Notations personnalisables* (Custom notations)

Nic Volanschi

Matchbox (myPatterns.free.fr)
&
Metaware (www.metaware.fr)

(*) N. Volanschi. Pattern Matching for the Masses using Custom Notations.
Science of Computer Programming (2012). DOI

Une fonctionnalité souvent oubliée

- Le ”pattern matching” a fait ses preuves
  - Programmes plus concis, plus lisibles
  - Encourage une programmation élégante
  - Permet des extensions très puissantes
- Pourtant, négligé par les langages courants
  - Absent dans: C/C++, Java, C#
  - Limité à du texte en: Perl, JavaScript, ...
  - Limité à du XML en: Xpath/XSLT, ...
- Préjugés courants:
  - ”Incompatible avec objets / types abstraits”
  - ”Rendu inutile par les appels virtuels”

Pourtant, des preuves existent...

- Compatibilité avec TDA (lang. fonctionnels)
  - Views [Wadler'87]
  - Active patterns [Erwig'97, F#'07], ...
- Préprocesseurs de Java:
  - **Pizza**, Matchete, JMatch, TOM
  - Scala (compilé en bytecode)
- Boudées par les concepteurs de langages industriels
- Solution: implantation *dans* le langage
  - En prime: syntaxe personnalisable

Notations personnalisables

- Ingrédients
  - Type: $T \subset D$
  - Langage de patterns: $L_T \subset (A \uplus \{\%\})^*$
  - Filtrage: $F_T(p) \subseteq T$
  - Projections: $\%_i : F_T(p) \rightarrow T_i$
- $match_T(d, p) \Leftrightarrow p \in L_T \wedge d \in F_T(p)$.
- Application à $d \in T$
  $p(d) = \langle \%_1(d), \ldots \%_{|p|}(d) \rangle$
- Notation = ensemble de ”déconstructeurs” en ”syntaxe concrète”

Ex: Notation de liste

- $(\forall n \geq 0)$ **[%$_1$, ... %$_n$]** : {d | length(d) = n}
  - ou: %$_i$ = le i^{ème} élément de la liste
- $L_T \subset$ {**[**, **]**, **,**, **%**}*
- Ex de patterns:
  - p = **[]** : {d | length(d) = 0},
    p([]) = <>
  - p' = **[%,%,%]** : {d | length(d) = 3}
    p'([5,7,9]) = <5, 7, 9>

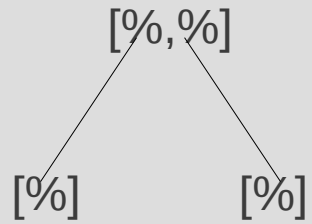
Ex: Notation pour nombres complexes

- Définition
  - **$\%_{re}$ + $\%_{im}$ i**
  - **$\%_{re}$** : {d | $\%_{im}$(d) = 0}
- Ex d'application
  - p = **$\%_{re}$ + $\%_{im}$ i**

    p(2+3i) = <2,3>

Composition de patterns

- Pattern composé = arbre de patterns
  - Obtenu en substituant des % par des patterns
  - Application = composition des déconstructeurs
- Ex:
  - p = **[[%][%]]** = <**[%]**,**[%]**> o **[%,%]**



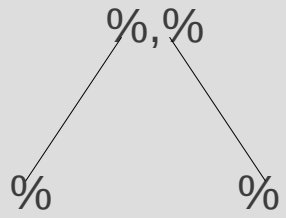
p([[5],[7]]) =

<**[%]**,**[%]**>(**[%,%]**([[5],[7]]) =

<**[%]**,**[%]**>(<[5],[7]>) = <<5>,<7>> = <5,7>

Notation ambigue

- (∀n ≥ 0) **%**$_1$, ... **%**$_n$: {d | length(d) = n}
  - ou: %$_i$ = le i^{ème} élément de la liste
- Ex: %,% =
  - <%,%> o %,% ?
  - %,% o % ?



- Problème classique de composition de langages
  - Solution: contraindre les langages de patterns
  - Solution: vérifier la composabilité
  - Solution: figer un ensemble de notations

Solution adoptée

- Solution pragmatique: notations parenthésées:
  - Tout pattern commence par ([{ et finit par }])
  - Aucune autre paranthèse dans les patterns
    - ou alors elles sont échappées: \(\) ...
- Conséquences
  - On peut retrouver l'arbre d'une composition
  - Tout langage de patterns peut être rendu parenthésé
  - Permet de composer sans restrictions des langages conçus indépendamments
- Exception de paranthésage: types de base

Ex: Notation des types de base

- Booléens
  - true : {true}
  - false : {false}
- Nombres
  - (∀n ∈ Int) n : {n}
  - (∀f ∈ Float) f : {f' ∈ Float | f' = f}
- Chaînes
  - (∀s ∈ String) 's' : {s}
  - (∀s ∈ String) ”s” : {s}
  - (∀s ∈ Regex) /r/ : {s ∈ String | s ~ r}
- Ex: [/^(.*)son/,/^Du(.*)/] ~ [”Smithson”, ”Dupont”] → <”Smith”,”pont”>

Ex: Notation JSON généralisée

- JSON = JavaScript Object Notation:
  - [{action:”click”,x:121,y:347}, {action:”hit,key=”q”}]
- JSON pour listes, généralisée à la Prolog:
  - $(\forall n \geq 0)$ **[%$_1$, . . . %$_n$]** : {d | length(d) = n}
  - $(\forall n \geq 0)$ **[%$_1$, . . . %$_n$|%$_{>n}$]** : {d | length(d) ≥ n}
- JSON pour objets:
  - $(\forall n \geq 0)(\forall key_i \in Identifier)i \in [1..n]$
  - **{**key_1**:%**$_{key1}$**, . . .** key_n**:%**$_{keyn}$**}** : {d | $(\forall i)\exists(d.key_i)$}
- Ex de pattern:
  - **[{action:”click”,x:%,y:%} | %]**

La bibliothèque Matchbox

- Implantation comme des bibliothèques minimalistes: Java & JavaScript
  - Une notation = une méhode `match()`
  - Une notation par défaut: JSON généralisée
  - Peut être rédéfinie par une notation spécialisée
- Ex: red-black trees
  - **[%$_l$ %$_v$ %$_r$]** : {d | d.color = black}
  - **(%$_l$ %$_v$ %$_r$)** : {d | d.color = red}
- Notations interprétées ou compilées
- Variable nommées, patterns non-linéaires

Ex: notation interprétée

```
Tree.prototype.matches = function(pat, off, sub) {
  off = matchChar((this.color == black?
                   "[": "("), pat, off);
  off = matchData(this.left, pat, off, sub);
  off = matchChar(" ", pat, off);
  off = matchData(this.value, pat, off, sub);
  off = matchChar(" ", pat, off);
  off = matchData(this.right, pat, off, sub);
  return matchChar((this.color == black?
                    "]": ")"), pat, off);
}
```

Ex: notation compilée

```
Tree.prototype.matcher = function(pat, off) {
  var color, fun1, fun2, fun3;
  if(pat.charAt(off) == "[") color = black;
  else if(pat.charAt(off) == "(") color = red;
  else throw "fail";
  off++;
  [off, fun1] = parseType(pat, off, Tree);
  off = matchChar(" ", pat, off);
  [off, fun2] = parseType(pat, off, Number);
  off = matchChar(" ", pat, off);
  [off, fun3] = parseType(pat, off, Tree);
  off = matchChar(((color == red)? ")": "]"), pat, off);
  return [off,
          function(data, sub) {
            if(data == null) throw "fail";
            if(data.color != color) throw "fail";
            fun1(data.left, sub);
            fun2(data.value, sub);
            fun3(data.right, sub);
          }];
}
```

Ex: utilisation de notation

```
Tree.prototype.balance = function() {
  var s = matchAny(this,
                ["[((%a %x %b) %y %c) %z %d]",
                 "[(%a %x (%b %y %c)) %z %d]",
                 "[%a %x ((%b %y %c) %z %d)]",
                 "[%a %x (%b %y (%c %z %d))]"]);
  if(s)
    return new Tree(red, new Tree(black, s.a, s.x, s.b),
                s.y, new Tree(black, s.c, s.z, s.d));
  return this;
};
```

Performances

Performance of matching notations (red-black trees).

Version	Hand-crafted	Compiled notation		Interpreted notation	
		linear	Non-linear	linear	non-linear
Java time (ms)	371	854	1444	1631	2257
Java slowdown		*2.3*	*3.9*	*4.4*	*6.1*
JavaScript time (ms)	99	317	365	521	632
JavaScript slowdown		*3.2*	*3.7*	*5.3*	*6.4*

Performance of regular expressions.

Version	Manual	Optimized regex	Regex
Java time (msec)	400	4225	5162
Java slowdown		*10.6*	*12.9*
JavaScript time (msec)	590	1714	1443
JavaScript slowdown		*2.9*	*2.5*

Applications

- Extensions:
  - Patterns comme constructeurs
  - Patterns disjonctifs, conjonctifs, contraints, subordonnés
- Applications
  - Langages de requêtes: JSLINQ, JsonQuery
  - Active patterns
  - Règles de réécriture

Application: Dijskstra/réécriture

- Dijkstra(G,Poids,sdeb)
  Initialisation(G,sdeb)
  Q := liste ordonné des nœuds /distance à N
  tant que Q n'est pas un ensemble vide
  faire s1 := premier(Q)
  Q := Q privé de s1
  pour chaque nœud s2 voisin de s1
  faire maj_distances(s1,s2);
  re-insérer ces noeuds

Réécriture: Dijkstra/Matchbox

```
let trans([Path+] -> [Path*])
    [{to:%n,dist:%a}|%t] →
     print("Distance to %n = %a via %v \n");
     (rewriteAll t with
       {to:%m} &
       {to:{links:{{to:%n,dist:%c}|%z}},dist:%b} / b > a + c
          →   {to:%m,dist:%(a + c),via:%n})
```

```
static void transRule(SortedSet<Path> set) {
    while(!set.isEmpty()) {
        final Subst s0 = pattern("[{to:%n,dist:%a}|%t]").and(pattern("[%h|%t]")).match(set);
        Path path = (Path)s0.get('h');
        set.remove(path);
        Matchbox.print(”dist to %n = %a via %v \n”, s0);
        Rewrite.Rule<Path> rule = new Rewrite.Rule<Path>() {
            { lhs = new Pattern.And(
                    pattern("{to:%m}"),
                    pattern("{to:{links:{{to:%n,dist:%c}|%z}},dist:%b}", s0).where(
                        new Subst.Check() {
                            public boolean check(Subst s) {
                                return ((Integer)s.get('b') >
                                        (Integer)s.get('a') + (Integer)s.get('c'));
                            }
                        }));
            }
            public Path rhs(Subst s) {
                s.set('d', (Integer)s.get('a') + (Integer)s.get('c'));
                return Matchbox.newInstance("{to:%m,dist:%d,via:%n}", s, Path.class);
            }
        };
        Matchbox.rewriteAll(set, rule);
    }
}
```

Impacts possibles

- Eco-système de contributeurs langage!
  - Tout programmeur peut inventer une notation
  - ...composable avec toute autre
- Large vitrine pour des techniques avancées
  - Pattern polymorphism
    - Function foo(pat) { ... = match(..., pat) }
  - Patterns calculés
  - Path polymorphism
  - Pattern structures?, signposts?, ...

Pattern calculus ~ Matchbox

PC	Mbox	Commentaires
[x,y]ĉxy	[%x|%y]	Notation associée au déconstruteur c – un pattern avec deux variables
[x,y]$\hat{c}_1(\hat{c}_2 x)y$	[[%x]|%y]	Deux notations imbriquées statiquement
[x,y]$c_1(\hat{c}_2 x)y$	c1 = [%_|%y] ... c1 o [%x]	Pattern dynamique composé avec pattern statique. (à implanter!)
[x,y]$\hat{c}_1(c_2 x)y$	c2 = [%x] ... [%_|%y] o c2 ou [$c2|%y]	Pattern statique composé avec pattern dynamique: composition ou interpolation de chaînes.
[y,z]ŷẑ	(% %)	Notation générique de déconstruction d'objet en un élément et un (ou des) sous-élément(s).

Conclusions

- Ergonomie des Notations proche des Regex:
  - Représentées comme des chaînes
  - Inteprétées ou compilées
  - Non-typées, non-vérifiées statiquement
  - Réserve les ([{ et }]), qu'on peut échapper
  - Performance similaire
- Avantages
  - Incluent les Regex comme cas particulier
  - Les notations compilées sont vérifiées
  - Les variables peuvent être nommées
  - Permettent des notations spécialisées / classe

Perspectives

- Limitations à enlever
  - Définition déclarative des langages de patterns
  - Test d'exhaustivité pour un match
  - Patterns typés ?
  - Formaliser les patterns non-déterministes
- Applications à explorer
  - Essai à grande échelle sur un système existant
  - Nouvelles techniques de programation objet?

Merci!

?

Ex*: Notation ensembliste

- Liste List<E> ou tableau E[ ] vue comme un ensemble
  - (∀n ≥ 0) **{%, . . . %}** : {d | length(d) = n}
  - (∀n ≥ 0) **{%, . . . %|%}** : {d | length(d) ≥ n}
- Ex de patterns de liste/ensemble:
  - [1,2,3] ~ {3,1,2}
  - [1,2,3] ~ {2|%}

Réécriture: Dijkstra/CDuce

```
let trans([Path+] -> [Path*])
  [<path to=<node id=n>_ dist=dn via=v>[]; tl] ->
     print_utf8(['Distance to '!n' = '!(string_of dn)' via '!v'\n']);
     sortPath(transform tl with
     p&<path to=m&<node..>links dist=dm ..>[] ->
       (match memberNode(n, links) with
          [<link dist=dnm..>[]] ->
           if dm <= dn + dnm
           then [p]
           else [<path to=m dist=(dn + dnm) via=n>[]]
        | _ -> [p]))
```