Principes de Programmation
Fiche de Révisions:
Réduction et équivalence de programmes

22 janvier 2019

1 Syntaxe d'un mini-Haskell

1.1 Notations

Égalité en mémoire On utilise le symbole $\equiv$ pour désigner une égalité en mémoire. Par exemple `[3,5,6]` $\equiv$ `3:[5,6]`.

Réduction On utilise `~>` ou $\rightsquigarrow$ pour parler de réduction, il s'agit d'une potentiel étape de calcul de Haskell. Attention, cela n'inclus pas les optimisations de Haskell et la version réelle peut être beaucoup plus efficace.

Équivalence L'équivalence $=$ ou $\simeq$ est une clôture réflexive, symétrique et transitive de la réduction. Cela permet de raisonner sur des comportement de programme.

Pretification des opérateurs [1] Les opérateurs Haskell sont en UTF8, ici on on va utiliser des versions stylisées plus jolies pour les équations. Ainsi `->` devient $\rightarrow$ et `<=` devient $\leq$.

1.2 Notions de bases

Types Dans ce documents, on parle jamais (ou très rarement) de type car ce sera fait dans un document séparé, mais il faut bien avoir en tête que tout ceci doit être typé.

Constructeurs Les constructeurs sont définit dans le poly sur les types comme les constructeurs de type de donnée algébriques. Un constructeur est un mot de lettres/chiffres commencement par une majuscule (ex : `VRAIS`, `Noeud`...). Les

1. Le mot pretification est un anglicisme désignant une façon de rentre plus jolie/élégant.

constructeurs ont une arrité [2] donnée par leur définition et leur type interdit de leur appliquer moins d'arguments que leur arrité. Un constructeur quelconque sera représenté par `C_1, C_2...`

Fonctions et variables globales Les fonctions globales et variables globales sont toutes représentés indistinctement par des mots de lettres/chiffres commencement par une minuscule. Les fonctions on toutes une arrité $n \geq 1$. En Haskell, une variable global est juste une fonction d'arrité 0, càd. sans argument, il n'y a donc pas de différence fondamental. On parlera donc indistinctement de fonction et variable globales d'arrité $n \geq 0$. La définition de variables globales sont définis plus tard.

Variables locales Les variables locales sont aussi notés par des mots de lettres/chiffres commençant par une minuscule. Mise à part le fait qu'elles n'ont pas d'arrité, il n'y a pas de différence visible entre variables local et global. Une fonction/variable (local ou global) quelconque sera représenté par `x,x_1,...` ou `f`.

Opérateurs Dans la suite, les opérateurs ne sont pas considérés car cela rajoute des cas inutiles. Un opérateur est une suite de symboles (ex : `+`, `<=`, `<<=`, `<>`,...) représentant une fonction qui attend deux arguments qui seront notés de manière infixe. Si on met un opérateur entre des parenthèses, il devient une fonction au sens habituel, on utilise donc toujours l'égalité :

$$s \triangleright t \quad \equiv \quad (\triangleright)\ s\ t$$

pour tout opérateur $\triangleright$ et tout termes s et t, de sorte à ne considérer que des fonctions dans la partie théorique.

Opérateur constructeur Un constructeur pour aussi être un symbole commençant pas ':' (ex : `(:)`, `:>`,...), dans second cas, on parle de d'opérateur constructeur, mais on ne l'utilisera pas mise à part pour le cons `(:)` de la liste. Un tel opérateur constructeur a forcément une arrité de 2. Les remarques sur les opérateurs s'appliquent.

1.3 Termes

définitions Un terme est donné par :

- — ou bien une variable,
- — ou bien un constructeur,
- — ou bien un entier,
- — ou bien une application $t_1\ t_2$ de deux termes t_1 et t_2,
- — ou bien une λ-expression `(\x->t)` pour un terme t, stylisé en $(\lambda x \to t)$.

Pour éviter les parenthèses, on associe à gauche, ainsi `Ou (Var 2) toto` veut dire `(Ou (Var 2)) toto`.

2. Une arrité est un entier donnant le nombre d'arguments qu'ils doivent avoir.

Forme normale Une forme normale est un terme particulier représenté uniquement par :

- un entier,
- un λ-expression $(\lambda x \to t)$ pour t un terme quelconque,
- de la forme $f\ t_1 \cdots t_n$ où f est une fonction global d'arrité $m > n$ et $t_1, ..., t_n$ sont des termes quelconques,
- ou bien de la forme $C\ t_1 \cdots t_n$ où C est un opérateur et $t_1 ... t_n$ sont quelconque.

Donnés pures Une donnée est une forme normale particulier représenté uniquement par :

- un entier,
- ou bien un constructeur d'arrité n suivit de n données.

Exemple : `Ou (Var 2) Vrais` est une donnée.
Remarque : Les données pures sont les seuls "vrais" observables.

Patterns Un pattern est un terme particulier représenté uniquement par :

- un entier,
- ou une variable,
- ou de la forme $C\ p_1 \cdots p_n$ où C est un constructeur d'arrité n et $p_1 ... p_n$ sont des patterns.

De plus une variable ne peux pas apparaître deux fois dans le même pattern.
Exemple : `Ou (Non x) y` est un pattern, de même que `x:y:l`, mais `(x:x:l)` n'en est pas un car il y a deux fois la même variable.

Définitions de fonctions et variables globales Une fonction globale (ou une variable globale) `maFonction` d'arrité n est définit par un programme de la forme

```
maFonction p_1_1 ... p_1_n = t_1
...
maFonction p_k_1 ... p_k_n = t_k
```

où `p_1_1,...,p_k_n` sont des patterns et `t_1...t_k` sont des termes ; de plus, on demande à ce que pour tout $i \leq k$, les variables apparaissant dans `p_i_1...p_i_n` soient différentes.

1.4 Substitution et reconnaissance de pattern

Substitution [3] Une substitution est une séquence, notée $\{x_1 := t_1, \ldots, x_n := t_n\}$ d'associations entre une variable et un terme.

3. Dans le cours, j'avais parlé d'instanciation, mais après réflection, ce nom colisinne avec le nom d'une autre notion que l'on vera plus tard.

Appliquer une substitution Lorsque l'on écrit $t\{x_1 := t_1, \ldots, x_n := t_n\}$ celà représente le terme t' obtenu depuis t en remplaceant tous les x_1 par t_1, ..., tous les x_n par t_n.
Exemples :

$$(x + x)\{x := 3\} \equiv 3 + 3 \qquad (\texttt{OU VRAI}\ x)\{y := 3, z := \texttt{FAUX}\} \equiv \texttt{OU VRAI}\ x$$
$$(x{:}l)\{x := 3, z := (\lambda x \to 2)\} \equiv 3{:}l \qquad (\lambda z \to \texttt{OU}\ x)\{x := \texttt{VRAI}\} \equiv \lambda z \to \texttt{OU VRAI}$$

Reconnaissance de pattern Un pattern p reconnaît par un terme t sous la substitution I si :

- p est une variable x et $I = \{x := t\}$,
- p et t sont des entiers et $p = t$,
- $p = C\ p_1 \cdots p_n$ et $t = C\ t_1 \cdots t_n$ avec p_1 reconnaît un terme t_1 sous la substitution I_1, ..., p_n est reconnaît un terme t_n sous la substitution I_n et avec [4] $I = I_1 \uplus \cdots \uplus I_n$.

On dit que p est reconnaît un terme t s'il le reconnaît sont une substitution.
Exercice : Vérifier que si un pattern reconnaît un terme t sous les substitutions I et J, alors $I = J$.

Rejet de pattern Un pattern p rejette un terme t si :

- p est un entier et t est une forme normale [5] différente de t,
- ou bien $p = C\ p_1 \cdots p_n$ et t est une forme normale qui n'est pas de la forme $C\ t_1 \cdots t_n$,
- ou bien $p = C\ p_1 \cdots p_n$ et $t = C\ t_1 \cdots t_n$, mais il y a $i \leq n$ tel que p_i rejette t_i.

Reconnaissance vs rejet
Exercice : Vérifier qu'un pattern ne peut pas à la fois reconnaître et rejeter un même terme.
Exercice : Trouver un pattern p et un terme t tel que p ne reconnaît pas t, mais ne ne rejette pas non plus.

2 Réduction en mini-Haskell

2.1 Définition

Réduction Une réduction sans contexte `~>` ou $\rightsquigarrow$ est une relation entre deux termes définit par les cas suivants.

4. Remarquez que les substitutions sont bien disjointes car il ne peut pas y avoir deux mêmes variables dans p.

5. Si t est bien typé, on peut prouver que si t est une forme normale, c'est alors forcément un entier.

β-réduction La β-réduction (ou beta-reduction) est définit comme la substitution

$$(\lambda x \to t)\ s \rightsquigarrow t\{x := s\}$$

où l'on remplace toutes les occurrences de `x` dans le terme `t` par `s`.

Réduction définitionnelle Si `maFonction` est définie par

$$\begin{aligned}
&\texttt{maFonction}\ p_{1,1} \cdots p_{1,n} = t_1 \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
&\texttt{maFonction}\ p_{k,1} \cdots p_{k,n} = t_k
\end{aligned}$$

alors on a la réduction

$$\texttt{maFonction}\ s_1 \cdots s_n \quad \rightsquigarrow \quad t_i\{x_1 := r_1, \ldots, x_m := r_m\}$$

pour $i \leq k$ tel que :
- pour tous $j \leq n$, $p_{i,j}$ reconnaît s_j sous une substitution I_j,
- $x_1 := r_1 ... x_m := r_m = I_1 \uplus \cdots \uplus I_n$,
- pour tout $i' < i$, il y a $j \leq n$ tel que
  - $p_{i',j}$ rejette t_j,
  - pour tout $j' < j$, $p_{i',j}$ accepte t_j.

η-réduction La η-réduction (ou eta-reduction) est définit comme

$$(\lambda x \to t\ x) \rightsquigarrow t$$

à condition que x n'apparaisse pas dans t.

Réduction sous contexte On écrit aussi que $t_1 \rightsquigarrow t_2$ s'il existe t et $s_1 \rightsquigarrow s2$ avec une des règles si dessus tel que $t_1 \equiv t\{x := s_1\}$ et $t_2 \equiv t\{x := s_2\}$.

Correction du résultat S'il existe une séquence $t \rightsquigarrow \cdots \rightsquigarrow d$ avec d une donnée pure, alors le résultat de l'évaluation de t sera bien d. Inversement, si t termine et rend une donnée pure d, alors il existe une séquence $t \rightsquigarrow \cdots \rightsquigarrow d$.

2.2 Utilisations

À venir...

2.3 L'exécution réel de Haskell (hors programme)

À venir...

3 Équivalence de programme

3.1 Définition

Pourquoi ? S'il est intéressant de comprendre comment se réduisent des programmes (études fines de complexité, optimisation, etc...), il est souvent plus intéressant de savoir dire lorsque deux programmes se comportent de la même manière. On façon de parler de ça est de parler d'équivalence de programme.

Équivalence Deux termes s et t sont équivalent si il existe une chaîne (potentiellement nulle) de la forme $s \leftrightsquigarrow \cdots \leftrightsquigarrow t$ où $s_1 \leftrightsquigarrow s_2$ est soit une réduction $s_1 \rightsquigarrow s_2$ ou une antiréduction de la forme $s_2 \rightsquigarrow s_1$.

Une verion plus simple En fait, cette définition est équivalente à la suivante : deux termes s et t sont équivalent si il existe une chaîne deux chaines de réductions de la forme $s \rightsquigarrow \cdots \rightsquigarrow r$ et $t \rightsquigarrow \cdots \rightsquigarrow r$ vers un même terme.

Correction du résultat Si deux programmes sont équivalents, alors ils sont observationnellement équivalents, c'est à dire que remplacer l'un par l'autre ne changera pas le programme à part sa vitesse d'éxécution.

3.2 Utilisation

À Venir