. *Réduction du non-déterminisme*

Il est possible d'analyser un filtre de segment afin de distinguer ceux qui, malgré les apparences, n'induisent aucun non-déterminisme. En effet, si la longueur à donner à une variable de segment est a priori indéterminée, on peut lever cette indétermination si aucune autre nouvelle variable de segment ne la suit.

Exemples :

$(!x''g!y)$ induit un non déterminisme sur x, mais non sur y une fois x fixé.

$(!x:z)$ n'induit aucun non déterminisme : x sera la donnée privée de son dernier élément.

$(:x!y!x)$ n'induit aucun non déterminisme : x étant spécifié par le premier élément de la donnée, la longueur de y s'en déduira.

$(!x:y!x:g!x)$ n'induit aucun non déterminisme : il suffit de vérifier que le reste de la division par 3 de la longueur de la donnée est égale à 2, et la longueur de x doit être le quotient de cette division.

Il est donc judicieux de modifier la clause de l'algorithme traitant les variables de segments afin de traiter à part tous ces cas et leurs combinaisons. On introduit la notion de **Longueur** d'une variable de segment, celle-ci pouvant être partiellement définie en fonction de la suite du filtre et de la donnée.

Cette clause devient alors :

match $((!x.f),(t.s))\langle + \rangle$**m**$\langle + \rangle$**succès** | **reprise**

$=$**match** $(f, z)\langle + \rangle$**m**$\langle + \rangle$**succès** | **reprise**

si **m**$(x)=v$ et $(t.s)=v$ @ z,

arrière (reprise) *si* **m**$(x)=v$ et $(t.s)\neq v$ @ z,

suite $(\{x\to(t.s)\}\cup$ **m**$\langle + \rangle$**succès** | **reprise**)

si **m**(x) non défini et $f=\text{nil}$,

match $(f,$ **reste**$(\mathbf{L}, (t.s)))\langle + \rangle\{x\to$ **tête**$(\mathbf{L}, (t.s))\}\cup$ **m**

$\langle + \rangle$ **succès** | **reprise**

si **m** (x) non défini et $\mathbf{L}=$ **Longueur** $((!x.f),(t.s))\geqq 0$

match $(f,(t.s))\langle + \rangle\{x\to\text{nil}\}\cup$ **m** $\langle + \rangle$ **succès**

| [**match** $((!x'.f), s)\langle + \rangle\{x\to(t.x')\}\cup$ **m** $\langle + \rangle$ **succès**] | **reprise**

si **m** (x) non défini et **Longueur** $((!x.f),(t.s))$ non défini ou négatif

avec

Longueur $((!x.f),(t.s))$

$=$[**Taille** $((t.s))$ - **Résidu** $(!x, f)$] /**Nombre** $(!x,(!x.f))$

Taille $((t.s))=1+$**Taille** (s);   **Taille** $(\text{nil})=0$;

c'est la fonction standard « length » de lisp

$$\textbf{Résidu}\ (!\ x,\ (!\ x\ .\ f)) = \textbf{Résidu}\ (!\ x, f);$$

les occurences suivantes de ! x sont ignorées

$$\textbf{Résidu}\ (!\ x,\ (!\ y\ .\ f)) = \textbf{Taille}\ (\mathbf{m}\ (y)) + \textbf{Résidu}\ (!\ x, f);\ \text{défini ssi } \mathbf{m}\ (y)\ \text{défini !}$$

$$\textbf{Résidu}\ (!\ x,\ (:\ y\ .\ f)) = 1 + \textbf{Résidu}\ (!\ x, f)$$

$$\textbf{Résidu}\ (!\ x,\ (''y\ .\ f)) = 1 + \textbf{Résidu}\ (!\ x, f)$$

$$\textbf{Résidu}\ (!\ x,\ \text{nil}) = 0$$

et

$$\textbf{Nombre}\ (!\ x,\ (!\ x\ .\ f)) = 1 + \textbf{Nombre}\ (!\ x, f)$$

$$\textbf{Nombre}\ (!\ x,\ (f_1\ .\ f_2)) = \textbf{Nombre}\ (!\ x, f_1) + \textbf{Nombre}\ (!\ x, f_2)$$

$$\textbf{Nombre}\ (!\ x, f) = 0\ \text{sinon.}$$

La traduction de cette nouvelle version en le-lisp ne présente pas de difficulté majeure, **tête**(**L**, S) se réduisant à prendre les **L** premiers éléments de S, **reste**(**L**, S) le reste.

7.2.2. *Compilation*

Nous avons laissé de côté l'approche compilation, mais celle-ci permet des optimisations plus poussées, le filtre pouvant alors être exploré plus à fond, et les concepts que nous utilisons peuvent aussi s'y appliquer. L'évaluation partielle prendra alors le relai. [6,7] (Emanuelson, 1980, 1982), [10] (Heering, 85).

7.3. Généralisations

7.3.1. *Enrichissements de l'algèbre des filtres et des données*

Un simple enrichissement de la signature ne semble pas poser de problème particulier, comme nous l'avons vu dans les parties 3 et 4 : cela se traduit simplement par l'ajout d'un nouveau cas par opérateur nouveau, avec enrichissement de la continuation de succès (type conjonctif). Un enrichissement des équations, ou quotient, conduira plutôt à un nouveau cas disjonctif, mais l'introduction d'une seule nouvelle équation peut en entraîner beaucoup d'autres, et il est impossible de discuter de ce type d'ajout de manière générale.

7.3.2. *Transformation de Wand adaptée à la structure de la continuation*

La généralisation semble possible pour des continuations de la forme $\lambda\xi.(\xi \bullet_1 \tau_1 \bullet_2 \tau_2 \ldots \bullet_n \tau_n)$ où chaque opérateur définit une structure de monoïde commutatif et où les opérateurs peuvent commuter entre eux par des règles de distributivité (i. e. : chaque $\bullet_x$ est distributif pour tous les opérateurs d'indice supérieur). Mais elle ne présente d'intérêt que si un problème concret se pose dans ces termes. Nous n'avons pas en vue d'application d'une telle généralisation pour le moment.

BIBLIOGRAPHIE

1. M. A. Arbib et E. G. Manes, *Arrows, Structures, and Functors — The Categorical Imperative*, Academic Press Inc, 1975.
2. R. S. Bird, *The Promotion and Accumulation Strategies in Transformational Programming*, A.C.M.-T.O.P.L.A.S., vol. 6, n° 4, octobre 1984, p. 487-504.
3. B. Courcelle et F. Lavandier, *Définitions récursives par cas*, R.A.I.R.O. Informatique théorique, vol. 18, n° 2, 1983, p. 91-129.
4. J. Darlington, *Program Transformations*, Functional Programming and its applications, Henderson and Turner, Cambridge University Press, 1982.
5. J.-L. Durieux, *Interprétation du langage PLASMA*, Actes 10e École de Printemps, « Compilation et Interprétation », LITP83.17, Paris, mars 82.
6. P. Emanuelson et A. Haraldsson, *On Compiling Embedded Languages in LISP*, Conf. Record LISP-Conference 1980, p. 208-215.
7. P. Emanuelson, *Systematic Specification and Compilation of Patterns*, Tech Rep. Software System Research Center, Linköping University, 1982.
8. D. Eppstein, *A Heuristic Approach to Program Inversion*, Actes I.J.C.A.I. 1985, p. 219-221.
9. J. Foisseau, R. Jacquart, M. Lemaître, M. Lemoine et G. Zanon, *Le système SPRAC : expression et gestion de spécifications, d'algorithmes et de représentations*, TSI, vol. 4, n° 2, p. 237-254.
10. J. Heering, *Partial Evaluation and Ω-Completeness of Algebraic Specifications*, Esprit Project GIPE, Deliverable 5, D5.A1, novembre 1985.
11. R. E. Korf, *Inversion of Applicative Programs*, Actes I.J.C.A.I. 1981, Vancouver, p. 1007-1009.
12. A. Pettorossi, *Transformation Strategies for Deriving On Line Programs*, Actes CAAP86, Nice, mars 86, Springer-LNCS, n° 214, p. 127-141.
13. E. Saint-James, *Fonctionnalité et filtrage : nouveaux algorithmes en logique combinatoire typée*, Thèse 3e Cycle, LITP84.39, Paris, juin 1982.
14. J. E. Stoy, *Mathematical Aspects of Functional Programming*, dans « Functional Programming and its Applications », p. 217-252, Darlington, Henderson and Turner editors, Cambridge University Press, 1982.
15. D. Turner, *Recursion Equations as a Programming Language*, dans « Functional Programming and its Applications », p. 1-28, Darlington, Henderson et Turner ed., Cambridge University Press, 1982.

16. M. Wand, *Continuation Based Program Transformation Strategies*, J.A.C.M., vol. 27, n° 1, 1980, p. 164-180.
17. D. S. Wile, *Program Developments : Formal Explanations of Implementations*, C.A.C.M., vol. 26, n° 11, 1983, p. 902-911.