Université Mohamed Khider – Biskra
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'Informatique
2ème année Master
Module : Logique de Réécriture et ses Applications

Spécifications équationnelles

Nb: Une grande partie du matériel présenté dans les diapos est inspirée du cours INF 3230 de l'université d'Oslo.

Signatures

- Type de donnée (Data types) : Signature multi-sortés (multi-sorted signature)

- Type de Données
- Sorts (type)
- Function symbols (Symboles de fonction/opérateurs)
- Ground termes (Terme de base)

Types de Données

- Type de donnée (Data types) : Int, Listes, Booleans, Piles, Arbres binaires, ….
- Un type de donnée a un domaine / ‘éléments’ et un ensemble de fonctions / opérations sur ce type de donnée.

Exemples:

- Le type de donnée Int a les éléments « les entiers », et les opérations + , *, max, >=, …
- Le type de donnée Bool a les éléments « true »et « false », et les opérations not , and , or , etc.
- Le type de donnée List a les opérations ‘append’ , ‘ remove element’, etc.

Programmation déclarative

- **Java** :
- Int , Bool , char sont des types prédéfinies.
- Les autres types de données (List, arbres, ensembles, graph, ..) doivent êtres déclarés par des classes, en utilisant des pointeurs/références, etc.

- **Maude** :
- flexible; tous les types de données sont définies directement.
- Pour des raisons de simplicité et de commodité, les types Entiers et String sont prédéfinies dans Maude.

Programmation déclarative avec Maude

- Tout les 'valeurs'/'expressions' sont construites par des symboles de fonction !
- Fonctions sont définies récursivement par un ensemble d'équations.
- Variables sont des variables mathématiques; elles ne réfèrent pas à des locations dans la mémoire de l'ordinateur !

=On peut pas affecter/assigner une valeur à une variable.

- Haut Niveau:
- Pas de pointeurs
- Pas d'aliasing (deux variables ou plus réfèrent au même objet)
- Pas de contrôle du programme
- Maude fait le travail pour vous !

1er programme Maude

- Un module fonctionnel :

```
fmod NAT-ADD is
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .

vars M N : Nat . --- This is a comment

eq 0 + M = M .
eq s(M) + N = s(M + N) .
Endfm
```

Une spécification Maude est un ensemble de déclarations (pas une liste), ainsi l'ordre de déclaration n'est pas important !

1er programme Maude (2)

- Définit un type/sort Nat
- Fonctions constructeurs (0 et s) construisent les valeurs/éléments de Nat :
- 0 représente le nombre 0
- s(0) représente le nombre 1
- s(s(0))représente le nombre 2, etc.

- Autre type de fonctions tel que + sont des fonctions ordinaires qui ce sont définies par des équations.
- _+_ notation ‘mix-fix’.
- Calculer/ simplifier une expression par l’application des équations de gauche à droite :

$$s(0) + s(0) \rightsquigarrow s(0 + s(0)) \rightsquigarrow s(s(0))$$

Exécution du Maude (séance TP)

1. Ecrire le module Maude / programme dans un fichier en utilisant votre éditeur de texte favori (nat-add.maude)

2. commencer l'exécution par la commande maude

3. Lire le fichier dans Maude

```
Maude> in nat-add.maude
```

1. Exécuter Maude :

```
Maude> red s(s(0)) + s(0) .
```

Résulta:

```
rewrites: 3 in 0ms cpu (0ms real)
result Nat: s(s(s(0)))
```

1. Exécuter une autre commande, une autre …etc.

2. Finir avec q (or quit)

Inclusion

Inclusion des modules précédemment entrée : including ou protecting :

```
fmod NAT-MULT is
protecting NAT-ADD .
op _*_ : Nat Nat -> Nat .

vars M N : Nat .

eq 0 * M = 0 .
eq s(M) * N = N + (M * N) .
Endfm
```

Maude ne fait pas de différence entre including et protecting .

Sorts / Types

Une Sorte correspond a un ‘type’. Elle est déclarée :

```
sort Nat .
```

Déclaration de plusieurs sorte :

```
sorts Nat Boolean List .
```

Symboles de Fonction

- Fonctions (symboles) f, g, h sont déclarées comme suit:

```
op f : s1 s2 s3 s4 -> s .
ops g h : s1 s2 -> s .
```

Où `s1, s2,..., s` sont des sortes.

- La chaîne s1 s2 est appelée l'arité (arity) de g, et s est appelée la sorte (valeur) de g.
- Une fonction avec une arité vide est appelé un constant.

```
op 0 : -> Nat .
```

- Notation Préfixe _ :

```
s(s(0)), g(a,b)
```

- Notation 'mix-fix' :

```
op _+_ : Nat Nat -> Nat .
```

On peut écrire ainsi s(0) + 0

Définitions d'une signature

Une définition est peut apparaître difficile, mais généralement elle décrit ce que vous attendez.

Définition (Signature)

Une Signature Multi-sortées (S, $\sum$) consiste en un ensemble S de sortes, et S* x S-famille de sortie $\sum = \{\sum_{w,s} \mid w \in S^*, s \in S\}$ de symboles de fonction. ($\sum_{w,s}$ est l'ensemble de symboles de fonctions avec l'arité w et la sorte valeur s.) on écrit souvent $f : w \rightarrow s \in \sum \; pour \; f \in \sum_{w,s}$. Si w est une liste vide, ainsi f est souvent appelée un constant (de sorte /type s)

Ground Terms (Termes de base)

• A ground term (terme de base) est construit à partir de symboles de fonction dans une forme correcte : 0, s(0), s(s(0)), s(0) + 0 et s(0) + (0 + 0) sont des termes de bases de la sorte Nat dans NAT-ADD

Définition (Signature)

Soit une Signature Multi-sortées $(S, \sum)$, on peut définir l'ensemble S-typés $\tau_{\sum} = \{\tau_{\sum,s} | s \in S\}$ de termes de base (ground terms) inductivement par les conditions suivantes:

1. $\sum_{\epsilon,s} \subseteq \mathcal{T}_{\sum,s}$;
2. si $f \in \sum_{s_1 \ldots s_n,s}$, et $t_1 \in \mathcal{T}_{\sum,s_1}, \ldots, t_n \in \mathcal{T}_{\sum,s_n}$, et $n \geq 1$, $f(t_1, \ldots, t_n) \in \mathcal{T}_{\sum,s}$.
3. en outre, chaque fonction $\mathcal{T}_{\sum,s}$ est la plus petit ensemble vérfiant les conditions précédentes

Définitions

- Soit la signature :

```
sorts s s' .                    ops a b : -> s .
op f : s -> s' .              op g : s s' -> s .
```

- soit les termes suivants:

```
a                                b                                (a)
f(a,b)                  f(f(a))                  g(b,f(a))
g(a,b)                  f()                      XV(a)
```

lesquels entre ces termes sont des :

- Termes de base ?
- Termes de base de sorte s ?
- Termes de base de sorte s' ?

Les éléments d'un type

- Comment définir les éléments d'un type ?
- List en Java

```
class ListNode {
int element;
ListNode next;
}
```

- Maude : pas de pointeurs, uniquement des 'valeurs'/ termes

- Symboles de fonctions :

- Ground terme (terme de base) construit par une fonction constructeur (ctor) définie le domaine/valeur ensemble/élément

- Autre fonction sont 'ordinaire'; des fonctions définies

Exemple : Valeurs Booléennes

- Quels sont les éléments d'un type Boolean ?

```
sort Boolean .
ops true false : -> Boolean [ctor] .
```

Quelques symboles de fonction à définir ?

```
op not_ : Boolean -> Boolean .
ops _and_ _or_ : Boolean Boolean -> Boolean .
```

Quelques ground terms (terme de base) de type boolean ?

```
True   false          not true       true or false
(true or false) and true
(true and true) and (false or not false)
```

Exemple : Liste des nombres naturels (version 2)

```
sort List .
```

Pour construire des listes :

- Besoin d'une liste vide

```
op nill : -> Liste [ctor] .
```

- Besoin d'ajouter les elements à la liste : "app"

```
op app : List Nat -> Liste [ctor] .
```

La liste " 1 2 3" est présentée par les termes de bases (ground term)

```
app(app(app(nil, s(0)), s(s(0))), s(s(s(0))))
```

Exemple : Liste de nombres naturels (version 2 et 3)

Opérateur d'ajout plus élégant :

```
op _++_ : List Nat -> Liste [ctor] .
```

La liste " 1 2 3" est présentée par les termes de bases (ground term)

```
nil ++ s(0) ++ s(s(0)) ++ s(s(s(0)))
```

Encore plus élégant ?

```
op _ _ : List Nat -> Liste [ctor] .
```

La liste " 1 2 3" est présentée par les termes de bases (ground term)

```
nil s(0) s(s(0)) s(s(s(0)))
```

Exemple : autres fonctions sur les Listes des nombres naturels

Exemple de fonction non constructeur sur les listes :

```
op op length : List -> Nat .
op concat : List List -> List .
op insertFront : Nat List -> List .
ops first last : List -> Nat .
op rest : List -> List .
op empty? : List -> Boolean .
op reverse : List -> List .
```

Exemple : signature des arbres binaires

- **Constructeurs :**

```
sort BinTree .
op niltree : -> BinTree [ctor] .
op btree : BinTree Nat BinTree -> BinTree [ctor] .
```

- **Plus beau avec :**

```
op _^_^_ : BinTree Nat BinTree -> BinTree [ctor] .
```

- Quelques fonction à définir:

```
ops preorder inorder postorder : BinTree -> List .
ops size weight : BinTree -> Nat .
op isSearchTree : BinTree -> Boolean .
op reverse : BinTree -> BinTree .
```

Exemple : Signature des ensembles

```
sort Set .
op empty : -> Set [ctor] .
op _;_ : Set Nat -> Set [ctor] .
```

- L'ensemble {2,4,5} est présenté par :

```
empty ; s(s(0)) ; s(s(s(s(0)))) ; s(s(s(s(s(0)))))
```

NB: l'ensemble {2,4,5} égale à l'ensemble {4,2,5} qui est représenté par le terme

```
empty ; s(s(s(s(0)))) ; s(s(0)) ; s(s(s(s(s(0)))))
```

Par la suite on verra comment représenté cette propriété

Définition des fonctions

- il faut définir des fonctions ordinaires tel que ‘+’
- Les constructeurs ne doivent pas êtres définis par des équations, leurs rôle est de construire les éléments d’un type.
- **Example**: on a les termes de base et on définit la fonction ‘+’ comme suit:

```
eq 0 + 0 = 0 .
eq s(0) + 0 = s(0) .
eq 0 + s(0) = s(0) .
eq s(0) + s(0) = s(s(0)) .
eq s(s(0)) + 0 = s(s(0)) .
...
```

Variables

- On a besoin de définir des variables mathématiques
- Chaque variable est d'un type ;

```
vars M N : Nat .
var X : Boolean .
```

- Maintenant; nous pouvons définir la fonction '+' plus efficacement comme suit:

```
eq 0 + M = M .
eq s(M) + N = s(M + N) .
```

Ce Que signifier que les égalités représentent toutes les possibilités de M et N.

Termes

- Un terme est construit à partir des variables et des symboles de fonctions.

  (termes de base/groud term : ce sont des termes sans variables)

- Exemple:

```
sorts s s' .
ops a b : -> s .
op f : s -> s' .
op g : s s' -> s .
var X : s . var Y : s' .
```

- Quels sont les termes parmi les éléments suivants ?

```
a              f(a)          f(X)          f(Y)

g(X,Y) g(a,f(X))     g(a,X)        g(a,Y)
```

Equations

- Une équation est un triplet, écrit , ou et sont des termes du même sorte/type.
- Une équation est écrite en Maude sous la forme

  eq t = t' .
- équations conditionnelles a la forme :

  ceq t = t' if u = v .

ou

ceq = if u1 = v1 /\ … /\ un = vn .

- **Exemple**

```
vars M N : Nat .
eq 0 + M = M .
eq s(M) + N = s(M + N) .
ceq min(M, N) = N if (N <= M) = true .
ceq min(M, N) = M if (N <= M) = false .
```

Spécification équationnelle

Definition (Specification equationnelle)

Une spécification équationnelle Multi-sortée est un triple (S, Σ, E) dont

- S **est un ensemble de sortes**
- Σ **est une signature**
- E **un ensemble d'equations (conditionnelles et / ou incondtionnelles)**

Sémantique opérationnelle

- Sémantique : c'est le sens d'un texte/ spécification.
- Mathématiques : une spécification définie une algèbre.
- Opérationnelle : comment la spécification sera exécutée

- **Sémantiques Opérationnelles** :utiliser les équations « de gauche à droite » pour simplifier les expressions (terme) autant que possible.

- Quelle équation est appliquée ? Et quand est ce que son application n'est pas déterminée?
- Simplification est appliquée autant que possible
- (pas de contrôle du programme, juste un ensemble d'équations)

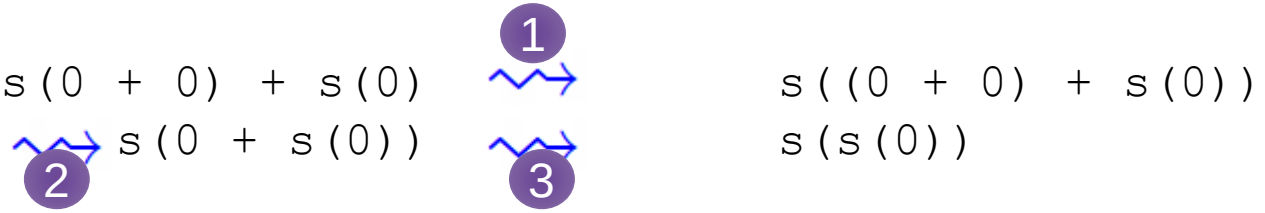
Computation

```
eq 0 + M = M .
eq s(M) + N = s(M + N) .
```

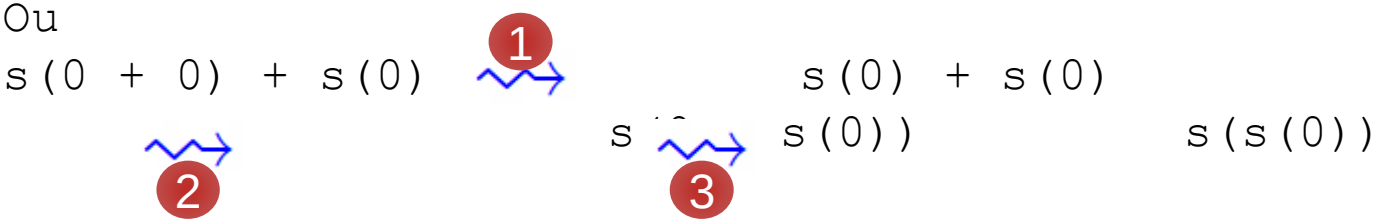
- Utilisez la commande Red pour calculer s(0 + 0) + s(0) :

```
Maude> red s(0 + 0) + s(0) .
```

est simplifiée (en utilise le symbole ⇝ pour un seul pas de simplification) :



Ou



Différents choix, même réponse : `result Nat: s(s(0))`

Besoins : Propriétés Souhaitées

- Propriétés souhaitées de (calcul de) spécifications :
- Terminaison : pas de boucle infinies.
- Résultat unique : les différents calcules possible doivent toujours donner la même réponse.
- Le résultat doit être un terme constructeur (constructor term)

- On doit s'assurer que notre spécification respecte ces propriétés !

- Est-ce que ces propriétés sont vérifiées pour NAT-ADD ?
- C'est bon de commencer par le terme s(0 + 0) + s(0)
- Mais existe il un terme a partir duquel la module ne termine pas son exécution ? Donne t-il un résultat unique ? Et/ou est ce que le résultats est un terme constructeur ou non ?!

Terminaison

Definition (Termination)

traduction: une spécification se termine si elle ne contient pas un calcul infini

$$t_0 \rightsquigarrow t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots$$

- Chaque terme t_0 doit terminer
- NAT-ADD se termine il faut le montrer sur tout les termes t_0 !!
- eq M = 0 + M .
- Ne termine pas

```
s(0)        0 + s(0)        0 + (0 + s(0))    ...
```

- Concernant, la co $\rightsquigarrow$ tativité : $\rightsquigarrow$

```
vars M N : Nat .      eq M + N = N + M
```

Terminaison (2)

- La terminaison est indécidable

- Il existe des techniques qui nous permettre souvent de vérifier la terminaison

- Petite exemple : la solution qu'on a donné réduit les appelles récursifs | devenir plus petit :

```
eq s(M) + N = s(M + N) .
```

-

Résultat unique

- Calculer la valeur d'une expression fonctionnelle

- Devrait obtenir le même résultat quel que soit l'ordre que Maude à choisit dans l'application des équations!

- Exemple :
- Toute les possibilités de calcul du terme S(0 + 0) + s(0) dans NAT-ADD donne le même résultat.
- Toujours un résultat unique dans NAT-ADD.
- La spécification suivante ne vérifiée pas l'unicité du résultat !

```
eq a = b . eq a = c .
```

- Valeur 'unique' est appelée forme normale

Résultat est un terme constructeur

```
Teacher: What is 5+8?
Student: 5+8.
```

- La réponse est correcte , mais elle n'est pas le résultat désiré
- 'résultat/valeur' d'un terme de base doit être un terme constructeur.
- Exemple :
- La valeur de s(0 + 0) + s(0) dans NAT-ADD est un terme constructeur s(s(0))

- Si on oublie l'équation

```
eq 0 + M = M .
```

- Donc la valeur de `s(0 + 0) + s(0)` est `s((0 + 0) + s(0))`

Définitions

- pour vérifier que les résultats doivent êtres des valeurs constructeur, chaque fonction non-constructrice doit être définie pour chaque terme de base constructeur.
- '+' doit être définie pour toute paire (t, t') de termes de bases constructeurs de sorte Nat.

```
eq 0 + M = M .
eq s(M) + N = s(M + N) .
```

Pour chaque paire de termes constructeur (t, t'), une équation pour s'appliquer pour t+t'

- Souvent une fonction est définie avec des équation pour chaque constructeur :

```
op f : ... Nat ... -> ...
eq f(..., 0, ...) = ... .
eq f(..., s(N), ...) = ... .
```

Définitions (2)

- Multiplication est définie pour tous les nombres naturels ?

```
op _*_ : Nat Nat -> Nat .
eq 0 * M = 0 .
eq s(M) * N = (M * N) + N .
```

- double est définie pour tous les nombres naturels avec une seule équation :

```
op double : Nat -> Nat .
eq double(M) = M + M .
```

- Est ce que la fonction minusTwo est définie pour tous les nombres naturels ?

```
op minusTwo : Nat -> Nat .
eq minusTwo(0) = 0 .
eq minusTwo(s(0)) = 0 .
eq minusTwo(s(s(N))) = N .
```

Définitions (3)

- Un autre exemple :

- ‘Cas’ des deux arguments :

```
op _<_ : Nat Nat -> Boolean .
eq N < 0 = false .
eq 0 < s(N) = true .
eq s(M) < s(N) = M < N .
```

- Des appels récursifs plus petit conduit à une terminaison

Fonctions Auxiliaires

- Parois on a besoin d'introduire des fonctions auxiliaires pour définir une fonction
- Exercice : étendre NAT-ADD avec la fonction

```
op _**_ : Nat Nat -> Nat .
```

Où **m ** n** doit être égale à **.**

- Solution : Impossible d'atteindre ça en utilisant uniquement des 0, s et + ! On doit introduire une fonction auxiliaire * :

```
fmod NAT-POWER is protecting NAT-ADD .
ops _*_ _**_ : Nat Nat -> Nat .
vars M N : Nat .
eq 0 * M = 0 .
eq s(M) * N = N + (M * N) .
eq M ** 0 = s(0) .
eq M ** s(N) = M * (M ** N) .
endfm
```

Exemple: Valeurs de vérité

```
fmod BOOLEAN is
sort Boolean .
ops true false : -> Boolean [ctor] .
op not_ : Boolean -> Boolean [prec 53] .
op _and_ : Boolean Boolean -> Boolean [prec 55] .
op _or_ : Boolean Boolean -> Boolean [prec 59] .
op _implies_ : Boolean Boolean -> Boolean [prec 61] .
var X : Boolean .
eq true and X = X .
eq false and X = false .
...
endfm
```

La fonction ‘and’ est complètement définie **?**

Exemple: Valeurs de vérité (2)

- L’attribut prec donne la précédence pour éviter l’écriture des parenthèses : ‘not true and false’ sera lit comme ‘(not true) and false’

- Remarque: Maude intègre aussi un module prédéfinie BOOL. Donc pour travailler avec notre spécification booléenne on doit d’abord exécuter la commande :

```
set include BOOL off .
```

Exemple: Listes

- Signature de listes :

```
fmod LIST-NAT1-SIGN is protecting NAT1 .
protecting BOOLEAN .
sort List .
op nil : -> List [ctor] .
op _ _ : List Nat -> List [ctor] .
op length : List -> Nat .
op concat : List List -> List .
op insertFront : Nat List -> List .
ops first last : List -> Nat .
```

- Remarquer la syntaxe de l'opérateur append '_ _'

- Exemple : liste '1 3' est représentée par le terme
  - `Nil s(0) s(s(s(0)))`

La fonction length sur les liste

- Signature de listes :

```
fmod LIST-NAT1 is including LIST-NAT1-SIGN .
vars N N' : Nat .
vars L L' : List .
eq length(nil) = 0 .
eq length(L N) = s(length(L)) .
```

- Définition ‘logiquement correcte’ ?

- Longueur définie pour toutes les listes ?

Exemple : Concaténation des listes

- concaténer deux listes :

```
eq concat(L, nil) = L .
eq concat(L, L', N) = concat(L, L', N) .
```

- correcte ?

- Définie pour toutes les paires de listes ?

Exemple : Premier élément dans une liste

- Premier éléments dans une lite :

```
eq first(nil) = 0 .
eq first(nil N) = N .
eq first(L N N') = first(L N) .
```

- Première équation ?
- Autrement correcte ?
- Définie pour toutes les listes ?

Exemple : Arbres binaires

- Les constructeur :

```
sort BinTree .
op niltree : -> BinTree [ctor] .
op btree : BinTree Nat BinTree -> BinTree [ctor] .
```

- Nombre d'éléments dans un arbre (récursivement sur les constructeurs):

```
op size : BinTree -> Nat .
vars BT BT' : BinTree .
vars N N' : Nat .
eq size(niltree) = 0 .
eq size(btree(BT, N, BT')) = s(size(BT) + size(BT')) .
```

- Chaque appel récursif est plus 'petit' : terminaison

Exemple : Arbres binaires (2)

- Faire retourner l'arbre comme une liste dans l'ordre du parcours est Pré-ordre :

```
op preorder : BinTree -> List .

eq preorder(niltree) = nil .

eq preorder(btree(BT, N, BT')) =
       insertFront(N,concat(preorder(BT), preorder(BT'))) .
```